Desarrollo de aplicaciones para Android

2015

Joan Ribas Lequerica

A mis hermanas por su fuerza y dedicación
a la familia en tiempos difíciles,
orgulloso de ser vuestro hermano.

Dedicado especialmente a mi hija Mar
una ilusión hecha realidad.

Agradecimientos

A mi mujer Laia, mi compañera y amiga inseparable.

A los formadores y profesores que hacen que seamos capaces de pensar por nosotros mismos.

A mi familia por ser como son y estar siempre que se les necesita.

A los que creen que el mayor tesoro es el conocimiento.

A todos aquellos que hacen que la crisis sea menos crisis para otros.

A ti que lees este libro en lugar de ver la televisión.

Sobre el autor

Joan Ribas Lequerica cursó Ingeniería Electrónica e Ingeniería de Telecomunicaciones en la Universidad de Valladolid así como Ingeniería de Organización Industrial en Barcelona. Es autor de numerosas publicaciones en revistas tecnológicas y libros, entre los que se encuentran "Programación en Delphi", "Guía práctica Web Services", "Arduino Práctico" y las versiones anteriores de esta obra y antiguo colaborador de proyectos GNU como Gentoo o Enlightenment. En los últimos años el autor ha trabajado en varios proyectos de desarrollo de aplicaciones móviles para empresa, lo que le permite aportar su experiencia a este libro.

Índice de contenidos

1 Cómo usar este libro

Destinatarios de este libro

Este libro está dirigido a todas las personas que les interese la programación para dispositivos con sistema operativo Android.
Es muy recomendable que el usuario tenga conocimientos básicos de programación orientada a objetos y principalmente Java ya que es el lenguaje en el que se presentan los ejemplos.
Este libro también está indicado para aquellos que conociendo la manera de programar Android en Eclipse, quieren avanzar hacia la programación en el nuevo entorno de desarrollo Android Studio. El lector tendrá la posibilidad de conocer el entorno de programación, realizar sus propios programas y si posee su propio dispositivo físico, probar los programas en él
Teniendo en cuenta el ecosistema de dispositivos, durante el libro se abordará el tema de la programación mediante *Fragments* y así ajustar mejor las aplicaciones a todo tipo de pantallas pudiendo de este modo aprovechar la misma aplicación tanto para tabletas como para teléfonos con pantallas reducidas, incluso para televisiones, incluyendo un apartado de diseño mediante estilos y animaciones. Además, también se desvelan los secretos de las herramientas puestas a disposición por Google para los desarrolladores.

Organización del libro

El manual que tiene entre sus manos se encuentra organizado en capítulos con ejemplos prácticos más varios apéndices. En cada uno de ellos se explican dis-tintas funcionalidades presentes en el sistema operativo Android, y cómo uti-lizarlas mediante programación. Se trata de un libro eminentemente práctico, es decir que los capítulos vienen acompañados de distintos ejemplos que le serán ser de utilidad al lector para afianzar conocimientos y descubrir cómo programar distintos aspectos de Android; es por esto que animo al lector a rea-lizar los ejemplos a la vez que vaya leyendo los capítulos.

Convenios empleados

A lo largo del texto del libro, se utilizarán ciertos convenios de tipos de letra y formatos para facilitar la lectura.
Debido a que parte de las herramientas utilizadas solamente están disponibles en inglés, en el texto se escribirán los nombres de botones, menús, etc. en inglés, y se proporcionará traducción en todo caso.

Las combinaciones de teclas que en la pantalla aparecen relacionadas con el signo más, como por ejemplo Ctrl+U, en este libro aparecen relacionadas con un guión e impresas en negrita, por ejemplo, **Control-U**.
Los nombres de botones, herramientas y combinaciones de teclas aparecen en negrita para facilitar su identificación; por ejemplo, el botón **Finish**.
Los nombres de cuadros de diálogo, menús, submenús y fichas aparecen en otro tipo de letra para facilitar su identificación, como por ejemplo, el menú Project.
Los accesos a los menús de la aplicación aparecen separados por el signo mayor que (>) y en el orden de la selección. Por ejemplo, File>Preferences.
Elementos como funciones, palabras clave del lenguaje, comandos de programación y en general todo lo relativo a código que se encuentre incrustado en texto explicativo, aparecerá destacado con un tipo de letra `Courier`, excepto los nombres de clases Java los cuales aparecerán identificados con letra *cursiva.*
En cuanto a los nombres de las variables y las clases, se utiliza el convenio Java, es decir *CamelCase*. Los nombres de los archivos de configración de los proyectos Android, se pueden encontrar en *CamelCase* o en *snake_case*, aunque serán en *snake_case* la mayoría, ya que ciertos tipos de archivo no soportan el *CamelCase*. En cuanto a los nombres de variables dentro de los archivos de configuración de los proyectos Android (por ejemplo definición de constantes) Google no ha dictado ningún convenio, por lo que puede encontrarlos tanto en *CamelCase* como en *snake_case*, así que en este libro, para que no le resulte extraño, se han utilizado ambos convenios dependiendo del proyecto.
En el libro aparecen resaltados una serie de temas o acontecimientos extraordinarios de la siguiente forma:

Nota:

Comentarios o noticias fuera de texto.

Advertencia:

Información importante a tener en cuenta para la integridad del trabajo o del sistema.

Truco:

Consejo o información que puede facilitar un trabajo.

2 Introducción a Android

En este capítulo aprenderá a:

- Descargar las herramientas necesarias.
- Instalar el entorno de desarrollo.
- Crear proyectos Android.
- Crear una maquina virtual Android.
- Ejecutar un proyecto en el emulador.

¿Qué es Android?

Supongo que a estas alturas, si ha comprado un libro sobre programación en Android, tendrá ya una ligera idea de a qué nos estamos refiriendo cuando hablamos de Android. Más allá de explicar que es un sistema operativo abierto creado por Google, vamos a profundizar sobre alguna de sus singularidades, y analizar a través de hechos, cómo se diferencia de otros muchos sistemas operativos para dispositivos móviles que hay disponibles hoy en día:

- **Android es una plataforma de desarrollo libre, y de código abierto:** El núcleo del sistema está basado en un Linux (versión 2.6 para versiones anteriores a Android 4.0 Ice Cream Sandwich y versión 3.0 del kernel para posteriores; actualmente en Android Kit Kat se utiliza un kernel con versión 3.4.10) al que se le han hecho ciertas modificaciones para que pueda ejecutarse en teléfonos y terminales móviles. Las modificaciones se han realizado para adaptarlo a los menores recursos de los dispositivos móviles, que aunque cada vez son más potentes, no dejan de tener menos recursos que un ordenador de sobremesa. El hecho de ser gratuito y de código abierto, hace que los fabricantes de móviles puedan utilizarlo en sus nuevos terminales sin pagar licencias de uso, lo que abarata el precio final de venta al público y además pueden, sin ningún impedimento legal, ajustar aspectos que no cuadren con las expectativas de los clientes (tales como la interfaz gráfica en el caso de HTC con su *Sense UI*) ya que cualquiera puede acceder a todo el código, modificarlo y distribuirlo nuevamente.
- **Gran cantidad de servicios disponibles:** Según vayamos programando en Android nos iremos dando cuenta de la cantidad de servicios que posee y lo fácil que puede ser utilizarlos, por ejemplo servicios de GPS incluidos mapas, lectores de códigos de barras o incluso una base de datos que servirá para mantener la información de nuestras aplicaciones convenientemente ordenada... y todo esto sin tener que instalar librerías externas o hacer configuraciones complejas como en otros entornos... simplemente está ahí para cuando se necesite. Todo sin olvidar que el principal impulsor de Android es Google con todo lo que a servicios se refiere, ya que cada vez que Google saca al mercado un nuevo servicio, al poco tiempo ofrece una API para poder utilizarlo desde nuestros programas. Los terminales poseen también gran variedad de sensores que permiten tener conocimiento del entorno que les rodea y así poder acceder a la información para conocer posición exacta del dispositivo, temperatura del mismo…

- **Aplicaciones hechas de componentes:** Como ya se verá más adelante, cada aplicación es realmente como un puzle que está hecho de varias piezas que luego se orquestarán para que cumpla las especificaciones del programa. Lo mejor de todo esto es que piezas realizadas para un programa pueden ser utilizadas por otro o sustituidas; incluso las piezas estándares de sistema. Esto permite que si no nos gusta la pantalla de selección de contactos o la manera que se visualizan los sms, podemos hacernos nosotros una y sustituir la estándar... y cuando nos hayamos cansado volver a dejar la estándar de nuevo...o también usarla en nuestro programa con muy poco esfuerzo. ¿No es fascinante poder aprovechar de esta manera el código?
- **Multitud de información:** Con cada nueva versión del sistema operativo, Google pone a disposición de los programadores la información necesaria para el uso de las nuevas funcionalidades, incluyendo una serie de ejemplos documentados. Si esto no nos es suficiente, Android ha tenido una gran penetración en el mercado y en Internet existen cantidades ingentes de información y foros donde poder preguntar y obtener respuestas a nuestras dudas.
- **Multimedia:** A lo largo de las sucesivas versiones de Android, se ha podido ver como su capacidad visual ha ido mejorando no solamente añadiendo cada vez mejor calidad de gráficos y sonido sino también añadiendo mayor soporte a formatos de video y audio y mejorando la fluidez de reproducción incluso en dispositivos poco potentes. Por ejemplo podemos encontrar que los gráficos 2D poseen filtros de antialias para una mejor percepción, gráficos 3D mediante OpenGL acelerado, RenderScript... un campo abonado para la obtención de grandes programas multimedia, juegos espectaculares. Toda una delicia para los sentidos. Actualmente podemos encontrar en el mercado varias consolas basadas en Android de fabricantes tan importantes como NVIDIA.
- **Seguridad:** En el mundo móvil es muy importante la seguridad, cada vez guardamos más datos sensibles en nuestro dispositivo, es nuestra agenda, nuestra secretaria, un resumen de nuestra vida y nuestros actos (historial de búsquedas, mails, sms…) incluso nuestro monedero. Si se piensa en que estos dispositivos permiten desde las aplicaciones realizar llamadas o usar nuestras cuentas personales de otros servicios, se ha de proveer al programador de unas herramientas no sólo para proteger sus datos sino para evitar que se pueda desconfiar de su aplicación. Es por esto que Android dispone de una serie de mecanismos por los cuales al desarrollar un programa se dicta a qué servicios o qué elementos del

teléfono utilizará, de manera que al instalar la aplicación, el usuario puede ver a cuál de las funcionalidades de su terminal le está dando acceso a esa aplicación. Además, mientras las aplicaciones se encuentran en ejecución, varias capas de seguridad certifican el aislamiento de los datos entre ellas, de modo que una aplicación no tiene acceso a los datos de la otra y en caso de que se quede colgada solamente afectará a esa aplicación.

- **Gestión del ciclo de vida automático:** Todo está pensado para dispositivos con poca capacidad de proceso, poca memoria y poca batería. En Android existen unos ciclos de vida para las aplicaciones, y la gestión de este ciclo de vida es llevada a cabo desde el mismo sistema operativo, esto hace que no nos tengamos que preocupar en cerrar algunas aplicaciones cuando queremos lanzar un nuevo programa, ya se encarga el sistema operativo por nosotros, liberando espacio o durmiendo las aplicaciones que no se estén usando en ese momento. En la versión 4.4 Kit Kat se añade un nuevo modelo de consulta de sensores (denominado consulta por lotes); esta nueva manera de consultar los sensores, reduce el uso de batería y recursos, siendo además totalmente transparente a las aplicaciones desarrolladas (incluso las antiguas) y no afectando tampoco al rendimiento de las mismas.
- **Múltiple hardware:** Android desde el principio se ha pensado para múltiples plataformas. En las primeras versiones, cuando aún los dispositivos de pantalla táctil eran muy caros, se explicaba que Android podía funcionar en cualquier tipo de teléfono, desde los de pantalla táctil hasta teléfonos sencillos, sin pantalla táctil, sin ruedecitas... el típico teléfono móvil de toda la vida. A la larga, los dispositivos táctiles se han ido abaratando y mostrando su superioridad en cuanto a experiencia de usuario, por lo que el mercado se ha decantado por ellos, y estos crecen en formas y tamaños; ahora no son de 3 pulgadas, son de 3, 4, 6, 7, 10,21… y Android se ha adaptado. Desde la salida de Android 3.0 se han ido dando paso para aunar la interfaz de los dispositivos de pantalla grande y los dispositivos más pequeños hasta que en la versión 4.2 la interfaz entre ellos es casi la misma salvo pequeñísimos detalles, ayudando al usuario a la transición entre los dispositivos y al programador en no tener que diferenciar mucho entre ellos. El rango de dispositivos sobre los que se ejecutan las aplicaciones Android va desde los conocidos teléfonos hasta microondas o lavavajillas, pasando por televisores de hasta 46 pulgadas, pudiendo reaprovechar las interfaces entre ellos.

- **Nuevas utilidades:** Siguiendo con el punto anterior, Android ha saltado de los teléfonos móviles y en una carrera de innovación, estamos viendo que se va a aplicando a cualquier elemento que pueda tener electrónica... y digo cualquier elemento que pueda tener electrónica, porque incluso si no la tiene, se le puede añadir... así por ejemplo podemos ver relojes, pulseras para controlarnos mientras hacemos ejercicio, detectores de humo, gafas, lentillas (si, está leyendo bien)... incluso los coches... actualmente se ha creado la Open Automotive Alliance (alianza abierta de automóviles) para incluir Android de manera estándar en los coches; algunas marcas que se han adherido a esta alianza son Honda, Hyundai, GM y Audi. Todo aquello en lo que pueda meterse un poco de silicio, tiene su controlador Android... y si no lo tiene en el momento que escribo estas líneas, puede que lo tenga cuando usted las lea.
- **Se programa en Java:** Existen muchos programadores Java, dependiendo de la fuente que se mire (SourceForge, Tiobe, SkillMarket...), se discute el primer puesto con C, siendo siempre alrededor de un 17% del mercado. Eso quiere decir que la legión de programadores de Java fácilmente podrán desarrollar aplicaciones para Android y un sistema con muchas aplicaciones disponibles en su *market* es un buen aliciente para que un usuario lo compre (actualmente el *market* oficial de Android posee más de un millón cien mil aplicaciones).
- **Se puede programar en C o C++:** Aunque esto sea desconocido para la gran mayoría, Android puede ser programado mediante C o C++; para ello Google pone a disposición de los programadores una serie de herramientas llamadas NDK (*Native Development Kit*) con librerías y cabeceras de ayuda. El uso del NDK no implica la necesidad de realizar toda la aplicación en C o C++, es posible hacen mediante el NDK una parte de la aplicación y el resto mediante Java ya que existen mecanismos para comunicarse entre ambas partes.
- **OHA:** Todos conocemos que detrás de Android está Google... pero no es el único. El sistema Android es el sistema operativo de la OHA (*Open Handset Alliance*), una alianza de varias empresas para crear un ecosistema abierto para los dispositivos móviles. ¿Y quién forma esta alianza? Pues grandes empresas como Telefónica, Vodafone, T-mobile, Samsung, Acer, Asus, Toshiba, Intel, Nvidia, Ebay, Google, y otros muchísimos más (para conocer más puede dirigirse a la web oficial de la OHA `http://www.openhandsetalliance.com/oha_members.html`)... a medida que Android ha cogido fuerza, esta alianza la ha perdido aunque hemos visto surgir nuevas como la Open Automotive Alliance (OAA) que hemos comentado anteriormente.

Herramientas necesarias

Para programar aplicaciones orientadas al sistema operativo Android, en el punto anterior se ha visto que se puede realizar mediante Java o mediante C/C++. Google desde un principio ha impulsado más el uso de Java frente a C/C++ posiblemente por ser más sencillo de programar o quizá por ser más numerosa la comunidad de programadores Java.
Sea como fuere, en este libro se explicará cómo utilizar las herramientas diseñadas para la programación mediante Java. Si el lector está preocupado por la lentitud de ejecución de su aplicación si se realiza mediante Java en lugar de en C, comentar que desde Google se han realizado varios artículos desmintiendo que una aplicación realizada en C sea mucho más rápida que la realizada mediante Java (en entornos Android) ya que como se verá en el próximo capítulo, la maquina virtual Dalvik se encarga de acelerar la ejecución.
Como herramientas para trabajar se necesitarán: un editor de texto, o mejor aún un entorno de desarrollo Java, un emulador de Android y las herramientas que permitan empaquetar el código Java en algo entendible para el emulador y/o para el terminal físico.

Preparación del entorno

Antes de empezar a programar para dispositivos Android, se debe adecuar el ordenador con tal de tener todas las herramientas que se han descrito anteriormente. Lo primero es necesario tener un ordenador compatible con los entornos disponibles; es posible desarrollar para Android en sistemas Windows, en Linux y en Mac OS X.
Como se va a programar usando Java, es imprescindible tener instalado el JDK (*Java Development Kit*) de Oracle en el ordenador y usar las versiones 1.6 o 1.7 (aunque la 1.5 puede llegar a ser válida, la versión 1.7 a última revisión es lo que se aconseja para Android Studio). La 1.4 no es válida para la programación destinada a Android en ningún caso.
En caso de no tenerlo instalado o ser una versión antigua, se puede descargar gratuitamente desde la propia web de Oracle (`http://www.oracle.com/technetwork/java/javase/downloads/index.html`).

Advertencia:

Es muy importante tener instalado el JDK, ya que solamente con el JRE no es posible desarrollar.

Para facilitar el desarrollo de las aplicaciones, se utilizará el programa Android Studio de Google. Android Studio es un entorno de desarrollo basado en IntelliJ IDEA, que a su vez es un entorno de programación Java. Android Studio es, para aquellos que tengan cultura Android, como el Eclipse junto con el plugin ADT. Google ha comenzado a decantarse por hacer publicidad del uso de Android Studio frente a Eclipse por ofrecer una solución más cercana a las necesidades durante la programación Android. No obstante Android Studio, al estar basado en IntelliJ, puede llegar a utilizarse para otros lenguajes de programación como PHP o Python.

Para la instalación de Android Studio, necesitamos descargarnos el archivo correspondiente desde la web `http://developer.android.com/sdk/installing/studio.html`; la descarga es gratuita. Dependiendo del sistema operativo que utilicemos, el nombre tipo de archivo varía; para Windows es un ejecutable `.exe`, para Mac OS X se trata de una imagen `.dmg` y para Linux de un comprimido `.tgz`. A lo largo del libro trabajaremos con la instalación de Windows, pero los ejemplos serán totalmente válidos para el resto de las plataformas. En el caso de Windows, el archivo descargado será algo semejante a `android-studio-bundle-132.893413-windows.exe`. Tras la instalación del entornos (que es un asistente como en otros programas), en la primera ejecución de Android Studio, se nos permite importar las preferencias de anteriores instalaciones de Android Studio; si no ha habido anteriores instalaciones o simplemente no nos interesa, la opción seleccionada por defecto nos es válida.

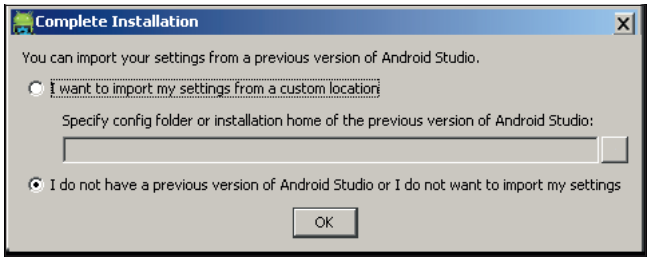

Figura 2.1. Pantalla de importación de preferencias

Cada vez que se ejecuta Android Studio, una de las tareas que realiza es ver si existen actualizaciones y ofrece al usuario la opción de descargarlas (véase las pantallas de la figura 2.2).

También es posible forzar la comprobación de actualizaciones desde la pantalla mostrada en la figura 2.3, concretamente desde el enlace disponible en la parte inferior de la pantalla (desde la palabra Check) o en la pantalla principal mediante el menú Help> Check for Update. Desde esta misma pantalla,

podemos acceder una serie de accesos rápidos (como crear nuevo proyecto o acceder a la configuración) y en la parte izquierda, se almacenarán los proyectos según se vayan realizando.

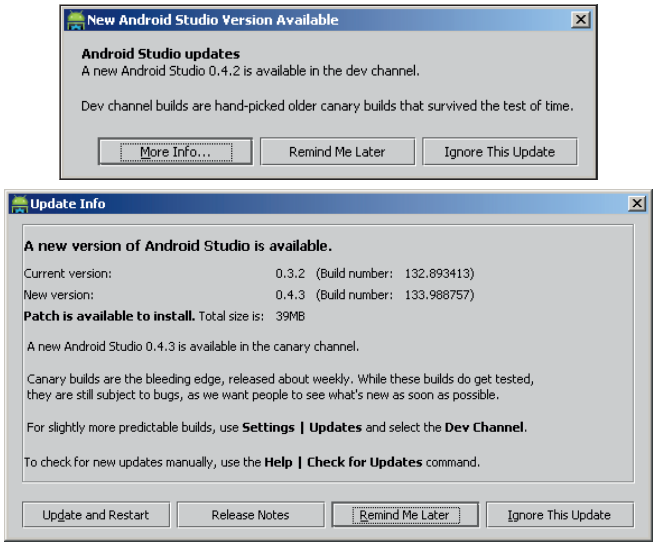

Figura 2.2. Descarga de actualizaciones

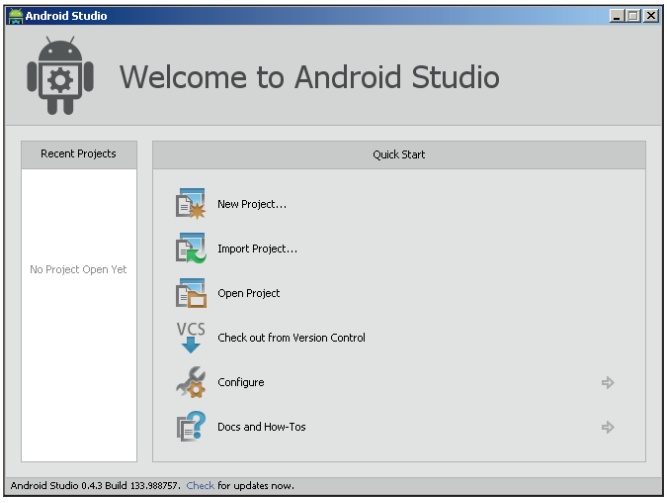

Figura 2.3. Pantalla de inicio de Android Studio

Cuando se realizan aplicaciones para Android, hay que tener en cuenta que existen numerosas versiones del sistema operativo, y que no todas tienen las mismas capacidades. Para poder trabajar con ellas, existe la utilidad "*SDK*

Manager" que permite gestionar las herramientas con las que trabajaremos, como drivers, herramientas de diseño, documentación o imágenes de sistema. Para acceder al *SDK Manager* vale con pulsar sobre Configure en la pantalla principal que nos guiará hacia una nueva pantalla donde se encuentra la entrada correspondiente a SDK Manager.

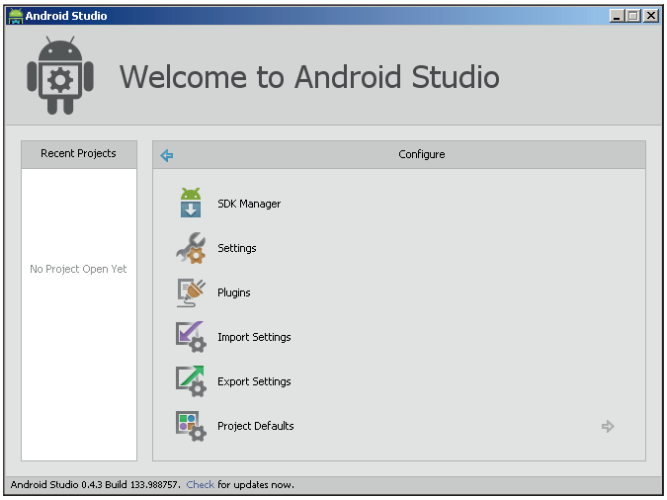

Figura 2.4. Pantalla principal de configuración

Si pulsamos sobre la opción de SDK Manager se abrirá una nueva ventana donde se muestran todos los elementos que podemos instalar y los que se encuentran actualmente instalados (incluyendo los que tienen pendientes actualizaciones).

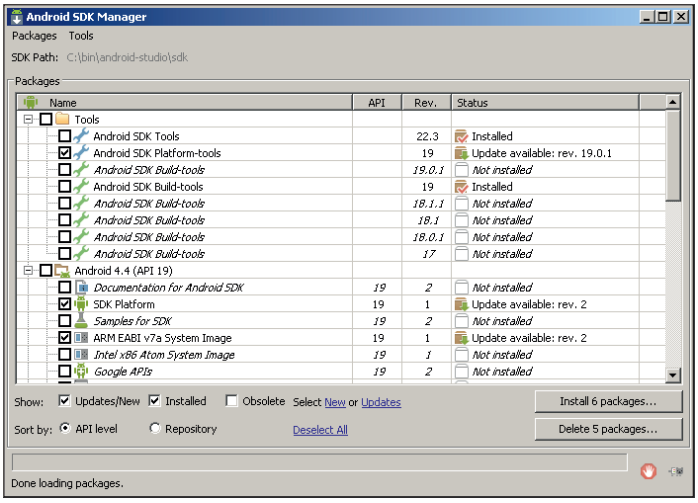

Figura 2.5. SDK Manager

Android se caracteriza por tener nombres de postres para las versiones de su sistema operativo y son las versiones que se conocen habitualmente; aunque no siempre ha sido así, las primeras versiones que salieron a la luz no tuvieron nombre oficial e incluso algunas versiones de prueba tuvieron nombres de robots de ficción como Astro Boy y Bender (por eso comienzan los postres en la C, nada que ver con saltarse Apple y Blackberry como dicen algunos). Además de las sabrosas versiones, existen versiones numéricas tanto del SDK como del nivel de API (*application programming interface*, interfaz para programación de aplicaciones). Cuando se realizan aplicaciones para Android, lo que más nos interesa es el nivel de API con la que trabajaremos. Las diferentes versiones existentes hasta el momento y su correspondencia son:

Versión	Nombre	Nivel
Android 1.0	Sin Nombre	(API level 1)
Android 1.1	Petit Four	(API level 2)
Android 1.5	Cupcake	(API level 3)
Android 1.6	Donut	(API level 4)
Android 2.0	Eclair	(API level 5)
Android 2.0.1	Eclair	(API level 6)
Android 2.1	Eclair	(API level 7)
Android 2.2–2.2.3	Froyo	(API level 8)
Android 2.3–2.3.2	Gingerbread	(API level 9)
Android 2.3.3–2.3.7	Gingerbread	(API level 10)
Android 3.0	Honeycomb	(API level 11)
Android 3.1	Honeycomb	(API level 12)
Android 3.2	Honeycomb	(API level 13)
Android 4.0–4.0.2	Ice Cream Sandwich	(API level 14)
Android 4.0.3–4.0.4	Ice Cream Sandwich	(API level 15)
Android 4.1	Jelly Bean	(API level 16)
Android 4.2	Jelly Bean	(API level 17)
Android 4.3	Jelly Bean	(API level 18)
Android 4.4	KitKat	API level 19)

Tabla 2.1. Versiones de Android

Si nos fijamos en la pantalla correspondiente al *SDK Manager*, en la parte superior izquierda aparece una etiqueta marcando el SDK Path, si vamos con el explorador de ficheros hasta esa ruta, veremos una serie de directorios:

- **add-ons:** Contiene add-ons para programar usando ciertas librerías que pueden estar presentes en dispositivos concretos.
- **docs:** Multitud de documentación accesible a través del archivo offline.html. Entre esta documentación se encuentran guías de desarrollo, API's... más adelante veremos cómo descargar documentación... aquí será donde se guarde. Este directorio puede no encontrarse disponible si no hay documentación.
- **extras:** Contienen librerías de distintos fabricantes para poder ajustar más el desarrollo a ciertas características de un dispositivo en concreto. También incluyen herramientas de Google como librerías de compatibilidad o el driver USB en Windows.
- **platforms:** Dentro de este directorio encontraremos un directorio distinto para cada una de las versiones Android que tengamos instaladas. Si ahora lo vemos vacío es porque aún no se ha descargado ninguna versión. Es otro de los directorios que puede no estar disponible hasta que no tenga contenido.
- **platform-tools:** Se encuentran herramientas para el desarrollo y depuración que son dependientes de la plataforma. Normalmente se actualizan cada vez que sale una versión (plataforma) nueva para incorporar las mejoras de esta. Son herramientas que mantienen la compatibilidad hacia atrás, por lo que no hay que preocuparse si se está desarrollando en versiones antiguas. Anteriormente estas herramientas se encontraban en el directorio **tools**.
- **samples:** Ejemplos repositorio de ejemplos descargados. Puede no estar si no hay ejemplos descargados.
- **sources:** Fuentes de las clases Android disponibles para la programación..
- **system-images:** Contiene imágenes de sistemas Android separadas por arquitectura, por ejemplo arm-eabi o x86.
- **tools:** Es donde residen las herramientas importantes para programar en Android. Aquí podemos encontrar el emulador, diversos monitores y otras muchas que iremos descubriendo a lo largo del libro.
- **temp:** Directorio temporal donde se almacenan los elementos descargados antes de proceder a su extracción en el directorio correspondiente.

Nota:

Para facilitar el uso de las herramientas ofrecidas por Android es muy recomendable añadir los directorios `tools` *y* `platform-tools` *a la variable de sistema PATH y así poder utilizar las herramientas*

tales como el `adb` *y otras desde cualquier directorio sin necesidad de incluir en el comando la ruta completa hasta éste. Para hacerlo se debe tener en cuenta el sistema que se esté utilizando:*

- **Windows:** *Pulse con el botón derecho del ratón sobre* **Mi PC** *(My Computer) y seleccione* Propiedades *(Properties). Dentro de la solapa* Avanzado *(Advanced) seleccione el botón* **Variables de Sistema** *(Enviroment Variables) y se mostrará un cuadro de diálogo. En él seleccione la variable PATH dentro de* Variables del Sistema *y añada la ruta completa hasta los directorios* `tools` *y* `platform-tools` *de la instalación del SDK.*
- **Linux:** *Se debe actualizar el fichero* `.bash_profile` *o* `.bashrc` *situado en el directorio del usuario. En estos ficheros se ha de buscar la línea donde se define la variable PATH y añadir el directorio donde estén las herramientas de Android. Si no se encuentra ninguna línea se debe crear:*

```
export PATH=${PATH}:<directorio_del_sdk>/tools:<directorio_
del_sdk>/platform-tools
```

- **Mac OS X:** *en el directorio del usuario, se debe buscar el fichero* `.bash_profile` *y hacer lo mismo que en el caso de Linux.*

SDK Manager

De los directorios descritos anteriormente, hemos visto que algunos podrían encontrarse vacíos y otros ni si quiera podrían existir; todo depende de la selección que realicemos para la descarga en el *SDK Manager*. En la parte de la izquierda de esta herramienta es posible ver entradas de tres tipos que a su vez contienen nuevas entradas que se hacen visibles al pulsar sobre las primeras:

- Tools: Son las posibles actualizaciones del conjunto de herramientas correspondientes a *tools* y *platform-tools*.
- Android X.X (API Z): Donde X.X es la versión de Android y Z es el nivel de API correspondiente a la versión. Son las distintas versiones disponibles de Android. Dentro de esta opción se encuentran las documentaciones, las imágenes para el emulador, ejemplos, las APIs de Google para acceso a mapas... Cada vez que Google saca una versión del sistema operativo Android para algún dispositivo físico, su homóloga para el emulador

aparece como descargable en esta herramienta. Cada plataforma incluye el sistema operativo (una imagen del mismo), pieles para el emulador, códigos de ejemplo y otras herramientas específicas de la versión. Así, podremos probar nuestros programas en cualquier versión del sistema Android incluso si no disponemos de dispositivo físico. En la entrada Samples for SDK, encontraremos ejemplos interesantes si queremos saber cómo se programa una nueva funcionalidad aparecida en cierta versión de API.

- Extras: Son los paquetes de librerías para facilitar el acceso a funcionalidades extras como soporte de fragmentos en versiones antiguas, librerías para ayudar a la integración de publicidad en las aplicaciones, uso de *cloud messaging* o acceso a los servicios de Google Play. Una entrada muy importante si estamos en Windows es la de Google USB Driver que se debemos instalar**.** Es un paquete que contiene unos drivers específicos de Windows para poder ejecutar y depurar aplicaciones en un dispositivo físico. No es necesario instalar este driver en sistemas Linux o Mac, ya que el driver que tienen estos sistemas operativos es válido. Igualmente, si no se desea depurar la aplicación desde el dispositivo físico, también se puede obviar su instalación aunque es muy recomendable hacerla.

Todos estos paquetes se descargan de unos repositorios concretos, algunos vienen previamente configurados y otros los podemos añadir nosotros. En caso de querer cargar nuevos repositorios, se debe pulsar sobre el menú Tools>Manage Add-on Sites... que abrirá una nueva ventana donde se podrán activar y desactivar ciertos repositorios al igual que añadir nuevos.

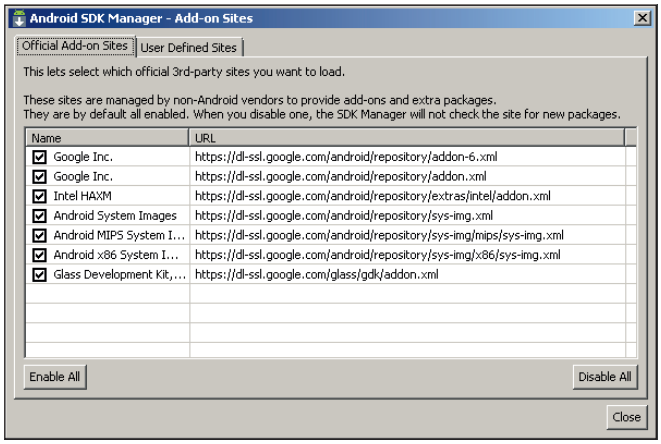

Figura 2.6. Repositorios disponibles.

En el mismo menú Tools existe la entrada Options... en caso de que necesitemos cambiar parámetros de conexión como el uso de proxy o activar la descarga *https* en caso de que falle la *http* al descargar o también acceder a paquetes que aún estén en fase de pruebas. Las otras dos entradas del menú corresponden a About, con información acerca de la versión de la herramienta y Manage AVDs... que más adelante veremos.
En la parte inferior de la pantalla, podremos mostrar u ocultar los nuevos elementos y actualizaciones de lo que se tiene instalado mediante el *checkbox* Updates/New, los paquetes instalados mediante Installed o los obsoletos (por aquello de la nostalgia) mediante Obsolete. Otra opción interesante es conocer las fuentes, los repositorios, de donde viene cada paquete, para ello haríamos servir el *radio button* Sort by Repository.
A lo largo del libro utilizaremos el SDK 4.4.2 (API 19), con lo que hay que asegurarse que se encuentre con status *installed* (instalado) o marcado para descarga el *SDK Manager.*

Advertencia:

Hay que tener en cuenta que en el futuro las versiones y las revisiones pueden cambiar. Se ha seleccionado esta versión para realizar los ejercicios por ser la más reciente; si el lector tiene un dispositivo con versión anterior, puede seleccionar la versión correspondiente para descargar, pero desde una versión superior siempre se puede permitir trabajar con una versión inferior.

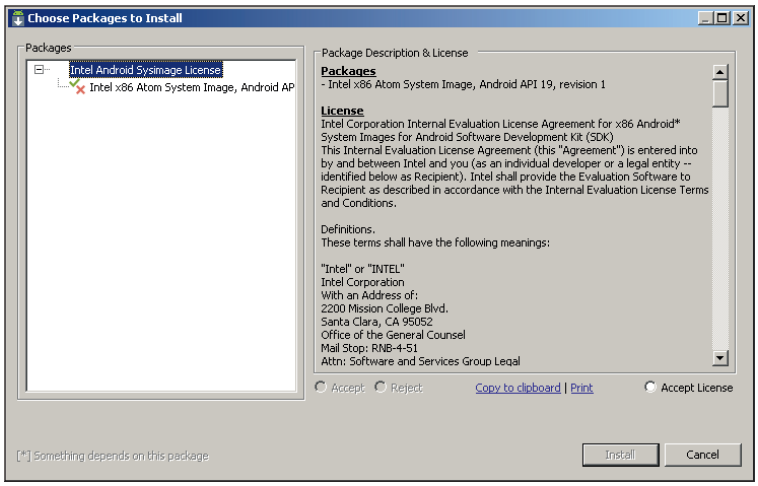

Figura 2.7. Descripción del paquete y licencia.

Tras seleccionar las opciones que se quieran, se pulsa sobre el botón Install X Packages... (X es el número de paquetes seleccionados para la instalación) y aparecerá una pantalla con información sobre las licencias.
Al aceptar las licencias y pulsar sobre el botón Install, se descargarán los componentes seleccionados e instalarán en los correspondientes directorios. La evolución del proceso se puede ver en la barra inferior de la ventana y pulsando en el botón situado abajo a la derecha muestra los reportes de actividad.

Nota:

Si tiene problemas al descargar la información del software o el propio software, asegúrese de que tiene marcada la opción de usar http en la sección de Tools>Options*.... También puede ser que necesite configurar el proxy también en esa misma sección.*

Tal y como ya sabe, las imágenes del sistema operativo Android se han almacenado en el directorio `platforms` dentro del directorio de instalación del SDK. Si se dirige allí con el explorador de archivos, verá que ya no está vacío, que se ha creado un directorio por cada una de las imágenes que se haya bajado. Si se ha marcado la plataforma especificada en el texto (*SDK Platform Android 4.2.2, API 19*) aparecerá un directorio llamado `android-19`. Dentro de estos directorios se encuentran más directorios con elementos referentes a la plataforma Android:

- **data:** Lugar de almacenamiento para fuentes y recursos.
- **images:** Lugar de almacenamiento de las imágenes de disco, incluyendo la del propio sistema operativo Android. Son las imágenes que se montarán en el emulador. Es posible que en un principio no exista el directorio; se creará cuando se genere un emulador Android para esta versión.
- **skins:** Son las pieles que podrá usar el emulador. Dependiendo de las resoluciones se usarán unas pieles u otras. Existen pieles diseñadas por usuarios en Internet que se pueden descargar e instalar en este directorio.
- **templates:** Almacenamiento de plantillas usadas por el SDK.
- **android.jar:** Aunque no es un directorio, se enumera aquí porque será utilizado en todos los proyectos. Este fichero es la librería contra la cual se compilarán las aplicaciones que se desarrollen para esta versión.

Además de gestionar las versiones de Android con las que trabajaremos, desde el *SDK Manager* se gestionan también los emuladores Android. La herramienta específica está disponible a través del menú `Tools>Manage AVDs...` AVD es la abreviatura de *Android Virtual Devices* o Dispositivos Virtuales Android.

Pulsando sobre el botón **New**... obtenemos una ventana sobre la cual generar nuestro dispositivo virtual, el emulador sobre el que probar las aplicaciones.

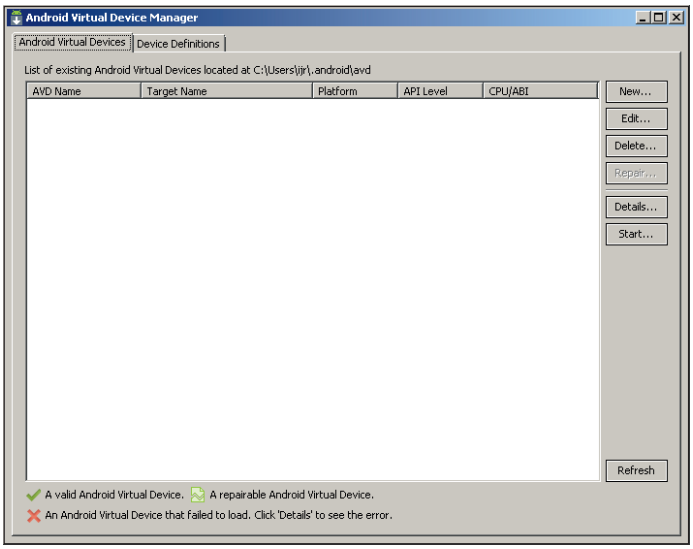

Figura 2.8. Gestión de AVD

Crearemos un dispositivo virtual para trabajar más adelante con él.
En la caja de texto AVD Name se podrá informar un nombre al dispositivo (recuerde que podemos hacer tantos dispositivos virtuales como queramos).Mediante el desplegable Device se puede seleccionar el dispositivo sobre el que probar la aplicación, por ejemplo con el Nexus 4; estos dispositivos se pueden configurar desde la pestaña Device Definitions de la pantalla que se puede ver en la figura 2.8. En la lista de selección Target es posible elegir qué versión de Android queremos usar como sistema operativo de nuestro nuevo dispositivo (se mostrarán solamente los instalados en el sistema). El siguiente desplegable corresponde a CPU/ABI que es el tipo de CPU que tendrá el dispositivo virtual, el más normal en este caso es el ARM aunque el usuario puede optar por usar Intel Atom. Mediante los *checkbox* de Keyboard Keyboard y Skin se controla si debe aparecer el teclado y los botones de hardware. Con los desplegables de cámaras podemos indicar que cuando se active la cámara en el emulador utilice una de nuestras webcam por ejemplo. Dentro de Memory Options se dispone de la cantidad de memoria que se desea que tenga el dispositivo; en entornos Windows si seleccionamos memoria superior a 768 Mb es posible que obtengamos error al ejecutar el dispositivo virtual, dependiendo de la configuración y estado del propio Windows, para evitar este inconveniente, podemos bajar la cantidad de memoria ram de 1907Mb (por defecto en el Nexus 4) a

500Mb. El Internal storage indica la memoria interna del dispositivo sobre el que se va a emular (que no es lo mismo que la RAM que es la memoria donde se ejecuta el programa, ni la SDCARD, que es la memoria externa). En el apartado SD Card permite seleccionar si se quiere emular una *SDCard* como almacenamiento en el dispositivo o no. En caso de que no se quiera, vale con dejar la caja de texto en blanco; si se quisiera usar alguna se le debe indicar un tamaño de almacenamiento y el asistente ya se encargará de crear un fichero en el ordenador con el tamaño indicado y lo usará como almacenamiento externo del dispositivo. También es posible usar un fichero anteriormente creado para otro dispositivo (a lo mejor puede interesarnos tener la misma información en dos dispositivos a la vez). Por último, tenemos las opciones de emulación Snapshot y Use Host GPU, la primera de ellas sirve para guardar el estado de la imagen entre distintas ejecuciones y la segunda sirve para habilitar la emulación del OpenGLES desde nuestro ordenador.

Para nuestro caso seleccionaremos como nombre `"avd_api19_128"`, en Device seleccionaremos Nexus 4, en el Target elegiremos Android 4.4.2 API Level 19. Vamos a darle una tarjeta SD de 128Mb, para ello seleccionamos la opción Size en el bloque SD Card e introducimos el tamaño en la caja de texto. Una vez configurado el terminal a crear, se pulsa sobre el botón OK para que realmente lo cree en el sistema.

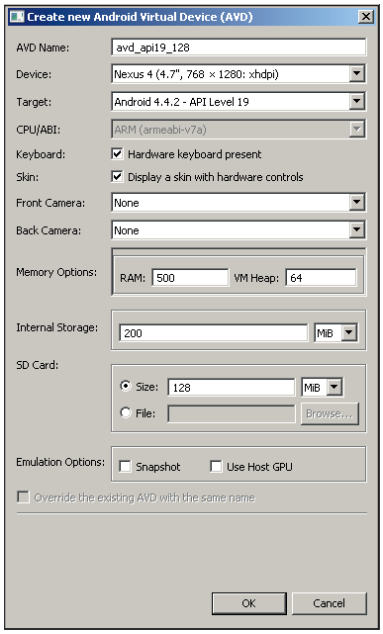

Figura 2.9. Creación de AVD

A partir de este momento tendremos el dispositivo recién creado accesible en la herramienta "*Android Virtual Device Manager*, desde donde podremos administrarlo, borrarlo o ejecutarlo. Si lo seleccionamos y pulsamos sobre el botón Start se iniciará el dispositivo virtual (emulador) y arrancará el sistema Android 4.4.2 (tal y como lo habíamos configurado). Antes de mostrar el emulador con la imagen del sistema operativo Android, aparece una pantalla donde se da la posibilidad de ajustar ciertos parámetros del emulador y de la imagen antes de ser ejecutada.

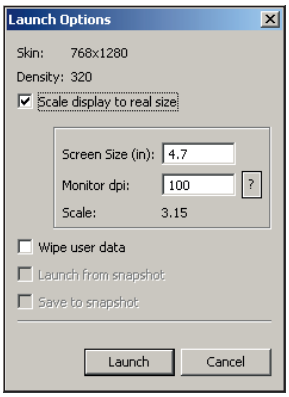

Figura 2.10. Opciones de ejecución del emulador.

Si se selecciona la casilla Scale display to real size, lo que se hará es indicar al emulador que se ajuste la salida en pantalla al tamaño que tendrá en el dispositivo real. Dependiendo del tamaño de pantalla seleccionado, es posible que en la pantalla del ordenador se vea pequeño, no obstante va bien para hacerse una idea de si una aplicación se verá bien o no en un dispositivo en concreto.

La segunda casilla (Wipe user data) sirve para reiniciar los datos de usuario, es decir, para reiniciar la imagen y borrar los datos que se hayan introducido hasta el momento. Deje las dos casillas en blanco y pulse sobre el botón **Launch** para que se ejecute la imagen sobre el emulador. Al cabo de unos instantes el emulador mostrará una pantalla bloqueada de Android. Para desbloquearla y poder utilizar el sistema, vale con pulsar y arrastrar el candado hacia la derecha.

En el emulador se puede trabajar como si fuera un dispositivo real, y podemos hacer cosas como cambiar el fondo de escritorio, emular llamadas… Pero lo que nos interesa ahora es ver como quedaría una primera aplicación.

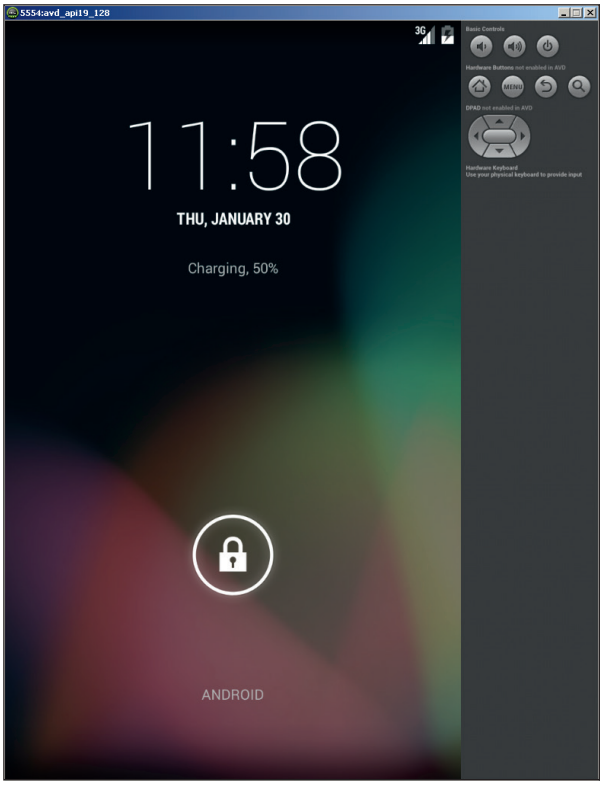

Figura 2.13. Emulador con la pantalla bloqueada.

Hola mundo

Es hora de hacer nuestra primera aplicación… no vamos a lanzar las campanas al vuelo, será una pequeña aplicación que, como no podría ser de otra manera, muestre un mensaje de Hola Mundo. Desde que en 1974 se utilizara como parte de un escrito de B. Kernighan es el ejemplo más utilizado dentro del mundo de la programación, así que no será este libro una excepción.
Abrimos Android Studio (si es que no está ya abierto) y desde la pantalla principal (figura 2.3) seleccionamos New Project.... En el campo Application Name escribiremos el nombre de la aplicación, por ejemplo "Mi Hola Mundo Androide", en el campo Module Name, indicamos el nombre que tendrá el módulo dentro del entorno de desarrollo; es un nombre interno que suele ser el mismo que el de la aplicación pero no tiene por qué (y en proyectos complejos podemos tener más de un módulo por aplicación), en nuestro caso será "Hola Mundo".

La siguiente caja de texto es para indicar el paquete Java en el que guardaremos la aplicación, para esta aplicación será "com.acme.holamundo". A continuación vienen varios desplegables que dejaremos con las selecciones por defecto. Estos desplegables permiten seleccionar la versión mínima de Android que se necesita para ejecutar la aplicación que estamos realizando, el siguiente sirve para decir la versión de SDK con la que se quiere trabajar (que debe ser la más alta en la que se conozca que funciona la aplicación), a continuación se selecciona la versión de Android con la que se compilará el proyecto (de todas aquellas versiones que se tengan instaladas en la máquina de desarrollo) y el último desplegable que es el referente al tema a utilizar, que depende de las versiones seleccionadas en los desplegables anteriores. El selector de Minimum Required SDK es muy importante puesto que sirve para discriminar Androids con sistemas operativos antiguos en caso de que se vaya a utilizar cierta característica que esté presente a partir de un nivel de API concreto y no podrán instalar la aplicación si no cumplen este mínimo nivel. Marcando el *checkbox* Create custom launcher icon, en el siguiente paso del asistente se nos mostrará una pantalla donde seleccionar el icono; lo dejaremos marcado. El *checkbox* llamado `Create Activity` hace que más adelante se nos pidan datos sobre la actividad a crear en el proyecto, déjelo marcado.
El siguiente *checkbox* es para cuando estemos trabajando en una librería, como no es el caso, lo dejaremos desmarcado.
El aspecto final de la pantalla será:

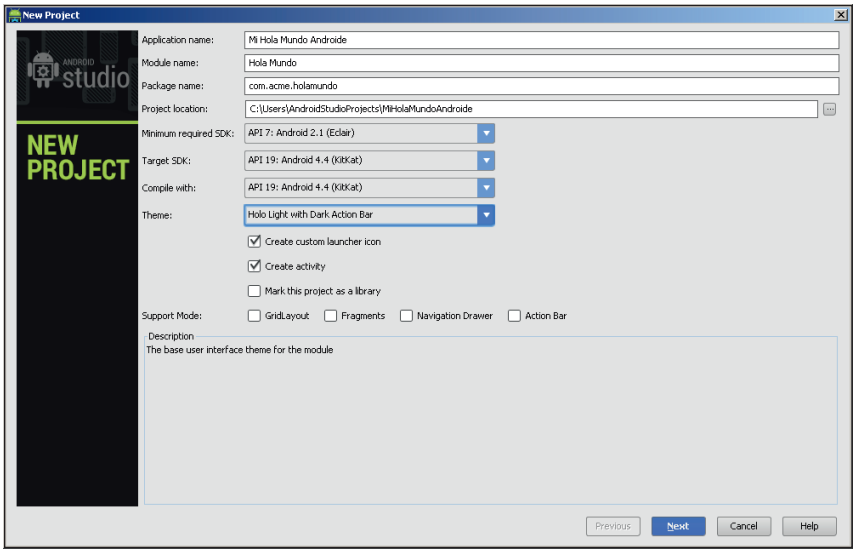

Figura 2.12. Pantalla de configuración para el proyecto Hola Mundo.

Pulsando sobre el botón **Next** avanzamos a la siguiente pantalla donde se nos ofrece la posibilidad de elegir el icono de la aplicación, ya que dejamos marcado el selector correspondiente. Seleccionando la opción Image podemos seleccionar un icono de nuestro sistema de archivos, pero si queremos aprovechar alguno de los ofrecidos por el SDK de Android, vale con seleccionar la opción Clipart.
En esta pantalla, además de seleccionar el icono de la aplicación, jugar con los colores de primer plano y fondo así como con las formas de los iconos; a la derecha podemos ver una previsualización de cómo sería el resultado final usando la configuración seleccionada. Cuando estemos conformes con el icono a usar en la aplicación pulsamos, sobre el botón **Next** para acceder a la siguiente pantalla.

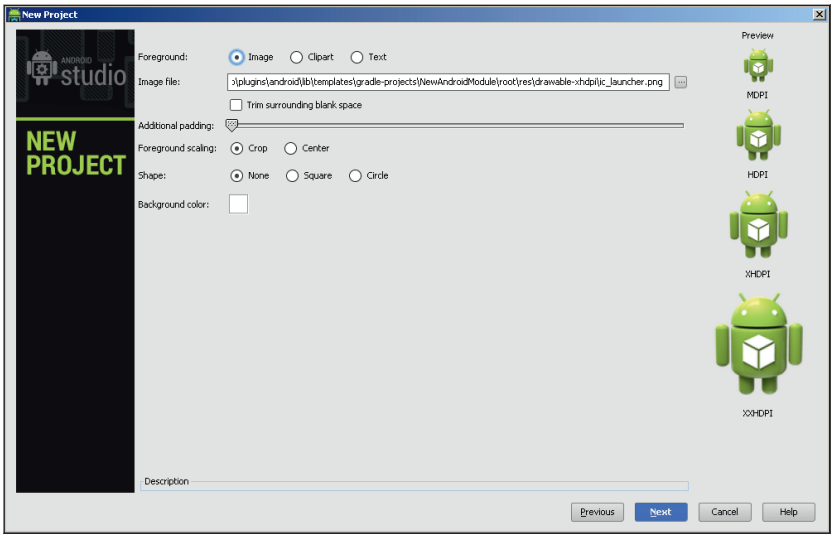

Figura 2.13. Selección del icono de la aplicación.

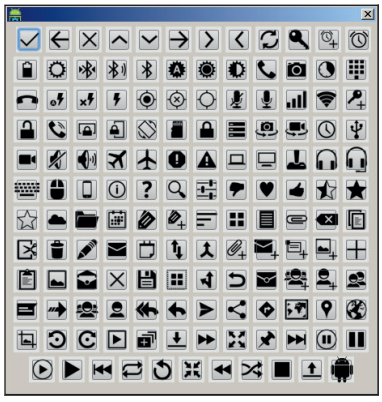

Figura 2.14. Iconos disponibles en el SDK.

En esta nueva pantalla se configura de qué tipo queremos que se genere la actividad principal; dependiendo de la selección en la primera pantalla del asistente, podremos seleccionar unas u otras opciones; por ejemplo la actividad de tipo MasterDetailFlow es para aplicaciones Android con una versión de API 11 o superior, es decir, si trabajamos con un nivel 10 de API, no podríamos usarla. Seleccionamos BlankActivity. Pulsando sobre **Next** llegamos a la última pantalla del asistente.

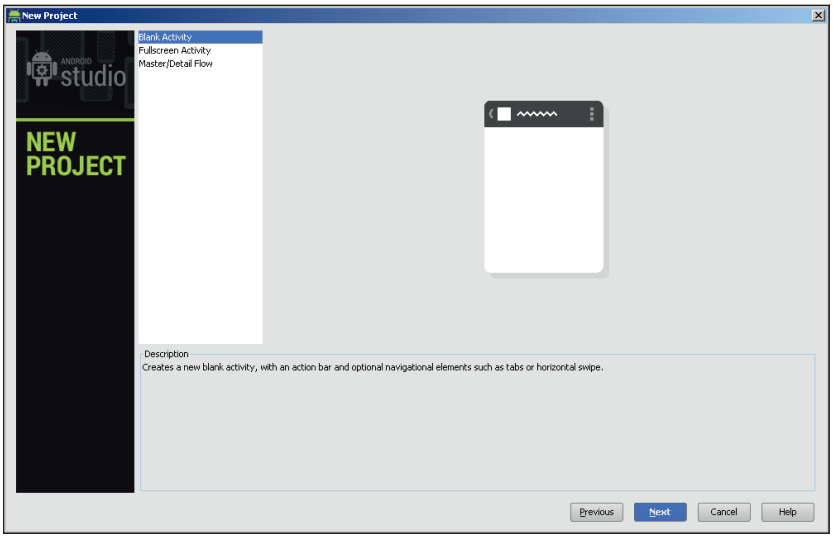

Figura 2.15. Tipo de actividad.

En esta última pantalla se termina de perfilar el esqueleto de la aplicación, seleccionando el nombre de la actividad principal, el tipo de disposición de los elementos en pantalla y características adicionales. Por el momento no cambie nada y más adelante ya irá descubriendo qué son el resto de opciones (véase la figura 2.16).

Pulsamos sobre **Finish** para que se genere el esqueleto de nuestro primer programa Android. En ese momento veremos un pequeño diálogo informándonos de la evolución de nuestra compilación.

En estos momentos ya se tiene un programa Android perfectamente ejecutable. Para poder lanzar la aplicación construida, se va a necesitar un dispositivo donde probarlo que puede ser bien un emulador o bien un terminal físico. En este primer intento lo haremos sobre un dispositivo virtual en el emulador (véase la figura 2.17).

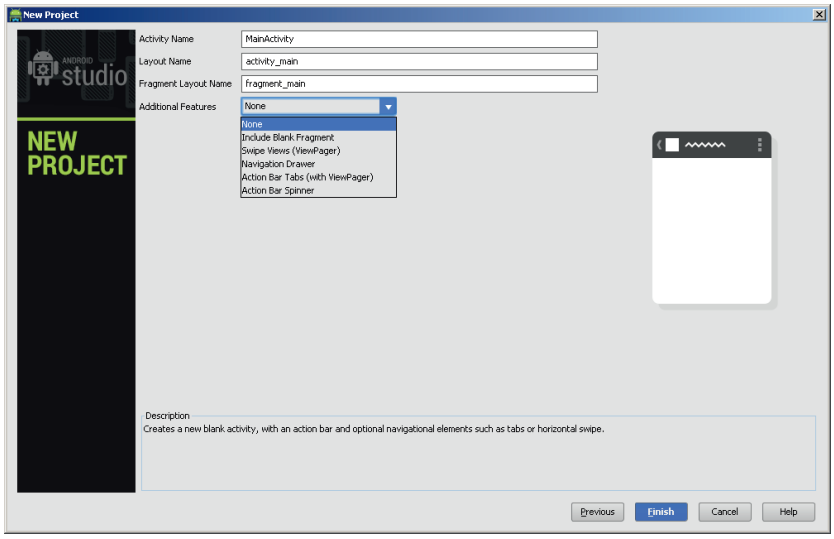

Figura 2.16. Configuración de las actividades y navegación.

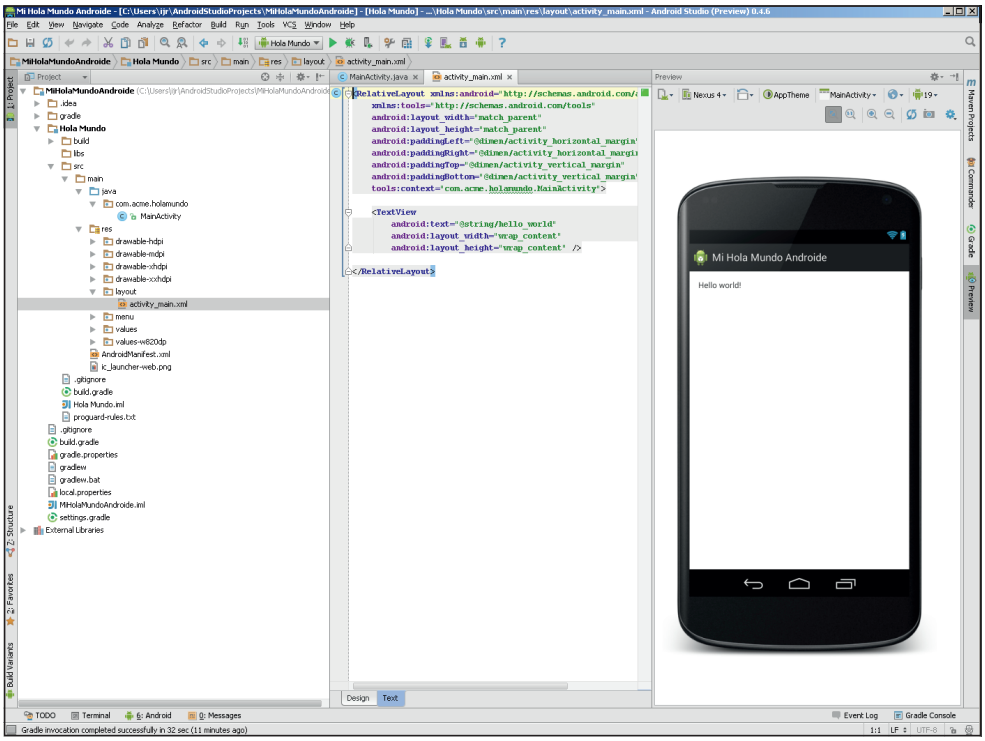

Figura 2.17. Android Studio con el proyecto generado.

Anteriormente hemos creado un dispositivo virtual. Si no está en ejecución en este momento, póngalo en marcha a través de la utilidad "*AVD Manager*" a la cual puede acceder bien como se vio anteriormente, mediante el menú Tools>Android>AVD Manager o mediante el botón de la barra de herramientas  . Una vez esté ejecutándose el emulador, instalaremos la aplicación en él y la ejecutaremos; todo esto se hace a través del botón de la barra de herramientas  . Se abrirá una ventana preguntándonos dónde queremos ejecutar la aplicación, si en un dispositivo que esté conectado o si queremos lanzar un nuevo emulador. Si hemos esperado a que el emulador se arrancara completamente, aparecerá en la lista superior

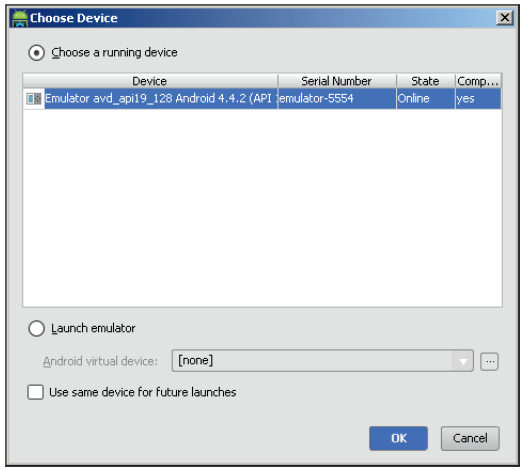

Figura 2.18. Selección del dispositivo donde ejecutar el programa.

Seleccione el emulador ya en ejecución o el dispositivo físico si dispone de él y lo tiene convenientemente conectado al ordenador. Pulsando sobre el botón OK se instalará y ejecutará la aplicación. Si todo ha ido bien veremos una pantalla con el texto "HelloWorld!" (se debe vigilar que la pantalla del emulador no esté bloqueada, si lo está, habrá que desbloquearla antes de poder ver nuestro programa). Si sale de la aplicación y quiere volver a ella, puede encontrarla en el menú donde se encuentran todas las aplicaciones Android instaladas en el dispositivo, pulsando en la parte inferior de la pantalla.

Hay que reconocer que no es la aplicación más espectacular, pero es nuestra primera aplicación. En próximos capítulos se diseccionará y modificará el código generado por el asistente y se explicarán aspectos fundamentales de las aplicaciones Android.

Figura 2.19. Programa Hola Mundo en ejecución.

3 Conceptos básicos

En este capítulo aprenderá a:

- Diferenciar los distintos bloques para componer una aplicación.
- Manejar los ciclos de vida de la aplicación.
- Conservar las variables entre los distintos estados de una aplicación.

En este capítulo se introducirán conceptos propios de la programación para Android y se explicará el funcionamiento de los ciclos de vida de las aplicaciones. El objetivo de el capítulo no es que el lector sea un experto en el uso de los componentes que se explican, sino que se vaya familiarizando con ciertos términos que se irán utilizando en otros capítulos y que se acabarán asimilando por su uso en los ejemplos de programación.

Maquina virtual Dalvik

Como se ha comentado en el primer capítulo de este libro y como habrá podido comprobar el lector durante la instalación del entorno de desarrollo (y si ha sido curioso y ha abierto las clases generadas en el ejemplo del Hola Mundo también lo habrá podido constatar), las aplicaciones Android se programan principalmente en Java (también es posible realizarlo en otros lenguajes como C pero no entraremos en ello), lo cual no quiere decir que el código que se ejecuta en el dispositivo sea un *bytecode* Java.

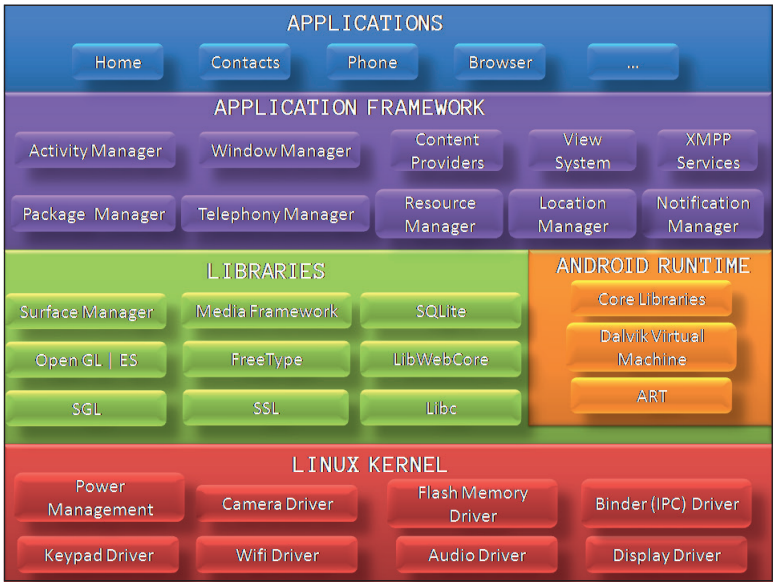

Figura 3.1. Arquitectura del sistema operativo Android

Si echamos un vistazo al gráfico de la arquitectura del sistema operativo Android, se puede observar que se compone de varias capas. Las capas superiores basan su funcionamiento en las capas inferiores, es decir, se apoyan en ellas

para llevar a cabo su cometido (eso no quita que elementos de la misma capa se necesiten mutuamente). Las aplicaciones que realizaremos se ejecutarán en la parte superior de la pila, en la capa de *Applications*. Estas aplicaciones harán uso de los múltiples gestores que Android proporciona en su capa denominada *Application Framework*. A su vez todos ellos necesitan de una serie de librerías que ayudarán a crear programas de manera más sencilla (como OpenGL o Webkit) y que están realizadas en código nativo C. Al lado de las librerías encontramos lo que se denomina el *Android Runtime*, compuesto por la *Dalvik Virtual Machine* (Máquina Virtual Dalvik), *ART Virtual Machine* (que veremos más adelante) y las librerías que lo acompañan. Esta máquina virtual será la encargada de traducir el *bytecode* de las aplicaciones en código nativo entendible por el dispositivo. Por último encontramos el *kernel* (núcleo) basado en Linux (versión 2.6 para versiones anteriores a Android 4.0 Ice Cream Sandwich y versión 3.0 del kernel para posteriores; actualmente en Android Kit Kat se utiliza un kernel con versión 3.4.10), que es el corazón del sistema operativo Android.

Cuando se compila una aplicación de Android en lugar de generar archivos .class o .jar con *bytecode* Java se generan unos archivos .dex (*Dalvik Executables*). Lo que hay dentro de estos archivos es el resultado de compilar el código java generado y combinarlo junto a las librerías utilizadas en uno o varios archivos .dex mediante la herramienta `dx`. Durante el proceso a archivos tipo .dex se reduce el tamaño del fichero resultante reutilizando información duplicada entre las distintas clases y convirtiendo el juego de instrucciones Java a un nuevo juego de instrucciones propio de la maquina Dalvik. Si se genera un archivo.dex y uno .jar sobre las mismas clases Java, el resultado será que el archivo .dex sin comprimir es más pequeño que el archivo .jar comprimido, lo que da cuenta de la optimización del bytecode resultante. Otro aspecto en el que la máquina Dalvik es muy superior a otras máquinas virtuales es en la gestión de memoria de los objetos obsoletos, o lo que se llama *Garbage Collector* (o recolector de basura). Cuando un objeto en Java deja de ser utilizado, no se hace como en otros lenguajes que se liberan los recursos mediante programación, sino que es el propio *Garbage Collector* quien revisa los objetos que no son utilizados y libera memoria. Cuantas más veces se ejecute este recolector de basura, más limpia estará la memoria y más recursos habrá disponibles, pero hay que tener en cuenta que cada vez que se ejecuta, se ocupan ciclos de CPU que no pueden ser usados por las aplicaciones, es decir las aplicaciones irán más lentas. Para Dalvik se ha realizado un *Garbage Collector* con nuevos algoritmos de modo que su eficiencia es mucho mayor.

Además de las ya nombradas, la máquina Dalvik ofrece otras muchas características que no vamos a entrar a nombrar aquí aunque hay una de gran relevancia y es la opción del JIT (*Just-In-Time compilation* o compilación en

tiempo real). Hasta hace poco, las aplicaciones podían ser o bien compiladas o bien interpretadas, siendo las primeras más rápidas por ser código nativo y siendo las segundas más versátiles puesto que cambiando la máquina que las interpretaba se conseguía que fueran multiplataforma pero más lentas ya que se tiene que estar constantemente interpretando de código de alto nivel a código nativo. Con el JIT se llega a una comunión entre las dos técnicas de programación. El *bytecode* generado se interpreta continuamente por la máquina virtual Dalvik, pero a su vez se va guardando en una caché para no tener que traducirlo a código nativo si ya se ha hecho anteriormente. Esta opción de JIT se introdujo en el Android 2.2 y la variación de velocidad fue más que notable; las aplicaciones se ejecutaban entre 2 y 5 veces más rápido con él que sin él, siempre dependiendo de la naturaleza y exigencias aplicación.

Máquina virtual ART

El problema más grande que presenta Dalvik en su versión actual, es que la primera vez que se ejecuta el programa, tarda mucho en estar disponible para el usuario. Siguiendo un poco el proceso, el usuario instalaría la aplicación en su dispositivo, al ejecutarla, Dalvik compilaría el código para poder ejecutarlo de manera más rápida, y el resultado de esta compilación se mantiene almacenado en unas memorias caché para acceder más rápido a él durante la ejecución. Cuando se mata la aplicación (bien manualmente o bien automáticamente por razones de rendimiento) y se vuelve a ejecutar la misma aplicación, implica que se debe volver a ejecutar todo el proceso de compilación, con el consiguiente gasto de batería (la compilación es un proceso "caro" en consumo de batería) y tiempo que implica.
En Android 4.4 KitKat aparece una nueva máquina virtual llamada ART que son las siglas de Android RunTime (efectivamente en el nombre no se han esmerado mucho). Actualmente se encuentra como una máquina virtual aparte, pero no se descarta que pueda llegar a sustituir a Dalvik o que se convierta en una nueva versión de ésta.
Entre las mejoras que ofrece ART frente a Dalvik podemos destacar dos:

- Nuevo GC (*Garbage Collector*, Recolector de "basura"). Se trata de un programa que se va ejecutando cada cierto tiempo, eliminando de la memoria objetos de los programas que no se usan, liberando así recursos. Es muy importante que el *Garbage Collector* sea eficiente, puesto que si no se ejecuta, se llenaría la memoria y no podríamos trabajar y si se ejecuta demasiadas

veces, además de impactar en la batería, tendríamos el problema de que mientras se ejecuta, no pueden ejecutarse otras aplicaciones, con lo que impactaría en el uso del dispositivo que se volvería torpe.

- Compilación de tipo AOT (*Ahead-of-Time*, antes de tiempo) en lugar de JIT. Este tipo de compilación mejora la velocidad (sobre todo en el tiempo de arranque de la aplicación) y reduce el *footprint* de memoria (la huella, la cantidad de memoria) utilizada.

Mediante AOT lo que se hace es que la compilación a código nativo se produce sólo una vez y lo que se guarda en el dispositivo es directamente código nativo y no bytecode eliminando así la latencia de la compilación cada vez que se ejecuta la aplicación. AOT se puede decir que es como JIT pero que sólo se ejecuta una vez, es decir cuando se descarga una aplicación y se instala, automáticamente se pre-compila para tenerla disponible cuando se quiera ejecutar. Con esto se tiene que el tiempo de instalación es mayor en ART que en Dalvik pero el de ejecución es mucho menor en ART, hablándose de reducciones de cerca 90% de tiempo necesario para ejecutar la aplicación, además de lo que implica en alargar la vida de la batería por no tener que compilar cada vez que se utiliza la aplicación.
Por defecto en Android 4.4 ART viene desactivado en favor de Dalvik ya que aún se encuentra en fase embrionaria e incluso puede haber aplicaciones que no funcionen correctamente. Para activarlo debemos dirigirnos a la aplicación de `Ajustes` de Android, seleccionar Opciones de desarrollador y Elegir tiempo de ejecución.

Nota:

ART en Android 4.4 KitKat se encuentra en fase de pruebas, por lo que es posible encontrar algunas aplicaciones que no funcionen de modo correcto. (Véase figura 3.2).

Hay que tener en cuenta que hemos estado diciendo que la compilación se produce al instalar la aplicación, eso quiere decir que si pasamos a trabajar con ART, hay que reiniciar el dispositivo y durante este primer arranque habrá que esperar un tiempo (dependiendo de las aplicaciones instaladas y de la potencia del dispositivo puede ser hasta más de media hora) para que se precompilen las aplicaciones que ya tenemos instaladas, pero esto solamente se producirá durante el primer arranque del sistema operativo tras el cambio, el resto de arranques volverán a ser rápidos puesto que ya tendrá pre-compiladas las aplicaciones.

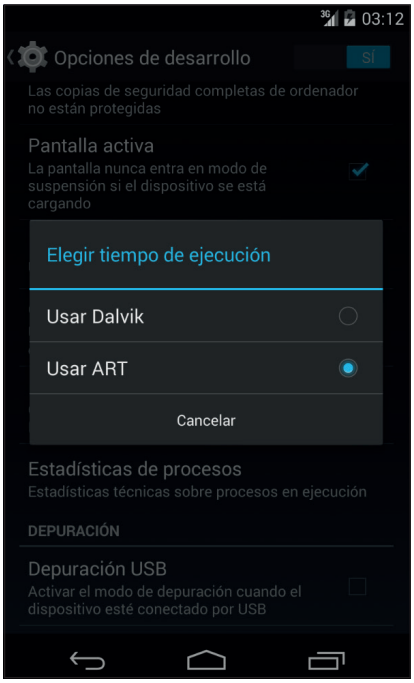

Figura 3.2. Selección de ART

Como dato curioso, comentar que la librería implicada en el *runtime* Dalvik es la `libdvm.so` mientras que en ART es la `libart.so`.

Bloques

A la hora de programar aplicaciones de escritorio, en caso de tener que comunicarnos con otros programas usamos APIs de sistema operativo, ficheros, bases de datos, conexiones sockets... Android también ofrece maneras de comunicarse entre aplicaciones de modo sencillo, reutilizable y sobre todo seguras. En este capítulo veremos las piezas que Android pone a nuestra disposición para que mediante su combinación, podamos crear aplicaciones. Para escribir una aplicación Android podemos utilizar cinco bloques fundamentales denominados: Activity (actividad), Broadcast Intent Receiver (receptor de emisiones de intentos), Service (Servicio), Content Provider (proveedor de contenido) y Fragment (fragmento) aparecido en la versión 3.0 de Android aunque puede ser utilizado en versiones anteriores mediante librerías de compatibilidad. Alguno de ellos tiene nombre lo suficientemente exótico como para necesitar una explicación más profunda.

Activity

Es el bloque más común de los cuatro, el que más usaremos en nuestras aplicaciones. Aunque es posible realizar una *Activity* sin representación gráfica, se puede decir que una *Activity* corresponde a una ventana o a un cuadro de diálogo en una aplicación de escritorio. Por ejemplo una aplicación para mantener una lista de la compra, podríamos definirla con una *Activity* para introducir un nuevo elemento a la lista, otra *Activity* para la lista en sí y otra *Actitity* para el detalle de la entrada y luego se deberían orquestar para mostrar cada una dependiendo de la acción realizada por el usuario. Una *Activity* es una clase donde mostraremos *Views* (vistas) para generar la interfaz de usuario y seremos capaces de responder a eventos que se realicen sobre ella. Pese a que el conjunto de ellas forman la aplicación, son entidades independientes que son capaces de llamarse entre ellas, pasándose parámetros y recibiendo respuestas de modo que su funcionamiento sea el de un todo. Así cuando una *Activity* necesita de otra nueva *Activity*, la configura, le prepara los parámetros de llamada, la inicia y espera su respuesta, por ejemplo podemos requerir la actividad que tiene el sistema por defecto para la selección de los contactos telefónicos y recoger el contacto seleccionado en nuestra aplicación, sin tener que programar el acceso a la base de datos de contactos telefónicos ni conocer su estructura. Si algo ya funciona... ¿Por qué volver a programarlo?

Cada vez que una *Activity* llama a otra, la actividad que crea la llamada se introduce en un histórico de llamadas, en una pila LIFO (*Last Input First Output*, último en entrar primero en salir), como si de una pila de libros se tratara, así el usuario pulsando el botón de "volver atrás" del teléfono podría recuperar la actividad anterior. Esto no quita que el programador mediante código pueda gestionar directamente la pila de llamadas y eliminar de ellas las actividades que no interesen. Android mantiene una pila de histórico de actividades distinta por cada aplicación que se lance, conservando así totalmente aisladas cada una de las aplicaciones que estén en funcionamiento.

A cada *Activity* se le asigna una ventana sobre la que se dibujará la interfaz de usuario. Esta ventana puede ocupar o no toda la pantalla del dispositivo, dependiendo de cómo haya sido configurada. Dentro de esta ventana el programador define qué elementos visuales y en qué lugar se van a mostrar. Al igual que en otras plataformas de programación, el contenido de la ventana se indica de modo jerárquico mediante vistas, que son objetos que implementan la clase *View* (vista). Así se tiene una *View* padre que contiene a las *View* hijas, y será cargo de cada padre conocer la disposición en pantalla de cada una de sus *View* hijas. Las *View* se dibujan en pantalla y son el nexo de unión entre las *Activity* y el usuario, ya que se encargan de recibir los eventos realizados por

éste sobre el espacio de pantalla donde se ha dibujado la *View*. Así podemos tener una *View* que sea un botón, una caja de texto, un desplegable... y cada una de estas *View* debe saber cómo responder a cada interacción. Muchos son los elementos gráficos que se derivan de la clase *View*, desde marcos que simplemente sirven para agrupar elementos hasta los botones, cajas de texto etc. De este modo se pueden clasificar en dos grupos que son las *View* que pueden contener otras *View* y las que no, las que consideramos *View* finales. Dentro de la distribución de las clases en los paquetes Java, las primeras las encontramos dentro de `android.view.*` mientras que las segundas están en `android.widget.*` aunque todas hereden la clase *android.view.View*.
A la hora de definir la jerarquía de una *View* en pantalla, se indica en cada actividad mediante la llamada al método.

```
activity.setContentView(viewHierarch)
```

Donde `viewHierarch` es un objeto que implementa la clase *View* y que a su vez es el padre de la jerarquía de la vista. Suena complicado, pero comprenderá en breve que no lo es en absoluto. Como ya se verá más adelante, esta jerarquía se puede hacer en tiempo de diseño de la aplicación mediante ficheros XML o bien mediante programación en tiempo de ejecución.

Broadcast Intent Receivers

Un *Broadcast Intent Receiver* (receptor de emisiones de intentos o más entendible: receptor de mensajes) es un componente que simplemente se encarga de recibir y reaccionar frente a ciertos mensajes emitidos por el sistema. Son muchos los mensajes que emite el sistema a lo largo de su ejecución, por ejemplo si se ha tomado una fotografía, si se ha activado el GPS, si se recibe una llamada… Incluso las aplicaciones pueden emitir sus propios mensajes para que otras aplicaciones se den por enteradas de que se ha realizado alguna acción, por ejemplo si una aplicación realiza cálculos y en su finalización tiene que avisar a otra para que realice alguna acción con el resultado de los cálculos realizados. Esta emisión se hace mediante el método:

```
Context.sendBroadcast();
```

Cada aplicación puede tener tantos *Broadcast Intent Receiver* como considere necesarios, del mismo modo que puede emitir tantos mensajes como sean oportunos, eso sí, todos los receptores deben extender la clase *BroadcastReceiver*.
Para que un *BroadcastReceiver* sea accesible al sistema, este debe registrarse mediante el fichero `AndroidManifest.xml` o mediante programación a través del método:

```
context.registerReceiver()
```

Los *Broadcast Intent Receiver* no tienen una interfaz gráfica asociada, pero como hemos visto, valdría con que lanzaran una actividad como reacción al mensaje recibido. Otra acción muy utilizada por los *Broadcast Intent Receiver* es la notificación al usuario a través del *NotificationManager*, que permite alertar de múltiples modos, como por ejemplo poniendo un mensaje en la barra de estado o haciendo vibrar el dispositivo o encendiendo los leds de aviso.
Para que un *Broadcast Intent Receiver* funcione, no es necesario que la aplicación esté ejecutándose, al estar registrado, es el propio sistema quien se encargará de lanzar la aplicación si hiciera falta cuando se reciba el mensaje.

Service

Las actividades tienen un periodo de vida corto y pueden estar ejecutándose y al poco tiempo ser desechadas. Los *Services* (servicios) están diseñados para mantenerse ejecutándose (si fuera necesario) sin depender de ninguna *Activity*. Cada servicio debe extender la clase *Service*. Típicos *Services* son aquellos que periódicamente se conectan a algún servidor para ver si ha cambiado información o un reproductor de música que puede seguir reproduciendo sonido aunque estemos viendo otra aplicación distinta del propio reproductor. Los servicios se inician mediante la llamada:

```
context.startService()
```

De este modo se ejecuta el código en segundo plano y no depende de la *Activity* que lo haya lanzado, así que la podremos cerrar sin problemas ya que se seguirá ejecutando el *Service*.
Es posible conectarse a un servicio e incluso iniciarlo si no está activo mediante el método:

```
context.bindService()
```

Una vez conectado al servicio, podrá comunicarse con él a través de la interfaz que el propio servicio haya hecho pública. En el caso del reproductor de música, el usuario podrá pausar la reproducción o cambiar de canción y volver a abandonar la *Activity* que se comunica con el servicio y el servicio seguirá funcionando.

Content providers

Los *Content Providers* (proveedores de contenido) proporcionan una capa de abstracción para acceder a datos almacenados por una aplicación de modo que puedan ser accesibles a otras aplicaciones. Las aplicaciones pueden guar-

dar su información en la base de datos SQLite que proporciona Android, en ficheros o en otro sistema de almacenamiento. Mediante los *Content Providers*, una aplicación puede hacer públicos sus datos a otras aplicaciones de manera sencilla y sin miedo a que se puedan corromper. Esta es la motivación de hacer un *Content Provider,* la de poder hacer accesibles los datos de nuestra aplicación a otras aplicaciones, pero manteniendo el control de cómo se acceden a ellos, como si fuera una API a nuestros datos. Los *Content Providers* deben extender la clase *ContentProvider* e implementar una serie de métodos estándar para hacer posible a otras aplicaciones grabar y leer los datos que él gestiona. Las aplicaciones no llaman estos métodos directamente, sino que lo que hacen es usar un objeto de la clase *ContentResolver* y utilizar sus métodos. Un *ContentResolver* es capaz de interactuar con cualquier *Content Provider* y facilitar así los procesos de lectura y escritura.

Fragment

Los *Fragmens* (fragmentos) aparecen a partir de la versión 3.0 de Android para solucionar el problema de las múltiples pantallas. Su cometido principal es la reutilización tanto de código de lógica de trabajo como de las interfaces de esos códigos. Hasta entonces la unidad indivisible en Android era la *Activity*, pero a partir de la versión 3.0, estas actividades pueden estar compuestas de *Fragments,* así, por ejemplo podemos tener una aplicación de correos donde tengamos una lista con los correos y al pulsar muestre el detalle del correo. Si se hace un fragmento para la lista y otro para el detalle del correo, se podrían mostrar o una u otra en pantallas pequeñas (teléfono) y las dos a la vez en pantallas grandes (tabletas). Más adelante se conocerá como utilizar estos bloques incluso con versiones de Android anteriores gracias al paquete de compatibilidad.

Intents

Ya se ha visto que las aplicaciones se componen mediante los cinco componentes recién explicados y también sabemos ya que los *Content Providers* se acceden mediante la llamada *ContentResolver* pero ¿cómo se accede o se activan los otros componentes *Activity, Broadcast Intent Receive* y *Service* (dejaremos los *Fragment* para capítulos posteriores)?
Se realiza mediante mensajes asíncronos llamados *Intent* (intentos). Los *Intent* son lanzados constantemente a lo largo del sistema para notificar diversos eventos, desde la inserción de una tarjeta SD o que el dispositivo se está que-

dando sin batería hasta eventos específicos de alguna aplicación (petición de ejecutar una nueva actividad o servicio…). Las aplicaciones no sólo pueden responder a los *Intent*, sino que pueden crear los suyos propios para lanzar otra actividad o para avisar de que algo ha pasado (por ejemplo ha acabado de descargar un fichero y alguien, tiene que atender la petición para abrir el fichero descargado).

Los intentos son objetos de la clase *Intent* que contienen los datos del mensaje a transmitir. Dentro de los datos transmitidos en el *Intent* hay que diferenciar dos partes que son la acción a realizar y los datos sobre los que realizar la acción. La acción a realizar viene definida por cada aplicación y puede ser del tipo VIEW (ver), EDIT (editar) o algo semejante. Por ejemplo para editar un contacto se enviaría la acción EDIT y como dato para actuar se enviaría la URI (*Uniform Resource Identifier*, Identificador Uniforme de Recurso) del elemento a editar.

Para el caso de los *Broadcast Intent Receiver*, el *Intent* contiene el nombre de la acción a anunciar. Por ejemplo se podría anunciar que un SMS ha llegado al dispositivo.

Para cada componente hay una llamada distinta:

- **Activity:** para activar una *Activity* (que es la acción más común de las tres), lo que se hace es llamar a los métodos `context.startActivity()` o `activity.startActivityForResult()` pasando como parámetro un objeto *Intent*. La diferencia entre usar uno u otro estriba en si la actividad que lanza la petición espera respuesta de la actividad lanzada o no. En caso de que espere respuesta se debe usar `activity.startActivityForResult()` . Por ejemplo nos puede interesar seleccionar un contacto de los disponibles en el dispositivo, entonces se lanza la actividad de mostrar contactos y se espera que retorne el contacto seleccionado por el usuario. El resultado se devuelve dentro de otro *Intent* que es pasado como parámetro de la llamada del método `onActivityResult()` dentro de la *Activity* que inició el proceso. La *Activity* que responde a la petición puede ver el *Intent* que ha causado su activación mediante la llamada `getIntent()`. Cuando se explique el primer ejemplo de aplicación con varias pantallas comprenderá perfectamente el uso de las llamadas a distintas *Activity*.
- **Broadcast:** Para iniciar un *Broadcast Receiver Intent*, se debe llamar al método `context.sendBroadcast()`, `context.sendOrderedBroadcast()` o `context.sendStickyBroadcast()`. Android se encargará de llamar a los métodos `onRecevie()` de cada uno de los *BroadcastReceiver* que se encuentren registrados en el sistema.

- **Service:** Para iniciar un *Service* o para indicar nuevas instrucciones a uno que ya esté en activo se realiza una llamada al método `context.startService()` pasándole como parámetro un objeto *Intent*. Haciendo esto, Android llamara al método `onStart()` del servicio usando el *Intent* como parámetro. Del mismo modo, se puede usar la llamada `context.bindService()` (que opcionalmente puede lanzar el servicio en caso de que no esté actualmente en ejecución) para crear una conexión entre el servicio y el componente que lo está llamando. En este caso, el servicio recibe el *Intent* mediante la llamada de su método `onBind()`. Una vez creada la conexión, la *Activity* que la ha realizado es capaz de llamar a los métodos descritos por el servicio.

Filtrado

Se ha visto que un *Intent* es una petición al ecosistema Android de que realice una acción, pero de alguna manera se debe indicar que una *Activity* es capaz de atender cierta petición de acción. Aquí es donde entran en juego los filtros para los *Intent*, los llamados *IntentFilter*. Un *IntentFilter* es una descripción de lo que una *Activity* o un *BroadcastReceiver* son capaces de hacer, qué peticiones de *Intent* pueden manejar. Por ejemplo una *Activity* que permita la visualización de imágenes, deberá publicar su *IntentFilter* como capaz de gestionar la acción VIEW cuando lo que se trata de procesar es una imagen. Las actividades publican sus *IntentFilters* mediante el fichero de configuración `AndroidManifest.xml`.
Ya se ha visto que en la navegación entre pantallas se realiza mediante objetos *Intent*; y a groso modo podemos decir que cuando el programa llama a la función `startActivity(myIntent)`, el sistema revisa los filtros de todas las aplicaciones instaladas en el sistema y selecciona aquellas *Activity* que mejor se adapten al *Intent* pasado como parámetro. En caso de haber más de una *Activity* capaz de gestionar el *Intent*, se le muestra una pantalla de selección al usuario. La *Activity* que se haya seleccionado es entonces lanzada e informada con el *Intent* que se utilizó como parámetro en la primera llamada. Esta manera de trabajar mediante objetos *Intent* proporciona múltiples ventajas, entre ellas:

- Las *Activity* pueden reutilizar funcionalidad de otros aplicaciones simplemente haciendo una petición mediante el *Intent.*
- Las *Activity* pueden ser reemplazadas en cualquier momento por nuevas *Activity* que cumplan sus mismos *IntentFilter*.

- El hecho de que todo el sistema Android funcione mediante este mecanismo proporciona la posibilidad de cambiar las *Activity* estándar por nuevas *Activity* más completas o utilizar las estándar en aplicaciones propias.

Más adelante en el libro, volveremos a tratar los filtros más en profundidad con ejemplos.

Ciclo de vida

Los componentes de las aplicaciones tienen unos ciclos de vida que dependerán de la situación en la que se encuentre en cada momento la aplicación, desde que se crea y es capaz de responder a eventos hasta que se destruye y se liberan todos los recursos utilizados, la aplicación pasará por diferentes estados.
Veamos entonces qué estados son estos en los que pueden estar, qué transiciones se pueden dar entre ellos, a quienes afecta y como controlar estos cambios.
A lo largo de una ejecución normal de una aplicación, sus *Activity* las podemos encontrar en alguno de los siguientes cuatro estados:

- **Activa:** Es cuando el usuario ve la actividad y puede interactuar con ella desde la pantalla, o dicho de otro modo, cuando está la primera en la pila de ejecución.
- **Pausada:** Cuando la actividad ha pasado a segundo plano, pero aun está visible, es cuando otra actividad se coloca sobre la actividad que pasa a pausa, pero la nueva actividad no tapa del todo al actividad anterior (bien porque sea transparente o porque sea de menor tamaño su interfaz); la actividad pausada pierde el foco pero se ve parte de ella. En esta situación, la actividad pausada puede ser matada por el sistema si se necesita liberar recursos para la nueva actividad, no obstante el *Window Manager* mantiene el control sobre la actividad.
- **Parada:** Cuando la actividad pasa a segundo plano y además está totalmente tapada por la nueva actividad, es decir queda totalmente eclipsada por la nueva interfaz. En este caso el sistema también puede optar por matar esta actividad si se necesitan más recursos de memoria de los disponibles.
- **Destruida:** La actividad no está ya disponible, se han liberado todos sus recursos y en caso de ser llamada, necesitaría comenzar un nuevo ciclo de vida.

El esquema de los distintos estados y los métodos ejecutados cuando la actividad varia de estado están representados en la figura 3.3.

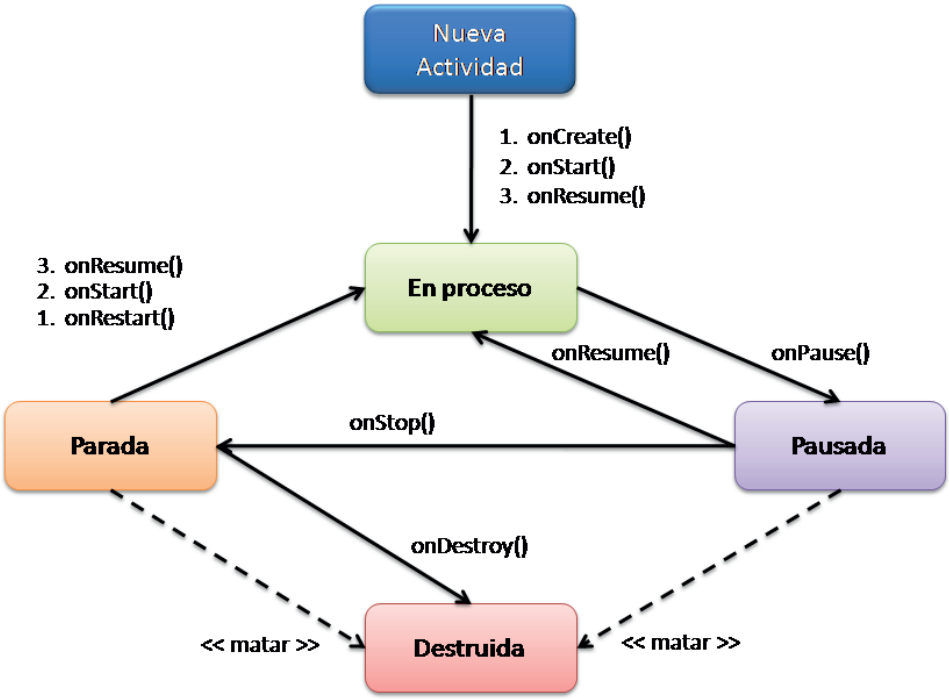

Figura 3.3. Estados de una Activity y paso entre ellos.

No importa si la actividad pasa a estar parada o pausada, dicha actividad mantiene toda la información referente al objeto que la representa cuando se encuentra en su nuevo estado, es decir al recuperar la actividad, las propiedades de la clase seguirán teniendo el valor que tenían cuando estaban activas. Se ha comentado que en el estado de pausa o parada el sistema puede pedir a la actividad que libere la memoria que está utilizando; esto se hace mediante la llamada al método `finish()` de la actividad. En caso de que se haya llamado a este método y se recupere la actividad posteriormente, ésta habrá perdido toda la información de la clase, teniéndose que inicializar desde el principio. Por ejemplo supongamos que tenemos una actividad con un botón y una etiqueta que muestra las veces que hemos pulsado dicho botón. El contador de las pulsaciones lo guardamos en una propiedad de la clase de la *Activity*. Ejecutamos y pulsamos cinco veces, en este momento tanto si se pausa como si se para, al recuperarse de nuevo esta actividad, se seguirá mos-

trando el valor cinco, pero si por lo que fuera se hubiera llamado al método `finish()` de la actividad; si se volviese a esta actividad, el valor de la variable en este caso sería cero. Por lo que si la información es importante, se deberá dotar de algún mecanismo de persistencia de los datos que se ejecute en caso de ser invocado el método `finish()`.
Los métodos que gestionan el ciclo de la actividad son los puntos de introducción de código para el control de la actividad.

Nota:

En todas las implementaciones de las llamadas a los métodos de gestión del ciclo de la actividad, debe estar la llamada a su correspondiente método de la superclase:

```
@Override
   public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
}
```

Los métodos para la gestión del ciclo de la *Activity* son:

- **onCreate():** Se llama nada más crear la *Activity*. Es donde normalmente se prepara la interfaz gráfica de la pantalla, se enlazan los datos con sus correspondientes métodos de visualización (listas, gráficos…). En caso de que exista, a esta función le llegará como parámetro un objeto de tipo *Bundle* con el estado anterior de esta *Activity* guardado por parejas clave-valor en el modo `nombre_variable=valor`. Para poder acceder a este *Bundle* la *Activity* tiene que haber guardado previamente su estado, de lo contrario el *Bundle* será nulo y no se podrá restablecer su estado. Más adelante veremos cómo guardar este estado. Tras esta función, el proceso sobre el que se ejecuta la *Activity* no puede ser destruido por el sistema. El siguiente método al que se llama es `onStart()`.
- **onRestart():** Se llama cuando una actividad se ha parado y vuelve a estar activa, justo antes de que comience de nuevo. Tras ella se llama al evento `onStart()` y no puede ser destruido su proceso ni durante ni tras su ejecución.
- **onStart():** Se ejecuta cuando la aplicación va a ser visible al usuario, justo antes de hacerlo. Dependiendo de si la aplicación pasa a segundo plano tapada por otra actividad que la oculte del todo, el siguiente método de ciclo de vida será `onStop()`, o en caso de que no la acabe de ocultar será `onResume()`.

- **onResume():** Se ejecuta antes de que el usuario pueda interactuar con la *Activity*. En este momento la *Activity* se encuentra en la parte superior de la pila de ejecución. El método que le sigue sería `onPause()`.
- **onPause():** Se llama cuando otra actividad va a ser llevada a primer plano, cuando va a dibujarse una nueva *Activity* sobre la *Activity* actual, o dicho en métodos de ciclo de vida, al método `onPause()` de una Activity se le llama cuando se llama al `onRestart()` de otra. En este momento debemos aprovechar para liberar todo aquello que consuma recursos (parar músicas, detener procesos…) o guardar datos de manera persistente. Con tal de no entorpecer la fluidez del sistema, este método debe contener acciones que no sean lentas en ejecución, puesto que hasta que este método no haya sido totalmente ejecutado, no se podrá ejecutar el método `onResume()` de la nueva actividad, por lo que no se mostrará en pantalla dando la sensación de lentitud en el sistema. Los métodos que pueden ejecutarse tras `onPause()` son `onResume()` en caso de que la nueva *Activity* no haya tapado totalmente a la anterior *Activity*, y ésta vuelva a primer plano, o `onStop()` en caso de que la nueva *Activity* tape por completo a la anterior, volviéndola invisible al usuario.
- **onStop()**: Cuando la *Activity* se hace invisible al usuario, entonces es cuando se ejecuta este método. El hecho de que se vuelva invisible una actividad se puede dar por dos circunstancias: que otra *Activity* ha pasado a primer plano y tape completamente a ésta o que la actividad haya sido destruida (se explica en el siguiente método). Le pueden seguir en el ciclo de vida dos métodos, dependiendo de si la *Activity* es destruida o no. En caso de ser destruida, el método que le sigue es `onDestroy()` y en caso de volver a ser visible (por ejemplo que desaparezca la Activity que la había hecho entrar en `onStop()` o que vuelva a ser llamada) entonces se ejecutaría `onRestart()`.
- **onDestroy():** Se llama antes de destruir una *Activity*. Durante la destrucción de la *Activity* se perderán todos los datos asociados a ella, de modo que si vuelve a ser llamada se ejecutará un nuevo ciclo de vida, por lo que es uno de los lugares para controlar la persistencia de datos. Se llama cuando alguien ejecuta sobre ella su método de finalización `finish()` o porque el sistema necesite más recursos y elimine la actividad para conseguirlos.

Pero ¿cómo determina el sistema que *Activity* se puede liberar en caso de necesitar recursos? El proceso seguido para cada tipo de actividad/proceso es el siguiente:

- Actividad de primer plano (*Foreground Activity*): Se trata de la *Activity* con la que el usuario interactúa, la que se encuentra en la pantalla. Se considera la actividad más importante y por consiguiente la que más preferencia tiene

sobre los recursos. Su proceso solamente se matará en última instancia, si utiliza más memoria de la que dispone el dispositivo. Si se da esta situación, la aplicación se mata antes de que el sistema se quede sin recursos y no sea capaz de responder al usuario.

- Actividad visible: Aquella actividad que es visible al usuario pero no es la *Foreground Activity*, es decir, tiene algo por encima de ella en pantalla, algo como un cuadro de diálogo que aunque no permite interactuar con ella sí permite verla. Se trata de una actividad preferencial y no se matará a no ser que la actividad que se muestra en primer plano necesite esos recursos para poder seguir funcionando.
- Actividad de fondo (*Background Activity*): Son las actividades que el usuario no ve y que han sido pausadas. Son actividades que no están marcadas como críticas y por lo tanto el sistema puede matarlas para recuperar recursos para las actividades que se encuentren en primer plano. Si el usuario reclama de nuevo la actividad que se ha matado y se vuelve a hacer visible en pantalla, se llama de nuevo al método `onCreate(Bundle)` de la *Activity*, y se le proporciona como parámetro el estado en el que estaba cuando se mató si anteriormente se guardó a través del método `onSaveInstanceState(Bundle)`, de modo que se pueda restaurar su estado como si simplemente se hubiera pausado.
- Proceso vacío: Son los procesos que no tienen asociados ninguna actividad ni dependen de ningún otro componente (*Service* o *BroadcastReceiver*). Son los primeros en ser sacrificados por el sistema si se necesitan recursos. Si un proceso es crítico, se debe realizar en el contexto de una *Activity*, un *BroadcastReceiver* o un *Service* para evitar que el sistema lo libere antes de tiempo (no es tan crítico en dispositivos con mucha memoria RAM, pero es muy normal en los primeros dispositivos que salieron a la venta). Si una *Activity* necesita hacer un proceso que dure mucho tiempo y que se ejecute independientemente de la actividad, la *Activity* debe lanzar un *Service* con el proceso en cuestión. Por ejemplo al seleccionar una descarga dentro de una actividad, mientras se descarga el objeto seleccionado debemos poder salir de la actividad y seguir trabajando con el dispositivo mientras se finaliza completamente de descargar, no importando si la *Activity* se cierra o se termina. Lanzando el proceso de descarga como un *Service* aseguramos que para el sistema este proceso tiene mayor preferencia que otros procesos no visibles.

  A todo esto hay que tener en cuenta que si se trabaja con elementos *Fragment*, éstos tienen su propio ciclo de vida. No veremos aquí este ciclo por no añadir más teoría; pero volveremos a ello en el capítulo 10.

Salvando el estado

Cuando el sistema decide que una actividad necesita más memoria de la que se encuentra disponible y decide matar alguna de las actividades que se encuentran pausadas, el usuario espera que si vuelve a la actividad matada la encuentre tal y como él la dejó, es decir se espera que el mecanismo de liberación de memoria sea transparente al usuario y que todas las actividades mantengan su estado a los ojos de la persona que utilice la aplicación. Para captar el estado de una *Activity* antes de que ésta sea matada se puede implementar el método `onSaveInstanceState()` de la propia actividad. Android llamará a este método antes de que la actividad pueda ser destruida, justo antes de llamar al método `onPause()`. En el método `onSaveInstanceState()` se encuentra disponible un objeto *Bundle* donde se pueden guardar los valores de las variables en parejas clave-valor. Cuando se reanude la *Activity* este *Bundle* estará de nuevo disponible en el método `onCreate()` y en un método que se ejecuta tras el método `onStart()` llamado `onRestoreInstanceState()`. Una vez recuperado el *Bundle* pasado por parámetro a estos métodos, se puede restaurar el estado que tenían las variables antes de que la actividad fuera matada. Hay que tener en cuenta que tanto el método `onSaveInstanceState()` como `onRestoreInstanceState()` no son parte de los métodos del ciclo de vida de una *Activity*, y como tales no son siempre llamados. Por ejemplo si un usuario pulsa la tecla de ir hacia atrás mientras está en una *Activity*, está diciendo que no quiere saber más de esa *Activity*, por lo que se destruirá por acción del usuario. En este caso no se llamará al evento `onSaveInstanceState()`. Esto nos abre una reflexión… si no siempre se llama a este método, no se debe utilizar para guardar datos persistentes en el tiempo, sino solamente datos para poder restaurar la *Activity*. Si se quieren guardar datos de modo persistente utilice el método `onPause()` para implementar la lógica y asegurarse de que se graban siempre, por ejemplo tras una *Activity* de configuración.

4 Entorno de programación para Android

En este capítulo aprenderá a:

- Manejar la estructura del proyecto Android.
- Crear configuraciones para distintos tipos de terminal.
- Realizar aplicaciones para múltiples idiomas.
- Controlar las opciones del archivo AndroidManifest.xml.
- Ejecutar programas en un dispositivo físico.
- Depurar aplicaciones.

En este capítulo se comenzará a trabajar con código y se aprovechará para entender qué elementos se pueden encontrar en un proyecto de aplicación Android y su utilidad. Hasta ahora se ha visto cómo ejecutar el emulador e instalar en él la aplicación, aquí se verá cómo depurarla y cómo instalarla en un dispositivo físico. Otro aspecto importante en el que se entrará en este capítulo es lo concerniente a la seguridad en las aplicaciones, qué opciones hay y cómo habilitarlas.

Estructura de una aplicación Android

En el segundo capítulo creamos la aplicación de ejemplo Hola Mundo. Vamos a retomarla para continuar explorando su contenido y realizar pequeñas modificaciones que nos ayuden a comprender todo lo que envuelve a la programación Android. Abra el entorno Android Studio y si no tiene cargado el proyecto *Hola Mundo* que se realizo en el capítulo 2, selecciónelo con tal de poder seguir trabajando sobre él.

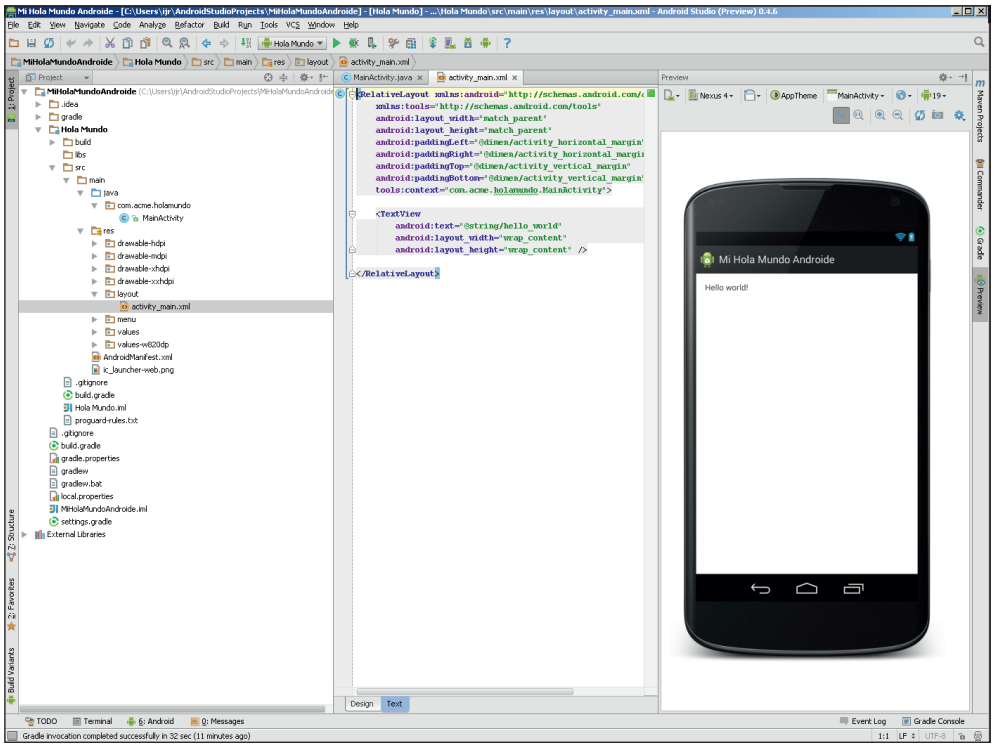

Figura 4.1. Proyecto Hola Mundo en Android Studio.

El entorno de programación Android Studio está basado en el entorno IntelliJ IDEA, y está conformado por distintos paneles que podemos mostrar, ocultar, cambiar de forma o posición... de modo que se adapten lo mejor posible en cada momento a nuestras necesidades. Antes de avanzar con la estructura del proyecto, vamos a ver un poco sobre la utilización y opciones de visualización del Android Studio.

Lo primero que podemos ver es en la parte izquierda un panel (en IntelliJ se denominan *tool window* o ventana herramienta, pero a lo largo del libro lo denominaremos panel al ser más descriptivo) con el árbol correspondiente al proyecto, más adelante veremos qué significa cada entrada del árbol. Si estamos acostumbrados a trabajar con Eclipse o si nos resulta excesiva la información de este árbol, podemos pulsar sobre el desplegable Project y seleccionar la opción Packages.

En la parte inferior izquierda de la pantalla podemos encontrar el botón  que nos da acceso a los distintos paneles disponibles. Funciona de dos maneras distintas dependiendo si se pulsa o si se posiciona solamente el ratón sobre él. En caso de posicionar el ratón sobre él aparecerá un desplegable con los paneles que podemos activar. El número de paneles dependerá del tipo archivo que tengamos cargado en el panel central, por ejemplo si tenemos un código Java, no nos aparecerá la posibilidad de activar el panel Preview ya que este está solo disponible si el archivo activo se trata de un XML con la definición de una pantalla.

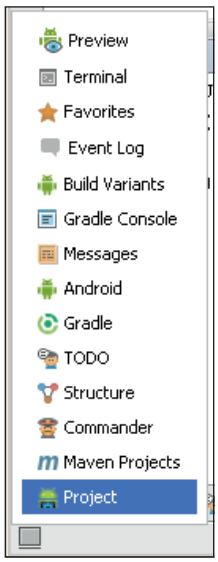

Figura 4.2. Diferentes paneles disponibles.

Y en caso de pulsarlo aparecerá o desaparecerá (el icono del botón cambia dependiendo del estado) un marco alrededor del entorno de programación con accesos a estos paneles.
Cada uno de los paneles tiene en su parte derecha al menos dos iconos que son  . El icono  nos permite ocultar el panel dando mayor visibilidad al resto de paneles visibles en ese momento. El icono  es en realidad un desplegable que nos dará distintas opciones dependiendo del tipo de panel que nos encontremos, entre otras podemos encontrar:

- Pinned Mode: Si lo tenemos marcado conseguimos que el panel se encuentre siempre visible. Esto es de especial utilidad si lo usamos en conjunción con la opción Floating Mode.
- Docked Mode: Al marcarlo hace que el panel se encaje con el resto de paneles visibles, sin sobreponerse a ningún otro. El panel siempre podemos redimensionarlo, pero si por ejemplo aumentamos el tamaño de un panel que tenga esta opción seleccionada, el resto de paneles disminuirán su tamaño de modo que se sigan viendo; en caso de no tener marcada la opción, el panel se incrementará por encima del resto de paneles tapando su visión.
- Floating Mode: Cuando está seleccionado, el panel se convierte en una pantalla aparte, separada de la principal. Esta opción es muy interesante cuando existe la posibilidad de trabajar con varios monitores.
- Split Mode: La pantalla principal está dividida en cinco zonas: superior, inferior, izquierda, derecha y central. La central está reservada para el código fuente y en el resto de las zonas podemos tener distintos paneles. Si se encuentran con el Split Mode activo, podemos tener visibles más de un panel por zona.

Si queremos mover una panel de una parte a otra de la pantalla principal, vale con pulsar con el botón derecho del ratón sobre la cabecera del panel y se desplegará un menú contextual con la opción de seleccionar la zona de la ventana principal donde se quiere mover el panel. Las opciones desplegadas son tres ya que el número de zonas donde podemos colocar el panel son cuatro y no aparece como opción la zona actual donde se encuentre el panel seleccionado.
Algunos paneles disponen a su vez de una serie de pestañas para intercambiar entre distinta información. Por ejemplo en el panel central donde mostraremos el código, se irá generando una nueva pestaña por cada archivo que abramos; si queremos modificar la posición de estas pestañas, podemos hacerlo mediante el menú Window>Editor Tabs o bien mediante el menú contextual que se obtiene pulsando con el botón derecho del ratón sobre las

mismas pestañas. Hay que tener en cuenta que algunas pestañas en modo edición tienen a su vez nuevas pestañas, como veremos por ejemplo cuando se diseñe una pantalla, que se dispone de una pestaña para la edición del código fuente y otro para el modo gráfico.

Ahora que ya sabemos un poco cómo movernos por el entorno, vamos a ir viendo otros aspectos más ligados a la programación Android.

Si nos fijamos de nuevo en el panel situado a la izquierda, podemos ver el esqueleto que se ha generado al crear la aplicación. El primer nodo raíz del proyecto es el nombre de la aplicación, que se informa en el campo Application Name cuando se crea la aplicación; este mismo nombre se utiliza para la creación del nombre del directorio en el que se almacenará el proyecto.

El primer nodo del árbol corresponde a la carpeta .idea, en ella se guardan archivos de configuración del proyecto relativos a como se debe comportar en el entorno de desarrollo, como por ejemplo directorios donde encontrar los archivos, posición y estado de los paneles....Normalmente no debemos tocar ningún archivo de esta carpeta.

La siguiente entrada corresponde a gradle. Gradle es una utilidad que veremos más en profundidad en el próximo capítulo y que sirve para ayudarnos en las tareas de construcción de los archivos instalables (entre otras cosas). En un principio tampoco debemos tocar nada de este directorio.

La siguiente entrada, es la que más nos interesa en este momento. En nuestro caso aparecerá con el nombre Hola Mundo que es el valor que se introdujo en el campo Module Name durante la creación del proyecto. Al desplegar el nodo encontraremos dos directorios y varios archivos. El primer directorio que podemos ver es el build y en él se guardan los archivos compilados en formato *dex*, recursos preparados para generar el archivo instalable, los propios archivos instalables, etc.; es decir todo lo que se genera a partir del código fuente y los propios instalables. No debemos tocar nada de este directorio puesto que cada vez que se genera el proyecto se sobrescribe los contenido en él.

Más interesante es la carpeta src/main, donde encontramos podemos encontrar:

- java: Es el directorio en el que alojaremos el código fuente java de la aplicación. Aquí accederemos para modificar y crear nuevos archivos de código Java. Su almacenamiento se realiza del mismo modo que en Java estándar: mediante paquetes.
- res: Se trata del directorio donde mantendremos la mayor parte de los recursos de nuestra aplicación, como por ejemplo imágenes, traducciones, diseños de pantalla, animaciones, estilos… Más adelante ya iremos viendo qué información se guarda en este directorio y cómo debe hacerse.

- AndroidManifest.xml: Es uno de los archivos más importante de la aplicación. En él se configuran aspectos tan importantes como el nombre de la aplicación, permisos que se le quieren dar (tales como acceso a internet, llamadas…), iconos a mostrar… Ya habrá tiempo para conocer sus entresijos en profundidad.

También dentro de la carpeta Hola Mundo encontramos una serie de archivos:

- .gitignore: Cuando varias personas trabajan en un mismo proyecto o si se quiere mantener el código fuente a salvo de borrados fortuitos, generar versiones... lo que se hace es mantener el código en un repositorio central. Existen varios tipos de repositorios CVS, SVN, Git...Android Studio viene preparado por defecto para trabajar con SNV, Mercurial y Git. Para Git, este es el fichero donde indicar qué elementos (ficheros y directorios) no se quieren guardar en el repositorio.
- build.gradle: En el capítulo siguiente se verá la utilidad Gradle; qué es, para qué sirve y cómo usarla. Por ahora nos conformaremos con saber que en este fichero se encuentra parte de la configuración de esta herramienta.
- Hola Mundo.iml: Se trata del fichero de configuración del módulo. Es un fichero propio del entorno IntelliJ IDEA.
- proguar-rules.txt: La herramienta ProGuard sirve para ofuscar, optimizar y reducir el código de la aplicación mediante la eliminación de código no utilizado y renombrando clases, atributos y métodos con nombres cortos e incomprensibles, de modo que se obtengan ficheros **.apk** (los ficheros instalables) de menor tamaño y más difíciles de realizar ingeniería inversa sobre ellos. Desde este archivo podemos indicar las reglas que debe seguir cuando se ejecute, que es durante la fase de construcción de los instalables.

Dentro del primer nodo raíz encontramos otra serie de archivos que por el momento no comentaremos.
En el segundo nodo raíz, denominado External libraries, nos encontramos con las librerías que utilizará nuestro proyecto y que vienen configuradas mediante Gradle. Al desplegarlo encontraremos unas entradas semejantes a:

- <Android API 19 Platform>: Se trata de las librerías referentes a la versión de Android que utilizaremos en nuestro desarrollo.
- <JDK>: Son las librerías Java necesarias para la programación y ejecución de aplicaciones en este lenguaje.
- support-v4-19.0.1: Es una librería de compatibilidad, que nos permitirá utilizar características de las nuevas versiones en dispositivos que tengan versiones anteriores de Android.

Recursos

Llamamos recursos a todos aquellos elementos que la aplicación hará uso y que no son código propiamente dicho. Dentro de la categoría de recursos entran elementos como iconos, música, colores, *layouts*, traducciones, y otros muchos elementos que puedan ser necesarios, incluso ficheros propios o crudos.

Si ha trabajado anteriormente con el entono de programación Eclipse, al generar el proyecto, además de la carpeta `res` podíamos ver otra carpeta (que se corresponde también con un directorio en el sistema de archivos): la carpeta `assets`. Dentro de `assets` se guardan los elementos de recursos que no queremos que sean modificados durante la generación del fichero **.apk**, es decir que en el archivo generado conservarán todo tal cual se almacena en la carpeta, incluido el nombre. En Android Studio esta carpeta no se crea por defecto, pero si la necesitáramos, la deberíamos crear dentro de la carpeta `main`. Si abrimos el fichero `Hola mundo.iml` encontraremos una entrada:

```
<option name="ASSETS_FOLDER_RELATIVE_PATH" value="/src/main/assets" />
```

Esta entrada indica donde espera el sistema encontrar este tipo de archivos.

En cuanto a la carpeta `res` tiene una estructura específica para mantener bien ordenados cada uno de los recursos dependiendo de su categoría, y muy importante… todos los ficheros contenidos en el directorio `res` están contenidos a su vez dentro de otro subdirectorio. A cada uno de los elementos contenidos en esta carpeta se le asignará automáticamente un identificador que estará contenido en la clase *R* que genera de forma autónoma el propio entorno de programación (y contenida en la carpeta `gen`), cosa que no ocurre con los ficheros contenidos en `assets`.

Dentro de la carpeta `res` se pueden encontrar los directorios con el contenido siguiente:

- animator: Archivos de tipo XML con información sobre animaciones de propiedades de una vista. Pueden encontrarse también en el directorio `anim`.
- anim: Archivos de tipo XML con información sobre animaciones en las que se realizan transformaciones sobre un mismo objeto.
- color: Archivos de tipo XML con definiciones de colores dependiendo del estado del objeto *View* sobre el que se aplique.
- drawable: Archivos gráficos de formato **.png**, **.jpg** o **.gif** y archivos de tipo XML definiendo animaciones, estilo, transiciones o estados en las que pueden entrar en juego varios objetos gráficos, formas geométricas, …

- layout: Archivos de tipo XML en los que se definen las disposiciones de los elementos de la interfaz en la pantalla.
- menu: Ficheros de tipo XML que definen menús de la aplicación. En este apartado se incluyen menús contextuales y submenús.
- raw: Archivos de cualquier tipo a los cuales se accede de manera cruda (*raw*). Son ficheros sin un formato concreto y así son tratados. Para abrirlos se debe utilizar la llamada `Resources.openRawResource(R.raw.nombre_fichero)` que devuelve un *InputStream* crudo. Como ya se ha explicado, otra manera de guardar este tipo de archivos se mediante la carpeta `assets`, donde además se podrán acceder de manera jerárquica como si de un sistema de archivos se tratara, mediante su nombre, pudiendo incluso crear nuevas subcarpetas.
- values: Ficheros XML que definen valores simples como enteros, constantes de texto o colores. Dentro de este directorio encontraremos que se definen múltiples recursos por fichero, no como en otros directorios donde cada fichero definía un recurso. Aquí, cada elemento dentro de los ficheros es lo que define el recurso, por ejemplo en el archivo de cadenas de texto `string.xml`, cada una de las entradas define una constante literal para poder ser usada en el programa.
- xml: Todo tipo de archivos con formato XML, desde archivos de configuración de *widgets*, hasta ficheros propios para una aplicación en concreto.

En muchas ocasiones será necesario proporcionar archivos de recurso dependiendo de una configuración dada o de la situación del dispositivo. Por ejemplo si el dispositivo está configurado en Francés, la aplicación debería mostrar los literales de texto en Francés, o si el dispositivo es de pantalla grande puede interesar mostrar un *layout* o unos iconos distintos a cuando la pantalla sea pequeña. Todo esto se consigue bien mediante programación detectando posiciones del terminal, configuración y especificaciones físicas y tras lo cual cargar los distintos archivos de recurso o bien a través de un método más automático: mediante el nombre de los recursos.
Para crear recursos dependientes de las propiedades del dispositivo, lo que se hace es crear dentro del directorio `res` nuevos directorios con la forma:
`<Nombre_directorio>-<modificador_configuración>`
Por ejemplo si abre el directorio `res` encontrará que existen (a partir de la versión 1.6, API 4) distintos directorios del tipo *drawable*:

- drawable-mdpi: Para los archivos gráficos para pantallas de densidad media (160dpi aprox.) que son las pantallas HVGA normales.

- drawable-hdpi: Para contener los archivos gráficos para pantallas de alta densidad (240 pdi aprox.).
- drawable-xhdpi: Para los archivos gráficos para pantallas de densidad muy alta(320dpi aprox.). Se introdujo en el API 8.
- drawable-xxhdpi: Para los archivos gráficos para pantallas de densidad muy muy alta(480dpi aprox.). El primer dispositivo en usar esta densidad fue el Nexus 10 aunque su densidad hacía que pudiera usar el xhdpi. Disponible a partir de API 16.

Dependiendo de la configuración del proyecto, es posible encontrar otros directorios como:

- drawable-nodpi: Para los archivos gráficos que no se quieren escalar dependiendo la densidad.
- drawable-ldpi: Para los archivos gráficos para pantallas de baja densidad (120dpi aprox.).
- drawable-tvdpi: Para los archivos gráficos para pantallas de densidad entre mdpi y hdpi, como puede suponer por el nombre, va dirigida a televisores(213dpi aprox.). Se introdujo en el API 13.

Los archivos que se vayan a utilizar como recursos, deben llamarse igual en todos los directorios, por ejemplo si se tiene un gráfico llamado "about.png", éste se debe guardar con la resolución y tamaño correspondiente con el nombre `"about.png"` en cada uno de los directorios, y cuando se acceda a él durante la programación mediante `R.drawable.about`, el sistema se encargará de mostrar el que más se ajuste al terminal y su configuración. Es posible informar valores por defecto para que en caso de que el sistema no encuentre un recurso que satisfaga la configuración correspondiente, use ese archivo por defecto. Imaginemos que se está haciendo una aplicación y se ha traducido para español, inglés y alemán; si alguien que tenga configurado su dispositivo en armenio tradicional quiere utilizar esta aplicación, deberá ser capaz, y es muy probable que no tengamos hecha la traducción. Lo que se hace en estos casos es tener un archivo de literales por defecto que Android al no encontrar los literales en armenio pueda utilizar como recurso.
Existen múltiples modificadores para la configuración de los recursos y todos ellos pueden combinarse para ajustar lo máximo posible a la configuración deseada. En el nombre de los directorios se pueden añadir tantos modificadores como se desee, simplemente separándolos con un guión, tal y como se ha hecho con los *drawable*, pero el orden en el que aparecen en el nombre del directorio viene definido por el orden en el que se encuentran en la siguiente tabla donde se describen cada uno de ellos:

Modificador	Valor	Descripción

MCC y MNC	`mcc214`
	`mcc214-mnc04`
	`mcc210-mnc003`
	`...`
		Es referido al *Mobile Country Code* (código móvil del país) y al *Mobile Network Code* (código de red móvil) por los que se identifica a cada proveedor de servicios móviles en cada país. Cada terminal se identifica de modo unívoco mediante lo que se denomina el IMSI *(International Mobile Subscriber Identity*, identificador internacional de subscripción móvil), éste se compone del MCC, el MNC y del MSIN (*Mobile Subscriber Identification Number*, número identificador del subscriptor móvil, que identifica al abonado). En este caso sólo interesa conocer el MCC y el MNC ya el MSIN es único por persona. Para saber los códigos MMC y MNC que corresponden a cada operadora en cada país, se puede consultar la Web `http://www.numberingplans.com/?page=plans&sub=imsinr`.
		Así, por ejemplo, Yoigo en España sería `mcc214-mnc04`

Idioma y región	`es`
	`en-rUS`
	`es-rES`
		El idioma viene definido por dos letras (siguiendo el estándar ISO 639-1) y de modo opcional se le puede añadir el código de la región (país) mediante otras dos letras (según ISO 3166-1-alpha-2) y separado por "-r" indicando la región.
		Los códigos ISO del idioma y del país pueden consultarse en la propia web de la organización de estándares: `http://www.iso.org/`
		El nombre del fichero no distingue entre mayúsculas y minúsculas y hay que tener en cuenta que la configuración del idioma puede cambiar mientras nuestra aplicación se encuentre en ejecución.

Tamaño de pantalla	`small`
	`normal`
	`large`
	`xlarge`
		`small`: Las pantallas basadas en el espacio disponible en una pantalla de baja densidad QVGA. Por ejemplo pantallas QVGA de baja densidad y pantallas VGA de alta densidad.

Modificador	Valor	Descripción
		`normal`: Las pantallas basadas en el espacio disponible en una pantalla tradicional HVGA de densidad media. Se considera normal si es inferior o igual a este tamaño. Por ejemplo WQVGA de baja densidad, HVGA de densidad media o WVGA de alta densidad. `large`: Las pantallas basadas en el espacio disponible en una pantalla VGA de densidad media. Por ejemplo pantallas de densidad media VGA y WVGA. `xlarge`: Las pantallas mucho mayores que una pantalla tradicional HVGA de densidad media. Son dispositivos que son demasiado grandes como para llevar en un bolsillo, lo que se denominan *tablets*. A diferencia de las anteriores, esta propiedad se hizo disponible en la API 9. En la Wikipedia se puede encontrar información sobre resoluciones y densidades y adjunta un gráfico muy explicativo: `http://en.wikipedia.org/wiki/Graphic_display_resolutions` Disponible a partir de la API 4.
Aspecto de la pantalla	`long` `notlong`	No hay que confundir con la orientación de la pantalla. Se refiere al aspecto de la pantalla en su posición natural. Se consideran `long` las de tipo WQVGA, WVGA, FWVGA y `notlong` las de tipo QVGA, HVGA, y VGA. Se introdujo en el API4.
Orientación de la pantalla	`port` `land`	No hay mucho que explicar. Cuando se encuentra en posición vertical será `port` *(portrait)* y en horizontal será `land` *(landscape).* Hay que tener en cuenta que la orientación puede cambiar muy a menudo durante la vida de una aplicación.

Modificador	Valor	Descripción
Modo Dock	`car` `desk`	Los dispositivos pueden colocarse en lo que se llaman *dock* o *dock station* (muelle), que son elementos en los cuales se encaja el terminal, bien para cargar la batería o bien para mantenerlo en una posición concreta que facilite su visión y manejo (o ambas cosas). Si el *dock* es el pensado para el coche, el valor será `car`, sino será `desk` (escritorio).
Modo noche	`night` `notnight`	Servirá para dotar a las aplicaciones de contrastes que ayuden a su visión durante la noche. Los valores a usar son `night` para los elementos a mostrar durante la noche y `notnight` para los del día. Añadida en la API 8.

Densidad de pantalla	`ldpi`
	`mdpi`
	`tvdpi`
	`hdpi`
	`xhdpi`
	`xxhdpi`
	`nodpi`
		Es uno de los puntos de configuración básicos en Android. Si durante la elección del recurso Android encuentra uno que encaje con la resolución de pantalla, entonces se utilizará éste.
		`ldpi`: Pantallas de baja densidad (120dpi).
		`mdpi`: Pantallas de densidad media (160dpi).
		`tvdpi`**:** Pantallas de televisión de 720p (213dpi aprox. A partir de API 13).
		`hdpi`: Pantallas de alta densidad (240dpi).
		`xhdpi`: Pantallas de densidad extra alta (320dpi. A partir de la API 8).
		`xxhdpi`: Pantallas de densidad extra extra alta (480dpi. A partir de API 16.
		`nodpi`: Se utiliza para bitmaps que no se quieren redimensionar dependiendo de la densidad de pantalla.
		Aunque más adelante se hace una explicación más profunda, comentar que dependiendo de la densidad de una pantalla, el gráfico se ve más pequeño cuanto mayor sea la densidad, es por eso por lo que se escalan, pero si se escala mucho pierde claridad y se debe utilizar entonces otro gráfico mayor.

Modificador	Valor	Descripción
		Un bitmap de 9x9 en una pantalla ldpi, se escalará a 18x18 en una pantalla hdpi con tal de mantener su aspecto, con la consiguiente pérdida de calidad. Es por eso que puede llegar a ser importante dar diferentes iconos para diferentes densidades de pantalla. Disponible desde la API 4.

Tipo de pantalla	`notouch`
	`stylus`
	`finger`
		Se refiere a si la pantalla es táctil o no y qué tipo de sensibilidad táctil tiene.
		`notouch`: El terminal no dispone de pantalla táctil (es poco frecuente, pero Android se pensó desde un principio para cualquier tipo de teléfono móvil).
		`stylus`: El dispositivo tiene una pantalla resistiva que normalmente se utilizan con un lápiz.
		`finger`: El dispositivo posee una pantalla capacitiva que puede ser utilizada con el dedo.

Disponibilidad de teclado	`Keysexposed`
	`keyshidden`
	`keyssoft`
		Se mostrarán unos u otros recursos dependiendo del estado y configuración de los teclados físicos y/o teclados software.
		`keysexposed`: Nos indica que el dispositivo tiene un teclado disponible. No importa que sea físico o no. Si el teclado software está presente, se utilizarán estos recursos aunque el teclado físico no lo esté. Si se tiene el teclado software desactivado, entonces estos recursos sólo se mostrarán cuando el teclado físico se encuentre disponible (hay terminales que pueden ocultar su teclado físico, como por ejemplo el primer Android que hubo, el G1).
		`keyshidden`: El dispositivo tiene teclado físico pero no está disponible y no tiene teclado software habilitado.
		`keyssoft`: El dispositivo tiene el teclado software habilitado y puede estar visible o no.

Modificador	Valor	Descripción
		Esta característica puede cambiar a lo largo de la vida de una aplicación (por ejemplo al destapar el teclado físico).

Método principal de introducción de texto	`nokeys`
	`qwerty`
	`12key`
		No se debe confundir con la característica anterior. Pues, en este caso se refiere a si físicamente existe o no, en la anterior era si estaban habilitados o disponibles. Esta característica no cambia a lo largo de la vida de una aplicación.
		`nokeys`: El dispositivo no posee teclado físico para la introducción de texto.
		`qwerty`: El dispositivo tiene un teclado de tipo querty que puede o no estar oculto.
		`12key`: El dispositivo tiene un teclado de 12 teclas (el de los móviles de toda la vida) que puede o no estar oculto.

Disponibilidad de los botones de navegación	`navexposed` `navhidden`	Se refiere a los dispositivos que tienen una tapa tras la que ocultar los botones de navegación. Es una característica que puede variar a lo largo de la vida de la aplicación. `navexposed`: Las teclas de navegación están disponibles al usuario. `Navhidden`: Las teclas de navegación no están disponibles.

Método de navegación que no sea la pantalla táctil	`nonav`
	`dpad`
	`trackball`
	`wheel`
		Es referido a los elementos físicos para navegar a lo largo de las pantallas sin necesidad de usar la pantalla táctil.
		`nonav`: No dispone de ningún otro elemento que no sea la pantalla táctil.
		`dpad`: El terminal dispone de una zona sensible al movimiento direccional, como los portátiles.
		`trackball`: El dispositivo tiene una pelotita o un lector óptico que detecta los movimientos en todas las direcciones.
		`wheel`: Solamente se dispone de una rueda semejante a la de los ratones de ordenador.

Modificador	Valor	Descripción

Versión del sistema (nivel de API)	`v2`
	`v3`
	`v9`
	`...`
		Se refiere al nivel mínimo de API disponible en el sistema. Por ejemplo v9 para el nivel de API 9 (Android 2.3 o superior) o v4 para nivel API 4 (Android 1.6).

Existen algunos otros modificadores aparecidos en el API 13 para delimitar los tamaños mínimos y disponibles de pantalla, destinados a aplicaciones complejas que se tengan que mostrar en múltiples dispositivos y que deban tener una interfaz de usuario muy pulida.
Para combinar varios de estos modificadores en los recursos ya se ha explicado que se deben informar en el nombre del directorio que contendrá el recurso pero se tienen que seguir unas normas:

- No se deben anidar los recursos con modificadores dentro de los recursos por defecto:

  Hacer un directorio `res/values/values`-es no es correcto. Lo correcto es hacerlo directamente sobre el directorio de recursos `res/values-es`.
- Se deben presentar los modificadores en el orden que se han presentado en la tabla:

  `drawable-car-port`: No es correcto.

  `drawable-port-car`: SI es correcto.
- Sólo se puede usar un modificador de un tipo por directorio.

  `values-es-res-fr-rca`: NO es correcto, se está intentando que un fichero de idioma valga tanto para español-España como para francés-Canadá. Lo correcto es hacer una entrada para cada uno.
- Los nombres pueden escribirse tanto en mayúsculas como en minúsculas, el procesador se encargará de ponerlos todos en minúsculas a la hora de analizarlos. El hecho de usar mayúsculas o minúsculas queda a la elección del programador aunque la recomendación son las minúsculas..

Ahora que ya sabemos para qué sirve cada directorio dentro de los recursos vamos a modificar alguno de sus valores y veremos cómo se traduce en la aplicación
Diríjase al directorio `src/mainres/values` dentro del módulo "Hola Mundo". Allí podrá ver un fichero llamado `strings.xml`. Para abrirlo vale con hacer doble click sobre él y se mostrará en la parte derecha de la pantalla.

El código será algo semejante a:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
   <string name="app_name">Mi Hola Mundo Androide</string>
   <string name="hello_world">Hello world!</string>
   <string name="action_settings">Settings</string>
</resources>
```

Que corresponden a las cadenas de texto que ya se han definido para la aplicación. Este tipo de archivos se utilizan para tener los literales separados de la capa de presentación, de modo que sea sencillo usarlos en el programa y traducirlos a los idiomas que sean necesarios mediante la técnica de hacer un directorio para cada idioma tal y como ya se ha visto.
Cada una de las entradas representa a una constante cadena. El identificador de la cadena viene referenciado por el atributo `name="nombre_cadena"` teniendo en cuenta que `nombre_cadena` no puede contener espacios. El valor de la cadena se encuentra dentro de las etiquetas `<string></string>`.
Cambie la entrada

```
<string name="hello_world">Hello world!</string>
```

Por

```
<string name="hello_world">¡Hola Mundo!</string>
```

Si ejecuta ahora el programa de nuevo, verá que la cadena mostrada ha cambiado. Esto es porque en algún lugar del código del proyecto, se especificaba que se usara el literal de texto llamado `"hello_world"`, para mostrarlo en pantalla, y como lo hemos variado, la aplicación ha variado.

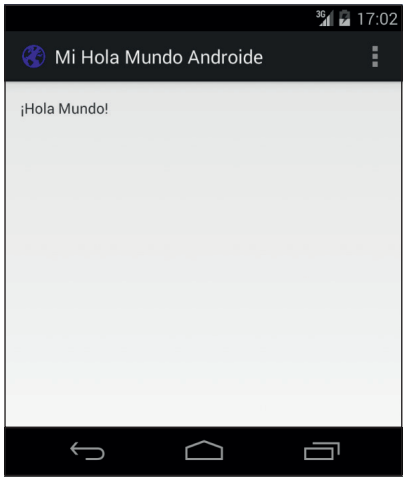

Figura 4.3. Nueva pantalla del proyecto "Hello World".

Es posible que a estas alturas le pique la curiosidad de dónde se especifica que esta cadena se muestre por pantalla. En el siguiente capítulo se verán con más profundidad los recursos de *layout*, pero vamos a introducirnos un poco en su utilidad. Los archivos de *layout* se utilizan para definir las interfaces gráficas que se mostrarán por pantalla. En ellos se especifica qué elementos se deben mostrar, en qué posición y con qué características.

Si despliega la carpeta `src/main/res/layout` dentro del proyecto verá un archivo llamado `activity_main.xml`, si el proyecto ha sido realizado con fragmentos, entonces además de este fichero verá otro llamado `fragment_main.xml`. Estos archivos corresponden al *layout* o disposición que se está mostrando en la pantalla de la aplicación. Si selecciona `fragment_main.xml` en caso de existir, o en su defecto `activity_main.xml`, verá que se abre en el panel central y que este editor incorpora dos pestañas en(por defecto en la parte inferior) llamadas Design y Text. La primera de ellas sirve para la generación de la interfaz de la pantalla mediante asistente gráfico y la segunda es para ver el código de los componentes de la pantalla.

Seleccionando la pestaña Text veremos el código fuente XML que será algo semejante a:

```
<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:paddingLeft="@dimen/activity_horizontal_margin"
   android:paddingRight="@dimen/activity_horizontal_margin"
   android:paddingTop="@dimen/activity_vertical_margin"
   android:paddingBottom="@dimen/activity_vertical_margin"
   tools:context="com.acme.holamundo.MainActivity$PlaceholderFragment">
   <TextView
      android:text="@string/hello_world"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content" />
</RelativeLayout>
```

Truco:

Si en los archivos de layout *en lugar de aparecer los textos como* `android:text="@string/hello_world"` *aparecen los literales del fichero* `strings.xml` *(por ejemplo* `android:text="¡Hola Mundo"`*) y sabemos que realmente están definidos en el fichero* `strings.xml`*, haciendo doble click sobre el literal, cambiará a su forma* `@string` *correspondiente.*

Esta es la manera en la que se definen las interfaces de usuario en Android. Si no está acostumbrado al uso de XML puede que al principio le resulte un poco complicado pero verá que en poco tiempo le resulta fácil modificar directamente el código XML. Este *layout* se compone tan solo de un elemento de tipo etiqueta de texto (*TextView*) donde se le indica que muestre en pantalla el valor de la cadena de texto definida bajo el identificador `"hello_world"` dentro del archivo `strings.xml`:

```
android:text="@string/hello_world "
```

El archivo AndroidManifest.xml

Manifest en castellano significa manifiesto, que según la RAE su definición es: "Descubierto, patente, claro. Escrito en que se hace pública declaración de doctrinas o propósitos de interés general". Este archivo, que se encuentra en la carpeta `src/main`, es donde se declara cómo es internamente la aplicación, qué actividades la componen, qué servicios existen...También se especifica la manera en la que alguna de sus piezas encaja en el ecosistema Android, por ejemplo qué actividades deben aparecer en el menú principal o qué servicios deben iniciarse bajo qué circunstancias.
El archivo es creado automáticamente cuando se crea un nuevo proyecto y se incluyen en él las opciones básicas para que una aplicación de una sola actividad y sin permisos especiales funcione sin problemas. Cuando genere aplicaciones Android se dará cuenta que tendrá que ir añadiendo opciones a este fichero con tal de poder navegar entre actividades, lanzar servicios o conectarse a internet por ejemplo. El número de entradas en el `AndroidManifest.xml` y su naturaleza dependen mucho de la aplicación que se esté codificando pero no creo que llegue a las 1953 líneas que posee la aplicación ApiDemos que ofrece Google.

Truco:

En IntelliJ IDEA y por lo tanto en Android Studio, algunos paneles incorporan lo que denominan fast search *o búsqueda rápida, que consiste en seleccionar el panel y al escribir directamente sobre él, el foco del panel va cambiando según la búsqueda de lo escrito. El panel de proyecto es de este tipo, por lo que para buscar el fichero* `AndroidManifest.xml` *simplemente debemos seleccionar el panel y comenzar a escribir el nombre del fichero para que lo localice.*

La estructura de un fichero `AndroidManifest.xml` completo tendría la forma:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest>
    <uses-permission />
    <permission />
    <permission-tree />
    <permission-group />
    <instrumentation />
    <uses-sdk />
    <uses-configuration />
    <uses-feature />
    <supports-screens />
    <application>
        <activity>
            <intent-filter>
                <action />
                <category />
                <data />
            </intent-filter>
            <meta-data />
        </activity>
        <activity-alias>
            <intent-filter></intent-filter>
            <meta-data />
        </activity-alias>
        <service>
            <intent-filter></intent-filter>
            <meta-data/>
        </service>
        <receiver>
            <intent-filter></intent-filter>
            <meta-data />
        </receiver>
        <provider>
            <grant-uri-permission />
            <meta-data />
        </provider>
        <uses-library />
    </application>
</manifest>
```

Pero normalmente encontraremos un grupo reducido de elementos:

- **uses-permission:** Indican los permisos que la aplicación necesita para funcionar de modo correcto, por ejemplo acceso a la cámara o escribir en los contactos.
- **permission:** Indican permisos que las actividades o servicios pueden requerir a otras aplicaciones para poder acceder a elementos de la aplicación como por ejemplo a los datos
- **instrumentation:** Sirve para indicar las clases que se deben invocar para cuestiones de monitorización y registro de logs.

- **uses-sdk:** Versión Android para la cual ha sido diseñada la aplicación. Tiene un atributo llamado `minSdkVersion` para informar el nivel mínimo de API Android y no provocar que una aplicación no funcione en un dispositivo solamente por no tenerlo actualizado.
- **supports-screens:** Por el momento la mayor parte de dispositivos son de tamaño muy parecido, pero estamos viendo como cada vez habrá más *tablets* en el mercado y es posible que se diseñe la aplicación sólo para pantallas de cierto tamaño. Mediante este elemento se puede especificar qué tamaño de pantallas están soportados por la aplicación y cuáles no. Por defecto están todos los tamaños soportados.
- **aplication:** Es el elemento más voluminoso de todo el documento. En él se encuentran definidas todas las *Activity*, los *Service*, los *ContentProvider* y los *BroadcastReceiver* que entran en juego junto con sus peculiaridades (filtros, iconos, etiquetas…).

Echemos un vistazo al `AndroidManifest.xml` que se ha creado automáticamente:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.acme.holamundo" >
    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name="com.acme.holamundo.MainActivity"
            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>
```

Como en todo documento XML se tiene un elemento raíz, que en este caso es el `<manifest>` donde se le ha definido un atributo aunque se le podrían definir muchos más:

- `package`: Es el paquete Java donde se encuentra la aplicación.

Dentro se encuentra el elemento `<application>` con los atributos:

- `android:allowBackup`: Indica si la aplicación se debe de guardar o no cuando se realiza una copia de seguridad del sistema. Por defecto es si, es decir que se guarda al realizar la copia de seguridad.

- `android:icon`: es el icono que se mostrará en el menú principal de Android (en el denominado *Launcher*), que en este caso está definido como el gráfico que se encuentra dentro del directorio `res/drawable` correspondiente y que se llama `ic_launcher`.
- `android:label`: es el texto que aparecerá bajo el icono en el *Launcher*. Lo más normal es que sea el nombre de la aplicación. En este caso así ha sido, se le ha asignado el texto que está definido en el archivo `strings.xml` bajo el identificador `app_name`.
- `android:theme`: es el tema que utilizará la aplicación como estilo. Podemos utilizar los temas por defecto de la aplicación o definir nuestros propios temas.

El elemento `<application>` tiene un elemento hijo de tipo `<activity>`. Este elemento representa a una clase de tipo *Activity* y por lo cual a una pantalla de la aplicación. Deberá haber tantas entradas de tipo `<activity>` como clases *Activity* se usen en la aplicación, como en este caso sólo hay una pantalla, solamente se encuentra una entrada, pero lo normal es tener varias (y olvidarse de declararlas en este archivo, con el consiguiente error en la ejecución de la aplicación). Los atributos que se encuentran definidos son:

- `android:name`: es el nombre de la clase que puede venir dado con todo el *path*, es decir con el paquete incluido como en este caso, o simplemente con el nombre de la clase y entonces se toma como paquete Java el que se definió como atributo en el elemento `<manifest>` que en nuestro caso aparecería como `".MainActivity"`. A la hora de informar este atributo, es posible usar el nombre de la clase solamente o junto con el paquete el funcionamiento es el mismo y queda a gusto del programador.
- `android:label`: es el texto que se mostrará al usuario cuando la actividad esté en pantalla. En caso de no informarse se utilizará el que se haya definido para el elemento `<application>`.

Dentro del elemento `<activity>`, encontramos un elemento `<intent-filter>` con dos entradas que condicionarán bajo qué circunstancias esta *Activity* debe ser mostrada. El elemento `<action>` indica que la *Activity* es el punto de entrada de una aplicación y por lo tanto no espera ningún dato como parámetro de entrada ni devolverá ningún resultado tras su ejecución. El elemento `<category>` indica que es la *Activity* inicial de una tarea y que debe ser mostrada en el *Launcher* junto al resto de aplicaciones. Más adelante veremos más sobre los filtros.

Advertencia:

Cuando su aplicación tenga más de una Activity *no debe olvidar declararlas todas en el* `AndroidManifest.xml` *de lo contrario no le dará error en tiempo de compilación pero sí en tiempo de ejecución, deteniéndose cuando intente acceder a alguna de las* Activity *no declaradas.*

Si miramos los ficheros `AndroidManifest.xml` de proyectos antiguos, generados con el programa Eclipse o incluso algunos creados con Android Studio es posible que veamos cuatro entradas (o alguna de ellas) muy importantes:

- `android:versionCode:` Representa la versión de código de la aplicación. Este atributo es un entero que sirve para tener control de si una versión es más nueva o no que otra. Simplemente si el número entero de este atributo es mayor, entonces se considera una versión más nueva.
- `android:versionName:` es la versión del código de la aplicación para los humanos. Es un atributo que puede tener cualquier tipo de texto para indicar la versión. No tiene porque estar en relación con el atributo `android:versionCode`, es decir la aplicación puede tener un `android:versionCode="7"` y el `android:versionName` puede ser "7", "7.0", "VII", "arccos (0,992)" o "mi versión inventada".
- `android:minSdkVersion:` es un entero representando el nivel mínimo de API Android que debe tener el dispositivo para ser capaz de ejecutar la aplicación. Es importante que se informe bien en el `Manifest.xml` o bien mediante otro mecanismo, ya que si no se informa toma por defecto en nivel 1, es decir, estaríamos indicando que sería compatible con todos los dispositivos (y puede que no sea así por alguna funcionalidad que tengamos).
- `android:targetSdkVersion:` es también un entero. Si no se informa toma el valor de `android:minSdkVersion`. Indica el nivel que se ha utilizado para probar la aplicación y el sistema no debe habilitar ningún sistema de compatibilidad hacia niveles de SDK superiores; pero eso no quiere decir que no se pueda instalar en niveles inferiores de SDK (para eso está la entrada anterior).

En el `AndroidManifest.xml` que acabamos de generar no se encuentran estas entradas porque Android Studio usa la herramienta Gradle para informarlas, aunque también se pueden informar desde aquí; ya iremos viendo todo esto en los siguientes capítulos.

Advertencia:

En los documentos del Market de Android, se recomienda encarecidamente el uso del atributo minSdkVersion*, con tal de ofrecer al usuario una mayor aproximación a los requerimientos mínimos de la aplicación en cuanto a nivel de API Android, ya que si se descarga la aplicación y no funciona la imagen de ésta para el usuario será pobre y puede puntuarla negativamente cuando solamente puede que sea que no tiene el sistema operativo Android lo suficientemente actualizado como para poder ejecutar la aplicación.*

Las opciones para configurar el `AndroidManifest.xml` son innumerables y en cada versión crecen, no obstante existe una documentación muy extensa que es muy recomendable tenerla a mano a modo de referencia sobre todo cuando se definen características que no se han usado anteriormente. La documentación se encuentra disponible en: `http://developer.android.com/guide/topics/manifest/.`

Ejecución de programas en dispositivo físico

Por el momento ya se ha visto como probar las aplicaciones desde el emulador que viene con el conjunto de herramientas de programación Android, pero en ocasiones es indispensable probar los desarrollos en dispositivos físicos; por ejemplo si no se tiene muy claro si una aplicación irá lenta o no en un dispositivo concreto porque su procesador sea antiguo o ver cómo reaccionan los acelerómetros del terminal… o simplemente para instalar la aplicación en el teléfono que llevamos siempre con nosotros.
La manera más sencilla de realizar esta tarea es habilitar la depuración en el dispositivo móvil, así al ejecutar una aplicación desde Android Studio podremos seleccionar si ejecutarla en el emulador o en el terminal físico. Diríjase a la aplicación de configuración en su dispositivo Android llamada Ajustes y dependiendo de la versión de Android que tenga, esta configuración se encuentra en uno de estos dos lugares:

- Seleccione Aplicaciones>Desarrollo. En esta pantalla existe una entrada llamada Depuración USB, selecciónela y vuelva a la pantalla principal del dispositivo.
- Seleccione Opciones de desarrollador. También en esta pantalla existe una entrada llamada Depuración USB, selecciónela y vuelva a la pantalla principal.

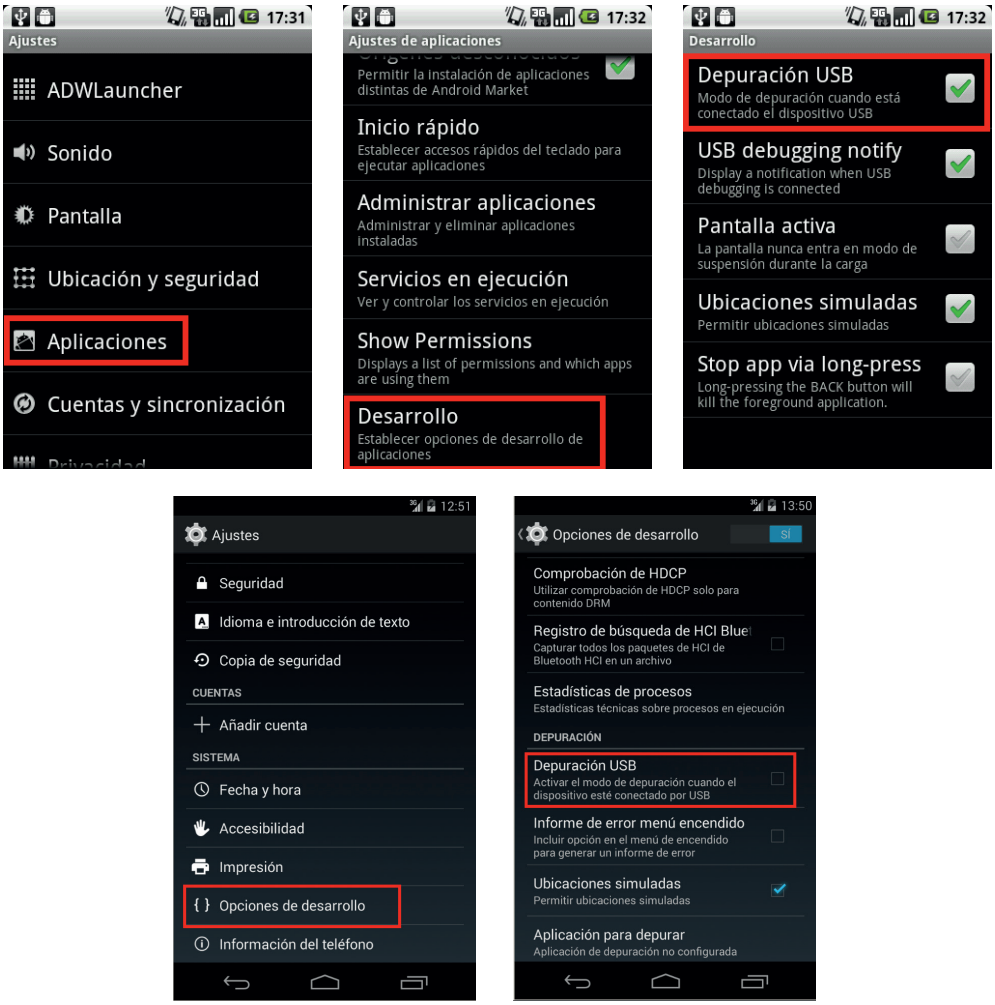

Figura 4.4. Activación de depuración en terminal físico.

Truco:

A partir de la versión 4.1, el Opciones de desarrollador *no está visible por defecto; para hacerlo visible, en la misma aplicación de* `Ajustes`*, seleccione* Acerca del dispositivo *y pulse repetidas veces de manera consecutiva sobre* Número de compilación*, cuando llegue a siete, se activará el menú de* Opciones de desarrollador*. Esto sólo hace falta hacerlo una vez; la entrada ya estará disponible para siempre.*

Enchufe el terminal a uno de los puertos USB mediante el cable correspondiente e inicie de nuevo la aplicación Hola Mundo. En este momento aparecerá una pantalla con una lista de dispositivos detectados (físicos y virtuales) y preguntando sobre cuál de los dispositivos se debe ejecutar la aplicación. En la figura 4.5 se puede ver la lista; en caso de que un dispositivo no cumpliera con el nivel mínimo de SDK, en la última columna aparecería marcado como no compatible.

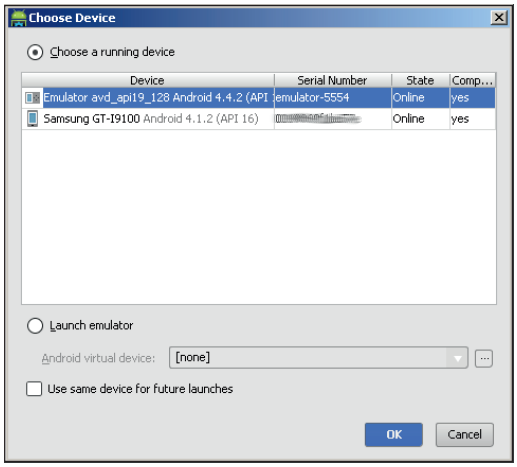

Figura 4.5. Pantalla de selección de dispositivo sobre el que ejecutar la aplicación.

Nota:

Tenga en cuenta que cuando se define el proyecto Android se selecciona un nivel mínimo de API. Si al ejecutar el proyecto, durante la instalación del ejecutable, el sistema Android detecta que su nivel de API es inferior al necesitado, NO ejecutará (ni instalará) el proyecto sobre el dispositivo.

Para distribuir la aplicación es posible exportarla de modo que el resultado sea un fichero de extensión .apk (*Android Package*, paquete Android) que puede ser enviado por correo, llevado en una memoria USB o subirlo a una web para descarga pública por ejemplo. El fichero *apk* debe ser accesible al sistema Android en el que se quiere instalar y tener permiso para hacerlo. Por defecto Android no deja que se instalen aplicaciones de orígenes distintos a su *market* (la tienda Android). Para habilitar el poder ejecutar aplicaciones de orígenes desconocidos se debe ir en el terminal al programa de Ajustes, una vez allí, nuevamente depende de la versión de Android en la que se encuentre:

- Seleccionar Aplicaciones y marcar la casilla correspondiente a Origenes Desconocidos.
- Seleccionar Seguridad y marcar la casilla correspondiente a Fuentes Desconocidas.

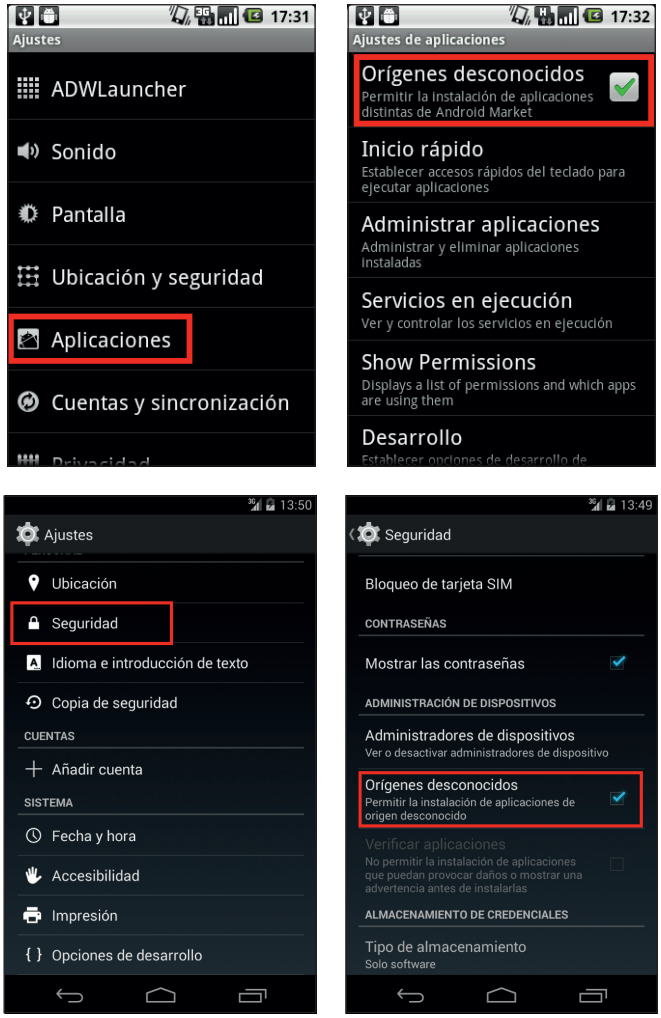

Figura 4.6. Habilitar instalación de aplicaciones de orígenes desconocidos.

A partir de ese momento ya se podría instalar la aplicación mediante su archivo *apk*, pero antes debemos generar el archivo instalable, cosa que veremos cuando conozcamos algo más sobre Gradle.

Depuración de programas

Tras ver como se ejecutan los programas en el emulador y en dispositivos físicos, es momento de saber cómo depurarlos en ambos tipos de dispositivo. Antes de comenzar a depurar, vamos a colocar un *breakpoint* (punto de ruptura) para que se detenga la ejecución. Al desplegar el árbol en el panel de proyecto situado en la parte izquierda de la pantalla, en el explorador, veremos una entrada `src/main/java`; dentro de esta carpeta se encontrará todo el código Java de la aplicación que vaya creando o que se haya creado de modo automático por parte de los asistentes de Android Studio. En esta carpeta del proyecto encontramos la clase *MainActivity* (que se corresponde con el archivo Java `MainActivity.java`) dentro del paquete `com.acme.holamundo` (también podemos buscarla haciendo uso del *fast search*, es decir escribiendo el nombre del archivo directamente en el panel de proyecto).
Esta clase es la *Activity* principal del programa, la que se llama nada más comenzar su ejecución. Para ver su contenido realizamos sobre el archivo `MainActivity.java` un doble click. Se abrirá entonces una nueva ventana con el código fuente, que será semejante a:

```
package com.acme.holamundo;

import android.support.v7.app.ActionBarActivity;
import android.os.Bundle;
import android.view.Menu;
import android.view.MenuItem;

public class MainActivity extends ActionBarActivity {

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

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {

        // Inflate the menu; this adds items to the action bar if it is
present.
        getMenuInflater().inflate(R.menu.main, menu);
        return true;
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        // Handle action bar item clicks here. The action bar will
        // automatically handle clicks on the Home/Up button, so long
        // as you specify a parent activity in AndroidManifest.xml.
```

```
        int id = item.getItemId();
        if (id == R.id.action_settings) {
            return true;
        }
        return super.onOptionsItemSelected(item);
    }
}
```

En las primeras líneas se define el paquete al que asignar la clase, los *imports* y la definición de la clase. Como es una actividad, la clase extiende la clase base *Activity*. Dentro de la clase *Activity* hay varios métodos que podemos sobrescribir y que ya se han descrito. El método `onCreate()` es siempre llamado a la hora de crear la actividad, tal y como ya vimos. Por ahora no vamos a entrar a analizar el código, simplemente colocaremos un *breakpoint* en la línea donde se llama a la clase superior.

```
super.onCreate(savedInstanceState);
```

Para poner el *breakpoint* se puede hacer mediante el menú de la aplicación Run>Toggle Line Breakpoint, combinación de teclas (control-F8) o haciendo click en el margen izquierdo del código, justo a la izquierda de la línea sobre la que se quiere poner el *breakpoint*. Al poner el *breakpoint* aparecerá un punto rojo en el margen y se marcará la línea.

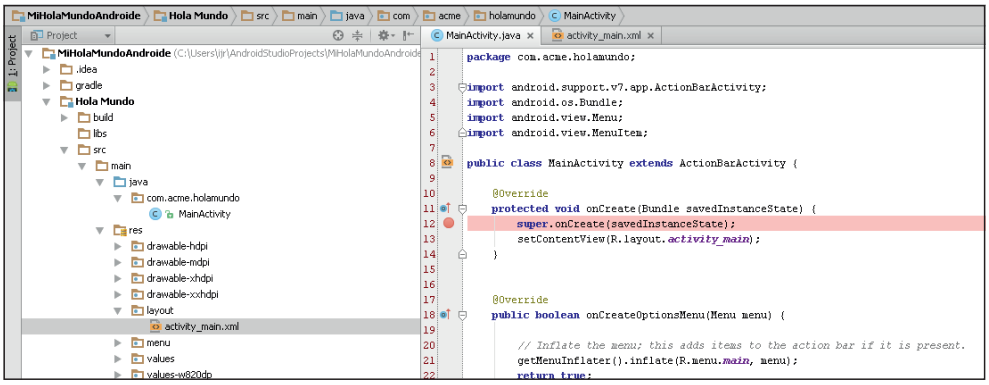

Figura 4.7. Colocación de un breakpoint.

Ahora en lugar de dar al botón de *play* como se hacía para la ejecución normal, pulse sobre la cucaracha que hay a la derecha del *play* . Al cabo de unos instantes, tras la instalación y puesta en marcha de la aplicación, la ejecución se detendrá y podremos depurar desde el panel de depuración o *debug* que si no lo tenemos a la vista, está disponible desde el icono situado abajo a la izquierda.

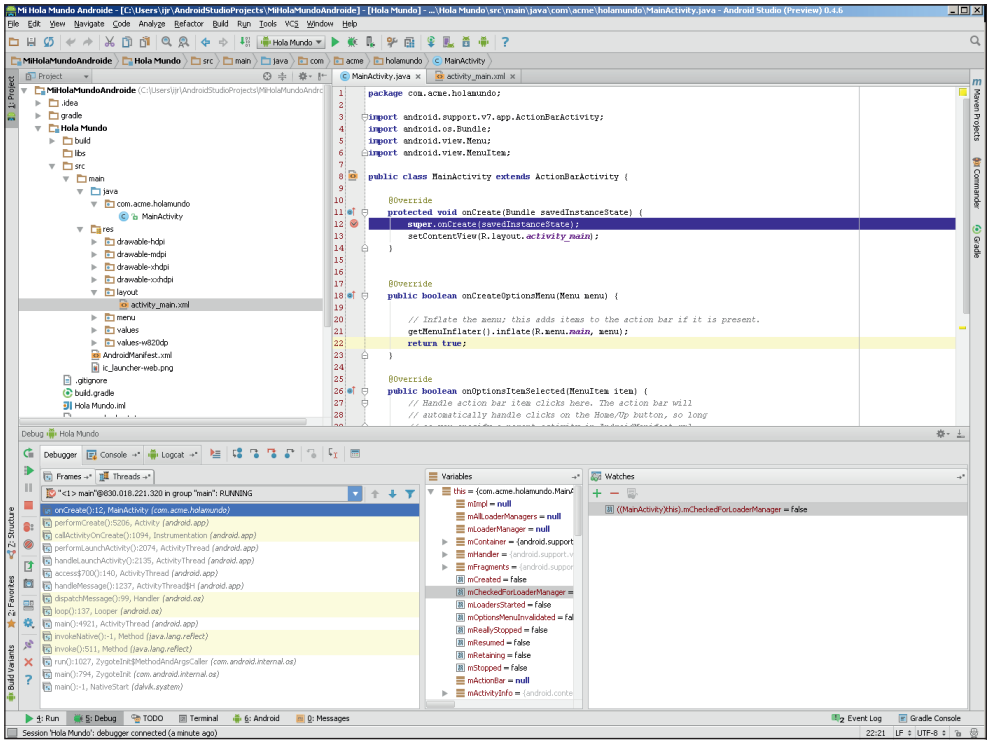

Figura 4.8. Android Studio mostrando el panel de depuración.

A partir de este momento es posible avanzar paso a paso en la ejecución del programa, ver el valor de las variables, inspeccionar objetos…
El modo de realizar la depuración sobre dispositivo físico o virtual es el mismo, solo que teniendo en cuenta que se debe mantener el dispositivo conectado al ordenador antes de pulsar sobre la cucaracha de modo que aparezca en la lista de selección de dispositivos.





5 Gradle

En este capítulo aprenderá a:

- Conocer la utilidad Gradle.
- Usar Gradle de modo genérico.
- Configurar tareas.

Las aplicaciones móviles han ido variando a lo largo del tiempo y han evolucionado desde pequeñas aplicaciones de unos cientos de líneas de código a aplicaciones complejas formadas por más de un ejecutable, entregadas en varios sabores (*lite*, de pago, a modo de prueba, *premium*…), archivos de datos externos e incluso suites de aplicaciones compuestas por varios programas. Ante este nuevo panorama, al programador se le abren nuevos problemas con los que debe lidiar, como por ejemplo como hacer que un mismo código valga para diferentes versiones de un mismo programa. Para ello en Android Studio (también sería posible usarlo en Eclipse) se presenta una herramienta de ayuda para el proceso de construcción de las aplicaciones llamada Gradle y que pasaremos a describir en este capítulo.

Nota:

Dado que Gradle se podría llegar a considerar un lenguaje de programación a su vez basado en Groovy (otro lenguaje de programación) y daría para escribir un libro completo se ha optado por desglosar su estudio en dos capítulos. Este primer capítulo se refiere a Gradle en general, para tener conocimiento básico de la herramienta. En el siguiente capítulo se verá Gradle aplicado en Android Studio.

La necesidad

Es posible que el lector crea que no es necesaria ninguna herramienta de automatización de procesos a la hora de crear el empaquetado de las aplicaciones y seguir como hasta ahora, trabajando sin ella. Esto es así si la aplicación es pequeña y no tiene muchas opciones o versiones, pero veamos algunos casos en los que la herramienta de automatización nos puede facilitar el trabajo. Imaginemos un juego de construir ciudades en el que queremos que en Navidad las casas aparezcan nevadas… podemos incluir los gráficos de las casas nevadas durante todo el año y mostrarlos sólo en navidad, o podemos hacer dos aplicaciones, una para todo el año y en navidad que se actualice con las imágenes e iconos cambiados para dar ese aspecto navideño, y por supuesto una vez pasada la navidad, volver a su aspecto normal. Si nos decantamos por la primera opción, aparte de tener un instalable de mayor tamaño, variando la fecha del dispositivo obtendríamos los gráficos de navidad en cualquier fecha del año y eso no es lo deseado. Muchos desarrolladores optan por la segunda opción, porque además se aprovecha para introducir modificaciones, corregir bugs y sin olvidar, que

cuando aparece una actualización de una aplicación, le recuerda al usuario su existencia y da a entender que se sigue trabajando en dicha aplicación. Esto implica copiar una serie de imágenes (con nieve o sin nieve dependiendo de cada época) en el directorio correspondiente del programa y realizar el empaquetado. Otro ejemplo sería si tenemos una aplicación que tenga una versión gratuita y otra de pago; para que mantengan el mismo aspecto, se debería realizar una sincronización de directorios de recursos o cambiar algunas clases dependiendo de la versión que queramos empaquetar. También podemos pensar en una suite de productos, como el Office, que es una aplicación compuesta de varios programas (Word, Powerpoint…), lo ideal sería poder trabajar en ellos de manera separada pero conjunta, es decir cada programa tiene su propio código porque deben hacer cosas distintas, pero tendrán mismos recursos gráficos o código común entre ellos (por ejemplo comprobación de licencias, iconos…) que podemos compartir a través de librerías. Es posible también que durante las pruebas de la aplicación accedamos a un servidor y que al realizarse la versión definitiva que entregaremos a los usuarios, los servidores deban ser cambiados para que se acceda a unos completamente distintos, o con claves distintas como sucede si se usa Google Maps y que antes de empaquetar el programa definitivo tengamos que reconfigurarlo. Como puede ver el lector y quizá no era consciente, son múltiples las situaciones en las que necesitamos hacer cambios en nuestros proyectos antes de crear el empaquetado de la aplicación.
Estas tareas de sincronización de directorios, modificación de contenidos, configuración de las aplicaciones dependiendo de si es la de prueba o no y otras muchas tareas pueden hacerse de modo manual o automatizado. Para la automatización de tareas en Java existen varias herramientas, quizá las dos más ampliamente extendidas y con varios años de trayectoria como sean Ant y Maven pero en lugar de usar estas, en nuestro caso usaremos Gradle.

Instalación

En este punto vamos a ver la manera en la que instalar Gradle en el sistema. Realmente no es necesario hacer una instalación adicional para trabajar con Gradle, ya que Android Studio viene con él incorporado, pero aprovecharemos para ver su instalación de modo aislado; dada la versatilidad de Gradle, puede que el lector vea interesante su uso para otras tareas distintas a la de la generación de aplicaciones Android (como por ejemplo copias de seguridad o como ayuda para otros lenguajes de programación).

Nota:

Si el lector prefiere no instalar Gradle de manera aislada, puede saltarse este punto e ir hasta el siguiente donde se explica cómo usar el incorporado en Android Studio para la realización de los ejemplos de este capítulo.

Lo primero que necesitamos para utilizar Gradle es una máquina virtual java 1.5 o superior; este requerimiento debería estar cumplido, ya que también era un requerimiento para la instalación de Android Studio. El fichero de instalación de Gradle, lo podemos encontrar en su web oficial: `http://www.gradle.org/`.
Una vez descargado el fichero zip, se debe descomprimir y en él encontraremos:

- Los binarios de Gradle.
- La guía de uso tanto en HTML como en PDF.
- La referencia DSL.
- Documentación API.
- Ejemplos.
- El código fuente a modo de referencia.

Advertencia:

No se debe construir Gradle desde el código disponible en este fichero, ya que su presencia es puramente académica. Si se desea construir su propio Gradle desde cero, se debe descargar desde los repositorios oficiales del código fuente.

Los binarios de Gradle se encuentran en directorio `bin` dentro del directorio donde hayamos extraído el fichero zip. Para hacer el ejecutable accesible desde cualquier directorio en la línea de comandos, añadiremos `GRADLE_HOME/bin` a la variable de sistema `PATH` (donde `GRADLE_HOME` es el directorio donde se ha extraído el fichero zip), del mismo modo que se hizo con el SDK de Android cuando se instaló el SDK Studio. Para probar si todo está bien configurado, podemos ejecutar desde la línea de comando la instrucción `gradle -v` y obtendremos una salida semejante a:

```
c:\gradle -v
------------------------------------------------------------
Gradle 1.10
------------------------------------------------------------
```

```
Build time:   2014-05-17 09:28:15 UTC
Build number: none
Revision:     36ced393628875ff15575fa03d16c1349ffe8bb6

Groovy:       1.8.6
Ant:          Apache Ant(TM) version 1.9.2 compiled on July 8 2013
Ivy:          2.2.0
JVM:          1.7.0_03 (Oracle Corporation 22.1-b02)
OS:           Windows 7 6.1 amd64
```

Acceso desde Android Studio

Aunque haya realizado una instalación aparte de Gradle, es interesante leerse este punto ya que cuando esté programando usando Android Studio, siempre le será más fácil acceder a las tareas del proyecto utilizando el Gradle empaquetado en el entorno de desarrollo.

La instalación de Android Studio incorpora Gradle en el directorio `<directorio de instalación>\sdk\tools\templates\gradle\wrapper`. En este directorio podremos encontrar el script para Windows (`gradlew.bat`) y para Unix (`gradlew`). Si el lector ha leído el punto anterior, se habrá fijado que el nombre del ejecutable en la versión de Android Studio no es `gradle`, sino `gradlew`; esto es porque realmente es un *wrapper* (un envoltorio); para las pruebas que haremos en este capítulo también nos es válido.

Este *wrapper*, en caso de ser preciso, descargará los archivos necesarios para su correcto funcionamiento y lo hará de modo transparente para nosotros. El *wrapper* lo encontraremos también en cada uno de los proyectos que realicemos con Gradle, permitiendo así poderlo distribuir junto con el proyecto y que otros desarrolladores trabajen con la misma versión de Gradle sin tener que hacer instalaciones externas.

Para facilitar el acceso, se recomienda añadir el directorio comentado anteriormente al PATH del sistema y poder usar el script de Gradle desde cualquier punto.

Para los ejemplos de este tema, podemos utilizar una línea de comando del sistema o bien usar la que tenemos a nuestra disposición en Android Studio; para acceder al panel de la línea de comando de Android Studio debemos pulsar sobre el icono situado abajo a la izquierda y seleccionar la entrada Terminal (véase figura 5.1).

A partir de este momento simplemente debemos recordar que aunque en los ejemplos se use `gradle` para invocar a Gradle, si se usa el de Android Studio, hay que usar `gradlew` para su invocación.

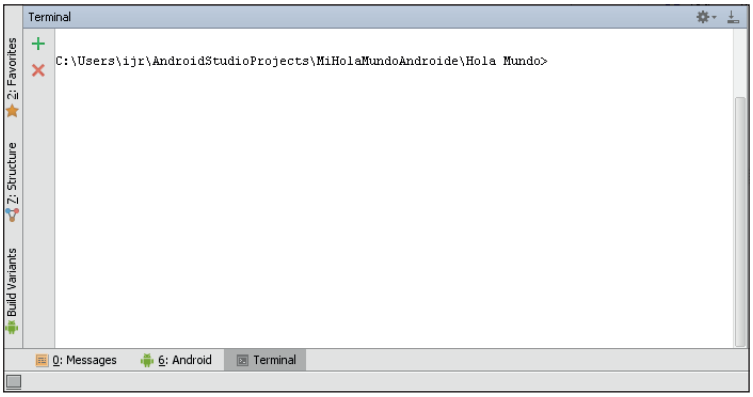

Figura 5.1. Panel de línea de comando

Tareas

Cuando se trabaja con Gradle, todo se basa principalmente en dos conceptos: Proyectos y Tareas. Los proyectos representan las piezas a compilar o construir, por ejemplo unas librerías y el ejecutable correspondiente y suele delimitarse por el árbol de directorios. Dentro de los proyectos tenemos las tareas, que son unidades de trabajo que efectúan una misión concreta: compilar los ficheros fuente, generar los documentos…

Para comenzar a realizar unas pequeñas pruebas con Gradle, no vamos a ser más originales que otros lenguajes de programación y vamos a realizar el Hola Mundo. Los scripts de Gradle deben ir en un fichero de texto llamado `build.gradle`, una vez creado, desde la línea de comando, iremos al directorio donde hayamos creado dicho fichero y lo ejecutaremos. En un directorio vacío de su ordenador cree un fichero `build.gradle` con el contenido:

```
task hola {
doLast {
println 'Hola mundo!'
}
}
```

Para ejecutarlo, tal y como hemos comentado anteriormente, desde la línea de comando y situados en el directorio en el que se encuentre el fichero `build.gradle`, se ejecuta la instrucción:

```
gradle hola
```

Tras la ejecución obtendremos una salida semejante a:

```
c:\tsgradle>gradle hola
:hola
```

```
Hola mundo!

BUILD SUCCESSFUL

Total time: 2.399 secs
```

Advertencia:

Si el sistema nos da un error indicando que no se encuentra el ejecutable, es posible que no hayamos incluido la ruta de Gradle en el `PATH` *del sistema (o lo hayamos incluido de modo incorrecto).*

En la salida obtenida podemos ver la frase de saludo que se había definido y una serie de datos respecto a la ejecución; para evitar que se muestre este log de ejecución, podemos utilizar la llamada `gradle -q nombre_tarea`. Existen varias maneras de representar una tarea, por ejemplo otro modo de escribir el script anterior de una forma más compacta (y más utilizada) sería:

```
task hola << {
println 'Hola mundo!'
}
```

El operador << es un alias para la instrucción `doLast` que indica que el código de ese bloque se debe ejecutar en última instancia. Del mismo modo que existe un `doLast` en la tarea, también tenemos la instrucción `doFirst`, que se ejecutará en primera instancia. En caso de que la tarea esté varias veces definida, se ejecutará tantas veces como se encuentre pero teniendo en cuenta tanto el orden de su definición como sus instrucciones `doLast` y `doFirst`, por ejemplo:

```
task hola << {
   println 'Hola tercero'
}

hola.doFirst  {
   println 'Hola segundo'
}

hola {
   doFirst  {
      println 'Hola primero'
   }
}

hola {
   doLast{
      println "Hola cuarto"
   }
}

hola.doLast  {
   println 'Hola quinto'
}
```

Daría como salida:

```
c:\tsgradle>gradle hola -q
Hola primero
Hola segundo
Hola tercero
Hola cuarto
Hola quinto
```

Podemos ver como la salida es mostrada en orden, esto se debe a que Gradle ha ordenado la ejecución teniendo en cuenta los `doFirst` y `doLast`. Cabe destacar que cada tarea puede ser accedida como un objeto del script y así acceder a sus propiedades, es por ejemplo el caso de `hola.doLast`.

Además de las propiedades ya disponibles, podemos crear nuevas propiedades para cubrir nuestras necesidades, por ejemplo:

```
task myTask {
   ext.newProperty = "El valor deseado"
}

task printTask  << {
   println myTask.newProperty
}
```

El prefijo `ext` que tiene la propiedad, se utiliza para generar la propiedad en el dominio "extra", que no deja de ser a su vez una propiedad dentro de la tarea, es decir tendremos una propiedad dentro de otra. Anteriormente se podía no utilizar este prefijo, pero actualmente si no se hace, dará un *warning* o no funcionará y es que a partir de la versión 2.0 (aún no disponible para uso general) se tiene previsto retirar las propiedades dinámicas y en las versiones actuales ya comienza a avisarse este hecho mediante un mensaje de *warning*.

Además de propiedades, podemos crear métodos propios donde encapsular código para poder volver a utilizarlo en distintas tareas, por ejemplo:

```
task hola << {
   println getMessage('Saludar')
}

task adios << {
   println getMessage('Despedirse')
}

String getMessage(String mensaje) {
   return 'Se ha seleccionado: ' + mensaje
}
```

Al ejecutarlo se obtendría

```
C:\tsgradle>gradle -q hola
Se ha seleccionado: Saludar
```

En ocasiones nos interesa enlazar varias tareas en cierto orden; Gradle permite generar dependencias entre tareas; en el siguiente ejemplo se pueden ver un par de tareas, de modo que una de ellas dependa de la otra, concretamente si se ejecuta la tarea `adios` se ejecutará también la tarea `hola`:

```
task hola << {
   println 'Hola mundo!'
}

task adios(dependsOn: hola) << {
   println "Adios, me marcho."
}
```

Ejecución de scripts Gradle

La ejecución de los scripts de Gradle constan de dos fases:

- Evaluación: en esta fase se realiza un análisis de los ficheros fuente del script de Gradle y se realiza un árbol de ejecución, determinando las tareas que deben ser ejecutadas, las dependencias entre ellas y el orden en el que debe realizarse la ejecución.
- Actuación: Es la fase de ejecución propiamente dicha, donde se efectúan las acciones definidas en las tareas y se realizarán en el orden establecido por el árbol de tareas que se haya determinado en la fase anterior.

El hecho de trabajar con dos fases en lugar de simplemente ejecutar el fichero de script, da mucha flexibilidad, por ejemplo permite que podamos definir las tareas en cualquier orden, no importa que la tarea dependiente esté definida antes o después de la tarea de la cual depende, incluso puede estar en otros ficheros. Esto es muy importante sobre todo para poder trabajar con construcciones multiproyecto.
El árbol de tareas a ejecutar que se genera en la fase de evaluación, puede ser consultado y tenido en cuenta para operar de un modo u otro. Supongamos el siguiente script:

```
def fileName

task build << {
   println "Generando fichero $fileName"
}

task publish(dependsOn: 'build') << {
   println "Publicando documentos."
}

gradle.taskGraph.whenReady {
```

```
taskGraph ->
   if (taskGraph.hasTask(publish)) {
      fileName = getVersion() + ".war"
   } else {
      fileName = "test.war"
   }
}

String getVersion(){
   "file_${buildTime()}"
}

def buildTime() {
    def date = new Date()
    def formattedDate = date.format('yyyyMMddHHmmss')
    return formattedDate
}
```

En él podemos ver dos tareas, una para generar el programa (tarea `build`) y otra para publicarlo (tarea `publish`); la tarea `publish` es dependiente de la `build`. En la tarea `build` mostraremos en pantalla el nombre del fichero a generar, que será distinto si simplemente se quiere generar o si bien se quiere generar y publicar (se ejecutarían ambas tareas).
Al principio del script podemos ver cómo se define una variable global mediante:

```
def fileName
```

También es posible que nos llame la atención la salida por pantalla

```
println "Generando fichero $fileName"
```

Cuando una cadena de texto se define con comillas dobles, si dentro de ella se encuentra el símbolo $, entonces se procesará como código lo que continúe a ese símbolo; en este caso es el nombre de una variable pero podría ser una función. En el script que estamos viendo se sustituirá `$fileName` por el valor de la variable `fileName`. Cómo hemos dicho, también es posible utilizar funciones incrustadas en las cadenas de texto de modo que genere la cadena usando el valor devuelto por la función; por ejemplo:

```
"file_${buildTime()}"
```

Que sería semejante a llamar a la función, asignar su resultado a una variable y este resultado concatenarlo con la cadena deseada:

```
theTime = buildTime()
"file_ " + theTime
```

Mediante la llamada a `gradle.taskGraph.whenReady`, le estamos diciendo a Gradle, que cuando termine de generar la lista y orden (el gráfico o árbol de ejecución) de tareas que debe procesar, entonces ejecute las instrucciones definidas entre llaves, en este caso, lo encerrado entre llaves es determinar si

existe la tarea `publish` en el gráfico y si existe obtener el nombre del fichero para publicación, y en caso de no existir, usar como nombre `test.war`. Otra particularidad que podemos ver en el script es que el método `getVersion()` no dispone de sentencia `return` y es que en Gradle no es obligatorio su uso, en caso de no ponerse, se devuelve el valor de la última sentencia ejecutada, que en este caso es la cadena de texto; de todas formas recomendamos su uso ya que queda mucho más claro el código. Dentro del método `buildTime()` usamos dos variables `date` y `formattedDate`, que tienen la palabra `def` precediéndolas; es posible que en algunos scripts veamos que se definen las variables sin usar la palabra `def`, esto es porque anteriormente tampoco era obligatorio su uso para definirlas (aunque veía muy bien por claridad del código). Al igual que con el caso de la variable global `fileName` y como ya hemos visto anteriormente, si se elimina el `def`, dependiendo de la versión de Gradle que usemos, funcionará perfectamente, dará un *warning* o no funcionará.

Advertencia:

A partir de la versión 2.0 de Gradle será obligatorio el uso de `def` *para la definición de las propiedades.*

Teniendo en cuenta lo anterior, si se usan versiones antiguas de Gradle, podríamos encontrarnos la función `buildTime()` como:

```
def buildTime() {
    date = new Date()
    formattedDate = date.format('yyyyMMddHHmmss')
}
```

Aunque actualmente nos diera un aviso durante la ejecución. E incluso podríamos ir un poco más allá y encontrarla como:

```
def buildTime() {
    new Date().format('yyyyMMddHHmmss')
}
```

Aunque por claridad de código, compatibilidad y teniendo en cuenta los cambios que vendrán en la versión 2.0, es recomendable usar siempre `def` y `return`. Cuando generemos scripts en Gradle, podemos utilizar clases Java a nuestra conveniencia, por ejemplo en el siguiente script trabajaremos con instanciando una clase `java.util.Date` y una `java.lang.String`:

```
task showTime << {
def pos = new java.util.Date()

   println  pos.toString()
   println  '-------------'
```

```
   String[] result = pos.toString().split("\\s");
      for (int x=0; x<result.length; x++){
               println result[x];
   }
}
```

Durante la ejecución, se creará una instancia de la clase `Date()`, se muestra en formato texto la fecha y tras ello se imprime en cada línea cada uno de los elementos de la fecha, esto se consigue mediante el método `split()` de la clase `String()` e iterando sobre el *array* resultante. La salida será:

```
c:\tsgradle >gradle sT
:showTime
Fri Feb 14 16:49:01 CET 2014
-------------
Fri
Feb
14
16:49:01
CET
2014
```

Más tareas… ¡es la guerra!

Utilizando como título una frase de Groucho Marx un poco modificada, en este apartado veremos diferentes conceptos ligados a la ejecución de tareas que siendo difíciles de clasificar, son importantes de conocer porque nos ayudarán a la hora de trabajar con Gradle.

Hasta ahora hemos ejecutado Gradle desde la línea de comando con el nombre de una tarea a ejecutar como parámetro; en caso de que queramos ejecutar varias tareas, es posible informar todas ellas como parámetros, simplemente anotando una tras otra teniendo en cuenta su orden, por ejemplo:

```
gradle tarea1 tarea2 tarea3
```

Para jugar un poco con las tareas y ver los siguientes conceptos vamos a utilizar un script donde se tienen dos tareas que dependen de una misma tarea base.

```
task tarea1(dependsOn: 'tareaBase') << {
      println "Soy la tarea 1"
}
task tarea2(dependsOn: 'tareaBase') << {
   println "Soy la tarea 2"
}
task tareaBase <<{
   println "Soy la tarea base"
}
```

Si acabamos de modificar el archivo es posible que nos acordemos del nombre de todas las tareas definidas; pero el tiempo (y la edad) se encargará de que las olvidemos. Para ver las tareas disponibles en un script se utiliza el parámetro `tasks` en la llamada:

```
c:\tsgradle >gradle tasks
:tasks

------------------------------------------------------------
All tasks runnable from root project
------------------------------------------------------------

Build Setup tasks
-----------------
init - Initializes a new Gradle build. [incubating]
wrapper - Generates Gradle wrapper files. [incubating]

Help tasks
----------
dependencies - Displays all dependencies declared in root project 'Temp'.
dependencyInsight - Displays the insight into a specific dependency in root
proj
ect 'Temp'.
help - Displays a help message
projects - Displays the sub-projects of root project 'Temp'.
properties - Displays the properties of root project 'Temp'.
tasks - Displays the tasks runnable from root project 'Temp'.

Other tasks
-----------
tarea1
tarea2

To see all tasks and more detail, run with --all.

BUILD SUCCESSFUL

Total time: 3.9 secs
```

Al utilizar este parámetro, se nos muestra la lista de tareas generales y las que no son dependidas de otras, es decir las tareas raíz; en nuestro caso, nótese que no aparece la tarea `tareaBase`. En caso de querer una lista más detallada podemos añadir el modificador `–all`, y en este caso se mostrarán tanto las tareas raíz como las dependientes:

```
c:\tsgradle>gradle tasks --all
[…..]
Other tasks
-----------
tarea1
    tareaBase
tarea2
    tareaBase

BUILD SUCCESSFUL
```

Se puede ver que en el listado, las tareas se engloban en grupos y vienen acompañadas de una pequeña descripción; esto facilita mucho las cosas cuando comienzan a tener muchas tareas o si hace mucho que se realizó el script. Para añadir el grupo y la descripción de la tarea, se haría mediante las propiedades de la tarea `group` y `description` respectivamente. Por ejemplo si añadimos al script:

```
tarea2 {
   description = 'Esta tarea imprime un mensaje en la pantalla'
   group = 'Relleno'
}
```

La nueva salida sería:

```
[…]
Relleno tasks
-------------
tarea2 - Esta tarea imprime un mensaje en la pantalla

Other tasks
-----------
tarea1
```

Si ejecutáramos de manera conjunta las tareas `tarea1` y `tarea2`, observaremos en la salida que Gradle es lo suficientemente "inteligente" como para ejecutar la tarea base una sola vez, ya que sería redundante volver a realizar la `tareaBase` cuando se ejecute la tarea2 puesto que las acciones a realizar en esta tarea ya se habrían efectuado y sería una pérdida de tiempo y recursos volverlas a realizar. Nuevamente esta "inteligencia" se debe al árbol de ejecución de tareas que se genera en la fase de análisis durante la ejecución del script.

Para llamar a las tareas hasta ahora estamos utilizando su nombre completo, pero es posible utilizar abreviaturas del mismo; veamos cómo. Cuando se indica el nombre de una tarea, Gradle intenta encontrar dicha tarea, en caso de que encontrarla la añade al árbol (lista) de ejecución de tareas y en caso de no encontrarla mira a ver si es la abreviatura de alguna tarea. Si existe una tarea que comience como tarea indicada será utilizada para añadirla al árbol de ejecución, si no buscará teniendo en cuenta abreviaturas de las tareas rompiendo el nombre de éstas en las mayúsculas o números. Sé que es un poco confusa la descripción así que veremos algunos ejemplos. En el caso del script anterior por ejemplo se podría llamar a la `tarea2` como `t2`, `ta2`, `tar2`…, lo mismo sucede para la `tarea1`; para la `tareaBase` podríamos usar abreviaturas como `tareaB`, `tB`, `tabas`… y su por ejemplo tuviéramos una tarea llamada `tareaConNombreLargoDelamuerte` podríamos usar `tC` o `tCNLD` por ejemplo. Una cosa se debe tener en cuenta… debe ser única la tarea encontrada; en el caso del script anterior, si escribi-

mos como nombre de la tarea `tare`, podría ser cualquiera de las tres tareas... por lo que obtendremos un error y no se ejecutará ninguna. Con esto quiero decir que no se debe tener miedo en poner nombres largos a las tareas si ayudan a identificarlas, ya que podemos usar sus abreviaturas durante su invocación.
Si existen unas tareas que son llamadas habitualmente, las podemos definir como tareas por defecto y al llamar a Gradle sin nombres de tareas a ejecutar como parámetros, en tal caso, ejecutará las definidas por defecto.
Las tareas por defecto se definen mediante la instrucción `defaultTasks`:

```
defaultTasks 'clean', 'build'

task clean << {
    println "Eliminando ficheros temporales"
}

task build << {
    println "Compilando fuentes..."
}

task mrproper << {
    println "Eliminando todos los ficheros"
}
```

Un poco de orden

Llegado cierto momento, es posible que los scripts sean lo suficientemente largos como para querer dividirlos en varios ficheros o puede que lo queramos hacer simplemente por cuestiones de reutilización u orden, el caso es que Gradle permite hacerlo. Para ello hay que utilizar la sentencia:

```
apply from: 'fichero_externo'
```

Por supuesto se pueden utilizar tantos ficheros externos como se desee. Como ejemplo vamos crear un fichero `build.gradle` en el directorio donde estamos trabajando, con el contenido:

```
apply from: 'common/external.gradle'

task main(dependsOn: 'external') << {
   println "Publicando documentos."
}
```

En él vemos que se define una tarea que depende de otra llamada `external` y que se referencia a un fichero llamado `external.gradle` dentro del directorio `common`. Vamos a crear ahora el directorio `common` dentro del directorio donde se encuentre el `build.gradle` anterior y creamos dentro de este nuevo directorio el fichero `external.gradle` con el contenido:

```
task external << {
   println "Estamos en el fichero externo."
}
```

Si ejecutamos la tarea `main`, veremos la salida de las dos tareas. A la hora de indicar el fichero a incluir en los scripts, si se encuentra en otro directorio, se debe utilizar siempre la barra "/" para separar directorios, aunque nos encontremos trabajando en Windows.

Plugins

Gradle nos permite realizar todo tipo de tareas de automatización, si bien, muchas de las tareas son comunes para un mismo lenguaje, por ejemplo, para C podemos tener tareas de compilación, de limpieza, de "*link*"… Para que cada usuario no se tenga que escribir sus propios scripts reinventando la rueda y pueda comenzar a usar Gradle desde un principio, se ofrece la posibilidad de usar *plugins* específicos para cada lenguaje de programación (y otras áreas), conteniendo las tareas más comunes de ellos. Estas tareas, además, vienen configuradas por defecto con las opciones más utilizadas para comenzar su uso con poco esfuerzo, pero el usuario puede sobreescribirlas para obtener un comportamiento distinto al preconfigurado.
Existen multitud de *plugins* para diferentes lenguajes de programación, procesos de integración, desarrollo de aplicaciones… y cada uno de ellos tiene diferentes tareas y convenciones.
En general se puede decir que al añadir un *plugin*, lo que estamos haciendo es:

- Añadir nuevas tareas a los proyectos
- Preconfigurar estas tareas facilitando así su uso
- Añadir dependencias
- Crear nuevas propiedades y métodos

Todo ello referido al *plugin* referenciado, por ejemplo si usamos un *plugin* de Java, no podemos esperar encontrar tareas referentes al link de programas C++.
Para añadir un *plugin* al script simplemente usamos la instrucción `apply plugin`, por ejemplo para utilizar el de Java sería:

```
apply plugin: 'java'
```

o

```
apply plugin: org.gradle.api.plugins.JavaPlugin
```

o

```
apply plugin: JavaPlugin
```

Y a partir de este momento podemos hacer uso de las tareas y propiedades referentes al *plugin* Java, por ejemplo crear los archivos JAR.

Nota:

Aunque Android se programa con Java, el plugin que se necesita para trabajar no es el de Java sino uno específico para Android que se verá en el próximo capítulo.

Niveles de log

Ya vimos que en la ejecución normal de las tareas se muestra en pantalla un log con el resumen del proceso realizado. En caso de que necesitemos un mayor grado de información, podemos cambiar el nivel de log utilizado para mostrar en la salida. Los distintos niveles disponibles son:

Nivel	Descripción
1	DEBUG Mensajes de depuración
2	INFO Mensajes de información
3	LIFECYCLE Mensajes de progreso
4	WARNING Mensajes de aviso
5	QUIET Mensajes de información importantes
6	ERROR Mensajes de error

Dependiendo de los modificadores que se le pongan al ejecutar las tareas, se mostrarán más o menos mensajes y se mostrarán los mensajes del nivel seleccionado y los mensajes correspondientes a niveles superiores.
Por defecto se usa el nivel 3 `LIFECYCLE` y los modificadores disponibles son:

- -q o --quiet mensajes QUIET y superiores
- -i o --info mensajes INFO y superiores
- -d o --debug mensajes DEBUG y superiores (todos los mensajes)
- -s o --stacktrace mostrar stacktraces
- -S o --full-stacktrace mostrar stacktraces completo. Mejor no usar esta opción puesto que son mensajes muy extensos que no aportan información

Gradle GUI

Además de poder trabajar desde línea de comando Gradle ofrece la posibilidad de utilizar un interface gráfico. Para acceder a este interface se debe ejecutar la sentencia:

```
gradle -gui
```

La herramienta leerá las tareas definidas en el directorio done se ejecute, mostrando un árbol con todas ellas.
Como se puede ver en la figura 5.2, la aplicación consta de una mitad superior y otra inferior. La mitad superior tiene a su vez cuarto pestañas:

- Task Tree: Árbol de las tareas disponibles junto a su descripción (si es la tiene)
- Favorites: Lista de comandos Gradle favoritos
- Command Line: Línea de comandos Gradle
- Setup: Configuración del comportamiento durante la ejecución de tareas

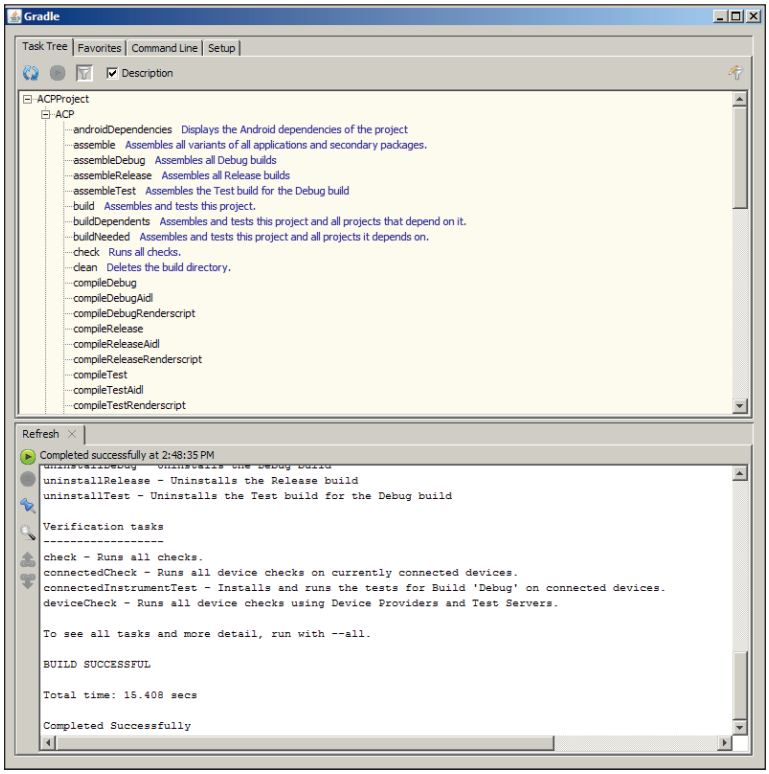

Figura 5.2. Tareas disponibles en Gradle GUI

La mitad inferior tiene una sola pestaña donde iremos viendo la salida de las ejecuciones de las tareas; además en la parte izquierda de esta zona de la pantalla tenemos una serie de botones que permiten volver a ejecutar de nuevo la tarea, detenerla en caso de que esté activa, bloquear el panel, realizar búsquedas en el texto de salida, moverse por los enlaces que se hayan generado (si hay alguno) y añadir a comandos favoritos la última ejecución.
Veamos un poco más de la parte superior. En la pestaña Task Tree tenemos disponible un árbol con las tareas detectadas durante la carga. Las tareas aparecen ordenadas de modo alfabético, pero en caso de tener muchas, mediante el icono situado arriba a la derecha  es posible crear filtros. Desde esta herramienta (de momento) no podemos crear nuevas tareas, pero en caso de crear o borrar alguna de ellas desde una herramienta externa, mediante el botón situado arriba a la izquierda  , podemos refrescar el contenido del árbol. Para ejecutar las tareas, podemos optar por hacer doble click en ellas o seleccionarlas y pulsar sobre el icono que contiene una flecha  .
En la pestaña Favorites mantendremos los comandos que más usemos (pueden ser tareas sueltas, combinación de tareas y parámetros…). Desde aquí también es posible la ejecución de los comandos guardados mediante el botón semejante al de la primera pestaña o mediante doble click sobre el comando a ejecutar. A través de los iconos situados a la derecha  es posible importar y exportar los comandos guardados como favoritos.
En la pestaña Command Line, como su nombre indica, disponemos de una línea de comandos Gradle. Cuando se ejecutan comandos desde esta pestaña, en el campo donde se indica el comando, NO hay que escribir el propio comando `gradle`, simplemente las tareas u opciones a ejecutar.
Por último, la pestaña Setup nos ofrece la posibilidad de controlar el directorio sobre el que trabajar (con su `build.gradle` correspondiente), el nivel de log e incluso seleccionar un script de sistema que se encargue de ejecutar Gradle en lugar de ejecutarlo directamente, esto es útil si hay que hacer tareas previas a la ejecución de Gradle y que se encuentran automatizadas en otros scripts.

Conclusiones

Aunque se hayan dado simplemente unas pequeñas pinceladas sobre el uso de Gradle, hemos podido atisbar parte de la flexibilidad y potencia que proporciona para realizar nuestros scripts. Ya habíamos comentado que Gradle ofrece muchas posibilidades (gestión de archivos, integración con Ant y Maven, distintos *plugins*…), tantas que no es posible abarcar en este capítulo, por lo que animo al lector a profundizar en la lectura sobre Gradle en documentos disponibles en su web oficial `http://www.gradle.org/`.

6 Gradle en Android Studio

En este capítulo aprenderá a:

- Conocer los archivos Gradle en Android.
- Distinguir las tareas existentes.
- Manejar Gradle desde Android Studio.

En este capítulo se quiere hacer una breve introducción a cómo encaja Gradle en Android Studio y presentar distintas configuraciones y modos de utilización por parte del usuario. Veremos una serie de procesos y opciones genéricas que vienen estando disponibles a lo largo de las versiones, no entrando en cuestiones muy específicas; no obstante, se recomienda al lector que si desea hacer tareas o procesos un poco complejos se dirija a la documentación oficial de Gradle sobre Android, ya que las tres piezas principales de este capítulo: Gradle, Android Studio (con sus *plugins*) y el propio Android; están en continua evolución y puede haber cambios en ciertas opciones, estructuras o nomenclaturas.

¿Por qué Gradle?

A lo mejor el lector se está preguntando ¿Porqué usar Gradle si existen otras herramientas de automatización más conocidas? Lo primero aclarar que puede utilizarse cualquier otra herramienta de automatización con la que nos sentamos cómodos, Ant, Make, Rake, Maven, programas Shell que nos hagamos a medida… o hacerlo todo a mano, pero vamos a intentar dar algunas razones por las que utilizar Gradle.

Dos de las herramientas de automatización de procesos más extendidas en el mundo Java son Apache Maven y Apache Ant y quizá haber adoptado su uso sería lo más natural. Veamos algunas ventajas y desventajas que presentan cada una de ellas.

Apache Maven permite definir la manera en la que se debe construir la aplicación y sus dependencias mediante una serie de convenciones, unas reglas predefinidas que pueden ser cambiadas. Estas dependencias pueden descargarse automáticamente de repositorios (tanto centrales como locales) previamente definidos. Podemos extender su funcionalidad mediante *plugins* ya existentes o realizarnos unos propios. Debemos realizar un fichero XML, que posee una estructura definida, con la configuración necesaria para la generación del programa y luego invocar al intérprete Maven para que ejecute cada uno de los pasos definidos. Normalmente esta configuración se encuentra en un fichero llamado `pom.xml` (*Project Object Model*, Modelo de objeto de proyecto). Maven se centra más en las dependencias que en la realización de tareas de configuración para la creación del ejecutable, dejando el proceso de las tareas a los *plugins* que se pueda crear el usuario, lo cual a veces complica la creación del script completo.

En cuanto a scripts de tareas, Apache Ant es mucho más flexible; es una herramienta que nos permite crear, de manera rápida, tareas a realizar durante la generación de los programas. Es posible que uno de los mayores problemas

que tiene esta herramienta es justamente en lo que es bueno Maven… el control de dependencias con librerías y repositorios, pero podemos apoyarnos en otras herramientas como Apache Ivy. Es muy extensible y el usuario puede generarse sus propias tareas o tipos (llamadas *antlibs*) mediante Java. Quizá el mayor problema de Ant + Ivy se dé en compilaciones multiproyecto y en la dificultad de mantenimiento de los scripts en proyectos complejos. Normalmente la configuración se encuentra en un fichero XML llamado `build.xml`.
Gradle intenta encontrar un punto común entre Ant y Maven, dando la facilidad de creación de tareas de Ant y las convenciones (valores por defecto) y el control de repositorios y dependencias de Maven; añadiendo además utilidades para las generaciones de aplicaciones multiproyecto. Con Gradle podemos ejecutar cualquier tarea que tengamos prevista y se encargará previamente de obtener sus dependencias y descargarlas en caso de que fuera necesario, da igual cómo esté distribuido el código (en directorios distintos, en repositorios…).
Como se vio en el capítulo anterior, en Gradle existen dos fases, evaluación y ejecución. Durante la evaluación, Gradle buscará los scripts en los directorios que se le haya indicado y comenzará su evaluación; será durante la fase de ejecución donde se realicen las acciones de las tareas evaluadas, teniendo en cuenta las interdependencias entre ellas. Como raíz de las dependencias de Gradle se encuentra el proyecto y sus dependencias con los JAR. Gradle es capaz de detectar dependencias entre proyectos y entre proyectos y JARs. En cuanto a repositorios, Gradle puede trabajar (subiendo o descargando datos) con repositorios Ivy, con repositorios Maven y otros muchos tipos de repositorios de código, tanto públicos como propios. En el caso de las compilaciones multiproyecto, Gradle se adapta a la estructura de archivos y proyectos que se tenga… no hace falta adaptar nuestras estructuras a unos patrones de directorios como en Maven, (lo cual es a veces muy difícil o imposible).
Gradle nos puede ayudar no sólo en la creación del programa, sino también en las pruebas, publicaciones, generación de paquetes… No está pensado solo para Android; podemos usarlo para ayudarnos en las tareas relacionadas con la creación de aplicaciones en cualquier lenguaje de programación, mantenimiento de documentación… o hasta donde nos llegue la imaginación.
Además Gradle utiliza un lenguaje específico de dominio (domain-specific language o DSL) en lugar de usar declaraciones XML. El lenguaje de programación utilizado para los scripts es Groovy, que es un lenguaje dinámico (igual que Ruby o Python) de muy fácil aprendizaje si se conoce Java (aunque Gradle permite automatizar tareas para cualquier lenguaje de programación, estaba destinado a proyectos Java y por eso se seleccionó Groovy como lenguaje de programación de las tareas). La aceptación de Gradle como herramienta de automatización por parte de la comunidad, está haciendo que casi todos los

entornos de desarrollo tengan uno o varios *plugins* o *add-on* de modo que se integren con ella de forma cómoda; por supuesto Android Studio lo tiene.
Si el lector aún no está convencido de usar Gradle en lugar de otras (o ninguna) herramienta de automatización de tareas, vamos a comentar una última razón por la que se realiza un capítulo en este libro: Google la ha elegido como la herramienta de automatización por defecto en Android Studio.

Estructura básica de build.gradle

Cuando generamos un nuevo proyecto en Android Studio, se crean de modo automático una serie de directorios en los cuales se albergarán los fuentes y recursos del proyecto. Del mismo modo se crea también de modo automático unos ficheros `build.gradle` con ciertas opciones que encajan con la estructura de proyecto creada. Recordemos que Gradle funciona con opciones informadas con valores por defecto y que podemos cambiarlas para ajustarlas a la morfología del proyecto, pero en este caso, como está recién creado y tiene la estructura predefinida, no hace falta cambiar nada para que funcione correctamente.
Cuando se genera el proyecto, se nos pide un nombre de módulo que traducido a sistema de archivos se convierte en un directorio dentro del directorio del proyecto. Gradle tendrá un fichero `build.gradle` a nivel de proyecto y otro dentro de cada módulo que se le añada (recordemos que podemos cambiar esta manera de trabajar, pero inicialmente está preparado para trabajar así).
Si abrimos el fichero `build.gradle` que encontramos a nivel de proyecto, su contenido será parecido a:

```
buildscript {
   repositories {
      mavenCentral()
   }
   dependencies {
      classpath 'com.android.tools.build:gradle:0.8.+'
   }
}
```

En él se define que se utilizará el repositorio central Maven y podemos ver que se tiene una dependencia sobre un objeto, que en este caso es la librería que contiene el *plugin* Android para Gradle, que sea de la familia de la versión 0.8. Al ser definido el repositorio Maven, tenemos ciertas ventajas, como por ejemplo si queremos comenzar a usar el *plugin* Android versión 0.9, simplemente cambiaríamos el valor en este fichero y el sistema se encargaría de descargar dicha versión del repositorio de manera transparente para nosotros. Este fichero se utilizará durante la compilación, y se usará para saber cómo se debe

comportar el proceso de compilado sin entrar en cuestiones del proyecto Android en sí, para eso tenemos el fichero propio del módulo.
El fichero `build.gradle` generado dentro del directorio del módulo es un poco diferente, si lo abrimos veremos que tiene un aspecto semejante a:

```
apply plugin: 'android'

android {
   compileSdkVersion 19
   buildToolsVersion "19.0.1"

   defaultConfig {
      minSdkVersion 14
      targetSdkVersion 19
      versionCode 34
      versionName "1.2 rc23"
   }
   buildTypes {
      release {
         runProguard false
         proguardFiles getDefaultProguardFile('proguard-android.txt'),
'proguard-rules.txt'
      }
   }
}

dependencies {
   compile 'com.android.support:appcompat-v7:+'
}
```

Dado que se va a utilizar Gradle para construir una aplicación Android, lo primero que se hace en el fichero es añadir el plugin correspondiente del mismo modo que vimos cómo añadir el de Java en el capítulo anterior; mediante la línea:

```
apply plugin: 'android'
```

Advertencia:

No se deben añadir los plugins `android` *y* `java` *en el mismo script dado que se producirían errores durante la ejecución.*

Tras esta línea se encuentra el objeto (bloque) `android{}`, que es el punto de entrada al DSL (*Domain Specific Language*, lenguaje específico de dominio) de Android y donde se configuran los parámetros que se utilizarán durante la construcción de la aplicación Android. Dentro de este objeto, la única entrada obligatoria es la propiedad `compileSdkVersion` que sirve para definir el nivel o versión de SDK que se utilizará para compilar el código. Éste parámetro (y otros tantos) se lo hemos proporcionado al proyecto durante el asistente de creación de la aplicación.

La entrada `buildToolsVersion` permite especificar la versión de las herramientas de Android Studio que se deben utilizar para generar la aplicación. Por último aparece un objeto con las dependencias del módulo llamado `dependencies{}`; estas dependencias son de la compilación del módulo, que no hay que confundir con las dependencias vistas en el archivo `build.gradle` anterior, que indicaban las dependencias que se tendrían que cumplir para realizar el proceso de compilación; en este caso se indica que depende de la librería de compatibilidad versión 7.

Nota:

Los SDK disponibles vienen referenciados por el fichero `local.properties` *bajo la clave* `sdk.dir` *que contiene el* path *al directorio donde se almacenan. También es posible crear la variable de sistema* `ANDROID_HOME` *apuntando a este directorio.*

Dentro del objeto `android{},` se encuentra el objeto `defaultConfig{}` en el que podemos encontrar algunas entradas que nos recordarán a las disponibles en el fichero `AndroidManifest.xml`. Desde este objeto podemos configurar los valores a utilizar para la generación del programa del mismo modo que lo haríamos desde el `AndroidManifest.xml`. En caso de que un valor no sea informado mediante DSL en el script, tomará su valor del y si está informado en el fichero `AndroidManifest.xml`, entonces tomará este último valor. Las entradas disponibles para configurar y sus valores por defecto son:

Propiedad	Valor por defecto en DSL	Valor por defecto
versionCode	-1	valor del manifest si existe
versionName	null	valor del manifest si existe
minSdkVersion	-1	valor del manifest si existe
targetSdkVersion	-1	valor del manifest si existe
packageName	null	valor del manifest si existe
testPackageName	null	paquete de la app + ".test"
testInstrumentation-Runner	null	android.test.InstrumentationTestRunner
signingConfig	null	null
proguardFile	No disponible	No disponible
proguardFiles	No disponible	No disponible

La diferencia entre las entradas `versionCode` y `versionName`, estriba en que la primera ha de ser un valor numérico, mientras que la segunda puede ser lo que queramos; esto se puede ver como que la primera es para que la entienda el ordenador y la segunda es para que la entendamos los humanos. Cuando se sube una aplicación al *Market*, las futuras versiones de la misma se controlan desde `versionCode`, siendo las versiones más nuevas las que más alto tengan el número de esta propiedad. Las entradas `minSdkVersion`, `targetSdkVersion` y `packageName` se refieren a las propiedades con el mismo nombre que se rellenan durante los pasos del asistente para generación de la aplicación. `testPackageName` y `testInstrumentationRunner` se utilizan para informar el nombre del paquete y el nombre de la clase de ejecución de las pruebas. La entrada `signingConfig` se refiere a la configuración para la firma de la aplicación cuando se genera el fichero `apk`. `proguardFile` y `proguardFiles` son referentes a como se debe comportar el ofuscador ProGuard.

Nota:

En las tablas se están mostrando los valores por defecto, cuando se dice "no disponible" es que no hay valor por defecto y es el propio usuario quien lo tiene que informar o no se utilizará.

Por su parte, el objeto `buildTypes{}` contiene la configuración para los distintos tipos de construcción de la aplicación. Por defecto, el *plugin* ya prepara la configuración para la construcción de la versión de desarrollo y la versión de entrega o definitiva. La configuración de desarrollo básicamente se diferencia en la manera que es firmado el fichero `.apk`, la información adicional para depurar y en las opciones de ofuscación, así podremos depurar la aplicación en un dispositivo físico durante la fase de desarrollo. La firma de la aplicación durante la etapa de desarrollo se realiza con un certificado generado de modo automático y cuyo usuario y clave es conocido por el sistema y así no es necesario introducirlo en cada compilación; al contrario que en la versión de entrega que usaremos un certificado que solamente nosotros sabremos el clave. Más adelante veremos cómo realizar la firma de la aplicación. Si quisiéramos modificar la estructura de nuestro proyecto para adecuarlo a nuestras necesidades, se debe variar la configuración dentro del objeto `android{}` de modo que Gradle sepa dónde encontrar cada recurso. Cada tipo fuente o recurso se puede mantener en directorios separados o que más convenga al usuario, simplemente se debe configurar su entrada correspondiente; por ejemplo:

```
android {
    sourceSets {
        main {
            manifest.srcFile 'AndroidManifestProd.xml'
            java.srcDirs = ['src/java']
            resources.srcDirs = ['src/resources']
            aidl.srcDirs = ['src/aidl']
            renderscript.srcDirs = ['src/renderScript']
            res.srcDirs = ['res']
            assets.srcDirs = ['assets']
        }

        instrumentTest.setRoot(pruebas')
    }
}
```

Tareas

Cuando se aplican *plugins* a los scripts, automáticamente se tienen disponibles nuevas tareas relacionadas con el *plugin*, preparadas para ser utilizadas. En nuestro caso, al utilizar el *plugin* Android, como es de suponer, las tareas son relacionadas con el ecosistema Android y la generación de sus aplicaciones; en concreto alguna de las tareas disponibles son:

- `assemble`: Ensambla las salidas del proyecto.
- `check`: Realiza todas las comprobaciones
- connectedCheck: Realiza todas las comprobaciones en las que se requiera un emulador o un dispositivo conectado. Las comprobaciones se realizaran en todos los dispositivos que se encuentren conectados al sistema de modo paralelo.
- `deviceCheck`: Realiza las comprobaciones usando las APIs para conectarse a los dispositivos remotos.
- `build`: Se encarga de ejecutar las tareas de assemble y check
- `clean`: Realiza la limpieza de los archivos de salida del proyecto

Si el lector ha utilizado anteriormente *plugin* Java en Gradle, le resultarán familiares las tareas, y es que recordemos que el *plugin* Android está basado en el Java y es por ello que comparte el nombre de muchas de las tareas (por ejemplo `assemble`, `check`, `build` y `clean`).
Por supuesto también existen tareas para la creación de la instalación y desinstalación de la aplicación para cada fase de construcción (desarrollo, test y liberada).

Dentro de la tarea `assemble`, existen subtareas que nos permiten realizar el ensamblado de la aplicación para trabajar en modo depuración, pruebas o distribuciónConcretamente las subtareas son `assembleDebug`, `assembleDebugTest` y `assembleRelease`, compartiendo todas ellas la tarea base `assemble`.
Del mismo modo que se vio en el capítulo anterior, en la instalación Gradle de Android Studio, es posible realizar las llamadas a las tareas con nombres abreviados; por ejemplo en lugar de llamar a la tarea `deviceCheck`, podríamos llamar a `dC` obteniendo el mismo resultado (a no ser que se haya generado alguna otra tarea que corresponda a esas iníciales), aunque como veremos más adelante, podemos ahorrarnos escribir en la línea de comando utilizando un panel de Android Studio.

Tipos de compilación

Por defecto, una vez creada la aplicación, ya tenemos disponible todo lo necesario para probarla tanto en nuestro dispositivo físico como en el emulador y esto se debe a que se han generado unas tareas que nos permitirán crear las versiones de *debug* (de depuración) y la de *release* (la definitiva). En el ejemplo de `build.gradle` que hemos visto anteriormente, existía la entrada:

```
buildTypes {
   release {
      runProguard false
      proguardFiles getDefaultProguardFile('proguard-android.txt'),
'proguard-rules.txt'
   }
}
```

Aquí se está configurando el tipo de compilación *release* para que no use el ofuscador ProGuard y (aunque se le diga que no lo use) configura los ficheros que debe utilizar en caso de usarse.
Nosotros podemos crear nuevos tipos de compilación con diferentes opciones para no estar cambiando constantemente el archivo de configuración, por ejemplo podemos tener un tipo de compilación `release` y otro llamado `releaseProguard` donde usemos el ofuscador. Cuando creamos un tipo de compilación nuevo, automáticamente se genera una tarea llamada `assemble<nombre_nuevo_tipo>`.
Las propiedades que podemos utilizar en cada tipo de compilación y sus valores por defecto son:

Nombre de propiedad	Valores *debug*	Valores *release*
debuggable	true	false
jniDebugBuild	false	false
renderscriptDebugBuild	false	false
renderscriptOptimLevel	3	3
packageNameSuffix	null	null
versionNameSuffix	null	null
signingConfig	android.signingConfigs.debug	null
zipAlign	false	true
runProguard	false	false
proguardFile	no disponible	no disponible
proguardFiles	no disponible	no disponible

Integración con el entorno Android Studio

Ya sabemos la manera de invocar las tareas Gradle desde línea de comando, incluso sabemos que tenemos la posibilidad de utilizar un entorno gráfico donde gestionarlas, pero siempre es más cómodo tener acceso a las tareas Gradle directamente desde el entorno de desarrollo Android Studio; para ello tenemos una serie de paneles que nos facilitarán el acceso.

Los ficheros de configuración Gradle se encargan de controlar ciertos parámetros de nuestro proyecto, esto quiere decir que cada vez que modificamos un `build.gradle`, hay otras zonas del proyecto que pueden verse afectadas. Se habrá fijado el lector que hoy por hoy Gradle no es una herramienta que podamos decir que sea muy rápida en ejecución, por lo que la sincronización entre los ficheros de configuración Gradle y el resto del proyecto no se hace en el momento, sino que se debe forzar. Es verdad que hay modificaciones que no requieren sincronización alguna (por ejemplo cambiar un mensaje de aviso de una tarea), pero hasta que no se sepa qué cambios requieren sincronización y que cambios no, mi recomendación es que cada vez que modifiquemos un fichero Gradle se sincronice con el proyecto. Para realizar la sincronización se debe pulsar el menú Tools>Android>Sync Project with Gradle Files o el icono  , tras lo cual se dispararán las tareas correspondientes. Por ejemplo si se cambia la entrada

```
classpath 'com.android.tools.build:gradle:0.8.+'
```

por

```
classpath 'com.android.tools.build:gradle:0.9.+'
```

se descargaría el nuevo *plugin* Gradle, se limpiarían los archivos generados y se generarían de nuevo con los nuevos componentes descargados.
Para mostrar las tareas definidas en el proyecto, se dispone de un panel que puede hacerse visible pulsando en el icono situado en la esquina inferior izquierda y seleccionando la entrada Gradle. El panel nos muestra no sólo las tareas disponibles sino también las recientemente utilizadas. Para lanzar una de las tareas simplemente vale con hacer doble click sobre ella momento en el que se ejecutará y pasará a formar parte de las tareas recientes.

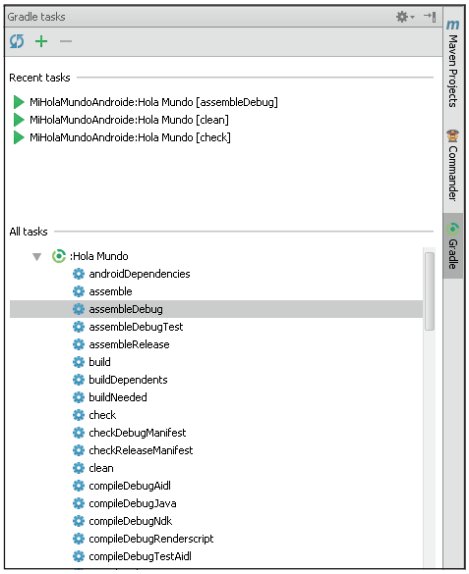

Figura 6.1. Panel de tareas Gradle.

Al ejecutar las tareas desde el panel, también se nos hacen disponibles en el desplegable situado en la parte superior del entorno como puede verse en la figura 6.2, donde podemos seleccionarlas y ejecutarlas de nuevo con el botón situado a la derecha del desplegable.

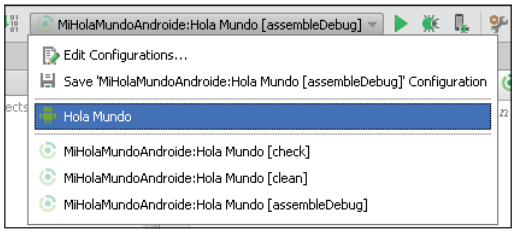

Figura 6.2. Configuraciones de ejecución.

Desde este desplegable podemos acceder también al editor de las configuraciones de ejecución, tanto de la aplicación (pudiendo configurar cosas como en qué dispositivo se debe ejecutar) como de las tareas Gradle. Muy importante es que cuando se pulsa sobre el botón  se ejecuta la tarea que tengamos seleccionada en el desplegable.
Las tareas generan una serie de información que es mostrada en el panel Gradle Console, que como todos los paneles, está accesible desde el icono inferior izquierdo.
Este panel puede no ser suficiente para la obtención de toda la información que esté ocurriendo durante los procesos de las tareas Gradle, en tal caso podemos apoyarnos en otros paneles como Messages o Event Log, Si la tarea es ejecutada desde el icono  su salida la obtendremos en el panel Run.

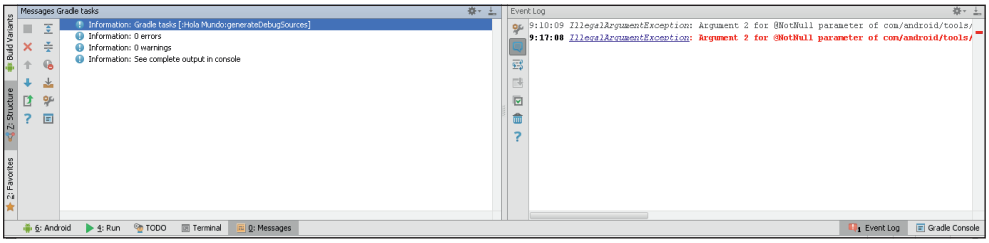

Figura 6.3. Paneles Event Log y Messages.

Tras la ejecución de alguna tarea o cambio en el proyecto, podemos necesitar compilar de nuevo todo el proyecto para que coja todos los cambios y desaparezcan errores de código; para realizar esta tarea tenemos disponible el botón  que realmente ejecuta la tarea `assemble` de Gradle en la variante seleccionada (debug, release...). Para seleccionar la variante con la que queremos trabajar, disponemos de otro panel llamado Build Variants, donde tenemos disponibles

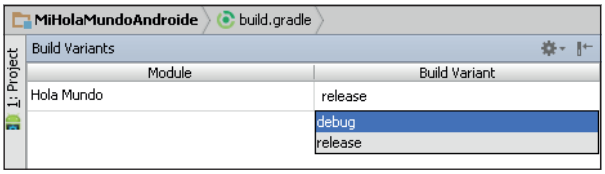

Figura 6.4. Panel Build Variants.

Así cuando esté seleccionado el Build Variant debug, se ejecutará la tarea `assembleDebug`, y cuando esté seleccionado release, la tarea a ejecutar será `assembleRelease`. Cuando generemos nosotros nuevos tipos de ensambla-

do mediante el fichero `build.gradle`, estas estarán disponibles en este panel y su correspondiente tarea de `assemble` podrá ser invocada del mismo modo que las tareas estándar.

Firma de aplicación

Cuando se genera el fichero instalable de la aplicación, éste debe ir firmado. Ya habíamos comentado que por defecto, Android Studio mantiene una configuración para la creación de las aplicaciones de depuración, en la que no es necesario dar datos sobre la firma que se utilizará, puesto que usa una por defecto. Realmente para la firma se necesita:

- Un almacén de claves
- Un tipo de almacén
- El password del almacén
- Un nombre alias de la clave
- El password de clave

El almacén (o *keystore*) utilizado para la depuración se encuentra en `$HOME/.android/debug.keystore` y los valores por defecto para acceder a ella se guardan en la configuración *SigningCongfig* de Gradle. Para los curiosos, los passwords de la configuración por defecto son "android" y el alias es "androiddebugkey", pero no debe preocuparnos que sea tan débil y conocida por todos, puesto que para la versión final de la aplicación deberemos firmarla con otra clave.
Si quisiéramos cambiar la configuración y generar un archivo de depuración con otra firma distinta se podría hacer variando la configuración Gradle.

```
android {
   signingConfigs {
      debug {
         storeFile file("debug.keystore")
      }

      secretDebugableConfig {
         storeFile file("secret.keystore")
         storePassword "pass1"
         keyAlias "AliasDeClave"
         keyPassword "pass2"
      }
   }

   buildTypes {
```

```
        secureDebug {
            debuggable true
            jniDebugBuild true
            signingConfig signingConfigs.secretDebugableConfig
        }
    }
}
```

En el ejemplo se informan los passwords en el propio fichero de configuración, lo cual no es muy seguro (sobre todo si compartimos el ordenador); podemos no informarlos y que el sistema nos los pida cuando los necesite, pero para el proceso de codificación/depuración de la aplicación, es muy molesto tener que estar introduciendo estos datos en cada prueba.

Figura 6.5. Pantalla de selección de almacén de claves.

Para realizar el archivo `apk` para distribución, podemos optar por dos opciones, la Gradle pura o mediante asistente.
Al asistente se accede mediante el menú Build>Generate Signed APK..., tras lo cual se mostrará una ventana avisando que la generación de firma de los proyectos Gradle pueden ser configurados directamente en los scripts. La siguiente pantalla sería la de la figura 6.5, donde podemos seleccionar un almacén de claves existente o crear uno nuevo.
Cuando decidimos generar uno nuevo, debemos rellenar una serie de cajas de texto como lugar donde se guardará el archivo creado, password del archivo, alias de la clave, password de la clave, información personal del alias y caducidad de la clave (véase pantallas de la figura 6.6).
Cuando seleccionamos el almacén de claves, podemos indicar que se guarden los passwords introducidos; en tal caso, deberemos proporcionar a su vez otro passwords de modo que estos passwords también se guarden cifrados en el sistema (véase figura 6.7).

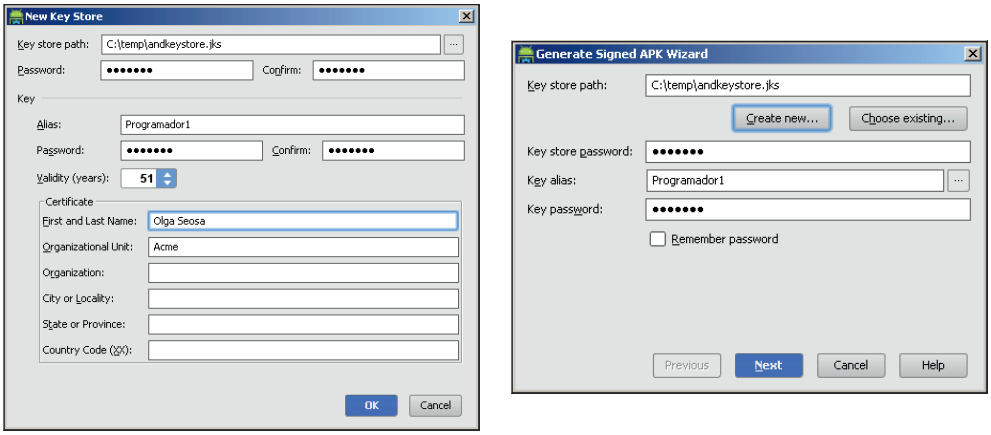

Figura 6.6. Pantallas de creación de clave y selección.

Figura 6.7. Petición de password para la base de passwords.

El último paso del asistente nos preguntará por la ruta de ficheros donde queremos depositar el archivo `.apk` generado, así como si se quiere utilizar ProGuard o no.

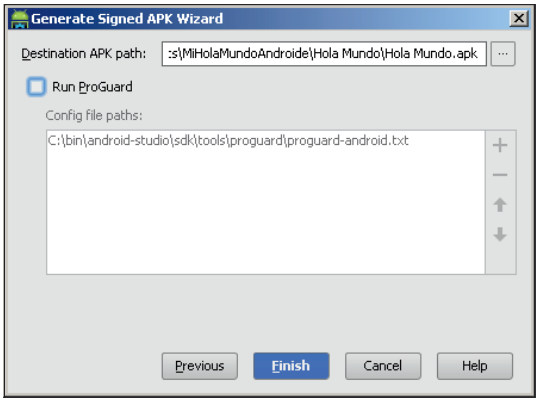

Figura 6.8. Configuración de ProGuard.

Tras la realización de estos pasos, tendríamos en el lugar indicado el fichero `.apk` generado y firmado con la clave seleccionada.
Si tras ejecutar este asistente analizamos los mensajes mostrados en los distintos paneles, veremos que las tareas realizadas son realmente las de `assembleRelease`.
Si se quisiera realizar mediante Gradle puro, se debería modificar el archivo `build.gradle` para añadir las condiciones de firma al tipo de construcción `assembleRelease` o crear uno nuevo con estas condiciones, por ejemplo:

```
signingConfigs {
   release {
      storeFile file("release.keystore")
      storePassword "pass1"
      keyAlias "AliasDeClave"
      keyPassword "pass2"
   }
}
```

Ya sólo quedaría ejecutar la tarea `assembleRelease` desde Android Studio y obtendríamos el fichero `.apk` en el directorio `build/apk`.

7 Interfaces de usuario

En este capítulo aprenderá a:

- Diseñar pantallas de forma eficiente utilizando diferentes disposiciones.
- Diferenciar las particularidades de cada *layout*.
- Reutilizar varias pantallas en una sola aplicación.
- Previsualizar las pantallas.
- Utilizar el editor gráfico.

Para que el usuario pueda comunicarse e interactuar con las distintas *Activity*, éstas ofrecen una interfaz gráfica con elementos que son capaces de reaccionan distintos eventos generados sobre la pantalla o sobre alguno de los botones del dispositivo. Las *Activity* tendrán asociadas tantas interfaces como el desarrollador crea necesarias, que podrán variar dependiendo la situación dentro de la aplicación.

Para la creación de estas interfaces gráficas, se puede optar por realizarlas en tiempo de ejecución (de manera dinámica mientras se ejecuta la aplicación) o en tiempo de diseño (mientras se programa la aplicación, lo cual facilita la tarea por el hecho de poder hacer uso de herramientas gráficas).

La manera más sencilla como hemos dicho es realizarlo en tiempo de diseño, pero dependiendo de la naturaleza de la aplicación, es posible que sea necesario hacer toda o parte de la interfaz mediante programación. Por ejemplo una aplicación que muestre un botón por cada letra del abecedario dependería del idioma para mostrar un número de botones u otro ya que no tienen todos los idiomas las mismas letras, por lo que en este caso sería necesario que la aplicación al ejecutarse detectara el idioma y una vez conocido el número de botones que debe tener y sus letras, realizar la interfaz de modo dinámico mediante programación.

En este capítulo nos centraremos en descubrir la manera en la que realizar las interfaces a la hora de diseñar la aplicación, conociendo los distintos elementos que podemos utilizar, sus usos y maneras de disponerlos en pantalla.

Generalidades

Del mismo modo que otros muchos lenguajes de programación como Adobe Flex, Mozilla XUL (*XML User interface Language*)… los documentos en los que se especifica el *layout* (disposición) para Android, son ficheros de texto en formato XML, que contienen de manera jerárquica los elementos (contenedores o *widgets*) que se quieren mostrar en pantalla. Estos archivos se almacenan en la carpeta `res/layout` del proyecto Android.

Los *layouts* están compuestos de elementos que pueden ser contenedores o *widgets*. Los contenedores son elementos visuales o no, que pueden a su vez contener otros contenedores o *widgets*; son elementos que descienden de la clase *ViewGroup* que a su vez desciende de la clase *View*. Los *widgets* de interfaces gráficas se pueden encontrar dentro del paquete `android.widget` y son casi todos ellos elementos visuales que descienden de la clase *View*.

Nota:

No se debe equivocar los widgets *que se utilizan para diseñar las pantallas de los* widgets *que se colocan en la pantalla principal de Android. Los segundos son unas pequeñas aplicaciones con características especiales que se verán más adelante en el capítulo 15 del libro.*

La manera en la que se mostrarán o cómo se comportaran los contenedores o los *widget* viene dado por los atributos que se le den a cada elemento en su definición dentro del documento XML. Por ejemplo para mostrar una caja de texto que tenga por defecto el contenido "contenido por defecto", se le deberá informar el atributo `android:text="contenido por defecto"`. Dependiendo del elemento que se esté definiendo, habrá unos atributos que serán obligatorios, otros optativos y otros que existirán para unos elementos y no para otros. Nos encontraremos también con atributos sobre la colocación en pantalla del elemento que, como veremos en este mismo capítulo tendrán sentido dependiendo del contenedor en el que se encuentren alojados.

Una vez que se tiene el fichero con el *layout* definido, éste se debe procesar mediante una herramienta llamada `aapt` (*Android Asset Packaging Tool*, herramienta de empaquetamiento de activos o recursos) que generará la correspondiente clase java de recursos llamada *R*. En esta clase generada por la herramienta se mantienen mediante constantes todos los recursos a los que la aplicación puede acceder (imágenes, cadenas, *layouts*…) de manera que sea más sencillo usarlos durante la programación. Esta clase la podemos encontrar en el directorio `build/source/r` dentro de cada proyecto. De la llamada al programa `aapt` no nos debemos de preocupar, ya que Android Studio lo hará de modo automático. En caso de querer programar sin utilizar este entorno, es posible usar herramientas como ant o Gradle para automatizar su llamada.

Lo mejor de tener la definición del interfaz en un fichero separado es que se puede diferenciar la capa de presentación de la lógica que la gestiona, cosa que no podríamos hacer si la capa de presentación se genera directamente mediante programación. Si es nuevo en la creación de interfaces mediante XML, es posible que al principio se encuentre un poco incómodo, pero poco a poco verá que es muy sencillo crear y modificar las pantallas. Además, se dispone de una herramienta para diseñar de modo *wysiwyg* (*what you see is what you get*, lo que ves es lo que obtienes) o incluso podemos ayudarnos de herramientas externas como "DroidDraw" (descargable de modo gratuita en `http://www.droiddraw.org/`) para facilitar su construcción.

Comenzaremos analizando el archivo de *layout* generado anteriormente en la aplicación "*Hola Mundo*". Para ello desplegamos el directorio `/src/`

`main/res/layout` de dicho proyecto; en este directorio encontraremos el fichero de *layout* denominado `activity_main.xml`. Si lo abrimos, se mostrará en el panel central una nueva vista con su contenido y dos pestañas, una denominad Design y otra llamada Text otra con el contenido XML del archivo:

```
<?xml version="1.0" encoding="utf-8"?>
<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:paddingLeft="@dimen/activity_horizontal_margin"
android:paddingRight="@dimen/activity_horizontal_margin"
android:paddingTop="@dimen/activity_vertical_margin"
android:paddingBottom="@dimen/activity_vertical_margin"
tools:context="com.acme.holamundo.MainActivity$PlaceholderFragment">

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

En primer lugar nos encontramos la línea de definición de fichero XML, esta línea es común a todos los archivos de este tipo, aunque es posible verlo con otros atributos u otros valores de atributos e incluso que no aparezca la línea en el archivo. En este caso el atributo `encoding="utf-8"` se utiliza para indicar el tipo de codificación para el texto almacenado; dependiendo del alfabeto del idioma en el que se escriba el fichero será necesaria una codificación u otra, como puede ser para poder mostrar caracteres cirílicos o kan ji.

La siguiente entrada XML ya es un elemento propiamente dicho del *layout*. En este caso define un contenedor de tipo *RelativeLayout*. Los contenedores, como su propio nombre indica, sirven para agrupar en su interior diferentes *widgets* (botones, imágenes, casillas de selección…) u otros contenedores.

Muy probablemente a lo largo de nuestros desarrollos el *RelativeLayout* (junto con el *LinearLayout*) sea uno de los contenedores que más utilicemos, ya que es muy flexible y permite ajustar muy bien la posición de los elementos aunque cambie el tamaño de la pantalla o que se varíe la posición del terminal de vertical a horizontal o viceversa.

Existen diferentes tipos de contenedores que a través de sus propiedades permiten tener un control absoluto de la colocación de cada elemento en la pantalla. El contenedor definido en este *layout* es de tipo relativo, es decir, los elementos definidos en su interior se colocarán relativos unos a otros en el mismo orden en el que aparecen en el archivo XML. Los dos atributos que comienzan por `layout` sirven para concretar el tamaño del contenedor (`android:layout_width` para definir la anchura y `android:layout_`

height para la altura), pudiendo tener los valores fill_parent o match_parent (ocupar todo el espacio del contenedor padre), wrap_content (ajustar al contenido) o un valor personalizado marcando el tamaño en alguna de las unidades que más adelante se verán. En este caso, como el contenedor definido es el elemento raíz de la estructura al indicar que sea de tipo match_parent se le está diciendo que ocupe toda la superficie de la pantalla. Usar fill_parent o match_parent es indiferente, aunque se aconseja usar match_parent, el resultado es el mismo. Los atributos que comienzan por xmlns definen espacios de nombres (*namespace*) estándar de XML, el xmlns:android="http://schemas.android.com/apk/res/android" sirve para definir el *namespace android* que será utilizado en el resto de la definición del XML, mientras que xmlns:tools="http://schemas.android.com/tools" sirve para el espacio de nombres para las ayudar al editor gráfico. Los atributos android:paddingLeft, android:paddingRight, android:paddingTop y android:paddingBottom sirven para indicar que se dejen unos espacios alrededor del componente a modo de márgenes internos. Los valores de estos atributos vienen dados por unas constantes definidas en el archivo dimens.xml dentro del directorio /src/main/res/values aunque se pueden incorporar valores numéricos directamente.

Nota:

Desde el nivel 8 del API de Android, el valor fill_parent *se ha renombrado a* match_parent*. A lo largo del libro se utilizarán indistintamente tanto un valor como otro ya que son igualmente válidos, aunque la tendencia debe ser ir abandonando el uso de* fill_parent*.*

Dentro del contenedor *RelativeLayout* se ha definido un *widget* de tipo *TextView*. Este elemento sirve para mostrar textos en pantalla, de modo semejante al *widget* etiqueta que se encuentra en otros entornos de programación. En sus propiedades vemos algunas ya vistas en el contenedor por ejemplo las referentes a su tamaño en el *layout*. Las propiedades definidas son:

- android:layout_width=" wrap_content": Se está diciendo que ocupe el espacio necesario a lo ancho (siempre sin excederse del tamaño de su contenedor padre).
- android:layout_height=" wrap_content ": Se indica que a lo alto solamente ocupe aquel espacio necesario para mostrar la información que debe mostrar.

- `android:text="@string/hello_world"` : Permite indicar el texto a mostrar en la etiqueta. Este texto se mostrará cuando se cree la etiqueta, pero se podría establecer mediante programación en cualquier momento de la ejecución del programa. Hay que fijarse en que comienza por *@string/xxx* donde *xxx* es un identificador de una constante cadena que habremos definido en el programa, en el archivo `strings.xml` dentro del directorio `/src/main/res/values`. Cuando el procesador XML encuentra *@string* en esta propiedad, la expande buscando su valor en el fichero de recursos *R*, a través de su identificador de cadena. El valor de la propiedad también se puede indicar de modo directo sin usar *@string*, por ejemplo, si fuera `android:text="Hola mundo"` el procesador XML no tendría nada que expandir por lo que el valor mostrado sería directamente el indicado en el XML ("Hola mundo"). Esta manera de informar los textos en las interfaces, no es una buena práctica ya que no permite usar varios idiomas, si extraemos los literales a un fichero XML de constantes de cadena, nos será más fácil poder traducir la aplicación a otros idiomas, más adelante en este mismo libro veremos cómo.

Ahora que ya conocemos un poco la estructura del archivo de *layout* vamos a ver más en profundidad los elementos que podemos utilizar para crear nuestras interfaces y jugar un poco con ellas para familiarizarnos, pero antes de entrar en detalle un pequeño inciso para explicar un atributo especial, el `android:id`. Este atributo puede estar presente o no en los elementos del XML y sirve para identificar de forma unívoca a cada elemento del *layout* para que más adelante podamos acceder mediante programación al objeto que representa en pantalla. Por ejemplo podríamos haber definido el anterior *TextView* como:

```
<TextView
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_centerHorizontal="true"
android:layout_centerVertical="true"
android:id="@+id/TextView01"
      android:text="@string/hello_world"/>
```

En este caso hemos añadido además un par de atributos que nos servirán para centrar horizontal y verticalmente el texto en la pantalla (al ser centrado horizontal y verticalmente se podría haber optado por `android:layout_centerInParent="true"`); vayamos a lo que nos interesa que es el identificador del *TextView* que para el ejemplo es `"TextView01"`. En la definición de los identificadores podemos encontrar que sean del tipo *@id/xxx* o del tipo *@+id/xxx* (donde *xxx* es el identificador). Al escribir *@id/xxx* se indica al procesador XML que se debe expandir el resto de la cadena y usarlo

como identificador que estará definido como recurso en *R*. Si lo que se escribe es *@+id/xxx* (nótese el signo +) se indica que aparte de tener que expandir el identificador, se debe de crear dicho identificador como recurso en *R* en caso de que no exista. Ahora que el elemento tiene un identificador, podremos acceder a él mediante programación y cambiar sus características.
Veamos alguno de los *layouts* que tenemos disponibles.

Tipos de layouts

En este punto se creará un proyecto que ayudará a comprender sus distintos comportamientos y opciones de los *layouts*. Para crear el nuevo proyecto Android, diríjase al menú File>New Project... y complete el asistente con los siguientes valores (los que no estén informados, pueden dejarse lo que vienen por defecto):

- Application name: Container Test
- Module name: Container Test
- Package name: com.acme.containertest
- Minimum Required SDK: API 14: Android 4.0 (IceCreamSandwich) o superior
- Aditional Features: Swipe views (Viewpager)

Es muy importante seleccionar la opción API 14 (o superior) por los *layouts* que usaremos y sólo están disponibles a partir de esa versión; en el desplegable Aditional features en el último punto del asistente, es donde seleccionaremos Swipe views ya que es el tipo de navegación que utilizaremos en el proyecto y en el que se basará toda la lógica para mostrar los distintos *layouts*. Sin querer analizar en profundidad el código, vamos a ajustarlo a nuestras necesidades. En el fichero `MainActivity.java` podemos encontrar el código:

```
public View onCreateView(LayoutInflater inflater, ViewGroup container,
            Bundle savedInstanceState) {
   View rootView = inflater.inflate(R.layout.fragment_main, container,
   false);
   TextView textView = (TextView) rootView.findViewById(R.id.section_label);
   textView.setText(Integer.toString(getArguments().getInt(ARG_SECTION_
   NUMBER)));
   return rootView;
}
```

En la primera línea del método, estamos diciendo que se utilizará como vista el *layout* definido en el fichero `fragment_main.xml` que se encuentra dentro del directorio `res/layout` del proyecto (recuerde que se procesa para que

forme parte de la clase *R* y así poder acceder de modo más sencillo). La segunda línea se encarga de obtener una referencia al *widget* (al objeto visual de tipo *View* y en concreto de tipo *TextView*) definido en el *layout* con identificador `section_label`; una vez obtenida la referencia se hace un *cast* para trabajar con el objeto recibido como si fuera de la clase *TextView* en lugar de *View* que es lo que devuelve la función `findViewById()`. La tercera línea establece el texto de la etiqueta que se mostrará por pantalla, que se obtiene de los parámetros de generación del fragmento. Por último se devuelve la vista generada.
Si ejecutamos el programa recién creado, tendremos tres vistas que van rotando según deslicemos el dedo horizontalmente por la pantalla, mostrando el índice de pantalla en el que nos encontramos.
Modificaremos el método para que quede:

```
public View onCreateView(LayoutInflater inflater, ViewGroup container,
                Bundle savedInstanceState) {
   View rootView = null;
   switch (getArguments().getInt(ARG_SECTION_NUMBER)){
      default:
         rootView = inflater.inflate(R.layout.fragment_main, container,
         false);
   }
   TextView textView = (TextView) rootView.findViewById(R.id.section_label);
   textView.setText(Integer.toString(getArguments().getInt(ARG_SECTION_
   NUMBER)));
   return rootView;
}
```

Simplemente hemos añadido una sentencia `switch()` que se utilizará para discernir que *layout* mostrar en cada momento. Ahora sólo queda ir generando los *layouts* a mostrar.

LinearLayout

Este es uno de los *layouts* más fáciles de usar, pero que funciona muy bien para hacer interfaces sencillas y para hacer bloques en las interfaces complejas. Este contenedor simplemente coloca los *widgets* uno detrás de otro según se declaren en el XML, pudiendo ser colocados vertical u horizontalmente dependiendo del parámetro `android:orientation`
Vamos a crear un nuevo *layout* en el proyecto creado anteriormente. para ello pulsamos con el botón derecho sobre el directorio `layout` y seleccionamos New>Layout resource file, lo que mostrará un diálogo con dos entradas File name para indicar el nombre del recurso y Root element para seleccionar el elemento raíz del *layout*; el primer campo lo rellenaremos con `linear_layout.xml` y en el segundo seleccionaremos `LinearLayout`.

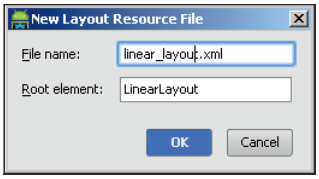

Figura 7.1. Creación de nuevo layout.

Advertencia:

El nombre de los ficheros de layout *no puede contener mayúsculas.*

Si seleccionamos y abrimos haciendo doble click, el recién creado fichero de *layout* para ver su contenido, en el panel central se abrirá una nueva pestaña con dos pestañas en su parte inferior. Si seleccionamos la pestaña Text veremos algo semejante a:

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

</LinearLayout>
```

En la parte derecha de la pantalla (su localización por defecto), aparecerá una previsualización de lo que se está diseñando en el *layout*, en caso de que no estuviera activa podemos abrir el panel nuevamente usando el botón situado abajo a la izquierda y seleccionando la opción Preview. La previsualización solamente está activa cuando se está trabajando sobre un fichero de *layout*, de lo contrario se oculta automáticamente. En capítulos posteriores se describen las opciones disponibles en este panel y como sacarle mayor provecho (véase figura 7.2.).
Vamos a variar el contenido del fichero de modo que quede:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:orientation="vertical"
   android:layout_width="match_parent"
   android:layout_height="match_parent">  >
   <TextView
      android:id="@+id/section_label"
      android:layout_width="match_parent"
      android:layout_height="wrap_content" />
   <TextView
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:text="@string/app_name "
```

```
      />
   <TextView
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:text="Otra etiqueta"
      />
</LinearLayout>
```

Figura 7.2. Panel de previsualización.

Lo que se ha hecho es añadir tres elementos de tipo *TextView* que se colocará una debajo de la otra, ya que estamos usando un *LinearLayout* y está configurado con el atributo `android:orientation="vertical"`. Se dejará una etiqueta con el identificador `section_label` para ver en tiempo de ejecución como varían las vistas (aprovechando parte del código que genera el asistente). Para comprobar el resultado mirar la previsualización o modificar el programa para que se muestre nuestro recién creado *layout*. La modificación que debe realizarse si se quiere probar en el programa, es añadir un caso al `switch()` que se ha creado anteriormente dentro de la función `onCreateView()` de la clase *PlaceholderFragment* en el fichero `MainActivity.java`:

```
switch (getArguments().getInt(ARG_SECTION_NUMBER)){
   case 1:
      rootView = inflater.inflate(R.layout.linear_layout, container, false);
      break;
```

```
    default:
        rootView = inflater.inflate(R.layout.fragment_main, container, false);
}
```

El resultado será:

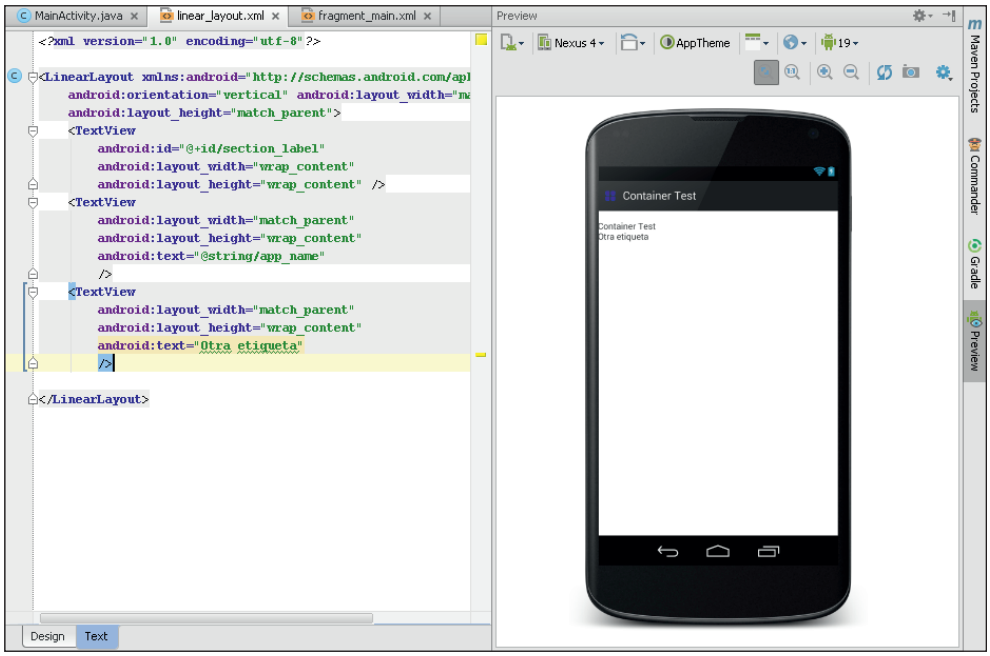

Figura 7.3. Elementos elementos en un LinearLayout en vertical.

Advertencia:

En layouts *complejos es posible que la previsualización no muestre el aspecto real o no muestre ningún aspecto. Esto puede deberse a que el* layout *dependa de varios archivos o condiciones que se tengan calcular en tiempo de ejecución.*

Si nuevamente modificamos el código del *layout* de manera que el atributo `android:orientation` del *LinearLayout* pase a ser `android:orientation="horizontal"`, el resultado será que el segundo y tercer *TextView* desaparecerán. Esto es porque el segundo *TextView* se ha puesto seguido horizontalmente del primer *TextView*, y éste tenía como atributo que tenía que ocupar horizontalmente todo lo que ocupara su padre (en este caso el container *LinearLayout*), por lo que el segundo *TextView* está fuera de pantalla y el tercero aún más allá. Ajustemos ahora el valor de la propiedad `android:layout_`

`width` del primer *TextView* y lo cambiaremos por `android:layout_width="wrap_content"`. Si visualiza de nuevo el *layout*, ahora sí que podrá ver las dos etiquetas (en modo previsualización, la primera etiqueta no tiene texto ya que se le proporciona durante la ejecución del programa, podemos añadir un texto cualquiera para probarlo, y durante la ejecución será sobrescrito). El hecho de que la segunda tenga la propiedad `android:layout_width="match_parent"` no importa para que se vea la primera etiqueta ya que el espacio de la primera etiqueta está reservado por estar definida antes y esta propiedad indica que ocupe, del espacio que le queda, todo lo posible hasta rellenar su contenedor horizontalmente.

Ya se ha visto colocar varios *widgets* uno detrás de otro y uno encima de otro mediante el *LinearLayout*, pero vamos a ver algunas propiedades más. Para ello pulsamos sobre la pestaña Design donde podemos editar de manera más sencilla el aspecto de nuestra vista. Si nos fijamos en la parte derecha aparece un panel llamado Component Tree donde se muestra la estructura jerárquica de los contenedores y *widgets* que forman parte del *layout*. Seleccionando cualquiera de ellos tanto en la previsualización como en el árbol disponible en el panel Component Tree, se resaltará el componente en la vista de previsualización y se mostrarán una serie de propiedades justo debajo del árbol. Es viable editar cualquiera de las propiedades desde este panel. A lo largo del diseño de la interfaz podemos utilizar tanto el editor gráfico como hacerlo en modo texto, intercambiando entre ellos, ya que se van sincronizando, pero lo que hay que tener cuidado cuando se modifica el código del *layout* manualmente es no "romper" la integridad del XML (por ejemplo dejar sin cerrar una etiqueta).

Como se puede observar, son muchas las propiedades de cada uno de los elementos que estamos usando en el XML, por lo que no entraremos en el detalle de cada una. A medida que se vayan utilizando a lo largo del libro, se irán explicando. No obstante existe una amplia documentación sobre las propiedades de cada elemento del interfaz en la propia web de Android `http://developer.android.com/reference/packages.html`.

Vamos a probar algunas propiedades más del *LinearLayout* y que están también presentes en otros componentes visuales. Para ello modificaremos directamente en la vista de propiedades y se podrá observar su cambio en el diseñador gráfico. Seleccione *LinearLayout* en el árbol del panel Component Tree situado a la derecha de la pantalla para que se muestren las propiedades del contenedor *LinearLayout*. Para modificar cada propiedad, vale con pulsar sobre la columna de la derecha correspondiente a la entrada a modificar en la vista Properties. Es muy recomendable que a la vez que se van modificando propiedades desde la vista Properties, se dirija al código para ver cómo se modifica y qué elementos XML corresponden a cada propiedad.

- **Background:** Permite fijar el color o la imagen de fondo del contenedor. Al modificarlo aparecerá en la columna de la derecha un botón con tres puntos suspensivos (los botones con puntos suspensivos aparecerán en todos los atributos que tengan ayuda para introducir su valor), al pulsar sobre él, un asistente nos ayudará para seleccionar una imagen a poner como fondo del contenedor o un color. En el caso del *Background*, se seleccionará el valor de la propiedad de entre los recursos disponibles en la aplicación. Para probar podemos elegir nuestro icono de aplicación, alguno de los recursos gráficos del sistema o algún color. El aspecto podría ser:

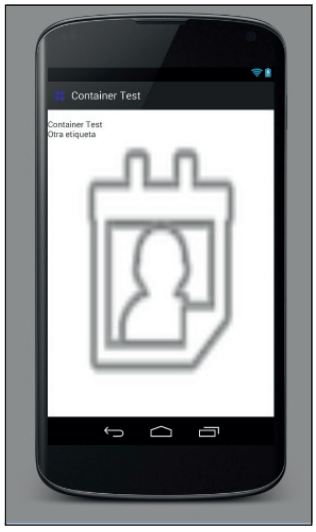

Figura 7.4. Resultado de establecer la propiedad Background.

- **Padding:** Es el espacio que dejará entre el borde y los elementos de su interior (sobre todos los elementos que contenga directamente). Existen variaciones de esta propiedad para cada uno de sus bordes (`padding bottom` para el borde inferior, `padding top` para el superior, `padding left` para la izquierda y `padding right` para la derecha). Si se modifica el atributo `padding top` y se le asigna el valor "10px" (sin las comillas) verá como las etiquetas se desplazan un poco hacia abajo (concretamente 10 pixeles). Esta propiedad sirve para dejar márgenes en los contenedores.
- **Layout Margin:** Es muy parecido a *Padding* sólo que en este caso se refiere al margen que dejara el *LinearLayout* por fuera de él, es decir respecto a los bordes de la pantalla. Al igual que en el caso del *Padding* existen variantes para cada uno de los bordes. Si se modifica el `layout margin right` y se le da el valor 40px se verá como se mueve el fondo de pantalla hacia la

izquierda, tal y como muestra la imagen. Si en lugar de modificar `layout margin right` modificáramos el `layout margin`, se aplicaría a todos los bordes a la vez.

- **Layout gravity:** Indica cómo debe actuar la gravedad (la atracción) respecto al *Layout*, es decir si al colocar un objeto se debe poner a la izquierda, centrado...

Con lo visto hasta ahora ya es posible crear interfaces algo más complejas a base de anidar contenedores del tipo *LinearLayout*. Veamos cómo sería por ejemplo un formulario para introducir un usuario. Buscamos tener un título centrado, con varias cajas de texto y un par de botones para guardar o cancelar. El XML podría ser:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:orientation="vertical"
   android:layout_height="fill_parent"
   android:layout_width="fill_parent">
<TextView
      android:id="@+id/section_label"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content" />
<TextView
      android:layout_height="wrap_content"
      android:text="Usuario" android:layout_width="fill_parent"
      android:gravity="center" android:textStyle="bold"
      android:textSize="18px" android:textColor="#FF0000"/>
<LinearLayout   android:orientation="horizontal"
      android:layout_height="wrap_content"
      android:layout_width="fill_parent">
<TextView
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Nombre: "
      />
<EditText android:text="" android:id="@+id/nameTxt" android:layout_
width="fill_parent" android:layout_height="wrap_content"></EditText>
</LinearLayout>
<LinearLayout
      android:orientation="horizontal"
      android:layout_height="wrap_content"
      android:layout_width="fill_parent">
<TextView
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Apellidos: "
      />
<EditText android:text="" android:id="@+id/surnameTxt" android:layout_
width="fill_parent" android:layout_height="wrap_content"></EditText>
</LinearLayout>
<LinearLayout     android:orientation="horizontal"
```

```
      android:layout_height="fill_parent"
      android:layout_width="fill_parent" android:gravity="bottom|center">
<Button android:text="Salvar" android:id="@+id/saveBtn" android:layout_
width="wrap_content" android:layout_height="wrap_content" ></Button>
<Button android:text="Cancelar" android:id="@+id/cancelBtn" android:layout_
width="wrap_content" android:layout_height="wrap_content" ></Button>
</LinearLayout>
</LinearLayout>
```

Y el resultado será:

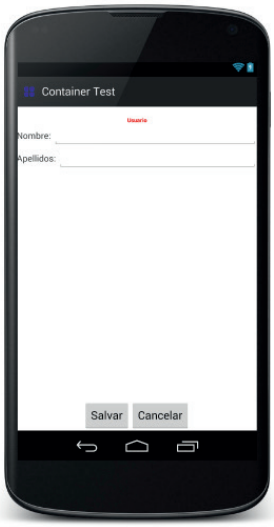

Figura 7.5. Pantalla de usuario realizada mediante LinearLayout.

Vamos a analizar un poco el código XML realizado. Lo primero que salta a la vista es que hay elementos que aún no se han visto y que hemos utilizado varios *LinearLayout* en lugar de solamente uno como hasta ahora.
Los nuevos elementos son dos `<EditText>` que mostrarán sendas cajas de texto editables y dos `<Button>` que representan a los botones.
La primera entrada dentro del *layout* corresponde a una etiqueta que nos servirá más adelante para conocer el número de vista en la que estamos en el programa.
Los *LinearLayout* han sido anidados, uno con alineación vertical y el resto con alineación horizontal para crear cada una de las líneas del formulario.
En el primer *TextView* se han aplicado unos nuevos atributos con tal de que quedara más interesante como título del formulario. Se ha centrado el texto haciendo que el *TextView* ocupara todo el ancho mediante `android:layout_width="fill_parent"` y centrando contenido del mismo a través de

`android:gravity="center"`. A la fuente de este *TextView* se le ha aplicado un estilo negrita con `android:textStyle="bold"`, se le ha fijado el tamaño mediante `android:textSize="18px"` y se ha usado el color rojo a través de `android:textColor="#FF0000"`.

Luego se pueden observar dos *LinearLayouts* prácticamente iguales para contener la información acerca del nombre y apellidos del usuario. De estos *LinearLayout* cabe destacar que están ajustados a contenido verticalmente y al contenedor padre horizontalmente de modo que cada uno de ellos cubre una línea de la pantalla en la cual se mostrarán los elementos que estén anidados en cada contenedor. Es muy importante que los *TextView* situados a la izquierda de los *EditText*, tengan su anchura fijada al tamaño de su contenido y no al tamaño del contenedor padre ya que de lo contrario los *EditText* no se mostrarían en pantalla.

El último *LinearLayout* se ha hecho que ocupe verticalmente toda la pantalla que queda libre para poder hacer que los botones se sitúen en la parte inferior de la pantalla, junto con la propiedad `android:gravity="bottom|center"` que hará que el contenido de este *LinearLayout* se centre y se sitúe en la parte inferior del mismo y centrada. Cuando se quiere dar más de un valor a una propiedad, como es el caso anterior, se separan todas ellas mediante el carácter pipe (|).

Hay que decir que para empezar no está mal… pero nosotros no queremos que no sólo no esté mal, sino que esté bien, así que vamos a variar un poco alguno de los elementos para acabar de ajustar un par de pequeños fallos que presenta la interfaz; veamos la manera de mejorar un poco el aspecto.

En primer lugar tanto las etiquetas como las cajas de texto se encuentran demasiado pegadas a los bordes de la pantalla. Se puede dar un poco de margen mediante un atributo ya conocido, añadiendo `android:padding="10px"` al *LinearLayout* principal (más adelante se verá que no es buena práctica usar como unidad de medida el pixel, pero por el momento es válido).

Separaremos también un poco el título del formulario, con tal de diferenciarlo de los de los elementos del mismo, añadiendo el `atributo android:paddingBottom="15px"` a la etiqueta *TextView* del título.

Normalmente entra mejor por los ojos cuando en una aplicación los elementos se encuentran perfectamente alineados y en esta interfaz las cajas de texto no se encuentran bien alineadas verticalmente. Aquí entra en juego un atributo que es nuevo `android:layout_weight`. Con este atributo se puede indicar cuanto "pesa" un elemento dentro de su contenedor. Sirve para indicar en qué modo se debe estirar o encoger un elemento para ocupar su espacio y el espacio se prorratea teniendo en cuenta el valor de este atributo. En caso de tener valor 0 no se modifica el elemento. En este *layout* tenemos dos con-

tenedores *LinearLayout* con un *TextView* y un *EditText* cada uno. Se puede indicar que tanto el *TextView* como el *EditText* intenten ocupar todo el espacio en horizontal que esté disponible haciéndolos "fill_parent" a ambos y luego prorratear el espacio mediante `android:layout_weight`. La entrada correspondiente al Nombre quedaría:

```
<LinearLayout
   android:orientation="horizontal"
   android:layout_height="wrap_content"
   android:layout_width="fill_parent">
<TextView
   android:layout_height="wrap_content"
   android:text="Nombre: " android:layout_width="fill_parent"
   android:layout_weight="3"/>
<EditText android:text="" android:id="@+id/nameTxt" android:layout_
width="fill_parent" android:layout_height="wrap_content" android:layout_
weight="1"></EditText>
</LinearLayout>
```

Los botones tampoco son del mismo tamaño así que podemos aplicar la misma técnica que acabamos de ver con tal de igualarlos. El nuevo código al completo quedaría:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:orientation="vertical"
   android:layout_height="fill_parent"
   android:layout_width="fill_parent" android:padding="10px">
<TextView
      android:id="@+id/section_label"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content" />
<TextView
   android:layout_height="wrap_content"
   android:text="Usuario" android:layout_width="fill_parent"
   android:gravity="center" android:textStyle="bold" android:textSize="18px"
   android:textColor="#FF0000" android:paddingBottom="15px"/>
<LinearLayout
   android:orientation="horizontal"
   android:layout_height="wrap_content"
   android:layout_width="fill_parent">
<TextView
   android:layout_height="wrap_content"
   android:text="Nombre: " android:layout_width="fill_parent"
   android:layout_weight="3"/>
<EditText android:text="" android:id="@+id/nameTxt" android:layout_
width="fill_parent" android:layout_height="wrap_content" android:layout_
weight="1"></EditText>
</LinearLayout>
<LinearLayout     android:orientation="horizontal"
   android:layout_height="wrap_content"
   android:layout_width="fill_parent">
<TextView
```

```
   android:layout_height="wrap_content"
   android:text="Apellidos: "
   android:layout_weight="3" android:layout_width="fill_parent"/>
<EditText android:text="" android:id="@+id/surnameTxt" android:layout_
width="fill_parent" android:layout_height="wrap_content" android:layout_
weight="1"></EditText>
</LinearLayout>
<LinearLayout     android:orientation="horizontal"
   android:layout_height="fill_parent"
   android:layout_width="fill_parent" android:gravity="bottom|center">
<Button android:text="Salvar" android:id="@+id/saveBtn" android:layout_
width="fill_parent" android:layout_height="wrap_content" android:layout_
weight="3"> </Button>
<Button android:text="Cancelar" android:id="@+id/cancelBtn" android:layout_
width="fill_parent" android:layout_height="wrap_content" android:layout_
weight="3"></Button>
</LinearLayout>
</LinearLayout>
```

En su nuevo aspecto se puede apreciar como los elementos quedan mejor integrados que anteriormente, simplemente aplicando unas pocas propiedades más.

Figura 7.6. Nueva pantalla de usuario realizada mediante LinearLayout.

Así mediante el *LinearLayout*, lo que se hace es dividir la pantalla en secciones que pueden ser dibujadas a través *layouts* verticales y horizontales e irla construyendo por secciones y hemos podido ver como cada elemento tiene unas propiedades fijas y otras que hereda de los padres que lo contienen y afectan a su posición tomando como referencia a estos. Todas las propiedades heredadas de los *layout* contenedores comienzan por `android:layout` y dependen de cada tipo de *layout* como iremos viendo en este capítulo.

TableLayout

Para conseguir el aspecto de la pantalla anterior se podría haber utilizado otro tipo de *layout*, por ejemplo el *TableLayout*. Para no perder la pantalla generada anteriormente y poder seguir experimentando con los archivos XML, lo que haremos es crear un nuevo *layout* sobre el que seguiremos haciendo pruebas. Existe un asistente para la facilitar creación de cualquier archivo de recurso XML y en este caso vamos a hacer uso de él. Nuevamente seleccionamos pulsamos con el botón derecho sobre la carpeta `src/main/res/layout` y en el menú contextual pulsamos sobre New>Layout resource file y en el diálogo resultante rellenamos el primer campo con `table_layout.xml` y en el segundo escribimos `TableLayout`.

Advertencia:

A la hora de nombrar los archivos de layout muy importante que escriba todo en minúsculas y que contenga la extensión **.xml**, *de lo contrario surgirán errores.*

Si abre el fichero del último *layout* creado, su código fuente es:

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

La primera diferencia con respecto al *LinearLayout* ya vemos que es que este *layout* no tiene el atributo `android:orientation` puesto que no hace falta. El *TableLayout* está pensado para colocar los elementos en disposición de filas y columnas y, aunque es posible tener como hijos otros elementos, lo normal es que el *TableLayout* tenga como hijos los elementos de tipo *TableRow*. Un *TableRow* representa una fila de la columna y cada elemento que esté contenido en el *TableRow* será una celda de la fila. Se podría pensar en el *TableLayout* como un *LinearLayout* vertical y en el *TableRow* como si fuera un *LinearLayout* horizontal y en cierto modo es así, pero ya veremos que no son exactamente iguales.

Cuando un *TableRow* se coloca dentro de un *TableLayout*, sus propiedades de anchura y altura son siempre `match_parent` y `wrap_content` respectivamente, es decir siempre ocupan de ancho lo mismo que su contenedor padre y de alto se ajustarán al contenido. El control sobre cómo se deben de mostrar cada una de las columnas recae directamente sobre el *TableLayout*. Existen dos parámetros llamados `android:stretchColumns` y

`android:shrinkColumns`. El primero sirve para indicar las columnas que se pueden estirar y el segundo las columnas que se pueden encoger. Esto permite que al tener que ocupar todo el ancho de la columna, podamos tener control de que columnas se estirarán o no con tal de completar todo el espacio. Las columnas se indican numéricamente empezando a contar desde el 0 y en caso de haber más de una columna, se indicarán separadas por comas, por ejemplo `android:stretchColumns="0,2,3"`. Igualmente, una columna puede ser marcada como que se puede estirar y encoger a la vez, dependiendo de lo que se necesite en cada caso. Para indicar que la propiedad se aplique a todas las columnas, en lugar de poner todos los números de columna, se puede escribir un asterisco `"*"`.

Los elementos de los *TableRow* poseen, entre otras propiedades, algunas para tener un control más preciso de su colocación en la tabla, por ejemplo y de modo semejante a HTML, un elemento puede ocupar más de una celda, para ello en el elemento que interese se debe poner el atributo `android:layout_span` o si queremos que se ubique en una columna en concreto, se hará mediante `android:layout_column`.

Para comprender mejor su funcionamiento vamos a ponernos manos a la obra. Cambie el código del fichero `table.xml` que acaba de crear por el siguiente:

```
<TableLayout
   xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="fill_parent"
   android:layout_height="fill_parent"    android:stretchColumns="1,2"
   android:shrinkColumns="1">
<TableRow android:id="@+id/TableRow01" android:layout_width="match_parent"
android:layout_height="wrap_content">
<Button android:text="Button01" android:id="@+id/Button01" android:layout_
height="wrap_content" android:layout_width="wrap_content"></Button>
<TextView android:id="@+id/section_label" android:layout_width="wrap_
content" android:layout_height="wrap_content" />
<Button android:text="Button03" android:id="@+id/Button03" android:layout_
width="wrap_content" android:layout_height="wrap_content"></Button>
</TableRow>
<TableRow android:id="@+id/TableRow02" android:layout_width="match_parent"
android:layout_height="wrap_content">
<Button android:layout_span="2" android:text="Button11" android:id="@+id/
Button11" android:layout_height="wrap_content" android:layout_width="wrap_
content"></Button>
<Button android:text="Button13" android:id="@+id/Button13" android:layout_
width="wrap_content" android:layout_height="wrap_content"></Button>
</TableRow>

<TableRow android:id="@+id/TableRow02" android:layout_width="match_parent"
android:layout_height="wrap_content">
<Button android:layout_column="1" android:text="Button22" android:id="@+id/
```

```
Button22" android:layout_width="wrap_content" android:layout_height="wrap_
content"></Button>
<Button android:text="Button23" android:id="@+id/Button23" android:layout_
width="wrap_content" android:layout_height="wrap_content"></Button>
</TableRow>
<TableRow android:id="@+id/TableRow02" android:layout_width="match_parent"
android:layout_height="wrap_content">
<Button android:text="Button31" android:id="@+id/Button31" android:layout_
width="wrap_content" android:layout_height="wrap_content"></Button>
<Button android:layout_column="2" android:text="Button33" android:id="@+id/
Button33" android:layout_width="wrap_content" android:layout_height="wrap_
content"></Button>
</TableRow>
</TableLayout>
```

Nuevamente para comprobar el resultado podemos mirar la previsualización o modificar el programa para que se muestre nuestro recién creado *layout*. La modificación que debe realizarse en el programa, es añadir otro caso al `switch()` que se ha creado y modificado anteriormente dentro de la función `onCreateView()` de la clase *PlaceholderFragment* en el fichero `MainActivity.java`:

```
switch (getArguments().getInt(ARG_SECTION_NUMBER)){
   case 1:
      rootView = inflater.inflate(R.layout.linear_layout, container, false);
      break;
   case 2:
      rootView = inflater.inflate(R.layout.table_layout, container, false);
      break;
   default:
      rootView = inflater.inflate(R.layout.fragment_main, container, false);
}
```

Si lo visualiza, el resultado es:

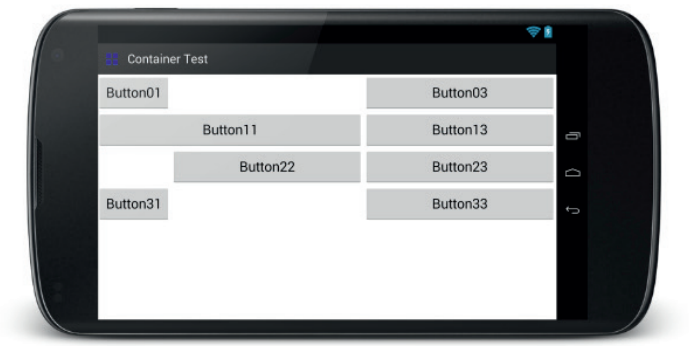

Figura 7.7. Resultado de la pantalla realizada con TableLayout.

Analicemos el código XML anterior. En la primera fila se han colocado dos botones y una etiqueta mediante un *TableRow*. Nótese que los espacios para la etiqueta `section_label` y el botón `Button03` son más grandes que el `Button01`. Esto es porque así lo hemos indicado mediante `android:stretchColumns="1,2"` donde se está diciendo que la columna 1 y 2 se pueden estirar con tal de cubrir todo el espacio en horizontal de la fila (hay que acordarse a la hora de numerar las columnas, que la numeración comienza en 0 no en 1).

Para ver de manera más clara cómo las columnas se estiran para conseguir ocupar toda la fila, podemos girar el dispositivo o dicho de otra manera poner en horizontal la pantalla. Si estamos haciendo las pruebas en el panel de previsualización, se puede cambiar la orientación mediante el tercer desplegable, siendo *Portrait* la pantalla en vertical y *Landscape* en horizontal. Si se está utilizando el emulador, éste se puede poner en posición horizontal mediante la combinación de teclas Ctrl-F11. Por último si se está probando en un dispositivo físico... supondremos que no hay que explicar.

Siguiendo con el análisis del código, en la segunda columna el primer elemento ocupa dos celdas, esto se consigue mediante el atributo `android:layout_span="2"` del elemento `<Button>`, que indica precisamente eso, que ocupe tantas columnas como valor tenga el atributo. Por último en las dos últimas filas, se ha forzado la colocación de un elemento mediante el parámetro `android:layout_column="2"`, es decir, se está forzando a que el elemento se coloque en la columna 2 (recuerde que se comienzan a contar las columnas desde 0).

Como puede ver, es *TableLayout* es otra manera muy rápida de diseñar pantallas colocando los elementos con mucho control sobre donde aparecerán, incluso si varía la orientación o el tamaño de la pantalla, ya que se adaptan a ella.

RelativeLayout

En este tipo de *layout* ya apareció en el primero de los ejercicios realizados, la posición de los elementos se calcula de manera relativa su elemento padre (por ejemplo colocar en la parte de abajo, o en el centro) o a la posición de los elementos hermanos (por ejemplo a la izquierda de un elemento, o debajo de éste). Cuando se maneja con soltura este tipo de *layout*, puede servir para aligerar mucho los diseños complejos de interfaces que con *LinearLayout* se complicarían en exceso.

Para comprender el funcionamiento se va a realizar la siguiente pantalla.

Figura 7.8. Pantalla a realizar mediante RelativeLayout.

Siguiendo los pasos realizados anteriormente para la creación del archivo de *layout* `table.xml`, crearemos uno nuevo llamándolo `relative_layout.xml` y que use un *RelativeLayout* como elemento base. Se modifica su código con el siguiente contenido:

```
<?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" >
   <TextView
   android:id="@+id/section_label"
   android:layout_alignParentLeft="true"
   android:layout_alignParentTop="true"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content" />

   <TextView
   android:id="@+id/label"
   android:layout_width="fill_parent"
   android:layout_height="wrap_content"
   android:layout_alignParentLeft="true"
   android:layout_below="@id/section_label"
   android:text="Un campo:" />

   <EditText
   android:id="@+id/entry"
   android:layout_width="fill_parent"
```

```
    android:layout_height="wrap_content"
    android:layout_alignParentLeft="true"
    android:layout_below="@+id/label"
    android:background="@android:drawable/editbox_background"/>

    <Button
    android:id="@+id/ok"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignParentRight="true"
    android:layout_below="@+id/entry"
    android:text="OK" />

    <Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignBaseline="@+id/ok"
    android:layout_toLeftOf="@+id/ok"
    android:text="Cancelar" />

    <TextView
    android:id="@+id/label2"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignParentLeft="true"
    android:paddingTop="10dp"
    android:layout_below="@+id/ok"
    android:text="Otro campo:" />

    <EditText
    android:id="@+id/entry2"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:layout_below="@+id/ok"
    android:layout_alignBottom="@+id/label2"
    android:layout_toRightOf="@+id/label2"
    android:paddingTop="10dp"
    android:background="@android:drawable/editbox_background"/>
</RelativeLayout>
```

Advertencia:

Aunque le aparezcan unas marcas amarillas en el código en la parte derecha, no se preocupe, esto es causado por Lint, un analizador de código al que volveremos en capítulos posteriores y que indica que el layout *es mejorable, por ejemplo extrayendo los textos. En capítulos posteriores conocerá cómo utilizarlo.*

Para este tipo de *layout* es muy importante poner identificadores a todos los elementos con tal de poder luego referenciar mejor los nuevos elementos. En este ejemplo el primer elemento definido en el *RelativeLayout* es un *TextView* que tiene como identificador `"android:id="@+id/section_`

label" y que nuevamente nos ayudara en ejecución a conocer el número de vista en el que nos encontramos. A partir de este elemento crearemos todo el *layout*. Después tenemos otro *TextView* con el identificador android:id="@+id/label" que está definido para que se coloque bajo el elemento anterior, esto se hace mediante la propiedad android:layout_below="@+id/section_label". El siguiente elemento es un *EditText* que también se colocará bajo el elemento anterior, nuevamente a través de la propiedad android:layout_below="@+id/label" . Al ser un elemento que ocupará todo el ancho de la pantalla, no hay que preocuparse más por este elemento.

Los dos siguientes elementos son dos botones, colocados bajo el campo de texto y uno al lado del otro. El primer botón que se crea es el de OK, que se le coloca bajo el *EditText* cuyo identificador es android:id="@+id/entry" y se alinea hacia la derecha del contenedor padre, esto se hace con los atributos android:layout_below="@+id/entry" y android:layout_alignParentRight="true".

Una vez posicionado el botón de OK, se crea el botón de Cancelar colocándolo a la izquierda del ya creado y posicionado botón de OK. También se debe cuidar su alineación y hacerlo horizontalmente con él botón OK ajustando sus bordes superiores. Mediante android:layout_toLeftOf="@+id/ok" se coloca a la izquierda del botón OK y mediante android:layout_alignBaseline="@+id/ok" se alinean.

Por último se ha añadido un *TextView* bajo el botón OK y un *EditText* al lado derecho de este nuevo *TextView*. Al *TextView* se le han configurado sus propiedades android:layout_below="@id/ok" para colocarlo bajo el botón OK y android:paddingTop="10dp". El atributo *paddingTop*, recordemos que lo que hace es establecer un pequeño espacio en blanco en la parte superior del elemento; podemos usar también el atributo *marginTop* que al igual que el anterior, dejaría un espacio en blanco, la diferencia estriba en el que el primero es un margen interno y el segundo un margen externo, pero ya lo iremos viendo. Al *EditText* se le ha configurado con las propiedades android:layout_toRightOf="@id/label2" para colocarlo a la derecha del *TextView* llamado "label2" y se le ha alineado con éste mediante android:layout_alignBaseline="@id/label2".La propiedad layout_toRightOf es semejante a la layout_toLeftOf salvo que en este caso lo coloca a la derecha del elemento indicado y la propiedad layout_alignBottom para obtener una alineación horizontal respecto a la parte baja del elemento. Con esta alineación, este elemento está recibiendo de manera indirecta el paddingTop definido en el elemento que se usa como referencia para la su colocación, que en este caso es el "label2".

Para probarlo en ejecución, se añade un nuevo caso al `switch()` de la función `onCreateView()` como se ha hecho anteriormente:

```
switch (getArguments().getInt(ARG_SECTION_NUMBER)){
   case 1:
      rootView = inflater.inflate(R.layout.linear_layout, container, false);
      break;
   case 2:
      rootView = inflater.inflate(R.layout.table_layout, container, false);
      break;
   case 3:
      rootView = inflater.inflate(R.layout.relative_layout, container,
false);
      break;

   default:
      rootView = inflater.inflate(R.layout.fragment_main, container, false);
}
```

Habrá podido observar que mediante "pon debajo de", "pon a la izquierda de", etc. se pueden realizar de modo rápido interfaces de usuario complejas. Como ejercicio puede intentar hacer este mismo *layout* usando contenedores *LinearLayout* y comparar el tamaño del XML resultante con el obtenido en este ejemplo.

AbsoluteLayout

Este será el *layout* que más le guste de todos los que hemos visto por su sencillez durante el diseño. En este *layout* podemos indicar el lugar exacto donde queremos que aparezca cada uno de los elementos añadidos, es decir, cuando se diseña, se puede arrastrar y soltar un elemento en el diseñador gráfico y el elemento se quedará en el sitio exacto donde lo soltemos, o dicho de otro modo, el elemento guardará sus coordenadas en la pantalla y se mostrará ahí … y ese es el problema, que siempre se mostrará ahí, por lo que cuando se use una pantalla mayor o cuando se gire la pantalla, la interfaz no se verá bien en la mayor parte de las ocasiones. Para mantener una buena compatibilidad visual entre distintos dispositivos y posiciones de pantalla, hay que evitar en lo posible utilizar este tipo de *layout*.

Creemos un nuevo fichero de *layout* de nombre `absolute_layout.xml` e intentemos generar un diseño semejante al que se ha realizado en para el *RelativeLayout*. En la pestaña Design podemos seleccionar de la parte izquierda los elementos que queremos añadir a nuestra pantalla y simplemente arrastrando y soltando los podemos ir colocando. Si no, podemos usar el siguiente código:

```
<AbsoluteLayout
   xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="fill_parent"
   android:layout_height="fill_parent">
   <TextView android:id="@+id/section_label" android:layout_height="wrap_
   content" android:layout_width="wrap_content" android:layout_x="1dip"
   android:layout_y="0dip"></TextView>
   <TextView android:id="@+id/TextView01" android:text="Un campo:"
   android:layout_height="wrap_content" android:layout_width="wrap_content"
   android:layout_x="1dp" android:layout_y="19dp"></TextView>
   <EditText android:layout_height="wrap_content" android:layout_
   width="fill_parent" android:id="@+id/EditText01" android:text=""
   android:layout_x="0dp" android:layout_y="40dp" android:background="@
   android:drawable/editbox_background"></EditText>
   <Button android:text="Ok" android:id="@+id/Button02" android:layout_
   height="wrap_content" android:layout_width="wrap_content" android:layout_
   x="309dp" android:layout_y="76dp"></Button>
   <Button android:text="Cancelar" android:id="@+id/Button01"
   android:layout_height="wrap_content" android:layout_width="wrap_content"
   android:layout_x="206dp" android:layout_y="76dp"></Button>
   <TextView android:id="@+id/TextView02" android:text="Otro campo:"
   android:layout_x="0dp" android:layout_height="wrap_content"
   android:layout_width="wrap_content" android:layout_y="131dp"></TextView>
   <EditText  android:layout_height="wrap_content" android:layout_
   width="match_parent" android:id="@+id/EditText02" android:text=""
   android:layout_x="83dp" android:layout_y="120dp" android:background="@
   android:drawable/editbox_background"></EditText>
</AbsoluteLayout>
```

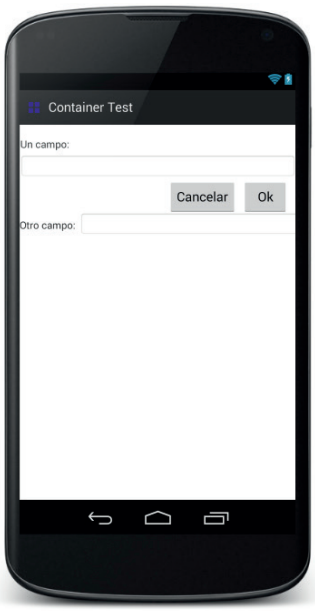

Figura 7.9. Pantalla realizada con AbsoluteLayout.

Lo más importante en este tipo de *layout* son las coordenadas de cada elemento, que vienen dadas mediante las propiedades `layout_x` y `layout_y` de cada uno de ellos.
No nos olvidemos de la etiqueta `section_label`,que aunque no se vea en la figura 7.9, está ahí en primera línea. Si no la ponemos, el programa fallará durante la ejecución, ya que intentará informarla con el número de vista en el que estamos y al no encontrarla dará un error.
Esta vez para probarla en ejecución no vale con variar solamente el método `onCreateView()`, ya que el programa está preparado para trabajar nada más con tres vistas. Comenzaremos añadiendo la entrada al `switch()` como otras veces y luego acabaremos de ajustar la aplicación.

```
switch (getArguments().getInt(ARG_SECTION_NUMBER)){
   case 1:
      rootView = inflater.inflate(R.layout.linear_layout, container, false);
      break;
   case 2:
      rootView = inflater.inflate(R.layout.table_layout, container, false);
      break;
   case 3:
      rootView = inflater.inflate(R.layout.relative_layout, container,
      false);
      break;
   case 4:
      rootView = inflater.inflate(R.layout.absolute_layout, container,
      false);
      break;

   default:
      rootView = inflater.inflate(R.layout.fragment_main, container, false);
}
```

Para adaptar el programa a las cuatro vistas que tenemos actualmente, se debe variar el método `getcount()` de la clase *SectionsPagerAdapter* de modo que refleje este hecho:

```
public int getCount() {
   // Show 4 total pages.
   return 4;
}
```

Al probarlo puede parecer que está todo correcto puesto que los elementos están allí donde los pusimos, pero ¿qué pasa cuando colocamos la pantalla en posición horizontal? Bueno, pues que los elementos siguen allí donde los pusimos, ninguno se recoloca, lo único que varía son los elementos definidos con anchura `"fill_parent"` que se estiran hasta conseguir ocupar todo el ancho de la pantalla, pero los botones se quedan en el mismo lugar, como descolgados. Y es que es este el problema de este *layout*, que al colo-

car mediante coordenadas los elementos, cuando el tamaño de la pantalla donde se dibujan dichos elementos cambia, se descuadra la composición, dejando espacios en blanco a la derecha o con elementos que no se visualizan por estar fuera de pantalla, por ejemplo si hubiéramos diseñado la interfaz de modo que existiera un botón abajo del todo pantalla mientras está en posición vertical, al pasar a posición horizontal desaparecerá ese botón por estar la coordenada correspondiente fuera de los límites de la pantalla en horizontal.

Atención:

El AbsoluteLayout *además de estar altamente desaconsejado por incompatibilidades, está obsoleto y abandonado por el equipo de desarrollo de Android.*

Otro aspecto a tener en cuenta es que cuando se utilice este tipo de *layout*, las coordenadas se las debemos dar en unidades dip (píxeles independientes de densidad), ya que si se las diéramos en unidades px (píxel), el efecto sería mayor. Más adelante se verá con más profundidad las distintas unidades con las que podemos con las que podemos especificar las medidas pero por el momento vamos a definir los dip (o dp) como unos píxeles independientes de la densidad de la pantalla, así, si las medidas las realizamos en dips en lugar de píxeles, no habrá tanta diferencia en el aspecto cuando un aplicación se muestre en una pantalla de alta densidad y una pantalla de baja densidad.

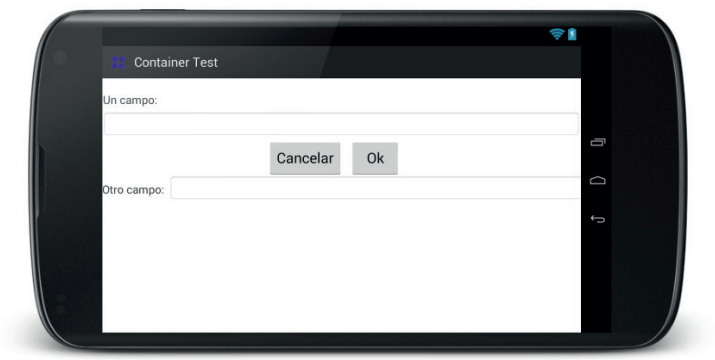

Figura 7.10. Pantalla en horizontal realizada con AbsoluteLayout.

FrameLayout

Este *layout* se comporta como una pila de elementos; según se van añadiendo elementos a él, se van colocando uno encima de otro, quedando ocultos los que están en la parte inferior.
Para probarlo vamos a generar un nuevo *layout* denominado `frame_layout.xml` con el siguiente contenido:

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="match_parent" android:layout_height="match_parent">
   <TextView
      android:id="@+id/section_label"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_gravity="center"/>

   <Button
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="texto muy largo"
      android:id="@+id/button"
      />
   <Button
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Corto"
      android:background="#ff0000"
      android:id="@+id/button"
      />
</FrameLayout>
```

En este caso al elemento *TextView* le hemos añadido la propiedad `android:layout_gravity="center"` para que se muestre en mitad de la pantalla y poder así ver el número de vista en el que nos encontramos. Los otros dos elementos son dos botones que nos ayudarán a comprender cómo funciona este tipo de *layout*. El primero de los botones tiene un texto más largo que el segundo de modo que nos aseguremos que sea más grande y así ver como el segundo botón cubre parte del primero; para acentuar la diferencia entre los botones, se ha añadido la propiedad `android:background="#ff0000"` al segundo botón; esta propiedad indica que tiene que tener un color rojo de fondo (la notación del color se ha dado en RGB aunque puede usarse aRGB si se quiere añadir un canal alfa).
Para poderlo probar modificamos nuevamente el código retocando el método `getcount()` de la clase *SectionsPagerAdapter*:

```
public int getCount() {
   // Show 5 total pages.
   return 5;
}
```

y el `switch()` del método `onCreateView()`:

```
switch (getArguments().getInt(ARG_SECTION_NUMBER)){
   case 1:
      rootView = inflater.inflate(R.layout.linear_layout, container, false);
      break;
   case 2:
      rootView = inflater.inflate(R.layout.table_layout, container, false);
      break;
   case 3:
      rootView = inflater.inflate(R.layout.relative_layout, container,
      false);
      break;
   case 4:
      rootView = inflater.inflate(R.layout.absolute_layout, container,
      false);
      break;
   case 5:
      rootView = inflater.inflate(R.layout.frame_layout, container, false);
      break;

   default:
      rootView = inflater.inflate(R.layout.fragment_main, container, false);
}
```

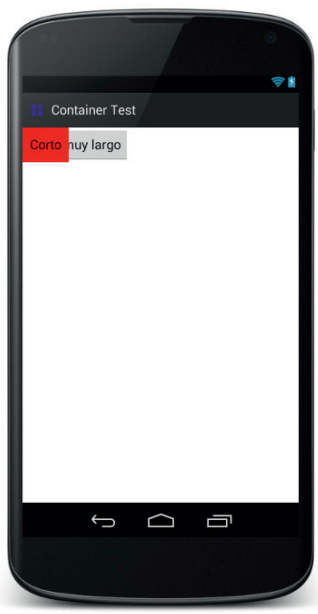

Figura 7.11. Pantalla realizada con FrameLayout.

GridLayout

Aunque existen otros *layouts* como el *DrawerLayout* o el *SlidingPaneLayout*, este será el último que comentemos en este capítulo. El *GridLayout* está disponible a partir del API 14, por lo que debemos asegurar que tengamos el proyecto preparado para ello. Se trata de un *layout* que muestra elementos colocados en una matriz de dos dimensiones. Se compone de una serie infinita de líneas que dividen la pantalla en una matriz generando las celdas en las que irán los elementos. Las celdas se referencias por los índices que ocupan siendo la propiedad de las líneas `android:layout_row` y la de las columnas `android:layout_column`. Hay que tener en cuenta que los índices se comienzan a numerar en 0.

El número de columnas y de filas viene dado por el valor de los atributos `android:columnCount` y `android:rowCount` respectivamente; estos atributos se definen dentro de la etiqueta `<GridView>`. En caso de que no se informen estos atributos, entonces el número de filas y columnas viene fijado por el índice más alto dado entre los elementos de la malla. Para aumentar el espacio entre los elementos en las celdas, se puede jugar con los atributos `margin` y `padding` o añadir elementos `<Space>` (que son elementos en transparentes) en las celdas intermedias.

Para ver su funcionamiento generaremos un nuevo *layout* denominado `grid_layout.xml` con el contenido:

```
<Listado> <GridLayout xmlns:android="http://schemas.android.com/apk/res/
android"
   android:layout_width="match_parent" android:layout_height="match_parent">
   <TextView
      android:id="@+id/section_label"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_row="0"
      android:layout_column="1" />
   <Button
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Botón 1 1"
      android:id="@+id/button"
      android:layout_row="1"
      android:layout_column="1" />
   <Button
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Botón 2 3"
      android:id="@+id/button2"
      android:layout_row="2"
      android:layout_column="3" />
```

```
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Botón 3 1"
        android:id="@+id/button2"
        android:layout_row="3"
        android:layout_column="1" />
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Botón 4 2"
        android:id="@+id/button2"
        android:layout_row="4"
        android:layout_column="2" />
</GridLayout>
```

Modificaremos los conocidos métodos para poder verlo en acción (última vez que los modificamos... prometido). El método `getcount()`:

```
public int getCount() {
    // Show 6 total pages.
    return 6;
}
```

y el `switch()` del método `onCreateView()`:

```
switch (getArguments().getInt(ARG_SECTION_NUMBER)){
    case 1:
        rootView = inflater.inflate(R.layout.linear_layout, container, false);
        break;
    case 2:
        rootView = inflater.inflate(R.layout.table_layout, container, false);
        break;
    case 3:
        rootView = inflater.inflate(R.layout.relative_layout, container, false);
        break;
    case 4:
        rootView = inflater.inflate(R.layout.absolute_layout, container, false);
        break;
    case 5:
        rootView = inflater.inflate(R.layout.frame_layout, container, false);
        break;
    case 6:
        rootView = inflater.inflate(R.layout.grid_layout, container, false);
        break;

    default:
        rootView = inflater.inflate(R.layout.fragment_main, container, false);
}
```

Podemos ver en el XML que cada elemento tiene asignadas las propiedades `android:layout_row` y `android:layout_column` para indicar su posición (también las hemos puesto como etiqueta del propio botón). En la imagen 7.12 vemos una columna libre a la izquierda; realmente no está libre, sino que está ocupada por la etiqueta `section_label`.

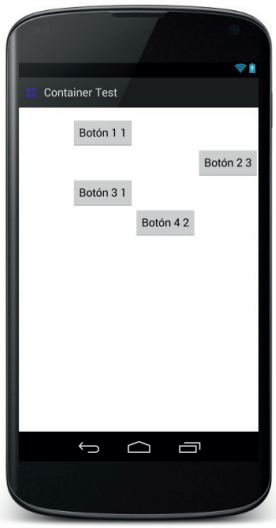

Figura 7.12. Pantalla con GridLayout.

En caso de necesitar que algún componente ocupe más de una fila o columna, se le puede informar las propiedades `android:layout_rowSpan` y `android:layout_columnSpan` respectivamente, proporcionando como valor el numero de celdas que debe ocupar.
Animo al lector a jugar con las propiedades de las filas y columnas para ver cómo se comporta este *layout*; por ejemplo modificando el botón situado en la fila 4 columna 2 para que esté en la fila 5, donde la fila 4 al no tener componentes se estirará.

Editor gráfico

Ahora que ya se ha visto como hacer los *layouts* mediante XML, vayamos a lo fácil, a conocer alguna de las opciones del editor gráfico. Este editor es una de las partes del SDK de Android que más ha variado y lo ha hecho para mejor. En las primeras versiones, era tedioso de utilizar y la previsualización que realizaba era de aquellas de "cualquier parecido con la realidad es pura coincidencia". En las nuevas versiones no sólo ayuda al diseño sino también a ver cómo quedará en bajo distintas configuraciones (versiones de Android, tamaños de pantalla, idiomas...).
En la parte izquierda de la pantalla se pueden ver una serie de persianas, que contienen perfectamente categorizados los diferentes objetos gráficos a utilizar en el diseño (Layouts, Widgets, TextFields...). Para añadirlos a la pantalla

simplemente hay que arrastrarlos a la representación de esta (justo en el panel derecho) o sobre el Component Tree de la derecha.

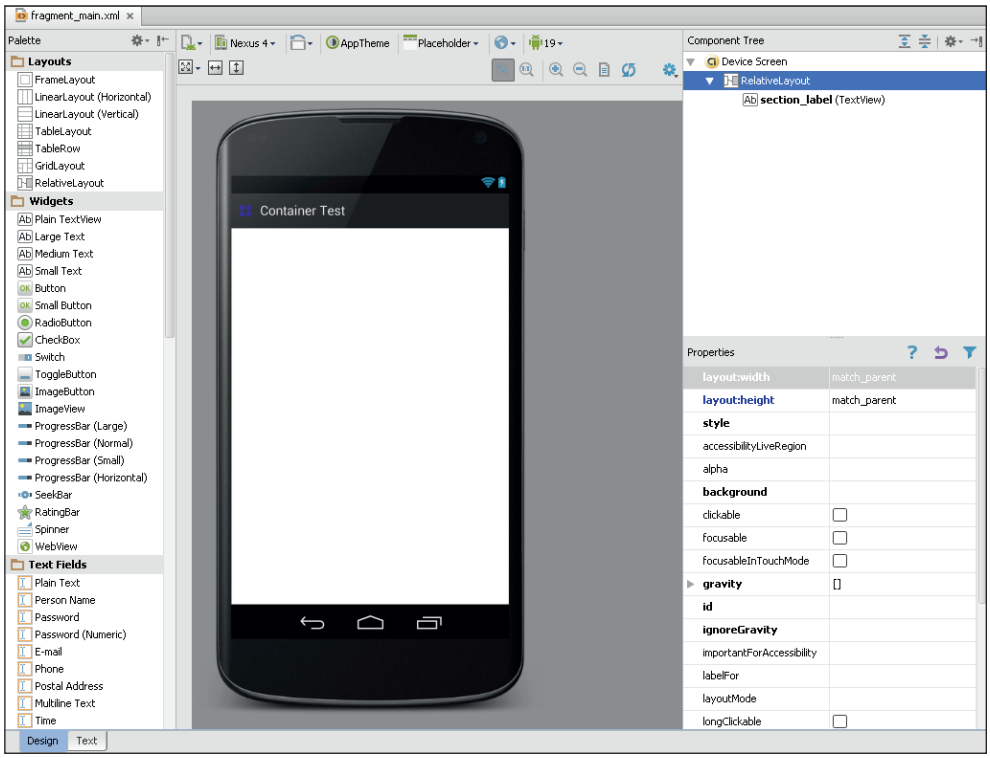

Figura 7.13. Vista general del editor gráfico.

En la parte superior de la pantalla de diseño gráfico podemos ver la barra de herramientas que permite ajustar ciertas propiedades de los elementos gráficos y sobre todo definir la previsualización que más interese.

Figura 7.14. Barra de configuración del editor gráfico.

El primer desplegable que encontramos en esta barra permite tener varias previsualizaciones a la vez en pantalla, de modo que podemos poner por ejemplo una en inglés y otra en español y ver las diferencias. Una opción muy interesante es generar una visualización en todos los tamaños de pantalla Preview

All Screen Sizes, que generará una entrada por cada tamaño de pantalla, y así de un vistazo poder ver su aspecto en diferentes modelos de terminales. Dentro de cada visualización podemos alejarla o acercarla, cambiar su nombre o eliminarla con los controles que aparecen cuando se pasa el ratón sobre ella.

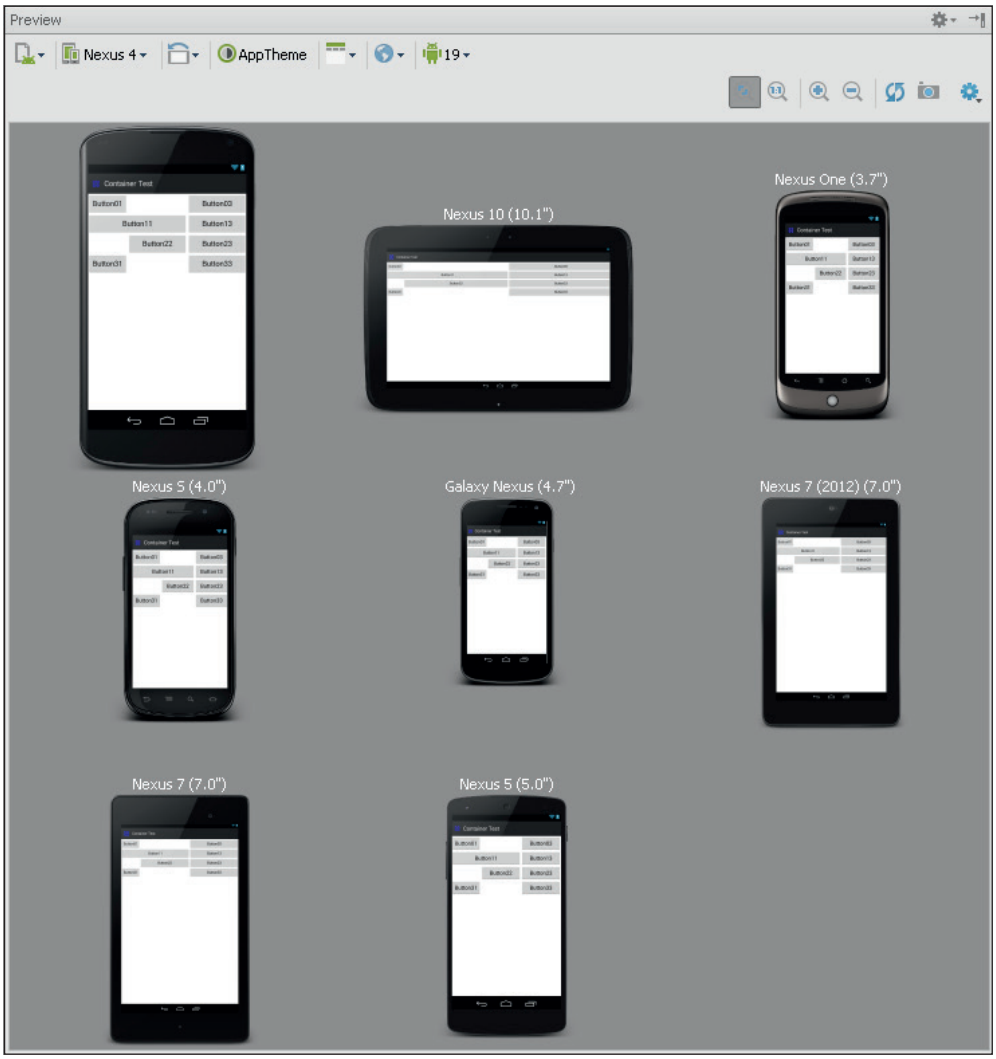

Figura 7.15. Varias visualizaciones simultáneas.

El segundo desplegable sirve para seleccionar el modelo de pantalla sobre el que trabajar, seleccionando tanto tamaño de pantalla como densidad de la misma. El tercer desplegable selecciona si queremos trabajar en vertical u ho-

rizontal, además de otros cambios como modo noche o si está enchufado en una base. El cuarto desplegable, es para seleccionar el estilo de tema con el que se vería. Además de los temas por defecto, si hubiéramos creado nosotros alguno (en capítulos posteriores los crearemos), también saldría. El quinto desplegable corresponde al control de clases, desde aquí se pueden cargar diferentes *layouts* en caso de trabajar con fragmentos incrustados o acceder a las clases java donde se trabaje con el *layout*. El siguiente desplegable que tiene una bola del mundo es para cargar distintos idiomas en la visualización, para utilizarlo previamente se han debido definir sus respectivos archivos de recurso. El último desplegable permite seleccionar la versión de Android sobre la que trabajar.

En la línea inferior de la barra aparecen ciertos botones divididos en dos grupos; los que están más a la izquierda cambian las propiedades más comunes de los objetos seleccionados, tales como altura, anchura, gravedad o márgenes y los de la derecha sirven para acercar y alejar las visualizaciones.

8 Interacción con la aplicación

En este capítulo aprenderá a:

- Controlar los eventos producidos en pantalla.
- Mostrar y ocultar elementos de pantalla mediante código.
- Realizar aplicaciones con múltiples pantallas compartiendo datos.
- Generar diferentes tipos alertas para el usuario.

Lo visto hasta ahora nos ha permitido ir definiendo interfaces de usuario para poder crear pantallas en las aplicaciones, pero por el momento, la única interacción que hemos tenido ha sido navegar entre dichas pantallas; es decir, que aún no se ha hecho código para responder a eventos.
En este capítulo se comenzará a ver cómo responder ante distintas acciones que el usuario aplique sobre la pantalla y a leer valores que se introduzcan en los *widgets*.

La caja de texto, la etiqueta y el botón

Con estos tres elementos será con los que comenzaremos a ver la programación de la interfaz y cómo hacer para que reaccionen a eventos. Haremos la típica aplicación donde se nos pida el nombre y tras dar a un botón se recupere el nombre introducido y salude mediante una etiqueta al nombre introducido… manos a la obra, vamos a ello.
Genere un nuevo proyecto Android con los siguientes valores (nuevamente, los valores que no se informan en esta lista, deben dejarse con el valor por defecto):

- Application name: Hola personalizado
- Module name: HolaTu
- Package name: com.acme.heyyou

Cuando el proyecto se haya terminado de generar, modificaremos el fichero `MainActivity.java` eliminando lo que no vamos a utilizar, de modo que quede:

```
package com.acme. heyyou;

import android.os.Bundle;
import android.support.v7.app.ActionBarActivity;

public class MainActivity extends ActionBarActivity {

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

En el método `onCreate()` simplemente hemos dejado la llamada al método `onCreate()` de la clase superior y la línea en la que se indica que *layout* vamos a utilizar en esta actividad, que en este caso es `R.layout.activity_main`.

En el archivo de *layout*, cambiaremos su contenido por:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:orientation="vertical" android:layout_width="fill_parent"
   android:layout_height="fill_parent">
   <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
      android:orientation="horizontal" android:layout_width="fill_parent"
      android:layout_height="wrap_content">
      <TextView android:id="@+id/textView" android:layout_width="wrap_
      content"
         android:layout_height="wrap_content" android:text="@string/name" />
      <EditText android:id="@+id/entry" android:layout_width="fill_parent"
         android:layout_height="wrap_content" android:background="@
         android:drawable/editbox_background" />
   </LinearLayout>
   <Button android:id="@+id/hello" android:text="@string/hello"
      android:layout_height="wrap_content" android:layout_width="wrap_
      content" android:layout_gravity="center"></Button>
   <TextView android:id="@+id/out" android:layout_width="wrap_content"
      android:layout_height="wrap_content"/>
</LinearLayout>
```

A estas alturas ya se conocen todos los elementos que aparecen en el *layout*. En un par de entradas aparecen unos textos que se toman del archivo `strings.xml`, que se encuentra en `src/main/res/values`, para modificarlo haga doble click sobre él y lo modificamos dejándolo.

```
<resources>
   <string name="hello_world">Hello world!</string>
   <string name="action_settings">Settings</string>
   <string name="app_name">Hola personalizado</string>
   <string name="hello">Hola </string>
   <string name="name">¿Cuál es su nombre?</string>
</resources>
```

Ya se tiene la interfaz creada. Abra el código de la clase *Activity* que gestionará el *layout*, es decir la clase *MainActivity* con tal de poder crear el código que permita interactuar con ella.

Para trabajar con los elementos de pantalla desde el código, lo primero que hay que hacer es recuperar el objeto que los representa y existe un método para ello `findViewById(identificador)`. Este método busca en el *layout* mostrado el elemento que tenga como identificador el que se haya pasado como parámetro y lo devuelve a modo de objeto. Si no encuentra ningún elemento con ese indicador, devolverá un nulo. Como se utiliza el mismo método para devolver cualquier tipo de *View* que esté definida en el *layout*, es tarea del programador realizar un *cast* a la clase correspondiente del objeto para poder acceder a los parámetros o métodos que sean necesarios, por ejemplo si lo que se recupera es un elemento imagen, le podremos indicar que imagen tiene que mostrar, pero si se recupera un elemento *TextView*, no se le puede dar una imagen como parámetro.

En el método `onCreate()`, tras las líneas de código existentes, será donde comenzaremos a recuperar los elementos y a darles funcionalidad.
El primer elemento que se recuperará es el botón para asignarle un código a ejecutar cuando se pulse. Se debe tener cuidado de guardar su referencia en un objeto del mismo tipo:

```
Button button = (Button)findViewById(R.id.hello);
```

Ya se tiene una referencia al botón definido en la interfaz de usuario, ahora se le puede registrar el código a ejecutar cuando sea pulsado; esto se realiza mediante un *listener* (de modo parecido a como se hace al trabajar con interfaces AWT o Swing):

```
button.setOnClickListener(new OnClickListener() {
      @Override
      public void onClick(View v) {
      // código a ejecutar cuando sea pulsado
//…
      }
      });
```

Cuando se pulse el botón lo que habrá que hacer es recuperar el elemento donde se está introduciendo el nombre del usuario y la etiqueta en la que se le enviará el saludo. Del primero se obtiene el nombre introducido y sobre el segundo se debe establecer el saludo. Para obtener el *EditText* procedemos de modo semejante a lo hecho con el botón:

```
EditText text = (EditText)findViewById(R.id.entry);
```

Una vez que se tiene el objeto *EditText*, ya se puede recuperar su valor. La clase *EditText* posee un método específico para ello, que será el que usemos:

```
String enteredName = text.getText().toString();
```

Una vez se tiene aislado en una variable de tipo *String* el valor que haya introducido el usuario en la caja de texto, se puede utilizar para saludarle. Aprovecharemos la cadena de texto utilizada en la etiqueta del botón como parte del saludo. Para recuperar este recurso nos valdremos de la clase *Resources*.

```
String salutation = getResources().getString(R.string.hello) + " " +
enteredName;
```

Y por último se debe escribir el saludo en el *TextView* de salida, así que se recupera y se establece el texto de la etiqueta:

```
TextView out = (TextView)findViewById(R.id.out);
out.setText(salutation);
```

El código completo del método onCreate es:

```
public void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
```

```
    setContentView(R.layout.activity_main);
    Button button = (Button)findViewById(R.id.hello);
    button.setOnClickListener(new View.OnClickListener() {
       @Override
       public void onClick(View v) {
          // código a ejecutar cuando sea pulsado
          EditText text = (EditText)findViewById(R.id.entry);
          String enteredName = text.getText().toString();
          String salutation = getResources().getString(R.string.hello) + " "
+ enteredName;
          TextView out = (TextView)findViewById(R.id.out);
          out.setText(salutation);
       }
    });
}
```

Si ejecuta la aplicación, puede escribir en la caja de texto superior y al pulsar sobre el botón aparecerá el saludo en la etiqueta inferior.

Figura 8.1. Saludo de la aplicación.

Vamos a añadir algún elemento más a la pantalla, por ejemplo una selección del tratamiento que el usuario espera en el saludo. La selección se hará mediante unos *radiobuttons*. El componente *RadioButton* se utiliza en aquellos casos en los que se tenga que elegir un valor entre varios, y el hecho de elegir ese valor signifique que no se puede elegir otro, por ejemplo el sexo de una persona, si es varón no puede ser mujer (nada de reflexiones metafísicas).

Para hacer bloques de *RadioButon* de modo que automáticamente cuando se seleccione uno de ellos se deseleccione el resto se utiliza el elemento *RadioGroup*. Colocaremos estos elementos en el archivo de *layout* justo encima de la definición del botón:

```
<RadioGroup android:id="@+id/RadioGroup01" android:layout_width="wrap_
content" android:layout_height="wrap_content" android:orientation="horizont
al">
<TextView android:text="@string/saludo" android:layout_width="wrap_content"
   android:layout_height="wrap_content"/>
<RadioButton android:text="@string/saludoSr" android:id="@+id/rdsr"
android:layout_width="wrap_content" android:layout_height="wrap_content"></
RadioButton>
<RadioButton android:text="@string/saludoSra" android:id="@+id/rdsra"
android:checked="true" android:layout_width="wrap_content" android:layout_
height="wrap_content"></RadioButton>
</RadioGroup>
<Button android:id="@+id/hello" android:text="@string/hello"
android:layout_height="wrap_content" android:layout_width="wrap_content"
android:layout_gravity="center"/>
```

Se crea un *RadioGroup* donde se le incluyen dos opciones, una para ser saludado como señor y otro como señora. Por defecto aparecerá seleccionado el segundo elemento, que es el correspondiente a la señora (discriminación positiva, que nadie se enfade). También hay que añadir las constantes correspondientes en el fichero `strings.xml`:

```
<string name="saludo">Saludo:</string>
<string name="saludoSr">Señor</string>
<string name="saludoSra">Señora</string>
```

Para obtener el *RadioButton* que se tiene seleccionado se puede hacer de dos formas distintas:

1. Obtener la referencia a cada elemento *RadioButton* y preguntarle si se encuentra seleccionado.
2. Obtener la referencia al elemento *RadioGroup* y preguntarle por el identificador del *RadioButton* que se encuentra seleccionado.

En caso de la primera opción, el código para obtener si el *RadioButton* correspondiente al saludo señor está seleccionado sería:

```
RadioButton radio = (RadioButton)findViewById(R.id.rdsr);
boolean estaSeleccionado = radio.isChecked();
```

Para el ejemplo se usará la segunda opción, obtener el *RadioGroup* y preguntarle por el elemento que tiene seleccionado. Modifique el código del evento `onClick()` para ajustarlo a:

```
// código a ejecutar cuando sea pulsado
EditText text = (EditText)findViewById(R.id.entry);
```

```
String salutation = null;
String enteredName = text.getText().toString();
// referencia al radioButton
RadioGroup radio = (RadioGroup)findViewById(R.id.RadioGroup01);
if (R.id.rdsr == radio.getCheckedRadioButtonId()){
      //para señor
      salutation = getResources().getString(R.string.saludoSr).toLowerCase();
}
else{
      salutation = getResources().getString(R.string.saludoSra).
      toLowerCase();
}
salutation = getResources().getString(R.string.hello) + " " + salutation + "
" + enteredName;
TextView out = (TextView)findViewById(R.id.out);
out.setText(salutation);
```

Se ha obtenido una referencia al elemento *RadioGroup* y mediante el método `getCheckedRadioButtonId()` se recupera el identificador de la entrada seleccionada. Se comprueba si el identificador seleccionado corresponde al del *RadioButton* del saludo del señor, y si es así se utiliza el saludo que se ha definido en las constantes, de lo contrario se utiliza el saludo para señora.
El hecho de utilizar el método `toLowerCase()`, es por cuestión de estética y no crear otra constante más, así se aprovecha la misma cadena para etiqueta del *RadioButton* que para el saludo.

Figura 8.2. Salida del saludo con opción de tratamiento.

Vamos avanzando y queremos más… vamos a añadir por ejemplo la posibilidad de mostrar una hora si es que el usuario así lo quiere. Se añadirá a la pantalla principal una casilla de selección donde el usuario podrá indicar si quiere introducir o no la fecha y hora. En caso de que quiera, se visualizarán unos componentes para tal efecto y se añadirá al mensaje de saludo.
Dentro del fichero de *layout*, introduzca las siguientes líneas entre el `<RadioGroup>` y el `<Button>`:

```
<CheckBox android:text="@string/showTime" android:id="@+id/checkBox"
android:layout_width="wrap_content" android:layout_height="wrap_content"></
CheckBox>
<TimePicker android:id="@+id/timePicker" android:layout_width="wrap_content"
android:layout_height="wrap_content" android:visibility="gone"></TimePicker>
<DatePicker android:id="@+id/datePicker" android:layout_width="wrap_content"
android:layout_height="wrap_content" android:visibility="gone"></DatePicker>
```

A las etiquetas `<TimePicker>` y `<DatePicker>` se les ha informado el atributo `android:visibility` como `"gone"` para hacer que no salgan en pantalla. Este atributo puede tener tres valores:

- **visible:** Se muestra en pantalla en la posición que le corresponde. Es el valor por defecto.
- **invisible:** El elemento se sitúa en la posición que le corresponde en la pantalla, no se muestra, pero guarda su sitio, es decir si tenemos en un *LinearLayout* vertical tres botones y al del medio se le pone el atributo como `"invisible"`, se verá un espacio en blanco entre el primer y tercer botón.
- **gone:** Es parecido al anterior pero en este caso el elemento no reserva su espacio. En el ejemplo de los tres botones que se ha explicado en el punto anterior, si se pusiera el atributo del segundo como `"gone"`, el efecto sería que el primer y tercer botón aparecerían uno seguido del otro, sin guardar espacio para el segundo.

Añadimos la constante cadena al fichero `strings.xml`:

```
<string name="showTime">Mostrar hora</string>
```

Ahora se debe ajustar el código para poder gestionar los nuevos elementos. En primer lugar se obtendrá la referencia al objeto *CheckBox* en el evento `onCreate()`, para poder controlar la visibilidad de los otros dos elementos.

```
CheckBox checkBox = (CheckBox)findViewById(R.id.checkBox);
```

Se le asignará un *Listener* con tal de que ejecute cierto código cada vez que su estado cambie, es decir que pase de seleccionado a no seleccionado o viceversa.

```
checkBox.setOnCheckedChangeListener( new CompoundButton.
OnCheckedChangeListener() {
@Override
```

```
   public void onCheckedChanged(CompoundButton buttonView, boolean
   isChecked) {
      int visibility = isChecked?View.VISIBLE:View.GONE;
      View view = findViewById(R.id.timePicker);
      view.setVisibility(visibility);
      view = findViewById(R.id.datePicker);
      view.setVisibility(visibility);
      }
   });
```

Dentro del método `onCkeckedChanged()` se controla el tipo de visibilidad que se quiere dar a los elementos para la toma del tiempo y la fecha, tomando como discriminador el parámetro `isChecked` que indica si la casilla de selección está marcada o no.

En la parte correspondiente al click del botón se debe programar la lógica para que dependiendo de si está o no está seleccionado la casilla, se muestre o no la hora y fecha. Dentro del método `onClick()` y justo antes de obtener la referencia al último *TextView*, se le añade:

```
// obtención de la hora y fecha
CheckBox timeCheckBox = (CheckBox)findViewById(R.id.checkBox);
if (timeCheckBox.isChecked()){
   DatePicker date = (DatePicker) findViewById(R.id.datePicker);
   String dateToShow = date.getDayOfMonth() + "/" + (date.getMonth() + 1) +
   "/" + date.getYear();
   TimePicker time = (TimePicker) findViewById(R.id.timePicker);
                  dateToShow += " " + time.getCurrentHour() + ":" + time.
                  getCurrentMinute();
   salutation += " " + dateToShow;
 }
```

En este código se obtiene la referencia a la casilla de selección y si está marcada, se extraen la fecha y la hora seleccionada por el usuario y se añaden al saludo. Hay que tener en cuenta que los meses se comienzan a numerar por el 0 en lugar de por el 1, por eso para obtener el mes se hace mediante `date.getMonth() + 1`.

El código completo del método `onCreate()` es:

```
public void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);
      Button button = (Button)findViewById(R.id.hello);
      button.setOnClickListener(new View.OnClickListener () {
         @Override
         public void onClick(View v) {
         // código a ejecutar cuando sea pulsado
         EditText text = (EditText)findViewById(R.id.entry);
         String salutation = null;
         String enteredName = text.getText().toString();
         // referencia al radioButton
         RadioGroup radio = (RadioGroup)findViewById(R.id.RadioGroup01);
         if (R.id.rdsr == radio.getCheckedRadioButtonId()){
```

```
        //para señor
        salutation = getResources().getString(R.string.saludoSr).
        toLowerCase();
        }
        else{
        salutation = getResources().getString(R.string.saludoSra).
        toLowerCase();
        }
        salutation = getResources().getString(R.string.hello) + " " +
        salutation + " " + enteredName;
        // obtención de la hora y fecha
        CheckBox timeCheckBox = (CheckBox)findViewById(R.id.checkBox);
        if (timeCheckBox.isChecked()){
           DatePicker date = (DatePicker) findViewById(R.id.datePicker);
           String dateToShow = date.getDayOfMonth() + "/" + (date.getMonth()
           + 1) + "/" + date.getYear();
           TimePicker time = (TimePicker) findViewById(R.id.timePicker);
           dateToShow += " " + time.getCurrentHour() + ":" + time.
           getCurrentMinute();
           salutation += " " + dateToShow;
        }
        // Salida del texto
        TextView out = (TextView)findViewById(R.id.out);
        out.setText(salutation);
        }
        });
        // Mostrar o no las fechas
        CheckBox checkBox = (CheckBox)findViewById(R.id.checkBox);
        checkBox.setOnCheckedChangeListener( new CompoundButton.
        OnCheckedChangeListener() {
     @Override
     public void onCheckedChanged(CompoundButton buttonView, boolean
     isChecked) {
        int visibility = isChecked?View.VISIBLE:View.GONE;
        View view = findViewById(R.id.timePicker);
        view.setVisibility(visibility);
        view = findViewById(R.id.datePicker);
        view.setVisibility(visibility);
        }
     });
  }
```

Si prueba ahora la aplicación, podrá utilizar la casilla de selección para mostrar y ocultar los nuevos elementos y añadirlos al saludo.

Pero ha podido surgir un pequeño problema dependiendo del tamaño de pantalla. Cuando se muestran los elementos para la selección de la fecha y la hora, hay tantos objetos en pantalla que no caben, y se está perdiendo el *TextView* donde mostramos el saludo por la parte inferior de la pantalla e incluso el botón. Para solucionar este pequeño inconveniente, usaremos en el *Layout* un elemento con desplazamiento llamado `<ScrollView>`, que es una vista que puede ser más grande que la propia pantalla y que se puede deslizar para

ver su contenido. El *ScrollView* jerárquicamente sólo acepta un hijo, pero éste puede ser todo lo complejo que se quiera. Para adecuarlo al ejemplo actual, simplemente valdría con tener de hijo el *LinearLayout* que ya se tiene definido. El *layout* con el *ScrollView* quedaría:

```
<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
android:id="@+id/ScrollView01" android:layout_width="match_parent"
android:layout_height="match_parent">
<LinearLayout
   android:orientation="vertical" android:layout_width="fill_parent"
   android:layout_height="fill_parent"
   >
   <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
      android:orientation="horizontal" android:layout_width="fill_parent"
      android:layout_height="wrap_content">
      <TextView android:id="@+id/textView" android:layout_width="wrap_
      content"
         android:layout_height="wrap_content" android:text="@string/name" />
      <EditText android:id="@+id/entry" android:layout_width="fill_parent"
         android:layout_height="wrap_content" android:background="@
         android:drawable/editbox_background" />
   </LinearLayout>
<RadioGroup android:id="@+id/RadioGroup01" android:layout_width="wrap_
content" android:layout_height="wrap_content" android:orientation="horizont
al">
   <TextView android:text="@string/saludo" android:layout_width="wrap_
   content"
      android:layout_height="wrap_content"/>
   <RadioButton android:text="@string/saludoSr" android:id="@+id/rdsr"
   android:layout_width="wrap_content" android:layout_height="wrap_
   content"></RadioButton>
   <RadioButton android:text="@string/saludoSra" android:id="@+id/
   rdsra" android:checked="true" android:layout_width="wrap_content"
   android:layout_height="wrap_content"></RadioButton>
</RadioGroup>
   <CheckBox android:text="@string/showTime" android:id="@+id/checkBox"
   android:layout_width="wrap_content" android:layout_height="wrap_
   content"></CheckBox>

<TimePicker android:id="@+id/timePicker" android:layout_width="wrap_content"
android:layout_height="wrap_content" android:visibility="gone"></TimePicker>
<DatePicker android:id="@+id/datePicker" android:layout_width="wrap_content"
android:layout_height="wrap_content" android:visibility="gone"></DatePicker>
<Button android:id="@+id/hello" android:text="@string/hello"
      android:layout_height="wrap_content" android:layout_width="wrap_
      content" android:layout_gravity="center"></Button>
<TextView android:id="@+id/out" android:layout_width="wrap_content"
      android:layout_height="wrap_content"/>
</LinearLayout>
</ScrollView>
```

Cuando no se pueda mostrar un elemento en la pantalla por problemas de espacio, ahora valdrá con desplazarla para que se pueda ver.

Figura 8.3. Saludo del programa con desplazamiento.

Otra pantalla por favor

Para ver el uso de otras actividades en la misma aplicación lo que haremos será mostrar el saludo en lugar de en la misma pantalla en una etiqueta, hacerlo en una pantalla completamente aparte.

Necesitaremos crear una nueva clase de tipo *Activity* y un *layout* para ella, la pantalla será tan sencilla como una etiqueta. Aunque podemos usar un asistente para crear una nueva actividad, ya lo veremos más adelante y en este caso lo haremos de modo manual; en el panel de la izquierda, vaya a la carpeta `/HolaTu/src/main/java/com.acme.holatu` y con el botón derecho del ratón pulse sobre ella ; en el menú emergente, seleccione la opción New>Java Class. En la caja de texto Name introduciremos el nombre de la clase que será *Salutation* y en la caja de texto Kind seleccionamos Class. Esta será la clase que haga de segunda actividad; más tarde volveremos a ella.

Por otro lado generamos el *layout* llamado `salutation.xml` con un `<TextView>` como elemento:

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

```
   android:layout_height="fill_parent">
   <TextView android:id="@+id/out" android:layout_width="wrap_content"
      android:layout_height="wrap_content"/>
</LinearLayout>
```

Antes de que se nos pase por alto, añadimos también la clase recién creada al fichero `AndroidManifest.xml`, ya que recuerde que de lo contrario fallará la aplicación. Lo escribimos tras la etiqueta `<activity>` que ya hay creada y dentro del elemento `<application>`:

```
…
</activity>
<activity android:name=".Salutation" />
</application>
```

Para lanzar la nueva actividad crearemos el código dentro del evento `click()` del botón. Lo que se hará será construir la cadena a mostrar y pasarla como parámetro a la nueva actividad. Cambiamos el código que mostraba el texto en la etiqueta de la actividad principal:

```
TextView out = (TextView)findViewById(R.id.out);
out.setText(salutation);
```

por uno encargado de hacerlo en la nueva actividad:

```
Intent intent = new Intent(MainActivity.this,Salutation.class);
intent.putExtra("salutation", salutation);
startActivity(intent);
```

Se instancia un objeto *Intent* para "intentar" lanzar la nueva actividad. Como sabemos qué *Activity* queremos que se lance, se le puede pasar como parámetro su nombre en lugar de una acción; también se le pasa como parámetro el contexto donde se debe ejecutar, que es dentro del proceso principal de la *Activity MainActivity*. Mediante el método `putExtra()` se le añaden una serie de parejas clave-valor que pueden ser recuperados más adelante en la nueva *Activity* que se lance; en este ejemplo lo que se le añade es la cadena de texto a mostrar y se hace bajo la clave "salutation". Se puede decir que esta es la manera de pasar parámetros a la nueva *Activity*. Más adelante se podrán recuperar todos estos parámetros en un *Bundle* mediante `getIntent()` (que retornará el objeto *Intent* que se configuró para llamar a la *Activity*) y luego mediante el método `getExtras()` (que devolverá el *Bundle* buscado). Por último se lanza la *Activity* mediante la llamada a `starActivity(intent)`, con el *Intent* que se ha configurado como parámetro.

Volvamos a la clase que se ha creado anteriormente. Esta clase debe ser una actividad, así que lo primero que se debe hacer es modificar para que extienda *android.app.Activity*.

```
public class Salutation extends Activity{
```

También debe tener un método `onCreate()` como todas las *Activity* que hemos visto. En él se debe especificar el *layout* a utilizar con la instrucción ya conocida:

```
setContentView(R.layout.salutation);
```

También en este método debe obtener el texto que se ha pasado como parámetro desde la *Activity* que ha iniciado el lanzamiento de esta nueva actividad; el parámetro estaba bajo la clave `"salutation"`. El objeto *Bundle* ofrece varios métodos dependiendo del tipo de dato buscado, por ejemplo:

```
String salutation = getIntent().getExtras().getString("salutation");
```

Y se debe mostrar en la etiqueta dispuesta a tal efecto:

```
TextView out = (TextView)findViewById(R.id.out);
out.setText(salutation);
```

El código completo de la clase *Salutation* quedará:

```
public class Salutation extends Activity {
   public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.salutation);
      String salutation = getIntent().getExtras().getString("salutation");
      TextView out = (TextView)findViewById(R.id.out);
      out.setText(salutation);
   }
}
```

Si se ejecuta la aplicación, al pulsar sobre el botón se mostrará la nueva actividad mostrando el texto de saludo (si todo ha ido bien). Para volver a la pantalla donde se selecciona el saludo vale con pulsar la tecla *back* (atrás) del dispositivo.

Te aviso: Alertas y tostadas

En ocasiones es necesario mostrar al usuario pequeños mensajes de alerta o de aviso, para los cuales no compensa hacer una pantalla completa. Por ejemplo en nuestro caso, queremos avisar al usuario si pulsa sobre el botón sin haber escrito nada en la caja de texto correspondiente al nombre. Android ofrece tres métodos para ello. Uno de ellos está más bien pensado para lanzar los mensajes cuando se producen en un servicio, es decir que no está la actividad que produce el mensaje como actividad con la que está interactuando el usuario. Las otras dos opciones se ajustan más a lo que se está buscando para este ejemplo y son mediante la clase *AlertDialog* y la *Toast* (tostadas). Tanto para mostrar el mensaje en un cuadro de diálogo como para mostrarlo mediante una *Toast* es necesario comprobar que no se ha informado el nombre. Esto se hace desde el

método `onClick()`, justo antes de comenzar a recuperar todos los datos restantes… si no ha introducido el nombre, no merece la pena seguir recuperando datos porque no se van a mostrar, luego al principio del código escribimos:

```
// código a ejecutar cuando sea pulsado
EditText text = (EditText)findViewById(R.id.entry);
// comprobar si existe nombre
if ("".equals(text.getText().toString().trim())){
// mostrar dialogo
showAlert();
//mostrar toast
      //showToast();
      return;
}
```

Si el texto introducido (y quitado los espacios en blanco, por si alguien intenta simplemente poner como nombre espacios) es vacío, entonces mostrar o bien el mensaje mediante un diálogo o mediante una *Toast*. Como es un ejemplo se han dejado anotados los dos métodos, pero sólo usaremos una a la vez; cuando quiera probar el mensaje diálogo debe descomentar el método `showAlert()` y comentar `showToast()` y hacerlo a la inversa si se quiere probar el mensaje "tostada". Incluiremos también un texto en el archivo `strings.xml` para mostrar en caso de que el usuario no haya completado el nombre:

```
<string name="noNameMsg">No se ha indicado nombre</string>
```

AlertDialog

La clase *AlertDialog* permite de una manera sencilla hacer visible un mensaje al usuario de modo que aparezca una pantalla sobre las demás con el mensaje indicado y que permanecerá allí hasta que el usuario pulse una tecla. Los diálogos se pueden configurar de varias maneras, por ejemplo seleccionando el número y tipo de botones, un título para el diálogo, funciones a ejecutar dependiendo del botón pulsado…
Lo primero es obtener una instancia de la clase *AlertDialog.Builder* que necesitará el contexto de la aplicación como parámetro:

```
AlertDialog.Builder alert = new AlertDialog.Builder(this);
```

Sobre este objeto ya se puede comenzar a configurar el aspecto y funcionalidad del diálogo, por ejemplo indicándole el texto a mostrar o estableciendo el botón a mostrar incluso con una función de *callback* a ejecutar en su pulsación:

```
alert.setMessage(text);
alert.setPositiveButton(android.R.string.ok, null);
```

Por último se debe mostrar mediante el método `show()`.El código del método que debe pertenecer a la clase *MainActivity* es:

```
protected void showAlert() {
CharSequence text = getResources().getString(R.string.noNameMsg);
   AlertDialog.Builder alert = new AlertDialog.Builder(this);
   alert.setMessage(text);
   alert.setPositiveButton(android.R.string.ok, null);
   alert.show();
}
```

Para los más atrevidos, se podría rescribir el método de modo más compacto como:

```
new AlertDialog.Builder(this).setMessage(getResources().getString(R.string.
noNameMsg)).setPositiveButton(android.R.string.ok, null).show();
```

Ya está todo preparado para que en caso de no informar el nombre del usuario, aparezca un mensaje mediante un cuadro de diálogo.

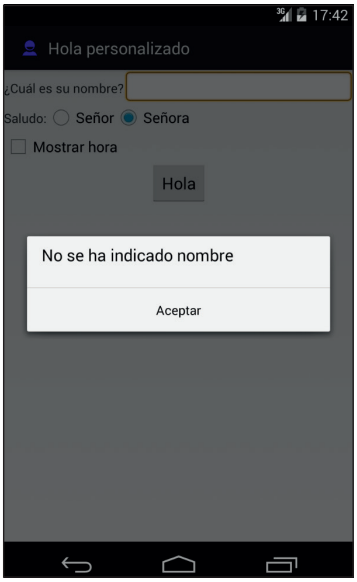

Figura 8.4. Pantalla de alerta.

Toast

La clase *Toast* ofrece los mecanismos para hacer visible un mensaje al usuario de modo que aparezca una pequeña cajita de texto sobre la pantalla y que permanecerá allí el tiempo que se le haya indicado mediante programación.

Se denomina mensaje tostada porque es como cuando las tostadas están listas, que saltan y se las puede ver durante unos instantes, y luego caen de nuevo desapareciendo.
Se debe crear un objeto *Toast* al que se le ha informado como parámetros el contexto de la aplicación, el texto a mostrar y el tiempo que se quiere mantener en pantalla:

```
Toast toast = Toast.makeText(context, text, duration);
```

Para finalizar y de igual modo al punto anterior, se debe hacer una llamada al método `show()`. El código completo del método que debe pertenecer a la clase *MainActivity* es:

```
protected void showToast() {
   Context context = getApplicationContext();
   CharSequence text = getResources().getString(R.string.noNameMsg);
   int duration = Toast.LENGTH_SHORT;
   Toast toast = Toast.makeText(context, text, duration);
   toast.show();
}
```

Igualmente, se puede rescribir el método para que sea más compacto:

```
Toast.makeText(getApplicationContext(),getResources().getString(R.string.
noNameMsg), Toast.LENGTH_SHORT).show();
```

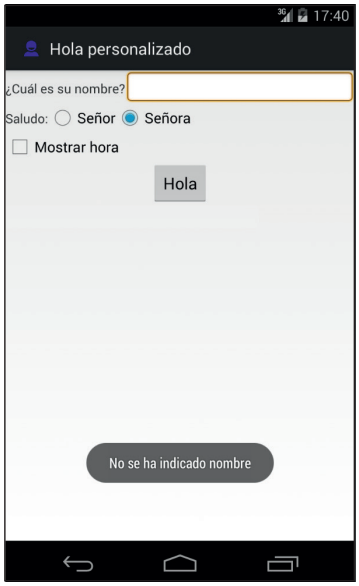

Figura 8.5. Toast con el aviso.

9

Flip: Un juego

En este capítulo aprenderá a:

- Reproducir archivos multimedia.
- Acceder a servicios del sistema como la vibración.
- Recuperar datos procesados por otra actividad.
- Crear menús de opciones.

Aunque aún no se ha visto como hacer animaciones o manejar gráficos, ya se tiene una base suficiente como para hacer una pequeña aplicación que será un juego de lógica. Se explorarán nuevas opciones como menús dentro de las actividades, obtener resultados tras la ejecución de una actividad, creación de interfaces dinámicos o el uso de permisos en el `AndroidManifest.xml`.

Reglas de juego

La idea es realizar un juego que en un principio tenga dos pantallas; la primera de ellas servirá para configurar la partida y una segunda que será la pantalla de juego. El objeto de esta aplicación no es enseñar como dibujar elementos en pantalla ni nada semejante, por lo que se utilizarán simples elementos *View* para su diseño.

El juego se trata de un tablero de NxM celdas, las cuales comenzarán teniendo distinto contenido y que cambiará según el usuario pulse sobre una u otra celda. El objetivo es dejar todas las celdas con el mismo contenido. La manera en la que cambiarán las celdas depende de la que se haya pulsado en cada momento. Respecto al contenido, éste siempre varía de modo cíclico, por ejemplo si se configura que el valor de la celda pueden ser tres números, y la celda comienza con el valor 1, cuando se pulse cambiará a 2, si se vuelve a pulsar cambiará a 3, si nuevamente se pulsa entonces su valor volverá a ser 1, luego 2… y así sucesivamente. Si solamente cambiara la celda que se pulsa, sería excesivamente sencillo, así que también cambiarán las de su alrededor. Supongamos un tablero de 3x3 celdas, en la figura 7.1 se muestra en rojo la celda pulsada (que cambiará su contenido) y en azul las celdas afectadas (que también cambiarán).

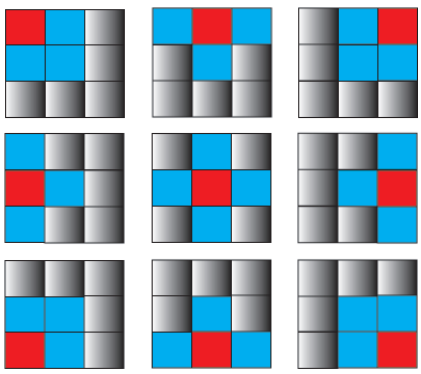

Figura 9.1. Esquema de movimientos para tablero 3x3.

Es decir, siembre cambiará el contenido de la celda pulsada y de todas las que tengan una arista común con ella. Además si la celda pulsada es una de las de las esquinas, además de todas las celdas indicadas anteriormente, también cambiará la que tenga en su vértice. Pues con estas reglas y con un contenido inicial de cada celda seleccionado al azar, habrá que dejar todas las celdas del tablero con el mismo contenido, por ejemplo que todas las celdas tengan el valor 2 ¿fácil? Habrá que demostrarlo cuando esté acabada la aplicación.

Pantalla inicial

Nada más entrar en la aplicación se mostrará la pantalla de configuración para la partida. Como parámetros de configuración se tendrá una serie de barras de desplazamiento para obtener el número de celdas que el usuario quiere tener en el tablero, unos botones para seleccionar el contenido de las celdas (que podrán ser números o colores por ejemplo) y unas casillas de selección para habilitar sonidos y vibraciones durante la partida. Todo ello lo colocaremos en un *LinearLayout* para obtener el aspecto mostrado en la figura 9.2.

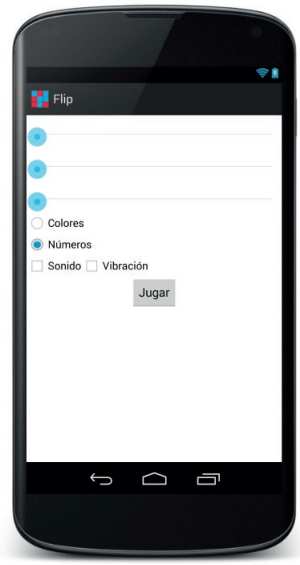

Figura 9.2. Aspecto de la pantalla de selección de partida en el diseñador.

Cree un nuevo proyecto con los siguientes parámetros (los que no están informados, nos sirven los que vienen por defecto):

- Application name: Flip
- Module Name: Flip
- Package name: com.acme.flip
- Activity Name: GameConfig

De igual modo que procedimos en el ejemplo anterior, una vez generado el proyecto, modificaremos el archivo `GameConfig.java` correspondiente a la actividad principal del programa, para dejarlo con lo justo para funcionar. La clase debe tener solamente el método:

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

Acto seguido modificamos el *layout* del archivo `activity_game_config.xml` (que es el archivo de *layout* principal) para obtener el aspecto mostrado en la figura 9.2. El código fuente es:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:orientation="vertical"
   android:layout_width="fill_parent"
   android:layout_height="fill_parent"
   >
   <!--  num celdas  -->
<TextView
   android:layout_width="fill_parent"
   android:layout_height="wrap_content"
   android:id="@+id/seekBarXtxt"
   />
<SeekBar android:id="@+id/seekBarX" android:layout_height="wrap_content"
android:max="6" android:layout_width="fill_parent"></SeekBar>
<TextView
   android:layout_width="fill_parent"
   android:layout_height="wrap_content"
   android:id="@+id/seekBarYtxt"
   />
<SeekBar android:id="@+id/seekBarY" android:max="6" android:layout_
height="wrap_content" android:layout_width="fill_parent"></SeekBar>
<TextView
   android:layout_width="fill_parent"
   android:layout_height="wrap_content"
   android:id="@+id/seekBarColorstxt"
   />
<SeekBar android:id="@+id/seekBarColors" android:max="5" android:layout_
height="wrap_content" android:layout_width="fill_parent"></SeekBar>
<!--  contenido -->
<RadioGroup android:id="@+id/radioGroup01" android:layout_width="wrap_
content" android:layout_height="wrap_content">
<RadioButton android:text="@string/radio_color" android:id="@+id/
```

```
radioButtonC" android:layout_width="wrap_content" android:layout_
height="wrap_content"></RadioButton>
<RadioButton android:text="@string/radio_number" android:id="@+id/
radioButtonN" android:layout_width="wrap_content" android:layout_
height="wrap_content" android:checked="true"></RadioButton>
</RadioGroup>
<!--  otras opciones -->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:orientation="horizontal"
   android:layout_width="fill_parent"
   android:layout_height="wrap_content"
   >
   <CheckBox android:text="@string/check_sound" android:id="@+id/
   checkBoxSound" android:layout_width="wrap_content" android:layout_
   height="wrap_content"></CheckBox>
   <CheckBox android:text="@string/check_vibrate" android:id="@+id/
   checkBoxvibrate" android:layout_width="wrap_content" android:layout_
   height="wrap_content"></CheckBox>
</LinearLayout>
<Button android:text="@string/startPlay" android:id="@+id/startBtn"
android:layout_width="wrap_content" android:layout_height="wrap_content"
android:gravity="center_horizontal|fill_horizontal|center" android:layout_
gravity="center_horizontal|center"/>
</LinearLayout>
```

Como elementos nuevos en este *layout* aparecen los `<SeekBar>` que son barras de desplazamiento que podremos utilizar para controlar el número de elementos y las tramas que deben mostrar. Dentro de estos elementos se han configurado sus valores máximos mediante el atributo `android:max`; el rango del *SeekBar* se establece entre 0 y este atributo.

Añadamos algo de código para controlar todos estos elementos. Abra el fichero `GameConfig.java` y diríjase al método on`Create()`. Prepararemos el botón que nos llevará al juego con las opciones indicadas; incluimos un método `startPlay()` para que más adelante podamos incluir su código de modo más sencillo, ya que ahora lo dejaremos sin rellenar.

```
// botón de inicio de partida
Button btn = (Button) findViewById(R.id.startBtn);
btn.setOnClickListener(new View.OnClickListener() {
   public void onClick(View view) {
      startPlay();
   }
});
```

El método `startPlay()` por el momento lo dejaremos en blanco y ya lo retomaremos más adelante, por lo que se define dentro de la clase *GameConfig* como:

```
protected void startPlay() {
// TODO Rellenar el código correspondiente
}
```

Sigamos introduciendo código al método `onCreate()`. Ahora nos encargaremos de la gestión de los *SeekBar*, comenzando por el que controla el número de celdas horizontales que tendrá el tablero. Se implementará el método `setOnSeekBarChangeListener()` de modo que cada vez que cambie el valor de la barra, se ejecute un código que actualice la etiqueta que muestra el valor actual seleccionado por ella. Este método necesita como parámetro un objeto del tipo *OnSeekBarChangeListener*, donde de los tres métodos que se deben sobrescribir, sólo daremos código al que nos interesa `onProgressChanged()`.

```
// control número de celdas horizontales
SeekBar xTiles = (SeekBar) findViewById(R.id.seekBarX);
xTiles.setOnSeekBarChangeListener(
   new SeekBar.OnSeekBarChangeListener() {
      @Override
         public void onProgressChanged(SeekBar seekBar,
      int progress, boolean fromUser) {
   updateXTiles(seekBar.getProgress());
         }
      @Override
      public void onStartTrackingTouch(SeekBar seekBar) {}
      @Override
      public void onStopTrackingTouch(SeekBar seekBar) {}
      });
```

Al igual que se hizo con el método `startPlay()`, dejamos preparado el método `updateXtiles()` dentro de la clase *GameConfig*, pero no se le da aún código.

```
private void updateXTiles(int progress) {
   // TODO Rellenar el código correspondiente
}
```

También estableceremos una llamada al método `updateXTiles()` dentro del método `onChange()`, para que actualice la etiqueta no sólo cuando varía el valor de la barra sino también cuando se crea. Para obtener el valor actual de la *SeekBar* se realiza mediante el método `getProgress()`, así pues añadamos la línea:

```
updateXTiles(xTiles.getProgress());
```

Ya tenemos la parte correspondiente a la configuración del número de elementos en horizontal del tablero dentro del método `onCreate()`. Se debe hacer lo mismo con las otras dos barras, la correspondiente a los elementos en vertical y el número de tramas, así que escribimos el código siguiente en el método `onCreate()` detrás del código ya creado:

```
// barra para las celdas verticales
SeekBar yTiles= (SeekBar) findViewById(R.id.seekBarY);
```

```
yTiles.setOnSeekBarChangeListener(
new OnSeekBarChangeListener() {
@Override
public void onProgressChanged(SeekBar seekBar,
      int progress, boolean fromUser) {
updateYTiles(seekBar.getProgress());
   }
@Override
public void onStartTrackingTouch(SeekBar seekBar) {}
@Override
public void onStopTrackingTouch(SeekBar seekBar) {}
      });
updateYTiles(yTiles.getProgress());
// barra para las tramas
SeekBar colors= (SeekBar) findViewById(R.id.seekBarColors);
colors.setOnSeekBarChangeListener(
new OnSeekBarChangeListener() {
@Override
public void onProgressChanged(SeekBar seekBar,
      int progress, boolean fromUser) {
updateColors(seekBar.getProgress());
   }
@Override
public void onStartTrackingTouch(SeekBar seekBar) {}
@Override
public void onStopTrackingTouch(SeekBar seekBar) {}
      });
updateColors(colors.getProgress());
```

e igualmente se crean los métodos de apoyo dentro de la unidad de compilación *GameConfig*.

```
private void updateYTiles(int progress) {
   // TODO Rellenar el código correspondiente
}

private void updateColors(int progress) {
   // TODO Rellenar el código correspondiente
}
```

Ya se tiene completado el método `onCreate()` y lo que faltaría es dar funcionalidad a estos métodos de apoyo que se han ido creando y que se ejecutarán cuando se modifiquen los valores de las barras de desplazamiento. Comenzaremos con el método `updateXtiles()`. Se debe recuperar la referencia al elemento del interfaz gráfico correspondiente a la etiqueta que mostrará el número de celdas en horizontal y establecer su valor. En el juego, el valor mínimo de celdas será 3 en horizontal y 3 en vertical, el problema es que las *SeekBar* comienzan a contar desde 0, por lo que cuando se muestre el valor de la *SeekBar* se le sumarán 3 unidades de modo que el valor mínimo serán 3 y el valor máximo será tres unidades más que el valor máximo establecido en el *layout* mediante el atributo `android:max`.

```
private void updateXTiles(int progress) {
      TextView tv = (TextView) findViewById(R.id.seekBarXtxt);
   tv.setText(getString(R.string.num_elem_x) + (progress + 3));
}
```

Y se realiza lo mismo para las otras dos barras:

```
private void updateYTiles(int progress) {
   TextView tv = (TextView) findViewById(R.id.seekBarYtxt);
   tv.setText(getString(R.string.num_elem_y) + (progress + 3));
}

private void updateColors(int progress) {
   TextView tv = (TextView) findViewById(R.id.seekBarColorstxt);
   tv.setText(getString(R.string.num_colors) + (progress + 2));
}
```

Establecemos los valores necesarios en el fichero `strings.xml`. Añada los siguientes valores a los ya existentes (ya que tenemos *layouts* creados por el asistente que usan los valores que actualmente tiene el fichero y si se borran daría error):

```
<resources>
    <string name="app_name">Flip</string>
    <string name="startPlay">Jugar</string>
    <string name="num_elem_x">Número elementos en eje X: </string>
    <string name="num_elem_y">Número elementos en eje Y: </string>
    <string name="num_colors">Número tramas: </string>
    <string name="radio_color">Colores</string>
    <string name="radio_number">Números</string>
    <string name="check_sound">Sonido</string>
    <string name="check_vibrate">Vibración</string>
   ...
```

Al ejecutar la aplicación, ya se pueden mover las barras y se verá como variarán los valores indicados por cada una de ellas en su etiqueta correspondiente. Por ahora dejaremos el método `startPlay ()` de la clase *GameConfig* sin código y ya lo retomaremos más adelante, cuando tengamos la aplicación más avanzada.

Menú

Existen tres tipos de menús dentro de las aplicaciones Android:

1. **Menú de opciones:** Es el menú principal de una *Activity* y aparece al apretar el botón físico "Menu" del terminal (en los terminales que lo tengan) o la tecla de menú en pantalla en los terminales actuales. Éste a su vez puede ser de varios tipos, dependiendo de versiones de Android y espacio en el *ActionBar* de las nuevas versiones.

2. **Menú de contexto:** Es un menú flotante que aparece cuando el usuario mantiene apretado durante unos instantes sobre una vista. Se denomina contextual porque depende del contexto donde se está pulsando. Se utiliza mucho en listas como mecanismo para mostrar opciones sobre el elemento seleccionado
3. **Submenú:** Se trata de una lista de elementos de menú que el usuario abre al apretar un elemento del menú tanto de un menú de opciones como de un menú de contexto. El submenú no soporta tener anidado otro submenú.

En esta aplicación ofreceremos un menú de opciones con tres entradas, una de ellas para ir a la configuración del jugador, otra para mostrar la ayuda de cómo jugar a Flip y la última será un "Acerca de", con información sobre el programador. Para definir los menús lo haremos como se ha hecho con otros elementos gráficos, a través de un fichero XML.
La manera en la que funciona el menú principal es la siguiente. Durante la creación de la *Activity*, se llama al método `onCreateOptionsMenu()`, donde transforma el fichero XML de configuración en las entradas del menú. Más adelante, durante la ejecución del programa, si el usuario pulsa alguna de las entradas del menú se ejecutará el método `onOptionsItemSelected()` y será donde se deberá discernir sobre la entrada en la que el usuario ha realizado la pulsación. Es posible que el lector se haya fijado, que cuando ha borrado los métodos de la clase *GameConfig* al principio del ejercicio, ha borrado este método y es que el asistente de generación de aplicaciones, también crea un menú y lo asigna a la aplicación.
El fichero creado lo encontramos en la carpeta `/src/main/res/menu`. Si se hubiera borrado o se necesitara uno nuevo, se puede crear pulsando sobre esa carpeta con el botón derecho del ratón y seleccionando el menú New>Menu resource file. Abramos el fichero generado, llamado `game_config.xml`, para modificarlo.
Por cada una de las entradas que se esperan en el menú, se necesita incluir un elemento `<item>` dentro del elemento raíz `<menu>`. Los atributos a definir serán:

- **id:** Es el identificador único de la entrada del menú. En tiempo de ejecución se usará para detectar la entrada que haya pulsado el usuario.
- **icon:** Icono a mostrar en el menú.
- **title:** Texto a mostrar junto al icono en el menú.

El código XML sería:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
   <item android:id="@+id/m_player"
```

```
      android:icon="@drawable/ic_player"
      android:title="@string/menu_player" />
   <item android:id="@+id/m_howto"
      android:icon="@drawable/ic_howto"
      android:title="@string/menu_howto" />
   <item android:id="@+id/m_about"
      android:icon="@drawable/ic_about"
      android:title="@string/menu_about" />
</menu>
```

Como se puede observar, se utilizan tres iconos (`ic_player`, `ic_howto`, `ic_about`) que deben estar dentro de alguno de los directorios *drawable* del proyecto, de no ser así, dará error al guardar el XML de definición del menú. Para agregar iconos al proyecto vale con copiarlos desde su gestor de ficheros del sistema operativo y pegarlos con el botón derecho sobre el directorio *drawable* que le interese; los formatos soportados son png, jpg y gif (como veremos más adelante, también podemos usar un asistente que nos generará las imágenes teniendo en cuenta las distintas resoluciones). Recuerde que los recursos son procesados y que el identificador del recurso se genera a partir del nombre del gráfico. Puede descargar los iconos utilizados de la web o utilizar los suyos propios (ajustando los nombres para que coincidan con los del fichero de configuración). A la de-recha del código del menú, podemos ir viendo como quedaría en el panel Preview.

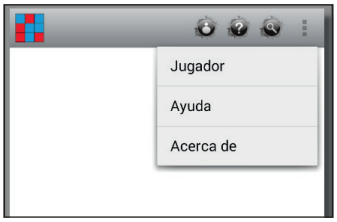

Figura 9.3. Visualización del menú.

En la parte de código Java se ha comentado que dentro de la clase *GameConfig*, durante su creación, se llamará al método `onCreateOptionsMenu()`, donde se deberá inflar (se denomina inflar, cuando se lee un fichero de texto y genera algún tipo de objeto gráfico, por ejemplo los *layout* también se inflan) el menú que se acaba de crear. Nos valdremos del objeto *MenuInflater* que no es más que un objeto capaz de leer un fichero de tipo menú dado por su recurso y convertirlo en un objeto de tipo *Menu* cuyos elementos serán los definidos en el fichero XML pasado como parámetro y que en este caso será utilizado como menú de opciones de la aplicación.

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

También hay que añadir ciertas constantes utilizadas en la definición del menú al archivo `strings.xml`. Aprovecharemos para añadir alguna más que después usaremos. El resumen de las entradas que hay que añadir es:

```
<string name="app_name">Flip</string>
<string name="startPlay">Jugar</string>
<string name="menu_howto">Ayuda</string>
<string name="menu_about">Acerca de</string>
<string name="menu_player">Jugador</string>
<string name="title_about">Acerca de Flip</string>
<string name="title_howto">Cómo jugar a Flip</string>

<string name="num_elem_x">Número elementos en eje X: </string>
<string name="num_elem_y">Número elementos en eje Y: </string>
<string name="num_colors">Número tramas: </string>
<string name="score_time">Tiempo: </string>
<string name="score_clicks">Pulsaciones: </string>

<string name="radio_color">Colores</string>
<string name="radio_number">Números</string>
<string name="check_sound">Sonido</string>
<string name="check_vibrate">Vibración</string>
<string name="game_end_1">Se finalizó el juego con un total de </string>
<string name="game_end_2"> pulsaciones</string>
```

Ya puede probar la aplicación y usar la tecla "Menu" del terminal o de la barra superior (en un capítulo posterior volveremos a ver cómo controlar los elementos de la barra superior) para ver como aparece el menú de opciones, pero ninguna entrada del menú responderá puesto que no se ha asignado aún código. Pongamos lógica al menú. El método `onOptionsItemSelected ()` tiene como parámetro un objeto del tipo *MenuItem* que representa a la entrada del menú que ha sido pulsada, por lo que dependiendo de ella se deberá ejecutar un código u otro. Mediante `ítem.getItemId()` se recupera el identificador buscado y es lo que usaremos para discernir:

```
@Override
   public boolean onOptionsItemSelected(MenuItem item) {
   switch(item.getItemId()) {
      case R.id.m_player:
         showPlayer();
         return true;
      case R.id.m_howto:
         showHowTo();
         return true;
      case R.id.m_about:
         showAbout();
         return true;
```

```
            }
            return super.onOptionsItemSelected( item);
        }
```

Los métodos `showPlayer()`, `showHowTo()` y `showAbout()` se declararán como métodos de la clase *GameConfig*, y por ahora dejaremos sin código `showPlayer()` para más adelante usarlo en otro capítulo. Respecto a los otros dos métodos, servirán para guiar al usuario a unas nuevas pantallas.

```
private void showAbout() {
   // TODO Rellenar el código correspondiente
}
private void showHowTo() {
   // TODO Rellenar el código correspondiente
}
private void showPlayer() {
   // TODO Rellenar el código correspondiente
}
```

Creamos las clases correspondientes a las actividades que acogerán cada una de las pantallas, dándoles como nombres `HowTo.java` y `About.java`. Estas clases se deben crear en el mismo paquete que `GameConfig.java`. Su contenido no presenta nada nuevo, simplemente cargarán un *layout*. El código de la clase *HowTo* será:

```
public class HowTo extends Activity {
   @Override
      public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.howto);
   }
}
```

Y respecto a la clase *About*:

```
public class About extends Activity {
@Override
   public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.about);
   }
}
```

Se deben crear también los *layout* que mostrarán cada una de estas pantallas. En la pantalla de ayuda se debería enseñar las reglas del juego y la pantalla de "Acerca de…" debería mostrar información acerca de la aplicación y el programador, pero como se trata de un ejemplo lo dejaremos con tan sólo una etiqueta con el texto directamente codificado para saber en qué pantalla nos encontramos, y se deja al lector como ejercicio realizar el contenido de estas vistas. El contenido del fichero de *layout* `howto.xml` es:

```
<LinearLayout
   xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="fill_parent"
   android:layout_height="fill_parent">
   <TextView
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:text="Ayuda"
   />
</LinearLayout>
```

Y el de `about.xml`:

```
<LinearLayout
   xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="fill_parent"
   android:layout_height="fill_parent">
      <TextView
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:text="Acerca de..."
      />
</LinearLayout>
```

Rellenamos los métodos que habíamos dejado apartados por el momento, para llevar al usuario a las pantallas recién creadas.

```
private void showAbout() {
   Intent i = new Intent(this, About.class);
   startActivity(i);
}
private void showHowTo() {
   Intent i = new Intent(this, HowTo.class);
   startActivity(i);
}
```

Y no olvide que para poder utilizar las nuevas *Activity*, hay que darlas de alta en el `AndroidManifest.xml`, dentro de la etiqueta `<aplication>` y tras la etiqueta `<activity>` ya existente. En esta ocasión aprovecharemos para usar el atributo `android:label` de las actividades que hará que se muestre ese texto como título de la actividad cuando esté en pantalla:

```
<activity android:name=".About" android:label="@string/title_about"/>
<activity android:name=".HowTo" android:label="@string/title_howto"/>
```

Ya tendremos terminado el menú de la aplicación.
Aunque en este ejemplo se ha realizado utilizando un archivo XML para la definición de los elementos del menú, también es posible hacerlo en tiempo de ejecución mediante programación. Para ello, por comodidad, se podrían definir constantes para la posición que ocuparía cada una de las entradas del menú:

```
private static final int HOWTO = Menu.FIRST ;
private static final int ABOUT = Menu.FIRST + 1;
```

y añadir los elementos al menú en el método `onCreateOptionsMenu()`

```
menu.add(0,HOWTO,0,R.string.menu_howto).setIcon(R.drawable.ic_howto);
```

En este momento, el menú ya es funcional.

Iniciando la partida

Antes de ir a configurar el tablero de juego, vamos a recuperar un método que se había dejado sin rellenar: `startPlay()`. Este método se tiene que encargar de recuperar de la interfaz los datos introducidos por el jugador y hacérselos llegar a la actividad que pintará el tablero con el que se jugará la partida. La manera de hacerlo se ha visto en un ejemplo anterior y es mediante el uso de un *Intent* al que se le añaden unos valores mediante parejas clave-valor. Lo primero es crear el *Intent* con la clase *Activity* a la que se quiere llamar. La clase en cuestión aún no existe (se crea en el punto siguiente) pero se llamará `GameField.java`, por lo que se invocará mediante:

```
Intent i = new Intent(this, GameField.class);
```

Tras ello, se irá cargando el *Intent* con todos los parámetros configurados por pantalla. Por ejemplo para recuperar el valor que se le ha dado al número de celdas en horizontal, se hará con:

```
SeekBar sb = (SeekBar) findViewById(R.id.SeekBarX);
i.putExtra("xtiles", sb.getProgress());
```

Para pasar parámetros al la nueva actividad lo hacemos mediante el *Intent* al cual se le van añadiendo estos parámetros usando el método `putExtra()`. Lo último que queda por hacer en este método es decirle a Android que comience la ejecución de la nueva *Activity*. En el ejemplo anterior se hizo a través de la llamada `startActivity(intent)`, en esta aplicación, interesa que cuando se acabe la actividad que se está llamando, retorne el resultado; o visto desde el punto de vista del juego, lo que se quiere es que cuando se acabe la partida, la actividad del tablero desaparezca y retorne el número de pulsaciones que ha realizado. Para lanzar una actividad de la cual interesa recuperar su resultado (a modo de llamada de una función que devuelve un resultado) se utiliza el método `startActivityForResult(intent, comando)`. En esta llamada se informa un objeto *Intent* del mismo modo que en la llamada anterior, pero se diferencia de ésta en que se debe informar también un comando a ejecutar. El comando es un entero que define el propio

programador y servirá para discernir del comando que se ha ejecutado cuando se vuelva de la actividad. Por ejemplo se puede utilizar una misma *Activity* para crear un usuario, modificarlo o visualizarlo; se invocaría con el mismo *Intent* pero con distinto comando, así cuando se retorne de la actividad, se podría mostrar un mensaje que se ha creado o se ha modificado el usuario, dependiendo del comando introducido; o si se llaman a dos *Activity* distintas, permite saber cuál ha sido llamada cuando regrese. Para este caso creamos un comando para lanzar la actividad que será un atributo de la clase *GameConfig*:

```
private static final int ACTION_PLAY = 1;
```

y el método `startPlay()` al completo sería:

```
private void startPlay() {
   Intent i = new Intent(this, GameField.class);
//configurar la partida
   SeekBar sb = (SeekBar) findViewById(R.id.seekBarX);
   i.putExtra("xtiles", sb.getProgress());
   sb = (SeekBar) findViewById(R.id.seekBarY);
   i.putExtra("ytiles", sb.getProgress());

   sb = (SeekBar) findViewById(R.id.seekBarColors);
   i.putExtra("numcolors", sb.getProgress());

   RadioButton r = (RadioButton) findViewById(R.id.radioButtonC);
   i.putExtra("tile", r.isChecked()?"C":"N");
   //control del sonido
   CheckBox chSound = (CheckBox) findViewById(R.id.checkBoxSound);
   i.putExtra("hasSound", chSound.isChecked());
   // control de la vibración
   CheckBox chVib = (CheckBox) findViewById(R.id.checkBoxvibrate);
   i.putExtra("hasVibration", chVib.isChecked());
   //comenzar activity
   startActivityForResult(i, ACTION_PLAY);
}
```

Cuando se retorna de la una actividad que se ha llamado mediante `startActivityForResult()` se ejecuta el método `onActivityResult()`. En él se tienen como parámetros el código de comando con el que se inició la actividad que ahora ha retornado, el código de retorno de la misma actividad, que servirá para informar de manera rápida si todo ha ido bien, o el usuario ha cancelado la actividad, o se ha producido un erro o cualquier otra eventualidad… y un objeto *Intent* al que se le ha podido rellenar su *Bundle*, desde la actividad que acaba de finalizar, con resultados del proceso de la actividad saliente. Para recuperar los datos simplemente se accede a su *Bundle*, llamado coloquialmente extras, y así acceder a la información mediante llamadas del tipo `data.getStringExtra("clave","pordefecto")` donde se le indica la clave del valor al que se quiere acceder y el valor por defecto en caso de que no exista ningún valor registrado bajo la clave in-

dicada. Por lo tanto si el código de ejecución devuelto por la actividad que retorna es correcto y el código de actividad con el que se llamó a la actividad tiene el valor de `ACTION_PLAY` (que se lo hemos asignado nosotros mismos) entonces mostraremos un mensaje de diálogo mostrando el resultado del número de pulsaciones de la partida. Añadimos el método a la clase *GameConfig*:

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent
data) {
if (resultCode == RESULT_OK){
switch ( requestCode){
   case ACTION_PLAY:
      new AlertDialog.Builder(this)            .setMessage(getResources().
getString(R.string.game_end_1) + data.getIntExtra("clicks", 0) +
getResources().getString(R.string.game_end_2))
               .setPositiveButton(android.R.string.ok,null)
               .show();
   break;
}
}
super.onActivityResult(requestCode, resultCode, data);
}
```

El tablero

El tablero de juego se compone de un número de celdas indefinido en tiempo de diseño de la aplicación, ya que dependiendo de la configuración de partida que elija el usuario se mostrarán más o menos celdas. Cada una de las celdas mostrará un contenido (que será el fondo de la celda) que irá cambiando si se pulsa dicha celda o alguna de su alrededor según las condiciones explicadas anteriormente.

Para representar cada una de las celdas aprovecharemos el botón típico de Android para extenderlo y darle algo más de funcionalidad, más concretamente lo aprovecharemos para que cada uno guarde su posición de celda del tablero y el contenido a mostrar en cada momento. Las coordenadas de colocación de cada celda en el tablero permanecerán inmutables durante la partida.

Las tramas a mostrar en las celdas serán o colores o números, y el número distinto de ellas viene dado por la configuración de la partida. Por ejemplo si el usuario selecciona números y 3 tramas distintas, los posibles contenidos de las celdas serán 1, 2 o 3, cambiando cíclicamente entre ellos.

Con estos datos creamos la clase `TileView.java` que representará cada una de las celdas con el siguiente código:

```
public class TileView extends Button {
// coordenadas
public int x = 0;
public int y = 0;
// trama a mostrar
private int index = 0;
//max tramas
private int topElements = 0;

public TileView(Context context, int x, int y, int topElements, int index,
int background) {
super(context);
   this.x = x; //coordenada X
   this.y = y; //coordenada Y
   this.topElements = topElements; //max tramas
   this.index = index; //índice de trama
   this.setBackgroundResource(background);
}
public int getNewIndex(){
index ++;
   //controlar si necesitamos volver a comenzar el ciclo de tramas
   if (index == topElements)index = 0;
   return index;
}
}
```

En el constructor se informarán las coordenadas que ocupa el botón/celda en el tablero, el número máximo de tramas que puede mostrar como contenido (que es definido por el usuario durante la configuración de la partida), el índice de la trama actual que debe mostrar de contenido y el identificador del recurso Android que se mostrará como contenido. Mediante el método `getNewIndex()` se volverá a calcular el nuevo índice de trama a mostrar. Si el número de tramas seleccionado es 3, y el índice actual es 2, cuando varíe el contenido lo hará al índice 3 que como es igual al número máximo de tramas a mostrar, debe volver al índice 0 (recuerde que las tramas se guardarán en unos *arrays* y que el contenido de estos se empieza a contar desde la posición 0).

Antes de ponernos manos a la obra con la clase *Activity* correspondiente al tablero, vamos a encargarnos de proporcionar al proyecto los gráficos y sonidos a utilizar en la partida. Como contenido de las celdas se usarán dos series de gráficos, unos numéricos y otros de colores llamados `ic_xc.png` para los colores y `ic_xn.png` para los números, siendo x un número entre 1 y 6, así tendremos los gráficos `ic_1c.png`, `ic_2c.png`... `ic_6c.png` para los colores (los colores se podrían definir mediante programación en lugar de usar un gráfico, pero para simplificar usaremos gráficos; más adelante ya usaremos programación) y de modo semejante para los números. Usaremos también un fichero de audio que haremos sonar cada vez que el jugador pulse una celda. Cree un nuevo directorio de nombre `raw` dentro de la carpeta `/src/`

main/res del proyecto. Dentro de esta nueva carpeta /src/main/res/raw guardaremos un archivo de sonido en formato mp3 y lo llamaremos touch.mp3. Todo este contenido puede ser descargado de la página web o usar los suyos propios.

Para el tablero usaremos un archivo de *layout* llamado gamefield.xml. En él se mostrarán las pulsaciones llevadas a cabo por el jugador, el tiempo que lleva jugando y por supuesto el propio tablero:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
    <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:gravity="center_horizontal|fill_horizontal">
    <Chronometer android:id="@+id/Chronometer" android:layout_width="wrap_
    content" android:layout_height="wrap_content" android:layout_
    gravity="right|fill_horizontal"></Chronometer>
    <TextView android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_marginLeft="100dip"
      android:id="@+id/clicksTxt" />
    </LinearLayout>
    <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:id="@+id/fieldLandscape"
    >
   </LinearLayout>
</LinearLayout>
```

El componente estándar <Chronometer> será el encargado de contar el tiempo que se lleva jugando. Su uso es muy sencillo, mediante código Java, una vez obtenida su referencia, se hace que comience a contar tiempo a través su método start() y se para de nuevo usando stop(). Entre estas dos llamadas, irá mostrando el tiempo transcurrido.

En el *LinearLayout* con identificador fieldLandscape será donde se vayan creando dinámicamente las celdas, que estarán agrupadas mediante otro *LínearLayout* por cada fila que se haya configurado la partida.

Ya estamos en disposición de afrontar la creación de la clase tablero. Genere una nueva clase que extienda la clase *Activity* y llámela GameField.java. Como atributos de la clase usaremos un *array* de constantes manteniendo los identificadores de los gráficos a mostrar cuando el usuario seleccione que quiere usar colores:

```
private static final int []colors = new int[]{
         R.drawable.ic_1c,
         R.drawable.ic_2c,
         R.drawable.ic_3c,
         R.drawable.ic_4c,
         R.drawable.ic_5c,
         R.drawable.ic_6c
   };
```

Usaremos otro *array* en caso de que seleccione usar los números:

```
private static final int []numbers = new int[]{
      R.drawable.ic_1n,
      R.drawable.ic_2n,
      R.drawable.ic_3n,
      R.drawable.ic_4n,
      R.drawable.ic_5n,
      R.drawable.ic_6n
   };
```

Y el resto de atributos:

```
//mantener el array que el usuario hay decidido utilizar
private int []pictures = null;
// Número máximo de celdas horizontales y verticales
private int topTileX =3;
private int topTileY =3;
// Número máximo de elementos a utilizar
private int topElements =2;
// Si ha seleccionado o no usar sonido y vibración
private boolean hasSound = false;
private boolean hasVibration = false;
// Array con los identificadores de las celdas cuando se añadan al layout,
// para poder recuperarlos durante la partida
private int ids[][] = null;
// Array para guardar los valores de los índices de cada una de las celdas.
// se utilizará para agilizar la comprobación de si la partida ha acabado o no.
private int values[][] = null;
// Contador con el número de pulsaciones que ha realizado el jugador
private int numberOfClicks=0;
// Para reproducir un sonido cuando el usuario pulse una celda
private MediaPlayer mp= null;
// Para hacer vibrar el dispositivo cuando el usuario pulse una celda
private Vibrator vibratorService = null;
// Mostrará en pantalla el las veces que el usuario ha pulsado una celda
private TextView tvNumberOfClicks = null;
```

El método `onCreate()` debe establecer el *layout* a utilizar, que será el recientemente creado `gamefield.xml`:

```
setContentView(R.layout.gamefield);
```

Para animar un poco la partida, se utilizará una referencia al servicio de vibración del sistema añadiéndola también al `onCreate()`:

```
vibratorService = (Vibrator)(getSystemService(Service.VIBRATOR_SERVICE));
```

y se creará el objeto *MediaPlayer* que reproducirá los sonidos con el recurso de audio a reproducir:

```
mp = MediaPlayer.create(this, R.raw.touch);
```

También se recuperará la referencia al *TextView* de la interfaz que mostrará el número de pulsaciones:

```
tvNumberOfClicks = (TextView) findViewById(R.id.clicksTxt);
```

Esta *Activity* es lanzada desde la *Activity* principal del programa, donde se habían configurado ciertos parámetros de la partida; los parámetros se habían introducido en un objeto *Bundle* para con tal de hacerlos accesibles a la actividad lanzada. Obtenemos estos parámetros para configurar la partida mediante los métodos del objeto *Bundle* ya conocidos:

```
Bundle extras = getIntent().getExtras();
topTileX = extras.getInt("xtiles") + 3;
topTileY = extras.getInt("ytiles") + 3;

topElements = extras.getInt("numcolors") + 2;
// usar colores o números
if ("C".equals(extras.getString("tile"))){
   pictures = colors;
}
else{
   pictures = numbers;
}
hasSound= extras.getBoolean("hasSound");
hasVibration= extras.getBoolean("hasVibration");
```

Para evitar que entre partidas se mantengan las celdas anteriores, limpiamos el *LinearLayout* que las contiene:

```
LinearLayout ll = (LinearLayout) findViewById(R.id.fieldLandscape);
ll.removeAllViews();
```

Cuando se creen las celdas para añadirlas a la vista, se les debe proporcionar un ancho y un alto. Esto se podría realizar jugando con los pesos de los tamaños de cada botón dentro del *LinearLayout*, pero lo haremos teniendo en cuenta las dimensiones de la pantalla. En el cálculo del ancho y alto que le corresponde a cada celda, se tiene que tener en cuenta el ancho y alto de la pantalla y el número de celdas que hay en cada sentido. El tamaño de la pantalla se obtiene a través del objeto *DisplayMetrics* que es informado mediante el método `getMetrics()`. Para el cálculo de la altura se tiene que tener en cuenta que en la parte superior se tiene entre otras cosas el marcador de pulsaciones y la barra inferior. Más adelante se enseñará cómo hacer una aplicación a pantalla completa con lo que no tendríamos que realizar estos cálculos, pero por el momento lo haremos así.

```
DisplayMetrics dm = new DisplayMetrics();
getWindowManager().getDefaultDisplay().getMetrics(dm);
int width = dm.widthPixels / topTileX;
int height = (dm.heightPixels - 180) / topTileY;
```

Se inicializan los *arrays* para guardar los identificadores en pantalla de las distintas celdas y los índices de trama de cada una de ellas:

```
ids =  new int [topTileX][topTileY];
values =  new int [topTileX][topTileY];
```

Las tramas que se mostrarán en las celdas al iniciar la partida, deben ser aleatorias y siempre dentro de los márgenes que el jugador haya configurado, si se configura que como máximo se muestren dos tramas, no se puede mostrar la tercera trama. Se inicializa el objeto *Random* con la fecha y hora del sistema en milisegundos, y se calcula un entero entre cero y el número máximo de tramas. Después se volverá a calcular otro entero aleatorio por cada celda que se añada al tablero:

```
Random r = new Random(System.currentTimeMillis());
int tilePictureToShow = r.nextInt(topElements);
```

Mediante dos bucles anidados se irán creando las celdas del tablero. Se tienen que crear tantos *LinearLayout* como filas tenga el tablero, es decir, tantos como elementos en el eje Y haya configurado el jugador. Del mismo modo, cada *LinearLayout* tendrá tantas entradas como elementos en el eje X estén configurados. Por cada celda que se cree, se debe guardar su posición, su trama mostrada, darle un identificador y guardarlo, configurar su altura y anchura, asignarle un código a ejecutar cada vez que se pulse y por ultimo y que no se nos olvide, añadirlo a la vista padre correspondiente.

```
int ident = 0;
for (int i = 0; i < topTileY ; i++ ){
   LinearLayout l2 = new LinearLayout(this);
   l2.setOrientation(LinearLayout.HORIZONTAL);
   for (int j = 0; j < topTileX ; j++ ){
      tilePictureToShow = r.nextInt(topElements);
      // guardamos la trama a mostrar
      values[j][i]= tilePictureToShow;
      TileView tv = new TileView(this,j,i,topElements, tilePictureToShow,
pictures[tilePictureToShow]);
      ident++;
   // se asigna un identificador al objeto creado
      tv.setId(ident);
      // se guarda el identificador en una matriz
      ids[j][i]= ident;
       tv.setHeight(height);
      tv.setWidth(width);
      tv.setOnClickListener(new View.OnClickListener() {
         public void onClick(View view) {
            hasClick(((TileView)view).x, ((TileView)view).y);
```

```
            }
        });
        l2.addView(tv);
    }
ll.addView(l2);
}
```

En el método `onClick()` de cada celda, se hace una llamada a `hasClick()` pasando como parámetros las coordenadas de la celda pulsada, y habrá que cambiar su contenido y el de las celdas que le rodean. Los cálculos de las celdas que deben variar se harán en esta llamada.
Para acabar con el método `onCreate()`, se pondrá en marcha el contador de tiempo:

```
Chronometer t = (Chronometer)findViewById(R.id.Chronometer);
t.start();
```

Como hemos dicho, en el método `hasClick()` de la clase *GameField*, se deben hacer los cambios del contenido de la celda pulsada, calcular las celdas (que además de la pulsada) deben cambiar y cambiarlas. La identificación de la celda pulsada se hace pasándole como parámetros sus coordenadas:

```
public void hasClick(int x, int y){}
```

Como se provocarán cambios en el contenido de las celdas, también se debe controlar si se ha finalizado la partida o no, es decir si tienen todas las celdas el mismo contenido. También en este método se tiene que comprobar si se ha configurado la partida para reproducir sonidos y/o vibrar y en tal caso hacerlo. Comenzamos a incluir código en el método `hasClick()`:

```
if (hasVibration) vibratorService.vibrate(100);
if (hasSound) mp.start();
```

Para no complicar en exceso este método, el cambio del contenido de las celdas lo delegaremos a otro método, dejando en este el cálculo de las celdas a cambiar. Comenzaremos cambiando la celda pulsada:

```
changeView(x, y);
```

Y continuaremos comprobando las celdas que deben cambiar por haber pulsado en la celda (x,y). Se debe tener cuidado con las celdas de las esquinas puesto que son especiales:

```
//esquinas del tablero
if (x==0 && y == 0){
   changeView(0, 1);
      changeView(1, 0);
   changeView(1, 1);
}
else if (x == 0 && y  == topTileY -1 ){
   changeView(0, topTileY-2);
```

```
    changeView(1, topTileY-2);
    changeView(1, topTileY -1);
}
else if (x == topTileX -1 && y  == 0 ){
    changeView( topTileX-2,0);
    changeView( topTileX-2,1);
    changeView( topTileX -1,1);
}
else if (x == topTileX -1 && y  == topTileY-1 ){
    changeView( topTileX-2,topTileY-1);
    changeView( topTileX-2,topTileY-2);
    changeView( topTileX -1,topTileY-2);
}
// lados del tablero
else if (x == 0){
    changeView( x,y-1);
    changeView( x,y+1);
    changeView( x+1 ,y);
}
else if (y == 0){
    changeView( x-1,y);
    changeView( x+1,y);
    changeView( x ,y+1);
}
else if (x == topTileX-1){
    changeView( x,y-1);
    changeView( x,y+1);
    changeView( x-1 ,y);
}
else if (y == topTileY-1){
    changeView( x-1,y);
    changeView( x+1,y);
    changeView( x ,y-1);
}
//resto
else{
    changeView( x-1,y);
    changeView( x+1,y);
    changeView( x,y-1);
    changeView( x,y+1);
}
```

Se actualiza el marcador de pulsaciones y se comprueba si se ha terminado la partida:

```
numberOfClicks++;
tvNumberOfClicks.setText( getString(R.string.score_clicks) + numberOfClicks
);
// se ha acabado la partida?
checkIfFinished();
```

El método `changeView()` debe recuperar la celda de la interfaz gráfica y modificar su contenido. La recuperación se hace mediante los identificadores que se han guardado en la `matriz ids[][]`. Una vez recuperado el objeto

TileView, se obtiene su nuevo índice de trama y se guarda en la matriz de índices `values[][]` que se utilizará para comprobar después si todas las celdas tienen el mismo contenido:

```
private void changeView(int x, int y){
      TileView tt = (TileView) findViewById(ids[x][y]);
      int newIndex = tt.getNewIndex();
      values[x][y] = newIndex;
      tt.setBackgroundResource(pictures[newIndex]);
      tt.invalidate();
    }
```

El contenido de la celda se varía mediante el método `setBackgroundResource()` que informa la imagen que se mostrará como fondo de la celda. Para obligar al sistema Android a refrescar el gráfico de la vista, se debe invalidar su contenido para que vuelva a redibujarla y tome el nuevo contenido que debe mostrar; para hacerlo nos valemos del método `invalidate()`.

El último método que se necesita codificar es el encargado de comprobar si la partida ha acabado o no. La comprobación se realiza iterando sobre la matriz que guarda los índices de trama de cada una de las celdas. Si la partida se ha acabado quiere decir que todas las celdas tendrán el mismo índice, es decir todas tendrán el mismo índice que la celda (0,0), por lo que se puede comprobar que cada celda tenga ese mismo valor, y en caso de encontrar una que no lo tenga, no merece la pena seguir comprobando el resto, simplemente se abandona el método y continúa la partida. Si todas las celdas tuvieran el mismo contenido, la partida se daría por terminada, por lo que se debería volver a la pantalla principal y anunciar al jugador que la partida ha finalizado.

Para volver a la actividad anterior, se utiliza el método `finish()` de la clase *Activity*, que dará por terminada la actividad actual y volverá a la actividad anterior. Si se quiere pasar algún tipo de dato como resultado de la ejecución de la actividad, se utilizará un objeto *Intent* donde se guardarán los datos a devolver en la forma de parejas clave-valor, de modo semejante a como se hace para pasar los datos cuando se crea una nueva actividad. También se puede dar un resultado de la ejecución de la actividad mediante `setResult()`, informando si ha ido todo bien, si el usuario ha cancelado la actividad… todos estos datos los debe procesar la actividad que lanzó la actividad que se está abandonando… deshaciendo el trabalenguas, los datos que se informen en la actividad *GameField*, los debe procesar *GameConfig*, ya que ésta lanzó aquella. El procesado de los datos se realiza a través del método `onActivityResult()`, que se ha codificado anteriormente. El código del método `checkIfFinished()` que se encarga de comprobar si ha terminado la partida es:

```
private void checkIfFinished() {
      int targetValue = values[0][0];
      for (int i =0; i < topTileY; i++){
         for (int j =0; j < topTileX; j++){
            if (values[j][i] != targetValue) return;
         }
      }
      Intent resultIntent = new Intent((String)null);
      resultIntent.putExtra("clicks", numberOfClicks);
      setResult(RESULT_OK,resultIntent);
      finish();
}
```

Y el código completo de la clase `GameField` queda:

```
public class GameField extends Activity {
//colores
private static final int []colors = new int[]{
      R.drawable.ic_1c,
      R.drawable.ic_2c,
      R.drawable.ic_3c,
      R.drawable.ic_4c,
      R.drawable.ic_5c,
      R.drawable.ic_6c
};
//numeros
private static final int []numbers = new int[]{
      R.drawable.ic_1n,
      R.drawable.ic_2n,
      R.drawable.ic_3n,
      R.drawable.ic_4n,
      R.drawable.ic_5n,
      R.drawable.ic_6n
};
//mantener el array que el usuario hay decidido utilizar
private int []pictures = null;
// Número máximo de celdas horizontales y verticales
private int topTileX =3;
private int topTileY =3;
// Número máximo de elementos a utilizar
private int topElements =2;
// Si ha seleccionado o no usar sonido y vibración
private boolean hasSound = false;
private boolean hasVibration = false;
// Array con los identificadores de las celdas cuando se añadan al layout,
// para poder recuperarlos durante la partida
private int ids[][] = null;
// Array para guardar los valores de los índices de cada una de las celdas.
// se utilizará para agilizar la comprobación de si la partida ha acabado o
no.
private int values[][] = null;
// Contador con el número de pulsaciones que ha realizado el jugador
private int numberOfClicks=0;
// Para reproducir un sonido cuando el usuario pulse una celda
private MediaPlayer mp= null;
```

```
// Para hacer vibrar el dispositivo cuando el usuario pulse una celda
private Vibrator vibratorService = null;
// Mostrará en pantalla el las veces que el usuario ha pulsado una celda
private TextView tvNumberOfClicks = null;
@Override
public void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
   setContentView(R.layout.gamefield);
   vibratorService = (Vibrator)(getSystemService(Service.VIBRATOR_SERVICE));
   mp = MediaPlayer.create(this, R.raw.touch);
   tvNumberOfClicks = (TextView) findViewById(R.id.clicksTxt);
   //obtención de parámetros de configuración
   Bundle extras = getIntent().getExtras();
   topTileX = extras.getInt("xtiles") + 3;
   topTileY = extras.getInt("ytiles") + 3;

   topElements = extras.getInt("numcolors") + 2;
   if ("C".equals(extras.getString("tile"))){
      pictures = colors;
   }
   else{
      pictures = numbers;
   }
   hasSound= extras.getBoolean("hasSound");
   hasVibration= extras.getBoolean("hasVibration");
   //limpiar el tablero
   LinearLayout ll = (LinearLayout) findViewById(R.id.fieldLandscape);
   ll.removeAllViews();
   //obtención de tamaño de pantalla
   DisplayMetrics dm = new DisplayMetrics();
   getWindowManager().getDefaultDisplay().getMetrics(dm);
   int width = dm.widthPixels / topTileX;
   int height = (dm.heightPixels - 180) / topTileY;
   //inicialización de arrays
   ids =  new int [topTileX][topTileY];
   values =  new int [topTileX][topTileY];
   //inicialización de números aleatorio
   Random r = new Random(System.currentTimeMillis());
   int tilePictureToShow = r.nextInt(topElements);
   // crear celdas
   int ident = 0;
   for (int i = 0; i < topTileY; i++) {
      LinearLayout l2 = new LinearLayout(this);
      l2.setOrientation(LinearLayout.HORIZONTAL);
      for (int j = 0; j < topTileX; j++) {
         tilePictureToShow = r.nextInt(topElements);
         // guardamos la trama a mostrar
         values[j][i] = tilePictureToShow;
         TileView tv = new TileView(this, j, i, topElements,
tilePictureToShow, pictures[tilePictureToShow]);
         ident++;
         // se asigna un identificador al objeto creado
         tv.setId(ident);
         // se guarda el identificador en una matriz
         ids[j][i] = ident;
```

```
            tv.setHeight(height);
            tv.setWidth(width);
            tv.setOnClickListener(new View.OnClickListener() {
            public void onClick(View view) {
               hasClick(((TileView) view).x, ((TileView) view).y);
               }
               });
            l2.addView(tv);
            }
         ll.addView(l2);
      }
      // cronómetro
      Chronometer t = (Chronometer)findViewById(R.id.Chronometer);
      t.start();
   }
   protected void hasClick(int x, int y) {
      // vibrar y/o sonar si está configurado
      if (hasVibration) vibratorService.vibrate(100);
      if (hasSound) mp.start();
      //cambiar la celda pulsada
      changeView(x, y);
      //esquinas del tablero
      if (x==0 && y == 0){
         changeView(0, 1);
            changeView(1, 0);
         changeView(1, 1);
      }
      else if (x == 0 && y  == topTileY -1 ){
         changeView(0, topTileY-2);
         changeView(1, topTileY-2);
         changeView(1, topTileY -1);
      }
      else if (x == topTileX -1 && y  == 0 ){
         changeView( topTileX-2,0);
         changeView( topTileX-2,1);
         changeView( topTileX -1,1);
      }
      else if (x == topTileX -1 && y  == topTileY-1 ){
         changeView( topTileX-2,topTileY-1);
         changeView( topTileX-2,topTileY-2);
         changeView( topTileX -1,topTileY-2);
      }
      // lados del tablero
      else if (x == 0){
         changeView( x,y-1);
         changeView( x,y+1);
         changeView( x+1 ,y);
      }
      else if (y == 0){
         changeView( x-1,y);
         changeView( x+1,y);
         changeView( x ,y+1);
      }
      else if (x == topTileX-1){
         changeView( x,y-1);
```

```
      changeView( x,y+1);
      changeView( x-1 ,y);
   }
   else if (y == topTileY-1){
      changeView( x-1,y);
      changeView( x+1,y);
      changeView( x ,y-1);
   }
   //resto
   else{
      changeView( x-1,y);
      changeView( x+1,y);
      changeView( x,y-1);
      changeView( x,y+1);
   }
   // actualiza marcador
   numberOfClicks++;
   tvNumberOfClicks.setText( getString(R.string.score_clicks) +
numberOfClicks );
   // se ha acabado la partida?
   checkIfFinished();
}
private void changeView(int x, int y){
      TileView tt = (TileView) findViewById(ids[x][y]);
      int newIndex = tt.getNewIndex();
      values[x][y] = newIndex;
      tt.setBackgroundResource(pictures[newIndex]);
      tt.invalidate();
}
private void checkIfFinished() {
      int targetValue = values[0][0];
      for (int i =0; i < topTileY; i++){
         for (int j =0; j < topTileX; j++){
            if (values[j][i] != targetValue) return;
         }
      }
      Intent resultIntent = new Intent((String)null);
      resultIntent.putExtra("clicks", numberOfClicks);
      setResult(RESULT_OK,resultIntent);
      finish();
}
}
```

Ya se podría probar la aplicación… o no. Si se ejecutara con el código visto hasta ahora, encontraríamos que todo iría bien en la configuración de la partida, pero cuando se pulsara el botón de iniciar el juego, la aplicación daría un error y se pararía. Esto es porque no se ha añadido la *Activity* que se está intentando lanzar a nuestro `AndroidManifest.xml`.
Introducimos la actividad en él:

```
<activity android:name=".HowTo" android:label="@string/title_howto"/>
<activity android:name=".GameField" />
</application>
```

Y aprovecharemos para pedir permiso para utilizar la vibración del terminal mediante la etiqueta <uses-permission> antes de la etiqueta <application>:

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

A través de <uses-permission> se le indica al sistema Android que la aplicación necesita que se le otorguen permisos especiales, en este caso se le ha requerido el uso de la opción vibrar pero pueden ser otros muchos, por ejemplo poder hacer llamadas, acceder a la cámara, establecer el fondo de pantalla… muchos y variados. Todos los permisos posibles se pueden encontrar en la documentación de Android.

Ahora ya está preparado para batir su propio record. Queda como ejercicio para el lector devolver también el tiempo empleado en resolver el tablero.

Figura 9.4. Imágenes del juego en acción.





10 Un diseño para múltiples formatos de pantalla

En este capítulo aprenderá a:

- Ver las ventajas de un diseño de pantallas fragmentado
- Cómo realizar aplicaciones utilizando *Fragments*
- Encapsular interfaces reutilizables en tabletas y teléfonos

Con la aparición en el mercado de las tabletas, se comenzaba a gestar un problema para los desarrolladores de aplicaciones Android, y es que dependiendo del terminal, habría que hacer unas interfaces distintas para adaptarse al tamaño de la pantalla. Hasta ahora hemos visto que mediante los distintos directorios de recursos, podíamos indicar que usara unos iconos o unos *layouts* en lugar de otros para ajustar el diseño, pero el problema radica que una pantalla de tableta es más de dos veces una de un teléfono, con lo que cuesta mucho (por no decir que es imposible) que una misma actividad tal y como las conocemos hasta ahora, quede bien visualmente al escalarla para que encaje en 10 pulgadas (tamaño normal de la tableta) o en apenas dos como un reloj. En la versión 3.0 de Android se comenzó el camino para que a lo largo de las versiones 4.x tanto tabletas como teléfonos disfrutaran del mismo sistema operativo y misma interfaz gráfica. Veamos cómo.

Fragmentos

Para aliviar el problema de tener que diseñar diferentes actividades para teléfonos y tabletas y multiplicar el trabajo, lo que se optó fue por hacer unas unidades de trabajo más pequeñas que las *Activity*, de modo que éstas se apoyaran en ellas. Así aparecen los *Fragment*(fragmentos), que son entidades con interfaz y lógica de negocio reutilizables a través de las *Activity* y solamente a través de ella, es decir deben existir dentro de una actividad que además se encargará de la gestión del ciclo de vida de dichos fragmentos. Para aclarar un poco, las actividades siguen existiendo pero en lugar de estar formadas (dicho burdamente) por una clase y un *layout*, ahora pueden estar formadas también por los *Fragment*; si como ya se verá, el trabajar mediante clases *Activity* hace que la aplicación sea muy modular y aprovechable, con las clases *Fragment* se va un paso más allá. Aunque más adelante se verá con más claridad, valga decir que para hacer visible un fragmento, éste debe existir dentro de un *ViewGroup* de la vista de la actividad o se debe hacer presente a través de la etiqueta `<fragment>` durante la definición del *layout.*
Supongamos que tenemos una aplicación de platos de cocina, es posible que en varias pantallas nos interese tener la ficha nutricional de un alimento dado, que se le pase por parámetro. Hasta ahora, o bien se hacia una *Activity* para ello o bien se tenía que duplicar código, ahora, simplemente se instanciaría el *Fragment* encargado de mostrar los datos dentro de cualquier actividad y ya lo tendríamos resuelto, aprovechando así mucho más el código. Pero las ventajas no quedan ahí y es que como una *Activity* puede tener muchos *Fragment* y un *Fragment* puede estar presente en varias *Activity*, se puede ahora ajustar las interfaces mucho mejor

entre tabletas y teléfonos aprovechando mucho más el código. Vamos a imaginar una aplicación que muestra una lista de equipos de fórmula 1, que seleccionando un equipo nos muestre las características de los coches y sus dos pilotos y que seleccionando cada uno de los pilotos de la escudería nos muestre su palmarés... demasiada información para una pantalla de teléfono que quizá el lector ya estaba pensando en diseñarlo mediante tres actividades (pantallas)... pero ¿y para una tableta? está claro que podría caber toda la información en una sola actividad; pues aquí es donde entran en juego los *Fragment*.

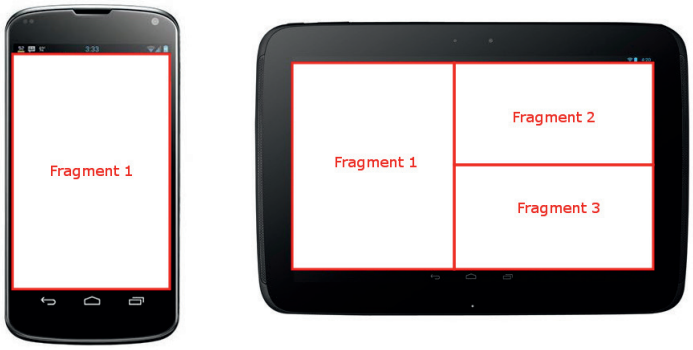

Figura 10.1. Diseño de la aplicación mediante tres Fragments.

Supongamos que la lista de equipos es el *Fragment* 1, el detalle del equipo es el *Fragment* 2 y que el detalle del piloto es el 3, para el teléfono se podría diseñar una sola actividad que se encargue de mostrar un *Fragment* en cada momento o tres actividades independientes que se llamen entre ellas y mostrando un *Fragment* cada una (mucho más versátil esta opción, porque si hacemos otra aplicación que de los resultados de cada carrera, podemos usar la misma actividad de detalle de piloto simplemente pasándole el piloto como parámetro) mientras que en la tableta se podrían mostrar los tres *Fragments* a la vez controlados por una sola actividad.

Para poder trabajar con *Fragments* es necesario tener instalada una versión igual o superior a Android 3.0 o instalar el paquete de compatibilidad. Actualmente ya se configura el proyecto de modo automático para que use esta librería de compatibilidad, y esto lo podemos ver dentro de la rama External Libraries en el panel Project de nuestro proyecto. Con esta librería se quiere hacer llegar funcionalidad de las nuevas versiones a versiones anteriores, pero hay que tener cuidado, porque no están todas las funcionalidades incluidas; y por ejemplo, en el caso de los *Fragments*, existen clases que se llaman de distinta manera en el paquete de compatibilidad, pese a que su funcionalidad es la misma.

Para crear un fragmento es necesario crear una subclase de tipo *Fragment*, que tiene unos métodos de *callback* muy semejantes a los que tienen las actividades, como por ejemplo `onCreate()`, `onPause()`...
Alguno de los métodos a implementar más comunes son:

- **onAttach()**: Indica que el fragmento se ha asociado a una actividad.
- **onCreate()**: Se ejecuta al crear el fragmento, es el lugar donde inicializar los elementos esenciales del fragmento y datos que se quieren mantener al volver a primer plano el fragmento tras haberlo pausado o parado.
- **onCreateView()**: Se llama cuando el fragmento debe dibujar su interfaz. El método devuelve un objeto de tipo *View* con la vista a mostrar, ésta se puede obtener inflando un recurso *layout* o construyéndolo mediante programación.
- **onPause()**: Es semejante al visto en las actividades. Indica que el fragmento va a desaparecer de pantalla y se utiliza para guardar los datos sensibles antes de que desaparezca el fragmento.
- **onDetach()**: Indica que el fragmento ya no esta asociado a la actividad.

En la figura 10.2 se pueden ver todos los métodos del ciclo de vida de un fragmento y el orden en que se producen.

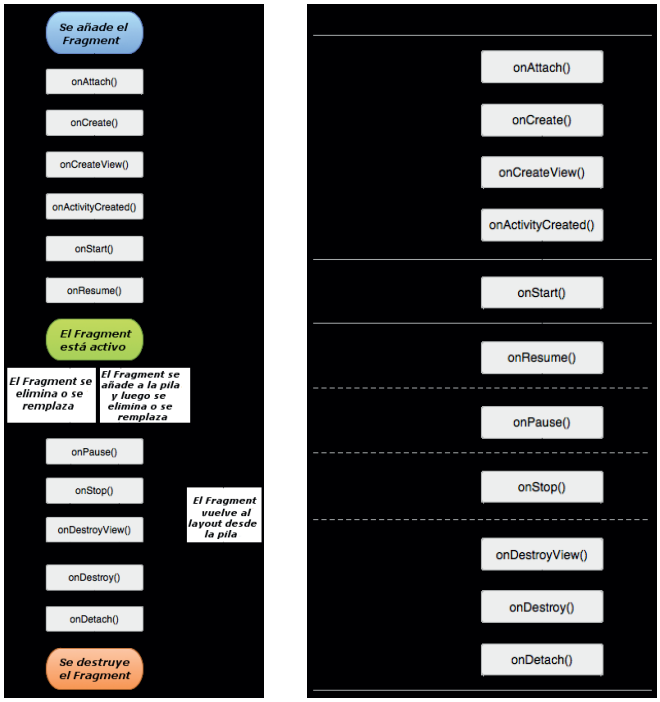

Figura 10.2. Ciclo de vida de un Fragment

Para que una actividad pueda alojar un fragmento, vale con extender la clase Activity a partir de la versión 3.0 de Android, pero si se quiere en versiones anteriores, es necesario usar la librería de compatibilidad que ofrece Google y en este caso, la clase debe extender la FragmentActivity en lugar de la Activity y tal y como se recalcará unas líneas más adelante, hay que tener cuidado de no mezclar clases de versiones Android mayores que al API 11 con las clases del paquete de compatibilidad, puesto que se llaman igual pero no son lo mismo y producen excepciones del tipo ClassCastException al intentar equipararlas.
La comunicación entre fragmentos se suele hacer a través de la actividad que los acoge, ya que ésta está siempre disponible en todos los fragmentos mediante la llamada `getActivity()`, que devuelve la actividad contenedora en ese momento de ese fragmento y la actividad puede conocer qué fragmentos tiene asociados y buscar uno concreto entre ellos.
Por último y antes de introducirnos en ejemplos de funcionamiento, veremos las dos formas que tiene una actividad de nutrirse de los fragmentos. La primera de las formas es introducir el fragmento directamente en la definición del XML del layout de la vista a través de la etiqueta `<fragment>` como si fuera un elemento más de la vista, y durante la ejecución, el sistema cambiará el elemento `<fragment>` por la vista devuelta por éste. Un ejemplo de definición de este tipo sería:

```
<fragment
   android:id="@+id/driver_list"
   android:name="com.acme.listdetail.DriverListFragment"
   android:layout_width="0dp"
   android:layout_height="match_parent"
   android:layout_weight="1" />
```

Donde `android:name` es el nombre de la clase *Fragment* que hayamos creado. Muy importante es indicar el identificador de la vista como se ha hecho otras veces mediante `android:id` ya que como veremos más adelante es necesario para identificar unívocamente el fragmento y poder actuar con él desde la actividad. También se podría utilizar el atributo `android:tag` para indicar el identificador, y en este caso el valor debería ser una cadena de texto que tendría que ser única. La otra manera de usar un fragmento en una actividad es mediante código, obteniendo el *ViewGroup* de la vista de la actividad donde se quiere mostrar, instanciando el fragmento y asignándolo a ese *ViewGroup*. Un ejemplo de esto sería:

```
DriverListFragment fragment = new DriverListFragment();
FragmentTransaction fragmentTransaction = getFragmentManager().
beginTransaction();
fragmentTransaction.add(R.id.fragment_container, fragment);
fragmentTransaction.commit();
```

Donde `DriverListFragment` es el fragmento a mostrar y `R.id.fragment_container` es el identificador de la vista donde se quiere mostrar por ejemplo un *FrameLayout*.
Se habrá fijado el lector en que en el código superior se utiliza un objeto del tipo *FragmentTransaction* para ejecutar la acción de mostrar el fragmento en pantalla, y así es, mediante este objeto podremos añadir, eliminar y sustituir fragmentos de las actividades, de uno en uno o varios a la vez, incluso incluir animaciones, pero para ello primero necesitamos el objeto *FragmentManager* que se obtiene en la actividad mediante la llamada `getFragmentManager()` (o `getSupportFragmentManager()` si se utiliza el paquete de compatibilidad) quien se encarga de mantener un control de los fragmentos disponibles y de el orden en el que se han ido mostrando en pantalla.
Por último acabar reseñando que al igual que en las actividades, dentro de los fragmentos existen algunos con características especiales, como el *ListFragment* para gestionar listas, el *PreferenceFragment* para las pantallas de preferencias de usuario o el *DialogFragment* para la generación de cuadros de diálogo y que se mostrará el funcionamiento al final del capítulo.

Pantallas de lista detalle

Es posible que a estas alturas el lector tenga un poco de lio en la cabeza con los fragmentos, créame que es normal, pero se aclarará mediante los ejemplos de la unidad.
Vamos a aprovechar el asistente de creación de proyectos para generar uno de tipo lista/detalle que trabaja con *Fragments* y ver el código obtenido.
Cree el proyecto con la siguiente configuración:

- Application name: ListDetail
- Module Name: ListDetail
- Package name: com.acme.listdetail
- Minimum Required SDK: API 11: Android 3.0 (Honeycomb) o superior
- Master/Detail Flow

En el paso del asistente sobre iconos, seleccione el que más le guste. El siguiente paso, pese a ser conocido vamos a cambiar la selección, seleccione el tipo `Mater/DetailFlow`. El siguiente y último paso del asistente es nuevo y viene dado por la selección que se ha realizado en el paso anterior. Ya que se está creando una actividad de tipo lista detalle, se nos pide que informemos

el tipo de dato que va a contener la lista y cómo queremos que se llame dicha lista para el ejemplo hagamos una lista de pilotos, entonces para informar los campos sería:

- Object Kind: Driver
- Object Kind Plural: Drivers

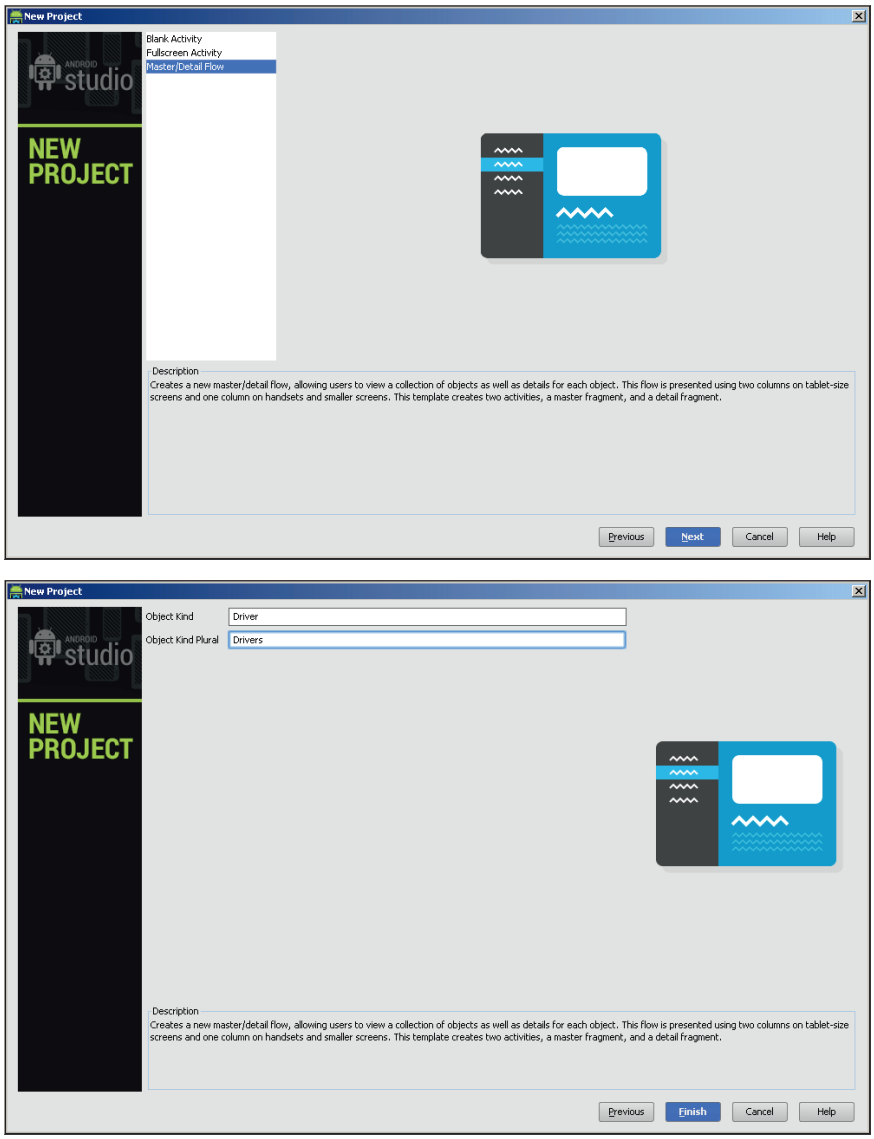

Figura 10.3. Selección de `MasterDetailFlow` y elementos a mostrar

Nota:

Es muy importante especificar en la entrada `Minimum Required SDK` *el valor API 11 como mínimo, ya que usaremos características que solamente están disponibles a partir de esta versión.*

Lo que se acaba de generar es una aplicación que en una tableta muestra dos fragmentos, una lista a la izquierda y al seleccionar algo sobre la lista se muestra el detalle en la derecha, pero esta misma aplicación si se ejecuta sobre un teléfono, mostrará una actividad con la lista y al seleccionar un elemento de la lista mostrará su detalle en otra actividad. Echemos un vistazo al código generado. Dentro del paquete `com.acme.listdetail` encontramos cuatro clases, dos correspondientes a actividades y dos a fragmentos. Si se analiza el archivo `ListDetail-Manifest.xml`, se puede ver que la actividad con la que se comienza la ejecución de la aplicación es la `DriverListActivity`, así que comenzaremos a analizar el código desde esta actividad. Dentro de la clase aparece definida una variable que servirá para saber si debe mostrar uno o dos fragmentos.

```
private boolean mTwoPane;
```

El método para saber si se deben mostrar uno o dos fragmentos es sencillo, en el método `onCreate()`, como anteriormente, se define el *layout* a cargar por parte de la actividad:

```
setContentView(R.layout.activity_driver_list);
```

Con lo que, y tal y como hemos ya aprendido, el sistema irá a cargar el *layout* llamado `activity_driver_list.xml` dentro del directorio `/src/main/res/layout/` o dentro de alguna de sus modificaciones (como puede ser por tamaño de pantalla), por ejemplo los *layouts* que existieran en `/src/main/res/layout-large` o `/src/main/res/layout-sw600dp` y que tuvieran el mismo nombre, de modo que pantallas grandes desde Android 3.0 irían a buscar ahí su *layout*, pero esto tiene una desventaja, y es que si queremos mantener *layouts* por ejemplo para televisiones y tabletas con Android 3.0, habría que mantener el fichero XML en los dos directorios, con lo que si el día de mañana se realiza una modificación, habría que modificar todos los ficheros XML de todos los directorios. Aquí es donde aparecen los alias. Si se fija, no existen estos directorios de *layout* y en su lugar existen dos directorios `/src/main/res/values-sw600dp/` y `/src/main/res/values-large/`, que ambos tienen el mismo contenido, un fichero con las líneas:

```
<resources>
   <item name="activity_driver_list" type="layout">@layout/activity_driver_
   twopane</item>
</resources>
```

Con esto lo que se le está diciendo al sistema es que cree un alias y que cuando tenga que cargar un *layout* llamado `activity_driver_list`, cargue en su lugar el definido en `@layout/activity_driver_twopane`, pero hay que tener en cuenta que como están en directorios con modificadores, esto sólo ocurrirá cuando se ejecute en un dispositivo que cumpla los modificadores de los directorios, en este caso que tenga un mínimo de 600dip de ancho (sw600 significa *smallest width* 600 dp y es una notación introducida a partir de Android 3.2 para denotar las pantallas grandes) o tenga pantalla grande en Android 3.x.Resumiendo, si se trata de un dispositivo con una anchura menor a 600dp (normalmente si es de menos de 7"), mostrará el *layout* `activity_driver_list.xml` y en caso de ser mayor mostrará `activity_driver_twopane.xml`.
Si se abren dichos ficheros de *layout*, se verá que en `activity_driver_list.xml` solamente se define un *Fragment* y en `activity_driver_twopane.xml` se define un *LinearLayout* con dos fragmentos, pero con la peculiaridad de que en los dos *layouts* aparece el identificador `@+id/driver_list` pero no `@+id/ driver_detail_container`, es decir que si se ejecutara en el `onCreate()` un `findViewById(R.id.driver_detail_container)` dependiendo del *layout* cargado tendría valor o no, y es la forma de saber si se están mostrando uno o dos *Fragments* (saber si es pantalla grande o no); en caso de que obtengamos nulo es que se está mostrando en pantalla pequeña porque no está disponible en pantalla y en caso de que tenga valor se estarán mostrando los dos fragmentos. Esto es lo que se hace en el `onCreate()`:

```
if (findViewById(R.id.driver_detail_container) != null) {
   mTwoPane = true;
   ((DriverListFragment) getSupportFragmentManager().findFragmentById(
               R.id.driver_list)).setActivateOnItemClick(true);
}
```

Ahora ya sabemos si tiene uno o dos paneles; en caso de ser dos paneles, se guarda la variable `mTwoPane` con valor positivo (que por defecto tenía valor negativo) y se le dice que cuando se pulse la lista, quede marcada la opción seleccionada. Esto lo hace llamando a una clase denominada *FragmentManager* que nos acompañará durante todo el desarrollo de aplicaciones con fragmentos. Esta clase se encarga de gestionar los *Fragments* mostrados en cada momento, tener una pila de llamadas entre ellos de modo que podamos volver al *Fragment* anterior (del mismo modo que se hace con las actividades), permite a su vez y mediante transacciones añadir, eliminar o sustituir *Fragments*, y prácticamente cualquier operación que se nos ocurra hacer con fragmentos tendremos que recurrir a esta clase. Para obtener el *FragmentManager*

en este caso se usa `getSupportFragmentManager()` desde la actividad, pero se podría haber utilizado `getFragmentManager()`, la diferencia es que el primero devuelve el *FragmentManager* del paquete de compatibilidad y el segundo devuelve la clase disponible de modo estándar desde Android 3.0. Puede fijarse en las External Libraries del proyecto, que se ha incluido el paquete de compatibilidad mediante el fichero `support-vx-xx.x.x.jar`. Hay que tener cuidado de qué tipo de *FragmentManager* se obtiene, porque las clases utilizadas por cada uno de ellos también son distintas, por ejemplo el *FragmentManager* de soporte utiliza la clase de fragmentos `android.support.v4.app.Fragment` mientras que el estándar utiliza `android.app.Fragment` que aunque se llamen igual y tengan mismas funcionalidades no son lo mismo. Como en la aplicación recién creada, todo se basa en el paquete de compatibilidad se utilizan todas las clases del paquete `android.support.v4.app` y por lo tanto se debe usar el `getSupportFragmentManager()`. Una vez obtenido el *FragmentManager,* lo que se hace es buscar el fragmento que tenga por identificador `@+id/driver_list`, de modo semejante a como se hace con las vistas para obtener la referencia de un objeto de pantalla, pero en lugar de llamar a buscar una vista `findViewById()` se busca un fragmento `findFragmentById()`; sobre el fragmento se realiza un *cast* a la clase *DriverListFragment* que es el tipo de clase del fragmento supuestamente encontrado, y una vez realizado el *cast* se llama al método `setActivateOnItemClick()` para que cuando la lista sea pulsada, deje marcado el elemento tocado, y es que en el caso de dos paneles, veremos el detalle del elemento pulsado, pero también interesa que se quede marcado el elemento para saber sobre cual se pulsó. En el caso de un fragmento no hace falta ya que la lista desaparece para mostrar el detalle.

Dentro del la misma clase *DriverListActivity* podemos ver el método `onItemSelected()`, que será el encargado de ejecutarse cada vez que se seleccione un elemento de la lista. Mirando su código:

```
if (mTwoPane) {
   Bundle arguments = new Bundle();
   arguments.putString(DriverDetailFragment.ARG_ITEM_ID, id);
   DriverDetailFragment fragment = new DriverDetailFragment();
   fragment.setArguments(arguments);
   getSupportFragmentManager().beginTransaction()
               .replace(R.id.driver_detail_container, fragment).commit();

   } else {
      Intent detailIntent = new Intent(this, DriverDetailActivity.class);
      detailIntent.putExtra(DriverDetailFragment.ARG_ITEM_ID, id);
      startActivity(detailIntent);
   }
```

Lo primero que hace es comprobar si se encuentra en el caso de dos paneles o de uno, el código que ejecutará en caso de ser un panel ya es de sobra conocido, simplemente crea un nuevo *Intent* al que le añade un parámetro con clave `DriverDetailFragment.ARG_ITEM_ID` y lanza su ejecución con `startActivity()`, así que centrémonos en la primera parte del `if`, que es la que se ejecutará en caso de que sean dos paneles. En este caso lo que se hace es crear un *Bundle* (que es el mismo tipo de objeto que se usa para pasar los datos en los *Intent*) que servirá para informar el elemento seleccionado de la lista y se le añaden los parámetros, acto seguido se instancia el fragmento que se quiere mostrar, en este caso `DriverDetailFragment()` y por último y una vez más mediante el *FragmentManager,* se muestra el fragmento. Para mostrarlo lo primero que se hace es obtener la instancia del gestor de fragmentos como ya se había visto, tras ellos, se comienza una transacción mediante el método `beginTransaction()`, dentro de esta transacción se añaden los fragmentos a modificar y por último se llama al método `commit()` para confirmar y ejecutar la transacción. En el código, la transacción realizada es remplazar el *Fragment* que se encuentre en la vista con el identificador `R.id.driver_detail_container` por el nuevo fragmento creado, en este caso el panel derecho del *layout* `activity_driver_twopane.xml`. En este caso la transacción era muy sencilla porque sólo entraba en juego un fragmento, pero en casos más complejos permite sincronizar el cambio de varios *Fragments* e incluso sincronizarlos con animaciones, para ello simplemente habría que indicar las modificaciones de los fragmentos entre el método `beginTransaction()` y el `commit()`, por ejemplo:

```
//se obtiene la transacción para darle la animación
FragmentTransaction ft = getFragmentManager().beginTransaction();
ft.setCustomAnimations(R.anim.slide_in_left, R.anim.slide_out_right);
//se realizan los cambios necesarios
DriverDetailFragment newFragment = new DriverDetailFragment();
DriverPhotoFragment newFragmentPhoto = new DriverPhotoFragment();
ft.replace(R.id.driver_detail_container, newFragment);
ft.replace(R.id.driver_photo_container, newFragmentPhoto);
//se ejecutan los cambios mediante animación
ft.commit();
```

Ya veremos algo más de esto, pero ahora sigamos con el análisis del código. Vamos a ver el mecanismo que se usa para llamar a este método `onItemSelected()` con el identificador del elemento seleccionado. Si se fija, la clase *DriverListActivity* implementa *DriverListFragment.Callbacks* esta clase está definida dentro de *DriverListFragment*. Dentro de la clase *DriverListActivity* podemos ver la definición del interface *Callbacks* y un objeto estático llamado `sDummyCallbacks`, su funcionamiento es el siguiente. Al crearse el fragmen-

to se asocia al objeto `mCallbacks` el objeto "dummy" `sDummyCallbacks`, y en cuanto se asocia el fragmento a una actividad se ejecutará el método `onAttach()` del propio *Fragment* donde se obtiene la actividad a la cual se ha asociado dicho fragmento, se comprueba si es del tipo *Callbacks* y en caso de serlo quiere decir que puede ser la encargada de recibir los eventos de selección de la lista, con lo que se guarda en el objeto `mCallbacks` para ser utilizado en otros métodos y en caso de no serlo se lanza una excepción. El fragmento que estamos observando no es un *Fragment* normal, sino que es un *ListFragment*, que es el semejante al *ListActivity* pero en fragmentos, es decir nos ofrece métodos como *onListItemClick()* que se ejecuta cada vez que se selecciona un elemento de la lista que tenga definida y es precisamente en este método donde se aprovecha para llamar al método definido en el *Callback*, en este caso el objeto que esté referenciado por `mCallbacks` que será la actividad *DriverListActivity*.

```
public void onListItemClick(ListView listView, View view, int position,
        long id) {
      super.onListItemClick(listView, view, position, id);
      mCallbacks.onItemSelected(DummyContent.ITEMS.get(position).id);
   }
```

Al desasociar el fragmento de la actividad, se ejecuta el método `onDetach()`, donde se vuelve a asociar el objeto "dummy" de recepción de llamadas.

```
public void onDetach() {
      super.onDetach();
      mCallbacks = sDummyCallbacks;
   }
```

La mayor parte de las veces la vista de un fragmento establece en el método `onCreateView()` aunque no en este caso. La vista de este fragmento se configura en el método `onCreate()` pero en lugar de hacer un `setContentView()` como se hacía en las actividades que hemos visto hasta ahora, se utilizará toda la vista como una lista, por el hecho de extender la clase *ListFragment* no hace falta indicar el *layout* ni decir que tiene un *ListView*, simplemente vale con asignar el adaptador de dicha lista con los valores a mostrar.

```
setListAdapter(new ArrayAdapter<DummyContent.DummyItem>(getActivity(),
   android.R.layout.simple_list_item_activated_1,android.R.id.text1,
DummyContent.ITEMS));
```

Para acabar de analizar el código queda simplemente ver la clase *DriverDetailFragment*, que mostrará el detalle del elemento pulsado en la lista. A parte del constructor, presenta dos métodos el `onCreate()` donde se obtiene el parámetro pasado que se corresponde con el identificador del ele-

mento pulsado y el `onCreateView()` que se encarga de inflar la vista a mostrar, definir sus valores del mismo modo que hemos venido haciendo en la actividad, solo que el `findViewById()` se debe ejecutar sobre el objeto de la vista recién inflada. Una vez que la vista está completamente configurada con los valores y métodos de *callback* (por ejemplo métodos para atender las pulsaciones de botones), se devuelve como resultado del método.

```
View rootView = inflater.inflate(R.layout.fragment_driver_detail,
            container, false);

      if (mItem != null) {
         ((TextView) rootView.findViewById(R.id.driver_detail))
               .setText(mItem.content);
      }

      return rootView;
```

Nota:

Para trabajar con fragmentos en versiones anteriores a la 3.0 (es decir con el paquete de compatibilidad) es necesario que las clases que vayan a ser actividades deben extender la FragmentActivity *en lugar de la* Activity*, en versiones posteriores a la 3.0 no hace falta ya que la* Activity *ya implementa la lógica necesaria para poder gestionar los fragmentos.*

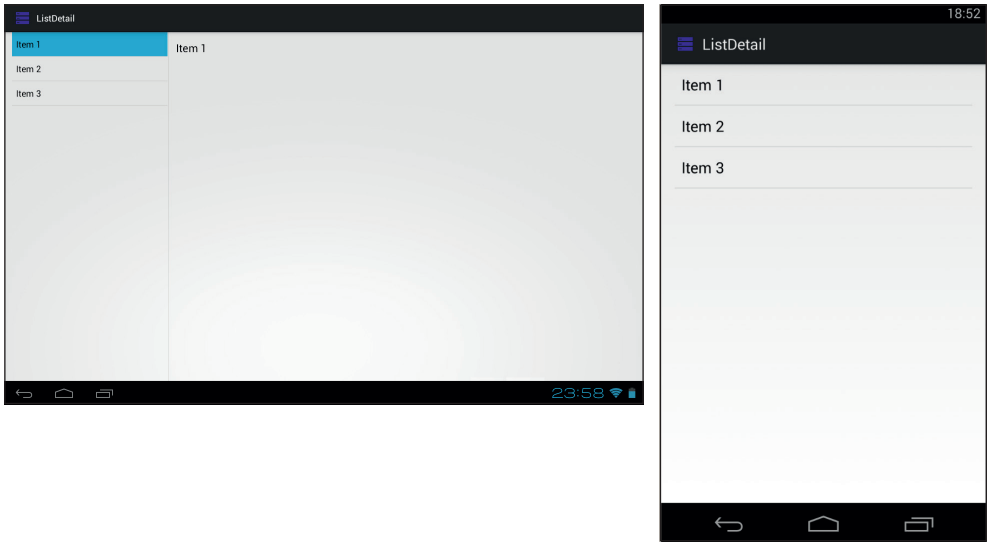

Figura 10.4. Aplicación ListDetail en ejecución

Ejemplo de uso de Fragments

Realizaremos un nuevo proyecto donde se pueda comprobar el uso y manejo de distintos fragmentos. Cree el proyecto:

- Application name: FragmentTest
- Project Name: FragmentTest
- Module name: com.acme.fragmenttest
- Minimum Required SDK: API 11: Android 3.0 (Honeycomb) o superior
- Bank Activity
- Activity Name: FragmentTest

Una vez generado el esqueleto de la aplicación, lo iremos modificando para ajustarlo a nuestra conveniencia. Lo primero que haremos es crear una pantalla donde pulsando un botón se vayan añadiendo nuevos *Fragment* y se visualicen, como si de una pila de libros se tratara.
Modificamos el *layout* de nombre `activity_fragment_test.xml` que será quien acoja el diseño de la pantalla principal, donde iremos introduciendo los nuevos *Fragment*. Como contenido del *fragment*, añadiremos un elemento `<FrameLayout>` que como recordará servía para mostrar distintas *View* colocadas una sobre otra; y para acabar necesitaremos un botón que servirá para ir añadiendo los diferentes *Fragment*. Sustituimos el contenido del *layout* por:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:orientation="vertical" android:padding="4dip"
   android:gravity="center_horizontal"
   android:layout_width="match_parent" android:layout_height="match_parent">

   <Button android:id="@+id/newFragment"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:text="Nuevo Fragment"/>

   <FrameLayout
   android:id="@+id/fragmentShow"
   android:layout_width="match_parent"
   android:layout_height="match_parent">
   </FrameLayout>

</LinearLayout>
```

Los *Fragment* a mostrar los tomaremos de dos archivos de *layout* diferentes; para ver mejor como se añaden nuevos *Fragment* lo que haremos es crear un contador de elementos añadidos y mostraremos un *layout* para cuando

el contador sea par y otro para cuando sea impar. Los *layouts* mencionados los guardaremos en sendos ficheros con nombre `layout1.xml` y `layout2.xml` respectivamente. El contenido de `layout1.xml` es muy sencillo, tan solo un elemento de texto que se encargará de mostrar el contador de fragmentos:

```
<?xml version="1.0" encoding="utf-8"?>
<TextView xmlns:android="http://schemas.android.com/apk/res/android"
android:id="@+id/text"
   android:layout_width="match_parent" android:layout_height="match_parent"
   android:gravity="center_vertical|center_horizontal"
   android:text="" android:textSize="20dip"/>
```

En cuanto al contenido de `layout2.xml`, el contador será un poco distinto, mostrando una etiqueta con el contador y una imagen (todo un derroche de diseño).

```
<?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:text="" android:id="@+id/text2"
android:layout_width="match_parent" android:layout_height="wrap_content"
   android:gravity="center_vertical|center_horizontal"
   android:textSize="20dip"/>
<ImageView  android:id="@+id/image"
android:layout_width="match_parent" android:layout_height="match_parent"
android:gravity="center_vertical|center_horizontal"
android:src="@drawable/ic_launcher"/>
</LinearLayout>
```

Pasemos al código. En la clase *FragmentTest*, crearemos una variable miembro llamada `mStackPosition` para mantener el contador de los *Fragment* añadidos. Ya dentro del método `onCreate()` daremos código al botón que se encargará de añadir los *Fragment* y comprobaremos si es la primera vez que se ejecuta el programa. En caso de ser así, se añadirá por defecto un *Fragment* al *FrameLayout*. La manera de saber si ya ha sido inicializado o no, es comprobando el parámetro *Bundle* `savedInstanceState` y ver si es nulo (implicaría que no ha sido inicializado), esto hace ya pensar que más adelante tendremos también que llamar al método `onSaveInstanceState()` donde nos encargaremos de indicar que ya ha sido inicializado el *FrameLayout* salvando este *Bundle*. En este parámetro mantendremos el valor de la variable `mStackPosition`, de modo que siempre sepamos cuantos elementos había en la actividad. Utilizaremos en este caso `getFragmentManager()` para el control de los fragmentos, es decir, no usaremos las clases del paquete de compatibilidad, por lo que es importante haber seleccionado las APIs correctas a la hora de generar el proyecto.

```
public class FragmentTest extends Activity {
// num Fragment
int mStackPosition = 1;
@Override
public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
   setContentView(R.layout.activity_fragment_test);

   // Botón de añadir fragments
   Button button = (Button)findViewById(R.id.newFragment);
   button.setOnClickListener(new View.OnClickListener() {
                public void onClick(View v) {
                   addFragment();
      }
   });

   if (savedInstanceState == null) {
      // añadir el primer fragment
      Fragment newFragment = SimpleFragment.newInstance(mStackPosition);
      FragmentTransaction ft = getFragmentManager().beginTransaction();
      ft.add(R.id.fragmentShow, newFragment).commit();
   } else {
      mStackPosition = savedInstanceState.getInt("position");
            }
}
}
```

Para añadir un nuevo *Fragment* al *FrameLayout*, nos valemos del objeto *FragmentTransaction* mediante su método `add()`, donde indicamos que lo que se quiere hacer es añadir el fragmento. El fragmento debe ser previamente instanciado; en este caso, se utiliza una clase llamada *SimpleFragment* (que extenderá la clase *Fragment*) y que aún no está definida. Pasemos a definir la clase *SimpleFragment* dentro de la clase *FragmentTest*.

```
public static class SimpleFragment extends Fragment {
   int mNum;
   static SimpleFragment newInstance(int number) {
      SimpleFragment f = new SimpleFragment();
      // Mantenemos el número para usarlo en cualquier momento.
      Bundle args = new Bundle();
      args.putInt("num", number);
      f.setArguments(args);
      return f;
   }

   @Override
   public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      // obtenemos el número que se habia pasado como argumento en
      // la creación de la instancia
      mNum = getArguments().getInt("num");
   }

   @Override
   public View onCreateView(LayoutInflater inflater, ViewGroup container,
```

```
    Bundle savedInstanceState) {
        View v  = null;
        // dependiendo de si es par o impar mostramos distintos layouts
        if (mNum % 2 == 0){
            v = inflater.inflate(R.layout.layout1, container, false);
            View tv = v.findViewById(R.id.text);
            //informamos el número de Fragment
            ((TextView)tv).setText("Fragmento número #" + mNum);
        }
        else{
            v = inflater.inflate(R.layout.layout2, container, false);
            View tv = v.findViewById(R.id.text2);
            //informamos el número de Fragment
            ((TextView)tv).setText("Fragmento número #" + mNum);
        }
        return v;
    }
}
```

La primera de las peculiaridades de la clase es que extiende la clase *Fragment*, con ello conseguimos heredar ciertos métodos que permitirán su manejo como parte de la actividad. Dentro del método `newInstance()` se crea el objeto *SimpleFragment* y se le asigna como argumentos un *Bundle* con el valor pasado en su llamada, que corresponde al número de elemento *Fragment* añadido al *FrameLayout*. Se mantiene en un *Bundle* para poder recuperar su valor en cualquier momento de su ciclo de vida, ya que este valor se utilizará a la hora de crear la vista a mostrar. Mediante el método `onCreate()` nos aseguramos de que el número es convenientemente rescatado del *Bundle* y guardado como variable miembro para usarlo posteriormente. Tenga en cuenta que en este ejemplo pueden añadirse tantos *Fragment* como se desee, y si el usuario desea volver a un *Fragment* anterior, debe encontrarlo tal y como lo dejó, es decir con su número de orden de creación informado. Por último devolvemos a través de `onCreateView()` la vista que se quiere mostrar al usuario. Tal y como comentamos, se mostrará un *layout* u otro dependiendo de si es par o impar, hecho que conoceremos mediante la simple comprobación `mNum % 2 == 0`. La manera de obtener el *layout* es mediante el uso del objeto *LayoutInflater*, que con el método `inflate()` permite inflar la vista asignada como parámetro.

Para poder probar la aplicación queda implementar el método `addFragment()`. En él se debe crear una nueva instancia del *Fragment* y lo que haremos es remplazar la existente en lugar de añadir una nueva, con tal de conservar mejor los recursos; esto se hace mediante el método `replace()` de la clase *FragmentTransaction*. Para mejorar el aspecto de la aplicación es posible usar animaciones propias a la hora de realizar las transacciones, esto se haría usando el método `setCustomAnimation()`, pero en nuestro caso

usaremos una animación estándar, utilizando para ello el método `setTransition()`, donde se le puede indicar el tipo de transición por defecto a utilizar. Por último, como queremos que se pueda navegar entre los distintos *Fragment* mostrados, lo que haremos será registrar la transacción en la pila de transacciones realizadas; mediante `addToBackStack()` decimos que esta transacción sea grabada y que pueda "deshacerse" mediante el botón back o mediante programación y así volver al estado anterior; o dicho de otro modo para volver al *Fragment* anterior. En esta llamada se puede pasar como parámetro una cadena que servirá de identificador único para más adelante trabajar con la pila de fragmentos y rescatarlos, en este caso no lo hacemos y simplemente informamos un *null*. El método `addFragment()` de la clase *FragmentTest*, quedaría:

```
void addFragment() {
      mStackPosition++;
   // Instanciamos nuevo Fragment
   Fragment newFragment = SimpleFragment.newInstance(mStackPosition);
   // Se añade el Fragment a la actividad
   FragmentTransaction ft = getFragmentManager().beginTransaction();
   ft.replace(R.id.fragmentShow, newFragment);
   ft.setTransition(FragmentTransaction.TRANSIT_FRAGMENT_FADE);
   // añadimos la transación a la pila
   ft.addToBackStack(null);
   ft.commit();
}
```

Antes de lanzarse a probar la aplicación, es recomendable llamar al método `onSaveInstanceState()`, donde guardaremos la última posición utilizada y así poderla recuperar en el método `onCreate()` en caso de ser necesario.

```
@Override
public void onSaveInstanceState(Bundle outState) {
super.onSaveInstanceState(outState);
      outState.putInt("position", mStackPosition);
}
```

Si ejecuta la aplicación sobre un dispositivo con Android mayor que 3.0, podrá ver la aplicación mostrando el icono de la misma en grande y a medida que pulse sobre el botón Nuevo Fragment se irán añadiendo nuevos *Fragment* y a su vez irá cambiando su contenido. Para volver a un *Fragment* anterior vale con pulsar el botón Back del dispositivo (véase figura 10.5).

Pero los *Fragment* sirven para ser utilizados en muchos más contextos, por ejemplo usarlos como pantallas de diálogo para mostrar información a lo largo del programa. Vamos a modificar el ejemplo y mostraremos dos modos de trabajar con diálogos; uno de ellos ya ha sido visto en otros ejemplos durante el libro y se trata de mostrar el diálogo a través de la clase *AlertDialog*

asignándole directamente los botones, mientras que en el segundo método mostraremos la manera de hacerlo usando un *Fragment*. Esta última manera, como podrá comprobar, proporciona mucha mayor flexibilidad a la hora de configurar la pantalla, ya que podemos utilizar directamente un *layout* o parte de él en su diseño, con lo que la pantalla podrá tener el aspecto que quiera. Los diálogos mostrados en el ejemplo tendrán tres botones, uno para añadir un nuevo *Fragment* a la aplicación (donde aprovecharemos el método ya existente), otro para ir hacia atrás en la pila de transacciones realizadas con los *Fragment* y un tercero para cancelar y no hacer nada. Además y por darle un poco de color le añadiremos un pequeño gráfico (el que usamos como icono en la aplicación).

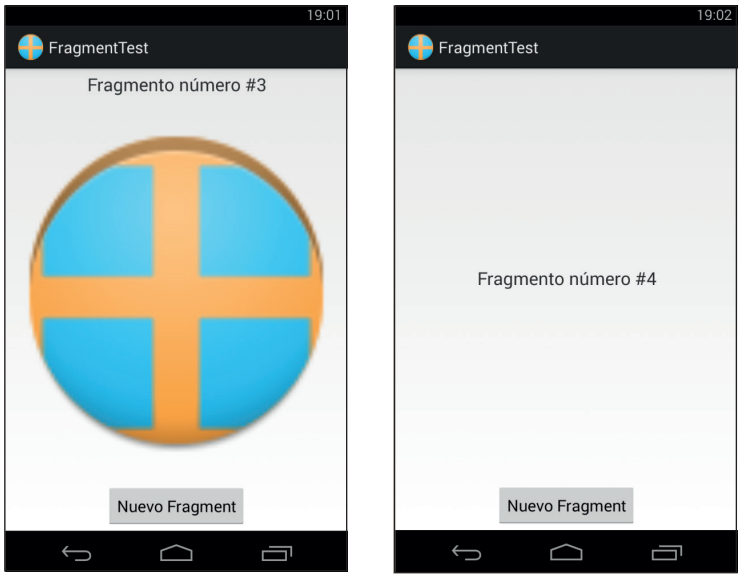

Figura 10.5. Aplicación sobre teléfono mostrando diferentes fragmentos.

Comenzaremos añadiendo un par de botones al *layout* de la aplicación que se encargarán de lanzar cada uno de los distintos modos de diálogo. Para ello modifique el archivo `activity_fragment_test.xml` para que quede:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:orientation="vertical" android:padding="4dip"
   android:gravity="center_horizontal"
   android:layout_width="match_parent" android:layout_height="match_parent">
   <FrameLayout
      android:id="@+id/fragmentShow"
      android:layout_width="match_parent"
      android:layout_height="match_parent"
```

```
      android:layout_weight="0.5" >
   </FrameLayout>

   <Button android:id="@+id/newFragment"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Nuevo Fragment">
   </Button>
   <LinearLayout
      android:layout_height="match_parent"
      android:layout_width="match_parent"
      android:layout_weight="1">
   <Button android:id="@+id/showDialogF"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Diálogo Fragment"
      android:layout_weight="1"
      />
   <Button android:id="@+id/showDialog"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Diálogo Normal"
      android:layout_weight="1"
      />
   </LinearLayout>
</LinearLayout>
```

Para el *layout* a mostrar por el *Fragment*, utilizaremos un fichero de *layout* propio llamado `dialog.xml` cuyo contenido será un icono, una pequeña descripción para el usuario y los tres botones mencionados anteriormente, que son el de añadir un *Fragment*, ir hacia atrás en la pila de transacciones y el de cancelar:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:orientation="vertical" >
   <RelativeLayout
      android:id="@+id/linearLayout1" android:layout_width="match_parent"
      android:layout_height="wrap_content" >
      <ImageView  android:id="@+id/image"
         android:layout_width="wrap_content" android:layout_height="wrap_
         content"
         android:layout_alignParentLeft="true" android:src="@drawable/ic_
         launcher" />
         <TextView android:id="@+id/textView1"
            android:layout_width="wrap_content" android:layout_height="wrap_
            content"
            android:layout_centerVertical="true" android:layout_
            marginLeft="10dip"
            android:layout_toRightOf="@+id/image" android:text="Selecciona
            acción a realizar" >
        </TextView>
    </RelativeLayout>
```

```
    <LinearLayout android:id="@+id/linearLayout1"
        android:layout_width="match_parent" android:layout_height="wrap_
        content" >
        <Button android:id="@+id/newFrag"
            android:layout_width="0dip" android:layout_height="wrap_content"
            android:layout_weight="1" android:text="Nuevo" />
        <Button android:id="@+id/back"
            android:layout_width="0dip" android:layout_height="wrap_content"
            android:layout_weight="1" android:text="Atrás"/>
        <Button android:id="@+id/cancel"
            android:layout_width="0dip" android:layout_height="wrap_content"
            android:layout_weight="1" android:text="Cancelar"/>
    </LinearLayout>
</LinearLayout>
```

Es hora de añadir código para mostrar los diálogos. Lo primero es recuperar los botones de la pantalla principal para asignarles código. Para ello, dentro del método `onCreate()` de la clase *FragmentTest* añadimos las siguientes líneas:

```
//botón para mostrar dialogo AlertDialog
Button buttonDialog = (Button)findViewById(R.id.showDialog);
buttonDialog.setOnClickListener(new View.OnClickListener() {
public void onClick(View v) {
         showDialog();
  }
      });
//botón para mostrar dialogo Fragment
Button buttonDialogF = (Button)findViewById(R.id.showDialogF);
buttonDialogF.setOnClickListener(new View.OnClickListener() {
public void onClick(View v) {
         showDialogF();
  }
      });
```

El método `showDialog()` construirá el diálogo mediante la clase *AlertDialog* añadiéndole directamente tres botones. En el primero de los botones llamaremos al método `addFragment()` para añadir nuevos elementos a la pantalla, en el segundo no haremos nada y servirá para cancelar la acción y por último el tercer botón servirá para moverse hacia atrás en la pila de transacciones de los *Fragments*. Para obtener la transacción anterior registrada en la pila se necesita acceder al *FragmentManager* con `getFragmentManager()` y tras ello pedirle que navegue a dicha transacción mediante `popBackStack()`. El método al completo que realiza todas estas acciones es:

```
void showDialog() {
   new AlertDialog.Builder(this)
   .setIcon(R.drawable.ic_launcher)
   .setTitle("Selecciona acción a realizar")
   .setPositiveButton("Nuevo",
```

```
        new DialogInterface.OnClickListener() {
        public void onClick(DialogInterface dialog, int whichButton) {
           addFragment();
        }
        }
           )
    .setNegativeButton("Cancelar",
        new DialogInterface.OnClickListener() {
        public void onClick(DialogInterface dialog, int whichButton) {
        // no hacer nada
        }
        }
           )
    .setNeutralButton("Atrás",
        new DialogInterface.OnClickListener() {
        public void onClick(DialogInterface dialog, int whichButton) {
           getFragmentManager().popBackStack();
              }
              }
           )
    .create().show();
}
```

Para el caso de trabajar con fragmentos en diálogos, deberemos obtener una instancia de una clase que extienda la clase *DialogFragment* y exponerla al usuario. En el ejemplo se muestra también como pasar parámetros a la instancia en caso de que se quisieran utilizar.

```
void showDialogF() {
      DialogFragment newFragment = MyAlertDialogFragment.newInstance(
         "Cadena de ejemplo como parámetro");
      newFragment.show(getFragmentManager(), "dialog");
}
```

Por último queda crear la clase *MyAlertDialogFragment* que debe extender la clase *DialogFragment*. La crearemos dentro de la clase *FragmentTest*.

```
public static class MyAlertDialogFragment extends DialogFragment {
public static MyAlertDialogFragment newInstance(String valor) {
   MyAlertDialogFragment frag = new MyAlertDialogFragment();
   // podemos aprovechar para pasar parámetros mediante bundle
   /*
   Bundle bundle = new Bundle();
   bundle("clave", valor);
   frag.setArguments(bundle);
   */
   return frag;
}

@Override
public Dialog onCreateDialog(Bundle savedInstanceState) {
   LayoutInflater inflater = (LayoutInflater)  getActivity().
getSystemService(LAYOUT_INFLATER_SERVICE);
```

```
    ViewGroup dlgview = (ViewGroup) inflater.inflate(
                R.layout.dialog, null);
    // botón nuevo Fragment
       Button buttonShow = (Button)dlgview.findViewById(R.id.newFrag);
    buttonShow.setOnClickListener(new View.OnClickListener() {
    public void onClick(View v) {
                    ((FragmentTest)getActivity() ).addFragment();
                }
                 });
    // botón cancelar
    Button buttonCancel = (Button)dlgview.findViewById(R.id.cancel);
    buttonCancel.setOnClickListener(new View.OnClickListener() {
                public void onClick(View v) {
                    dismiss();
    }
    });
    // botón ir a Fragment anterior
    Button buttonBack = (Button)dlgview.findViewById(R.id.back);
    buttonBack.setOnClickListener(new View.OnClickListener() {
    public void onClick(View v) {
                    getFragmentManager().popBackStack();
    }
    });

       // asignar el dialog a la vista
       return new AlertDialog.Builder(getActivity()).setView(dlgview).
       create();
}
}
```

En el método `newInstance()` de la clase se obtiene el valor pasado por parámetro y éste se podría guardar como parte de un *Bundle* para utilizarlo más adelante durante la vida de la clase. Se ha dejado comentado ya que no tiene otro uso más que el didáctico.

Es en el método `onCreateDialog()` donde se infla el *layout* seleccionado para mostrar como diálogo; esto se hace mediante la instancia del *LayoutInflater* tal y como se ha hecho en otras ocasiones. Una vez obtenido un objeto *View* a partir del *layout*, se le asigna el código a cada uno de los botones para que realicen las funciones necesarias y descritas anteriormente.

Por último se construye el diálogo mediante la clase ya conocida *AlertDialog* a la que se le asigna como vista el *View* obtenido al principio del método.

Si prueba ahora la aplicación, podrá lanzar los distintos diálogos mediante los dos nuevos botones. A diferencia del diálogo creado directamente mediante el *AlertDialog*, el diálogo generado mediante *Fragment* no se elimina de pantalla tras pulsar una tecla, sino que se mantiene a la vista hasta pulsar sobre el botón Cancelar. Para que se eliminara tras cada pulsación valdría con añadir la llamada al método `dismiss()` tras cada una de las llamadas a los otros dos botones.

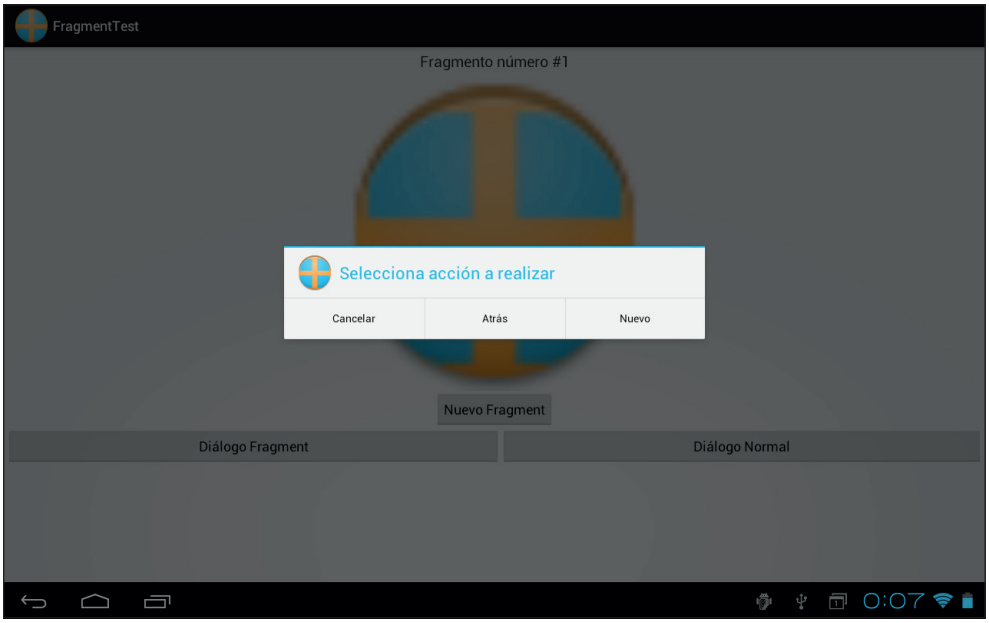

Figura 10.6. Diálogo de la aplicación.

11 Persistencia básica

En este capítulo aprenderá a:

- Guardar datos de diferentes maneras en el dispositivo.
- Manejar fragmentos.
- Seleccionar la manera de almacenar los datos dependiendo de las circunstancias.
- Acceder a las memorias externas como la tarjeta SD.
- Enviar y recibir datos por red.

Lo más normal es que las aplicaciones trabajen con datos; datos que muy probablemente se quieran guardar entre distintas ejecuciones de la aplicación, entre distintos usuarios... lo que se busca es mantener datos en el dispositivo para poder trabajar con ellos en cualquier momento. Los datos pueden ser de cualquier índole, desde los records de un juego, a la lista de contactos, valores de las acciones, todo lo que se le ocurra. Lejos quedan aquellos dispositivos móviles en los que al acabarse la batería se perdía toda la información ahora Android ofrece distintas posibilidades a la hora de guardar información, dependiendo de la naturaleza y uso que se les quiera dar, por ejemplo tamaño, privacidad, facilidad de acceso...
Según su mecanismo de acceso, estos métodos los podemos clasificar en:

- Preferencias
- Ficheros
- Red
- Bases de datos

Preferencias

Es una manera muy sencilla pero potente y cómoda de guardar datos en el dispositivo. Los datos se guardan parejas de clave-valor, es decir se guarda un valor dándole un nombre y se recupera más adelante a través de ese mismo nombre. Con este método solamente es posible guardar datos de tipos primitivos: `int`, `boolean`, `float`, `long` y `String`. Para gestionar este tipo de persistencia y acceder a todo su potencial, se accede a través de la clase *SharedPreferences*. El objeto *SharedPreferences* puede ser obtenido mediante dos llamadas distintas del contexto:

- **getSharedPreferences():** Se debe usar este método si se tiene pensado que la aplicación mantenga varios bloques de preferencias que se distinguirán por su nombre. Por ejemplo si tuviéramos un juego donde se puedan guardar los datos (nombre y última puntuación por ejemplo) del jugador 1 y del jugador 2, podemos guardarlos en bloques separados. Sería como guardarlos en dos ficheros distintos.
- **getPreferences():** Si a diferencia del caso anterior solamente interesa guardar un bloque de propiedades, se debe utilizar este método; se puede usar el anterior, pero este es más sencillo.

Ambos métodos tienen como parámetro el modo en el que se accede. Este modo indica la privacidad que va a acompañar a los datos, siendo posible:

- Context.MODE_PRIVATE (valor 0): Es el modo por defecto y hace que los datos solamente sean accesibles por la propia aplicación.
- Context.MODE_WORLD_READABLE (valor 1): Permite que otras aplicaciones tengan acceso de lectura a los datos guardados.
- Context.MODE_WORLD_WRITABLE (valor 2): Permite al resto de aplicaciones tener control total sobre los datos guardados.

Para probar este método de persistencia, cree un nuevo proyecto con los siguientes datos:

- Application name: Preferencias
- Module Name: Preferencias
- Package name: com.acme.preferences
- Activity Name: MainActivity
- Aditional Features: Include Blank Fragment

Figura 11.1. Pantalla de introducción de datos.

Es este proyecto crearemos una pantalla donde podamos indicar la clave y el valor de cada dato que se quiera guardar, y mediante la pulsación de un botón, los guardaremos o recuperaremos. Además se ofrecerá la posibilidad de utilizar cualquiera de los métodos de grabación vistos anteriormente, dependiendo de si se informa o no un campo con el nombre de las *SharedPreferences*.

Trabajaremos con fragmentos, así que es importante seleccionar la opción Include Blank Fragment del desplegable Aditional Features de la última pantalla del asistente. La pantalla que se quiere obtener tiene el aspecto de la figura 11.1. Modificamos entonces el *layout* de la pantalla para que concuerde con la apariencia que se está buscando. Esta vez trabajaremos desde el *Fragment*, asique variamos el *layout* `fragment_main.xml` para que tenga de contenido:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:orientation="vertical" android:layout_width="fill_parent"
   android:layout_height="fill_parent">
   <TextView android:id="@+id/TextView01" android:layout_width="fill_parent"
      android:layout_height="wrap_content" android:text="@string/block_name" />
   <EditText android:layout_width="fill_parent"
      android:layout_height="wrap_content" android:id="@+id/edBlock"
      android:text="" android:background="@android:drawable/editbox_
      background" />
   <TextView android:id="@+id/TextView01" android:layout_width="fill_parent"
      android:layout_height="wrap_content" android:text="@string/key" />
   <EditText android:layout_width="fill_parent"
      android:layout_height="wrap_content" android:id="@+id/edKey"
      android:text="" android:background="@android:drawable/editbox_
      background" />
   <TextView android:layout_width="fill_parent"
      android:layout_height="wrap_content" android:text="@string/value" />
   <EditText android:layout_width="fill_parent"
      android:layout_height="wrap_content" android:id="@+id/edValue"
      android:text="" android:background="@android:drawable/editbox_
      background" />
   <LinearLayout android:id="@+id/LinearLayout01" android:layout_width="fill_
   parent" android:layout_height="wrap_content" android:orientation="horizontal"
   android:gravity="center" android:paddingTop="10dp">
      <Button android:text="@string/save" android:id="@+id/btnSalvar"
      android:layout_width="wrap_content" android:layout_height="wrap_content"/>
      <Button android:text="@string/retrieve" android:id="@+id/btnReq"
      android:layout_width="wrap_content" android:layout_height="wrap_content"/>
   </LinearLayout>
</LinearLayout>
```

Y modificamos el archivo `stings.xml`. Aprovechamos para incluir no sólo las constantes que se usan en el *layout*, sino también las que se utilizarán como mensajes de error a lo largo de la aplicación. Añada al contenido del fichero :

```
<resources>
   <string name="app_name">Preferencias</string>
   <!--  layout -->
   <string name="block_name">Nombre de bloque</string>
   <string name="key">Clave</string>
    <string name="value">Valor</string>
   <string name="save">Salvar</string>
   <string name="retrieve">Recuperar</string>
   <!-- alertas -->
   <string name="no_key">No se ha introducido una clave</string>
   <string name="no_value">No se ha introducido un valor</string>
```

```
    <string name="default_value">No existe ese valor</string>
    ...
```

Como puede observar en la interfaz, existe una primera caja de texto donde el usuario podrá indicar el nombre del bloque donde guardar las preferencias, si es que así lo desea. Después se tienen dos cajas de texto más, una de ellas para guardar la clave del valor que se quiere almacenar y en la otra el valor en sí. Por último existen dos botones, uno de ellos para guardar los datos y otro para recuperarlos. Lo que se hará es recuperar el valor y mostrarlo sobre la misma caja de texto que se ha utilizado para introducirlo.
En la parte del código vamos a trabajar desde el *Fragment*; en la clase *PlaceholderFragment* dentro de la clase *MainActivity* crearemos unas variables de tipo miembro para guardar la referencia a los *EditText* de la interfaz, de modo que podamos usarlas en varios métodos sin tener que obtenerla cada vez.

```
EditText block = null;
EditText key = null;
EditText value = null;
```

En el método `onCreateView()` será donde se obtendrán las referencias a estos *EditText* a partir del identificador que se le ha dado a cada uno en el *layout*.

```
block = (EditText) rootView.findViewById(R.id.edBlock);
key = (EditText) rootView.findViewById(R.id.edKey);
value = (EditText) rootView.findViewById(R.id.edValue);
```

Y asociamos el código para los botones de salvar y recuperar dentro del mismo método:

```
// asignamos código al botón de salvar
Button btnSalvar = (Button) rootView.findViewById(R.id.btnSalvar);
btnSalvar.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        String blockValue = block.getText().toString().trim();
        String keyValue = key.getText().toString().trim();
        String valueValue = value.getText().toString().trim();
        ((MainActivity)getActivity()).save(blockValue,keyValue,valueValue);
    }
});
// asignamos código al botón de recuperar
Button btnRecuperar = (Button) rootView.findViewById(R.id.btnReq);
btnRecuperar.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        String blockValue = block.getText().toString().trim();
        String keyValue = key.getText().toString().trim();
        String valueValue = ((MainActivity)getActivity()).
        retrieve(blockValue,keyValue);
        value.setText(valueValue);
    }
});
```

Dentro de los métodos `onClick()` correspondientes a los botones, obtenemos los valores a pasar como parámetros a los métodos de guardar y recuperar; la llamada a éstas funciones se realiza obteniendo la referencia de la actividad en la que se encuentra el *Fragment* mediante `getActivity()` y realizando un *cast* a la clase *MainActivity* .

Las funciones con la lógica para salvar y recuperar, las crearemos como métodos de la propia clase *MainActivity*.

El código correspondiente para la función `save()` sería:

```
protected void save(String blockValue,  String keyValue,String valueValue) {
      // comprobar si ha informado la clave, de lo contrario se retorna
      if ("".equals(keyValue)){
         showMessage(R.string.no_key);
         return;
      }
  //  comprobar si ha informado el valor, de lo contrario se retorna
      if ("".equals(valueValue)){
         showMessage(R.string.no_value);
         return;
      }
      //salvar preferencias
      SharedPreferences preferences = null;

      if (!"".equals(blockValue)){
         preferences = getSharedPreferences(blockValue, 0);
      }
      else{
         preferences = getPreferences(0);
      }

      SharedPreferences.Editor editor = preferences.edit();
   editor.putString(keyValue, valueValue);
   // guardar el trabajo
   editor.commit();
  }
```

Se comienza comprobando que los parámetros pasados a la función están informados, en caso de no estarlo, se mostrará un mensaje de error y se retornará de la función. Caso especial es el parámetro `blockValue`, se comprueba si contiene datos para saber qué tipo de llamada se tiene que hacer para obtener el objeto *SharedPreferences*, si mediante nombre o no. Una vez se tiene el objeto *SharedPreferences*, se necesita un editor para poder modificar su contenido, que se hace mediante la llamada al método `edit()` del objeto *SharedPreferences*.

El editor ofrece distintos métodos dependiendo del tipo de dato que se le quiera asignar. Por último se deben guardar los cambios hechos sobre el editor mediante la sentencia `commit()`.

En cuanto al método `retrieve()`:

```
protected String retrieve(String blockValue,String keyValue) {
   // comprobar si ha informado la clave, de lo contrario se retorna
   if ("".equals(keyValue)){
      showMessage(R.string.no_key);
      return "";
   }
   SharedPreferences preferences = null;

   if (!"".equals(blockValue)){
      preferences = getSharedPreferences(blockValue, 0);
   }
   else{
      preferences = getPreferences(0);
   }
   //obtenemos el valor, y si no existe mostramos el valor por defecto
   String valueValue = preferences.getString(keyValue,      getResources().
getString(R.string.default_value));

   // devolvemos el valor
   return valueValue;
}
```

El código es bastante semejante al método `save()` pero en este caso no tenemos que recuperar el valor del dato guardado, sino informarlo con lo que se haya obtenido al intentar recuperar el valor del sistema de preferencias. Del mismo modo que el editor disponía de varios métodos para informar el valor dependiendo del tipo de dato informado, el objeto *SharedPreferences* tiene distintos métodos para recuperar diferentes tipos de datos; en este caso se usa `getString()` para poder recuperar el valor que era una cadena de texto. A la hora de hacer la llamada a cualquiera de los métodos de recuperación de valor, se le deben informar dos parámetros, el primero de ellos es la clave del valor que se desea recuperar y el segundo es el valor a entregar en caso de que no se haya recuperado nada; en el código anterior, se devuelve una constante definida en el archivo `strings.xml`. Por último se devuelve el valor obtenido para que pueda actualizase el contenido de la caja de texto en pantalla (que puede ser el guardado en las preferencias o el valor por defecto).

Para mostrar los mensajes de alerta, utilizaremos las ventanas *Toast* (tostada), que son pequeños mensajes emergentes que aparecen durante un breve espacio de tiempo y que ya se han usado en otras ocasiones.

```
private void showMessage(int message){
   Context context = getApplicationContext();
   CharSequence text = getResources().getString(message);
   int duration = Toast.LENGTH_SHORT;
   Toast toast = Toast.makeText(context, text, duration);
   toast.show();
}
```

El programa está listo para ser probado. Se le puede dar un valor a la entrada clave y otro a la entrada valor y pulsar sobre **Salvar**. Si se borra la entrada valor y se pulsa sobre recuperar, veremos que vuelve a poner el valor que se le había dado anteriormente. Hay que tener en cuenta que si un valor se guarda bajo la dupla nombre de bloque y nombre de clave, se ha de recuperar con la misma dupla, es decir especificando tanto nombre de bloque como clave, de lo contrario aparecerá el texto por defecto. Ahora si se guarda un valor bajo una clave, ya se puede reiniciar el dispositivo que el valor permanecerá accesible la siguiente vez que se quiera acceder a él; por ejemplo, guarde un valor, salga de la aplicación o reinicie el dispositivo, vuelva a la aplicación y trate de recuperarlo… el valor sigue allí.
Del mismo modo que se pueden guardar elementos mediante el objeto *SharedPreferences*, también es posible borrarlos. Para esta tarea, el objeto Sh*aredPreferences.Editor* posee un método llamado `remove()`, al que se le informa como parámetro la clave de la entrada que se quiere eliminar. Queda como ejercicio para el lector añadir nuevos mensajes de información avisando de que se ha guardado o recuperado el dato y añadir la lógica para eliminar elementos guardados mediante *SharedPreferences*.

Ficheros

Aunque Android ofrece la posibilidad de guardar los datos perfectamente estructurados mediante el método visto anteriormente o mediante bases de datos, en ocasiones es preferible guardar la información en un simple fichero. Con Android podemos acceder a ficheros que se hayan empaquetado junto con la aplicación y a ficheros creados en medios externos como puede ser la tarjeta SD.

Ficheros de recurso

¿Le gustaría tener la lista de los Reyes Godos en su dispositivo? ¿Tentador verdad? Vamos a ello. Podríamos optar por generar la lista a través del archivo de recursos `strings.xml` utilizando la etiqueta `<string-array>`. Dentro de este elemento se definiría cada una de las entradas que debe ser tomada como elemento del *array*, que en este ejemplo es el nombre del rey junto con los años de reinado.

```
<string-array name="REYES_GODOS">
   <item>Recaredo (586-601)</item>
   <item>Liuva II (601-603)</item>
   <item>Witérico (603-610)</item>
```

```
...
<item>Rodrigo (710-711)</item>
</string-array>
```

Los *arrays* así creados, están disponibles como un recurso más dentro de la clase *R* y pueden ser recuperados en tiempo de programación mediante `R.array.nombre_array` tal por ejemplo para este caso sería:

```
String[] godos = getResources().getStringArray(R.array.REYES_GODOS);
```

Siguiendo esta manera de actuar, podríamos tener distintos *arrays* para diferentes idiomas, igual que con las cadenas de texto.
También sería posible distribuir esta lista mediante un fichero aparte, que será el punto de partida para el ejemplo para este punto. Como vamos a tratar con un archivo crudo (sin formato, aunque usemos en nuestro caso un XML, realmente podría ser texto plano, binario…), lo más sencillo es guardarlo en el directorio `/src/main/res/raw`. Como el fichero se encuentra dentro del directorio `res`, podremos acceder a él a través de la llamada `getResources()`, que retorna una referencia al objeto *Resources* que a su vez ofrece el método `openRawResource()`, que será quien realmente nos de acceso al fichero. Como parámetro de la llamada `openRawResource()` se espera un identificador en lugar de una ruta a fichero como se podría suponer (trabajaríamos con una ruta si hubiéramos guardado el fichero en la ruta `/src/main/assets`) y es que, todo elemento dentro del directorio `/src/main/res` tiene un identificador único en la clase *R*. Al abrir el recurso, el sistema devuelve un *InputStream*, lo que implica que la única acción posible sobre el recurso será su lectura, es decir, que es no es posible su modificación desde el programa. La única manera de modificar estos ficheros es haciéndolo desde el entorno de desarrollo, crear un nuevo ejecutable e instalar la nueva versión de la aplicación.
Hagamos un nuevo proyecto para probar el acceso a ficheros. Créelo con los siguientes parámetros:

- Application name: Ficheros
- Module Name: Ficheros
- Package name: com.acme.files
- Activity Name: ResourceFile

Cree el directorio `/src/main/res/raw` en el proyecto y añada el fichero `godos.xml` con el siguiente contenido:

```
<godos>
   <godo>Recaredo (586-601)</godo>
   <godo>Liuva II (601-603)</godo>
   <godo>Witérico (603-610)</godo>
   <godo>Gundemaro (610-612)</godo>
```

```
    <godo>Sisebuto (612-621)</godo>
    <godo>Recaredo II (621)</godo>
    <godo>Suínthila (621-631)</godo>
    <godo>Sisenando (631-636)</godo>
    <godo>Khíntila (636-639)</godo>
    <godo>Tulga (639-642)</godo>
    <godo>Khindasvinto, rey único (642-649)</godo>
    <godo>Khindasvinto y Recesvinto (649-653)</godo>
    <godo>Recesvinto, rey único (653-672)</godo>
    <godo>Wamba (672-680)</godo>
    <godo>Ervigio (680-687)</godo>
    <godo>Egica, rey único (687-698/700)</godo>
    <godo>Egica y Witiza (698/700-702)</godo>
    <godo>Witiza, rey único (702-710)</godo>
    <godo>Rodrigo (710-711)</godo>
</godos>
```

Lo siguiente que haremos es definir el *layout* a utilizar. Creamos un nuevo *layout* con el nombre `activity_resource_file.xml` y como mostraremos el resultado en una lista, le añadimos el elemento `<ListView>`. Cuando se trabaja con listas en Android, podemos proceder de varios modos en cuanto a actividades y *layouts*. Si nuestra actividad extiende la clase *ListActivity,* heredará varios métodos de ayuda para la gestión de la lista, siempre y cuando ésta tenga el atributo de identificación `android:id="@android:id/list"`. Igualmente se puede trabajar con una actividad heredada de *ListActivity* y que no se le asigne *layout* alguno; en este caso, la vista que se mostrará será una lista que ocupará toda la pantalla (o el fragmento de ella). Si se trabaja con la actividad heredada de *ListActivity* y se le asigna *layout*, **obligatoriamente** debe existir el `<ListView>` con el atributo `android:id="@android:id/list"`; de lo contrario dará error durante la ejecución. También podemos optar por hacer todo a mano, es decir, trabajar con una clase que no extienda de *ListActivity* y un *layout* que su lista no tenga el identificador `android:id="@android:id/list"`; en caso de trabajar con más de una lista en el *layout* esta es la opción más adecuada.

Para el ejemplo extenderemos la clase *ListActivity*, es decir, se debe indicar como identificador del elemento `<ListView>` el atributo `android:id="@android:id/list"`. Para las filas de la lista, podemos usar *layout* propios, pero esto lo veremos más adelante, en este caso utilizaremos uno de los que vienen por defecto, el `android.R.layout.simple_list_item_1`. Android tiene especificados varios *layouts* que son de uso frecuente, por ejemplo para las listas en las que cada fila mostrarán un texto (como se puede ver es algo muy frecuente), Android dispone del *layout* `android.R.layout.simple_list_item_1`, en caso de ser dos elementos entonces se podría usar el *layout* `android.R.layout.simple_list_item_2`, o el `android.R.layout.simple_list_item_checked` en caso de querer la lista con una caja de selección …

El contenido del fichero de *layout* `activity_resource_file.xml` quedaría:

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

En cuanto al código para gestionarlo, lo haremos una vez más sobre el evento `onCreate()` de la clase principal *ResourceFile*. Como vamos a utilizar listas, haremos que la clase extienda a *ListActivity* en lugar de extender *Activity*. Lo que se hará es recuperar el contenido del fichero y guardarlo en un *ArrayList*, para más adelante usarlo como adaptador para la lista.

En el método `onCreate()`, indicamos el *layout* a utilizar y preparamos el *ArrayList* que contendrá la lista de godos una vez leída del fichero:

```
super.onCreate(savedInstanceState);
   setContentView(R.layout.activity_resource_file);
   ArrayList<String> godos = new ArrayList<String>();
```

Para leer el fichero haremos un bloque `try/catch` para informar si algo ha ido mal. En primer lugar se obtiene el *InputStream* del recurso y se construirá el documento XML con su contenido:

```
try {
//obtenemos el recurso
InputStream in = getResources().openRawResource(R.raw.godos);
   //generamos el documento XML
DocumentBuilder builder = DocumentBuilderFactory.newInstance().
newDocumentBuilder();
Document doc = builder.parse(in,null);
```

Una vez se tiene el documento se pueden extraer los nodos que hemos llamado `<godo>` y se recorren para incluirlos en el *ArrayList* que se usará para la lista, sin olvidar de cerrar el *InputStream* tras su lectura:

```
NodeList fileGodos = doc.getElementsByTagName("godo");
// se recorren los  nodos guardándolos en el array
for (int i = 0; i < fileGodos.getLength(); i++){
godos.add(((Element)fileGodos.item(i)).getTextContent());
}
in.close();
}
catch(Throwable t){
Toast.makeText(this, "Error:" + t.getMessage(), Toast.LENGTH_LONG).show();
}
```

Por último, se debe crear un adaptador para que la lista pueda mostrar los elementos. Existen varios tipos de adaptadores para poder gestionar de manera diferente cada fuente de datos, por ejemplo en este caso como lo que queremos mostrar en la lista es un *array* usaremos la clase *ArrayAdapter* como adaptador (aunque hay otros como *CursorAdapter, ResourceCursorAdapter, SimpleAdapter, WrapperListAdapter*...). Para la creación de este adaptador se le deben pasar como parámetros un contexto de aplicación, un *layout* a usar por cada fila y el *array* con los elementos a mostrar. Mediante el método `setListAdapter()` se le asigna a la clase el adaptador con todos los datos referentes a la lista y la manera en la que los tiene que mostrar. Este método viene heredado de la clase *ListActivity*, si hubiéramos usado una *Activity*, deberíamos obtener la referencia al objeto lista y ejecutar el método sobre dicho objeto. En caso de querer mostrar listas con celdas complejas, es muy probable que tengamos que crearnos nuestras propias clases *ListAdapter* como se verá más adelante.

En el mismo método `onCreate()` añadimos:

```
setListAdapter(new ArrayAdapter<String>(this, android.R.layout. simple_list_
item_1, godos));
```

Fíjese que el *layout* a utilizar en cada línea es el ofrecido por Android en lugar de crear nosotros uno propio.

Figura 11.2. Lista de reyes godos.

Ficheros externos

Uno de los hándicaps más importantes a la hora de trabajar con los ficheros de recurso es que no se pueden modificar, por lo que no sirven para guardar preferencias de usuario o datos de las aplicaciones. Para ello se dispone de funcionalidad de lectura y escritura sobre archivos externos a la aplicación, de modo semejante a como se hace en aplicaciones Java normales. Mediante las llamadas `openFileInput()` y `openFileOutput()` se obtienen un *InputStream* y un *OutputStream* respectivamente con los que trabajar. Lo que se debe hacer es obtener el *Stream* que se necesite, bien de lectura o bien de escritura, realizar las operaciones y por último cerrar el *Stream* recibido. Cuando dos aplicaciones acceden a la vez a un mismo fichero, cada una de ellas recibe una copia del mismo, pudiendo llegar a sobrescribirse; para evitar este problema, sobre todo si son archivos críticos (por ejemplo configuración del sistema), se puede optar por el uso de un objeto *ContentProvider* que sea el único que acceda al fichero y pueda gestionar bloqueos.

Como ejemplo de trabajo con ficheros externos se va a hacer un pequeño editor de textos y para ello se va a modificar el proyecto realizado para los ficheros de recurso de modo que podamos ejecutar dos actividades diferentes desde el *Launcher*. Hasta el momento, cada proyecto que se ha realizado tenía una sola entrada en el menú principal de Android, lo que se pretende ahora es que exista una entrada para acceder a la actividad de ficheros de recurso y otra al ejemplo de archivos externos; para ello modifique el fichero `AndroidManifest.xml` añadiéndole una nueva actividad (que más adelante crearemos).

```
<activity android:name=".ExternalFile"
android:icon="@drawable/icon_text"
            android:label="@string/app_name_ext" android:taskAffinity="com.
            acme.ExternalFile">
<intent-filter>
         <action android:name="android.intent.action.MAIN" />
         <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
</activity>
```

También hay que retocar la entrada de la actividad `ResourceFile` para dejarla:

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

La etiqueta `android:taskAffinity` sirve para poder determinar dos tareas distintas dentro del sistema Android, sino, al compartir la misma tarea, siempre se lanzaría la primera actividad definida, puesto que cada tarea sólo puede tener una actividad raíz; esto anteriormente no era así, pero se cambió para tener mayor control sobre las aplicaciones complejas. La actividad `ExternalFile` es muy semejante a la que ya existía en el documento `AndroidManifest.xml`, lo único que cambia son los atributos `android:name` y `android:label` dentro de la etiqueta `<activity>` y mediante su `<intent-filter>` lo configuramos para hacerlo disponible en el *Launcher*. También cambiaremos el texto a mostrar para la otra actividad, de modo que se diferencien mejor:

```
<activity android:name=".ResourceFile"
                  android:label="@string/app_name_res">
```

Nota:

Se ha añadido un icono a una de las entradas del Launcher *dejando la primera de ellas con el icono por defecto de la aplicación. Se podrían dejar ambas actividades con el mismo icono o especificar uno distinto dentro de cada definición de actividad en el* `AndroidManifest.xml`.

Modificamos el fichero `strings.xml` con las constantes necesarias que se acaban de utilizar en el `AndroidManifest.xml` y daremos valor a otras que se utilizarán como textos de botones y etiquetas en la pantalla que se definirá en breves instantes:

```
<resources>
      <string name="app_name">Ficheros</string>
      <string name="app_name_ext">Ficheros Externos</string>
      <string name="app_name_res">Ficheros Recurso</string>
      <string name="save">Salvar</string>
      <string name="load">Cargar</string>
      <string name="filename">Nombre del Fichero</string>
   ...
```

El editor de textos tendrá un campo donde el usuario podrá seleccionar el nombre del fichero con el que quiere guardar su documento, dos botones, uno para leer el fichero y otro para guardar y por último un campo de texto donde podrá introducir el texto a guardar. Lo llamaremos `external_file.xml`.

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

```
    <LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal" >
    <TextView android:text="@string/filename"  android:layout_width="wrap_
    content" android:layout_height="wrap_content"></TextView>
    <EditText  android:id="@+id/filename" android:layout_width="fill_parent"
    android:layout_height="wrap_content"></EditText>
</LinearLayout>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal" >
    <Button android:text="@string/save" android:id="@+id/save" android:layout_
    width="wrap_content" android:layout_height="wrap_content"/>
    <Button android:text="@string/load" android:id="@+id/load" android:layout_
    width="wrap_content" android:layout_height="wrap_content"/>
    </LinearLayout>
    <EditText android:id="@+id/editor" android:layout_width="fill_parent"
    android:layout_height="fill_parent" android:gravity="top"></EditText>
</LinearLayout>
```

En el `<EditText>` correspondiente al editor, se le ha puesto el atributo `android:gravity="top"` con tal de que el texto aparezca arriba del todo del editor, si no se hubiera puesto, entonces el texto aparecería en mitad del editor.

Una vez definida la entrada de la *Activity* en el `AndroidManifest.xml` y su *layout*, procederemos a crear la clase *Activity* correspondiente, llamada `ExternalFile.java` que extenderá la clase *Activity*. Dentro de su método `onCreate()` se asignará el código a ejecutar para los dos botones de salvar y leer fichero

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.external_file);
    Button btn = (Button) findViewById(R.id.save);
    btn.setOnClickListener(new View.OnClickListener() {
    @Override
        public void onClick(View v) {
            save();
                }
    });

    btn = (Button) findViewById(R.id.load);
    btn.setOnClickListener(new View.OnClickListener() {
    @Override
        public void onClick(View v) {
            load();
                }
    });
}
```

En los métodos de la clase *ExternalFile* será donde definamos la lógica efectiva para cada uno de los casos. Para salvar el documento se debe obtener el nombre del fichero con el que se quiere grabar y el contenido a salvar. Mediante la llamada a `openFileOutput()` y el nombre de fichero como parámetro, se obtendrá el *OutputStream* sobre el que escribir. La llamada `openFileOutput()` acepta como segundo parámetro un entero que indica el modo en el que se quiere abrir/crear el fichero. Los posibles valores vienen dados mediante constantes de la clase *Context*:

- MODE_APPEND: Para en caso de que exista el fichero, añadir datos en lugar de borrarlo y comenzar de nuevo.
- MODE_PRIVATE: Para indicar que el fichero sólo puede ser accedido desde la aplicación que lo está creando.
- MODE_WORLD_READABLE: El fichero puede ser accedido en modo lectura desde otras aplicaciones.
- MODE_WORLD_WRITABLE: El fichero puede ser accedido tanto en modo lectura como en escritura por otras aplicaciones.

En nuestro caso usaremos el valor por defecto: el MODE_PRIVATE.

```
protected void save() {
EditText editor = (EditText) findViewById(R.id.editor);
EditText filename = (EditText) findViewById(R.id.filename);
OutputStreamWriter out;
try {
out = new OutputStreamWriter(openFileOutput(filename.getText().toString(),
0));
out.write(editor.getText().toString());
out.flush();
out.close();
   showMessage("Se ha grabado el documento");
} catch (Throwable t) {
      showMessage("Error: " + t.getLocalizedMessage());
}
}
```

Para el caso de la lectura los pasos son muy semejantes, solamente teniendo en cuenta que debemos leer de un *InputStream* para recuperar el contenido del fichero y para ello nos ayudaremos de un *BufferedReader* que se irá leyendo e incluyendo su contenido en un *StringBuffer* que finalmente se convertirá en una cadena de texto que será mostrada por el objeto *EditText* de la interfaz.

```
protected void load() {
   EditText editor = (EditText) findViewById(R.id.editor);
   EditText filename = (EditText) findViewById(R.id.filename);
   InputStreamReader in;
```

```
    try {
       in = new InputStreamReader(openFileInput(filename.getText().
       toString()));
       BufferedReader buff = new BufferedReader(in);
       String strTmp = null;
       StringBuffer strBuff = new StringBuffer();
       while ((strTmp = buff.readLine())!=null){
          strBuff.append(strTmp + "\n");
       }
       in.close();
       editor.setText(strBuff.toString());
       showMessage("Se ha leido el documento");
    } catch (Throwable t) {
       showMessage("Error: " + t.getLocalizedMessage());
    }
}
```

Y no olvidemos la pequeña función de apoyo para mostrar mensajes al usuario

```
protected void showMessage(String message){
   Toast.makeText(this, message, Toast.LENGTH_LONG);
}
```

Al ejecutar la aplicación, podremos acceder desde el *Launcher* a las dos actividades definidas en el proyecto. Si se selecciona la denominada "Ficheros Externos" llegaremos hasta nuestro nuevo editor de textos, donde podremos escribir y guardar documentos. No se ha incluido código para tratar los errores, ni mostrar mensajes de guardado, ni otras muchas cosas que caben esperar en un editor, pero se deja como ejercicio al lector mejorar el aspecto, fiabilidad y ergonomía de la aplicación, que con lo aprendido hasta ahora podrá mejorarla con estos pequeños detalles. Para probarla, puede escribir en el área inferior un texto, asignarle un nombre de fichero en la caja de texto superior y guardar. Para ver que se recupera el texto guardado podemos modificar de nuevo el texto inferior y pulsar sobre el botón Cargar (con lo que aparecería el texto anterior), salir de la aplicación y comenzar de nuevo o añadir un nuevo botón para limpiar el texto introducido.

Nota:

Si tras ejecutar la aplicación y guardar los ficheros intenta buscarlos en el raíz de su memoria SD, se llevará una amarga sorpresa al ver que no están ahí y es que al usar los métodos `openFileInput()` *y* `openFileOutput()`*, estos se guardan en el espacio reservado para cada aplicación dentro del sistema Android, no de modo visible en la tarjeta SD.*

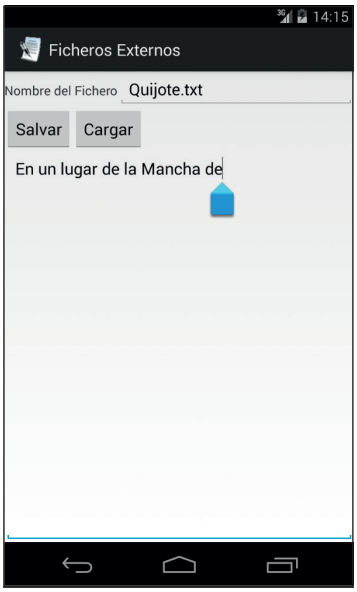

Figura 11.3. Pantalla del mini editor de textos.

Para trabajar con ficheros en las memorias externas, o en las memorias de los dispositivos pero que no son reservadas para las aplicaciones, se deben utilizar otra serie de métodos, y siempre se debe tener en cuenta que la tarjeta de memoria puede estar o no accesible (el usuario puede haberla extraído). Mediante la llamada a `Environment.getExternalStorageDirectory()` se obtiene un objeto *File* que apunta al directorio raíz de la memoria externa. A partir de ese momento se puede tratar como si se estuviera en una aplicación Java al uso. Para guardar el documento en un archivo sobre la memoria SD, el código quedaría:

```
EditText editor = (EditText) findViewById(R.id.editor);
EditText filename = (EditText) findViewById(R.id.filename);
OutputStreamWriter out;
try {
   File f = Environment.getExternalStorageDirectory();
   FileOutputStream fos = new FileOutputStream(new File (f, filename.
   getText().toString()));
   fos.write(editor.getText().toString().getBytes());
   fos.flush();
   fos.close();
   showMessage("Se ha grabado el documento");
} catch (Throwable t) {
   showMessage("Error: " + t.getLocalizedMessage());
}
```

Del mismo modo se actuaría para su lectura. El código presentado anteriormente no tiene ningún tipo de control sobre si se ha extraído o no la tarjeta, lo cual puede derivar en errores a la hora de escribir el fichero. Para conocer el estado de la memoria externa se puede utilizar el método `getExternalStorageState()` de la clase *Enviroment*.
Otro tema a tener en cuenta es que si usamos código anterior para guardar los ficheros, lo estaremos escribiendo en el raíz de la tarjeta SD, lo cual no es muy limpio y crearía "polución" en la tarjeta; lo más limpio es usar un directorio para almacenar de modo ordenado los archivos; este directorio en la memoria externa lo podemos fijar nosotros mediante programación o ayudarnos del método `getExternalFilesDir()` de la clase *Context*.

Red

El hecho de poder almacenar datos de una aplicación móvil utilizando la red hace que de alguna manera tenga un almacenamiento ilimitado. Está claro que el método de almacenamiento recae en la parte servidora, es decir en el servicio web, el FTP o lo que se utilice para cada caso.
Android pone a disposición de los programadores múltiples opciones, desde clases para el manejo del protocolo HTTP hasta la posibilidad de tratar directamente con *sockets*, pudiendo así implementar todo tipo de protocolos de comunicación de red.
Para obtener los datos de una conexión HTTP, se podría utilizar un código semejante a:

```
try {
URL url = new URL("http://myserver.com/myservice");
StringBuffer strBuff = new StringBuffer();
BufferedReader in = new BufferedReader(new InputStreamReader(url.
openStream(),"ISO-8859-1"));
String str;
while ((str = in.readLine()) != null){
strBuff.append(str);
}
in.close();
} catch (Throwable t) {
t.printStackTrace();
}
```

Donde al final tendremos en el *StringBuffer* el contenido de la dirección web a la que llamamos, por lo que podrá ser una página web estática o el resultado del procesado de los parámetros que se hayan podido pasar como parámetros dentro de la URL.

En caso de querer trabajar con sockets, el código podría ser algo como:

```
try {
InterfaceAddress server = Inet4Address.getByName("myserver");
Socket clientSocket = new Socket(server,80);
} catch (Throwable t) {
t.printStackTrace();
}
```

Por supuesto, no debemos olvidar que las conexiones a Internet pueden tardar en devolver los resultados, por lo que no deben ejecutarse en el hilo principal de la aplicación, sino que debemos de valernos de hilos secundarios, por ejemplo valiéndonos de la clase *AsyncTask*.

Nota:

No olvide que al trabajar con conexiones es necesario el permiso `INTERNET` *en el* `AndroidManifest.xml`*, mediante la entrada:*

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

Base de datos

Este método es el más completo de todos ya que ofrece mucha versatilidad y la posibilidad de guardar los datos estructurados para luego recuperarlos por ejemplo mediante consultas complejas. Su funcionamiento se verá en el próximo capítulo.

12 Base de datos

En este capítulo aprenderá a:

- Crear una base de datos.
- Gestionar las consultas y modificaciones de la base de datos.
- Obtener cursores con datos y aplicarlos a una lista.
- Definir y controlar menús contextuales.

Android pone a disposición de los programadores, de modo estándar, una base de datos SQLite completa que permite almacenar información estructurada en tablas para más adelante acceder a ella mediante sentencias de tipo SQL. Para facilitar su utilización, existen múltiples clases con ayudas al acceso a la base de datos, haciendo muy sencillo su uso y mantenimiento.

Principios

La manera en la que se aconseja trabajar con SQLite es crear una clase que extienda la clase *SQLiteOpenHelper* de modo que se encapsule toda la lógica de acceso a la base de datos y a su vez se tenga disponible todas las utilidades ofrecidas por Android para su manejo.

En el método `onCreate()` de la clase extendida es donde se debe establecer la creación de la base de datos, mientras que para el mantenimiento de la estructura, en caso de que se quieran añadir o quitar tanto tablas como campos, existe un método `onUpgrade()` que se ejecuta para pasar de una versión a otra de esquema de base de datos; dentro de este método será donde tengamos que eliminar tablas, crear nuevas o alterar las existentes con tal de ajustarse a la estructura de la nueva versión.

Para obtener un acceso en modo lectura a la base de datos, *SQLiteOpenHelper* ofrece el método `getReadableDatabase()` quien devuelve un objeto de tipo *SQLiteDatabase* sobre el que realizar las operaciones de consulta de datos; si se desea también poder escribir estos datos, la llamada a efectuar sería sobre el método `getWritableDatabase()`.

Desde la clase *SQLiteDatabase* es desde donde se realizarán las acciones para el control de los datos, pudiendo crear tablas, consultarlas, eliminar contenido, ejecutar comandos SQL y todas las tareas asociadas a un trabajo con base de datos. Para la realización de estas consultas, se exponen diversos métodos, de modo que una misma consulta se podría hacer de varias maneras, mediante asistentes, apoyándose sobre la clase *SQLiteQueryBuilder* o directamente con sentencias SQL. Una vez que se haya terminado de utilizar el objeto *SQLiteDatabase* obtenido por la aplicación, se debe cerrar su conexión mediante la llamada a su método `close()`.

Cuando se realiza una consulta sobre la base de datos, se recibe un objeto *Cursor* con el resultado de la misma, y es mediante él cómo se tiene acceso a cada dato obtenido; su utilización simplifica muchas tareas comunes, por ejemplo para crear una lista a partir de una consulta a la base de datos, podríamos usar la clase *CursorAdapter* de modo que la propia *ListView* gestione

los campos a mostrar sin tener que preocuparnos de avanzar por los resultados o cerrar el cursor de la búsqueda.
El objeto de clase *Cursor* consume recursos del sistema, y por lo tanto tiene que ser gestionado bien manualmente cerrándolo cuando no se vaya a utilizar, o bien aprovechando el ciclo de vida de la *Activity* que lo use.
Cada base de datos puede tener una o varias tablas y se reconoce por un nombre que debe ser único dentro de la aplicación, pero puede ser igual al nombre de otra base de datos.

Lista de tareas

Como no puede ser de otro modo, afianzaremos la teoría explicada anteriormente mediante una pequeña aplicación de tareas que podremos utilizar para realizar la lista de la compra, a modo de agenda.... Se trata de una lista donde aparecerán todas las tareas que se tienen que realizar, indicando el nombre de la tarea, el lugar donde se debe realizar y la importancia de su realización, que se mostrará mediante un icono. Cuando se pulse sobre uno de los elementos de la lista se podrá ver su detalle, mientras que si se produce una pulsación larga, se mostrará un menú de contexto donde se podrá seleccionar si se desea eliminar la entrada o editarla. Durante la creación de la aplicación, aparecerán nuevos temas, como por ejemplo los menús contextuales o la creación de un adaptador propio para mostrar en la lista un icono u otro en función de la importancia de la entrada.
Comience creando un nuevo proyecto:

- Application name: Lista de Tareas
- Module Name: Lista de tareas
- Package name: com.acme.todolist
- Create Activity: ItemList

En la pantalla principal se mostrará una lista que contendrá las tareas que se deben realizar. En caso de no existir elementos, mostraremos un mensaje indicándolo. Para las clases que extienden *ActivityList* existen dos elementos del *layout* que nos facilitan esta labor, el primero de ellos es la lista en sí, que viene representada por el elemento `<ListView>` y que se identifica mediante el atributo `android:id="@android:id/list"` y el otro elemento es un `<TextView>` a mostrar cuando no existan elementos a mostrar en la lista, que se identifica por el atributo `android:id="@android:id/empty"`. Dicho de otro modo, si existen valores para la lista se mostrará el elemento

<ListView> con identificador android:id/list, de lo contrario se mostrará el elemento <TextView> con identificador android:id/empty. Esto no significa que todas las <ListView> tengan que tener el identificador android:id/list, sino que en caso de tenerlo, se facilita el trabajo, a parte, que sólo puede haber una lista con este identificador por vista. Modifique el *layout* activity_item_list.xml para que contenga estos elementos:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:orientation="vertical"
   android:layout_width="fill_parent"
   android:layout_height="fill_parent">
      <TextView android:layout_width="match_parent" android:text="@string/
      pending"  android:layout_height="wrap_content" android:gravity="center"
      android:textColor="#FFAA00CC" android:textSize="22dp"
      android:typeface="serif"/>
      <TextView android:id="@android:id/empty" android:layout_width="wrap_
      content" android:text="@string/lstNoElements"  android:layout_
      height="wrap_content" />

      <ListView android:id="@android:id/list"
      android:layout_width="fill_parent" android:layout_height="fill_parent"
      ></ListView>
</LinearLayout>
```

Vamos a dejar apartada por un momento la clase principal, para centrarnos en la clase encargada del control y acceso a la base de datos. Vamos a crear una clase plana que contenga otra clase que a su vez debe extender la clase *SQLiteOpenHelper*, de modo que tengamos perfectamente encapsulado todo el acceso a la base de datos. Cree una nueva clase llamada *DataBaseHelper* dentro del paquete com.acme.todolist. En ella crearemos una serie de constantes para establecer el nombre de todos los campos de la tabla utilizada para guardar las tareas, su propia sentencia de creación, el nombre de la base de datos, su versión y ciertas variables atributo que mantienen las referencias a objetos como la propia base *SQLiteDatabase*.

Dentro de la clase *DataBaseHelper* cree una clase privada llamada *DataBaseHelperInternal* que será quien extenderá la clase *SQLiteOpenHelper* y por lo tanto quien se encargará de crear las tablas y detectar cambios de versiones. Nosotros para el cambio de versión, simplemente eliminaremos la tabla existente y la crearemos de nuevo, perdiendo todos los datos existentes en cada cambio de versión, pero es posible que en otras aplicaciones le interese salvaguardar los datos antes de modificar la estructura de la base de datos. La tabla contendrá un identificador único que será un entero, el nombre de la tarea a realizar, el lugar donde se debe realizar, un texto libre y un nivel de importancia o de urgencia de su realización. Para este caso sólo usaremos una tabla en la base de datos, pero es posible utilizar tantas tablas como sean necesarias.

```
public class DataBaseHelper {
private Context mCtx = null;
private DataBaseHelperInternal mDbHelper = null;
private SQLiteDatabase mDb = null;
private static final String DATABASE_NAME = "TODOLIST";
private static final int DATABASE_VERSION = 3;
// tabla y campos
private static final String DATABASE_TABLE_TODOLIST = "todolist";
public static final String SL_ID = "_id";
public static final String SL_ITEM = "task";
public static final String SL_PLACE = "place";
public static final String SL_IMPORTANCE = "importance";
public static final String SL_DESCRIPTION = "description";

// SQL de creación de la tabla
private static final String DATABASE_CREATE_TODOLIST =
"create table "+ DATABASE_TABLE_TODOLIST +" ("+SL_ID+" integer primary key,
"+SL_ITEM+" text not null, "+SL_PLACE+" text not null, "
    +SL_IMPORTANCE+" integer not null, "+SL_DESCRIPTION+" text)";
//constructor
public DataBaseHelper(Context ctx) {
   this.mCtx = ctx;
}
//clase privada para control de la SQLite
private static class DataBaseHelperInternal extends SQLiteOpenHelper {
    public DataBaseHelperInternal(Context context) {
      super(context, DATABASE_NAME, null, DATABASE_VERSION);       }

@Override
   public void onCreate(SQLiteDatabase db) {
      createTables(db);
   }
   @Override
   public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion)
{
      deleteTables(db);
      createTables(db);
}
   private void createTables(SQLiteDatabase db) {
      db.execSQL(DATABASE_CREATE_TODOLIST);
   }

   private void deleteTables(SQLiteDatabase db) {
      db.execSQL("DROP TABLE IF EXISTS " + DATABASE_TABLE_TODOLIST);
   }
}
}
```

Tanto para la creación como para el borrado de las tablas se ha utilizado sentencias SQL estándar. En la sentencia de creación, al utilizar `integer primary key` como atributo, se le está indicando que el campo `id` de la tabla será la clave principal e implícitamente, que es un campo autoincrementado, es decir que no nos tendremos que preocupar de darle valor a la hora de guardar los datos, que él automáticamente se irá asignando valores incrementales y únicos (es como el atributo *autoincrement* de algunas bases de datos).

Para el inicio y fin de uso de la base de datos, proveeremos a la clase *DataBaseHelper* de un método de apertura y otro de cierre de la base. Durante la apertura es donde se obtiene la referencia al objeto *SQLiteDatabase* que será quien reciba finalmente todas las peticiones SQL.

```
public DataBaseHelper open() throws SQLException {
   mDbHelper = new DataBaseHelperInternal(mCtx);
   mDb = mDbHelper.getWritableDatabase();
   return this;
}

public void close() {
mDbHelper.close();
}
```

En relación a la base de datos necesitaríamos también los métodos de lógica de negocio para manejar la información, es decir, un método que nos devuelva todos los elementos que se tienen que realizar (las tareas), un método para obtener un elemento en particular, un método para eliminar entradas de la base de datos, otro para crear nuevas y un último para modificar las ya existentes. Para obtener los datos de la lista principal del programa, usaremos el método `query()` de la clase *SQLiteDatabase*, que mediante sus múltiples parámetros permite indicar los campos a seleccionar, las condiciones que deben cumplir estos campos, orden en el que se tienen que dar… en nuestro caso solamente especificaremos que se quiere consultar sobre la tabla `DATABASE_TABLE_TODOLIST`, que se quieren obtener ciertos campos informados a través de un *array* con sus nombres y que se quiere obtener el resultado ordenando por importancia, que viene dada por el campo `SL_IMPORTANCE`.

```
//obtener todos los elementos
public Cursor getItems() {
   return mDb.query(DATABASE_TABLE_TODOLIST, new String[] {SL_ID, SL_ITEM,
   SL_PLACE,SL_IMPORTANCE}, null, null, null, null,SL_IMPORTANCE);
}
```

Para la creación de una entrada en la tabla, usaremos el método `insert()`, donde le informaremos la tabla donde insertar los valores y los propios valores a insertar valiéndonos para ello de la clase *ContentValues*, donde informaremos mediante parejas clave-valor, los datos de cada uno de los campos.

```
//crear elemento
public long insertItem(String item, String place, String description, int
importance){
ContentValues initialValues = new ContentValues();
   initialValues.put(SL_IMPORTANCE, importance);
   initialValues.put(SL_ITEM, item);
   initialValues.put(SL_PLACE, place);
   initialValues.put(SL_DESCRIPTION, description);
   return mDb.insert(DATABASE_TABLE_TODOLIST, null, initialValues);
}
```

Para la creación de los elementos en la base de datos (elementos de la lista de tareas) utilizaremos una actividad aparte, que más adelante configuraremos también para la modificación de contenido. Para acceder a esta pantalla crearemos un menú de opciones con una única opción, la de crear una nueva tarea, a realizar a la que le daremos un icono para que resulte más vistoso. En el directorio `/src/res/main/menu` encontraremos un archivo de menú llamado `item_list.xml`; si no lo tiene, genere uno nuevo pulsando con el botón derecho en la carpeta y seleccionando New>Menú Resource File y le damos el contenido:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
   <item android:id="@+id/new_item"          android:icon="@drawable/ic_add"
   android:title="@string/newItem" />
</menu>
```

Para añadir el icono a mostrar en el menú, se puede seleccionar desde el navegador de archivos del sistema y pegarlo en la carpeta *drawable* correspondiente, pero esto tiene una pega, sólo estaremos haciendo accesible el icono en un tamaño. Android Studio ofrece la posibilidad de generar iconos para todos los tamaños de pantalla a partir de una imagen dada. Para acceder a esta funcionalidad, se debe pulsar con el botón derecho sobre la carpeta `/src/main/res` y pulsar sobre New>Image Asset, y se nos mostrará una pantalla similar a la que usamos en la selección del icono de la aplicación; en ella podremos seleccionar el nombre con el que lo queremos guardar, tamaños...

El menú se mostrará en la *Activity* principal, por lo que en la clase *ItemList* se deben sobrescribir los métodos correspondientes al control de los menús y que ya se vieron anteriormente:

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

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        // Handle item selection
        switch (item.getItemId()) {
        case R.id.new_item:
         Intent intent = new Intent (this,Item.class);
         startActivityForResult(intent, NEW_ITEM);
            return true;
        default:
            return super.onOptionsItemSelected(item);
        }
    }
```

Cuando se selecciona el elemento del menú, se iniciará una nueva actividad mostrando una pantalla donde rellenar los datos necesarios para crear un nuevo elemento, la llamada se realiza mediante `startActivityForResult()`, por lo que se necesita un código de acción a realizar. Añada este código de acción junto con el del resto de acciones que necesitaremos más adelante (crear, editar y mostrar entradas). Además aprovecharemos para añadir un par más de variables miembro que servirán para tener control del último elemento pulsado en la lista (cuando la mostremos) y la referencia a la clase que accederá a base de datos. Cree este código en la clase *ItemList* a nivel de atributo de la clase.

```
//acciones
public static final int NEW_ITEM = 1;
public static final int EDIT_ITEM = 2;
public static final int SHOW_ITEM = 3;

//elemento seleccionado
private int lastRowSelected = 0;
public static DataBaseHelper mDbHelper = null;
```

Se debe crear ahora una nueva actividad que se encargue de mostrar la pantalla para la introducción de datos de nuevos elementos, la que será llamada desde el menú recién creado. Para ello genere una nueva clase llamada *Item* y que extienda la clase *Activity*. En ella simplemente se recogerán los datos del elemento a introducir en la base de datos y se llamará al método correspondiente de la clase *DataBaseHelper* que hemos creado y que se encarga de ello:

```
public class Item extends Activity {
//referencias a elementos de pantalla
TextView item = null;
TextView place = null;
TextView description  = null;
TextView importance  = null;
//identificador de entrada
Integer  rowId = null;
@Override
public void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
      setContentView(R.layout.new_item);
   //botón de salvar
      Button saveBtn = (Button) findViewById(R.id.add);
         saveBtn.setOnClickListener(new View.OnClickListener() {
      public void onClick(View view) {
      setResult(RESULT_OK);
      saveData();
      finish();
      }
   });
// obtener referencias
   item = (TextView) findViewById(R.id.item);
```

```
      place = (TextView) findViewById(R.id.place);
      description  = (TextView) findViewById(R.id.description);
      importance  = (TextView) findViewById(R.id.importance);
}
protected void saveData() {
//obtener datos
   String itemText = item.getText().toString();
   String placeText = place.getText().toString();
   String descriptionText = description.getText().toString();
   String importanceText = importance.getText().toString();
   //insertar
   try{
      ItemList.mDbHelper.open();
      ItemList.mDbHelper.insertItem(itemText, placeText, descriptionText,
      Integer.parseInt(importanceText));
      ItemList.mDbHelper.close();
   } catch (SQLException e) {
      e.printStackTrace();
      showMessage(R.string.dataError);
   }
    }
}
```

El código para salvar los datos lo hemos asociado a un botón porque más adelante nos ayudará a "deshabilitarlo" cuando esta misma *Activity* se utilice para mostrar el elemento en modo lectura. El archivo de *layout* llamado `new_item.xml` que mostrará la *Activity* recién creada tendrá distintas cajas de texto para cada una de los datos de la tarea; simplemente se tendrá cuidado que la importancia de la tarea sea un entero:

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

<Button
android:id="@+id/add"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_alignParentBottom="true"
android:layout_centerHorizontal="true"
android:text="@string/save" />

<LinearLayout
android:layout_width="fill_parent"
android:layout_height="fill_parent"
android:layout_alignParentLeft="true"
android:layout_alignParentTop="true"
android:layout_above="@+id/add"
android:orientation="vertical">
   <TableRow
   android:id="@+id/idRow"
   android:layout_width="match_parent"
   android:layout_height="wrap_content"
   android:visibility="gone">
```

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

    <TextView
    android:id="@+id/identificator"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"/>
</TableRow>

<TableRow
android:layout_width="match_parent"
android:layout_height="wrap_content">

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

    <EditText
    android:id="@+id/item"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"/>
</TableRow>

<TableRow
android:layout_width="match_parent"
android:layout_height="wrap_content">

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

    <EditText
    android:id="@+id/place"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"/>
</TableRow>

<TableRow
android:layout_width="match_parent"
android:layout_height="wrap_content">

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

    <EditText
    android:id="@+id/importance"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:inputType="number"/>
</TableRow>

<TableRow
android:layout_width="match_parent"
android:layout_height="match_parent">
```

```
        <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:gravity="top"
        android:text="@string/description"/>

        <EditText
        android:id="@+id/description"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:gravity="top" />
    </TableRow>
</LinearLayout>
</RelativeLayout>
```

Para tener algo utilizable ya sólo nos queda modificar la *Activity* principal de modo que sea capaz de recuperar la información de la base de datos y mostrarla en una lista.

ArrayAdapter

Comentamos anteriormente que el hecho de trabajar con cursores facilitaba mucho tareas como por ejemplo a la hora de mostrar información en listas, también se habló que en ocasiones los *layouts* proporcionados por Android para la confección de las entradas de las listas no satisfacen del todo las necesidades de la aplicación; en este caso vamos a mostrar cómo crear un adaptador de modo que pueda usar lista de datos para obtener la información y formatearla según un *layout* dado para cada entrada de la lista.

Volvamos a la clase principal, donde se mostrarán los datos. Cuando se quiere asignar los datos a mostrar en el elemento *ListView*, se hace utilizado el método `setListAdapter()` de la clase *ListActivity* o bien obteniendo la referencia al propio objeto *ListView* y ejecutando sobre él el método .En este caso no utilizaremos un adaptador genérico, sino que lo crearemos nosotros para tener mayor control sobre su presentación.

Antes de crear el adaptador, es necesario abrir la base de datos a la que queremos acceder. En el método `onCreate()` de la clase *ItemList* realizamos la conexión a la base de datos y obtenemos la información de ésta:

```
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_item_list);
    // abrir la base de datos
    mDbHelper = new DataBaseHelper(this);
    try {
```

```
                fillData();
    } catch (SQLException e) {
       e.printStackTrace();
       showMessage(R.string.dataError);
          }
}
```

Creamos un método para mostrar errores, del mismo modo que lo hemos usado en ejercicios anteriores; en la clase ItemList.

```
private void showMessage(int message){
        Context context = getApplicationContext();
        CharSequence text = getResources().getString(message);
        int duration = Toast.LENGTH_SHORT;
        Toast toast = Toast.makeText(context, text, duration);
        toast.show();
    }
```

El método `fillData()` se encarga de obtener los elementos a publicar y crear el adaptador que utilizará la lista para mostrar los elementos (que aún no hemos creado). Lo que se debe hacer es obtener una conexión a la base de datos y realizar la llamada. Cuando tengamos la información de la base de datos, la obtendremos en un *Cursor*, que deberemos ir recorriendo mediante `moveToNext()` para extraer cada línea de datos. Mediante llamadas a `getInt()`, `getString()`, `getFloat()`...podemos ir consultando el valor que tiene cada columna en la fila de resultados en la que nos encontremos. El parámetro que espera este tipo de funciones de consulta, es el número de columna que se quiere consultar, que puede informarse directamente o, si tenemos el nombre de la columna que queremos consultar, apoyarnos en el método `getColumnIndex()` que tiene como parámetro precisamente el nombre de una columna y retorna la posición que ocupa en el cursor.

```
private void fillData() {
   // se abre la base de datos y se obtienen los elementos
   mDbHelper.open();
   Cursor itemCursor = mDbHelper.getItems();
   ListEntry item = null;
   ArrayList<ListEntry> resultList = new ArrayList<ListEntry>();
   // se procesa el resultado
   while (itemCursor.moveToNext()) {
      int id = itemCursor.getInt(itemCursor.getColumnIndex(DataBaseHelper.
      SL_ID));
      String task = itemCursor.getString(itemCursor.
      getColumnIndex(DataBaseHelper.SL_ITEM));
      String place = itemCursor.getString(itemCursor.
      getColumnIndex(DataBaseHelper.SL_PLACE));
      int importance = itemCursor.getInt(itemCursor.
      getColumnIndex(DataBaseHelper.SL_IMPORTANCE));
      item = new ListEntry();
```

```
        item.id = id;
        item.task = task;
        item.place = place;
        item.importance= importance;
        resultList.add(item);
    }
    //cerramos la base de datos
    itemCursor.close();
    mDbHelper.close();
    //se genera el adaptador
    TaskAdapter items = new TaskAdapter(this, R.layout.row_list, resultList,
    getLayoutInflater());
    //asignar adaptador a la lista
    setListAdapter(items);
}
```

La clase *TaskAdapter* utilizada como adaptador para la lista, debe ser capaz de manejar listas de elementos (porque así lo requiere nuestra aplicación), así que aprovecharemos una clase disponible en Android que nos ayudará a ello. Cree una clase privada dentro de *ItemList* de modo que extienda la clase *ArrayAdapter*.

Dentro del constructor de la clase guardarán los elementos importantes como la lista de objetos a mostrar en unas variables de la clase de tipo atributo. Como se quiere tener un completo control a la hora de mostrar los elementos, vamos a sobrescribir el método *getView()* de la clase *ArrayAdapter*; en él se puede acceder a todos los elementos definidos en el *layout* usado para la línea de la lista, mediante `findViewById()` (como hemos hecho en otras ocasiones) y establecer su valor unitariamente, así es posible informar de una manera u otra cierto elemento dependiendo del valor que tenga otro, como en este ejemplo haremos con los iconos mostrados en cada fila de la lista.

```
private class TaskAdapter extends ArrayAdapter<ListEntry> {
    private LayoutInflater mInflater;
    private List<ListEntry> mObjects;

    private TaskAdapter(Context context, int resource, List<ListEntry>
    objects, LayoutInflater mInflater) {
        super(context, resource, objects);
        this.mInflater = mInflater;
        this.mObjects = objects;
    }

    @Override
    public View getView(int position, View convertView, ViewGroup parent) {
        ListEntry listEntry = mObjects.get(position);
        // obtención de la vista de la línea de la tabla
        View row = mInflater.inflate(R.layout.row_list, null);
        //rellenamos datos
        TextView place = (TextView) row.findViewById(R.id.row_place);
        TextView item = (TextView) row.findViewById(R.id.row_item);
        place.setText(listEntry.place);
```

```
        item.setText(listEntry.task);

        // dependiendo de la importancia, se muestran distintos iconos
        ImageView icon = (ImageView) row.findViewById(R.id.row_importance);
        icon.setTag(new Integer(listEntry.id));
        switch (listEntry.importance) {
        case 1:
           icon.setImageResource(R.drawable.ic_green);
           break;
        case 2:
           icon.setImageResource(R.drawable.ic_yellow);
           break;
        default:
           icon.setImageResource(R.drawable.ic_red);
           break;
        }
        return row;
    }
}
```

Cada vez que se va a pintar una nueva línea de la lista, se ejecuta la función `getView()` y como parámetro se tiene la posición en la lista que ocupa esa línea, por lo que se tiene control total por si se quiere incluso pintar cada línea de una manera distinta, para conseguirlo sólo hay que inflar un *layout* distinto dependiendo de la posición, pero no es nuestro caso; sólo usaremos un *layout* para mostrar las filas de la lista.

El archivo de *layout* `row_list` contiene un icono para mostrar la importancia o urgencia del elemento a comprar, la descripción del elemento y el lugar donde se tiene que comprar:

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

    <ImageView
        android:id="@+id/row_importance"
        android:layout_width="50dip"
        android:layout_height="50dip"
        android:layout_alignParentRight="true" >
    </ImageView>

    <TextView
        android:id="@+id/row_place"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_alignParentLeft="true"
        android:gravity="center_vertical" />

    <TextView
        android:id="@+id/row_item"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_toLeftOf="@+id/row_importance"
```

```
        android:layout_below="@+id/row_place"
        android:layout_alignParentBottom="true"
        android:ellipsize="marquee"
        android:fadingEdge="horizontal"
        android:gravity="right" />
</RelativeLayout>
```

Hemos utilizado una clase llamada *ListEntry* para pasar los datos entre la actividad principal y el adaptador de la lista. Esta clase la definiremos como una clase privada dentro de la propia *ItemList* y será realmente sencilla, sólo necesitamos mantener las cuatro propiedades que queremos mostrar en la lista (no se han generado los *setters* y *getters* de las propiedades, pero si el lector se siente más cómodo se puede trabajar con ellos).

```
private class ListEntry {
        int id;
        String task;
        String place;
        int importance;
    }
```

Figura 12.1. Lista vacía.

Para poder probar la aplicación, añada la línea correspondiente a la actividad *Item* en el `AndroidManifest.xml` y modifique el archivo `strings.xml` con todas las constantes de texto que hemos utilizado y que utilizaremos a lo largo del ejemplo:

```
<activity android:name=".Item"/>
```

y ajuste los textos:

```
<resources>
   <string name="app_name">Lista de Tareas</string>
   <string name="lstNoElements">No hay nada pendiente</string>
   <string name="pending">Tarea pendiente...</string>
   <string name="newItem">Nuevo elemento</string>
   <string name="deleteItem">Eliminar elemento</string>
   <string name="editItem">Editar elemento</string>
   <string name="alrtDelete">Eliminar elemento</string>
   <string name="alrtDeleteEntry">Se va a eliminar un elemento. ¿Está
seguro?</string>
   <string name="element">Tarea</string>
   <string name="place">Lugar</string>
   <string name="importance">Importancia</string>
   <string name="description">Descripción</string>
   <string name="save">Salvar</string>
   <string name="identificator">Identificador</string>
   <string name="dataError">Error al obtener los datos</string>
   ...
```

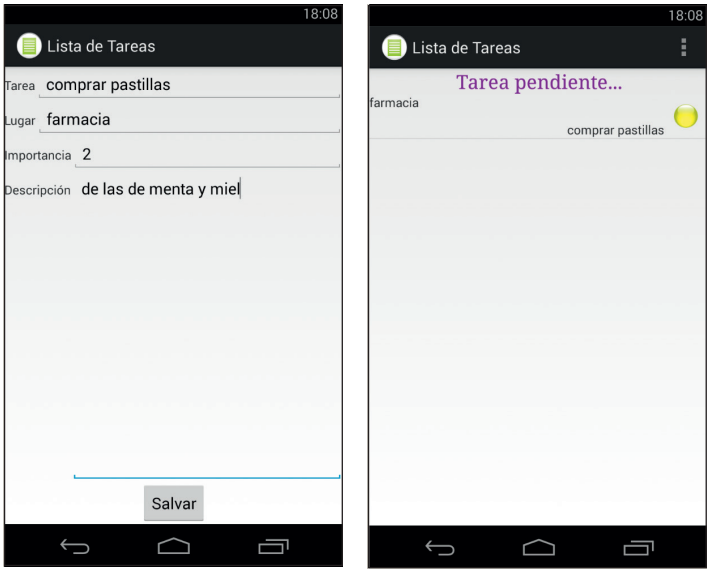

Figura 12.2. Pantalla de creación de la tarea y lista con ella.

Menú contextual

Actualmente nuestra aplicación ya puede crear elementos y visualizarlos en una lista, pero no es posible ver sus detalles, eliminarlos o modificarlos. Para implementar la opción de modificar y eliminar los elementos utilizaremos lo que se denomina menú contextual, que no es otra cosa que un menú que se

muestra al usuario cuando mantiene una pulsación larga sobre algún elemento de la pantalla. En el ejemplo que estamos trabajando, lo haremos sobre cada elemento de la lista, de modo que el menú muestre la opción de editar o eliminar dicho elemento.

Si comparamos el menú contextual con el menú de opciones que es el que hemos estado usando hasta ahora, podemos decir que la creación del menú contextual se realiza mediante el método `onCreateContextMenu()` en lugar del `onCreateOptionsMenu()` y que tras su pulsación se ejecuta el método `onContextItemSelected()` en lugar de `onOptionsItemSelected()` pero que su definición se realiza mediante XML, del mismo modo que el menú de opciones.

Cree un archivo XML con la definición del menú incluyendo las opciones de edición y borrado de elementos y llámelo `context.xml`:

```
<menu  xmlns:android="http://schemas.android.com/apk/res/android">
<item android:id="@+id/edit_item" android:title="@string/editItem"/>   <item
android:id="@+id/delete_item" android:title="@string/deleteItem" />
</menu>
```

Para el control de este nuevo menú, implemente en la clase principal *ItemList* los métodos necesarios y nombrados anteriormente `onCreateContextMenu()` para crearlo:

```
@Override
public void onCreateContextMenu(ContextMenu menu, View v,
   ContextMenu.ContextMenuInfo menuInfo) {
      super.onCreateContextMenu(menu, v, menuInfo);
   MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.context, menu);
}
```

Y `onContextItemSelected()` para saber qué elemento del menú ha sido el seleccionado. Lo primero que haremos en este método, será guardar el identificador de la entrada de la lista seleccionada para usarla más adelante. En este método se tiene como parámetro un objeto *MenuItem* con toda la información sobre el elemento del menú que ha sido pulsado. El identificador que buscamos para poder editar o borrar el elemento mostrado se puede obtener mediante el atributo `id` del objeto de clase *AdapterContextMenuInfo*; mediante `getMenuInfo()` se obtendrá información de la vista que está enlazada con el menú mostrado, en este caso la entrada de la lista.

En caso de que se pulse sobre borrar, se mostrará una pantalla preguntando al usuario si está seguro de lo que va a hacer; esto se ha implementado con una *AlertDialog*, al que le definiremos dos botones, uno para aceptar y otro para cancelar la acción, en caso de aceptar, se llamará al método `deleteEntry()`, donde borraremos la entrada.

En caso de que se pulse sobre editar, lo que se hará será introducir el identificador de la entrada en un *Intent* y lanzar una nueva *Activity* con la acción `EDIT_ITEM`. Aprovecharemos la actividad ya creada *Item*, que más adelante adecuaremos para que sea capaz de editar elementos además de crearlos.

```
@Override
public boolean onContextItemSelected(MenuItem item) {
   AdapterView.AdapterContextMenuInfo delW = (AdapterView.
   AdapterContextMenuInfo) item
                  .getMenuInfo();
//salvar identificador del elemento pulsado
      lastRowSelected = delW.position;
      // comprobar el elemento seleccionado
   switch (item.getItemId()) {
   case R.id.delete_item:
   //preguntar si está seguro de borrarlo
      new AlertDialog.Builder(this).setTitle(
         this.getString(R.string.alrtDelete)).setMessage(
         R.string.alrtDeleteEntry).setPositiveButton(
         android.R.string.ok, new AlertDialog.OnClickListener() {
         public void onClick(DialogInterface dlg, int i) {
            deleteEntry();
         }                          }).setNegativeButton(android.R.string.
                                      cancel, null).show();
   return true;
   case R.id.edit_item:
      //nueva actividad con el identificador como parámetro
      Intent i = new Intent(this, Item.class);
      i.putExtra(DataBaseHelper.SL_ID, ((ListEntry)getListAdapter().
      getItem(lastRowSelected)).id);
      startActivityForResult(i, EDIT_ITEM);
      return true;
   default:
      return super.onContextItemSelected(item);
   }
    }
```

Antes de seguir implementando los métodos que nos faltan, hay que indicar al sistema que la lista debe mostrar un menú contextual, esto se realiza mediante `registerForContextMenu()` al que se le pasa como parámetro la *View* que debe reaccionar y mostrar el menú contextual, en nuestro caso la propia lista. Para implementarlo, añádalo donde se configura la pantalla con la lista de la aplicación, al final del todo del método `onCreate()` de la clase *ItemList*.

```
registerForContextMenu(getListView());
```

El método `deleteEntry()` se apoyará en la clase *DataBaseHelper* para borrar la entrada.

```
protected void deleteEntry() {
   try{
      mDbHelper.open();
```

```
      mDbHelper.delete(((ListEntry)getListAdapter().
getItem(lastRowSelected)).id);
      mDbHelper.close();
      fillData();
   }catch (SQLException e){
      e.printStackTrace();
      showMessage(R.string.dataError);
   }
}
```

Actualmente no se tiene preparada la clase *DataBaseHelper* para poder realizar borrados; vuelva a la clase para acabar de implementar los métodos restantes que son el de borrado, el de obtención de datos para ver el detalle y el de modificación del elemento.

El borrado de los elementos se realiza mediante el método `delete()` de la clase *SQLiteDatabase*, quien acepta como parámetros la tabla donde se quiere eliminar los elementos, una condición (que puede ser compuesta o no) para seleccionar los elementos a eliminar y un *array* con los valores que se darán a cada una de las ocurrencias de "?" dentro de la condición dada.

```
public int delete(int lastRowSelected) {
return mDb.delete(DATABASE_TABLE_TODOLIST, SL_ID + "=?", new String[]{
Integer.toString(lastRowSelected)});
}
```

Cuando el usuario decide obtener más información sobre un elemento, debemos obtener todos los datos disponibles para ese elemento en concreto. Esto lo implementaremos también en la clase *DataBaseHelper* a través del método `getItem()`, donde se devuelve un cursor con los datos referentes al elemento seleccionado. Para la implementación, en lugar de utilizar el método `query()` ya conocido, aprovecharemos para conocer otro modo; se hará mediante la llamada al método `rawQuery()` donde acepta como parámetros una cadena con la sentencia SQL de la búsqueda y una serie de valores que servirán para sustituir por orden cada una de las ocurrencias del valor "?" dentro de la cadena SQL.

```
public Cursor getItem(int itemId){
   return mDb.rawQuery(" select "+ SL_ITEM+","+ SL_PLACE+","+ SL_
DESCRIPTION+","+SL_IMPORTANCE + "," + SL_ID + " from " + DATABASE_TABLE_
TODOLIST  + " where " + SL_ID + "= ?",new String[]{Integer.toString(itemId)});
}
```

Por último nos debemos encargar de la modificación de elementos ya existentes; el mecanismo es semejante a los métodos anteriores.

```
public int updateItem(int ident, String item, String place,
            String description, int importance) {
ContentValues cv = new ContentValues();
   cv.put(SL_ITEM, item);
   cv.put(SL_PLACE, place);
```

```
    cv.put(SL_DESCRIPTION, description);
    cv.put(SL_IMPORTANCE, importance);
    return mDb.update(DATABASE_TABLE_TODOLIST, cv, SL_ID + "=?", new String[]
    {Integer.toString(ident)});
}
```

Mediante la llamada al método `update()`, conseguimos actualizar todos los elementos que cumplan la condición dada; al trabajar con la condición de que tengan el identificador dado y este identificador sea la clave primaria en la tabla, sólo actualizaremos un elemento. Retorna el número de elementos que se han actualizado (en este caso siempre uno).

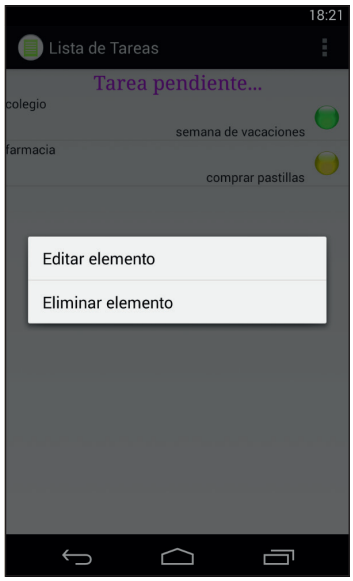

Figura 12.3. Menú contextual.

Ahora podría probar la aplicación y ver cómo puede ya borrar elementos simplemente manteniendo pulsado sobre ellos y seleccionando la entrada eliminar, pero si en lugar de eliminarlos los tratamos de editar, nos encontraremos que nos envía a la pantalla para añadir un elemento nuevo, y es que vamos a utilizar esta misma *Activity* para crear, editar y mostrar los elementos, por lo que hay que ampliar un poco su lógica para que dependiendo de la acción que vaya a realizar, se comporte de un modo o de otro.

Diríjase a la clase *Item* para añadir la lógica necesaria para el control de las distintas acciones que debe realizar. Controlaremos todo desde el método `onCreate()`; en él recuperaremos los extras con los que ha sido llamada la actividad (en caso de que existan) y se intenta obtener el indicador del ele-

mento a mostrar, este indicador se guardará en la variable `rowId` y servirá para controlar si se debe modificar una entrada (si `rowId` no es nulo, es decir si se ha informado desde *ItemList*) o si se debe crear. El hecho de que `rowId` sea nulo o no, también implicará que sea visible o no el número de identificador en pantalla, haciendo desaparecer o mostrando el elemento del *layout* `R.id.idRow`. Además como se quiere utilizar la pantalla para mostrar en modo lectura los detalles del elemento, controlaremos si el código de acción es `ItemList.SHOW_ITEM` y en este caso mostraremos un *layout* distinto al que mostraremos para los otros casos; este nuevo *layout* llamado `detail_item.xml` (que debe crear en este momento) es muy parecido al *layout* usado hasta ahora en esta actividad, solo que trabajando con `<TextView>` en lugar de `<EditText>`:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >
    <TableRow  android:id="@+id/idRow"
        android:layout_width="match_parent"
        android:layout_height="wrap_content">
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/identificator"/>
        <TextView android:id="@+id/identificator"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:gravity="right" />
    </TableRow>
    <TableRow
        android:layout_width="match_parent"
        android:layout_height="wrap_content" >
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/element" />
        <TextView android:id="@+id/item"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:gravity="right" />
    </TableRow>
    <TableRow
        android:layout_width="match_parent"
        android:layout_height="wrap_content" >
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/place" />
        <TextView android:id="@+id/place"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
```

```
            android:gravity="right" />
    </TableRow>
    <TableRow
        android:layout_width="match_parent"
        android:layout_height="wrap_content" >
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/importance" />
        <TextView  android:id="@+id/importance"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:gravity="right"/>
    </TableRow>
    <TableRow
        android:layout_width="match_parent"
        android:layout_height="wrap_content" >
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/description"/>
        <TextView android:id="@+id/description"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:gravity="right"/>
    </TableRow>
</LinearLayout>
```

Para controlar todo lo indicado anteriormente, el método `onCreate()` de la clase *Item* quedaría:

```
@Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
      //obtención de extras, identificador y acción
        Bundle extras = getIntent().getExtras();
        rowId = (savedInstanceState == null) ? null :
          (Integer) savedInstanceState.getSerializable(DataBaseHelper.SL_
          ID);
      if (rowId == null) {
          rowId = extras != null ? extras.getInt(DataBaseHelper.SL_ID):
          null;
      }
      // es solo para visualizar?
      if (extras != null && extras.getInt("action")== ItemList.SHOW_ITEM) {
         setContentView(R.layout.detail_item);
      }
      else{
         setContentView(R.layout.new_item);
         //botón de salvar
         Button saveBtn = (Button) findViewById(R.id.add);
            saveBtn.setOnClickListener(new View.OnClickListener() {
               public void onClick(View view) {
                    setResult(RESULT_OK);
                    saveData();
```

```
                finish();
            }
        });
    }
    // obtener referencias
      item = (TextView) findViewById(R.id.item);
    place = (TextView) findViewById(R.id.place);
    description  = (TextView) findViewById(R.id.description);
    importance  = (TextView) findViewById(R.id.importance);
    //identificador visible o no
    TableRow tr = (TableRow) findViewById(R.id.idRow);
    if (rowId!=null){
       tr.setVisibility(View.VISIBLE);
    populateFieldsFromDB();
  }
     else{
       tr.setVisibility(View.GONE);
     }
}
```

El método `populateFieldsFromDB()` se encargará de obtener los datos precisos del elemento a visualizar y mostrarlos en las etiquetas correspondientes. La llamada a la base de datos ya está coificada en la clase *DataBaseHelper*, por lo que el método quedaría:

```
private void populateFieldsFromDB() {
try {
   ItemList.mDbHelper.open();
   Cursor c = ItemList.mDbHelper.getItem(rowId.intValue());
   if (c.moveToFirst()) {
      //diferentes maneras de obtener los datos del cursor
      //Mediante nombre de columna y lanza excepción si no existe
      item.setText(c.getString(c.getColumnIndexOrThrow(DataBaseHelper.SL_
ITEM)));
      //Mediante nombre de columna y devuelve -1 si no existe
      place.setText(c.getString(c.getColumnIndex(DataBaseHelper.SL_PLACE)));
      //Mediante posición del campo en el cursor
      description.setText(c.getString(2));
      importance.setText(Integer.toString(c.getInt(3)));
      TextView id = (TextView) findViewById(R.id.identificator);
      id.setText(Integer.toString(c.getInt(4)));
   }
   c.close();
   ItemList.mDbHelper.close();
} catch (SQLException e) {
   e.printStackTrace();
   showMessage(R.string.dataError);
}
}
```

La obtención de los distintos campos del cursor se ha hecho de maneras distintas para ver las opciones disponibles para ello, personalmente la que más me gusta es la primera, pero es mucho más rápido el acceso directamente por número de campo en lugar de por nombre, por lo que si el método tiene un

tiempo de ejecución crítico, se debería acceder por número de columna.
El método `saveData()` de la clase *Item* que se estaba utilizando para insertar el elemento en la base de datos, también debe cambiarse para detectar si se está realizando una inserción o una modificación de un elemento ya existente, utilizando los métodos ya implementados en la clase *DataBaseHelper*:

```
protected void saveData() {
   //obtener datos
   String itemText = item.getText().toString();
   String placeText = place.getText().toString();
   String descriptionText = description.getText().toString();
   String importanceText = importance.getText().toString();
   try {
      ItemList.mDbHelper.open();
      if (rowId == null){
      //insertar
      ItemList.mDbHelper.insertItem(itemText, placeText, descriptionText,
      Integer.parseInt(importanceText));
      }
      else{
      //actualizar
      TextView tv = (TextView)findViewById(R.id.identificator);
      String ident = tv.getText().toString();
      ItemList.mDbHelper.updateItem(Integer.parseInt(ident),itemText,
      placeText, descriptionText, Integer.parseInt(importanceText));
      }
      ItemList.mDbHelper.close();
   } catch (SQLException e) {
      e.printStackTrace();
      showMessage(R.string.dataError);
   }
 }
```

Ya sólo nos queda indicar que cuando se pulse sobre un elemento de la base de datos con una pulsación corta, lo muestre en modo "sólo lectura" dentro de la actividad *Item*. La clase *ListView* tiene un método llamado `onListItemClick()` que se ejecuta cada vez que un elemento de la lista es pulsado; a este método le llega la lista sobre la que ha pulsado, la vista, la posición en la lista y el identificador del registro. Para conocer el registro de la base de datos, recuperamos el *tag* guardado durante la creación de la lista en el objeto *ImageView* llamado `R.id.row_importance` , indicando como código de actividad `SHOW_ITEM`.

```
@Override
protected void onListItemClick(ListView l, View v, int position, long id) {
        Intent i = new Intent(this, Item.class);
        int rowId = (Integer)v.findViewById(R.id.row_importance).getTag();
        i.putExtra(DataBaseHelper.SL_ID, rowId);
        i.putExtra("action", SHOW_ITEM);
        startActivityForResult(i, SHOW_ITEM);
}
```

Figura 12.4. Detalle de tarea.

Habrá notado el lector, que la lista no se refresca cuando se añade u nuevo elemento o se modifica alguno existente. Para paliar este defecto, lo que haremos será refrescar la lista cuando se venga de la tarea de editar o crear nuevas entradas. Las llamadas a la pantalla de edición de información se han hecho con `startActivityForResult()` lo que quiere decir que al volver a la actividad que las ha llamado, lo hará llamando al método onActivityResult(). Aprovecharemos ese método para refrescar la lista si se ha vuelto de las acciones editar o nuevo elemento (es decir `requestCode` es igual a `EDIT_ITEM` o a `NEW_ITEM`) y si el resultado es correcto (lo será siempre que salvemos algún dato, puesto que lo forzamos desde el método `save()` de la clase *Item* mediante `setResult(RESULT_OK)`). El código del método quedaría:

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent
data) {
   super.onActivityResult(requestCode, resultCode, data);
   if (requestCode == EDIT_ITEM || requestCode == NEW_ITEM){
      if (resultCode == Activity.RESULT_OK){
         try {
            fillData();
         } catch (SQLException e) {
            e.printStackTrace();
            showMessage(R.string.dataError);
         }
      }
   }
}
```

Mejorando la lista

En este ejemplo no es posible apreciarlo, pero cuando las listas tienen muchas entradas y sobre todo si son complejas (muchas imágenes por ejemplo), podemos encontrarnos que al realizar un desplazamiento de la lista, se vea que va a trompicones, esto se debe a que tarde mucho en generar las nuevas líneas a mostrar. Para evitar este comportamiento podemos ayudarnos del patrón *ViewHolder*. Se trata de un patrón estático para almacenar referencias a los objetos gráficos de la lista, de modo que al generar nuevas entradas en la lista pueda acceder a sus elementos visuales mucho más rápido. Veamos cómo usarlo en nuestra lista.
Comenzaremos generando la clase estática *ViewHolder* dentro de la clase *ItemList*; esta clase debe contener una entrada por cada elemento que se muestre en la línea de la lista, en nuestro caso dos *TextView* y un *ImageView*:

```
static class ViewHolder{
   TextView place;
   TextView task;
   ImageView importance;
}
```

En el método `getView()` del adaptador de la lista será donde utilicemos esta clase. El segundo parámetro de este método, es la vista anteriormente utilizada, que trataremos de volverla a usar previo chequeo de si realmente es nula o no. Si es nula, crearemos el ViewHolder para mantener los elementos de la lista, es decir sus referencias. Una vez sabemos que no es nula la vista procedemos a rellenar los datos. Para mantener el objeto ViewHolder unido a la vista real del la línea, usamos su propiedad `tag`, que puede almacenar cualquier tipo de dato (anteriormente ya la hemos usado en el ImageView para guardar el identificador de la entrada de la base de datos).
El nuevo método `getView()` quedaría:

```
@Override
public View getView(int position, View convertView, ViewGroup parent) {
   View row  = convertView;
   //si es nulo lo creamos
   if (row == null){
      // obtención de la vista de la línea de la tabla
      row = mInflater.inflate(R.layout.row_list, null);
      ViewHolder holder = new ViewHolder();
      holder.place = (TextView) row.findViewById(R.id.row_place);
      holder.task = (TextView) row.findViewById(R.id.row_item);
      holder.importance = (ImageView) row.findViewById(R.id.row_importance);
      row .setTag(holder);
   }

   ListEntry listEntry = mObjects.get(position);
   //rellenamos datos
```

```
    ViewHolder holder = (ViewHolder)row.getTag();
    holder.place.setText( listEntry.place);
    holder.task.setText(listEntry.task);
    // dependiendo de la importancia, se muestran distintos iconos
    holder.importance.setTag(new Integer(listEntry.id));

    switch (listEntry.importance) {
       case 1:
          holder.importance.setImageResource(R.drawable.ic_ico1);
          break;
       case 2:
          holder.importance.setImageResource(R.drawable.ic_ico2);
                      break;
       default:
          holder.importance.setImageResource(R.drawable.ic_ico3);
          break;
    }
    return row ;
}
```

Ahora la aplicación está totalmente acabada; quizá no es la mejor que haya visto en cuestión visual (ya veremos cómo mejorarla), pero es totalmente funcional. Puede pulsar sobre un elemento para obtener su detalle o mantener pulsado para editarlo o borrarlo. Como posibles mejoras podría añadir nuevos campos a la base de datos o a la lista y por supuesto darle algo de color para mejorar su aspecto ya que se han usado *layouts* muy simples.

13 Intents

En este capítulo aprenderá a:

- Conocer la composición de un *Intent*.
- Configurar los *Intent* para que sean atendidos por otros programas.
- Diferenciar entre llamadas explícitas e implícitas.
- Utilizar filtros en las actividades para atender a llamadas explícitas.
- Utilizar la cámara desde nuestras aplicaciones.

Una de las características más peculiares de Android es su programación basada en objetos de tipo *Intent*. Es mediante estos mensajes, la manera en la que se comunican las aplicaciones tanto internamente como entre ellas, ofreciendo una serie de posibilidades que hacen de este tipo de programación una solución muy interesante en cuanto a versatilidad y libertad de configuración de cada sistema por poder seleccionar qué *Activity* responde a qué *Intent*. En los ejemplos que se han ido construyendo a lo largo del libro se ha podido ver cómo hacer una aplicación con más de una pantalla y la manera de hacer visible esas nuevas pantallas mediante objetos *Intent*. Hasta ahí todo bien, pero Android va más allá y permite utilizar los componentes de otras aplicaciones o del sistema estándar, simplemente mediante la configuración de un objeto *Intent* y su propagación. Aquí descubriremos cómo.

Desgranando el Intent

Un *Intent* es la abstracción de la operación que se quiere realizar; es una clase que permite enlazar los componentes de la misma o diferente aplicación. La manera de enlazar estos componentes es mediante la información que el propio *Intent* lleva consigo.

Datos del Intent

El *Intent* es un paquete contenedor de información, que será procesada por el sistema Android para saber quién es el receptor (o quienes), y una vez el receptor está en posesión del *Intent*, lo usará para saber qué acción tiene que hacer sobre qué datos; por lo que se tiene informar al receptor los datos sobre los que actuar y la acción a realizar sobre ellos.
El *Intent* más simple, se puede considerar aquel que llama de forma explícita a una clase para que atienda la petición y es el *Intent* que se ha estado utilizando hasta ahora, donde se informaba directamente la clase que se quería como destinataria del proceso; pero son muchos los tipos de datos que un *Intent* puede incorporar y mucho más compleja su lógica para obtener el componente que se ejecutará.
Dentro de los distintos datos que se pueden encontrar en un *Intent* podemos distinguir:

- Nombre del componente: Es un dato opcional. Se trata del nombre del componente que se quiere que reciba el *Intent*. Se informa mediante la dupla: nombre del paquete en el que se encuentra declarado en el

`AndroidManifest.xml` correspondiente a la aplicación donde está definido el componente destino y su nombre de clase completa, lo que normalmente se llama *fully qualified class name* (que no tienen porque coincidir). Para el caso en el que el componente destino se encuentre en el mismo proyecto que el componente que inicia el *Intent* el código utilizado sería algo ya conocido:

```
Intent i = new Intent(this, MyClass.class);
```

Si el componente se encuentra en otra aplicación, el código será algo semejante a:

```
Intent i = new Intent();
i.setClassName( "com.acme", "com.acme.IntentReceiver");
```

Teniendo en cuenta que el paquete del componente y el registrado en el `AndroidManifest.xml` no tienen porqué coincidir necesariamente.

Si el nombre del componente se encuentra informado, Android enviará el *Intent* a una instancia de la clase referida por él. Además de mediante el constructor, el nombre del componente se puede informar mediante los métodos `setComponent()`, `setClass()` y `setClassName()` de la clase *Intent*.

- Acción: Es una cadena de texto informando la acción a realizar. Dentro del *Intent* puede establecerse mediante el constructor o mediante la llamada el método `setAction()`. Dentro de la clase *Intent* se encuentran muchas de estas acciones ya disponibles como constantes por ejemplo `ACTION_CALL` (para iniciar una llamada telefónica) o `ACTION_SET_WALLPAPER` (para configurar el fondo de pantalla) entre otras muchas.

  Dependiendo de la acción se deberán informar ciertos datos para que sean tratados por esta acción. Android recomienda usar nombres de acciones que determinen de forma unívoca la acción que van a realizar y en la medida de lo posible que den idea de los datos a utilizar.

  Las acciones también pueden encontrarse definidas en otras clases, donde estarán las específicas a esa clase o componente. Es posible crearse sus propias acciones, lo único que hay que tener en cuenta es que se aconseja que en la cadena que constituye la acción, se encuentre el nombre del paquete de la aplicación, por ejemplo

```
public static final String ACTION_DELETE = "com.acme.app.DELETE";
```

- Datos: Contiene la URI de los datos sobre los que el componente que reciba el *Intent* debe trabajar. Como puede imaginar, cada tipo de acción lleva asociado un dato sobre el que trabajar distinto; si lo que se quiere

editar enviar un SMS, no tiene sentido que el dato que le acompañe en la petición sea una dirección web. Por ejemplo para una acción del tipo `ACTION_CALL`, la URI debe ser la de un teléfono: tel:9832517133, pero en otras ocasiones, la acción no es tan exacta con el tipo de dato que se le debe indicar, y será precisamente el tipo de dato quien decida qué componente es el que lo atenderá, por ejemplo la acción `ACTION_EDIT` puede servirnos para editar un contacto o una foto, dependiendo del tipo de dato pasado se ejecutará un componente u otro; es donde entra en juego el tipo de dato. Los datos pueden ser establecidos mediante la llamada `setData()` de la clase *Intent*.

- Tipo de datos: Está muy ligado con el punto anterior, tanto que muchas veces dependiendo del tipo de datos, los datos serán procesados por un componente o por otro. Para establecer el tipo de dato existe el método `setType()`, pero dado que los datos suelen venir acompañados del tipo de dato, existe también la llamada `setDataAndType()` para informar ambas cosas a la vez. Es posible que a veces no nos haga falta indicar el tipo de dato porque viene implícito en la URI de los datos (por ejemplo al escribir tel:834293424 como URI, ya se sabe que es un teléfono), pero si el dato es un fichero, no hay manera de saber qué tipo de fichero es, y por lo tanto no se sabe tampoco el componente que debe procesarlo, entonces es cuando se debe informar el tipo MIME del fichero para que pueda ser procesado (un fichero de audio no se procesa igual que uno de texto). En un ejemplo de este capítulo se verá cómo es posible llamar a una actividad en concreto mediante el uso de una URI que lleva implícito el tipo de dato o mediante el uso del tipo de dato y el dato por separado.
- Categoría: Es una información adicional sobre el tipo de componente que debe atender al objeto *Intent*. Mediante repetidas llamadas al método `addCategory()` se pueden añadir tantas categorías como se necesiten dentro del objeto *Intent*. Como ejemplo de algunas categorías podríamos citar `CATEGORY_LAUNCHER` (la actividad puede ser la actividad inicial de una aplicación y se mostrará en el menú principal de Android, en el *Launcher*) o `CATEGORY_BROWSEABLE` (para indicar que la actividad destino puede ser invocada por el navegador para mostrar datos que estén referenciados por un enlace, como por ejemplo una película o un sonido).
- Extras: es una serie de datos guardados de la manera clave-valor que sirven para indicar información adicional al componente que se encargue de atender el *Intent*. Existen algunas acciones que deben venir acompañadas de sus respectivos extras para poder ser correctamente atendidas, por ejemplo en caso de querer enviar un SMS, el cuerpo del SMS debe ir

en un extra llamado "sms_body". Si en su programa necesita una nueva acción que haga uso de parámetros pasados desde la actividad que inicia la llamada, el modo de pasarle los parámetros será mediante este método; las claves utilizadas para ello serán las que usted quiera, siempre teniendo en cuenta que deberá utilizar las mismas para guardar y recuperar de nuevo el valor. Para informar los extras dentro de un *Intent* se puede utilizar el método `putExtra()` o bien informar un *Bundle* aparte y luego hacerlo accesible al *Intent* mediante el método `putExtras()`.

- Flags: O banderas, sirven para indicar el comportamiento del componente que atienda la petición, por ejemplo `FLAG_ACTIVITY_SINGLE_TOP` evitará que se lance una nueva actividad si la actividad que debe atender el *Intent* ya se encuentra en lo alto de la pila de ejecución. Se pueden establecer estas banderas mediante la llamada `setFlag()`.

Nota:

Aunque en el texto los nombres de las categorías, acciones… aparecen en su forma abreviada (por ejemplo `CATEGORY_BROWSEABLE`*) para facilitar la lectura, cuando se defina el* `AndroidManifest.xml` *deben aparecer con el nombre del paquete donde están definidos (por ejemplo* `android.intent.category.BROWSABLE`*).*

De todos los tipos de datos explicados anteriormente, los más importantes son los datos, la acción y la categoría, ya que con ellos será con los que debemos configurar el *Intent* con tal de conseguir llamar al componente deseado.

Propagación

Hasta ahora se ha visto el *Intent* como el mecanismo para iniciar una nueva pantalla dentro de nuestra aplicación, pero también sirve para poder iniciar otros componentes de Android, todo depende de la llamada que se haga una vez configurado:

- Activity: Para el uso en las *Activity* podemos utilizar dos tipos de llamadas `Context.startActivity()` o `Context.startActivityForResult()`. En ambos casos se le debe pasar un *Intent* como parámetro, que será quien determine la *Activity* a ejecutar. Si se ha utilizado el método `Context.startActivityForResult()` para la llamada de la nueva *Activity*, esta puede devolver otro *Intent* con el resultado de la ejecución de la actividad.

- Service: Se puede usar un *Intent* para iniciar un *Service* mediante la llamada `Context.StartService()` o para modificar su comportamiento. Igualmente se puede usar con la llamada `Context.bindService()` para establecer conexión entre el componente que está lanzando el *Intent* y el *Service*.
- Broadcast: Se pueden usar los *Intent* para enviar mensajes a todos los *BroadcastReceivers* que estén registrados mediante el uso de los siguientes métodos: `Context.sendBroadcast ()` (el envío se realiza a todos los receptores a la vez, no se puede cancelar y no se pueden pasar datos entre receptores), `Context.sendOrderedBroadcast()` (el envío se realiza a todos los receptores por orden de preferencia y se puede incluso indicar un último receptor que obtenga el resultado de cada una de las ejecuciones; se permite pasar datos entre receptores), `Context.sendStickyBroadcast()` (una vez finalizada la emisión, el *Intent* permanece accesible y se puede recuperar su información) o `Context.sendStickyOrderedBroadcast()` (una mezcla de los dos mecanismos anteriores).

Cada circuito de emisión está aislado del resto, esto asegura que un *Intent* lanzado mediante `sendBroadcast()`, será recibido solamente por un *BroadcastReceiver* y no por una *Activity*.

Resolución

Las llamadas a los componentes mediante *Intents* se pueden realizar de modo explícito o implícito, es decir de modo explícito indicando exactamente la clase del componente en particular que debe atender el *Intent* o bien de modo implícito, que sería indicando las características que debe tener el componente que atenderá la petición. El modo más rápido de realizar las llamadas es el explícito, ya que casi no se tiene que configurar el objeto *Intent* y además se asegura que se lanza el componente exacto especificado… pero claro, siempre y cuando se sepa el nombre exacto de la clase que se debe informar en el *Intent* y esto no es algo común fuera de componentes que hayamos realizado nosotros mismos o nos hayan proporcionado su documentación. Para el resto de los casos se utilizará el método implícito, con todo lo que ello lleva consigo. Es posible que al utilizar el método implícito más de un componente pueda atender la petición del *Intent*, por ejemplo si se quiere abrir un fichero de audio, puede darse que haya más de una aplicación en el sistema que pueda abrir ese fichero, es por esto que se tiene que tener en cuenta la manera en la que el sistema Android establece el componente a utilizar. Hay que decir que mientras que para el caso de las actividades y los servicios solamente se establece un receptor, para el caso de los *BoradcastReceivers* pueden establecerse

varios, tantos como cumplan los requisitos del *Intent*.
La manera en la que se discriminan los componentes que no atenderán al *Intent* se basan en la comparación de los contenidos del propio *Intent* con los filtros establecidos para cada componente. Mediante los filtros se puede establecer las capacidades de cada componente en el fichero `AndroidManifest.xml` y delimitar así los *Intent* que son capaces de atender. Por defecto un componente no puede recibir nada que no sea una petición explícita, o dicho de otro modo, para ser capaz de recibir peticiones implícitas debe tener definidas estas características, estos filtros. Por otro lado, el hecho de tener definidos filtros no quita que el componente pueda ser llamado explícitamente por otros componentes. Los filtros se pueden establecer respecto a los datos a procesar, la acción a realizar o la categoría del componente destino.

Advertencia:

Las llamadas implícitas hacen uso de los filtros pero las llamadas explícitas no. Cuando se produce una llamada explicita a un componente, éste siempre atenderá la petición.

Filtros

Los filtros para los *Intent* permiten al programador definir las capacidades de los componentes de modo que el sistema sepa qué peticiones puede atender y cuáles no. La definición de los filtros no se pueden realizar en el código Java, ya que el sistema debería ejecutar todas las aplicaciones para conocer sus capacidades; en lugar de esto, se definen en el fichero `AndroidManifest.xml` y son procesadas cuando el sistema instala la aplicación, de modo que mantiene un seguimiento de todas las capacidades de cada componente. Existe una excepción a la afirmación anterior y son los filtros que se definen para los *BroadcastReceivers*, que son definidos dinámicamente mediante código (concretamente mediante `Context.registerReceiver()`). Los filtros son instancias de la clase *IntentFilter* y en el `AndroidManifest.xml` son representados mediante la etiqueta `<intent-filter>`.
Cada petición de *Intent* implícita es comprobada a tres niveles y el componente que atienda dicho *Intent* debe cumplir todos ellos. Las comprobaciones a realizar son:

- Acción a ejecutar
- Datos sobre los que actuar
- Categoría

Un mismo componente puede tener definidos varios filtros, por lo que puede que uno de ellos no cuadre con lo especificado en el *Intent* pero otro de ellos sí, en este caso se considera que el componente cumple con lo requerido por el *Intent*.

Filtros por acción

Por ejemplo si tenemos una aplicación en la que se mantienen los clientes, podemos utilizar la misma pantalla para crear uno nuevo, para mostrarlo o para editarlo. Entonces se le podrían crear unos filtros de modo que la *Activity* pertinente se pudiera invocar de modo implícito mediante ellos desde cualquier otra aplicación, sin necesidad de que la otra aplicación deba saber el nombre de la clase y el paquete en el que está definida:

```
<intent-filter … >
    <action android:name="com.acme.app.EDIT_CUSTOMER" />
    <action android:name="com.acme.app.SHOW_CUSTOMER" />
    <action android:name="com.acme.app.NEW_CUSTOMER" />
</intent-filter>
```

Aunque el filtro sea uno, sus acciones son varias, esto quiere decir que cuando se lance una petición, para que esta *Activity* sea la seleccionada, la acción informada en el *Intent* debe ser una de las definidas en esta *Activity* y vale con que simplemente una de ellas concuerde.
No es obligatorio ni cumplimentar todos los filtros en el *Intent* ni tenerlos definidos en la *Activity;* por lo que la primera selección de los componente se realiza teniendo en cuenta que si el *Intent* no tiene definido ningún filtro de acción para (en nuestro ejemplo) la actividad, entonces la *Activity* es seleccionada como posible candidata (recuerde que hay que pasar los otros filtros) y si la *Activity* no tiene designado ningún filtro y sí lo tiene el *Intent*, entonces la *Activity* es descartada.

Filtros por categoría

El *Intent* puede llevar definidas ninguna, una o varias categorías. En caso de tener varias categorías, el filtro debe contener todas ellas; puede tener más, pero nunca menos.
Un *Intent* sin categorías definidas siempre pasará el filtro, siendo el componente tomado como válido. La afirmación anterior tiene un "pero" y es que cuando se emite una llamada implícita mediante `startActivity()`, se introduce automáticamente una categoría en el *Intent*, la `android.intent.category.DEFAULT`, por lo que todas las *Activity* que puedan ser llamadas de modo implícito, deben tener definido este filtro, así aseguramos que otras aplicaciones puedan usar nuestras actividades (o protegerlas para que no las usen).

Filtros por tipo de dato

Los filtros de este tipo permiten un control del dato a procesar por contenido y/o por URI. Es posible crearse sus propias URI mediante el esqueleto:

```
<tipo>: //<host>:<puerto>/<ruta>
```

Así por ejemplo podríamos definir para nuestra aplicación la URI:

```
cliente://com.acme.customers:50000/clientes/2010/impagados
```

Dónde *cliente* es el *tipo*, *com.acme.customers* es el *host*, *50000* corresponde al *puerto* y *clientes/2010/impagados* es la *ruta*, y todo ello corresponde a la URI que da acceso al filtro por tipo de dato. A la hora de crear la URI no es necesario especificar todos los componentes, es posible por ejemplo simplemente especificar el tipo y la ruta:

```
cliente://2300423
```

El tipo de dato se especifica mediante un código MIME. En el archivo `AndroidManifest.xml` este dato es representado mediante el atributo `type` dentro del elemento `<data>`. Los tipos MIME se definen por el esqueleto *tipo/subtipo*, así tenemos que el dato de tipo texto puede ser de varios subtipos, entre ellos *text/plain* (texto plano), *text/html* (texto html), *text/vcard* (texto de una tarjeta virtual) u otros, y se puede usar comodines en su definición, por ejemplo para definir todos los tipos de texto podemos usar *text/** o para video *video/**.
A la hora de comprobar los filtros respecto a la URI, se comparan solo las partes de la URI que estén definidas en el filtro. Se podría definir sólo el *tipo,* de modo que cualquier URI que sea de ese *tipo* será atendida por el filtro. Del mismo modo, el filtro permite definir la ruta utilizando ciertos comodines para indicar que tiene que coincidir partes de la ruta. En la comparación, para que el filtro sea válido, debe cumplir tanto las especificaciones de la URI como del tipo de dato en caso de que se hayan definido. Las reglas de cumplimiento son:

1. Si el objeto *Intent* no tiene especificado ni una URI ni un tipo de dato, el componente será válido si tampoco tienen ninguno de ellos definido en su filtro.
2. Si el objeto *Intent* tiene definido el tipo de dato pero no la URI, entonces el componente deberá tener definido (al menos) el mimo tipo de dato en su filtro y no tener definido ningún filtro sobre la URI.
3. Si el objeto *Intent* tiene definido la URI pero no el tipo de dato y éste no puede ser conocido por la URI, entonces el componente será seleccionado si tiene definido en su filtro la misma URI y no tiene definido ningún tipo de dato.
4. Por último si el objeto *Intent* tiene definido tanto el URI como el tipo de dato, entonces el filtro del componente será válido para el tipo de dato si alguno de los tipos definidos concuerdan con el del *Intent*. En cuanto

a la parte de los URI, será válido si alguno de los URI definidos en el filtro concuerda también con la del *Intent* o si el URI del *Intent* es del tipo "*content:*" o "*file:*" y el filtro no tiene definido ninguno de tipo URI.

Algunos ejemplos de filtro de datos serían:

```
<data android:scheme="http" android:type="video/*" />
<data android:mimeType="video/*" />
```

Aunque en los ejemplos aquí mostrados aparecen cada uno de los tipos de filtro por separado, se debe tener en cuenta que se pueden utilizar conjuntamente. Por ejemplo, en todas las aplicaciones que se han realizado hasta el momento, para indicar la actividad encargada de aparecer en el *Launcher* de Android se ha definido un filtro combinación de una acción y una categoría:

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

Ejemplos de llamadas implícitas

Para ver mejor cómo funcionan las llamadas implícitas a actividades externas a nuestra aplicación, vamos a realizar un pequeño ejemplo en el que se realizará una petición al navegador web y también se trabajará con parte de la funcionalidad típica de un teléfono móvil: llamadas y sms.
Cree un nuevo proyecto con la estructura:

- Application name: IntentTest
- Module Name: IntentTest
- Package name: com.acme.intenttest
- Activity Name: MainActivity

Crearemos una pantalla donde el usuario pueda introducir una dirección web y pulsando sobre un botón se abra el navegador con la dirección introducida por el usuario. Modifiquemos en primera instancia el *layout* generado para introducir una caja de texto y un botón:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:orientation="vertical" android:layout_width="fill_parent"
   android:layout_height="fill_parent">
   <TextView android:layout_width="fill_parent"
      android:layout_height="wrap_content" android:text="Dirección:" />
   <EditText android:layout_width="fill_parent"
      android:layout_height="wrap_content" android:id="@+id/url"></EditText>
   <Button android:text="Ir a Web" android:id="@+id/web"
      android:layout_width="fill_parent" android:layout_height="wrap_content" />
</LinearLayout>
```

Verá que no se han tenido en cuenta la internacionalización de los textos y directamente se han escrito los literales sobre el atributo `android:text` en lugar de hacerlo en el fichero `strings.xml`. Es sólo por comodidad, pero recuerde que para que sus aplicaciones tengan un aspecto profesional, deben soportar varios idiomas y para eso es necesario extraer todos los literales en el fichero `strings.xml`.

En el evento `onCreate()` de la clase *MainActivity* recuperamos el botón creado y le asignamos código.

```
Button web = (Button) findViewById(R.id.web);
  web.setOnClickListener(new View.OnClickListener() {
   @Override
   public void onClick(View v) {
      launchweb();
   }
  });
```

El método llamado por el botón cuando sea pulsado se define como un método de la clase *MainActivity*:

```
protected void launchweb() {
   EditText url = (EditText) findViewById(R.id.url);
      // intent con acción + URI
   Intent i = new Intent(Intent.ACTION_VIEW, Uri.parse(url.getText().
toString()));
   startActivity(i);
}
```

En este método se obtiene la dirección web introducida por el usuario y se instancia un objeto *Intent* al que se le informa que se quiere realizar la acción `ACTION_VIEW` (acción ver) y que los datos que se quieren ver van especificados en la dirección que ha introducido el usuario, que es una dirección web.

En este ejemplo no hay control alguno sobre lo que el usuario introduce en la caja de texto, por lo que si introduce una secuencia cualquiera de letras al azar, la aplicación daría un error de tipo *ActivityNotFoundException*, y es que se espera que el sistema sea capaz de encontrar una actividad que pueda gestionar el tipo de dirección indicado en su dato, y al no encontrarlo se lanza esta excepción que se debería controlar.

Del modo en que se ha hecho el código de la llamada, si supiéramos la URI de una imagen en concreto del dispositivo, se podría escribir en esta caja de texto dicha URI y en lugar del navegador se mostraría la actividad capaz de mostrar una imagen. Puede probar de modo sencillo introduciendo en la caja de texto "tel://123456" (sin las comillas) y pulsando sobre el botón… verá lo que sucede.

Antes de probar la aplicación, no olvide que para poder trabajar con Internet es necesario tener permisos a nivel del fichero `AndroidManifest.xml`, por lo que introducimos la etiqueta correspondiente:

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

Para ver un ejemplo más de la llamada a una actividad externa, utilizaremos el teléfono como objetivo. Se va a modificar la pantalla de modo que el usuario pueda informar un número de teléfono, al que más adelante podremos llamar o enviar un SMS con la dirección introducida anteriormente. Para introducir estos cambios, el fichero de *layout* queda:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:orientation="vertical" android:layout_width="fill_parent"
   android:layout_height="fill_parent">
   <TextView android:layout_width="fill_parent"
      android:layout_height="wrap_content" android:text="Dirección:" />
   <EditText android:layout_width="fill_parent"
      android:layout_height="wrap_content" android:id="@+id/url"></EditText>
   <Button android:text="Ir a Web" android:id="@+id/web"
      android:layout_width="fill_parent" android:layout_height="wrap_content" />

   <TextView android:layout_width="fill_parent"
      android:layout_height="wrap_content" android:text="Teléfono:" />
   <EditText android:layout_width="fill_parent"
      android:layout_height="wrap_content" android:id="@+id/tlfnum"></EditText>
   <Button android:text="Enviar por SMS" android:id="@+id/sms"
      android:layout_width="fill_parent" android:layout_height="wrap_content" />
   <Button android:text="Llamar por teléfono" android:id="@+id/tlf"
      android:layout_width="fill_parent" android:layout_height="wrap_content" />
</LinearLayout>
```

Figura 13.1. Pantalla de prueba de Intent implícito.

En el código, efectuamos los cambios necesarios para asignar código a los dos nuevos botones añadidos a la interfaz, modifique el método `onCreate()` añadiendo las líneas:

```
Button tlf = (Button) findViewById(R.id.tlf);
tlf.setOnClickListener(new View.OnClickListener() {
@Override
   public void onClick(View v) {
      launchTlf();
   }
   });

Button sms = (Button) findViewById(R.id.sms);
sms.setOnClickListener(new View.OnClickListener() {
   @Override
   public void onClick(View v) {
      launchSms();
   }
   });
```

Los métodos asignados a cada uno de los botones serán los encargados de realizar las operaciones buscadas.

En el caso del método `launchTlf()` se instanciará un *Intent* para realizar una llamada, que corresponde a la acción `ACTION_CALL`, y como dato sobre el que realizar la llamada se usará el teléfono introducido por el usuario, que se encuentra en la caja de texto incluida para ello. El dato se codificará en una URI que es capaz de interpretar el filtro de la actividad encargada del teléfono, esta URI es del tipo:

```
tel:número_teléfono
```

El método sería:

```
protected void launchTlf() {
   EditText tlfNum = (EditText) findViewById(R.id.tlfnum);
   Intent i = new Intent(Intent.ACTION_CALL, Uri.parse("tel:" + tlfNum.
getText().toString()));
   startActivity(i);
}
```

Para el caso del SMS se procederá de un modo semejante salvo una excepción, la URI en este caso en lugar de ser referente al teléfono, es referente a los SMS, por lo que será del tipo:

```
sms:número_teléfono
```

Por otro lado, la *Activity* que se encarga de permitir editar un SMS acepta un parámetro llamado *sms_body*, mediante el cual se le puede informar el cuerpo del SMS a enviar; este parámetro se pasa a través del uso del método `putExtra()` del objeto *Intent*.

```
protected void launchSms() {
   EditText url = (EditText) findViewById(R.id.url);
   EditText tlfNum = (EditText) findViewById(R.id.tlfnum);
   Intent i = new Intent(Intent.ACTION_VIEW, Uri.parse("sms:" + tlfNum.
   getText().toString()));
```

```
    i.putExtra("sms_body", url.getText().toString());
    startActivity(i);
}
```

Fíjese que para una misma acción `ACTION_VIEW`, se están obteniendo actividades resultantes distintas dependiendo del contenido de su dato.
Pero indicando la URI no es la única manera de llegar a la *Activity* de escritura de SMS, por ejemplo, se podría haber llegado indicándole al objeto *Intent* el tipo de dato:

```
protected void launchSms() {
    EditText tlfNum = (EditText) findViewById(R.id.tlfnum);
    Intent i = new Intent(Intent.ACTION_VIEW);
    i.putExtra("address", tlfNum.getText().toString());
    i.setType("vnd.android-dir/mms-sms");
    i.putExtra("sms_body", url.getText().toString());
    startActivity(i);
}
```

Donde se ha cambiado la URI por el tipo de dato a tratar y haciendo accesible el destinatario también a través de un extra. Antes de poder probar a realizar llamadas, es necesario dar permisos en la aplicación de lo contrario la aplicación terminará al intentar llamar. Modifique el dichero `AndroidManifest.xml` para añadir el permiso:

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

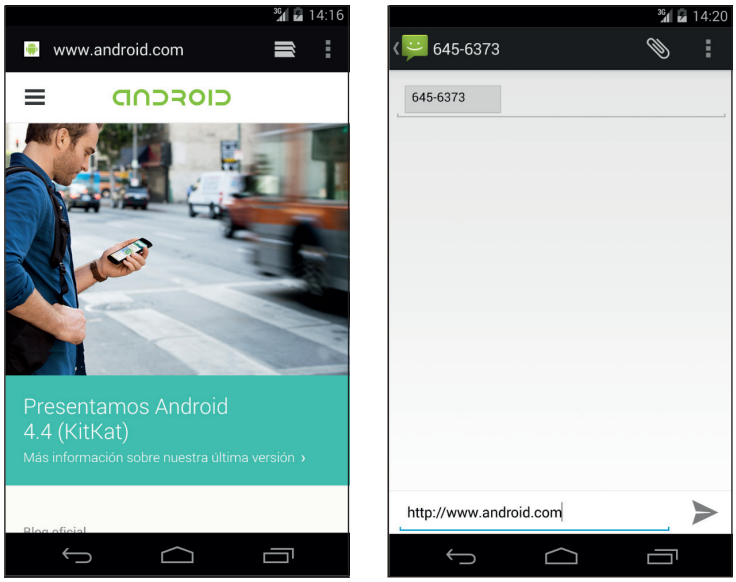

Figura 13.2. Ejecución de distintos Intent.

Mejorando Flip

Ahora que ya se conoce el mecanismo por el que llamar a otras actividades y cómo guardar datos en el sistema, se puede aprovechar para modificar el juego Flip y personalizarlo un poco más, de modo que cada jugador pueda poner sus datos e incluso elegir un avatar.
Lo primero que haremos será crear la ventana donde el jugador podrá configurar sus datos. Se le proporcionará una imagen por defecto que podrá cambiar en cualquier momento, una caja de texto para que introduzca su nombre y otra para la edad y un selector donde elegir el sexo.
El aspecto final será:

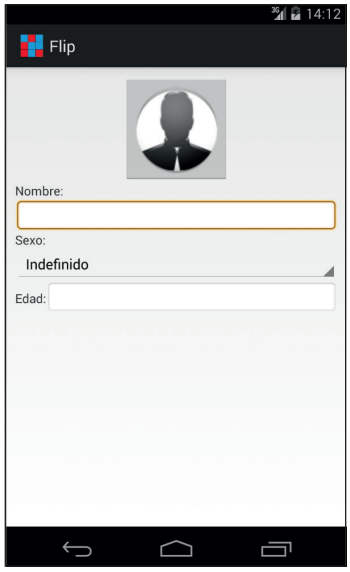

Figura 13.3. Pantalla de configuración del jugador en Flip.

Para ello creamos un fichero de *layout* y lo llamamos `user.xml` que será donde definamos la pantalla. Para la imagen del avatar usaremos un elemento `<ImageButton>`, que nos permitirá tener las características de un botón y además mostrar una imagen (aunque un elemento de tipo `<ImageView>` también puede ser pulsado), así que podremos mostrar la imagen actual del avatar, y en caso de ser pulsada reaccionar. Realmente la clase *Button* es una subclase directa de la clase *TextView*, mientras que la *ImageButton* lo es de la clase *ImageView*, por lo que una hará que el botón muestre un texto y la otra una imagen. El sexo del jugador lo seleccionaremos mediante una lista des-

plegable, que en Android está representada por el elemento `<Spinner>`, que se alimentará mediante un *array* de constantes con los posibles sexos a elegir. En cuanto a la edad, introduciremos una caja de texto con los atributos `android:numeric="integer"` y `android:maxLength="2"` fijados, de modo que en ella sólo se puedan introducir enteros y de un máximo de dos cifras (juego recomendado de 0 a 99 años). El código XML quedaría:

```
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  android:orientation="vertical"
  android:padding="10dp">
<ImageButton android:id="@+id/butAvatarImage" android:layout_width="120dp"
android:layout_height="120dp"  android:scaleType="centerCrop"
android:src="@drawable/user" android:layout_gravity="center"
android:cropToPadding="false"></ImageButton>
<TextView android:text="Nombre:"  android:layout_width="wrap_content"
android:layout_height="wrap_content"></TextView>
<EditText android:text="" android:id="@+id/avatarName" android:layout_
width="fill_parent" android:layout_height="wrap_content"
android:background="@android:drawable/editbox_background"></EditText>
<TextView android:text="Sexo:"  android:layout_width="wrap_content"
android:layout_height="wrap_content"></TextView>
<Spinner android:id="@+id/avatarSex" android:layout_width="fill_parent"
android:layout_height="wrap_content"></Spinner>
<LinearLayout
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  android:orientation="horizontal">
<TextView android:text="Edad:"  android:layout_width="wrap_content"
android:layout_height="wrap_content"></TextView>
<EditText android:text="" android:id="@+id/avatarAge"
android:layout_width="fill_parent" android:layout_height="wrap_
content"  android:background="@android:drawable/editbox_background"
android:numeric="integer" android:maxLength="2"></EditText>
</LinearLayout>
</LinearLayout>
```

Creamos la clase *Activity* correspondiente al *layout* anterior, la llamaremos `ConfigUser.java`, muy importante es que no extiende *Activity* sino *FragmentActivity* (ya que usaremos fragmentos en modo compatibilidad) y por el momento la dejaremos con el código mínimo para mostrar su *layout*:

```
public class ConfigUser extends Activity {
   @Override
      public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.user);
   }
}
```

Para mostrar esta actividad, se debe realizar un *Intent* desde el menú de opciones de la actividad principal. Aunque se pudiera utilizar una llamada explícita tal y como se ha hecho en otras ocasiones, en este caso se utilizará una llamada implícita. En la clase *GameConfig*, busque un método que se quedó sin rellenar llamado `showPlayer()` y añádale el código para la petición de una *Activity* capaz de atender una acción `ACTION_EDIT` del tipo de dato `"com.acme.flip/flip-user"` (que es un tipo MIME inventado para la ocasión):

```
private void showPlayer() {
   Intent i = new Intent(Intent.ACTION_EDIT);
   i.setType("com.acme.flip/flip-user");
   startActivityForResult(i, ACTION_CONFUSER);
}
```

Como se utiliza la llamada `startActivityForResult()`, se le debe informar un código de acción, por lo que creamos la constante `ACTION_CONFUSER` seguida de la ya existente `ACTION_PLAY`:

```
private static final int ACTION_PLAY = 1;
private static final int ACTION_CONFUSER = 2;
```

Se da de alta la clase nueva en el `AndroidManifest.xml` teniendo en cuenta que se deben definir los filtros que determinen que la actividad es capaz de atender peticiones de `ACTION_EDIT` sobre el tipo de dato `"com.acme.flip/flip-user"`:

```
<activity android:name=".ConfigUser" >
<intent-filter>
<action android:name="android.intent.action.EDIT" />
      <data android:mimeType="com.acme.flip/flip-user" />
      <category android:name="android.intent.category.DEFAULT" />
 </intent-filter>
</activity>
```

Advertencia:

Para que la actividad pueda ser llamada de forma implícita es necesaria la definición de la categoría `DEFAULT` *siempre que la actividad no tenga ya definida la acción* `MAIN` *y categoría* `LAUNCHER`.

Ya se tiene la *Activity* lista para ser accedida, aunque sin mucha funcionalidad.Hay dos aspectos a tratar en esta pantalla, uno es la selección del avatar y otro es guardar los datos para recuperarlos más adelante. En primer lugar nos ocuparemos de la selección del avatar.

Selección de avatar

Cuando pulse sobre la imagen del avatar, se le ofrecerá al jugador la posibilidad de utilizar como avatar una imagen que tenga almacenada en su ordenador o bien una imagen que sea tomada por la cámara de fotos. El método de selección entre las dos opciones será mediante una pequeña pantalla de *popup*.
La imagen del botón puede ser configurada de dos modos, uno es en tiempo de diseño mediante la propiedad `android:src` y el otro es mediante programación a través de uno de los métodos siguientes dependiendo de cada circunstancia y del tipo de acceso que tengamos al recurso a mostrar:

- setImageBitmap(): En caso de tener un objeto *Bitmap* válido.
- setImageDrawable():En caso de tener un objeto *Drawable* válido.
- setImageResource(): En caso de que el objeto a mostrar tenga un identificador de recurso válido (por ejemplo usando la clase *R*).
- setImageURI(): En caso de tener que acceder a una URI donde resida el objeto a mostrar.

En este caso, en tiempo de diseño no conocemos la imagen que se debe mostrar en el botón, así que habrá que usar alguno de los métodos anteriores, en concreto `setImageURI()`, ya que se leerá el recurso directamente de un fichero.

Truco:

A la hora de mostrar imágenes en un ImageButton*, es altamente recomendable que las imágenes se encuentren guardadas localmente, ya que aunque es posible obtenerlas mediante acceso a servidores remotos, esto haría que la aplicación no respondiera de modo fluido, creando malas sensaciones al usuario.*

En la clase *ConfigUser* crearemos unas constantes para las distintas acciones de seleccionar el avatar y definimos una variable miembro para mantener la referencia al elemento `<ImageButton>` del *layout*:

```
private  ImageButton avatar = null;
//acciones
static final int ACTION_AVATAR_PHOTO = 1;
static final int ACTION_AVATAR_GALLERY = 2;
```

Dentro del evento `onCreate()` obtendremos dicha referencia y se le asignará código a ejecutar al recibir una pulsación:

```
//código para el imageButton
avatar = (ImageButton) findViewById(R.id.butAvatarImage);
avatar.setOnClickListener(new OnClickListener() {
public void onClick(View arg0) {
      showPhotoDialog();
    }
});
```

El hecho de extender la clase *Activity*, proporciona la posibilidad de mostrar pantallas en modo *popup* con fragmentos de una manera sencilla. Hay que tener en cuenta que esta manera sólo está disponible en versiones posteriores a la versión 3.0 API 11, si se pretende utilizar anteriores, bien se puede usar el método `showDialog()` (actualmente obsoleto) o bien usar la librería de compatibilidad. En nuestro caso, como seleccionamos de API mínimo el 7 (el que viene por defecto), debemos usar la librería de compatibilidad y por lo tanto la clase que llama al fragmento diálogo debe extender *FragmentActivity*. La creación de los diálogos se realiza de modo semejante a la de las pantallas, mediante un *layout* de interfaz definido en un fichero y asignación de código a cada elemento que lo necesite. En nuestro caso el diálogo para seleccionar el método de obtención de la fotografía del avatar estará compuesto por una serie de botones, uno para captar una fotografía, otro para seleccionar una imagen ya existente, otro para poner de nuevo la imagen por defecto y otro para cancelar la operación.
El *layout* del cuadro de diálogo guardado en el archivo `fragment_select_avatar.xml` será:

```
<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:text="@string/selectav_camera" android:id="@+id/butAvCamera"
android:layout_width="fill_parent" android:layout_height="wrap_content"></
Button>
<Button android:text="@string/selectav_gallery" android:id="@+id/
butAvGallery" android:layout_width="fill_parent" android:layout_
height="wrap_content"></Button>
<Button android:text="@string/selectav_default" android:id="@+id/
butAvDefault" android:layout_width="fill_parent" android:layout_
height="wrap_content"></Button>
<Button android:text="@string/selectav_cancel" android:id="@+id/butAvCancel"
android:layout_width="fill_parent" android:layout_height="wrap_content"></
Button>
</LinearLayout>
```

Con sus correspondientes entradas en `strings.xml`:

```
<string name="selectav_method">Metodo de selección</string>
<string name="selectav_camera">Cámara</string>
<string name="selectav_gallery">Galería</string>
```

```
<string name="selectav_default">Por defecto</string>
<string name="selectav_cancel">Cancelar</string>
```

Para controlar este *layout*, realizaremos una clase que extienda la clase *DialogFragment*, de modo que heredemos todos los métodos de ayuda. La clase se llamará `PhotoSelectorDialog` y la colocaremos en el mismo *package* que el resto de la aplicación, aunque es muy recomendable que en aplicaciones grandes se clasifiquen las clases bien por área de utilidad o bien separando actividades y fragmentos para localizar rápido la entidad buscada. Su contenido:

```
public class PhotoSelectorDialog extends DialogFragment {
   private EditText mEditText;

   public PhotoSelectorFrg() {
      // Se necesita un constructor vacío para DialogFragment
   }

   @Override
   public View onCreateView(LayoutInflater inflater, ViewGroup container,
                            Bundle savedInstanceState) {
      View dlgview = inflater.inflate(R.layout.fragment_select_avatar,
      container);

      //ponemos el título al diálogo
       getDialog().setTitle(R.string.selectav_method);

      //cámara
      Button camera = (Button) dlgview.findViewById(R.id.butAvCamera);
      camera.setOnClickListener(new View.OnClickListener() {
         public void onClick(View arg0) {
      }
      });

      //Galeria
      Button gallery = (Button) dlgview.findViewById(R.id.butAvGallery);
      gallery.setOnClickListener(new View.OnClickListener() {
         public void onClick(View arg0) {
      }
      });

      //por defecto
      Button defaultAvatar = (Button) dlgview.findViewById(R.id.butAvDefault);
      defaultAvatar.setOnClickListener(new View.OnClickListener() {
         public void onClick(View arg0) {
      }
      });

      // cancelar
      Button cancel = (Button) dlgview.findViewById(R.id.butAvCancel);
      cancel.setOnClickListener(new View.OnClickListener() {
      public void onClick(View arg0) {
         dismiss();
      }
      });
      return dlgview;
      }
}
```

Recuerde que cuando importe la clase *DialogFragment* debe ser la *android.support.v4.app.DialogFragment* y no la *android.app.DialogFragment*, ya que estamos usando la librería de compatibilidad para poder trabajar con API 7.
En el método `onCreateView()` de esta clase, se infla el *layout* `R.layout.fragment_select_avatar`, se le ha dado un título al diálogo mediante `getDialog().setTitle()` y se le han asignado acciones a los botones... bueno, realmente sólo hemos asignado acción a uno de ellos, el resto simplemente los hemos dejado preparados para más adelante.

Truco:

Del mismo modo que con las actividades, disponemos de un asistente para generar fragmentos. Para acceder a él se debe pulsar con el botón derecho sobre el proyecto y seleccinar en el menú New > Android Component.

Dentro del método `showPhotoDialog()` de la clase *ConfigUser* será donde se llame a este diálogo. Se realizará obteniendo la referencia al gestor de fragmentos del paquete de compatibilidad, instanciando el fragmento a mostrar y haciéndolo visible:

```
private void showPhotoDialog() {
   FragmentManager fm = getSupportFragmentManager();
   PhotoSelectorDialog avatarDialog= new PhotoSelectorDialog();
   avatarDialog.show(fm, "photo");
    }
```

Antes de poder probar el diálogo, hay que asegurarnos que la clase *ConfigUser* extiende la clase *FragmentActivity* (véase figura 13.4.).
Para capturar una nueva imagen a través de la cámara del dispositivo utilizaremos una llamada implícita a alguna actividad que nos deje hacer uso de ella y que más adelante podamos recuperar, por lo que se debe llamar a la actividad mediante el método `startActivityForResult()`. Dentro de `onCreateView()` asignamos código al primer botón variando su código:

```
camera.setOnClickListener(new OnClickListener() {
   public void onClick(View arg0) {
       ((ConfigUser) getActivity()).avatarPhoto();
                 dismiss();
   }
});
```

En este método estamos llamando al método `avatarPhoto()` de la clase *ConfigUser*, para hacerlo obtenemos la actividad a la que está asociada el fragmento mediante `getActvity()` y realizamos un cast a la clase *ConfigUser*

que es la que realmente está asociada .Tras la llamada, eliminamos el diálogo de selección.
En la clase *ConfigUser* tendremos que crear el método `avatarPhoto()`:

```
public void avatarPhoto() {
   Intent pictureIntent = new Intent(android.provider.MediaStore.ACTION_IMAGE_
   CAPTURE);startActivityForResult(pictureIntent, ACTION_AVATAR_PHOTO);
}
```

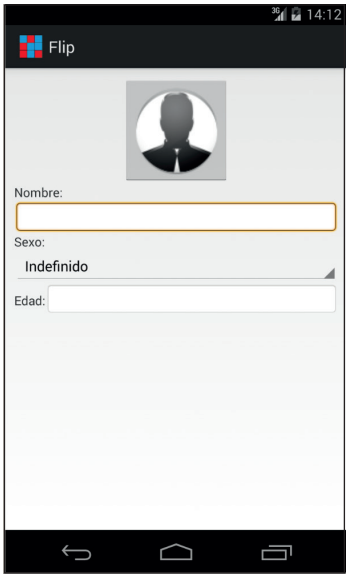

Figura 13.4. Diálogo de selección de avatar.

El botón de cámara realmente lanzará un *Intent* para que se abra la actividad que corresponda a la acción `android.provider.MediaStore.ACTION_IMAGE_CAPTURE`, que dependiendo de cada sistema se abrirá una *Activity* u otra. Por defecto se abrirá la acción de la cámara de sistema, pero si el usuario tiene otros programas instalados que se han registrado como receptores de la acción, es posible que muestre otra aplicación distinta. Como el *Intent* se lanza mediante `startActivityForResult()`, cuando la ejecución vuelva a la actividad, lo hará llamando al método `startActivityForResult()` con el código `ACTION_AVATAR_PHOTO` de modo que cuando retorne podremos actuar en consecuencia dentro del método `onActivityResult()` de la clase *ConfigUser*.
Para poder seleccionar una imagen de la galería, utilizaremos también alguna *Activity* que nos ofrezca esta posibilidad en lugar de programar nosotros una. En este caso, para el botón que guiará a la galería de imágenes haremos algo

semejante, se crea un *Intent* para la acción seleccionar (`ACTION_PICK`) un elemento del tipo imagen (`"image/*"`) y se lanza la petición con el código de acción `ACTION_AVATAR_GALLERY` para poder detectarlo después en el método `onActivityResult()` y recuperar el elemento seleccionado. Varíe el código del botón correspondiente dentro del método `onCreateDialog()`:

```
gallery.setOnClickListener(new OnClickListener() {
   public void onClick(View arg0) {
      ((ConfigUser) getActivity()).avatarGallery();
      dismiss();
   }
});
```

Igualmente, generamos el método en la clase *ConfigUser.*

```
public void avatarGallery() {
        Intent pickPhoto = new Intent(Intent.ACTION_PICK);
        pickPhoto.setType("image/*");
        startActivityForResult(pickPhoto, ACTION_AVATAR_GALLERY);
    }
```

El código correspondiente al botón de seleccionar la imagen por defecto del usuario por ahora lo dejamos sin código y más adelante volveremos sobre él. Ya es posible utilizar tanto la cámara de fotos como la galería para seleccionar la imagen que se quiere mostrar como avatar, pero una vez seleccionada la imagen no se está haciendo nada con ella, es decir, se toma la foto pero la foto es olvidada porque no se guarda en el sistema ni se muestra en la aplicación. Todo esto lo controlaremos desde el método `onActivityResult()` dentro de la clase *ConfigUser*.

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
if (resultCode == Activity.RESULT_CANCELED){
    // no hacer nada
}
else if (resultCode == Activity.RESULT_OK) {
    switch(requestCode) {
    case ACTION_AVATAR_PHOTO:
       Bitmap cameraPic = (Bitmap) data.getExtras().get("data");
       saveAvatar(cameraPic);
    break;
    case ACTION_AVATAR_GALLERY:
      Uri photoUri = data.getData();
      try {
      Bitmap galleryPic = MediaStore.Images.Media.
getBitmap(getContentResolver(), photoUri);
      saveAvatar(galleryPic);
      } catch (Exception e) {
         new AlertDialog.Builder(this)
              .setMessage( "ERROR: " + e.getLocalizedMessage())
                 .setPositiveButton(android.R.string.ok,null)
```

```
                .show();
        }
    break;
    }
 }
 }
```

Lo primero es comprobar que la actividad requerida no haya sido cancelada, es decir, por ejemplo una vez estando en la galería, el usuario decide no cambiar la imagen del avatar y pulsa sobre la tecla volver, entonces no se debe realizar ninguna acción.

```
if (resultCode == Activity.RESULT_CANCELED){
    // no hacer nada
}
else if (resultCode == Activity.RESULT_OK) {
…
}
```

En este código no tenemos que hacer nada, pero es posible que en otras ocasiones se deban liberar recursos o informar al usuario de que se ha cancelado. En caso de que el código devuelto por la actividad sea `Activity.RESULT_OK`, quiere decir que todo ha ido correcto y que el usuario o ha aceptado la foto realizada o ha seleccionado una imagen, así que mediante un `switch` vemos el código de petición que se había realizado para saber cómo se debe recuperar el objeto gráfico:

```
switch(requestCode) {
case ACTION_AVATAR_PHOTO:
      …
break;
   case ACTION_AVATAR_GALLERY:
…
break;
}
```

En caso de haberse usado la cámara, la foto tomada se devuelve a modo de objeto *Bitmap* dentro del *Bundle* devuelto por la actividad bajo la clave `"data"`. Para obtenerlo hemos utilizado:

```
Bitmap cameraPic = (Bitmap) data.getExtras().get("data");
```

En el caso de que el usuario prefiriera seleccionarlo de mediante la galería, el método de recuperación del gráfico es distinto, ya que la actividad solamente devuelve la URI que apunta al gráfico seleccionado, no el gráfico en si, por lo que debemos generar un *Bitmap* a partir de esa URI:

```
Uri photoUri = data.getData();
```

Y leemos el contenido de dicha URI en un objeto gráfico *Bitmap*:

```
Bitmap galleryPic = Media.getBitmap(getContentResolver(), photoUri);
```

Se le ha rodeado por un bloque try-catch para avisar al usuario si se produjera un error al obtener el gráfico indicado.
Una vez tenemos en los dos casos un objeto *Bitmap*, llamamos al método `saveAvatar()` para más adelante guardar dicho *Bitmap* como un fichero de tipo gráfico al que acceder en cualquier momento, pero por el momento sólo lo asignaremos al *ImageButton*. El código que se encarga de hacer esto es:

```
private void saveAvatar(Bitmap bitmap){
    // mostramos el avatar en el botón
    ImageButton avatar = (ImageButton) findViewById(R.id.butAvatarImage);
     avatar.setImageBitmap(bitmap);
}
```

La aplicación ya deja tomar fotografías o seleccionar una imagen para utilizar como avatar, el problema es que cuando se sale de la aplicación y se vuelve a entrar, estos datos, junto con el resto de los datos del jugador, se han perdido; habrá que solucionarlo.

Guardar configuración

En el capítulo sobre persistencia se vio una manera muy sencilla sobre como guardar datos de una aplicación a través de las preferencias. Este será el método que utilizaremos para guardar la configuración de los usuarios.
En la clase *ConfigUser* se define una variable que contendrá el objeto *SharedPreferences* que se utilizará para guardar y recuperar los valores configurados. También se deben definir las claves que permitirán recuperar más adelante cada uno de esos datos a nivel de atributo de la clase.

```
private SharedPreferences preferences = null;
//claves
static final String AVATAR = "avatar";
static final String NAME = "name";
static final String YEARS = "age";
static final String SEX = "sex";
```

Ya en el método `onCreate()`, se obtiene la referencia a un objeto *SharedPreferences* para poder trabajar con él, añada la línea:

```
//obtener las preferencias
preferences = getPreferences(0);
```

También hay que cargar los datos en la lista de sexos disponibles. Para ello se crea un *array* con los textos posibles a elegir, y se le asigna al *Spinner* a través de un adaptador para *arrays* (*ArrayAdapter*). En la creación del *ArrayAdapter*, se le debe proporcionar un *layout*, que será el que utilizará para dar formato a cada elemento de la lista antes de mostrarlo. Como en nuestro caso interesa simplemente que muestre el texto que corresponda a la entrada del *array* que vaya a

mostrar en cada momento, podemos utilizar un *layout* que ya da por defecto Android para estos casos, llamado `R.layout.simple_spinner_item`. En caso de querer mostrar listas más complejas, se podría utilizar un *layout* propio o incluso crearnos nuestros propios adaptadores para poder tener un control mayor de la apariencia de cada elemento, modifique `onCreate()` añadiendo:

```
// cargar spiner con valores
Spinner sex = (Spinner) findViewById(R.id.avatarSex);
String array_spinner[] =new String[3];
Resources res = getResources();
array_spinner[0]= res.getString(R.string.usr_unknown);// "Indefinido"
array_spinner[1]=res.getString(R.string.usr_man);// Hombre;
array_spinner[2]=res.getString(R.string.usr_woman);// Mujer;
ArrayAdapter adapter = new ArrayAdapter(this,
        android.R.layout.simple_spinner_item, array_spinner);
sex.setAdapter(adapter);
```

El hecho de haber creado el *array* mediante código es para mostrar que a la hora de definir arrays, se puede hacer en tiempo de diseño mediante constantes en el archivo `strings.xml`, archivos externos o en tiempo de ejecución. En este caso los valores del *Spinner* los hemos asignado mediante constantes de texto, pero podríamos haber utilizado variables, por ejemplo si obteniendo los valores de una base de datos. Las constantes utilizadas y definidas en `strings.xml` son:

```
<string name="usr_man">Hombre</string>
<string name="usr_woman">Mujer</string>
<string name="usr_unknown">Indefinido</string>
```

Y establecemos los valores guardados en las preferencias también el método `onCreate()` de la clase *ConfigUser*:

```
// establecer el avatar seleccionado anteriormente
setAvatar();
// restablecer el resto de valores
EditText ed = (EditText) findViewById(R.id.avatarName);
ed.setText(preferences.getString(NAME, ""));
ed = (EditText) findViewById(R.id.avatarAge);
ed.setText(preferences.getString(YEARS, ""));
String sexString = preferences.getString(SEX, "");
// lista desplegable
sex.setSelection(adapter.getPosition(sexString));
```

En este último bloque de código del método `onCreate()` se establece el avatar que había seleccionado la última vez mediante `setAvatar()`. Se ha aislado en un método propio ya que será utilizado en varios puntos del programa y así evitaremos repetir código. También se establecen los valores guardados para las cajas de texto y el *Spinner*.
La manera de seleccionar un elemento en un *Spinner* es mediante el método `setSelection()`, este método sólo acepta un entero como parámetro. El

entero representa la posición del elemento que tiene que ser seleccionado, pero en nuestras preferencias vamos a guardar no la posición, sino el literal y aunque no es una buena práctica lo haremos así porque nos servirá para conocer cómo recuperar la posición de un elemento en un *Spinner* a través de su literal, y esto se hace mediante su adaptador. El adaptador posee un método llamado `getPosition()` al que se le puede pasar como parámetro un objeto y el método devolverá la posición en la que se encuentra el objeto pasado, por eso en el código, una vez recuperado el literal de las preferencias, lo utilizamos para conseguir la posición que ocupa en el adaptador y así poder seleccionarlo en el *Spinner,* lo hacemos mediante*:*

```
String sexString = preferences.getString(SEX, "");
sex.setSelection(adapter.getPosition(sexString));
```

Para recuperar y asignar el avatar gráfico al botón se ha utilizado la función `setAvatar()` que será:

```
private void setAvatar(){
// intentar recuperar el avatar:
     String uriString = preferences.getString(AVATAR, "android.resource://
com.acme.flip/drawable/user");
        avatar.setImageURI(Uri.parse(uriString));
   }
```

En ella se puede ver cómo se recupera el valor de las preferencias y cómo se asigna la imagen al objeto *ImageButton* mediante `setImageURI()`. Esto quiere decir, que cuando guardemos el avatar que quiera utilizar el usuario en las preferencias, habrá que salvar la URI a ese recurso. Al recuperar la cadena `AVATAR` de las propiedades, se le ha dicho que en caso de no encontrarla, devuelva por defecto la cadena de texto: `android.resource://com.acme.flip/drawable/user`.

Esta es una manera que tiene Android de referirse mediante URI a sus recursos. Con ello estamos apuntando al recurso (`android.resource://`) que está dentro del paquete java `com.acme.flip` (que es el paquete principal de nuestra aplicación) y que concretamente se refiere a un recurso gráfico llamado `user` (`drawable/user`), donde `user` será efectivamente algún gráfico que tengamos guardado en la carpeta `res/drawable` de nuestro proyecto. Es decir esta URI tiene la forma:

```
android.resource://<paquete>/<tipo de recurso>/<nombre de recurso>
```

Otra manera de acceder mediante URI a los recursos, es si se conoce el identificador de recurso, el entero que se asigna en la clase *R*. En este caso la URI sería

```
android.resource://<paquete>/<id de recurso>
```

por ejemplo

```
android.resource://com.acme.flip/" + R.drawable.user;
```

Del cuadro de diálogo que permitía la selección de la imagen del avatar, nos habíamos dejado uno de los botones sin código, el correspondiente a dejar la imagen por defecto, es hora de recuperarlo y asignarle el siguiente código:

```
defaultAvatar.setOnClickListener(new OnClickListener() {
   public void onClick(View arg0) {
      ((ConfigUser) getActivity()).avatarDefault();
      dismiss();
   }
});
```

Y su correspondiente función en la clase *ConfigUser*:

```
public void avatarDefault() {
   SharedPreferences.Editor editor = preferences.edit();
   editor.remove(AVATAR);
   editor.commit();
   setAvatar();
  }
```

Como ya tenemos una función específica para que recupere el avatar de las preferencias y en caso de que no encuentre nada devuelva el avatar por defecto, podemos aprovecharla eliminando la preferencia correspondiente al avatar, así, cuando se le diga a la aplicación que nuevamente vuelva a poner el avatar correspondiente en el *ImageButton*, no encontrará nada y pondrá el avatar por defecto. Esto es lo que se ha hecho en el código anterior. Mediante el método `remove()` se elimina el contenido de la clave `AVATAR` de las preferencias y al llamar al método `setAvatar()` ya no lo encontrará y mostrará la imagen por defecto, que es la esperada.

Hasta ahora hemos recuperado los elementos de las preferencias, pero aún no los hemos guardado. En lugar de poner un botón de salvar, lo haremos en el método `onPause()` de modo que todo (excepto la imagen del avatar que usaremos otro mecanismo) quedará guardado al abandonar la *Activity*.

```
@Override
protected void onPause() {
   super.onPause();
//guardar preferencias
   SharedPreferences.Editor edit = preferences.edit();
   EditText ed = (EditText) findViewById(R.id.avatarName);
   edit.putString(NAME, ed.getText().toString());
   ed = (EditText) findViewById(R.id.avatarAge);
   edit.putString(YEARS, ed.getText().toString());
   Spinner sp  = (Spinner) findViewById(R.id.avatarSex);
   String sex = sp.getSelectedItem().toString();
   edit.putString(SEX, sex);
   edit.commit();
}
```

Lo que se hace es obtener un editor de las preferencias y se van guardando todos los valores de la configuración del avatar, recuperando cada uno de los

elementos de pantalla y obteniendo los datos introducidos y por último se guarda el editor mediante `edit.commit()`.

Para guardar la imagen del avatar lo que se hará es aprovechar el método `saveAvatar()` que ya se tiene con algo de código. Hasta ahora simplemente se colocaba la imagen en el *ImageButton* con lo que se perdía cuando volvíamos a entrar en el programa. Lo que se hará será guardar físicamente la imagen (el objeto *Bitmap*) en un fichero (de nombre `avatar.jpg`) para poder acceder a él más adelante.

Se guarda mediante el método que ofrece la clase *Bitmap* llamado `compress()` que guarda una versión comprimida del gráfico en un *OutputStream* que se le pase como parámetro. Los tipos posibles de compresión son JPEG y PNG, en este caso lo guardaremos como .jpg con una calidad del 90% (a más calidad mayor tamaño de fichero). Para la obtención del *OutputStream*, usamos el método `openFileOutput()` proporcionado por el *Context* de la aplicación, a este método se le informa el nombre del fichero con el que se quiere guardar. La URI del archivo recién creado se obtiene mediante una utilidad de la clase *Uri* que permite obtener el objeto *Uri* a partir de un objeto *File*:

```
Uri imageUri = Uri.fromFile(new File(getFilesDir(), strAvatarFilename));
```

A través del método `getPath()` del objeto *Uri* se puede obtener una cadena de texto que representa unívocamente al URI. Ahora al tener una cadena de de texto ya se puede guardar en las preferencias mediante un editor. Rodearemos también el código mediante un try-catch para informar al usuario en caso de que haya ido algo mal a la hora de guardar el fichero. El código completo quedaría:

```
private void saveAvatar(Bitmap bitmap){
// mostramos el avatar en el botón
ImageButton avatar = (ImageButton) findViewById(R.id.butAvatarImage);
avatar.setImageBitmap(bitmap);
//Se guarda el avatar en fichero y en las preferencias el nombre del mismo
String strAvatarFilename = "avatar.jpg";
SharedPreferences.Editor editor = preferences.edit();
try {
   bitmap.compress(Bitmap.CompressFormat.JPEG,
       90, openFileOutput(strAvatarFilename, MODE_PRIVATE));
   Uri imageUri = Uri.fromFile(new File(getFilesDir(), strAvatarFilename));
   editor.putString(AVATAR, imageUri.getPath());
   editor.commit();
} catch (FileNotFoundException e) {
         new AlertDialog.Builder(this)
           .setMessage( "ERROR: " + e.getMessage())
           .setPositiveButton(android.R.string.ok,null)
           .show();
   }
}
```

La aplicación ya está lista para ser utilizada y batir sus propios records.





14 Gráficos

En este capítulo aprenderá a:

- El uso de las clases *Drawable*.
- Qué son los *Nine Patch Drawables*.
- Diferenciar los tipos de animaciones.
- Configurar pinceles y pintar sobre un *Canvas*.
- Utilizar el Navigation Drawer.

El sistema operativo Android ofrece múltiples posibilidades a la hora de trabajar con gráficos, tanto en 2D como en 3D. En cuanto a los gráficos 3D simplemente reseñar que implementa el estándar OpenGL ES 2.0 para dispositivos portátiles desde la API 8 de versión de Android, OpenGL ES 3.0 desde la API 18 e implementando OpenGL ES 1.0 y 1.0 en versiones anteriores; pero no se verá nada en este libro por cuestiones de espacio ya que la programación en 3D es compleja y extensa.
En cuanto a los gráficos 2D Android provee de clases para trabajar con ficheros de imágenes ya existentes en formatos conocidos como png o jpg entre otros, y también de herramientas para crear imágenes mediante programación.
Cuando se va a realizar una aplicación que deba dibujar algún gráfico 2D en pantalla, se debe pensar en qué modo se quiere realizar, ya que dependiendo de las exigencias será mejor realizarlo de una manera o de otra. Las opciones posibles son:

- Mediante *View*: Es la más sencilla de realizar, incluso existe un elemento gráfico llamado *ImageView* que sirve para mostrar imágenes. La pega que tiene este método es que no es muy flexible. Es el indicado para imágenes estáticas.
- Mediante *Canvas*: El objeto *Canvas* permite dibujar directamente sobre él figuras geométricas, gráficos leídos de fichero, etc…, lo que lo hace ideal para pequeñas animaciones y dibujos generados por programación. Una manera ideal de utilizar el objeto *Canvas* es implementar una clase que extienda la clase *View* y sobrescribir el método `onDraw()` del mismo, ya que tendremos acceso a su *Canvas*. La manera de forzar a que la vista ejecute el método `onDraw()` y por lo tanto forzar su redibujado es mediante el método `invalídate()`. En este capítulo se verá como implementar este método.
- Mediante *SurfaceView*: Esta manera de trabajar está especialmente indicada cuando se van a realizar muchas actualizaciones de la información dibujada en la superficie. La superficie a pintar es controlada por un *thread* distinto del principal de la aplicación, lo que hace que este segundo *thread* pueda ir dibujando sin esperar a bloqueos que puedan sucederse en el *thread* principal tales como pulsaciones en pantalla o cálculos matemáticos. Más adelante en el capítulo 16 se verá un ejemplo.

Drawable

La clase *Drawable* es una clase abstracta para representar todo aquello que se pueda dibujar (normalmente en un *Canvas*), por ejemplo los recursos gráficos que se añaden a la aplicación o formas geométricas e incluso animaciones. El

uso de elementos "*drawables*" tiene como hándicap que no pueden recibir eventos de modo sencillo, por lo que deberán ser embebidos en otras vistas si es que se desea que los reciban. A este tipo de objetos se les puede informar un parámetro alfa para indicar su nivel de transparencia, que a la hora de ser dibujados se combinará con el parámetro alfa informado en el pincel *Paint* utilizado para dibujarlo.

Existen multitud de clases directamente descendientes de la clase *Drawable,* algunas de ellas representan imágenes estáticas, como por ejemplo la *BitmapDrawable*, otras de ellas son animaciones como *TransitionDrawable*, otras representan estados *StateListDrawable*… pero el hecho de poder trabajar con todas ellas como objetos *Drawables* posibilita hacer interfaces más configurables. Cuando se utilizan *Drawables*, podemos estar indicado un simple gráfico, un *StateListDrawable* para indicar el estado de seleccionado o no seleccionado, una animación, o una forma geométrica o un gradiente o…. las posibilidades son múltiples, como múltiples son las utilidades.

Las clases *Drawables* normalmente encapsulan otras clases para darles mayor funcionalidad, por ejemplo la clase *BitmapDrawable* contiene una clase *Bitmap* a la que provee de utilidades para poder ser dibujada.

Cuando se quiere dibujar algún tipo de forma geométrica la clase *ShapeDrawable* servirá para cubrir nuestras necesidades. Mediante esta clase se puede dibujar en pantalla desde formas ovaladas hasta rectángulos, arcos o rutas de puntos mediante sus respectivas clases (`OvalShape`, `PathShape`, `ArcShape`…) utilizando el pincel *Paint* que se desee en cada caso.

Otro objeto *Drawable* que nos podemos encontrar es el denominado *NinePatchDrawable*, que es un archivo de tipo png y con extensión `.9.png` utilizado para imágenes que serán estiradas y encogidas, como por ejemplo el fondo de imágenes de botones. Se trata de una imagen a la que se le añade un borde de 1 pixel sobre el cual se delimita la zona que se puede estirar o encoger mediante una línea negra en los bordes izquierdo y superior. Imagine un botón con las esquinas redondeadas; si se estira mucho toda la imagen proporcionalmente, las esquinas quedarán muy grandes; mediante un "nine patch" se puede hacer que sólo la parte central se redimensione, manteniendo las esquinas intactas. Se utilizan mucho para definir estilos para aplicar a los elementos gráficos de la pantalla. Más adelante en el libro se explicará su uso y un programa que facilita su creación.

Los objetos *Drawables* pueden representar múltiples objetos gráficos, por ejemplo imágenes cargadas desde un archivo, dibujos geométricos, estados… pero también animaciones.

Introducción a las animaciones

En Android las animaciones se clasifican en dos tipos, las que trabajan sobre un mismo objeto (denominadas animaciones *tween*) y las que trabajan sobre varias imágenes (lo que normalmente se conoce como animación) mostrando una tras otra como si de una película se tratara (llamadas animaciones *frame*).

Las animaciones *tween* son simples transformaciones sobre un objeto *View*. Las transformaciones que se pueden aplicar son rotación, traslación, escala y modificación de la transparencia. Las animaciones se pueden definir bien mediante un archivo XML que se guardará en el directorio `/src/main/res/anim` de proyecto o bien desde código Java. Cuando se definen estas animaciones, se informa cuando deben ejecutarse, cuánto debe durar la animación y por supuesto qué se quiere hacer.

Como todo documento XML, las definiciones tienen un elemento raíz que puede ser del tipo `<alpha>` para indicar cambios de transparencia, `<rotate>` para rotaciones, `<scale>` para escalas, `<translate>` para traslaciones y `<set>` para efectuar varias de estas acciones bien en paralelo o bien secuencialmente. También es posible el uso del elemento *Interpolator* en cualquiera de sus variantes (*AccelerateDecelerateInterpolator*, *AccelerateInterpolator*, *BounceInterpolator*…) para calcular la tasa de cambio de imagen en la animación. Por ejemplo se podría crear una animación para rotar 180º un objeto, que la rotación se realice desde su punto medio, que dure un segundo y que tarde siete en comenzar mediante:

```
<set xmlns:android="http://schemas.android.com/apk/res/android">
<rotate
android:fromDegrees="0" android:toDegrees="180"
android:toYScale="0.0" android:startOffset="5000"
android:duration="1000" android:pivotX="50%" android:pivotY="50%"/>
</set>
```

Si la guardamos en un fichero llamado `rotate.xml`, se la podríamos asignar a un botón (o a cualquier otra *View*) a través de:

```
Button b = (Button) findViewById(R.id.Button01);
Animation rotate = AnimationUtils.loadAnimation(MainActivity.this, R.anim.
rotate);
b.setAnimation(rotate);
```

Las animaciones *frame* también es posible definirlas mediante XML o mediante código Java a través de la clase *AnimationDrawable*. Para el caso del XML se define un documento con un elemento raíz del tipo `<animation-list>` y con todos los componentes de la animación y su duración "en pantalla". Por ejemplo:

```
<animation-list xmlns:android="http://schemas.android.com/apk/res/android"
      android:oneshot="false">
   <item android:drawable="@drawable/frame1" android:duration="100" />
   <item android:drawable="@drawable/frame2" android:duration="600" />
   <item android:drawable="@drawable/frame3" android:duration="600" />
   <item android:drawable="@drawable/frame4" android:duration="200" />
</animation-list>
```

Con este XML se define una animación compuesta de cuatro cuadros que durará 1.5 segundos en total y que al acabar con el último cuadro volverá a repetir el primero y así sucesivamente. En caso de querer que al acabar la animación se quede en el último cuadro, se debe asignar al atributo `android:oneshot` el valor `"true"`.
Si por ejemplo está asignado como fondo para un *ImageButton*, podríamos hacer que la animación comenzara mediante:

```
ImageButton ib = (ImageButton) findViewById(R.id.Button01);
AnimationDrawable animation  = (AnimationDrawable) ib.getBackground();
animation.start();
```

Más adelante cuando se hayan adquirido algunos conocimientos más, volveremos a atender las animaciones con varios ejemplos.

La pizarra

A todos cuando éramos pequeños nos gustaba mancharnos las manos con témperas y pintar con ellas sobre papeles (y sobre la mesa cuando se acababa el papel) y luego cuando venían los padres nos reñían al ver los papeles, la mesa y la ropa toda manchada. Vamos a hacer una aplicación donde podamos pintar con el dedo sin mancharnos y guardar nuestro trabajo para la posteridad; vamos a hacer una pizarra.
Cree un nuevo proyecto con la estructura:

- Application name: Pizarra
- Module Name: Pizarra
- Package name: com.acme.blackboard
- Minimum required SDK: API 13: Android 3.2 (Honeycomb) o superior
- Activity Name: Blackboard
- Aditional Features: Navigation Drawer

El *Navigation Drawer* (cajón de navegación), como su nombre indica, es un componente principalmente utilizado para controlar a qué sección (pantalla) de la aplicación se quiere ir. Si nos fijamos en el código generado, esto se realiza mediante fragmentos en el método *onNavigationDrawerItemSelected()* de

la clase *Blackboard*, este componente lo podemos encontrar en muchas aplicaciones por ejemplo en la oficial de Gmail. En nuestro caso en lugar de usarlo para navegar, lo usaremos para albergar la paleta de colores de nuestra pizarra, no nos molestaremos en eliminar los métodos que no se usen, simplemente lo adecuaremos para que funcione a nuestras necesidades.
Para pintar sobre una superficie, se podría optar por utilizar una vista cualquiera y dibujar sobre ella, pero para obtener un mayor control sobre el objeto, vamos a crearnos nuestra propia vista y así poder indicar qué dibujar y bajo qué circunstancias hacerlo. Cree una clase interna *Board* que extienda la clase *View* `dentro` de la clase principal donde iremos incluyendo el código que se encarga de pintar.

```
public class Board extends View{
 public Board(Context context) {
    super(context);
 }
 }
```

Como lo que se quiere es poder pintar con el dedo, se debe detectar el lugar donde empieza la pulsación sobre la pantalla, donde termina e ir pintando ese camino trazado. Los trazos a pintar se almacenarán en un objeto *Path* (ruta) que será lo que luego se irá pintando en la superficie. También será necesario mantener un objeto *Bitmap* que será el fondo de nuestra pizarra y un objeto *Canvas* que contendrá este *Bitmap*. Para crear el fondo de la pizarra se utiliza un *Bitmap* del mismo tamaño que la pantalla, pero como en un principio se desconoce el tamaño de la pantalla donde se va a ejecutar, se obtendrá esas dimensiones mediante el objeto *Display*. También como miembro de esta clase se usarán dos coordenadas para conocer la localización de la última pulsación y una constante para definir la tolerancia del trazo que definirá el tamaño mínimo de movimiento del dedo para detectarlo como trazo.

```
public class Board extends View{
  private Bitmap mBitmap = null;
  private Canvas mCanvas = null;
  private Path mPath = null;
  private float mX, mY;
  private static final float TOLERANCE = 4;
  private Paint mPaint = null;

  public Board(Context context, AttributeSet attrs) {
     super(context, attrs);
     init(context);
  }

  public Board(Context context) {
     super(context);
     init(context);
  }
  }
```

Se han creado dos constructores para esta clase, esto es para asegurarnos que funciona de modo correcto cuando lo insertemos en la vista. Los dos constructores llaman al método `init()` que se encargará de configurar la pantalla y el pincel a utilizar.

```
private void init(Context context){
   //obtener pantalla
   Display display = ((WindowManager) context.getSystemService(Context.
   WINDOW_SERVICE)).getDefaultDisplay();
   Point point = new Point();
   display.getSize(point);
   mBitmap = Bitmap.createBitmap(point.x,point.y, Bitmap.Config.ARGB_8888);
   mCanvas = new Canvas(mBitmap);
   mPath = new Path();
   //preparamos el pincel
   mPaint = new Paint();
   mPaint.setStyle(Paint.Style.STROKE);
   mPaint.setStrokeJoin(Paint.Join.ROUND);
   mPaint.setStrokeCap(Paint.Cap.ROUND);
   mPaint.setAntiAlias(true);
   mPaint.setDither(true);
   mPaint.setColor(0XFF00E1FF);
   mPaint.setStrokeWidth(10);
    }
```

Esta vista que se está creando debe controlar cada vez que se pulsa sobre ella, se mueve el dedo o se deja de pulsar o dicho de otro modo, cada vez que se produce un evento del tipo `MotionEvent.ACTION_DOWN`, `MotionEvent.ACTION_MOVE` o `MotionEvent.ACTION_UP`; el lugar donde hacer esto es en el método `onTouchEvent()`, por lo que se sobrescribirá con el código:

```
@Override
public boolean onTouchEvent(MotionEvent event){
   float x = event.getX();
   float y = event.getY();
   switch(event.getAction()){
      case MotionEvent.ACTION_DOWN:
         touchStart(x,y);
         invalidate();
         break;
      case MotionEvent.ACTION_MOVE:
         touchMove(x,y);
         invalidate();
         break;
      case MotionEvent.ACTION_UP:
         touchUp();
         invalidate();
         break;
   }
   return true;
}
```

Cada uno de los casos termina con una llamada al método `invalidate()` para forzar el refresco de la vista, para hacer que se ejecute el método `onDraw()`.
En cada uno de los métodos correspondientes a los eventos anteriores se debe realizar una acción distinta.
En caso de producirse un evento `ACTION_DOWN` es que se ha comenzado a tocar la pantalla, por lo que se inicializará el objeto *Path* que mantiene los trazos y se guardarán las coordenadas como punto de inicio de dicho *Path*:

```
private void touchStart(float x, float y) {
      mPath.reset();
      mPath.moveTo(x, y);
      mX = x;
      mY = y;
   }
```

En caso de producirse un evento `ACTION_MOVE` es que se está moviendo el dedo por la pantalla, por lo que se debe actualizar el *Path* y las variables que guardan las últimas coordenadas donde se ha tocado. Para no sobrecargar el procesador no siempre se actualizan las coordenadas, sino que se tiene en cuenta la tolerancia marcada por la constante `TOLERANCE`. Si el procesador es rápido, la tolerancia se puede disminuir y en caso de ser lento se puede aumentar. Para que el trazo quede más natural, lo que se hará es en lugar de tomar el *Path* como una línea recta entre los dos puntos, aplicar una curva Béizer cuadrática.

```
private void touchMove(float x, float y) {
if (Math.abs(x - mX)>= TOLERANCE ||Math.abs(y - mY)>= TOLERANCE){
   mPath.quadTo(mX, mY, (x + mX)/2,(y + mY)/2);
   mX = x;
   mY = y;
}
}
```

Por último, si se obtiene el evento `ACTION_UP` será porque el usuario ha levantado el dedo de la pantalla, por lo que se da por finalizado el trazo y se guarda en el *Bitmap* de la clase que se está creando mediante el *Canvas* que lo contiene, es como hacer un *commit* del trazo que se tiene en pantalla. Para pintar el trazo se utiliza un objeto *Paint* que aún no se ha definido y que contendrá el color y estilo de la línea:

```
private void touchUp() {
   mPath.lineTo(mX, mY);
   mCanvas.drawPath(mPath, mPaint);
   mPath.reset();
}
```

Mediante la llamada al método `invalídate()` se debería actualizar en pantalla los trazos que mantiene el objeto *Path*. Esto se hace en el método `onDraw()` que se debe sobrescribir. En este método se tiene pintar todo el *Canvas*, que en nuestro caso sería todo lo que hay en la pizarra, es decir, el fondo de la pizarra, el *Bitmap* donde estamos guardando todos los trazos ya aplicados y por último el trazo que se está realizando ahora y que se encuentra en la variable *mPath*.

```
@Override
protected void onDraw(Canvas canvas){
   //fondo
   canvas.drawColor(0XFFBBBBBB);
   // lo ya pintado
   canvas.drawBitmap(mBitmap, 0,0, null);
   //el trazo actual
   canvas.drawPath(mPath, mPaint);
}
```

Por ahora abandonaremos la clase *Board* para centrarnos en la clase principal, la *Blackboard*. En el método `onCreateView ()` es donde se establece la vista que se usará en la aplicación; se puede ver que usará el *layout* `R.layout.fragment_blackboard` y luego obtiene una referencia a un *TextView*. Dado que vamos a modificar este *layout* eliminando estos componentes, también los debemos de eliminar del código. Modificamos el método para que quede:

```
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container,
                Bundle savedInstanceState) {
   View rootView = inflater.inflate(R.layout.fragment_blackboard, container,
false);
   return rootView;
}
```

Nos dirigimos ahora al *layout* `fragment_blackboard` para que sea capaz de mostrar nuestra vista personalizada. Eliminamos todo su contenido e introducimos una vista con un atributo `class` que tenga como valor nuestra clase de *View* (la clase *Board*). El contenido del fichero debe quedar:

```
<view
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    class="com.acme.blackboard.Board"
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/ board_view"></view>
```

Ya se puede utilizar la aplicación para pintar con un lindo color azul, que si no le gusta puede cambiar sin ningún problema en el método `init()`, concretamente cuando se configura el color del pincel utilizado con `mPaint.setColor()`.

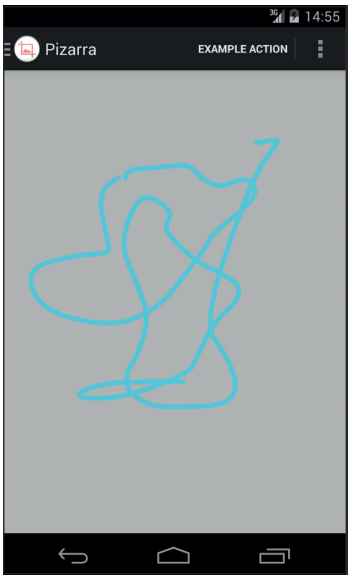

Figura 14.1. Pizarra con trazos azules.

Menu: Salvando el trabajo

La verdad es que es una lástima poder perder ciertos dibujos realizados, así que vamos a añadir una opción de guardar el trabajo sobre un fichero. Aprovecharemos el fichero de menú realizado por el asistente durante la creación del proyecto. Si abrimos el fichero `blackboard.xml` situado en `/src/main/menu`, veremos que su contenido es el siguiente:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    tools:context="com.acme.blackboard.Blackboard" >
   <item android:id="@+id/action_example"
   android:title="@string/action_example"
   app:showAsAction="withText|ifRoom" />
   <item android:id="@+id/action_settings"
   android:title="@string/action_settings"
   android:orderInCategory="100"
   app:showAsAction="never" />
</menu>
```

Sin querer entrar demasiado en qué es cada atributo (puesto que se verá más adelante), decir que el primer elemento `<item>` es el que vemos como un botón en la parte superior de la pantalla, mientras que el segundo es el que vemos si pulsamos sobre el botón de menú; se controla mediante la propiedad

app:showAsAction. Cuando se pulsa cualquiera de los dos elementos, se encarga de atenderlos el mismo método que ya conocemos onOptionsItemSelected() y será ahí donde tendremos que discernir cuál de ellos se ha pulsado, discriminando mediante su identificador. Modificamos el contenido para mostrar dos botones, uno para salvar el dibujo y otro para limpiar la pantalla:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    tools:context="com.acme.blackboard.Blackboard" >
   <item android:id="@+id/action_save"
   android:title="@string/action_save"
   android:icon="@android:drawable/ic_menu_save"
   app:showAsAction="withText|ifRoom" />
   <item android:id="@+id/action_clean"
   android:icon="@android:drawable/ic_menu_delete"
   android:title="@string/action_clean"
   app:showAsAction="withText|ifRoom" />
</menu>
```

Como se han eliminado entradas en el menú, debemos eliminar las líneas de código generadas que hacían referencias a estas entradas; concretamente debemos eliminar ciertas líneas del fichero NavigationDrawerFragment.java, en el método onOptionsItemSelected() que debe quedar:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
   if (mDrawerToggle.onOptionsItemSelected(item)) {
      return true;
   }
   return super.onOptionsItemSelected(item);
    }
```

Debemos modificar ahora el comportamiento de la aplicación cuando una de las dos entradas sea seleccionada. En la clase Blackboard cambiamos el método encargado de atender las pulsaciones de los menús, para que tenga el siguiente aspecto:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
Board board = (Board) findViewById(R.id.board_view);
switch(item.getItemId()){
   case R.id.action_save:
      Bitmap bitmap = board.getBitmap();
      String dir = Environment.getExternalStorageDirectory().toString();
      File file = new File(dir,"dibujo.png");
      OutputStream out;
      try{
         out = new FileOutputStream(file);
         bitmap.compress(Bitmap.CompressFormat.PNG, 100, out);
         out.flush();
         out.close();
```

```
            Toast.makeText(Blackboard.this, getResources().getString(R.string.
            savedOn) + dir, Toast.LENGTH_SHORT);
            } catch(Exception e){
               new AlertDialog.Builder(this).setMessage("ERROR: " +
               e.getLocalizedMessage())
                                    .setPositiveButton(android.R.string.ok, null)
                                    .show();
            }
            break;
         case R.id.action_clean:
            board.clear(Blackboard.this);
            break;
      }
      return super.onOptionsItemSelected(item);
      }
```

En caso de seleccionar la opción de salvar la imagen, se recupera el *Bitmap* que mantiene el objeto *Board* y se graba en el sistema tal y como se ha hecho en otras ocasiones. Si se selecciona la opción de limpiar, se llamará al método `clear()` del objeto *Board* (que aún no está definido).

Salvar la imagen siempre en un archivo llamado `"dibujo.png"` no es una solución elegante si es que se quiere guardar una buena pinacoteca ya que iremos sobrescribiendo el fichero de destino, pero a estas alturas al lector no le costará modificar el código para permitir al usuario seleccionar el nombre con el que guardar la imagen.

Para poder escribir en la memoria externa se necesita el permiso correspondiente en el `AndroidManifest.xml`:

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

Volvamos a la clase *Board* para implementar los dos métodos que se acaban de utilizar. Para el caso del método `getBitmap()` simplemente se retorna el *Bitmap* alojado en la clase y que contiene la pintura realizada:

```
public Bitmap getBitmap(){
   return mBitmap;
}
```

En cuanto al método `clear()`, lo que se hace es crear un nuevo *Bitmap* del modo semejante a como se hace en el constructor de la clase y se fuerza el redibujado de la vista:

```
public void clear(){
   Display display = ((WindowManager)  context.getSystemService(Context.
   WINDOW_SERVICE)).getDefaultDisplay();
   Point point = new Point();
   display.getSize(point);
   mBitmap = Bitmap.createBitmap(point.x,point.y, Bitmap.Config.ARGB_8888);
   mCanvas = new Canvas(mBitmap);
   invalidate();
}
```

Añadimos las cadenas de texto necesarias al fichero `strings.xml`:

```
<string name="action_save">Salvar</string>
<string name="action_clean">Limpiar</string>
<string name="savedOn">Se ha salvado el dibujo en </string>
```

La aplicación ya puede ser utilizada para guardar los dibujos.

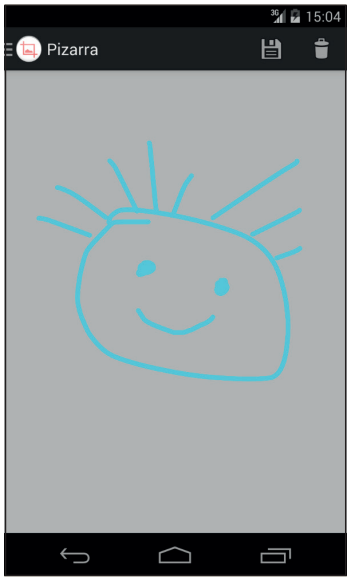

Figura 14.2. Pizarra con opciones de salvar y limpiar.

Drawer: La paleta

Como todo buen pintor seguro que necesita más color, que usar solamente el color azul se le ha quedado corto para realizar grandes obras. Vamos a añadir una paleta de colores donde el usuario pueda seleccionar el color con el que pintar. Para mostrar la paleta se va a utilizar el menú de navegación que tenemos disponible pulsando arriba a la izquierda, en el icono de la aplicación. Este elemento suele utilizarse para navegar entre pantallas de la aplicación, pero nosotros lo acomodaremos a nuestras necesidades.
Si miramos el *layout* correspondiente a ese fragmento, que es el fichero `fragment_navigation_drawer.xml`, veremos que está compuesto tan sólo por una vista de tipo `<ListView>`, así pues, en la clase Java que lo controla debe haber un adaptador de alguno de los tipos *ListAdapter*, concretamente usa un *ArrayAdapter* y lo hace en el método `onCreateView()`. Modificaremos esta clase para crear nuestro propio adaptador y poder mostrar la paleta de colores.

Lo que vamos a mostrar es una lista con una primera entrada para "la goma de borrar" y poder así retocar si nos equivocamos y luego una lista de colores; como la primera entrada es un poco distinta (no es un color) lo que haremos será utilizar dos *layouts* distintos, uno para la primera entrada y otro para el resto.
Comenzaremos creando un *layout* para la primera entrada, contendrá una imagen de una goma de borrar y un texto; lo llamaremos `first_row.xml` y como contenido tendrá:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:padding="10dip"
    android:layout_height="match_parent">

   <ImageView
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:id="@+id/imageView"
   android:layout_gravity="center_vertical"
   android:layout_marginRight="10dip"
   android:adjustViewBounds="false"
   android:src="@drawable/ic_rubber" />

   <TextView
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:textAppearance="?android:attr/textAppearanceMedium"
   android:text="Borrador"
   android:layout_gravity="center_vertical"
   android:id="@+id/textView" />

</LinearLayout>
```

Nuevamente hemos escrito el texto directamente en el *layout* por cuestión de simplicidad, pero recuerde que debe mantenerlo en el fichero `strings.xml`.
El *layout* no tiene nada nuevo, como característica especial comentar el atributo `android:textAppearance` que nos permite modificar el aspecto del texto; más adelante se verá más sobre estilos y temas.
Para el *layout* a utilizar en las líneas de los colores, mostraremos una caja con el color correspondiente y una etiqueta con su nombre; el fichero lo llamaremos `row.xml` y su contenido será:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="match_parent"
   android:orientation="horizontal"
   android:layout_height="match_parent"
   android:padding="10dip">
   <View
      android:layout_width="30dip"
      android:layout_height="30dip"
      android:id="@+id/row_bg"
      android:background="@android:color/darker_gray"
```

```
      android:layout_gravity="center"
      android:layout_marginRight="20dip"
      ></View>
   <TextView
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="sadasd"
      android:textAppearance="?android:attr/textAppearanceLarge"
      android:id="@+id/row_text"
      android:layout_gravity="center" />
</LinearLayout >
```

Para mostrar estos *layouts* en la lista, crearemos una clase propia de tipo *ArrayAdapter* y una serie de clases para apoyarnos.
La primera clase que crearemos, será la que nos ayudará a tener los datos a mostrar en la lista, es decir el nombre del color y su representación de tipo entero correspondiente. Creamos una clase interna dentro de la clase *NavigationDrawerFragment* llamada *ListEntry*:

```
 class ListEntry {
   String colorName;
   int color;

   public ListEntry (String colorName, int color){
      this.color = color;
      this.colorName = colorName;
   }
    }
```

También crearemos en *NavigationDrawerFragment* la clase *ViewHolder* que usaremos durante la presentación de información en la lista:

```
static class ViewHolder{
   TextView colorName;
   View background;
}
```

En el adaptador tenemos que tener en cuenta que la primera posición debe cargar un *layout* distinto al resto. Como usaremos la técnica del *ViewHolder*, se debe comprobar que no sólo la vista a reutilizar no es nula, sino que si la vista a reutilizar no es nula pero en su *tag* no hay un objeto, quiere decir que la vista a reutilizar es la del *layout* de la primera entrada y ésta no tiene *ViewHolder* asignado a su *tag*, por lo que habrá que crearlo.
Teniendo en cuenta lo anterior, el adaptador (que lo crearemos como clase privada dentro de la clase *NavigationDrawerFragment*) quedaría:

```
private class ColorAdapter extends ArrayAdapter<ListEntry> {
   private LayoutInflater mInflater;
   private List<ListEntry> mObjects;

   private ColorAdapter(Context context, int resource, List<ListEntry>
   objects, LayoutInflater mInflater) {
```

```
        super(context, resource, objects);
        this.mInflater = mInflater;
        this.mObjects = objects;
    }

    @Override
    public View getView(int position, View convertView, ViewGroup parent) {
        View row  = convertView;
        if (position == 0){
            row = mInflater.inflate(R.layout.first_row, null);
        }
        else{
            //si es nulo lo creamos
            if (row == null || row.getTag() == null){
                // obtención de la vista de la línea de la tabla
                row = mInflater.inflate(R.layout.row, null);
                ViewHolder holder = new ViewHolder();
                holder.background = row.findViewById(R.id.row_bg);
                holder.colorName = (TextView) row.findViewById(R.id.row_text);
                row .setTag(holder);
            }

            ListEntry listEntry = mObjects.get(position);
            //rellenamos datos
            ViewHolder holder = (ViewHolder)row.getTag();
            holder.colorName.setText( listEntry.colorName);
            holder.background.setBackgroundColor(listEntry.color);
        }
        return row ;
    }
     }
```

Antes de centrarnos en el uso del adaptador, vamos a crear una variable de tipo atributo de la clase *NavigationDrawerFragment* con tal de que mantenga los colores configurados; la haremos de tipo público para poder acceder a ella sin necesidad de generar un *getter*.

```
public ArrayList<ListEntry> mColors;
```

Para utilizar el adaptador que acabamos de crear, nos dirigimos al método `onCreateView()`, donde vimos que actualmente, se estaba utilizando un *ArrayAdapter*.

Además de crear el adaptador, crearemos los datos a utilizar en él. Como nuestro adaptador espera una lista de elementos de tipo *ListEntry*, la generaremos teniendo en cuenta que el primer elemento será utilizado para borrar, por lo que no guardaremos ningún color. El método finalmente quedaría:

```
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container,
            Bundle savedInstanceState) {
    mDrawerListView = (ListView) inflater.inflate(
    R.layout.fragment_navigation_drawer, container, false);
    mDrawerListView.setOnItemClickListener(new AdapterView.
```

```
OnItemClickListener() {
      @Override
      public void onItemClick(AdapterView<?> parent, View view, int
position, long id) {
         selectItem(position);
      }
   });
   //preparamos los colores
   mColors = new ArrayList<ListEntry>();
   mColors.add(new ListEntry("", -1));
   mColors.add(new ListEntry("Blanco", Color.WHITE));
   mColors.add(new ListEntry("Amarillo", Color.YELLOW));
   mColors.add(new ListEntry("Verde", Color.GREEN));
   mColors.add(new ListEntry("Cian", Color.CYAN));
   mColors.add(new ListEntry("Azul", Color.BLUE));
   mColors.add(new ListEntry("Magenta", Color.MAGENTA));
   mColors.add(new ListEntry("Rosa", 0xFFFBE2E2));
   mColors.add(new ListEntry("Rojo", Color.RED));
   mColors.add(new ListEntry("Gris claro", Color.LTGRAY));
   mColors.add(new ListEntry("Gris", Color.GRAY));
   mColors.add(new ListEntry("Gris oscuro", Color.DKGRAY));
   mColors.add(new ListEntry("Negro", Color.BLACK));

   //se asigna el adaptador
   mDrawerListView.setAdapter(new ColorAdapter(getActivity(), R.layout.row,
   mColors,    getActivity().getLayoutInflater()));

   mDrawerListView.setItemChecked(mCurrentSelectedPosition, true);
   return mDrawerListView;
}
```

En este caso se han definido los colores utilizando los ya existentes como constantes dentro de la clase *Color*; si nos fijamos, el rosa está añadido de modo manual usando su valor numérico en formato AARRGGBB, donde la AA es la componente alfa para la transparencia, la RR es la cantidad de rojo, el GG el es la cantidad de verde y el BB es la cantidad de azul. Respecto al canal alfa, FF significa que es opaco mientras que 00 es totalmente transparente. Para hacer un rosa semitransparente podríamos usar:

```
mColors.add(new ListEntry("Rosa transparente", 0x40FBE2E2));
```

Si ejecutamos ahora el programa, veremos que en el panel de navegación ya tenemos disponibles los colores para utilizar. El problema es que por mucho que los seleccionemos, seguimos pintando en azul e incluso desaparece el dibujo que tuviéramos realizado hasta el momento. Vamos a modificar un poco más el código para poder pintar con colores.

Según el método anterior, se ha establecido que cuando se seleccione una entrada del panel de navegación, se ejecute el método `selectItem()`, esto se ha hecho mediante la llamada a `mDrawerListView .setOnItemClickListener()`. Lo que realmente hace este método es cerrar el panel y ejecutar un

método de la clase *Blackboard*, que es la que se encarga de realizar las tareas del elemento seleccionado; este método es el `onNavigationDrawerItemSelected()`.Vamos a modificar este método, pero debemos tener en cuenta que es el método que se encarga de poner la clase *Board* en la pantalla, con lo que tendremos que mover su código a otro lado; por el momento lo modificamos para que cambie el color al pincel o configure la goma de borrar dependiendo de la posición en la que se haya pulsado:

```
@Override
public void onNavigationDrawerItemSelected(int position) {
Board board = (Board) findViewById(R.id.board_view);
if (board != null){
   if (position == 0){
      board.setEraser();
   }
   else{
      board.setPaintColor(mNavigationDrawerFragment.mColors.get(position).
color);
   }
}
 }
```

Los métodos `setEraser()` y `setPaintColor()` de la clase *Board* aún no existen, ya volveremos a ellos unas líneas más adelante. Como hemos dicho, las líneas de código que acabamos de eliminar son sumamente importantes ya que colocan el fragmento conteniendo la clase *Board*; ya que solamente usaremos el fragmento una vez, colocaremos el código en el `onCreate()` de la clase *Blackboard*, justo al final, tras la configuración del *drawer*.

```
FragmentManager fragmentManager = getSupportFragmentManager();
fragmentManager.beginTransaction()
               .replace(R.id.container, PlaceholderFragment.newInstance(0))
               .commit();
```

Vamos a completar la clase *Board* con los dos métodos que nos faltan. A la hora de codificar los pinceles hay uno de ellos que es especial y es el de borrar. Para él utilizaremos el método del pincel `setXfermode()`, que nos valdrá para indicar en la manera que se tiene que transmitir el trazo cuando se pinte.

```
 public void setEraser(){
   mPaint.setXfermode(new PorterDuffXfermode(
   PorterDuff.Mode.CLEAR));
}
```

Para establecer el pincel nos ayudamos de un método auxiliar llamado `setColor()`, en él se establece realmente el color del pincel y restaura el atributo `Xfermode` (*xfermode* significa *transfer mode*, es la manera en la que se transfiere el pincel a la hora de pintar y lo hemos usado para borrar):

```
public void setPaintColor(int color){
   mPaint.setColor(color);
   mPaint.setXfermode(null);
}
```

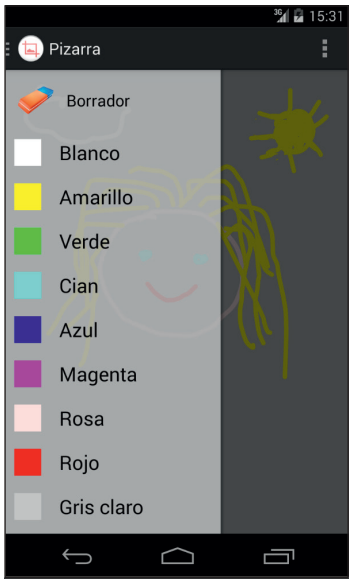

Figura 14.3. Paleta de la pizarra.

Dentro de la clase *PorterDuff*, podemos encontrar distintos modos de transferir la trazada, por ejemplo mediante `PorterDuff.Mode.LIGHTEN` se consigue una trazada más clara y con `PorterDuff.Mode.XOR` que aparezca trazo donde no lo hay y que desaparezca al tocar un trazo ya existente. Juegue con ellos para descubrir nuevos efectos (véase tabla 14.3).
Como ejercicio animo al lector a añadir distintos tamaños de pinceles; para hacer el trazo más o menos grueso se usaría la llamada:

```
mPaint.setStrokeWidth(10);
```

a la que se le pasa como parámetro el grosor deseado. Mediante algún tipo de selector puede hacer que el usuario que modifique el valor. Como ve, ya le queda poco para tener su propio programa de dibujo.

15 Widgets

En este capítulo aprenderá a:

- Crear *widgets* de escritorio.
- Parametrizar los *widgets* para que muestren una pantalla de configuración.
- Mantener múltiples instancias del mismo *widget* con distintas informaciones.
- Controlar los tiempos de refresco de los *widgets.*
- Informar al usuario mediante la barra de alertas del sistema.

A partir de la versión 1.5 de Android están disponibles unos pequeños programas que se ejecutan y muestran sobre el escritorio denominados *widgets*. Los *widgets* fueron muy mejorados en la versión 3.0 y 4.0 de Android, por lo que dependiendo de la versión, habrá funcionalidad disponible o no. Por defecto el sistema Android viene acompañado por una serie de ellos como por ejemplo un buscador, un reloj…El modo de acceder a estos *widgets* depende de la versión y el gestor de aplicaciones que se tenga instalado; suele ser manteniendo pulsado sobre el fondo de escritorio tras lo cual aparecerá un menú con varias opciones entre la que se encuentra Widgets, que al ser seleccionada nos guía hasta una lista que mostrará todos los disponibles en el dispositivo o en las versiones a partir de la 3.0 suele encontrarse en una pestaña dentro del selector de aplicaciones, aunque como hemos dicho, todo depende del gestor de aplicaciones que se tenga instalado.

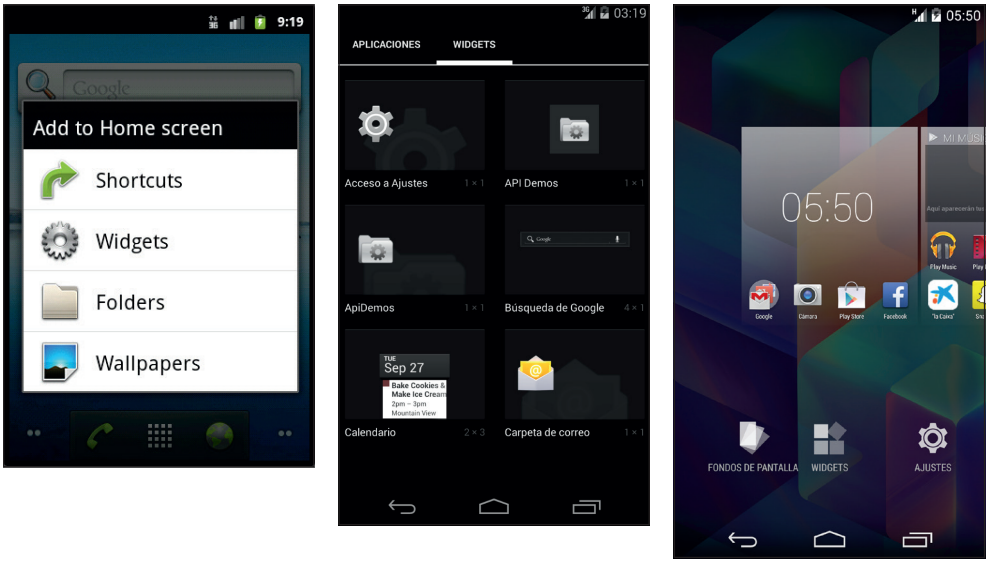

Figura 15.1. Widgets del sistema en diferentes versiones.

Al seleccionar un *widget* es posible colocarlo en cualquier lugar del escritorio siempre que no esté ya ocupado por un icono o por otro *widget* (menos a partir de la versión 4.2, que se recolocarán y escalarán para que quepa); el espacio libre necesario depende de cada uno en concreto. Una vez colocado sobre el escritorio, el *widget* puede ser movido o eliminado. Tanto para moverlo como para eliminarlo simplemente hay que mantener pulsado sobre él, y al cabo de unos instantes aparecerá como seleccionado, y si se mueve el dedo sobre la pantalla el *widget* seguirá los movimientos; allí donde levantemos el

dedo, el *widget* se colocará. Para eliminarlo sólo tenemos que llevarlo a la parte inferior o superior de la pantalla donde aparecerá una papelera (cuando el *widget* está seleccionado) y soltarlo sobre ella.
Los *widgets* permiten mostrar información o poner ciertas funcionalidades de modo que la tengan siempre al alcance, porque no olvidemos que el teléfono normalmente se encuentra mostrando la pantalla principal (la denominada *Home*) o alguno de sus escritorios. Los *widgets* que acompañan a Android suelen ser puntos de entrada para acceder a alguna *Activity* con mayor funcionalidad, por ejemplo al pulsar sobre el reloj, nos guiará hasta la aplicación de selección de alarmas o si se pulsa sobre la lista de correos no leídos, nos llevaría al programa de gestión de correo.
Hasta ahora hemos visto que las aplicaciones estaban compuestas de clases *Activity* con una de ellas como *Activity* principal que sería el punto de entrada del programa; pero no es obligatorio. Si hiciéramos una aplicación sin *Activity* principal, lo que pasaría es que no se mostraría entre las aplicaciones instaladas, (puede probarlo eliminando los filtros de la actividad en el `AndroidManifest.xml`) pero si la aplicación tiene alguna clase que implemente *AppwidgetProwider*, entonces se mostrará entre los *widgets* disponibles. Las aplicaciones pueden tener tantas *Activity* o *widgets* asociados como se quiera, pueden no tener ninguno o puede haber aplicaciones que sean sólo *widgets* sin tener ninguna *Activity* asociada.
Visto de una manera muy superficial, podemos decir que cada *widget* es un *BroadcastReceiver* (un receptor de mensajes) que está ligado a un XML que describe al propio *widget*. El *widget*, debe reaccionar a estos mensajes de *broadcast* y actualizar sus *View* para mostrar la información pertinente.

Ejemplo de widget

La mejor manera de entender qué es un *widget*, es realizar uno desde cero; en éste capítulo desarrollaremos contador de tiempo al que se le podrá poner una fecha y él determinará el tiempo que ha pasado desde ella o que falta para llegar a la fecha. Como aliciente, también descubriremos la manera de informar al usuario utilizando la barra de alertas del dispositivo, donde mostraremos un mensaje al llegar a la fecha indicada si ésta es una fecha futura. Como en el tema de *widgets* ha habido cambios a lo largo de las versiones, se irán comentando características disponibles en algunas de ellas.
Para comprender mejor la realización del *widget*, se irá haciendo por fases e introduciendo nuevas características poco a poco. La aplicación tendrá de especial que en primer lugar no se creará una actividad principal y en segun-

do que podrá haber varias instancias ejecutándose al mismo tiempo (varios contadores de tiempo), circunstancia que se ha de tener muy en cuenta. Otra circunstancia en donde se habrá de tener cuidado es el tiempo de refresco de la información. Los *widgets* se deben refrescar lo mínimo posible para evitar el desgaste innecesario de la batería (por lo que ejemplo de este capítulo, que se debe refrescar cada segundo, está bien como material didáctico, pero no sería útil en un dispositivo ya que duraría poco la batería). A la hora de definir el *widget*, se ofrece un parámetro para indicar el intervalo de tiempo que debe transcurrir entre cada actualización, pero esta actualización se realiza incluso si la pantalla se encuentra apagada, por lo que veremos otro mecanismo que no sea tan costoso en términos de eficiencia; aunque en ocasiones es bueno que se refresque si está la pantalla apagada.
Comencemos como siempre, creando un nuevo proyecto pero en este caso sin actividad principal:

- Application name: Cronometros
- Module Name: Cronometros
- Package name: com.acme.cronos
- Minimum Required SDK: API 17: Android 4.2 (Jelly Bean) o superior
- Create Activity: Desmarque la casilla `Create activity`

En el recién creado esqueleto de la aplicación, podemos ver que en la carpeta `src/main/java` no existe ningún archivo de código ya que no tenemos ninguna *Activity* creada por defecto.
Lo primero que se necesita para un *widget* es su descriptor, que nuevamente será un archivo en formato XML. Para crearlo se puede hacer de modo manual creando un fichero XML con los datos necesarios, o bien aprovechar el asistente que Android Studio nos ofrece. Para acceder al asistente, pulsamos con el botón derecho sobre el árbol del proyecto y seleccionamos el menú New>Android Component y nos mostrará el asistente para la creación del *widget*. Seleccionamos App Widget y en el segundo paso lo completamos como en la figura 15.2.
Tras la ejecución del asistente, se nos habrán creado varios archivos, veamos cada uno de ellos y su utilidad. El primero que veremos es precisamente el descriptor del *widget*; lo encontramos en `src/main/res/xml` bajo el nombre `crono_widget_info.xml` y su contenido es:

```
<?xml version="1.0" encoding="utf-8"?>
<appwidget-provider xmlns:android="http://schemas.android.com/apk/res/
android"
    android:minWidth="40dp"
    android:minHeight="40dp"
```

```
    android:updatePeriodMillis="86400000"
    android:previewImage="@drawable/example_appwidget_preview"
    android:initialLayout="@layout/crono_widget"
    android:resizeMode="horizontal|vertical"
    android:widgetCategory="home_screen|keyguard"
    android:initialKeyguardLayout="@layout/crono_widget">
</appwidget-provider>
```

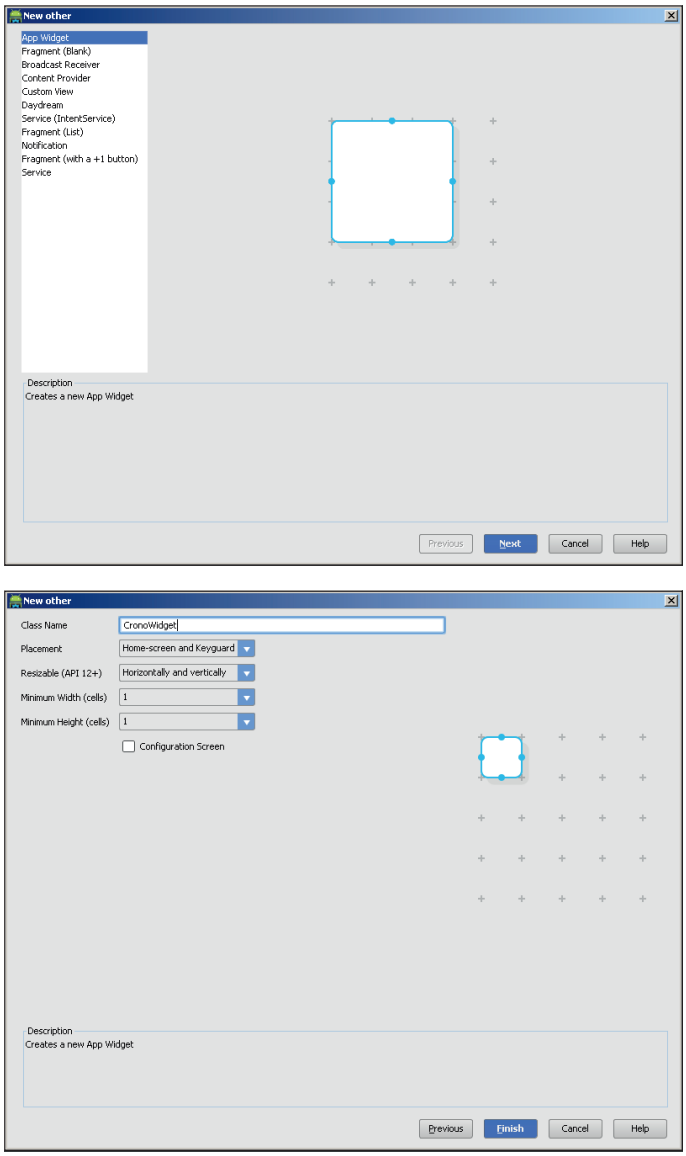

Figura 15.2. Configuración del asistente de Widgets

En este archivo podremos encontrar (dependiendo de las opciones seleccionadas en el asistente):

- minWidth maxWidth: Valores de tamaño mínimo y máximo de la anchura. Hay que tener en cuenta que a partir de Android 3.1 los *widgets* son redimensionables al colocarlos en pantalla.
- minHeight maxHeight: Valores de tamaño mínimo y máximo de la altura. Hay que tener en cuenta que a partir de Android 3.1 los *widgets* son redimensionables al colocarlos en pantalla.
- updatePeriodMillis : El intervalo de tiempo entre cada actualización en milisegundos.
- previewImage: Imagen de previsualización del *widget* para mostrar en el selector.
- initialLayout: El *layout* inicial en el *home screen*.
- resizeMode : Modo de redimensionamiento, que puede ser en ambos ejes, en ninguno o en cada uno de ellos por separado.
- widgetCategory: Tipo de *widget*, si es sólo para *home screen*, para la pantalla de bloqueo o para ambas.
- initialKeyguardLayout: El *layout* a utilizar en la pantalla de bloqueo.
- configure: La *Activity* encargada de mostrar al usuario una interfaz para gestionar la configuración del *widget*.

Muchos de estos parámetros son opcionales y otros sólo están en ciertas versiones de Android, por ejemplo, el atributo `configure` no aparece en nuestro proyecto por no haber seleccionado la casilla correspondiente en el asistente; se lo añadiremos al final a mano para ver más clara la diferencia entre informarlo o no.

Tal y como se ha configurado nuestro fichero por defecto, tendremos un *widget* de tamaño mínimo de 40x40 dips que se refrescará cada 24 horas (86400000 milisegundos), que se puede redimensionar en todos los sentidos, que puede ser utilizado tanto en el *home screen* como en la pantalla de bloqueo, que no tiene pantalla de configuración y que presentará el aspecto que tengan los *layouts* indicados.

Vamos a hacer un paréntesis para conocer mejor las medidas utilizadas, y luego proseguiremos con los ficheros generados. A la hora de definir los tamaños de los *widgets* no debe olvidar que se deberán diseñar pensando que la pantalla en ocasiones se mostrará en vertical y otras en horizontal, dependiendo de la posición del terminal; es por esto que la pantalla principal de Android, donde se colocarán los *widgets*, utiliza su propia escala de medida

llamada celda; y aquí se desvelará el secreto de las unidades `dip` que hemos usado anteriormente en alguno de los ejemplos. La pantalla principal está compuesta por una matriz de 4x4 celdas en los teléfonos y 8x7 en tabletas, sobre las cuales podremos colocar los *widget* (siempre que no estén ya ocupadas las celdas o que usemos Android 4.2 o superior). Hacer el cálculo exacto del tamaño es muy complejo, ya que depende mucho de los tamaños y densidades de pantalla, un primer cálculo lo podemos hacer diciendo que el tamaño de cada celda depende de la posición del dispositivo; cuando está en modo vertical (*portrait*), cada celda mide aproximadamente 80 x100 pixeles (ancho x alto) mientras que cuando la pantalla está en modo apaisado (*landscape*) las celdas miden 106 x 74.

No obstante para mantener la compatibilidad entre los distintos dispositivos y las diferentes densidades de pantalla (dependiendo de la calidad de la pantalla, la resolución varía, y por lo tanto el tamaño de los pixeles y su número) se utiliza como escala de medida el dip o dp (*density-independent pixel*, o pixeles independiente de densidad). Con esto lo que hace Android es escalar el pixel real al tamaño que haga falta para poderlo mostrar según el dip que se le haya indicado. Por ejemplo un icono de 100x100 pixeles cuando se muestra en un dispositivo de alta densidad, Android lo escalará a una imagen de 150x150 para que mantenga su aspecto. Esto hace que si decidimos que nuestro *widget* ocupe toda la pantalla de ancho, lo diseñamos sobre un terminal con poca densidad y si describimos su tamaño en dip en lugar de en píxeles, al llevarlo a un terminal de mayor densidad no se verá un *widget* pequeño, sino que también ocupará todo el ancho de la pantalla porque el contenido se escalará correctamente (dependiendo de los gráficos que usemos, puede llegar a verse pixelado, pero para esto están los archivos de recurso de imágenes para distintas densidades y los 9-patch que veremos más adelante).

Nota:

Recuerde que para mantener compatibilidad entre dispositivos es muy importante dentro de lo que cabe evitar todas las medidas absolutas, es decir utilizar siempre que se pueda la unidad **dp** *para indicar tamaños, usar medidas relativas para la colocación de elementos en el interfaz y por supuesto evitar el* AbsoluteLayout.

Es posible que haya creado un poco de confusión con las unidades de medida que podemos encontrarnos, así que hagamos pequeño un inciso para describirlas:

- **px:** Píxeles, son los píxeles de la pantalla, varían entre distintos dispositivos.
- **in:** Pulgadas, basado sobre el tamaño físico de la pantalla.
- **pt:** Puntos, es un 1/72 de pulgada sobre el tamaño físico de la pantalla
- **mm:** Milímetros, basado sobre el tamaño físico de la pantalla.
- **dp** o **dip:** Píxeles independientes de densidad. A la hora de indicar tamaños, el compilador de Android acepta las dos unidades dp o dip. Es una unidad abstracta basada en la densidad de imagen de la pantalla. Las densidades de pantalla se miden en dpi (no confundir con dip) que son puntos por pulgada (*dots per inch*), o dicho de otra forma, cuantos puntos se pueden mostrar linealmente en una pulgada de pantalla. Está claro que cuanto mejor sea la pantalla, mayor densidad tendrá y por consiguiente más puntos mostrará (por eso si equiparamos un punto a un pixel, en una pantalla de alta densidad todo se verá más pequeño). El dip es relativo a una pantalla de 160dpi; así, un dp es el tamaño de un pixel en una pantalla de 160 dpi.
- **sp:** Píxeles independientes de escala. Son semejantes a los dp con la diferencia de que se escalan también teniendo en cuenta las preferencias de usuario en cuanto al tamaño de fuente. Esta unidad es la que se recomienda cuando se especifican tamaños de letra, así quedará la fuente ajustada dependiendo de la densidad de pantalla y de las preferencias de usuario.

Se deberá entonces calcular el tamaño del *widget* en dip. Para un cálculo aproximado, se recomienda utilizar la fórmula :

```
Tamaño en dip = (número de celdas * 70dip) - 30dip
```

Pero a la hora de hacer el cálculo también hay que tener en cuenta los *padding* y *margin* aplicados, por lo que es preciso ser conservadores a la hora de hacer el cálculo del tamaño mínimo... y siempre recordando que se trata del tamaño mínimo del *widget*, ya que el usuario (dependiendo de la versión) posiblemente lo redimensione automáticamente.
Si se quiere trabajar con cuatro celdas de ancho por una de alto sería:

```
(4* 70)-30 = 250dp y (1* 74)-2 =40dp
```

En la definición del *widget* se le ha indicado que se tiene que utilizar el *layout* `crono_widget.xml`; si nos dirigimos al directorio `/src/main/res/layout` lo encontraremos. Su contenido es un simple *TextView*:

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="@dimen/widget_margin"
```

```
    android:background="#09C" >

    <TextView
        android:id="@+id/appwidget_text"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerHorizontal="true"
        android:layout_centerVertical="true"
        android:text="@string/appwidget_text"
        android:textColor="#ffffff"
        android:textSize="24sp"
        android:textStyle="bold|italic"
        android:layout_margin="8dp"
        android:contentDescription="@string/appwidget_text"
        android:background="#09C"/>

</RelativeLayout>
```

Truco:

Vamos a hacer otro pequeño inciso y aparcaremos el ejemplo por unos instantes para explicar un pequeño truco. Con la aparición de Android 4.0, el sistema es capaz de gestionar automáticamente los margin *exteriores de los* widgets *para acomodarlos mejor a las pantallas, por lo que se recomienda no añadir ningún margen a los* widgtes *que se vayan a ejecutar en versiones posteriores a esta versión; está claro que el desarrollador que sube una aplicación al* Market*, no sabe qué tipo de dispositivo tendrá el usuario final, pero para conseguir esto podemos hacernos uso de los directorios de recursos, y crear unas dimensiones para el margen dependiendo de la versión de Android sobre la que se muestre el widget, para ello, creamos un directorio llamado* `/src/main /res/values-v14/` *y en él un fichero con el valor:*

```
<resources><dimen name="widget_margins">0dp</dimen></resources>
```

Y en el directorio `/src/main /res/values` *creamos otro con el mismo nombre y lo informaremos con:*

```
<resources><dimen name="widget_margins">10dp</dimen></resources>
```

A partir de este momento podemos usar la dimensión creada a la hora de definir el layout*:*

```
android:layout_margin="@dimen/widget_margins"
```

Esto hará que cuando se muestre sobre un Android anterior a la API 14 se tome el valor de `/src/main /res/values` *10dip y cuando sea superior tome el de* `/src/main /res/values-v14/` *es decir 0dip, y se ajustará perfectamente. Lo mejor de todo es que el asistente ya se ha encargado de hacer esto por nosotros.*

Volvamos al ejemplo. Las actualizaciones de los *widgets* se realizan mediante mensajes *broadcast*, por lo que se tiene que atender este mensaje, para ello se deben unir los datos del *BroadcastReceiver* con el *widget* que se acaba de definir. Pero además hemos dicho que para que exista un *widget* debe existir una clase dentro del proyecto que extienda la *AppWidgetProvider*, y se debe registrar como parte del *BroadcastReceiver* para que sea capaz de gestionar las peticiones de *broadcast*. Todo esto se hace mediante la definición del *BroadcastReceiver* dentro de la etiqueta `<aplication>` en fichero `AndroidManifest.xml`, donde se le asignará un filtro para que atienda la petición de `APPWIDGET_UPDATE` y un recurso que será la descripción del *widget*. Si abrimos el fichero `AndroidManifest.xml`, veremos que el asistente ha generado el receptor por nosotros:

```
<application
android:allowBackup="true"
android:icon="@drawable/ic_launcher"
android:label="@string/app_name"
android:theme="@style/AppTheme" >
   <receiver android:name="com.acme.cronos.CronoWidget" >
      <intent-filter>
      <action android:name="android.appwidget.action.APPWIDGET_UPDATE" />
   </intent-filter>

   <meta-data
      android:name="android.appwidget.provider"
      android:resource="@xml/crono_widget_info" />
   </receiver>
</application>
```

El hecho de que estemos creando un filtro para atender a las peticiones de `UPDATE` del *widget*, hace que también atienda automáticamente a las peticiones de `DELETE`, `ENABLE` y `DISABLE`; no hace falta definir ningún filtro adicional para estos mensajes *broadcast*. Dentro de la etiqueta `<receiver>` podemos añadir el atributo `label` para definir el texto que mostrará el sistema Android en el selector de *widgets*; si no se indica un valor, se utilizará el definido en el atributo `label` de la etiqueta `<aplication>`. Si se tiene más de un *widget* en una misma aplicación, es muy recomendable informarlo, ya que de lo contrario aparecerán todos con la misma descripción.
En la etiqueta `<receiver>` se define clase que será la encargada de gestionar las peticiones de `UPDATE`, en este caso *CronoWidget* que también ha sido creada por el asistente. Su contenido:

```
public class CronoWidget extends AppWidgetProvider {

@Override
public void onUpdate(Context context, AppWidgetManager appWidgetManager,
int[] appWidgetIds) {
   // There may be multiple widgets active, so update all of them
```

```
    final int N = appWidgetIds.length;
    for (int i=0; i<N; i++) {
       updateAppWidget(context, appWidgetManager, appWidgetIds[i]);
    }
}

@Override
public void onEnabled(Context context) {
    // Enter relevant functionality for when the first widget is created
}

@Override
public void onDisabled(Context context) {
    // Enter relevant functionality for when the last widget is disabled
}

static void updateAppWidget(Context context, AppWidgetManager appWidgetManager,
            int appWidgetId) {

    CharSequence widgetText = context.getString(R.string.appwidget_text);
    // Construct the RemoteViews object
    RemoteViews views = new RemoteViews(context.getPackageName(), R.layout.
crono_widget);
    views.setTextViewText(R.id.appwidget_text, widgetText);

    // Instruct the widget manager to update the widget
    appWidgetManager.updateAppWidget(appWidgetId, views);
}
}
```

Al haber extendido la clase *AppWidgetProvider* , tenemos disponibles una serie ayudas para atender los mensajes de *broadcast* para *widgets*, pero implica que debemos implementar ciertos métodos (ya implementados por el asistente):

- `onUpdate()`:Se ejecuta cuando se ha creado el *widget* y en las llamadas periódicas que se hacen siguiendo lo indicado en el tiempo de refresco de su archivo de configuración. Se tiene como parámetro un *array* con los identificadores de los *widgets* de un ese tipo, por lo que las actualizaciones se deben realizar en un bucle, y así permitir múltiples instancias de un mismo *widget*, trabajando con distintos datos a la vez.
- `onDisabled()`: Se ejecuta cuando no queda ningún *widget* de ese tipo activo en la pantalla principal de Android. Se usa para liberar recursos comunes.
- `onEnabled()`: Se ejecuta al colocase la primera instancia del *widget* sobre la pantalla principal.

Aunque no implementados en este ejemplo, existen otros métodos proporcionados por haber extendido *AppWidgetProvider* con ayudas a la gestión de los *widgets*:

- `onDeleted()`: Se ejecuta al borrar un *widget*. Al igual que el método `onUpdate ()`, como parámetro tiene un *array* con los identificadores de los *widgets* a borrar, por lo que tiene que tratarse dentro de un bucle.

- `onReceive()`: Es el punto de entrada a la recepción de los menajes de *broadcast*. Al delegar sobre el método de su clase heredada, ésta se encarga de llamar al método correspondiente (`onDeleted()`, `onUpdate()`…). Si usamos mensajes de *broadcast* propios, aquí sería donde deberíamos insertar el código para gestionarlos. Los mensajes estándar (`UPDATE`, `DELETE`…) también podrían atenderse en este método, pero no es buena práctica hacerlo; en su lugar se deben de usar los métodos correspondientes ya vistos `onUpdate()`, `onDelete()`...

En estos momentos ya podemos probar nuestro primer *widget*. No es momento de emocionarse, puesto que simplemente mostrará una etiqueta con un valor fijo dado por el propio asistente de creación del proyecto (véase figura 15.3).

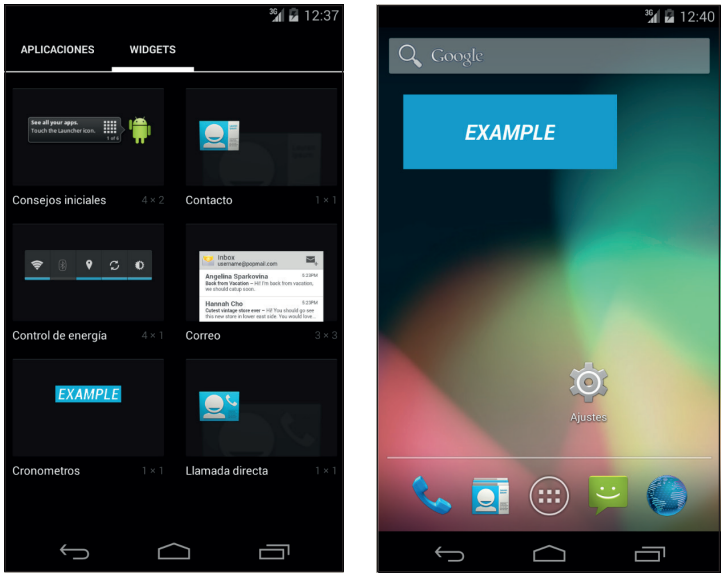

Figura 15.3. Primera versión del Widget en pantalla.

Al realizar la ejecución, es posible que salga la una ventana con una serie de opciones. Esto sucede porque en el asistente de creación del proyecto no le asignamos ninguna *Activity*. Para continuar seleccionaremos la opción Do not launch Activity y tras ello, pulsaremos en el botón Run. La ventana que nos ha salido es la configuración de ejecución de arranque que puede ser accedida en cualquier momento desde el desplegable situado en la parte superior de Android Studio, tal y como muestra la figura 15.4b.

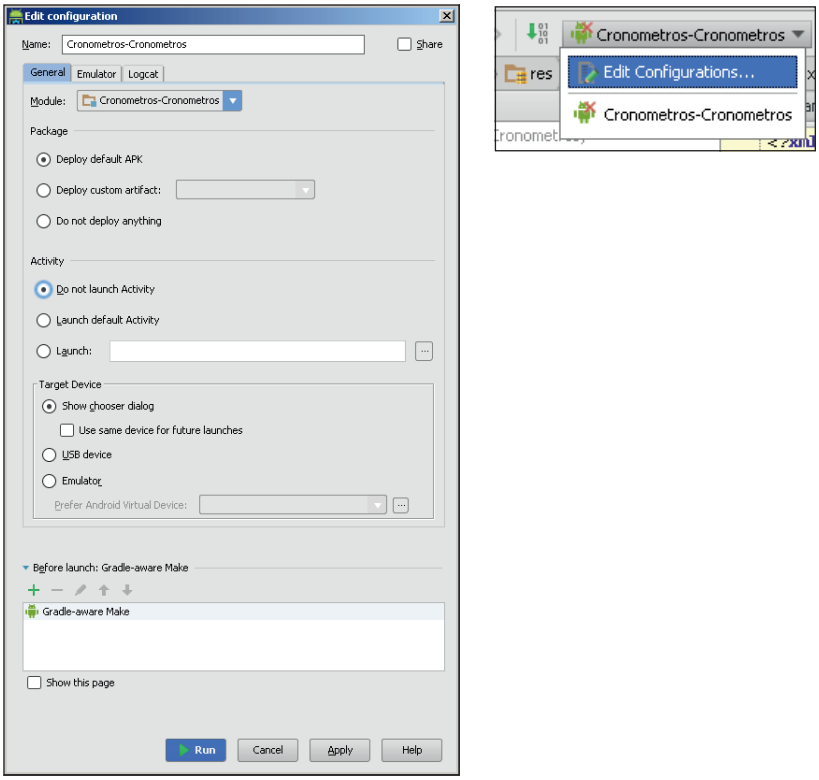

Figura 15.4. Pantalla de configuración de ejecución y acceso a ésta.

Para probarlo en nuestro dispositivo debemos ir a la zona de selección de *widgets* (que dependiendo de la versión ya hemos visto que es distinta) y seleccionarlo para colocarlo en el *home screen*. Se verá un cuadro azul con la palabra *EXAMPLE*. Veremos que es un *widget* muy pequeño y es que le hemos puesto de tamaño 40x40 dip, pero podemos redimensionarlo manteniéndolo pulsado (dependiendo de la versión de Android).
Lo cambiaremos para que sea un poco más ancho variando el archivo `crono_widget_info.xml`, cambiando:

```
android:minWidth="40dp"
por
android:minWidth="250dp"
```

Nota:

Para versiones 4.2 y superiores, también podemos colocar el widget *en la pantalla de bloqueo del terminal.*

Modificando el contenido

Lo siguiente a hacer es programarlo para que sea capaz de contar el tiempo entre dos fechas dadas y mostrar el valor en el *widget*.
Como se va a permitir tener varios cronómetros en pantalla con distintas fechas, habrá que mantener un control por separado de cada uno de ellos; Android proporciona un identificador a cada *widget* de modo que podamos usarlo para tal fin; en los métodos de la clase que extiende a *AppWidgetProvider* estos identificadores se tienen disponibles mediante parámetro, pero en el servicio que modifique el contenido del *widget*, se deben recuperar mediante los extras del *Intent* que lo lanza:

```
intent.getExtras().getInt(AppWidgetManager.EXTRA_APPWIDGET_ID);
```

Vamos a crear una clase de tipo servicio que se encargará de forzar las actualizaciones cada segundo ya que si por el *widget* en si fuera, se actualizaría cada 24 horas (tal y como lo hemos configurado en su descriptor `cronowidget_desc.xml` mediante `android:updatePeriodMillis="86400000"`). Existe un asistente para la creación de servicios en Android, pero en lugar de utilizarlo, configuraremos el servicio a mano y más adelante en otro ejercicio, realizaremos el servicio mediante el asistente. Para crear el servicio generamos la clase interna pública *CronoService* que extienda la clase *Service* dentro de la clase *CronoWidget* y sobrescribimos el método `onStartCommand()`.:

```
public static class CronoService extends Service {
@Override
public IBinder onBind(Intent arg0) {
   // TODO Auto-generated method stub
   return null;
}
@Override
public int onStartCommand(Intent intent,int flags,  int startId) {
// con este comando podemos ampliar la funcionalidad del servicio, aunque
ahora no lo haremos
   String command = intent.getAction();
   //obtenemos el identificador del widget
   int appWidgetId = intent.getExtras().getInt(
            AppWidgetManager.EXTRA_APPWIDGET_ID);
      RemoteViews remoteView = new RemoteViews(getApplicationContext()
               .getPackageName(), R.layout.crono_widget);
         AppWidgetManager appWidgetManager = AppWidgetManager
         .getInstance(getApplicationContext());
      SharedPreferences prefs = getApplicationContext().
      getSharedPreferences("crono", 0);
// fecha límite
   long goal = 0;
   if (prefs.contains("date"  + appWidgetId)){
```

```
      goal = prefs.getLong("date"  + appWidgetId, -1);
   }
   else{
      // poner fecha límite con valor de ahora
goal = System.currentTimeMillis();
         SharedPreferences.Editor edit = prefs.edit();
         edit.putLong("date" + appWidgetId, goal);
         edit.commit();
   }
   //calculamos el tiempo que falta o ha sobrepasado
   long past = System.currentTimeMillis() - goal ;
   //días
   int days = (int) Math.floor(past / (long) (60*60*24*1000));
   past = past - days • (long) (60*60*24*1000);
   //horas
   int hours = (int) Math.floor(past / (60*60*1000));
past = past - hours • (long) (60*60*1000);
    //minutos
      int mins = (int) Math.round(past / (60*1000));
   past = past - mins * (long) (60 * 1000);
   //segundos
   int secs = (int) Math.floor(past / (1000));
   if (past > 0){
      //hacer algo
   }
   //Actualizamos la vista
   remoteView.setTextViewText(R.id.appwidget_tex, days + " : " + hours
               + " : " + mins + " : " + secs );
   // aplicamos los cambios
   appWidgetManager.updateAppWidget(appWidgetId, remoteView);
   return START_STICKY;
}
}
```

Mediante la `START_STICKY`, se indica al sistema cómo se debe comportar si el servicio ha sido matado; en este caso es relanzado por el sistema. Y registramos el servicio en el AndroidManifest.xml, seguido de la definición del receiver realizada anteriormente añadiendo la línea:

```
<service android:name="CronoWidget$CronoService"/>
```

Con el $ estamos referenciando a la clase interna *CronoService* dentro de la clase *CronoWidget*. En el fichero `string.xml` añadimos una la entrada para cuando aún no se tengan datos en el *widget*.

```
<string name="sindatos">Sin Datos</string>
```

En el fichero de *layout* haremos que use esta nueva entrada de texto por defecto, modificando el texto mostrado por el *TextView*:

```
...
android:text="@string/sindatos"
...
```

Añadimos un método para crear la llamada al servicio que se disparará pasado un tiempo dado por el parámetro `updateRate`, lo añadimos a la clase *CronoWidget*.

```
public static void setAlarm(Context context, int appWidgetId, int
updateRate) {
PendingIntent newPending = makeControlPendingIntent(context,CronoWidget.
UPDATE,appWidgetId);
AlarmManager alarms = (AlarmManager) context.getSystemService(Context.ALARM_
SERVICE);
if (updateRate >= 0) {
   alarms.setRepeating(AlarmManager.ELAPSED_REALTIME, SystemClock.
elapsedRealtime(), updateRate, newPending);
} else {
// si hay u updateRate negativo, se cancela el refresco
   alarms.cancel(newPending);
}
}
```

Utilizaremos un *PendingIntent* que es un tipo especial de *Intent* que permanece en el sistema aunque el proceso creador desaparezca. Además, el *PendingIntent* hace que el componente que lo atienda herede los permisos e identidad de la aplicación que lo ha generado, implicando que el proceso lo realice como si fuera la misma aplicación generadora, es como delegar el proceso sin perder la identidad. Añada el método a la clase *CronoWidget*.

```
public static PendingIntent makeControlPendingIntent(Context context, String
command, int appWidgetId) {
Intent active = new Intent(context,CronoService.class);
   active.setAction(command);
   active.putExtra(AppWidgetManager.EXTRA_APPWIDGET_ID, appWidgetId);
   //La uri es para hacer el PendingIntent único
   //así si existen varias instancias no se sobrescribirán
   Uri data = Uri.withAppendedPath(Uri.parse("cronowidget://widget/
id/#"+command+appWidgetId), String.valueOf(appWidgetId));
   active.setData(data);
   return(PendingIntent.getService(context, 0, active, PendingIntent.FLAG_
UPDATE_CURRENT));
}
```

El servicio de actualización se inicia automáticamente cuando se instancia un *widget*, pero del mismo modo que se inicia el servicio hay que pararlo, para ello se modifica el método `onDisable()` la clase *AppWidgetProvider* de modo el servicio se detenga cuando no haya ningún *widget* activo.

```
@Override
public void onDeleted(Context context, int[] appWidgetIds) {
    for (int appWidgetId : appWidgetIds) {
           setAlarm(context, appWidgetId, -1);
    }
    super.onDeleted(context, appWidgetIds);
}
```

Cada vez que se actualice el *widget*, se debe preparar otra "alarma" para que se dispare de nuevo la actualización transcurrido un tiempo `UPDATE_RATE`.

```
@Override
public void onUpdate(Context context, AppWidgetManager appWidgetManager,
int[] appWidgetIds) {
for (int appWidgetId : appWidgetIds) {
setAlarm(context, appWidgetId, UPDATE_RATE);
   }
     super.onUpdate(context, appWidgetManager, appWidgetIds);
}
```

Por último, antes de probar de nuevo el *widget*, cree las constantes a nivel de clase que aún se necesitan: una para el tiempo de refresco y otra para el comando a pasar al *widget*.

```
private static int UPDATE_RATE = 1000; // un segundo
public static final String UPDATE = "update";
```

El *widget* ya es capaz de variar con el tiempo su contenido. Para probarlo quite primero el que tenga ya en la pantalla principal. Puede ser que en algunos segundos no vea cambio del contenido y en otros vea que se cuentan dos segundos en lugar de uno. Esto se debe a la granularidad con la que se está informando el *widget*; para minimizar el efecto puede bajar la constante `UPDATE_RATE`, aunque esto derivará en un consumo mayor de batería pero ante todo recuerde que los *widgets* no están pensados para estas tasas de refresco tan altas y que se está usando de modo académico; además, la actualización de los *widgets* no es algo prioritario para el sistema, por lo que se puede encolar la petición de actualización y hacerlo cuando tenga recursos libre. Aunque no es necesario si es buena política que ya que no se va a utilizar el parámetro `android:updatePeriodMillis` que se había establecido a 86400000 milisegundos en el archivo `cronowidget_desc.xml`, establecerlo a 0 milisegundos:

```
android:updatePeriodMillis="0"
```

En caso de que no varíe y muestre un "Sin Datos" quiere decir que no está ejecutando bien el servicio de actualización de la interfaz. Pruebe a desinstalar la versión anterior, e instalar nuevamente la versión con modificación del tiempo. Si sigue sin funcionar revise los logs del dispositivo.

Advertencia:

Los widgets *se han diseñado para mostrar información que no se actualice muy frecuentemente, ya que a mayor frecuencia de refresco, menor duración de batería; sobre todo si los datos mostrados se obtienen de Internet.*

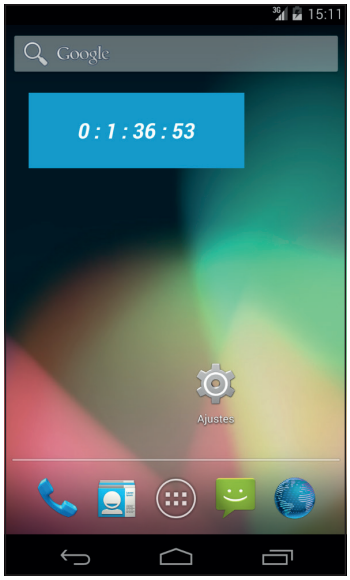

Figura 15.5. Widget contando tiempo (días : horas : minutos : segundos).

Alertas al usuario

Para indicar al usuario que se ha alcanzado una fecha en concreto vamos a utilizar la barra de notificaciones superior que tiene Android. La notificación puede ser todo lo compleja que se quiera por ejemplo utilizando estilos o *layouts* propios, iluminación de leds e incluso notificaciones desplegables. Normalmente los mensajes de aviso en la barra de alertas se emiten para no molestar al usuario o cuando la aplicación no está en primer plano, por lo que son emitidos desde servicios más que desde actividades.

Para realizar una alerta sencilla se haría mediante el servicio *NotificationManager* obtenido de la clase *Context* mediante `getSystemService(Context.NOTIFICATION_SERVICE)`. Las alertas se representan mediante una instancia de la clase *Notification* donde se configurarán los atributos del mensaje que se quiere mostrar (qué y cómo) y que será entregada más adelante a la instancia de la clase *NotificationManager* para que se encargue de mostrarla. El constructor de la clase *Notification* se llama pasándole como parámetros el icono a mostrar en el aviso, el texto que mostrará dicho aviso cuando esté comprimido en la barra y el momento en el que se debe mostrar. Más adelante a la notificación se le pueden informar muchas otras propiedades, como la actividad a mostrar cuando se pulse (a través de un *Intent*), el texto expandido, sonido a emitir cuando se reciba, configuración de leds… En nuestro caso sería algo semejante a:

```
NotificationManager notificationManager = (NotificationManager)
getSystemService(Context.NOTIFICATION_SERVICE);
Notification notification = new Notification(R.drawable.ic_
launcher,getResources().getString(R.string.alrt_tkt),goal);
Context context = getApplicationContext();
// preparar notificación
Intent notificationText = new Intent(this,CronoWidget.class);
PendingIntent contentIntent = PendingIntent.getActivity(this, 0,
notificationText, 0);
notification.setLatestEventInfo(context, getResources().getString(R.
string.alrt_titlt) + new Date(goal).toGMTString(), prefs.getString("msg"  +
appWidgetId, getResources().getString(R.string.alrt_nomsg) ), contentIntent);
notificationManager.notify(1,notification);
SharedPreferences.Editor edit = prefs.edit();
edit.putBoolean("showmsg" + appWidgetId, false);
edit.commit();
```

En nuestro caso tomaremos ventaja del asistente disponible en Android Studio para encapsular toda esta lógica y hacer mucho más fácil su uso. Con el botón derecho del ratón pulsamos sobre el árbol del proyecto para obtener un menú en el que seleccionar New>Android Component.

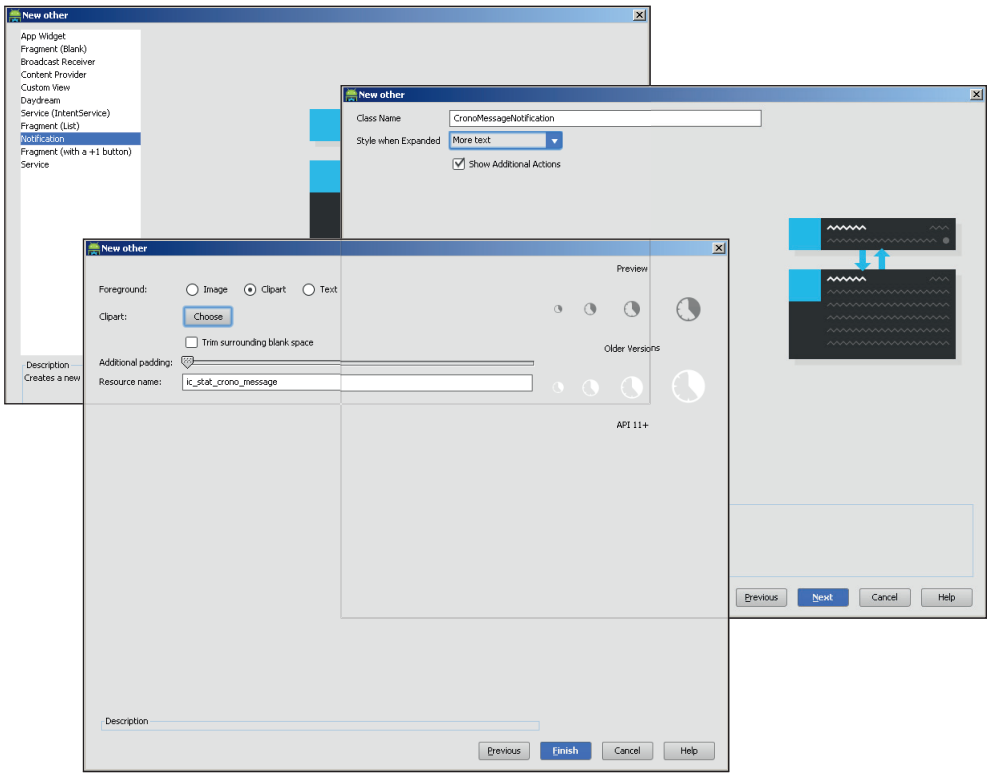

Figura 15.6. Asistente para creación de notificaciones.

Seleccionaremos la opción Notification en la primera pantalla , More Text en el desplegable Style when Expanded, como nombre de clase usaremos *CronoMessageNotification* y como icono el que más nos guste. Tras la ejecución del asistente, tendremos en el directorio del código, una nueva entrada con el fichero generado; si lo abrimos veremos que en el método `notify()` tenemos (escondido, eso sí) un código semejante al visto anteriormente para lanzar la notificación; será el método que usaremos. Modificaremos las entradas utilizadas en esta clase para la generación de los mensajes y que podemos encontrar en el fichero `strings.xml` para adecuarlas un poco mejor a nuestros cronómetros.

```
<string name="crono_message_notification_title_template">Mensaje: %1$s</string>
    <string name="crono_message_notification_placeholder_text_template">Se
ha cumplido el tiempo del cronómetro. Tu mensaje es %1$s.</string>
```

Para lanzar la notificación, en el método `onStartCommand()`, modifique:

```
if (past > 0){
   //hacer algo
}
```

por

```
// si ha pasado la fecha, mostrar el mensaje si aun no se ha hecho
if (past > 0 && prefs.getBoolean("showmsg"  + appWidgetId, true)){
   SimpleDateFormat formatter = new SimpleDateFormat("dd/MM/yy");
               CronoMessageNotification.notify(getApplicationContext(),
formatter.format(goal), 1);
}
```

En la alerta se mostrará un texto almacenado en el sistema de preferencias bajo la clave `"msg"+identificador_del_widget`, de modo que cada *widget* podrá tener el suyo y podrá ser configurable.

Si prueba el *widget* ya podrá disfrutar de ver las alertas en la barra superior, pero no tiene mucho sentido ya que el mensaje se muestra nada más iniciar el *widget*.

Mirando un poco más el código de la clase generada, en el método `notify()`, veremos que no sólo nos ha creado los textos esperados, sino que también nos ha añadido otras funcionalidades interesantes, por ejemplo que cuando se pulse nos lleve a una dirección web mediante:

```
.setContentIntent(
   PendingIntent.getActivity(
      context,0,
      new Intent(Intent.ACTION_VIEW, Uri.parse("http://www.google.com")),
      PendingIntent.FLAG_UPDATE_CURRENT))
```

Como se puede observar, no es más que una llamada mediante *Intent*, con lo que se puede aplicar todo lo aprendido hasta el momento para llamar a cualquier aplicación desde aquí. También sindica cómo añadir textos adicionales:

```
.setStyle(new NotificationCompat.BigTextStyle()
    .bigText(text)
    .setBigContentTitle(title)
    .setSummaryText("Dummy summary text"))
```

Incluso a compartir con otras aplicaciones

```
.addAction(
    R.drawable.ic_action_stat_share,
    res.getString(R.string.action_share),
    PendingIntent.getActivity(
    context,0,
    Intent.createChooser(new Intent(Intent.ACTION_SEND)
    .setType("text/plain")
    .putExtra(Intent.EXTRA_TEXT, "Dummy text"), "Dummy title"),
                PendingIntent.FLAG_UPDATE_CURRENT))
```

Además, la clase generada se encarga de lidiar con los problemas que pueden acarrearnos las distintas opciones de presentación teniendo en cuenta la versión del sistema operativo en el que se vaya a ejecutar.

Configuración del widget

Antes de colocar el *widget* sobre la pantalla se le puede dar al usuario una opción de configurarlo, por ejemplo si se quiere mostrar el valor de una acción de bolsa, se tiene que indicar que acción queremos mostrar o si es un lector de RSS, la pantalla debería dejar seleccionar la fuente RSS a mostrar.
En este ejemplo se le permitirá al usuario seleccionar la fecha referencia, la hora y un texto a mostrar en la barra de alertas cuando se haya cumplido la fecha límite.
El archivo de *layout* para la pantalla de configuración lo llamaremos `crono-config.xml` y su contenido:

```
<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="wrap_content" android:layout_height="wrap_content">
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="wrap_content" android:layout_height="wrap_content"
    android:orientation="vertical">
   <TextView android:id="@+id/TextView02" android:layout_width="wrap_content"
   android:layout_height="wrap_content" android:text="@string/cfg_date" />
   <DatePicker android:layout_width="wrap_content" android:id="@+id/datePicker"
   android:layout_height="wrap_content" />
   <TextView android:id="@+id/TextView02" android:layout_width="wrap_content"
   android:text="@string/cfg_time" android:layout_height="wrap_content" />
   <TimePicker android:layout_width="wrap_content" android:id="@+id/timePicker"
   android:layout_height="wrap_content" />
   <TextView android:id="@+id/TextView03" android:layout_width="wrap_content"
   android:text="@string/cfg_text" android:layout_height="wrap_content" />
   <EditText android:id="@+id/textAlrm" android:layout_width="fill_parent"
   android:layout_height="wrap_content"   android:background="@
```

```
   android:drawable/editbox_background"></EditText>
   <LinearLayout android:id="@+id/LinearLayout01"
   android:layout_width="wrap_content" android:layout_height="wrap_content"
   android:orientation="horizontal">
      <Button android:layout_width="wrap_content"
      android:layout_height="wrap_content" android:id="@+id/okbutton"
      android:text="OK" />
      <Button android:layout_width="wrap_content"
      android:layout_height="wrap_content" android:text="Cancel"
      android:id="@+id/cancelbutton" />
   </LinearLayout>
</LinearLayout>
</ScrollView>
```

Dé valor a las constantes de texto:

```
<string name="cfg_date">Fecha</string>
  <string name="cfg_time">Hora</string>
  <string name="cfg_text">Texto de la alarma</string>
```

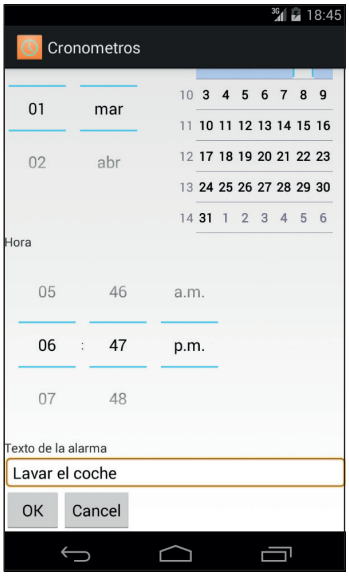

Figura 15.7. Aspecto de la pantalla de configuración.

Al haber una pantalla, detrás tendremos una actividad encargada de gestionar el funcionamiento de ella, a esta clase la llamaremos *CronoConfig* y se creará dentro del paquete `com.acme.cronos` con el siguiente código:

```
public class CronoConfig extends Activity {
private Context self = this;
private int appWidgetId;
@Override
protected void onCreate(Bundle savedInstanceState) {
```

```
        super.onCreate(savedInstanceState);
        // se obtiene el appWidgetId del widget que se va a configurar
        Intent launchIntent = getIntent();
        Bundle extras = launchIntent.getExtras();
        appWidgetId = extras.getInt(AppWidgetManager.EXTRA_APPWIDGET_ID,
                                    AppWidgetManager.INVALID_APPWIDGET_ID);
        // Se preparan los datos de lo que tiene que devolver en caso de que
        // el usuario pulse sobre el botón Cancel
        Intent cancelResultValue = new Intent();
        cancelResultValue.putExtra(AppWidgetManager.EXTRA_APPWIDGET_ID,
                                    appWidgetId);
        setResult(RESULT_CANCELED, cancelResultValue);
        // Cargamos el layout
        setContentView(R.layout.cronoconfig);
        // Obtenemos el botón OK y le asignamos su código correspondiente
        Button ok = (Button) findViewById(R.id.okbutton);
        ok.setOnClickListener(new View.OnClickListener() {
           public void onClick(View arg0) {
           // obtenemos la fecha
           DatePicker dp = (DatePicker)findViewById(R.id.datePicker);
           // obtenemos la hora
           TimePicker tp = (TimePicker) findViewById(R.id.timePicker);
           // obtenemos el texto de alarma
           EditText et = (EditText) findViewById(R.id.textAlrm);
           GregorianCalendar date = new GregorianCalendar(dp.getYear(),
           dp.getMonth(), dp.getDayOfMonth(),tp.getCurrentHour(),tp.
           getCurrentMinute());
           // Guardamos los datos referentes a cada widget que queramos configurar
           // en este caso guardamos fecha y mensaje
           SharedPreferences prefs = self.getSharedPreferences("crono", 0);
           SharedPreferences.Editor edit = prefs.edit();
           edit.putLong("date" + appWidgetId, date.getTime().getTime());
           edit.putString("msg" + appWidgetId, et.getText().toString());
           edit.commit();
           // Cambiamos el resultado que anteriormente habíamos configurado para
           cancel
           // a resultado OK
           Intent resultValue = new Intent();
                                    resultValue.putExtra(AppWidgetManager.EXTRA_
                                    APPWIDGET_ID,
                                                appWidgetId);
           setResult(RESULT_OK, resultValue);
           // finalizamos la Activity
           finish();
           }
        });
        // Botón Cancel
        Button cancel = (Button) findViewById(R.id.cancelbutton);
        cancel.setOnClickListener(new View.OnClickListener() {
           public void onClick(View arg0) {
              // finaliza la Activity
              finish();
            }
        });
}
}
```

Importante es también la recuperación de los datos de la pantalla y guardarlos en el sistema de preferencias con tal de que el *widget* pueda recuperarlos más adelante; si los datos son dependientes de cada instancia del *widget* (como en este caso), entonces se deben guardar teniendo en cuenta la identificación de cada uno de ellos, por ejemplo para el mensaje se utiliza la clave "`msg" + appWidgetId`.

Para hacer visible la pantalla de configuración antes de que se muestre el *widget* se debe modificar la descripción del propio *widget* para indicar que ahora en lugar de lanzarse directamente, tiene que lanzar una actividad que establezca los valores para su funcionamiento. La descripción se encontraba en el fichero `cronowidget_desc.xml`, al que se le añade el atributo `android:configure` con la clase encargada de la configuración:

```
<appwidget-provider xmlns:android="http://schemas.android.com/apk/res/android"
    android:minWidth="250dp"
    android:minHeight="40dp"
    android:updatePeriodMillis="86400000"
    android:previewImage="@drawable/example_appwidget_preview"
    android:initialLayout="@layout/crono_widget"
    android:resizeMode="horizontal|vertical"
    android:widgetCategory="home_screen|keyguard"
    android:initialKeyguardLayout="@layout/crono_widget"
    android:configure="com.acme.cronos.CronoConfig">
</appwidget-provider>
```

No nos debemos olvidar registrar la actividad en el `AndroidManifest.xml`, teniendo en cuenta que es una actividad para la configuración de un *widget*, por lo que tendrá su filtro correspondiente:

```
…
<activity android:name=".CronoConfig" android:label="@string/app_name">
   <intent-filter>
   <action android:name="android.appwidget.action.APPWIDGET_CONFIGURE" />
    </intent-filter>
</activity>
</application>
```

Anteriormente para probar el *widget* antes de proporcionarle la pantalla de configuración, se ponía como fecha límite el mismo instante en el que se creaba, pero ahora existe una fecha, por lo que el mensaje de alerta no se debe emitir hasta que la fecha esté configurada y sea posterior a ésta. Para ello cambie del método `onStartCommand()`, el bloque que comienza en la línea:

```
long goal = 0;
```

hasta el final, por el siguiente código:

```
long goal = prefs.getLong("date"  + appWidgetId, -1);
if (goal != -1){
      //calculamos el tiempo que falta o ha sobrepasado
      long past = System.currentTimeMillis() - goal ;
```

```
        //días
        int days = (int) Math.floor(past / (long) (60*60*24*1000));
        past = past - days * (long) (60*60*24*1000);
        //horas
        int hours = (int) Math.floor(past / (60*60*1000));
        past = past - hours * (long) (60*60*1000);
        //minutos
        int mins = (int) Math.round(past / (60•1000));
        past = past - mins * (long) (60*1000);
    //segundos
    int secs = (int) Math.floor(past / (1000));

        if (past > 0 && prefs.getBoolean("show_msg"  + appWidgetId, true)){
           SimpleDateFormat formatter = new SimpleDateFormat("dd/MM/yy");
        String message = prefs.getString("msg" + appWidgetId, formatter.
        format(goal));
        CronoMessageNotification.notify(getApplicationContext(), message, 1);
        //marcar como mostrado
        SharedPreferences.Editor edit = prefs.edit();
        edit.putBoolean("show_msg" + appWidgetId, false);
        edit.commit();
        }
    //Actualizamos la vista
        remoteView.setTextViewText(R.id.appwidget_text, days + " : " + hours
                    + " : " + mins + " : " + secs );

        // aplicamos los cambios
        appWidgetManager.updateAppWidget(appWidgetId, remoteView);
    }
    return START_STICKY;
}
super.onStart(intent, startId);
```

En la clase *CronoMessageNotification* se define la imagen a mostrar a la izquierda en la notificación. La que se pone por defecto es bastante sosa, así que podemos variarla cambiando:

```
final Bitmap picture = BitmapFactory.decodeResource(res, R.drawable.example_
picture);
```

por

```
final Bitmap picture = BitmapFactory.decodeResource(res, R.drawable.ic_
launcher);
```

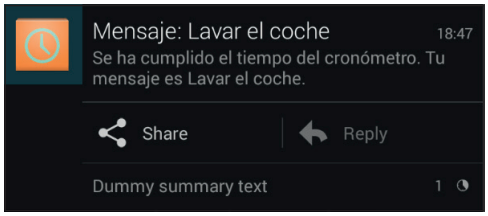

Figura 15.8. Alarma mostrada por el widget.

El contador de tiempo ya está listo para ser utilizado.
Como mejoras a añadir se podría hacer que el usuario pudiera seleccionar si los contadores de tiempo se deben detener o no al llegar a la fecha indicada, añadirle sonidos… y por supuesto, mejorar su aspecto, otro aspecto a tener en cuenta es que no se está informando la característica `android:previewImage` del archivo `crono_widget_info.xml` con una imagen "decente"; informándola con un gráfico se mostraría como previsualización del *widget* en la pantalla de selección de *widgets* en Android 3.0 y superiores.
En el diseño de los *layouts* de los *widgets* no es posible utilizar cualquier *View* ni cualquier *Layout*, en la definición de su interfaz hasta la versión 3.0 sólo puede incluirse los siguientes elementos:

- FrameLayout
- LinearLayout
- RelativeLayout
- Analog clock
- Button
- Chronometer
- ImageButton
- ImageView
- ProgressBar
- TextView

A partir de la versión 3.0 de Android es posible usar otros elementos:

- ViewFlipper
- ListView
- GridView
- StackView
- AdapterViewFlipper

16 Sensores y localización

En este capítulo aprenderá a:

- Utilizar los distintos sensores disponibles en el dispositivo.
- Controlar la información obtenida por los acelerómetros.
- Obtener la posición GPS de diversas maneras.
- Medir los campos magnéticos.

Los dispositivos Android suelen venir acompañados con una serie de sensores que hacen que el aparato pueda interactuar con el medio que lo rodea. Lejos quedan aquellos días en los que nos sorprendíamos porque un teléfono incorporara cámara, hoy los terminales traen consigo giroscopios y acelerómetros para permitir conocer su posición y movimientos, sensores magnéticos, altímetros, sensores de luz, termómetros, detectores de proximidad… y cada vez más y más precisos, abriendo un abanico muy interesante de aplicaciones por programar, porque ¿a caso alguien pensaba hace cinco años que se podría trabajar con realidad aumentada desde un móvil?

Generalidades de los sensores

Los sensores permiten al terminal tener un conocimiento del entorno que le rodea y de alguna manera abren la posibilidad de interactuar con el mismo, pero los sensores no son más que piezas electrónicas que captan información del exterior y es la aplicación la que se debe encargar de transformar dicha información en datos con los que trabajar, por ejemplo detectando que si el dispositivo se mueve hacia la derecha, el coche del juego que se programe deberá girar también hacia la derecha.

Cada sensor es un mundo y necesita distinto tratamiento y programación, y como las situaciones cambian, también deben cambiar las maneras en las que tratar los sensores, por ejemplo puede interesar que cuando quede poca batería, se baje la precisión o el tiempo de refresco de los sensores de modo que sea menos costoso energéticamente. Con la versión 4.4 KitKat, aparece una nueva manera de tratar los sensores que es mediante bloques de información que permite ahorrar más en la batería.

Las actividades que trabajan con sensores deben implementar la interfaz *SensorEventListener* que nos obligará a implementar los métodos `onAccuracyChanged()` y `onSensorChanged()`.

El método `onAccuracyChanged()` se ejecutará cada vez que cambie la precisión de un sensor mientras que `onSensorChanged()` lo hará cada vez que se produzca un cambio en alguno de los sensores.

Existen muchos sensores y depende del tipo de hardware sobre el que se ejecute la aplicación, habrá algunos que estén disponibles y otros no (siempre consultable mediante el objeto *PackageManager*), y puede ser que algunas aplicaciones necesiten ser informadas cada muy breve espacio de tiempo y otras no serán tan exigentes en cuanto a los tiempos de información; es por eso que cada actividad que implemente el interfaz *SensorEventListener* se debe registrar para escuchar a los sensores que le interese mediante el *SensorMa-*

nager, indicando el sensor que quiere escuchar y el ritmo en el que quiere ser informado. El *SensorManager* es un servicio que permite acceder a la funcionalidad de los sensores y es accesible mediante la llamada al método `getSystemService()` de la clase *Context*, pasándole como parámetro el nombre del servicio deseado: `SENSOR_SERVICE`.

Del mismo modo que una aplicación se registra para obtener información acerca de los sensores, se debe eliminar el registro cuando esta información ya no interese, por ejemplo cuando la aplicación deje de estar en primer plano o cuando se salga de ella. Lo normal es registrarlo en el método `onResume()` de la *Activity* y eliminar el registro en el método `onPause()`.

Dentro del método `onSensorChanged()` será donde se determinen los nuevos valores del sensor y por lo tanto donde se deberá actuar en consecuencia ya dependiendo de la lógica de cada aplicación. Este método tiene como parámetro un objeto del tipo *SensorEvent* que contendrá los datos del sensor que haya lanzado el evento. Como es posible que se esté registrado a más de un sensor, los datos recibidos pueden ser de cualquiera de ellos y la información será distinta (si el evento lo envía el sensor magnético los datos serán micro teslas, si lo envía el sensor luminoso serán lux) por lo que se deberá discriminar por tipo de sensor que está disponible mediante `event.sensor.getType()`.

A los datos leídos por el sensor se accede mediante `event.values`, donde `values` es un *array* de tipo *float* cuyo contenido, como ya se ha explicado, depende del sensor que los haya informado.

Acelerómetro

Quizá uno de los sensores que más llama la atención y que resulta más práctico a la hora de hacer nuestras aplicaciones más interesantes sea el acelerómetro, además, es uno de los sensores que se encuentra presente en todos los teléfonos actuales. Con este tipo de sensor es posible determinar la posición del teléfono teniendo en cuenta como eje de coordenadas el punto medio del propio terminal (véase figura 16.1).

Es decir, este sensor no nos permitirá conocer la orientación del terminal en el mundo real; si se pone el terminal en horizontal y se agarra como si fuera un volante, se pueden detectar giros la izquierda y la derecha del terminal, pero no de nuestro cuerpo, no podemos saber si se está mirando al norte o al sur (puesto que eso se trata de una translación/rotación del eje de coordenadas en el que se apoya este sensor). No obstante es muy fácil de programar y nos servirá para realizar el primer ejemplo de este capítulo, donde haremos

una bolita que se vaya moviendo según se mueva el terminal y teniendo en cuenta que no se pueda salir de los bordes de la pantalla.

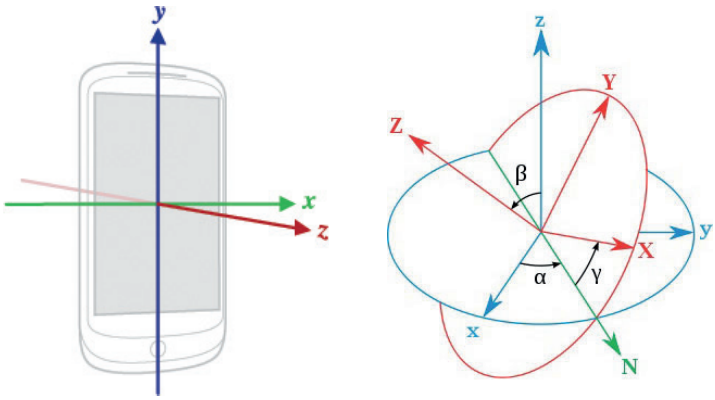

Figura 16.1. Ejes y ángulos de acelerómetros y giroscopios.

Antes de ir al ejemplo, aclarar una cuestión. Es posible que el lector tenga dudas sobre el funcionamiento de los acelerómetros y de otro de los sensores que actualmente tienen todos los móviles; los giroscopios. El acelerómetro mide la aceleración lineal en uno de los tres ejes (x, y, z); cada eje tiene su propio acelerómetro por lo que aunque pueden encontrarse acelerómetros de un solo eje o de dos ejes (con usos muy específicos de mediciones lineales), lo normal es encontrar acelerómetros de tres ejes. Los giroscopios en cambio miden velocidades angulares en α, β, γ y son los sensores más indicados para sistemas de estabilización y cambios de orientación. A diferencia de los acelerómetros, los giroscopios miden cambios; no tienen una referencia fija. Tanto los acelerómetros como los giroscopios tienen una escala de medida y están preparados para medir hasta una magnitud; no es lo mismo medir una aceleración en un cohete que una aceleración en un coche de juguete de cuerda. Con esto quiero decir, que si estamos midiendo aceleraciones y agitamos muy fuerte el móvil, nos dará lecturas fuera de rango, es decir inexactas.

Volvamos a los acelerómetros. Para no complicar los cálculos matemáticos haremos que la bola simule ser de metal, así no se tendrán que calcular rebotes. A parte del uso del sensor acelerómetro, para el dibujado de la bola se utilizará una clase propia que en lugar de extender la clase *View* que ya se ha hecho en otros ejemplos, extenderá la clase *SurfaceView*. Esta clase se utiliza cuando la actualización de la superficie de la vista es continua y necesita refrescos rápidos, esto se debe a que permite trabajar con distintos *threads* que la vayan actualizando lo que hace su uso ideal para animaciones o juegos.

Otro dato a tener en cuenta sobre la *SurfaceView* es que es opaca, por lo que se utiliza detrás de otras *View*. Es más complicada de programar y necesita muchos más recursos que una *View* normal, así que se debe utilizar sólo en caso necesario.
Cree el proyecto con las siguientes características:

- Application name: MetalBall
- Project Name: MetalBall
- Module name: com.acme.metalball
- Activity Name: MetalBall

Ocuparemos toda la pantalla con el elemento *SurfaceView*, elemento que se creará mediante programación en el evento `onCreate()` de la clase principal *MetalBall,* que a su vez debe extender la clase *Activity* e implementar la interfaz *SensorEventListener* con tal de poder reaccionar a los eventos emitidos por los sensores. En el método `onCreate()` será también donde se obtengan las referencias al acelerómetro, que es el sensor que queremos registrar con tal de obtener sus valores. Por último configuraremos la pantalla para que sea utilizada solamente en posición horizontal y eliminaremos el título de la actividad para que quede más real.
La petición para que la pantalla se muestre en horizontal, se puede realizar desde el archivo `AndroidManifest.xml` o mediante programación a través del método `setRequestedOrientation()` de la clase *Avtivity*. La configuración de la ventana mostrada por la aplicación se hace mediante el método `requestWindowFeature()` también de la clase *Activity*, donde podremos indicar una serie de características definidas en la clase *Window* tales como poner un icono, quitar el título… En este caso haremos que sea pantalla completa, aprovechando el `View.SYSTEM_UI_FLAG_IMMERSIVE` que ofrecen las nuevas versiones para poder tener cada pixel de la pantalla para la aplicación (desaparecerán todas las barras y botones de sistema)

```
public class MetalBall extends Activity implements SensorEventListener {
private  SensorManager mSensorManager;
private  Sensor mAccelerometer;

public  GroundView ground = null;
@Override
public void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
   // obtenemos referencias al servicio
   mSensorManager = (SensorManager)getSystemService(SENSOR_SERVICE);
   //interesa el acelerómetro
   mAccelerometer = mSensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
   //configuramos la pantalla
   setRequestedOrientation (ActivityInfo.SCREEN_ORIENTATION_LANDSCAPE);
```

```
        requestWindowFeature(Window.FEATURE_NO_TITLE);
        getWindow().setFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN,
        WindowManager.LayoutParams.FLAG_FULLSCREEN);
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB){
           getWindow().getDecorView().setSystemUiVisibility(
              View.SYSTEM_UI_FLAG_LAYOUT_STABLE |
              View.SYSTEM_UI_FLAG_LAYOUT_HIDE_NAVIGATION |
              View.SYSTEM_UI_FLAG_LAYOUT_FULLSCREEN |
              View.SYSTEM_UI_FLAG_HIDE_NAVIGATION |
              View.SYSTEM_UI_FLAG_FULLSCREEN |
              View.SYSTEM_UI_FLAG_IMMERSIVE);
        }
        //se indica la vista
        ground = new GroundView(this);
        setContentView(ground);
    }
}
```

Vimos que el hecho de implementar la interface *SensorEventListener* obligaba a implementar dos métodos, el de cambio de precisión en la medida (que para este ejemplo no nos interesa) y el de cambio en los valores del sensor, que es el método que debe hacer que varíe el contenido de la pantalla:

```
public void onAccuracyChanged(Sensor sensor, int accuracy) {}

public void onSensorChanged(SensorEvent event) {
ground.updateMe(event.values[1],event.values[0]);
}
```

En el evento `onSensorChanged()` se recibe un parámetro de tipo *SensorEvent* con un *array* con los valores que se han leído en el sensor, como simplemente registraremos uno, no hace falta discriminar el tipo, siempre será ese "uno", en otro ejemplo ya se verá como discriminar. El método se encarga de llamar al método del objeto `ground` llamado `updateMe()`. Este objeto será una instancia de una clase que aún no hemos realizado y que extenderá la *SurfaceView* (la encargada de mostrar algo por pantalla) y el método llamado es para indicar las nuevas coordenadas que tendrá la bola dibujada. Al llamarse el método `onSensorChanged()` con un evento del tipo acelerómetro, los valores del *array* `values` son :

- values[0]: Aceleración en el eje X.
- values[1]: Aceleración en el eje Y.
- values[2]: Aceleración en el eje Z.

Pero se tiene que tener en cuenta que hemos obligado a usar el teléfono en posición horizontal, por lo que los valores de X e Y cuando se llama al método se intercambian. La lectura realizada ya nos es válida para el ejemplo, pero si realmente se quiere obtener la aceleración real, se debe restar la acción del

campo magnético terrestre, por lo que habría que registrar el sensor para campos magnéticos y tener en cuenta también su lectura. En un ejemplo posterior se verá cómo hacerlo.
La llamada al método indicando que han cambiado los valores de cierto sensor, sólo se produce si se ha registrado ese cierto sensor mediante el *SensorManager*. El registro lo realizamos en el método `onResume()` y lo eliminamos del registro en `onPause()` .

```
protected void onResume() {
        super.onResume();
        mSensorManager.registerListener(this, mAccelerometer,
        SensorManager.SENSOR_DELAY_GAME);
}
protected void onPause() {
   super.onPause();
   mSensorManager.unregisterListener(this);
}
```

Cuando se registra el sensor se debe indicar el periodo cada cuanto debe obtener los datos. Esta frecuencia depende mucho del tipo de dato leído (hay sensores que varían más que otros) y la naturaleza de la aplicación. En este caso como queremos que el movimiento de la bola por la pantalla sea fluido seleccionamos una frecuencia de muestreo `SensorManager.SENSOR_DELAY_GAME` que es la segunda más alta.

Advertencia:

No olvide eliminar el registro de un sensor cuando no se vaya a utilizar, así el dispositivo no gastará más recursos de los necesarios, incrementando la duración de la batería.

SurfaceView

A la hora de programar el uso de un *SurfaceView*, lo más diferenciador respecto a la típica *View* es que es necesario el uso de un *Thread* para la actualización de los contenido, lo que hará que en caso de tener múltiples actualizaciones, podamos utilizar procesos distintos que conseguirán un redibujado mucho más fluido. Todos los métodos de la clase *SurfaceView* y de *SurfaceHolder.callback* serán llamados desde él.
Cree la clase pública interna *GroundView* dentro de la clase *MetalBall*.

```
public class GroundView extends SurfaceView implements SurfaceHolder.
Callback{}
```

pulse con el botón derecho en el nombre dentro del código de la clase recién creada y en el menú seleccione la opción Generate>Override Methods para implementar los métodos que son necesarios (`surfaceDestroyed()`, `surfaceChanged ()` y `surfaceCreated()`).
Dentro de esta clase recién creada, mantendremos algunos atributos para guardar características como las coordenadas donde se debe dibujar la bola, el último incremento sufrido por la posición de la misma (una manera vulgar pero efectiva de emular la gravedad), el tamaño de la pantalla y de la imagen de la bola para detectar los choques y un par de referencias más para el *thread* y el icono de la bola.

```
// coordenadas de la bola
private float cx = 10;
private float cy = 10;
// ultimo incremento de posición
private float lastGX = 0;
private float lastGY = 0;
// tamaño del grafico de la bola
private int pictureHeight =0;
private int pictureWidth =0;
private Bitmap icon = null;
// tamano de pantalla
private int width = 0;
private int height = 0;
//está tocando el borde?
private boolean noBorderX = false;
private boolean noBorderY = false;
private Vibrator vibratorService= null;
private DrawThread thread;
```

En el constructor de la clase *GroundView* se encarga de asignarse a sí misma como *callback* de las llamadas al objeto *SurfaceView* de modo que pueda tener el control sobre el dibujo en la superficie; además recopilará alguno de los valores de los atributos de la clase definida anteriormente.

```
public GroundView(Context context) {
   super(context);
   //asignamos el callback a la superficie subyacente del SurfaceView
   getHolder().addCallback(this);
   //se crea en thread
   thread = new DrawThread(getHolder(), this);
   //Obtenemos referencias y tamaños de objetos a dibujar
   Display display = ((WindowManager) getSystemService(Context.WINDOW_
SERVICE)).getDefaultDisplay();
   width = display.getWidth();
   height = display.getHeight();
   icon = BitmapFactory.decodeResource(getResources(), R.drawable.ball);
   pictureHeight = icon.getHeight();
   pictureWidth = icon.getWidth();
   vibratorService = (Vibrator)(getSystemService(Service.VIBRATOR_SERVICE)) ;
}
```

Una vez se cree la superficie sobre la que pintar dentro de el *SurfaceView*, se debe iniciar el *thread* que se encargará de dibujar sobre ella, en caso de hacerlo antes obtendríamos un error y la aplicación quedaría inconsistente. Para evitar que esto suceda lo haremos en el método `surfaceCreated()`, que se llamará una vez creada la superficie. Dentro de *GroundView* añadimos:

```
@Override
public void surfaceCreated(SurfaceHolder holder) {
   thread.setRunning(true);
   thread.start();
}
```

El método `onDraw()` que se encargará de pintar la bola en las coordenadas en las que se encuentre en cada momento es muy sencillo, sólo debe pintar el fondo y la bola, sin preocuparse de nada más.

```
@Override
protected void onDraw(Canvas canvas) {
   if (canvas != null){
   canvas.drawColor(0xFFAAAAAA);
   canvas.drawBitmap(icon, cx, cy, null);
   }
}
```

Por último, en la clase *GroundView* queda actualizar la posición de la bola y detectar si está tocando una pared (borde de la pantalla) o no. A este método se le daba como parámetros los valores de los acelerómetros para las coordenadas x e y, por lo que se tendrán que utilizar estos valores para calcular la nueva posición de la bola teniendo en cuenta la pseudo aceleración que le queremos imprimir y sabiendo que no se puede salir de la pantalla. Para dar más realismo, vamos a añadirle una pequeña vibración cuando choque con los bordes.

```
public void updateMe(float inx, float iny) {
   //actualiza aceleración
   lastGX += inx;
      lastGY += iny;
   //actualiza posicion
   cx += (lastGX);
   cy += (lastGY);

   // comprobar bordes
   // si choca vibrar y poner aceleración a 0
   // no dejar que sobrepase los bordes de la pantalla
   if (cx > (width-pictureWidth)){
      cx = width-pictureWidth;
      lastGX =0;
      if (noBorderX){
               vibratorService.vibrate(100);
               noBorderX = false;
            }
   }
   else
```

```
            if (cx < (0)){
            cx = 0;
            lastGX =0;
            if (noBorderX){
               vibratorService.vibrate(100);
               noBorderX = false;
            }
         }
         else{
               noBorderX = true;
         }
         if (cy > (height-pictureHeight)){
            cy = height-pictureHeight;
            lastGY =0;
            if (noBorderY){
               vibratorService.vibrate(100);
               noBorderY = false;
            }
         }
         else
         if (cy < (0)){
            cy = 0;
            lastGY =0;
            if (noBorderY){
               vibratorService.vibrate(100);
               noBorderY = false;
            }
         }
         else{
            noBorderY = true;
         }

         //forzar onDraw invalidando la superficie
         invalidate();
      }
```

Y no olvide añadir el permiso para vibrar al `AndroidManifest.xml`.

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

Para finalizar con la aplicación quedaría dar código al *thread* que se encargará de bloquear la superficie para que ningún otro *thread* pueda utilizarla, dibujar sobre ella y desbloquearla de nuevo para que otros *threads* puedan seguir redibujando

```
class DrawThread extends Thread {
   private SurfaceHolder surfaceHolder;
   private GroundView panel;
   private boolean run = false;

   public DrawThread(SurfaceHolder surfaceHolder, GroundView panel) {
      this.surfaceHolder = surfaceHolder;
      this.panel = panel;
   }

   public void setRunning(boolean run) {
```

```
        this.run = run;
    }

    @Override
    public void run() {
        Canvas c;
        while (run) {
            c = null;
            try {
                c = surfaceHolder.lockCanvas(null);
                synchronized (surfaceHolder) {
                    panel.onDraw(c);
                }
            } finally {
                // hacer esto en un bloque a finally
                // Si ocurre un error, no se deja el SurfaceView
                // inconsistente
                if (c != null) {
                    surfaceHolder.unlockCanvasAndPost(c);
                }
            }
                    }
    }
}
```

Para probar la aplicación es totalmente recomendable hacerlo sobre un dispositivo físico, ya que en el emulador no se podrá observar de modo correcto como varía la posición de la bola con respecto a la posición del terminal. Aunque en este caso el redibujado de la pantalla funciona de modo correcto si las exigencias son mucho mayores (por ejemplo haciendo un juego de disparos se tendrían que dibujar distintos disparos por diferentes partes de la pantalla), lo que se debería hacer es no redibujar toda la pantalla, sino solo aquella parte de la pantalla que ha sido modificada, utilizando también el método `invalidate()` pero dándole como parámetro el rectángulo que debe volver a redibujar.

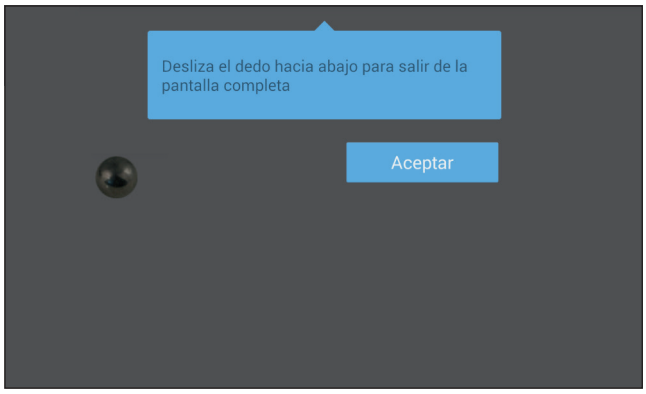

Figura 16.2. MetalBall en funcionamiento.

Por cierto, se había comentado que a la hora de registrar el sensor, se le podía indicar el tiempo de refresco. Habíamos utilizado el `valor SensorManager.SENSOR_DELAY_GAME`, pruebe a cambiarlo por `SENSOR_DELAY_FASTEST` o por `SENSOR_DELAY_NORMAL` por ejemplo. ¿Nota la diferencia?

Posición

La mayor parte de los terminales con Android disponen de un sistema de GPS incorporado para obtener el posicionamiento global del dispositivo, pero incluso en caso de no tenerlo o estar desactivado, podemos valernos de otros métodos de obtención de localización. Las aplicaciones de ello son múltiples: la navegación por rutas, el uso de redes sociales localizadas, geolocalización de fotografías, realidad aumentada… Del mismo modo que mediante la clase *SensorManager* se accedía a las funcionalidades de los sensores, la clase *LocationManager* nos abre las puertas de todas las funcionalidades de la localización. También a semejanza del acelerómetro, para trabajar con la localización se tiene que implementar la interfaz *LocationListener*, que obliga a implementar cuatro métodos:

- onLocationChanged(): Se disparará cada vez que se produzca una modificación de la posición.
- onStatusChanged(): Cuando el proveedor de la localización cambia su estatus, por ejemplo deja de estar disponible.
- onProviderEnabled(): Cuando se habilita el proveedor.
- onProviderDisabled(): Cuando se deshabilita el proveedor.

Como los dispositivos Android suelen ser portátiles, las baterías no son uno de sus fuertes, así que hay que cuidarlas y el GPS no ayuda; también se puede dar el caso que estemos trabajando con una aplicación que necesite usar la localización y que al entrar en un edificio se pierda la cobertura GPS, en este caso debemos dar al usuario una localización alternativa para que pueda seguir utilizando la aplicación.
LocationManager es algo más que un simple proveedor de los servicios de GPS, dispone de una serie de métodos que permiten, por ejemplo, obtener la lista de todos los proveedores de localización disponibles en ese momento para el dispositivo (puede haber incluso proveedores generados por programación) mediante el método `getAllProviders()`, obtener el proveedor de localización que más se ajuste a las necesidades de la aplicación mediante `getBestProvider()`, dándole como parámetros el criterio a utilizar para

seleccionarlo o un método muy interesante que lanza un *Intent* cuando se acerca a una localización indicada `addProximityAlert()`.
Para entender mejor el funcionamiento de los servicios de localización y afianzar el de los sensores haremos una aplicación que mostrará en pantalla la localización y posición global del teléfono, no sólo su longitud y latitud, sino también su orientación y giros respecto a sus ejes. Crearemos un nuevo proyecto Android con los valores:

- Application name: Posicioname
- Project Name: Posicioname
- Module name: com.acme.position
- Activity Name: Position

La pantalla utilizada será una lista de etiquetas donde se mostrará la información captada para la localización exacta del dispositivo, modificaremos el *layout* creado con dichas etiquetas:

```
<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:background="#6500ff"
   android:textColor="#ffffff"
   android:text="Acelerómetro" />
   <TextView
   android:id="@+id/xbox"
   android:layout_width="fill_parent"
   android:layout_height="wrap_content"
   android:text="Valor X " />
   <TextView
   android:id="@+id/ybox"
   android:layout_width="fill_parent"
   android:layout_height="wrap_content"
   android:text="Valor Y" />
   <TextView
   android:id="@+id/zbox"
   android:layout_width="fill_parent"
   android:layout_height="wrap_content"
   android:text="Valor Z" />
   <TextView
   android:layout_width="fill_parent"
   android:layout_height="wrap_content"
   android:background="#6500ff"
   android:textColor="#ffffff"
   android:text="Orientación" />
   <TextView
   android:id="@+id/xboxo"
```

```
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Azimuth" />
    <TextView
    android:id="@+id/yboxo"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Pitch" />
    <TextView
    android:id="@+id/zboxo"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Roll" />
    <TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:background="#6500ff"
    android:textColor="#ffffff"
    android:text="Localización: " />
    <TextView
    android:id="@+id/gpsbox"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    />
    <TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:background="#6500ff"
    android:textColor="#ffffff"
    android:text="Campos magnéticos:" />
    <TextView   android:id="@+id/magneticbox"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content" />
</LinearLayout>
```

En la clase principal implementaremos las interfaces *SensorEventListener* y *LocationListener* de modo que podamos leer tanto los acelerómetros como la localización. Al implementar estas interfaces nos obliga a sobrescribir los métodos:

- onLocationChanged() (por *LocationListener*)
- onProviderDisabled() (por *LocationListener*)
- onProviderEnabled() (por *LocationListener*)
- onStatusChanged() (por *LocationListener*)
- onAccuracyChanged() (por *SensorEventListener*)
- onSensorChanged() (por *SensorEventListener*)

Dentro del la clase crearemos unas variables donde se guardarán las referencias a los sensores, a los elementos *TextView* de la interfaz donde escribiremos los resultados y una serie de *arrays* que utilizaremos para los cálculos de posición.

```
public class Position extends Activity implements SensorEventListener,
LocationListener {
   // sensores y localizacion
   private  SensorManager sensorManager = null;
   private  LocationManager locationManager = null;
   private  Sensor accelerometer = null;
   private  Sensor orientation = null;
   private  Sensor magnetic = null;
      // textos de pantalla
   private TextView xViewA = null;
   private TextView yViewA = null;
   private TextView zViewA = null;
   private TextView xViewO = null;
   private TextView yViewO = null;
   private TextView zViewO = null;
   private TextView gpsLoc = null;
   private TextView magneticView = null;
      // para el calculo de posicion
   private static final int matrix_size = 16;
   private float[] Rm = new float[matrix_size];
   private float[] outR = new float[matrix_size];
   private float[] I = new float[matrix_size];
   private float[] values = new float[3];
   private float[] mags = new float[3];
   private float[] accels = new float[3];
```

En el método `onCreate()` haremos la petición de los servicios para los sensores y para la localización, pero por ahora nos centraremos en los sensores. Esta vez queremos tener más información sobre la posición en la que se encuentra el terminal, por lo que además de leer los acelerómetros leeremos también el campo magnético recibido por el sensor correspondiente. Aunque no las vayamos a usar aún, aprovecharemos para recupera la referencia a todas las etiquetas de texto que rellenaremos durante el proyecto:

```
public void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   setContentView(R.layout.activity_position);
   //obtención de los sensores
   sensorManager = (SensorManager)getSystemService(SENSOR_SERVICE);
   accelerometer = sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
   magnetic= sensorManager.getDefaultSensor(Sensor.TYPE_MAGNETIC_FIELD);

   //referencias a los textos de pantalla
   xViewA = (TextView) findViewById(R.id.xbox);
   yViewA = (TextView) findViewById(R.id.ybox);
   zViewA = (TextView) findViewById(R.id.zbox);
   xViewO = (TextView) findViewById(R.id.xboxo);
   yViewO = (TextView) findViewById(R.id.yboxo);
   zViewO = (TextView) findViewById(R.id.zboxo);
   gpsLoc = (TextView)findViewById(R.id.gpsbox);
}
```

Del mismo modo que se hizo en el ejemplo anterior, registre los dos sensores para poder leer su resultado y no olvide de eliminar el registro cuando no se vayan a utilizar más. Usaremos de nuevo los métodos `onResume()` y `onPause()` para estas acciones:

```
@Override
protected void onResume() {
   super.onResume();
   sensorManager.registerListener(this, accelerometer, SensorManager.SENSOR_
   DELAY_NORMAL);
   sensorManager.registerListener(this, magnetic, SensorManager.SENSOR_
   DELAY_NORMAL);
}
@Override
protected void onPause() {
   super.onPause();
   sensorManager.unregisterListener(this);
}
```

El método donde se recibirán las lecturas de los sensores (el `onSensorChanged()`) en este caso debe tener en cuenta que va a recibir valores de dos sensores distintos y tiene que discriminarlos con tal de realizar una lectura válida del dato obtenido. La discriminación se realiza a través del tipo de sensor:

```
switch (event.sensor.getType()){}
```

Se han registrado dos sensores, el acelerómetro y el medidor de campo magnético, para el cálculo de la posición exacta del teléfono se tendrán en cuenta los valores de los acelerómetros y se corregirán con las mediciones del campo magnético con tal de conocer la posición exacta. No entraremos en detalles sobre el cálculo de la matriz de rotación puesto que escapa al alcance de este libro. El método quedaría:

```
@Override
public void onSensorChanged(SensorEvent sensorEvent) {
   switch (sensorEvent.sensor.getType()) {
      case Sensor.TYPE_MAGNETIC_FIELD:
         mags = sensorEvent.values.clone();
         break;
      case Sensor.TYPE_ACCELEROMETER:
         accels = sensorEvent.values.clone();
         xViewA.setText("Aceleración X: " + sensorEvent.values[0]);
      yViewA.setText("Aceleración Y: " + sensorEvent.values[1]);
         zViewA.setText("Aceleración Z: " + sensorEvent.values[2]);
         break;
       }
       if (mags != null && accels != null) {
          SensorManager.getRotationMatrix(Rm, I, accels, mags);
       //   Correción de la pantalla si esta en horizontal
          SensorManager.remapCoordinateSystem(Rm,
```

```
                SensorManager.AXIS_X,
                SensorManager.AXIS_Z, outR);
            SensorManager.getOrientation(outR, values);
            xViewO.setText("Orientación X: " +  values[0]);
            yViewO.setText("Orientación Y: " +  values[1]);
            zViewO.setText("Orientación Z: " +  values[2]);
        }
}
```

Truco:

Los valores se muestran en radianes, si se desea obtener en grados se tienen que transformar mediante por ejemplo `Convert.radToDeg(values[0])`.

Ya no hay más que hacer en cuanto a los sensores, la aplicación ahora muestra su posición respecto a sus ejes y su orientación. De aquí a tener nuestra propia brújula es simplemente interpretar los valores obtenidos y tener un dibujo de una brújula al que se le apliquen rotaciones teniendo en cuenta dicho valor. ¿Se anima?

Localización

Vamos a encargarnos ahora de la localización. En un principio se intentará utilizar el GPS (no se comprobará si existe o no, pero sería buena práctica hacerlo; aquí simplemente se supone que se tiene y se intenta utilizar) y si no está disponible se procederá a utilizar otro tipo de localización. La recuperación del servicio y el registro para la obtención de la localización, lo realizaremos en el evento `onCreate()`.

Cuando se selecciona un proveedor de localización para que informe a la aplicación se realiza mediante el método `requestLocationUpdates()`, que puede ser llamado con distintos tipos de parámetros para configurar no sólo el proveedor que queremos que nos informe, sino también las condiciones en las que lo debe hacer. En nuestro caso indicaremos que nos informe cada 30 segundos o cada vez que la posición varíe en más de 20 metros, añada al método `onCreate()`:

```
// localizacion
locationManager = (LocationManager) getSystemService (Context.LOCATION_SERVICE);
if ( locationManager == null ) {
   Toast.makeText(this, "Error al recuperar el LocationManager", Toast.
LENGTH_LONG).show();
}
```

```
try{
   // recepción de datos del GPS
   locationManager.requestLocationUpdates(LocationManager.GPS_PROVIDER,
   30000,20,   (LocationListener)this);
} catch (Exception e){
   Toast.makeText(this, "Error al inicializar el GPS", Toast.LENGTH_LONG).
   show();
}
```

Cada vez que se produzca o bien una variación de posición mayor de la marcada o que haya pasado más tiempo del indicado, nuestra aplicación será informada en el método `onLocationChanged()`. Mostraremos la información recibida junto con el proveedor que la proporciona.

```
@Override
public void onLocationChanged(Location location) {
String lat = String.valueOf(location.getLatitude());
String lon = String.valueOf(location.getLongitude());
gpsLoc.setText(location.getProvider() + ": lat="+lat+"  lon="+lon);
}
```

Para detectar si el GPS está deshabilitado se utiliza el método `onProviderDisabled()`. En él se informará al usuario de que el proveedor está desactivado y procederemos a utilizar otro proveedor distinto; en este caso se usará el de red, que no es tan preciso, pero funciona en situaciones más desfavorables que el GPS, por ejemplo dentro de un edificio.

```
@Override
public void onProviderDisabled(String provider) {
Toast.makeText(this, "Provider deshabilitado", Toast.LENGTH_LONG).show();
if ("GPS".equalsIgnoreCase(provider)){
   locationManager.requestLocationUpdates(LocationManager.NETWORK_PROVIDER,
30000,20, (LocationListener)this);
        }
}
```

Del mismo modo que informamos cuando se desactiva el proveedor, informaremos cuando vuelve a activarse:

```
@Override
public void onProviderEnabled(String provider) {
   Toast.makeText(this, "Provider habilitado", Toast.LENGTH_LONG).show();
}
```

Para acabar con la programación de la aplicación, queda otorgarle permisos para acceder a la posición del dispositivo. Como se va a utilizar tanto el GPS como la posición por red, hay que obtener dos permisos, el de posición "fina" y el de posición "aproximada":

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

Si ejecuta la aplicación podrá ver como se informa la longitud y latitud de su posición. Puede probar a activar y desactivar el GPS y (tras unos instantes ya que necesita conectar con el satélite) verá como la etiqueta referente a la latitud y longitud varía su información junto con la descripción del proveedor de los datos. Ahora ya puede programar una brújula con posicionamiento GPS… una herramienta imprescindible si practica senderismo.

Campos magnéticos

Ya que obtenemos los valores de los campos magnéticos para realizar la corrección del posicionamiento del dispositivo, vamos a aprovechar para mostrarlos por pantalla.
Añada al final del método `onCreate()`

```
//magnético
magneticView = (TextView) findViewById(R.id.magneticbox);
```

Y en el método `onSensorChanged()` modifique el caso `Sensor.TYPE_MAGNETIC_FIELD` par mostrar los valores por pantalla.

```
case Sensor.TYPE_MAGNETIC_FIELD:
   mags = sensorEvent.values.clone();
   magneticView.setText(String.format("x: %f\ny: %f\nz: %f", new Object[]
{sensorEvent.values[0], sensorEvent.values[1], sensorEvent.values[2]}));
   break;
```

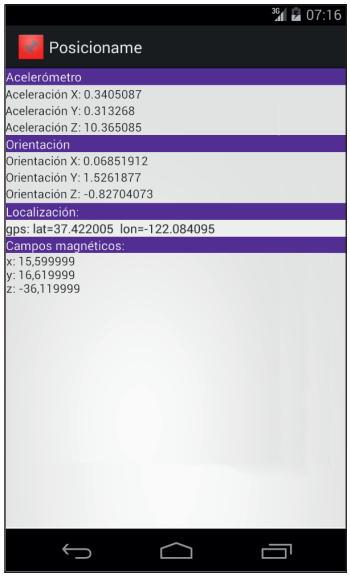

Figura 16.3. Aplicación mostrando los valores obtenidos de los sensores.





17 Multitouch y gestos

En este capítulo aprenderá a:

- Entender los eventos generados cuando se tienen pulsaciones simultaneas en pantalla.
- Controlar la posición de cada uno de los punteros.
- Cómo controlar los gestos en pantalla.

Entendemos por *multitouch* cuando el usuario apoya dos o más dedos sobre la pantalla en lugar de realizar las pulsaciones con uno. Pese a que Android fue concebido para trabajar tanto con dispositivos táctiles como no táctiles, los segundos son los que han inundado el mercado, tanto es así que incluso el primer dispositivo con este sistema fue precisamente táctil; esto puede llevar a pensar que Android es un sistema operativo para dispositivos táctiles y no es así, de hecho ofrece mecanismos para, mediante programación, conocer el hardware que tiene el usuario y en función de él hacer cosas como mostrar una interfaz de aplicación u otra.

Cuando entramos en el terreno del *multitouch*, las cosas cambian. En un principio Google tuvo que lidiar con unas patentes de cierta empresa de fabricante de móviles y lo dejó un poco apartado, por lo que en las primeras versiones Android el *multitouch* no era viable. A partir de la versión 2.0, Android permite realizar algunas acciones como la denominada *pinch and zoom* (pellizcar el dispositivo).

El hecho de que tenga soporte táctil no quiere decir que pueda soportar más de un dedo a la vez y si se está diseñando una aplicación que tenga acciones a realizar con *multitouch*, es algo que hay que controlar, aunque en estos momentos casi todos los dispositivos soportan al menos cinco puntos de presión.

En los primeros dispositivos (sobre todo los de tipo resistivo), los eventos de *multitouch* no funcionaban de modo correcto, intercambiando coordenadas entre distintos dedos o simplemente perdiendo control de alguno de ellos, también podemos encontrarnos usuarios con movilidad reducida o poca destreza como personas mayores, por lo que no está de más siempre dar opciones distintas al *multitouch*, como poner botones que hagan la misma función.

En este capítulo construiremos una aplicación para realizar un seguimiento de los dedos detectando el movimiento de cada uno de ellos por separado.

Cómo funciona

Para ver el funcionamiento del sistema *multitouch* en Android lo mejor es ver lo que el sistema capta en cada evento sobre la pantalla; para ello nos valdremos del log de sistema y el ejemplo será realmente corto de programar, pero como mala noticia hay que decir que el emulador de Android no soporta *multitouch* de forma nativa; si bien es posible realizar *tethering* desde un dispositivo real que ejecute la aplicación *SdkControllerMultitouch* disponible en el SDK, pero como para esta emulación también es necesario un dispositivo, lo mejor es probarlo directamente terminal físico.

Cree un nuevo proyecto con los siguientes parámetros:

- Application name: Multitouch Test
- Module Name: Multitouch Test
- Package name: com.acme.multitouch
- Activity Name: Pointers

La clase *Activity* posee un método `onTouchEvent()` que se llama cuando se lanza un evento por parte del usuario y ninguna de las *View* que tenga asociadas la *Activity* se encarga de atenderlo; aprovecharemos esta circunstancia para no tener que variar el *layout* y atender los eventos directamente desde la *Activity*. Vaya a la clase *Pointers* del proyecto donde implementaremos el método `onTouchEvent()` de modo que escriba en el *Log* de salida los eventos que se vayan recibiendo.
Mirando la documentación sobre el evento `onTouch()`, descubrimos que el parámetro que recibe el método es de tipo *MotionEvent* que a su vez expone el método `getAction()` para conocer el tipo de evento que se ha producido. En la documentación también aparecen los tipos de eventos que se pueden recibir pero como están codificados mediante números enteros, crearemos un *array* de cadenas de texto representando a cada uno de los eventos con tal de luego poder generar un log legible. Para escribir en el log del sistema, Android posee una clase con toda la utilidad necesaria con tal de facilitar la tarea; estamos hablando de la clase *Log*. *Log* expone diferentes métodos dependiendo de la severidad con la que se quiere emitir el mensaje de log, en nuestro caso usaremos la severidad *debug* que está representada por el método `d()`; este método acepta como parámetros una etiqueta identificativa del log y el mensaje a guardar. El método quedaría:

```
@Override
public boolean onTouchEvent(MotionEvent event) {
   final String TAG = "Pointer";
   String actions[] = { "DOWN", "UP", "MOVE", "CANCEL", "OUTSIDE",  "POINTER_
DOWN", "POINTER_UP" };
   StringBuilder sb = new StringBuilder();
   int action = event.getAction();
   int actionCode = action & MotionEvent.ACTION_MASK;
   sb.append("evento ACTION_" ).append(actions [actionCode]);
   Log.d(TAG, sb.toString());
   return super.onTouchEvent(event);
}
```

Ejecute la aplicación sobre el emulador y vaya pulsando sobre la pantalla. Dentro de Android Studio podremos ver la salida en el panel Android (seleccionándolo desde el icono situado en la parte inferior izquierda); en la pestaña Devices|logcat .

La salida será semejante a:

```
02-28 18:21:48.776  15715-15715/com.acme.multitouch D/Pointer: evento
                    ACTION_MOVE
02-28 18:21:48.816  15715-15715/com.acme.multitouch D/Pointer: evento
                    ACTION_UP
02-28 18:21:49.261  15715-15715/com.acme.multitouch D/Pointer: evento
                    ACTION_DOWN
02-28 18:21:49.361  15715-15715/com.acme.multitouch D/Pointer: evento
                    ACTION_MOVE
02-28 18:21:49.381  15715-15715/com.acme.multitouch D/Pointer: evento
                    ACTION_MOVE
02-28 18:21:49.396  15715-15715/com.acme.multitouch D/Pointer: evento
                    ACTION_MOVE
02-28 18:21:49.411  15715-15715/com.acme.multitouch D/Pointer: evento
                    ACTION_MOVE
02-28 18:21:49.441  15715-15715/com.acme.multitouch D/Pointer: evento
                    ACTION_MOVE
02-28 18:21:49.451  15715-15715/com.acme.multitouch D/Pointer: evento
                    ACTION_MOVE
02-28 18:21:49.461  15715-15715/com.acme.multitouch D/Pointer: evento
                    ACTION_MOVE
02-28 18:21:49.481  15715-15715/com.acme.multitouch D/Pointer: evento
                    ACTION_MOVE
02-28 18:21:49.496  15715-15715/com.acme.multitouch D/Pointer: evento
                    ACTION_MOVE
```

Si no consigue ver la salida anterior y tiene más de un emulador conectado o algún dispositivo físico, es posible que esté mirando el log de otro dispositivo distinto al log del terminal donde se están generando los métodos. Fíjese en el desplegable situado a la izquierda del panel Android, que no tenga seleccionado otro dispositivo diferente al utilizado para generar la prueba. También puede ser que esté filtrando por otro proceso. Seleccione el proceso com.acme.multitouch.

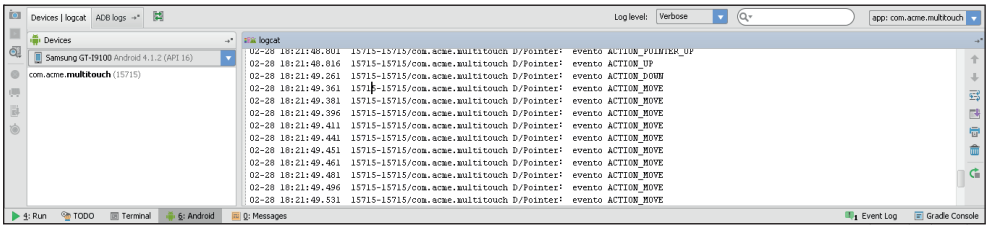

Figura 17.1. Panel Android.

Si aún así no consigue ver la salida y sólo tiene un dispositivo, es posible que el nivel de log lo tenga seleccionado demasiado alto, en la parte superior del panel existe un desplegable con la severidad de los reportes a filtrar llamado Log level, debe seleccionar *debug* o *verbose*.

Cada vez que se pulsa sobre la pantalla del emulador se genera un evento ACTION_DOWN, al moverse se genera un ACTION_MOVE y al levantar el dedo del botón del ratón se genera el evento ACTION_UP. Probémoslo en el dispositivo físico.

Para que se pueda leer los logs del dispositivo físico en el Android Studio, se debe tener activada la opción *debug* en el dispositivo como ya se explicó y realizar las pruebas con el dispositivo conectado al puerto USB del ordenador. Lance la aplicación y pruebe ahora a realizar pulsaciones en la pantalla, verá que se generan los mismos eventos que en el emulador; hasta ahora todo es como se esperaba. Pruebe a pulsar con dos o más dedos, verá una salida semejante a:

```
02-28 18:44:38.901  15715-15715/com.acme.multitouch D/Pointer: evento
                    ACTION_DOWN
02-28 18:44:39.166  15715-15715/com.acme.multitouch D/Pointer: evento
                    ACTION_MOVE
02-28 18:44:39.216  15715-15715/com.acme.multitouch D/Pointer: evento
                    ACTION_MOVE
02-28 18:44:39.266  15715-15715/com.acme.multitouch D/Pointer: evento
                    ACTION_MOVE
02-28 18:44:39.311  15715-15715/com.acme.multitouch D/Pointer: evento
                    ACTION_POINTER_DOWN
02-28 18:44:39.316  15715-15715/com.acme.multitouch D/Pointer: evento
                    ACTION_MOVE
02-28 18:44:39.366  15715-15715/com.acme.multitouch D/Pointer: evento
                    ACTION_MOVE
02-28 18:44:39.436  15715-15715/com.acme.multitouch D/Pointer: evento
                    ACTION_MOVE
02-28 18:44:39.466  15715-15715/com.acme.multitouch D/Pointer: evento
                    ACTION_POINTER_UP
02-28 18:44:39.466  15715-15715/com.acme.multitouch D/Pointer: evento ACTION_UP
```

Han aparecido mensajes que anteriormente no estaban ACTION_POINTER_DOWN y ACTION_POINTER_UP. Estos eventos se producen cuando se detecta más de una presión en el dispositivo; cuando se apoya más de un dedo sobre la pantalla. A cada dedo apoyado sobre la pantalla se le denomina puntero. Supongo que al lector le queda claro que significa ACTION_MOVE, pero vamos a aclarar un poco más el resto de eventos:

- ACTION_DOWN: Se dispara cuando el primer dedo pulsa la pantalla. El índice del puntero para este caso es 0. Marca el inicio de un gesto en pantalla.
- ACTION_POINTER_DOWN: Se dispara cada vez que un dedo extra toca la pantalla. El índice de este puntero es obtenido por la funcion `getActionIndex()`.
- ACTION_POINTER_UP: Se dispara cuando un dedo deja de tocar la pantalla siempre y cuando quede algún dedo tocándola.

- ACTION_UP: Se dispara cuando el último dedo deja de tocar la pantalla. En caso de estar haciendo un gesto sobre pantalla se supone acabado el gesto.
- ACTION_CANCEL: Se dispara si se está realizando un gesto en pantalla y este es abortado por alguna razon (por ejemplo una llamada).

Mejoremos un poco la función de log para obtener más información sobre los punteros disponibles en la pantalla.
La clase *MotionEvent* dispone de toda la información que se necesita para conocer el estado y posición de cada uno de los punteros existentes en cada momento. Cuando simplemente hay una pulsación o al programa no le interesa conocer si hay más de una, entonces se pueden utilizar los métodos abreviados que son `getX()` o `getY()` para obtener su posición y `getPressure()` para obtener la presión de la pulsación entre otros. Cuando se quiere mantener el control de dos o más pulsaciones esta manera de trabajar no es válida.
Para este caso la clase *MotionEvent* mantiene un *array* con todos los punteros que se detectan en pantalla y permite indicar a los métodos explicados anteriormente, el índice del puntero del que se quiere obtener la información, por ejemplo si se tienen dos dedos apoyados, para obtener la posición del segundo dedo, sería mediante `getX(1)`, `getY(1)` (recuerde que el primer índice de un *array* es 0). En caso de no indicar el índice, la información devuelta es la del primer puntero. Sabiendo que el número de punteros detectados en pantalla se obtiene mediante la llamada a `getPointerCount()`, variemos el método anterior para obtener más información, Simplemente interrogaremos un poco más al objeto `event` y mostraremos sus datos en pantalla. Modifique las líneas:

```
sb.append("evento ACTION_" ).append(actions [actionCode]);
Log.d(TAG, sb.toString());
```

Por

```
sb.append("evento ACTION_" ).append(actions[actionCode]);
sb.append("[" );
for (int i = 0; i < event.getPointerCount(); i++) {
   sb.append("{" ).append(i);
   sb.append(" pid=" ).append(event.getPointerId(i));
   sb.append("}->(" ).append((int) event.getX(i));
   sb.append("," ).append((int) event.getY(i));
   sb.append("," ).append(event.getPressure(i));
   sb.append(")");
   if (i + 1 < event.getPointerCount()) sb.append(";" );
}
sb.append("]" );
Log.d(TAG, sb.toString());
```

La salida mostrará el índice, el identificador, la posición y la presión de cada uno de los punteros detectados.

```
02-28 19:04:52.146  16572-16572/com.acme.multitouch D/Pointer: evento
                    ACTION_MOVE[{0 pid=0}->(183,407,0.0375);{1 pid=1}-
                    >(240,575,0.0375)]
02-28 19:04:52.161  16572-16572/com.acme.multitouch D/Pointer: evento
                    ACTION_MOVE[{0 pid=0}->(182,408,0.0375);{1 pid=1}-
                    >(237,576,0.0375)]
02-28 19:04:52.181  16572-16572/com.acme.multitouch D/Pointer: evento
                    ACTION_MOVE[{0 pid=0}->(181,408,0.0375);{1 pid=1}-
                    >(233,577,0.0375)]
02-28 19:04:52.281  16572-16572/com.acme.multitouch D/Pointer: evento
                    ACTION_MOVE[{0 pid=0}->(177,409,0.05);{1 pid=1}-
                    >(225,579,0.0375)]
02-28 19:04:52.291  16572-16572/com.acme.multitouch D/Pointer: evento
                    ACTION_POINTER_UP[{0 pid=0}->(177,409,0.05);{1 pid=1}-
                    >(225,579,0.0375)]
02-28 19:04:52.296  16572-16572/com.acme.multitouch D/Pointer: evento
                    ACTION_MOVE[{0 pid=0}->(176,410,0.05)]
02-28 19:04:52.311  16572-16572/com.acme.multitouch D/Pointer: evento
                    ACTION_MOVE[{0 pid=0}->(175,410,0.05)]
02-28 19:04:52.481  16572-16572/com.acme.multitouch D/Pointer: evento
                    ACTION_MOVE[{0 pid=0}->(174,412,0.05)]
02-28 19:04:52.516  16572-16572/com.acme.multitouch D/Pointer: evento
                    ACTION_UP[{0 pid=0}->(174,412,0.05)]
```

Probando Multitouch

Para acabar de conocer el funcionamiento del *multitouch* en dispositivos Android haremos una aplicación que muestre una circunferencia bajo cada presión que se detecte en pantalla y vaya modificando su posición cuando dicha presión varíe. La aplicación funcionará del mimo modo que la que se hizo para poder pintar con los dedos, sólo que en este caso, solo se pintará la posición en cada momento de cada uno de los punteros en lugar de tener que mantener el trazo. Para la detección de las pulsaciones y su dibujo en pantalla se creará una clase que extienda la clase *View*, tal y como se hizo para la pizarra. Para el caso del evento `onCreate()`, configuraremos dicha clase como contenido de la interfaz de la actividad (esta vez no se usará el archivo de *layout*). También se aprovechará el método `onCreate()` para preparar el pincel que pintará las circunferencias que se mostrarán indicando cada una de las presiones detectadas y para emitir un mensaje al usuario avisándole de cuantos punteros se espera que soporte su dispositivo.

Mediante la llamada al método `hasSystemFeature()` de la clase *PackageManager* se puede ir interrogando al sistema sobre las características del hardware del terminal, simplemente pasando como parámetro la constante

cadena de la característica por la que se le quiere preguntar y él devolverá si está disponible o no, por ejemplo si se quiere conocer si el terminal tiene Bluetooth se usaría:

```
getPackageManager().hasSystemFeature(PackageManager. FEATURE_BLUETOOTH);
```

En nuestro caso existen varias preguntas a hacer para poder determinar el número de punteros esperados y se irán haciendo de modo incremental hasta que una de ellas falle y será la que determine el número máximo de ellos. Mediante la característica `FEATURE_TOUCHSCREEN` se sabe si el dispositivo tiene o no pantalla táctil, con `FEATURE_TOUCHSCREEN_MULTITOUCH` se sabe si soporta dos punteros (por lo que si no soporta esta opción, será que sólo es posible usar un puntero) y `FEATURE_TOUCHSCREEN_MULTITOUCH_DISTINCT` indica que se puede seguir varios punteros de modo independiente. Existe un nivel más que ha aparecido en la API 9 llamado `FEATURE_TOUCHSCREEN_MULTITOUCH_JAZZHAND` que se podría denominar *multitouch* real, donde es capaz de detectar todo tipo de movimientos de cualquier número de punteros y tratarlos todos ellos de modo independiente, pero este modo no lo tendremos en cuenta en nuestro código, simplemente si pasa todos los filtros anteriores, se dirá que soporta todo número de punteros sin dar más importancia.

Elimine todo el código anterior o comience un nuevo proyecto con el método `onCreate()`:

```
Paint mPaint = null;
@Override
protected void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   //se indica contenido de la pantalla
   View v = new Board(this);
   setContentView(v);
   //preparamos el pincel
   mPaint = new Paint();
   mPaint.setAntiAlias(true);
   mPaint.setDither(true);
   mPaint.setColor(0xFFFF0000);
   mPaint.setStyle(Paint.Style.STROKE);
   mPaint.setStrokeJoin(Paint.Join.ROUND);
   mPaint.setStrokeCap(Paint.Cap.ROUND);
   mPaint.setStrokeWidth(4);
   //mostrar mensaje de numero de punteros soportados
   Resources res = getResources();
   if (!getPackageManager().hasSystemFeature(PackageManager.FEATURE_
   TOUCHSCREEN)){
      showMessage( res.getString(R.string.notouch));
   }
   else if (!getPackageManager().hasSystemFeature(PackageManager.FEATURE_
TOUCHSCREEN_MULTITOUCH)){
```

```
            showMessage( res.getString(R.string.onefinger));
        }
        else if (!getPackageManager().hasSystemFeature(PackageManager.FEATURE_
        TOUCHSCREEN_MULTITOUCH_DISTINCT)){
            showMessage( res.getString(R.string.twofingers));
        }
        else {
            showMessage( res.getString(R.string.multifinger));
        }
    }
```

Para mostrar el mensaje, podemos usar cualquiera de los métodos ya conocidos, por ejemplo un mensaje *Toast*, que es el menos intrusivo.

```
private void showMessage(String msg) {
    Toast.makeText(this,msg,Toast.LENGTH_SHORT).show();
}
```

Queda ahora realizar el código para la clase *Board*. Dentro de la clase *Pointers*, cree la clase interna *Board* que extienda la clase *View* y sobrescribiremos los métodos `onTouchEvent()` y `onDraw()`. En el método `onTouchEvent()` nos tenemos que encargar de capturar los eventos que interesan (todos aquellos que tienen que ver con que se detecte una nueva pulsación o movimiento), guardar las coordenadas donde se encuentra cada pulsación en un *ArrayList* e invalidar la vista para forzar la llamada a `onDraw()`.

El método `onDraw()` será donde se recorrerá el *ArrayList* en el que se encuentran las coordenadas en que se han localizado las pulsaciones y se irán pintando las circunferencias tomando como centro dichas coordenadas.

```
public class Board extends View {
private ArrayList<PointF> pointers = null;
public Board(Context c) {
    super(c);
}
@Override
protected void onDraw(Canvas canvas) {
    canvas.drawColor(0xFFAAAAAA);
    if (pointers != null){
        PointF p = null;
            // iteramos sobre cada pareja de corrdenadas
            Iterator<PointF> iter = pointers.iterator();
            while (iter.hasNext()){
                p = iter.next();
                // se pinta una circunferencia de radio 40 y centro
                // las coordenadas de la pulsación
                canvas.drawCircle(p.x, p.y, 40, mPaint);
            }
    }
}
```

```
@Override
public boolean onTouchEvent(MotionEvent event) {
   switch (event.getAction()) {
      case MotionEvent.ACTION_DOWN:
      case MotionEvent.ACTION_POINTER_DOWN:
      case MotionEvent.ACTION_MOVE:
      case MotionEvent.ACTION_POINTER_UP:
         pointers = new ArrayList<PointF>();
         for (int i =0; i < event.getPointerCount(); i++){
            pointers.add(newPointF(event.getX(i),event.getY(i)));
         }
         invalidate();
         break;
      case MotionEvent.ACTION_UP:
         // no quedan dedos pulsando, limpiamos el array
         pointers.clear();
         invalidate();
         break;
   }
   return true;
}
}
```

Modifique el fichero `strings.xml` con las constantes necesarias:

```
<string name="notouch">No soporta pantalla táctil</string>
    <string name="onefinger">Soporta un puntero</string>
    <string name="twofingers">Soporta dos punteros</string>
    <string name="multifinger">Soporta múltiples punteros</string>
```

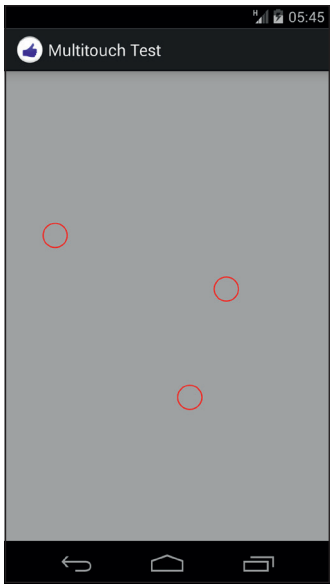

Figura 17.2. Multitouch Test en acción.

Cuando se pruebe la aplicación, en cada lugar donde se produzca una pulsación se dibujará una circunferencia roja. Si se dirige de nuevo al panel *Android* a revisar los logs (y no ha borrado el método `onTouchEvent()` de la clase principal), verá que ya no se están registrando los eventos tal y como se hacía anteriormente. Esto es porque la *Activity* ahora contiene un elemento *View* que es quien está gestionando estos eventos. Si se fija en el evento `onTouchEvent()` de la clase *Board*, se devuelve un `true` al final de la ejecución; esto es para indicar que la clase ha gestionado el evento y que no necesita ser propagado "hacia abajo", hacia vistas que estén definidas por debajo de la actual o hacia la actividad que la gestiona. Si se quiere volver a tener el registro de los eventos en el log, tal y como se tenían antes, simplemente habría que devolver `false` en este método para indicar que el evento no se ha gestionado y entonces se encargaran de gestionarlo las vistas definidas por debajo de esta y en último caso su actividad.
Ahora que ya sabe cómo controlar varias pulsaciones en pantalla le invito a mejorar el programa pizarra visto anteriormente para poder pintar con varios dedos a la vez.

Gestures

Anteriormente hemos hablado de los gestos que se pueden hacer sobre pantalla y de cómo Android se basa en ellos para capturar los movimientos de los dedos. Entraremos a ver un poco más sobre los gestos.
Desde el método `onTouchEvent()` que hemos visto anteriormente, podríamos generar nuestros propios algoritmos para diferenciar distintos tipos de gestos teniendo en cuenta el número de dedos apoyados, movimiento que realizan y velocidad a la que lo hacen. Alguno de estos gestos es complicado de controlar y es por eso que existen una serie de interfaces y clases para ayudarnos en la tarea.
Instanciando un objeto de tipo *GestureDetector* podemos pasarle como parámetro una clase que implemente la interfaz *GestureDetector.OnGestureListener* y en esta clase seremos capaces de controlar gestos como la pulsación larga, el deslizado lento (*scroll*) o el deslizado rápido (*fling*); para ello vale con implementar los métodos que nos interesen. Para controlar la doble pulsación disponemos de la interfaz *GestureDetector.OnDoubleTapListener*, que simplemente implementándola en nuestra clase, hará que dispongamos de una serie de métodos que se dispararán ante este tipo de pulsación.
Para implementar su uso, como decíamos se deben implementar las interfaces (en este caso usaremos las dos) y completar los métodos que éstas exponen.

```
public class MainActivity extends Activity implements
        GestureDetector.OnGestureListener,
        GestureDetector.OnDoubleTapListener{

private static final String DEBUG_TAG = "Gestures";
private GestureDetector mDetector;

@Override
public void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   setContentView(R.layout.activity_main);
   // se instancia el detector de gestos teniendo como parámetros
   // el contexto de aplicación y la clase que implementa GestureDetector.
OnGestureListener
   mDetector = new GestureDetector(MainActivity.this, MainActivity.this);
   //asignamos al detector de gestos para que escuche el doble toque
   mDetector.setOnDoubleTapListener(MainActivity.this);
}

@Override
public boolean onTouchEvent(MotionEvent event){
   //llamamos al detector
   mDetector.onTouchEvent(event);
   // Obligatorio llamar a la clase superior!!
   return super.onTouchEvent(event);
}

// sobrescribimos los métodos de las interfaces implementadas
@Override
public boolean onDown(MotionEvent event) {
   Log.d(DEBUG_TAG,"onDown");
   return true;
}

// añadir el resto de métodos
....

@Override
public boolean onDoubleTapEvent(MotionEvent event) {
   Log.d(DEBUG_TAG, "onDoubleTapEvent");
   return true;
    }
```

Es importante llamar en el método `onTouchEvent()` al detector de gestos. En Android Studio, cuando escribimos que una clase extiende una interfaz pero no implementamos sus métodos, dará error (como es lógico); pulsando sobre el nombre de la clase podremos seleccionar implementar los métodos faltantes, como muestra la figura 17.3.

En caso de querer utilizar el gesto de pellizco como el que se usa en el zoom (también denominado *pinch & zoom*), podríamos valernos de los métodos vistos hasta el momento y nuevamente hacer nosotros el cálculo, pero no es necesario puesto que disponemos de la clase *ScaleGestureDetector* que nos facilitará la tarea. Para trabajar con *ScaleGestureDetector* necesitamos instan-

ciarlo pasando al constructor el contexto de la aplicación y una instancia de una clase que implemente la interfaz *ScaleGestureDetector.OnScaleGestureListener* (para facilitar aún más su uso, podemos usar una clase que extienda *ScaleGestureDetector.SimpleOnScaleGestureListener*).

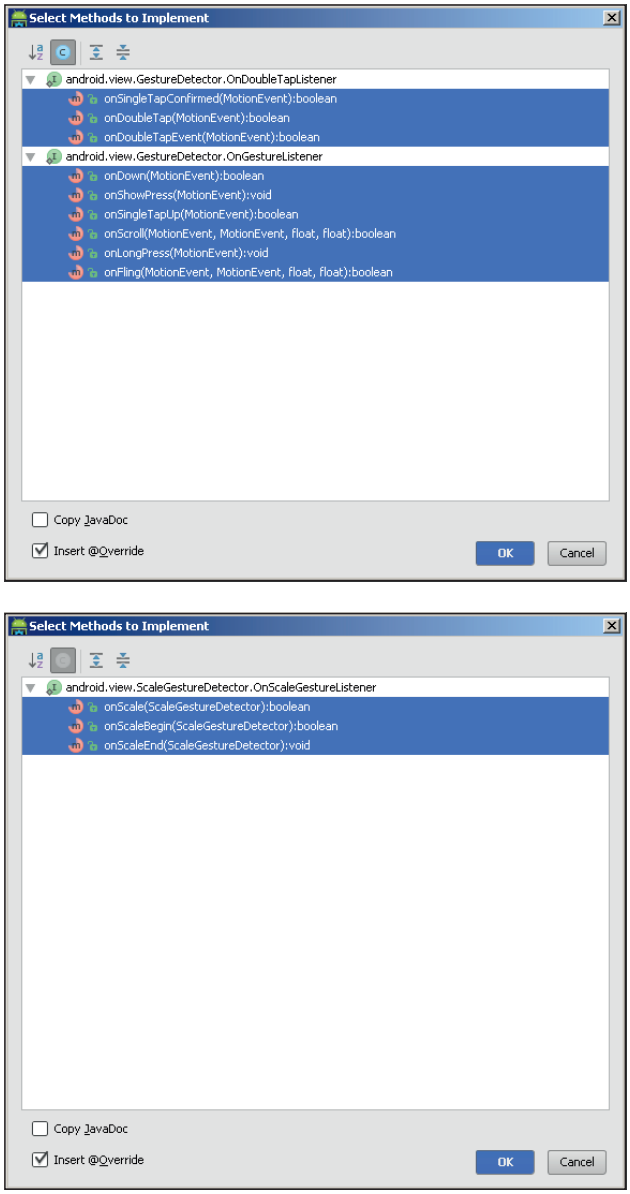

Figura 17.3. Implementación de los métodos de las interfaces.

El uso básico sería :

```
public class MainActivity extends Activity implements
   ScaleGestureDetector.OnScaleGestureListener{

private static final String DEBUG_TAG = "Pinch";
private ScaleGestureDetector mScaleDetector;
private float mScaleFactor = 1.f;

@Override
public void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   setContentView(R.layout.activity_main);
   // se instancia el detector de escalado teniendo como parámetros
   // el contexto de aplicación y la clase que implementa el listener
   mScaleDetector = new ScaleGestureDetector(MainActivity.this,
MainActivity.this);
}

@Override
public boolean onTouchEvent(MotionEvent event){
   //llamamos al detector de escalado
   mScaleDetector.onTouchEvent(ev);
   // Obligatorio llamar a la clase superior!!
   return super.onTouchEvent(event);
}

// sobrescribimos los métodos de las interfaces implementadas
@Override
public boolean onScale(ScaleGestureDetector scaleGestureDetector) {
   mScaleFactor *= detector.getScaleFactor();

   //fijamos máximo y mínimo de la escala.
   mScaleFactor = Math.max(0.1f, Math.min(mScaleFactor, 10.0f));
   Log.d(DEBUG_TAG,"onScale, valor de escala: " + );
   return true;

}

@Override
public boolean onScaleBegin(ScaleGestureDetector scaleGestureDetector) {
   Log.d(DEBUG_TAG,"onScaleBegin");
   return true;
}

@Override
public void onScaleEnd(ScaleGestureDetector scaleGestureDetector) {
   Log.d(DEBUG_TAG,"onScaleEnd");
}

}
```

Como en el caso anterior, importante llamar en el método `onTouchEvent()` al detector de escalado. En el método `onScale()` recogemos el valor de la escala que tiene el detector y lo multiplicamos por el valor que teníamos hasta el momento almacenado en `mScaleFactor`, obteniendo así la nueva escala. Con la línea:

```
mScaleFactor = Math.max(0.1f, Math.min(mScaleFactor, 10.0f));
```

nos aseguramos que la escala no es más pequeña que un décimo ni más grande que diez veces el valor original. Ahora solo tendríamos que tener algo que escalar, por ejemplo un zoom que vaya de 1 a 100, al que le daremos un valor inicial de 10, así cuando se pellizque la pantalla, el atributo `mScaleFactor` irá disminuyendo hasta tener un valor de 0.1 y si se realiza el gesto del pellizco pero separando los dedos, `mScaleFactor` llegará a tener el valor 10. Así podríamos tener:

```
protected static final int INITIAL_ZOOM = 10;
...
currentZoom = INITIAL_ZOOM * mScaleFactor
```





18 Fondos de pantalla en movimiento

En este capítulo aprenderá a:

- Crear fondos de pantalla.
- Animar los fondos de pantalla a partir de varios gráficos.
- Añadir configuraciones al fondo de pantalla.
- Generar eventos que modifiquen el fondo.

Al igual que en los ordenadores personales, los dispositivos móviles permiten ajustar su aspecto a las preferencias del usuario en mayor o menor medida. Uno de los aspectos que todos los dispositivos móviles ofrecen es la capacidad de cambiar el fondo pantalla, de modo que muestre una imagen o un color definido por el usuario. Centrándonos en los dispositivos Android, esta opción como es lógico, está disponible desde la primera versión del sistema, pero desde la versión 2.1 se ofrecen unos fondos muy especiales, los fondos de pantalla en movimiento. Para poderlos utilizar lo primero, como hemos dicho, es tener una versión de Android superior a la 2.1; dependiendo de la versión con la que se trate, el procedimiento será distinto, pero por normal general se accede a ellos manteniendo pulsado el fondo de la pantalla actual y seleccionando diversos menús:

Figura 18.1. Ejemplo de acceso a fondos de pantalla en teléfono

A la hora de diseñar un fondo de pantalla animado, se puede ver como una aplicación informativa o lúdica, que debe interferir lo mínimo posible en el funcionamiento normal del dispositivo, es decir, no se deben diseñar fondos animados que presenten muchos movimientos o colores y formas estridentes ya que dificulta la visión de los iconos y widgets que tenga el usuario en la pantalla y se hace incómodo su uso. Otro aspecto que hay que tener en cuenta a la hora de pensar un fondo animado es el consumo de batería. Ha de pensar el lector que pese a que el motor que hace que el fondo se mueva y se

ejecuten los métodos de actualización se detiene momentáneamente cuando dicho fondo no se muestra por pantalla (por ejemplo al lanzar una aplicación), también es cierto que mientras se enciende el teléfono y se navega por los escritorios, el consumo que realice puede impactar en la vida de la batería. Imagine que hace un fondo de pantalla animado que dada una localización muestra un sol si está despejado y unas nubes si se encuentra nublado, dos gráficos simplemente sin animación ninguna. Se podría programar para que cada segundo se realice una conexión con algún servicio meteorológico, descargar la información del lugar y mostrar el gráfico correspondiente como fondo de pantalla; esto daría una exactitud pasmosa al fondo de pantalla, ya que estaría totalmente actualizado, pero también haría que el usuario lo desinstalara rápidamente por quedarse sin batería y por tener un alto consumo de datos. También podría pasar que se refrescara la información cada 3 horas por ejemplo, pero que sin embargo, en lugar de tener dos gráficos estáticos, se mostrara cuando lloviera unas nubes en movimiento con gotas cayendo aleatoriamente y con cálculos físicos de choque de partículas líquidas, realismo total, pero vida de la batería mínima y movimiento ralentizado por consumo de CPU. Con esto quiero decir al lector que una vez más debe cuidar los recursos del dispositivo, y hacer que sus aplicaciones, widgets y fondos de pantalla sean agradables y útiles a la vez que convivan con otras aplicaciones cediendo los recursos.

Para el diseño de los fondos en movimiento en Android, se puede decir que los límites los pondrá la propia imaginación, entre las cosas que se pueden hacer se encuentran:

- Hacer animaciones simples, mediante la muestra de distintos gráficos de manera coordinada
- Dibujar en la pantalla dependiendo de un dato obtenido de un servidor externo
- Mostrar diferente información o colores dependiendo de datos propios del dispositivo como podría ser número de correos electrónicos sin leer
- Reaccionar a eventos como pulsaciones del usuario sobre la pantalla
- Leer información de los sensores y plasmarla sobre el fondo de pantalla
- ...

El funcionamiento es muy sencillo, se debe hacer un servicio encargado de instanciar un objeto de la clase WallpaperService.Engine y dibujar en pantalla lo que se quiera ir mostrando, bien actualizándola mediante algún tipo de temporizador, o bien mediante algún otro disparador, por ejemplo una llamada de teléfono si queremos mostrar como fondo de pantalla las últimas

llamadas recibidas o el método de callback de posicionamiento si queremos mostrar la última posición conocida, pero no tenemos que preocuparnos de las dificultades de gestión del propio escritorio.
Aunque para comprender mejor el funcionamiento lo mejor es hacer uno de estos fondos de pantalla partiendo de cero.

Ejemplo de fondo de pantalla en movimiento

Una vez más se tratará de que el lector adquiera conocimiento del modo de creación de las aplicaciones mediante un ejemplo práctico que incluya las opciones más comunes del desarrollo, en este caso para el fondo en movimiento, se aprovechará parte del código realizado anteriormente, más concretamente el código utilizado en el proyecto MetalBall, en el que la bola se movía dependiendo de los valores leídos por el sensor. Se tratará de hacer un fondo que muestre dicha bola moviéndose sobre un fondo de pantalla seleccionado y con opción de sustituir la bola por otros elementos, pero en este caso se eliminará la vibración que se realizaba cuando la bola tocaba los bordes, por preservar la batería y la paciencia del usuario, ya que podría ser realmente incómodo que el dispositivo vibrara durante el uso normal del mismo.
Cree un nuevo proyecto con la estructura:

- Application name: MetalWallpaper
- Module Name: MetalWallpaper
- Package name: com.acme.metalwallpaper
- Minimum required SDK: API 11: Android 3.0 (Honeycomb) o superior
- Desmarque la casilla `Create custom launcher icon`
- Desmarque la casilla `Create activity`

En este caso hemos desmarcado tanto la casilla de crear icono como la de crear actividad dado que el fondo de pantalla no necesita una actividad propia puesto que se basa, como veremos, en un servicio, y porque tampoco necesita un icono ya que no está accesible desde el Launcher de aplicaciones, sino que necesita una imagen mayor para mostrar en el selector de fondos de pantalla en movimiento, más adelante también nos encargaremos de dicha imagen.
Lo primero que se necesita para un fondo de escritorio en movimiento es un fichero de descripción del propio fondo que como ya es debe estar imaginando se trata de un archivo XML, dentro del cual se informará la imagen del fondo a mostrar en la pantalla de selección (una previsualización o un

logo...), una descripción sobre lo que hace el fondo de pantalla y una clase de configuración, que se encargaría de mostrar las preferencias para ajustar el fondo al gusto del usuario, por ejemplo si hacemos el fondo que muestre el tiempo que hace en una ciudad, habremos de dejar seleccionar la ciudad al usuario. El archivo XML debe encontrarse alojado en el directorio `/src/main/res/xml` del proyecto; llámelo `wallpaper.xml`, y dele el siguiente contenido:

```
<wallpaper
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:thumbnail="@drawable/ball"
    android:description="@string/lwp_description"/>
```

Donde `android:thumbnail` es la imagen a mostrar en el selector de fondos (que en este caso aprovecharemos la propia bola a mostrar, aunque muchas veces se pone una captura del fondo en acción), y `android:description` es la descripción del fondo, que hace o para que sirve. Dentro del fichero strings.xml hay que dar contenido a la descripción:

```
<resources>
    <string name="app_name">MetalWallPaper</string>
    <string name="lwp_description">Muestra una bola de metal que se desplaza
por la pantalla con el movimiento del dispositivo</string>
</resources>
```

La descripción del fondo de pantalla se enlaza con la aplicación a través de un servicio que se debe definir en el archivo `AndroidManifest.xml`. El servicio se encargará de gestionar la vida del fondo de pantalla y permitir que se muestre bajo los iconos disponibles en los escritorios. Para la creación del servicio, podemos optar por generar la entrada en el archivo `AndroidManifest.xml` y la clase correspondiente a mano, o utilizar el asistente. Para acceder al asistente se debe pulsar con el botón derecho del ratón sobre el árbol del proyecto y seleccionar el menú New>Android Component, tras ello seleccionar la opción Service y seleccionar como nombre de la clase WallpaperService. No obstante este asistente sólo nos creará el esqueleto, que deberemos completar debidamente modificando los archivos para adaptarlos a nuestras necesidades. Dentro de la etiqueta `<application>` del `AndroidManifest.xml`, hacemos disponible el servicio:

```
<service android:name="com.acme.metalwallpaper.WallpaperService"
   android:enabled="true" android:label="@string/app_name"
   android:permission="android.permission.BIND_WALLPAPER">
   <intent-filter android:priority="1">
      <action android:name="android.service.wallpaper.WallpaperService" />
   </intent-filter>
   <meta-data android:name="android.service.wallpaper"
              android:resource="@xml/wallpaper" />
</service>
```

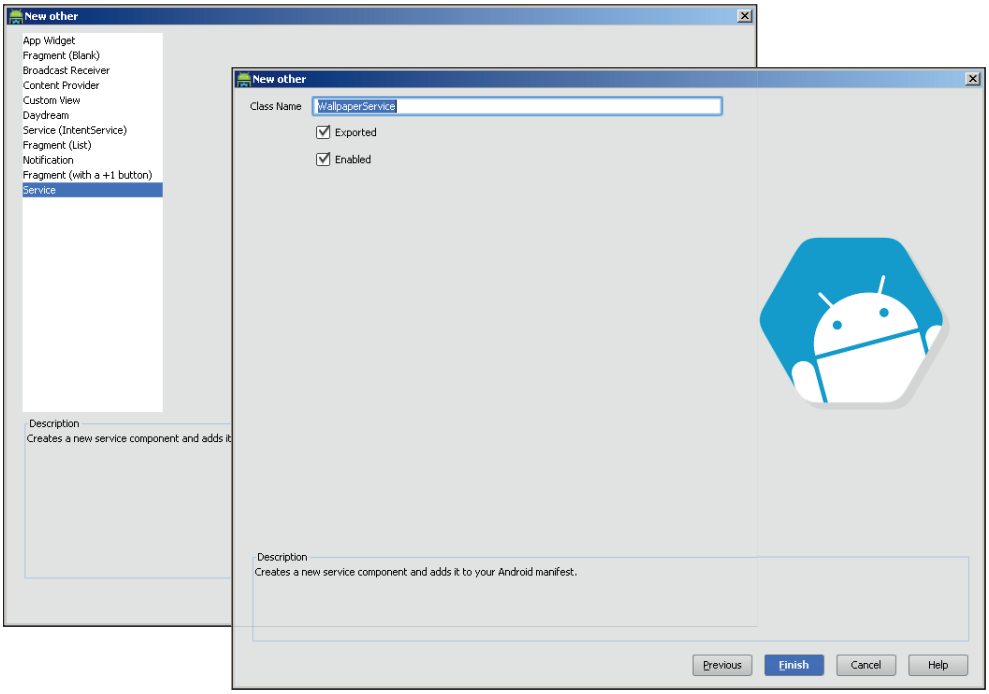

Figura 18.2. Asistente para generación de servicios

Donde dentro de `<service>` el `android:name` es la clase del servicio que atenderá la vida del fondo de pantalla, `android:enabled=true` sirve para indicar que el componente es instanciable por el sistema, por defecto es positivo e informarlo es optativo, `android:label` permite especificar el nombre del fondo de pantalla (no confundir con la descripción, que se encuentra en el archivo de configuración realizado anteriormente) y `android:permission` para indicar que necesitará permiso para utilizarse como fondo de pantalla y poder permanecer en la pantalla principal, en el home screen. Como parte de la etiqueta `<service>` podemos encontrar dos componentes, el `<intent-filter>` ya conocido y el `<meta-data>`, que se trata de un elemento que sirve para informar parámetros adicionales a su elemento padre, es decir, en este caso se están informando datos al elemento `<service>`, al servicio. Dentro del `<intent-filter>` se indica que este elemento será un fondo de pantalla y como tal debe salir en el selector de fondos animados. Por otro lado dentro del elemento `<meta-data>` se informa al servicio que el descriptor del fondo de pantalla se encuentra en el recurso `android:resource="@xml/wallpaper"`. Un servicio puede ser de múltiples tipos y es por eso que la información adicional se pasa mediante metadatos en lugar de hacerlo mediante algún atributo concreto.

Una última cosa a realizar en el `AndroidManifest.xml` antes de abandonarlo es indicar que usaremos los permisos de fondo de pantalla en movimiento y es que al poder hacer uso de recursos compartidos y ocupar la pantalla principal del dispositivo, es necesario de alguna manera indicárselo al usuario, es por eso que es necesario indicar este uso. Tal y como los utilizados anteriormente, se debe declarar dentro de la etiqueta `<manifest>`, fuera de la `<application>` y preferiblemente antes de la declaración de ésta.

```
<uses-feature android:name="android.software.live_wallpaper" />
```

Una vez tenemos todo preparado para que la aplicación sepa de la existencia del servicio para el fondo de pantalla, es hora de crearlo. Realmente el servicio como tal no tiene que hacer mucho, simplemente devolver una instancia de la clase WallpapeService.Engine que se encargará de todo. Si no ha utilizado el asistente, genere una nueva clase que se llame como el servicio escrito en el `AndroidManifest.xml`, es decir `WallpaperService.java` dentro del paquete `com.acme.wallpaper`; si ha utilizado el asistente, ya la tendrá creada. Para poderla utilizar como este tipo de servicio debe extender la clase android.service.wallpaper.WallpaperService. Al haber extendido esta clase, nos obliga a implementar el método `onCreateEngine()` que tiene se encarga de crear y devolver el motor (clase WallpaperService.Engine) encargado de pintar todo lo que suceda en el fondo animado que estamos creando. El motor que controlará todo se implementará unas líneas más adelante y lo llamaremos MetalBGEngine, así que devolveremos una instancia de él en el método `onCreateEngine()`. La clase por ahora presenta el siguiente aspecto:

```
public class WallpaperService extends android.service.wallpaper.
WallpaperService {
   @Override
   public Engine onCreateEngine() {
      return new MetalBGEngine();
   }
}
```

En nuestra clase motor, además de extender la ya nombrada clase Engine, deberá implementar la clase SensorEventListener para ser capaces de detectar los movimientos del teléfono y así hacer que la bola "ruede" por la pantalla, del mismo modo que se hizo en el capítulo 16 en el ejemplo de los sensores. Cree la clase MetalBGEngine como clase interna del servicio recién creado, e implemente los métodos obligados por SensorEventListener .Tendrá un aspecto semejante a:

```
public class MetalBGEngine extends Engine implements SensorEventListener{
   @Override
   public void onAccuracyChanged(Sensor sensor, int accuracy) {
      // TODO Auto-generated method stub
```

```
    }
    @Override
    public void onSensorChanged(SensorEvent event) {
       // TODO Auto-generated method stub
    }
}
```

Como ya se ha comentado, el servicio no haría más que devolver el objeto Engine que será quien se encargue de todo, así pues, todo el código que se verá a partir de ahora, irá dentro de la clase MetalBGEngine. Comenzaremos por crear propiedades de clase para guardar datos que se necesitarán para poder pintar la bola en la pantalla, tales como las posiciones de la bola, tamaños de pantalla o referencias a los sensores:

```
//handler para el thread
private final Handler mHandler = new Handler();
//muestra si el fondo de pantalla está visible
private boolean mVisible;
//contexto de la aplicación
private Context mContext = null;
//fondo estático de pantalla, gráfico sobre el que se moverá la bola
private Bitmap mBackground = null;
//sensores
private SensorManager mSensorManager;
private Sensor mAccelerometer;
//posiciones y tamaños de la bola
private float lastGX;
private float lastGY;
private float cy = 10;
private float cx = 10;
private int mPictureWidth;
private boolean mNoBorderX = false;
private int mPictureHeight;
private boolean mNoBorderY = false;
//gráfico de la bola
private Bitmap mBall = null;

//tamaños de pantalla
private int mWidth;
private int mHeight;
//thread de dibujado
private final Runnable mDrawBackgroud = new Runnable() {
   public void run() {
      drawFrame();
   }
};
```

Por ahora el método `drawFrame()` se queda sin implementar, ya volveremos a él más adelante. A continuación necesitaremos un constructor de la clase MetalBGEngine en la que simplemente obtendremos una referencia al contexto de la aplicación Android y configuraremos la aplicación a través del método `configBg()`:

```
MetalBGEngine() {
   mContext = getApplicationContext();
   configBg();
}

private void configBg() {
   //obtenemos referencias a los gráficos a pintar
   mBackground = BitmapFactory.decodeResource(mContext.getResources(),
         R.drawable.bg);
   mBall = BitmapFactory.decodeResource(mContext.getResources(),
            R.drawable.ball);
   mPictureHeight = mBall.getHeight();
   mPictureWidth = mBall.getWidth();
   // obtenemos referencias al servicio
   mSensorManager = (SensorManager) mContext
         .getSystemService(Activity.SENSOR_SERVICE);
   // interesa el acelerómetro
   mAccelerometer = mSensorManager
         .getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
   mSensorManager.registerListener(this, mAccelerometer,
         SensorManager.SENSOR_DELAY_GAME);
}
```

En el método `configBg()` se prepara el objeto para cuando sea necesario que pinte en pantalla tenga todo lo necesario, como pueden ser el dibujo de fondo, el dibujo de la bola, las medidas de la bola a dibujar y los accesos a la lectura de los acelerómetros, de modo que sólo queda calcular la posición en la que dibujar la bola, dibujarla sobre el fondo estático y mostrarlo en pantalla, pero antes de ponernos con ello, hay que asegurar que el fondo de pantalla en movimiento no se sigue redibujando en situaciones como que se apague la pantalla o se inicie una aplicación, y tener en consideración temas como que el usuario gire el teléfono y pase de posición horizontal a vertical. Lo haremos a través de algunos métodos de callback.
Comenzaremos añadiendo el método a ejecutar el paso previo a cuando el motor deja de existir:

```
@Override
public void onDestroy() {
   super.onDestroy();
   mSensorManager.unregisterListener(this);
   mHandler.removeCallbacks(mDrawBackgroud);
}
```

En este método se aprovechará para dejar de escuchar los sensores, puesto que ya no se usarán más y para hacer que el thread encargado del dibujo no se siga ejecutando (puesto que ya no existe dónde dibujar).
El siguiente método que añadiremos es para cuando deja de verse el fondo de pantalla, por ejemplo cuando se ejecuta una aplicación:

```
@Override
public void onVisibilityChanged(boolean visible) {
   mVisible = visible;
   if (visible) {
      mSensorManager.registerListener(this, mAccelerometer,
            SensorManager.SENSOR_DELAY_GAME);
         drawFrame();
   } else {
      mSensorManager.unregisterListener(this);
      mHandler.removeCallbacks(mDrawBackgroud);
   }
}
```

Está claro que si se deja de ver el fondo de pantalla, es inútil seguir actualizándolo, simplemente conseguiríamos gastar batería y obtener algún que otro leak de memoria. Lo que se hace en el método `onVidibilityChanged()` es controlar si ha habido un cambio de visible a invisible y viceversa. El parámetro visible del método indica el nuevo estado del fondo de pantalla, es decir si actualmente se encuentra visible o no. En caso de estar visible, se registra el acelerómetro para poder controlar la posición de la bola y se redibuja el fondo de pantalla estático y la bola en la posición correspondiente, en caso de estar invisible, al igual que en `onDestroy()`, se elimina el registro del acelerómetro para dejar de "escucharlo" y se evita que el thread encargado de pintar lo siga haciendo.
En el posible caso que el usuario gire la pantalla y por ejemplo pase de vertical a horizontal, las condiciones de trabajo cambian, ya que el ancho y alto de la pantalla habrá cambiado (raro es el dispositivo que el alto y ancho sea igual).

```
@Override
public void onSurfaceChanged(SurfaceHolder holder, int format,
      int width, int height) {
   super.onSurfaceChanged(holder, format, width, height);
   this.mWidth = width;
   this.mHeight = height;
}
```

El método `onSurfaceChanged()` se ejecuta cada vez que el tamaño de la superficie sobre la que se pinta (el objeto SurfaceHolder) cambia de tamaño. En este caso lo que se hace es mantener los nuevos valores de altura y anchura para controlar el movimiento de la bola. Puede ser que lo que se destruya sea precisamente el SurfaceHolder, con lo que se ejecutaría el método `onSurfaceDestroyed()` que implementaremos del siguiente modo:

```
@Override
public void onSurfaceDestroyed(SurfaceHolder holder) {
      super.onSurfaceDestroyed(holder);
   mVisible = false;
   mHandler.removeCallbacks(mDrawBackgroud);
}
```

En el caso que lo que se destruya el SurfaceHolder, está claro que no se puede pintar sobre él, por lo que se debe detener el thread de dibujado. Por último y para acabar con estos métodos de callback, quedaría el caso en el que el usuario se mueve de un escritorio a otro. En los dispositivos Android se pueden tener tantos escritorios virtuales como se quiera, y para pasar de uno a otro vale con deslizar el dedo por pantalla de izquierda a derecha o de derecha a izquierda cuando se está en la pantalla principal. Este movimiento hace que se desplacen los iconos posicionados en estas pantallas, dejando ver nuevos iconos (o widgets) pero también desplaza el fondo de pantalla. Nuestro fondo de pantalla animado será notificado mediante el método `onOffsetsChanged()`, que implementaremos con el siguiente código.

```
@Override
public void onOffsetsChanged(float xOffset, float yOffset, float xStep,
      float yStep, int xPixels, int yPixels) {
   drawFrame();
}
```

En este caso simplemente se obliga a redibujar el fondo sobre el que se dibuja la bola por si hay que ajustar su posición. Más adelante el método `drawFrame()` también se encargará de llamar al método que dibuje la bola. Aquí también se podrían calcular desplazamientos parciales del fondo de pantalla y dar más realismo haciendo que el fondo se mueva a la vez que lo hacen los escritorios.
Hemos dejado sin implementar un método que sin embargo hemos utilizado en varias partes del código, se trata del método `drawFrame()`. En este método lo que se hará principalmente será bloquear la superficie de dibujo para poder dibujar sobre ella el fondo de pantalla estático sobre el que se moverá la bola para crear la sensación de movimiento, aunque el dibujo en sí se realiza en otro método.

```
void drawFrame() {
   final SurfaceHolder holder = getSurfaceHolder();

   Canvas c = null;
   try {
      c = holder.lockCanvas();
      if (c != null) {
         // dibuja la bola
         paintBall (c);
   }
   } finally {
      if (c != null)
         holder.unlockCanvasAndPost(c);
   }

   mHandler.removeCallbacks(mDrawBackgroud);
   if (mVisible) {
      mHandler.postDelayed(mDrawBackgroud, 1000 / 25);
   }
}
```

Aquí lo que se está haciendo es obtener la superficie sobre la que poder pintar y bloquearla, tras ello pintar sobre dicha superficie y nuevamente desbloquearla; finalmente si está el fondo visible, mediante la instrucción `mHandler.postDelayed(mDrawBackgroud, 1000 / 25)`, deja preparado para el que el thread se vuelva a ejecutar en un cuarto de segundo, es decir, cada 250 milisegundos aseguraremos que se ejecuta él thread `mDrawBackgroud`, que si se fija en el código, lo único que hace es volver a invocar este método. Veremos que para este caso no es del todo necesario, ya que forzaremos el dibujado desde otro código, pero para fondos en movimiento que no tengan otro "disparador" de redibujado, esta sería la manera de hacerlo.
Dibujemos ahora la bola y el fondo mediante el método:

```
private void paintBall(Canvas canvas) {
   RectF rf = new RectF(0, 0, mWidth, mHeight);       canvas.
drawBitmap(mBackground, null, rf, null);
   canvas.drawBitmap(mBall, cx, cy, null);
}
```

El método para pintar la bola sobre la superficie lo que hace es generar un rectángulo de las dimensiones de la pantalla, sobre el dibujar el fondo de pantalla estático y sobre éste se dibuja la bola en la posición que corresponda según se haya ido detectando el movimiento del teléfono. Como ya se vio en el capítulo sobre sensores, para obtener las variaciones de la bola, se debían leer los acelerómetros para conocer la inclinación del teléfono y así poder actualizar la posición de la bola a partir de la posición actual. La lectura se realizaba a través del método de callback `onSensorChanged()` del que se tienen que obtener las variaciones del eje "x" y del eje "y", correspondientes a los valores 0 y 1 del array de valores pasados como parámetro al método.
Implemente para ello el método del siguiente modo:

```
@Override
public void onSensorChanged(SensorEvent event) {
   updateMe(event.values[0], event.values[1]);
}
```

Por último para poder probar el fondo de pantalla en movimiento queda implementar el método `updateMe()` que debe encargar de controlar la posición de la bola a partir de los valores leídos por el sensor y tener en cuenta que no puede salirse de los bordes de la pantalla.

```
public void updateMe(float inx, float iny) {
// actualiza aceleración
lastGX -= inx / 10;
lastGY += iny / 10;
// actualiza posición
```

```
cx += (lastGX);
cy += (lastGY);
// comprobar bordes
// poner aceleración a 0
// no dejar que sobrepase los bordes de la pantalla
if (cx > (mWidth - mPictureWidth)) {
   cx = mWidth - mPictureWidth;
   lastGX = 0;
   if (mNoBorderX) {
         mNoBorderX = false;
   }
} else if (cx < (0)) {
   cx = 0;
   lastGX = 0;
   if (mNoBorderX) {
         mNoBorderX = false;
   }
} else {
   mNoBorderX = true;
}
if (cy > (mHeight - mPictureHeight)) {
   cy = mHeight - mPictureHeight;
   lastGY = 0;
   if (mNoBorderY) {
      mNoBorderY = false;
   }
} else if (cy < (0)) {
   cy = 0;
   lastGY = 0;
   if (mNoBorderY) {
      mNoBorderY = false;
   }
} else {
   mNoBorderY = true;
}
}
```

Lógicamente necesitaremos crear los gráficos necesarios: uno para la bola llamado `ball` y otro para el fondo de pantalla sobre el que se deslizará la bola llamado `bg` y ya podríamos probar nuestro fondo animado; para hacerlo, se debe ejecutar del mismo modo que hasta ahora, pero se ha de tener en cuenta que no se podrá encontrar ningún icono ni se lanzará en el dispositivo por defecto, sino que se debe pulsar en el fondo de la pantalla principal para que salga el menú visto en las figuras.

Si al intentar instalar el fondo de pantalla se nos muestra una pantalla semejante a la figura 18.3 no hay porqué preocuparse; dado que no se ha creado ninguna actividad Android Studio no ha generado la configuración por defecto (es posible que en futuras versiones sí la realice por defecto); simplemente se debe seleccionar Do not launch Activity y pulsar sobre el botón **Run**.

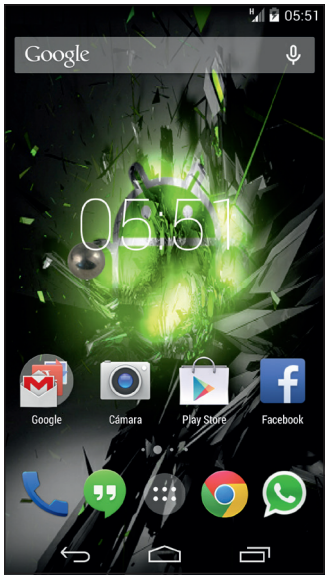

Figura 18.3. Fondo de pantalla en movimiento funcionando

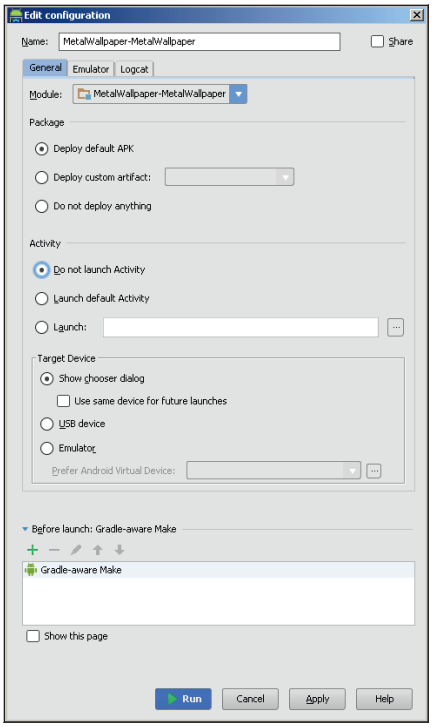

Figura 18.4. Configuración de ejecución

Para hacer más atractivo el fondo de pantalla en movimiento, vamos a añadirle la posibilidad de configurarlo a gusto del usuario. Le daremos la oportunidad de seleccionar diferentes objetos para que se desplacen por la pantalla y que pueda elegir el fondo de pantalla que más le guste.
Para comenzar, necesitaremos indicar que el fondo de pantalla animado tiene una clase para configurarlo; esto se hace desde el archivo de definición del fondo, el `/src/main/res/xml/wallpaper.xml`, donde se le añade el atributo que determina la clase Java que se usará para su configuración. El nuevo aspecto del fichero sería:

```
<?xml version="1.0" encoding="utf-8"?>
<wallpaper
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:thumbnail="@drawable/ball"
    android:description="@string/lwp_description"
    android:settingsActivity="com.acme.metalwallpaper.
WallpaperPreferenceActivity"/>
```

Claro está que necesitaremos la clase que atienda estas preferencias, así pues crearemos la clase llamada `WallpaperPreferenceActivity.java`. Trabajaremos mediante fragmentos, pero en caso de necesidad, comentar que se podría trabajar sin ellos, simplemente haciendo que la actividad creada, en lugar de extender la clase Activity, extendiera la clase PreferenceActivity. Los fragmentos que usaremos serán un poco especiales, en lugar de extender la clase Fragment, extenderán PreferenceFragment. La clase PreferenceFragment como su propio nombre indica es un tipo de fragmento que sirve para mostrar las preferencias de usuario para poder configurar las aplicaciones. Su utilidad radica en que dado un layout (que tiene un formato especial como veremos dentro de muy poco), se utilizan los identificadores de los elementos dispuestos sobre él, como claves de entradas en una SharedPreferences, con lo que es ideal para guardar datos de configuración de una manera rápida y cómoda. La clase encargada de las preferencias quedaría

```
public class WallpaperPreferenceActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        //generar el fragmento
        getFragmentManager().beginTransaction()
                .replace(android.R.id.content, new SettingsFragment())
                .commit();
    }

    public static class SettingsFragment extends PreferenceFragment {
        public static final String SHARED_PREFS_NAME = "BallSettings";
        public static final String PREFERENCE_WP_MODE = "lwp_mode";

        @Override
```

```
        public void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
            getPreferenceManager().setSharedPreferencesName(SHARED_PREFS_
            NAME);
            //añadir layout
            addPreferencesFromResource(R.xml.wallpaper_settings);
        }
    }
}
```

Como se puede ver, en el método `onCreate()` simplemente se genera un fragmento de una clase interna que será quien gestione toda la lógica de las preferencias. Dentro de la clase privada estática, hemos definido dos constantes a nivel de atributos de clase, `SHARED_PREFS_NAME` lo utilizaremos para mantener el nombre del archivo de preferencias y `PREFERENCE_WP_MODE` se utilizará para guardar el dato de que tipo de objeto que quiere mostrar en pantalla en lugar de la bola. Los nombres y los valores son totalmente arbitrarios, así que si el lector no se encuentra cómodo con ellos, tenga toda la libertad de cambiarlos. Dentro del método `onCreate()` del fragmento, las cosas también son un poco distintas a los ejercicios que se han visto anteriormente. La llamada al objeto padre sigue existiendo, a continuación se le indica el nombre del fichero de preferencias sobre el que se quiere trabajar, en el que se guardarán las selecciones del usuario, y por último vemos que ya no está el método `setContentView()`, sino que en su lugar se encuentra `addPreferencesFromResource()`, que hace prácticamente lo mismo (cargar un fichero XML con datos sobre lo que debe mostrar en pantalla) y es que al ser un fragmento del tipo PreferenceFragment, su layout es un poco peculiar. Para definir dicho layout lo primero que haremos es crear el archivo XML llamado `wallpaper_settings.xml` pero en lugar de hacerlo dentro del directorio `/src/main/res/layout`, lo crearemos en `/src/main/res/xml`. Su contenido será:

```
<?xml version="1.0" encoding="UTF-8"?>
<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android"
        android:title="@string/wp_conf_tittle"   android:key="lwp_settings">
    <ListPreference
            android:key="lwp_mode"
            android:title="@string/wp_title"
            android:summary="@string/wp_sum"
            android:entries="@array/wallpaper_piecenames"
            android:entryValues="@array/wallpaper_ballmode" />
</PreferenceScreen>
```

En este archivo se crea una sección llamada `<PreferenceScreen>` donde se definirán cada uno de los objetos a mostrar en las preferencias; acompaña como atributo el título a mostrar cuando el fragmento se exponga en pantalla. Dentro se ha incluido la etiqueta `<ListPreference>`, que sirve

para indicar que se mostrará una lista de elementos del cual se debe seleccionar uno, que será el que se guarde en las preferencias, su sintaxis es muy sencilla, mediante el atributo `android:key` se indica bajo que clave se debe guardar en el fichero de preferencias, el atributo `android:title` sirve para darle un título a la entrada dentro de la pantalla de preferencias, `android:summary` es la descripción de qué hace ese atributo, se muestra debajo del título cuando está en la pantalla de preferencias, `android:entries` se trata de un array de valores a mostrar en la lista y por último `android:entryValues` son los valores correspondientes a las entradas mostradas, es decir, cuando se seleccione la segunda entrada del `android:entries`, lo que se guardará en las propiedades es la segunda entrada de `android:entryValues`.

Durante la definición del fichero de propiedades, se han llamado a una serie de valores cadena que aún no se han creado, modifiquemos el fichero `strings.xml` para añadir estos valores, éste quedaría.

```
<resources>
    <string name="app_name">MetalWallPaper</string>
    <string name="lwp_description">Muestra un objeto de metal que se
    desplaza por la pantalla con el movimiento del dispositivo</string>
    <string name="wp_title">Objeto</string>
    <string name="wp_sum">Selecciona el objero a desplazar por la pantalla</
    string>
    <string name="wp_conf_tittle">Configuración:</string>
    <string-array name="wallpaper_piecenames">
        <item>"Bola"</item>
        <item>"Anillo"</item>
        <item>"Moneda"</item>
    </string-array>
    <string-array name="wallpaper_ballmode">
        <item>0</item>
        <item>1</item>
        <item>2</item>
    </string-array>
</resources>
```

No debemos olvidar que hay que añadir la actividad de preferencias recién creada al `AndroidManifest.xml` de lo contrario el fondo de pantalla animado se detendría al intentar configurarlo:

```
<activity
      android:name="com.acme.metalwallpaper.WallpaperPreferenceActivity"
      android:exported="true"
      android:label="@string/app_name"
      android:theme="@android:style/Theme.Light.WallpaperSettings" />
```

Ya podría probar la aplicación e incluso al seleccionar el fondo de pantalla podría hacer los ajustes necesarios a través de la pantalla de preferencias y guardarlos en el dispositivo... pero la única pega es que aún no los recupe-

ramos para modificar el comportamiento del fondo en movimiento. Retoquemos la clase `MetalBGEngine` (dentro del fichero `WallpaperService.java`) para que sea capar de detectar los cambios en la configuración y reaccionar a ellos.
Haremos que implemente un interfaz que hace que reacciones a los cambios de preferencias modificando la definición de la clase, cambiando:

```
<public class MetalBGEngine extends Engine implements SensorEventListener{
```

por

```
<public class MetalBGEngine extends Engine implements SensorEventListener,
        SharedPreferences.OnSharedPreferenceChangeListener {
```

Añadimos dos atributos a la clase, uno para mantener una referencia a las preferencias y así poder "escuchar" los cambios y otro que mantenga el número de entrada de tipo de bola seleccionado por el usuario en de las preferencias, es decir, el tipo de objeto que se debe mostrar en pantalla.

```
private SharedPreferences mPrefs;
private int mPieceIndex;
```

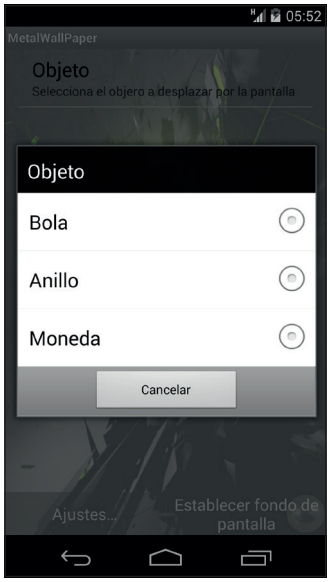

Figura 18.5 Pantalla de configuración del fondo en movimiento

En el constructor del motor, obtendremos la referencia a las preferencias, utilizando el mismo nombre de archivo que cuando se creó en la clase de configuración.

```
MetalBGEngine() {
   mContext = getApplicationContext();
   mPrefs = WallpaperService.this.getSharedPreferences(
   WallpaperPreferenceActivity.SettingsFragment.SHARED_PREFS_NAME, 0);
   mPrefs.registerOnSharedPreferenceChangeListener(this);
   onSharedPreferenceChanged(mPrefs, null);
}
```

Mediante `mPrefs.registerOnSharedPreferenceChangeListener(this)` lo que se hace es registrar la clase actual (la MetalBGEngine) para que escuche los cambios que se puedan realizar en el archivo de preferencias `mPrefs`.

```
public void onSharedPreferenceChanged(SharedPreferences prefs, String key) {
    mPieceIndex =    Integer.parseInt(prefs.getString(WallpaperPreferenceActi
                     vity.SettingsFragment.PREFERENCE_WP_MODE,"0"));
    configBg();
 }
```

Cada vez que se produzca un cambio se ejecutará el método `onSharedPreferenceChanged()` donde se obtiene el número de pieza guardado en las preferencias, que corresponde con el seleccionado en las mismas. Es la manera de tener el fondo de pantalla sincronizado con las preferencias que ha seleccionado el usuario. A la hora de recoger el valor de la clave `WallpaperPreferenceActivity. SettingsFragment.PREFERENCE_WP_MODE`, es posible que éste no exista, por ejemplo en la primera ejecución del fondo, ya que nunca se ha configurado; entonces hay que devolver algún valor por defecto y en este caso se devuelve un "0" que es el índice de la figura correspondiente a la bola. Por último se llama al método `configBg()` para que prepare el objeto a dibujar respecto a lo leído de las preferencias; lo modificamos para que quede:

```
private void configBg() {
   mBackground = BitmapFactory.decodeResource(mContext.getResources(),
         R.drawable.bg);
   int pieceResource;
   switch (mPieceIndex) {
      case 0:
         pieceResource = R.drawable.ball;
         break;
      case 1:
         pieceResource = R.drawable.ring;
         break;
      case 2:
         pieceResource = R.drawable.coin;
         break;
      default:
         pieceResource = R.drawable.ball;
         break;
      }
```

```
        mBall = BitmapFactory.decodeResource(mContext.getResources(),
                pieceResource);
        mPictureHeight = mBall.getHeight();
        mPictureWidth = mBall.getWidth();
        // obtenemos referencias al servicio
        mSensorManager = (SensorManager) mContext
              .getSystemService(Activity.SENSOR_SERVICE);
        // interesa el acelerómetro
        mAccelerometer = mSensorManager
              .getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
        mSensorManager.registerListener(this, mAccelerometer,
              SensorManager.SENSOR_DELAY_GAME);
    }
```

Como puede ver es todo muy parecido a lo que ya se tenía salvo que en este caso dependiendo del valor que se haya seleccionado durante la configuración. Hay que crear también las imágenes correspondientes a las entradas `R.drawable.coin` y `R.drawable.ring`, usado los gráficos que más nos gusten.

Si prueba el fondo en movimiento con la figura de la bola en como objeto a desplazarse por la pantalla, el efecto es bastante realista, pero no así con el anillo o la moneda, ya que al chocar con los bordes, en la vida real, rodarían. Sin querer llegar a simular a la perfección el comportamiento físico de una pieza al rodar por una superficie, sí le vamos a dar un poco más de realismo creando un poco de rotación en el gráfico cuando toque las paredes. Lo pri-mero que se necesitará para ello es mantener el ángulo actual de la figura, y cada vez que toque una pared, dependiendo de la pared y de la posición de los acelerómetros, se calculará y actualizará el grado de giro. Además, para ajustarlo un poco más a la realidad, se usará el valor de la lectura del acelerómetro para que adecúe el giro de la pieza del mismo modo que se adecúa la velocidad de desplazamiento. Cree una nueva variable dentro de la clase MetalBGEngine que guardará los grados de rotación actuales de la pieza:

```
private float mBallDeg = 0;
```

El cálculo de los grados a rotar la figura se realizará en el mismo método que se realiza el cálculo de posiciones, en el método `updateMe()`, al final del mismo método se añadirá la detección de choques con los bordes y adecua-ción de los grados de rotación.

```
//rotación
if (mPieceIndex != 0) {
    int rotationFactor = 2;
    if (cx == 0) {
        // Izquierda
```

```
            if ((iny > 0 && cy < (mHeight - mPictureHeight)) ||(iny < 0 && cy >
            0)){
               mBallDeg += iny * rotationFactor;
            }
         } else if (cy == 0) {
            // Superior
            if ((inx < 0 && cx < (mWidth - mPictureWidth)) || (inx > 0 && cx > 0)) {
               mBallDeg += inx * rotationFactor;
            }
         } else if (cx == (mWidth - mPictureWidth)) {
            // derecha
            if ((iny < 0 && cy > 0) || (iny > 0 && cy < (mHeight -
            mPictureHeight))) {
               mBallDeg -= iny * rotationFactor;
            }
         } else if (cy == (mHeight - mPictureHeight)) {
            // Inferior
            if ((inx > 0 && cx > 0) || (inx < 0 && cx < (mWidth - mPictureWidth))) {
               mBallDeg -= inx * rotationFactor;
            }
         }
      }
```

El cálculo solo se realizará para las figuras que no sean la bola (no quedaría bien el efecto en esta figura), tras ello se da valor a la constante que marcará el factor multiplicador de la lectura del acelerómetro, que hará que cuando más se incline el dispositivo, más rápido gire (ya que más rápido se desplaza la figura). Cada uno de los bordes debe tener en cuenta qué acelerómetros entran en juego, por ejemplo para el caso del borde inferior, se detecta cuando la posición de la bola en el eje Y (la variable `cy`) es igual que el tamaño de la pantalla menos el tamaño de la figura. Una vez que sabemos que está en el borde inferior, se debe girar sólo si hay lectura en el acelerómetro correspondiente al eje X (si está inclinado hacia la izquierda o derecha) y siempre y cuando la posición de la bola (la variable `cy`) no esté fuera de los límites de la pantalla, que viene dado por la coordenada 0 y por el ancho de la pantalla menos el ancho de la figura, si no se tuviera esto en cuenta, la figura rotaría mientras se encontrara apoyada en las esquinas. El último cálculo que hay que realizar es el número de grados a rotar, que viene dado por el acelerómetro tras aplicarle el factor corrector.

Ya se tienen los grados correspondientes a aplicar a la transformación de la imagen, así pues los aplicaremos justo antes de dibujarla. Para ello se debe modificar el método `paintBall()`. Lo primero es rotar la imagen a partir de su centro geométrico, de modo que parezca que rueda, y generar una nueva imagen a partir de esta

```
Matrix matrix = new Matrix();
matrix.setRotate(mBallDeg, mPictureWidth / 2, mBall.getHeight() / 2);
Bitmap targetBitmap1 = Bitmap.createBitmap(mBall, 0, 0,
               mPictureWidth, mPictureHeight, matrix, false);
```

El problema que se plantea ahora, es que la imagen generada tras rotar la original, es mayor de la que necesitamos y el tamaño de ésta, viene dado por el ángulo que se le haya girado, tomando su tamaño máximo cada 45 grados, para verlo mejor, simplemente vale con coger un cuadrado (piense que aunque vea la moneda o el anillo circular, realmente el gráfico es cuadrado, como muestra la figura 18.6) y rotarlo para ver que necesitamos un lienzo mayor para poder pintarlo, y más concretamente y si Pitágoras no nos engaño, el nuevo tamaño máximo del lado de la imagen generada es igual a la raíz cuadrada del la suma de los cuadrados de los lados de la imagen original.

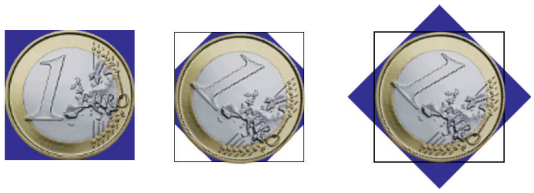

Figura 18.6. Al girar una imagen se modifica el tamaño de la misma

Lo que haremos será volver a transformar la imagen, recortándola y quedándonos con la parte central de la misma, que mida de alto y de ancho como la imagen original, dicho de otra manera recortaremos los bordes que nos sobran para quedarnos con la figura que interesa:

```
float translateH = (targetBitmap1.getHeight() - mPictureHeight) / 2;
float translateW = (targetBitmap1.getWidth() - mPictureWidth) / 2;
matrix = new Matrix();
Bitmap targetBitmap2 = Bitmap.createBitmap(targetBitmap1, (int) translateW,
(int) translateH,    mPictureWidth, mPictureHeight, matrix, false);
```

Ya tenemos el gráfico girado y centrado listo para ser pintado sobre el lienzo que contiene el fondo estático.

```
canvas.drawBitmap(targetBitmap2, cx, cy, null);
```

El método paintBall() completo quedaría:

```
private void paintBall(Canvas canvas) {
   RectF rf = new RectF(0, 0, mWidth, mHeight);
   canvas.drawBitmap(mBackground, null, rf, null);
   // actualiza el giro
   Matrix matrix = new Matrix();
   matrix.setRotate(mBallDeg, mPictureWidth / 2, mBall.getHeight() / 2);
   Bitmap targetBitmap1 = Bitmap.createBitmap(mBall, 0, 0,
         mPictureWidth, mPictureHeight, matrix, false);
   float translateH = (targetBitmap1.getHeight() - mPictureHeight) / 2;
   float translateW = (targetBitmap1.getWidth() - mPictureWidth) / 2;
   matrix = new Matrix();
```

```
    Bitmap targetBitmap2 = Bitmap.createBitmap(targetBitmap1, (int)
    translateW, (int) translateH,          mPictureWidth,     mPictureHeight,
    matrix, false);
    canvas.drawBitmap(targetBitmap2, cx, cy, null);
}
```

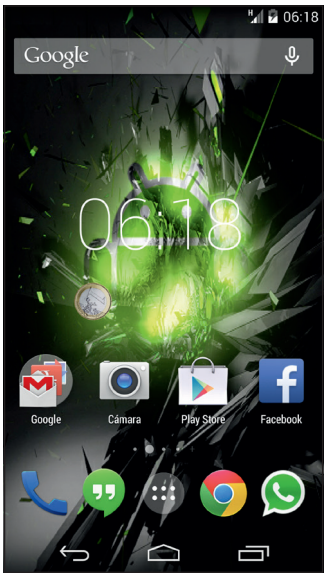

Figura 18.7. Fondo de pantalla en movimiento con la moneda





19 Wearables

En este capítulo aprenderá a:

- Descubrir los wearables.
- Crear interfaces para wearables.
- Configurar el entorno para trabajar con el emulador.
- Conocer los fundamentos de las aplicaciones para estos dispositivos.

Quizá debamos comenzar este capítulo comentando a qué se denomina dispositivos wearables; este concepto hace referencia a los dispositivos electrónicos que pueden incorporarse en alguna parte de nuestro cuerpo de modo que interactúe con el usuario y/o con otros dispositivos de modo que puedan realizar su función. Dentro de estos dispositivos podemos colocar a los relojes inteligentes, pulseras, bandas deportivas de monitorización, zapatillas de deporte con GPS y cuenta pasos…
Claro está que estos dispositivos tienen pantallas mucho menores que los teléfonos y las tabletas y es por ello que Google ha preparado una serie de interfaces adecuados tanto a la usabilidad de la pantalla como a su tamaño (hoy por hoy no es factible escribir un correo electrónico con un teclado en la pantalla de un reloj). Todo lo referente a esta tecnología es muy reciente y es posible que las API de las llamadas y las opciones tanto de configuración como de manejo cambien, aunque este capítulo nos servirá para comprender cómo funciona y conocer su fundamento.

El modelo

Debido al tamaño de la pantalla, los dispositivos wearables están pensados para den información precisa en el momento preciso, evitando lo máximo que el usuario interactúe con la interfaz. Cuando diseñemos las aplicaciones debemos pensar en éstas de modo inteligente, dando la máxima información necesaria (a ser posible adelantándose a las necesidades del usuario) pero de manera concisa para que quepa en la pantalla de reducidas dimensiones; esto sin olvidar que se debe mantener la interactuación con la pantalla lo menor posible.

Las tarjetas

El modelo de trabajo en los wearables cambia un poco a lo que estamos acostumbrados en las aplicaciones vistas hasta el momento. Como se ha comentado anteriormente, basaremos la funcionalidad en una serie de tarjetas y el modelo en este caso se basa en dos tipos de tarjetas que definen su función: las sugeridas y las demandadas

Tarjetas sugeridas

Denominadas Context Stream (o flujo de contexto). Consiste en una serie de tarjetas (semejantes a las de Google Now) entre las cuales se puede navegar mediante desplazamientos verticales que contienen información relevante al usuario (notificaciones, información del tiempo, resultados deportivos…).

Las aplicaciones pueden incorporar nuevas tarjetas al Context Stream para informar de datos relevantes en ese momento. Lejos de ser una simple notificación, estar tarjetas pueden ser deslizadas horizontalmente de derecha a izquierda para mostrar nuevas pantallas, por ejemplo con botones para tomar decisiones o con información adicional. Para descartar las tarjetas vale con deslizarlas de izquierda a derecha.

Tarjetas demandas

Es posible que en ocasiones el Context Stream no sea capaz de anticiparse a lo que el usuario quiere conocer en ese momento, para ello, podemos demandar la tarjeta de acciones pulsando sobre la "g" de la pantalla inicial o pronunciando la frase mágica "Ok Google". La tarjeta de acciones es una lista de operaciones que podemos seleccionar simplemente pulsando sobre ellas; entre éstas se encuentran las acciones por voz (entre las que podemos registrar las nuestras propias), que lo que harán será enviar un Intent de modo que la aplicación lo pueda atender.

La interfaz

Las notificaciones en los dispositivos wearables aparecen en forma de tarjetas dentro del Context Stream, es por esto que nuestras aplicaciones deben emitir dichas notificaciones sólo cuando realmente deba hacerse con tal de no molestar al usuario y que éste acabe desinstalando nuestra aplicación; del mismo modo, las notificaciones activas (aquellas que producen vibración) deben ser utilizadas sólo en caso de que se requiera atención inmediata por parte del usuario. Una vez más teniendo en cuenta el tamaño de la pantalla, la notificación sólo debe mostrar la información necesaria, es decir mejor usar palabras, frases cortas o imágenes descriptivas en lugar de grandes párrafos. Si la información no cupiera, se mostraría truncada y pulsando en el elemento se abría para mostrarlo completamente.

Cuando se genera la notificación, se le puede aportar un nivel de prioridad que sirve para indicar el nivel de urgencia; sólo las que dependen de la hora (por ejemplo una cita de reunión) deben llevar prioridad alta.

Tras las tarjetas normalmente se muestran imágenes de modo que añadan atractivo a la información; el contenido de las imágenes debe ser acorde con la información mostrada, por ejemplo, si mostráramos el tiempo de una ciudad, la imagen podría ser de un monumento típico de esa ciudad o si se avisa del cumpleaños de una persona, mostrar la imagen de su perfil. Las imágenes no se muestran completas, sino que son cubiertas parcialmente por la infor-

mación publicada, concretamente, es la parte inferior la que queda oculta. Las imágenes tienen que ser de al menos 320×320 pixeles en `hdpi`, teniendo en cuenta que cuando se desliza la tarjeta hacia la izquierda, el fondo se desplaza, las imágenes que son de tipo horizontal son las que mejor quedan.

Para que el usuario sea capaz de identificar qué aplicación ha lanzado la notificación, el sistema mostrará automáticamente el icono de la aplicación en la propia tarjeta, dejando el espacio principal para la información que se quiere transmitir. En caso de no querer que se muestre, se puede llamar al método `setHintHideIcon()`.

A las notificaciones les pueden acompañar comandos o acciones que actuarán sobre ella, aunque no es obligatorio. En caso de existir, las acciones aparecen a la derecha de la notificación (es decir hay que deslizar de derecha a izquierda para acceder a ellas) pudiendo tener hasta tres acciones distintas. La acción más común se debe situar en primer lugar para acceder a ella tan sólo haciendo un deslizamiento y haciéndola así más accesible.

Visualmente, las acciones se componen de una etiqueta descriptiva que debe contener un verbo indicando la acción y no ocupar más de una línea (de lo contrario se truncará) y un icono que se recomienda que sea de tamaño 64x64dip de tipo PNG de color blanco con fondo transparente.

Además de las acciones, a la derecha de la notificación principal, podemos añadir nuevas páginas (pages) para mostrar información añadida. En ocasiones encontraremos que el espacio para mostrar la información es insuficiente o que puede completarse con datos adicionales. Supongamos por ejemplo que al pasar cerca de la farmacia nos salta una alerta, la notificación principal podría avisar que hay que comprar y la página siguiente (deslizando la tarjeta hacia la izquierda) podría mostrar la lista de la compra, o si somos seguidores de un equipo y se muestra el resultado de un partido, en la siguiente página podríamos mostrar sus próximos rivales y fechas.

Se pueden añadir tantas páginas como se quiera (o ninguna) pero si además de páginas tenemos acciones, se recomienda no añadir más de tres, de lo contrario las acciones quedarían lejos de la mano del usuario y sería incómodo su uso.

Si una aplicación tiene múltiples notificaciones (por ejemplo recepción de un correo electrónico), en lugar de generar una tarjeta para cada notificación y saturar el Context Stream, lo que se debe hacer es apilar todas estas notificaciones en una sola tarjeta. Cuando se muestren en el dispositivo wearable, el usuario puede seleccionar cada elemento de la pila y expandirlo para acceder a su contenido. Tanto las acciones como las páginas adicionales pueden depender directamente de la naturaleza de la entrada expandida y serán accesibles, del modo ya conocido, una vez se seleccione la entrada de la pila.

En ocasiones necesitaremos proporcionar una respuesta como acción a una notificación, por ejemplo si nos salta la alarma de una reunión podemos tener la opción de posponerla, aceptarla, rechazarla... Para este caso, podemos utilizar comandos de voz (que deben ser lo más escuetos posible) aunque a esta posibilidad, se recomienda acompañarla de una serie de respuestas preconcebidas (hasta cinco diferentes) para el caso en el que el usuario no pueda o quiera usar el comando de voz, por ejemplo si ya se está en una reunión, no quedaría bien ponerse a hablar con el reloj.

Notificaciones

Añadir nuevas tarjetas al Context Stream es más sencillo de lo que se puede suponer, simplemente debemos crear notificaciones en el teléfono o tableta al que esté conectado el dispositivo wearable, para poder ver en éste dichas notificaciones; así de simple.

Con esto, si emparejáramos ahora un dispositivo wearable Android con un dispositivo en el que se estuvieran ejecutando el widget que se creó en el capítulo 15, ya seríamos capaces de ver en el wearable la notificación que se produce al alcanzar el tiempo configurado en el widget.

Esta sería la integración más rápida y sencilla, aunque como hemos visto, son varias las opciones que podemos aplicar a la visualización de las notificaciones en el wearable. Crearemos una aplicación a modo de prueba para generar dichas notificaciones y jugar un poco con las opciones disponibles.

Antes de proseguir con la aplicación, vamos a preparar el entorno para poder trabajar. Comentar que a día de hoy (cuando se escriben estas líneas) no existen dispositivos wearables físicos en el mercado, por lo que tendremos que trabajar con el emulador que Google nos proporciona para este tipo de dispositivos.

Cuando se trabaje con dispositivos físicos simplemente habrá que emparejarlos mediante Bluetooth, pero para trabajar con el emulador la forma es la siguiente: lo primero que se ha de hacer es descargar su imagen mediante el SDK Manager, como lo hicimos con las imágenes de los emuladores de los dispositivos. En este caso se debe seleccionar la entrada Android Wear ARM EABI v7a System Image.

Figura 9.1. Descarga de la imagen para emulador.

Una vez descargada, hay que crear el emulador del mismo modo que lo hicimos anteriormente, desde el `AVD Manager`, sólo que en el desplegable `Device`, nos debemos asegurar de seleccionar Android Wear Round si queremos un emulador con pantalla redonda o Android Wear Square si lo queremos con ella cuadrada. Al acabar ejecute el emulador recién creado. Una vez en marcha el emulador no es muy útil sin la ayuda de un dispositivo que esté emparejado con él, vamos a configurarlo.
En el dispositivo físico que usemos, por ejemplo un teléfono, debemos instalar la aplicación `Android Wear` (Preview o no, dependiendo de la versión), disponible en el Market. Al ejecutar esta aplicación aparecerá una pantalla diciendo que está como no conectado (Not connected), esto es porque no está emparejado con el dispositivo wearable aún. Conectamos mediante USB el dispositivo al ordenador y desde la línea de comando ejecutamos:

```
adb -d forward tcp:5601 tcp:5601
```

Con ello haremos que el emulador de wearable y el dispositivo físico puedan ya comunicarse, en ese momento la aplicación del dispositivo físico aparecerá como conectada y el emulador mostrará la hora y la g típica de Google.

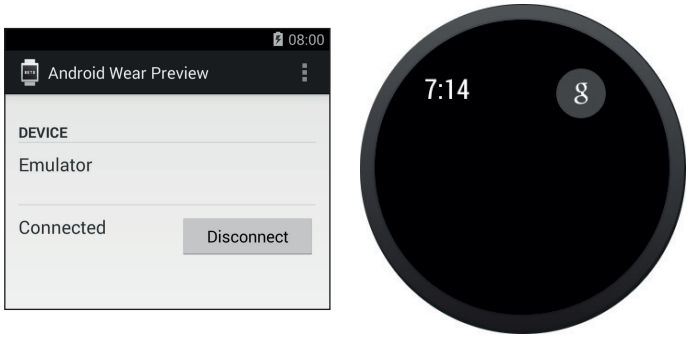

Figura 19.2. Emulador conectado.

A partir de este momento el emulador ya es capaz de recibir todas las notificaciones del dispositivo físico, por ejemplo si usáramos los el widget anteriormente creado o si recibiéramos un email (véase figura 19.3).
Ahora que ya lo tenemos todo preparado, podemos crear nuestra aplicación. Generaremos un nuevo proyecto con la configuración:

- Application name: Wearable
- Module Name: Wearable
- Package name: com.acme.wearable

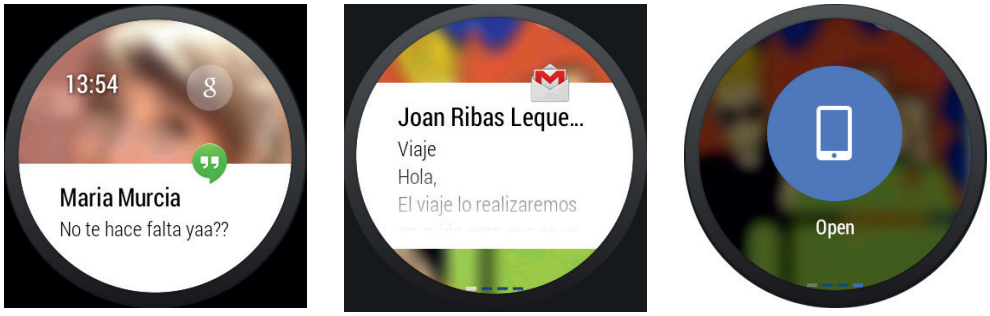

Figura 19.3. Recepción notificaciones y acción.

Para realizar las pruebas utilizaremos una pantalla con un texto y un botón, cuyo código iremos modificando. Hay que recordar que las notificaciones están pensadas para ser enviadas en fondo, es decir cuando la aplicación no está presente (si queremos mostrar alertas cuando la aplicación está presente se usan diálogos o tostadas); no obstante y por facilidad, lanzaremos las notificaciones pulsando el botón. El layout quedaría.

```
<LinearLayout 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:orientation="vertical"    android:padding="@dimen/activity_
    horizontal_margin"
    tools:context="com.acme.wearable.MainActivity">
    <TextView
        android:id="@+id/textView"
        android:layout_width="wrap_content" android:layout_height="wrap_
        content"
        android:text="Texto" />
    <EditText
        android:id="@+id/editText"
        android:layout_width="match_parent" android:layout_height="wrap_
        content"
        android:layout_alignParentStart="true"
        android:layout_below="@+id/textView" />
    <Button
        android:id="@+id/button"
        android:layout_width="wrap_content" android:layout_height="wrap_
        content"
        android:layout_alignParentStart="true" android:layout_below="@+id/
        editText"
        android:onClick="sendNotification" android:text="Enviar" />
</LinearLayout>
```

Podemos trabajar con las notificaciones normales, pero para poder sacar todo el partido a las notificaciones de los wearables, necesitamos incorporar a nuestro proyecto la librería de soporte, llamada actualmente `wearable-preview-`

`support.jar` y que por el momento está solo accesible descargándola de la página oficial de Google bajo invitación, pero que estará disponible en un futuro a través del SDK Manager. Para añadirla a nuestro proyecto se debe crear una carpeta `libs` en el directorio de nuestro módulo `Wearable` y poner el fichero `jar` anterior en ella. El proyecto se debe configurar para que use esa librería modificando el archivo `build.gradle` del módulo añadiendo la dependencia:

```
dependencies {
   compile 'com.android.support:support-v4:18.0.+'
   compile files('../libs/wearable-preview-support.jar')
   compile fileTree(dir: 'libs', include: ['*.jar'])
}
```

En cuanto al código, en la clase MainActivity debemos simplemente implementar el método:

```
public void sendNotification (View v){}
```

Advertencia:

Si no podemos acceder a la librería de soporte wearable o queremos usar los métodos estándar, no podremos acceder a toda la funcionalidad pero sí a parte de ella. En este caso, hay algunas clases que cambiarán el nombre, por ejemplo en la librería de wearable usaremos NotificationManagerCompat *y en el modo estándar es la* NotificationManager, *pero el mecanismo de emisión de notificaciones es el mismo.*

Comenzaremos ahora a añadir código a este método para emitir las notificaciones. En lugar de optar por utilizar el asistente como ya hicimos, lo haremos a mano.

```
public void sendNotification (View v){
   int notificationId = 001;
   //Intent a ejecutar en la pulsación
   Intent viewIntent = new Intent(this, MainActivity.class);
   viewIntent.putExtra("AlarmId", 2);
   PendingIntent viewPendingIntent =
   PendingIntent.getActivity(this, 0, viewIntent, 0);
   //configuración de la notificación
   NotificationCompat.Builder notificationBuilder =
               new NotificationCompat.Builder(this)
                       .setSmallIcon(R.drawable.ic_launcher)
                       .setContentTitle("Test Notificaciones")
                       .setContentText(((EditText)findViewById(R.
                        id.editText)).getText())
                       .setContentIntent(viewPendingIntent);
```

```
    // obtención del servicio
    NotificationManagerCompat notificationManager =
                NotificationManagerCompat.from(this);

    // notificación.
    notificationManager.notify(notificationId, notificationBuilder.build());
}
```

En este ejemplo lo primero que creamos es un Intent que se ejecutará cuando el usuario pulse en la alerta del dispositivo móvil, pudiendo así abrir la aplicación que emitió esta notificación (acordémonos que normalmente las notificaciones se lanzan no estando la aplicación activa). Esto nos permite que al activarse la actividad indicada en el Intent (en este caso hemos puesto la misma que lanza la notificación MainActivity, pero se puede poner cualquiera), tengamos disponibles los datos pasados en el mismo y actuar en consecuencia; en el ejemplo se pasa el parámetro "`AlarmId`" con valor 2, entonces en el `onCreate()` del la actividad extraeríamos los datos para atenderlos mediante `getIntent().getExtras()`, donde tendríamos un Bundle con los datos pasados como parámetro. En el caso del wearable, para abrir el Intent indicado, debemos deslizar la tarjeta hacia la izquierda para que muestre la opción de abrir y pulsando sobre ella se abriría la actividad en el móvil.

Volviendo de nuevo al código, después se configura la notificación en sí; en este caso no es muy compleja y simplemente se muestra el contenido indicado en la caja de texto de la pantalla un título puesto a mano y el icono de la aplicación. Por último se obtiene el servicio de notificaciones y se emite la notificación. Ya podríamos probar la aplicación tanto en el dispositivo móvil como en el wearable (véase figura 19.4).

Si quisiéramos añadir alguna acción más a parte de la realizada mediante `setContentIntent()`, podemos utilizar el método `addAction()` al que se le pasarían como parámetros el icono a mostrar, el texto para la etiqueta y un objeto de tipo PendingIntent con los datos necesarios para que la actividad lo atienda cuando se pulse:

Figura 19.4. Notificación.

```
Intent mapIntent = new Intent(Intent.ACTION_VIEW);
Uri tel = Uri.parse("tel://9839988123");
mapIntent.setData(tel);
PendingIntent mapPendingIntent =
PendingIntent.getActivity(this, 0, mapIntent, 0);

notificationBuilder.addAction(R.drawable.ic_phone,
                "Llamar Tlf.", mapPendingIntent);
```

Para que la notificación quede un poco más bonita, podemos hacer que el texto que se muestra en ella sea mayor de lo normal mediante la llamada a `setStyle()`, del mismo modo también podemos poner un fondo a la notificación utilizando el método `setLargeIcon()`. Para la elección de la imagen a utilizar en el fondo hay que tener en cuenta que será parcialmente ocultado por el texto de la notificación.

```
...
//configuración de la notificación
NotificationCompat.BigTextStyle style = new NotificationCompat.
BigTextStyle();
style.bigText(((EditText) findViewById(R.id.editText)).getText());
NotificationCompat.Builder notificationBuilder =
new NotificationCompat.Builder(this)
    .setSmallIcon(R.drawable.ic_launcher)
    .setContentTitle("Test Notificaciones")
    .setContentIntent(viewPendingIntent)
              .setStyle(style )
    .setLargeIcon(BitmapFactory.decodeResource(getResources(), R.drawable.
     backgnd));
//acción
Intent mapIntent = new Intent(Intent.ACTION_VIEW);
Uri tel = Uri.parse("tel://9839988123");
mapIntent.setData(tel);
PendingIntent mapPendingIntent = PendingIntent.getActivity(this, 0,
mapIntent, 0);
 notificationBuilder.addAction(R.drawable.ic_phone,
            "Llamar Tlf.", mapPendingIntent);
// obtención del servicio
...
```

Figura 19.5. Notificación con fondo y acción.

Para añadir nuevas páginas a nuestra notificación a la derecha de la página principal (que recordemos se acceden deslizando la tarjeta hacia la izquierda), simplemente tenemos que llamar al método `addPage()` de la notificación con la nueva página. Podremos añadir tantas páginas como queramos, siempre sin perder de vista que la información más relevante debe ir en las primeras páginas y que las acciones se colocan tras las páginas adicionales, es decir cuantas más páginas, mas deslizamientos deberemos realizar para llegar a las acciones.

Para la prueba vamos a añadir una segunda caja de texto en el layout de la aplicación que tenga como identificador `android:id="@+id/editText2"` y la usaremos para indicar el texto de la segunda página de la notificación. Es algo que ya sabemos hacer por lo que no se mostrará el código del layout y el del método con todo lo visto hasta ahora quedaría:

```
public void sendNotification (View v){
   int notificationId = 001;
   //Intent a ejecutar en la pulsación
   Intent viewIntent = new Intent(this, MainActivity.class);
   viewIntent.putExtra("AlarmId", 2);
   PendingIntent viewPendingIntent =
   PendingIntent.getActivity(this, 0, viewIntent, 0);
   //configuración de la notificación
   NotificationCompat.BigTextStyle style = new NotificationCompat.
   BigTextStyle();
   style.bigText(((EditText) findViewById(R.id.editText)).getText());
   NotificationCompat.Builder notificationBuilder =
      new NotificationCompat.Builder(this)
      .setSmallIcon(R.drawable.ic_launcher)
      .setContentTitle("Página 1")
      .setContentIntent(viewPendingIntent)
      .setStyle(style )
      .setLargeIcon(BitmapFactory.decodeResource(getResources(), R.drawable.
       backgnd));

   Intent mapIntent = new Intent(Intent.ACTION_VIEW);
   Uri tel = Uri.parse("tel://9839988123");
   mapIntent.setData(tel);
   PendingIntent mapPendingIntent =
   PendingIntent.getActivity(this, 0, mapIntent, 0);

   notificationBuilder.addAction(R.drawable.ic_phone, "Llamar Tlf.",
   mapPendingIntent);

   //segunda página
   Notification secondPage =  new NotificationCompat.Builder(this)
      .setContentTitle("Página 2")
      .setContentText(((EditText) findViewById(R.id.editText2)).getText())
      .build();
    Notification completeNotification =
                new WearableNotifications.Builder(notificationBuilder)
                        .addPage(secondPage)
                        .build();
```

```
    // obtención del servicio
    NotificationManagerCompat notificationManager =
    NotificationManagerCompat.from(this);
    // notificación.
    notificationManager.notify(notificationId, completeNotification);
}
```

Como habrá podido comprobar el lector se están usando cadenas de texto como literales de las etiquetas; en una aplicación real deberíamos utilizar igualmente el archivo strings.xml, en este caso lo estamos haciendo así para facilitar la explicación.

Si recibimos más de una notificación de la misma aplicación, es posible que queramos mostrarlas todas en el dispositivo wearable para que se tenga constancia de todas, el ejemplo más claro son los emails o conversaciones con una persona. En lugar de generar una entrada en el Context Stream, lo que haremos será agruparlas en una tarjeta de modo que sea el usuario quien despliegue la que le interese, haciendo así una doble tarea, por un lado poneos a disposición del usuario todos los avisos y por otro lado no polucionamos el Context Stream con un montón de tarjetas. Todo esto se consigue mediante el método `setGroup()`, a quien se le pasa el identificador de grupo donde se debe incluir la nueva notificación, que será una cadena de texto (normalmente mantenida como constante) y una posición opcional, en caso de no informarse la posición, la nueva notificación se colocará en la primera posición dentro de la pila.

```
Notification completeNotification = new WearableNotifications.
Builder(builder)
    .addPage(secondPage)
    .setGroup("groupId", position)
    .build();
```

Para obtener también agrupadas las notificaciones en el dispositivo, lo que se hace es además de añadir cada notificación al grupo, se crea una a modo de resumen y que debe indicarse con la posición `WearableNotifications.GROUP_ORDER_SUMMARY`:

```
Notification summaryNotification = new WearableNotifications.
Builder(builder)
.setGroup("groupId", WearableNotifications.GROUP_ORDER_SUMMARY)
.build();
```

Habíamos comentado que las notificaciones podían llevar asociadas una serie de acciones entre las cuales el usuario podía elegir. A veces esto no es suficiente, a veces necesitamos mayor flexibilidad porque por ejemplo las acciones a realizar dependan de la hora del envío de la notificación, porque tenemos una serie de respuestas preconcebidas, por ejemplo de respuesta a

un email o simplemente porque es incómodo tener que realizar muchos deslizamientos hasta llegar a todas las acciones. Para ello podemos ayudarnos de las acciones por voz. Para trabajar con los comandos por voz simplemente tenemos que añadir a la notificación mediante el método `addRemoteInputForContentIntent()` un objeto de la clase RemoteInput. Este objeto puede tener definida una etiqueta indicando lo que se espera en la entrada de voz y hasta cinco opciones distintas a modo de cadena de texto. Estas opciones servirán para que en caso de estar en un lugar donde no podamos usar la voz, al menos podamos seleccionar alguna respuesta.

```
RemoteInput remoteInput = new RemoteInput.Builder("ExtraVoice")
      .setLabel("Elige cena")
      .setChoices(new String[]{"Carne","Pescado","Sopa"})
      .build()
```

Por supuesto, el array de opciones puede ser dinámico o crearlo como un array dentro del fichero `strings.xml`, aquí se ha informado directamente por comodidad.

Nota:

Actualmente en el emulador no funciona el reconocimiento de voz, por lo que para probar las acciones de voz se debe utilizar el teclado del ordenador y para ello el checkbox Hardware keyboard present *debe estar cuando realizamos la máquina virtual del emulador.*

Para probar esta parte modificaremos un poco más el código para que quede claro cómo reaccionaría la aplicación.
En primer lugar, vamos a variar el código del método `onCreate()` para leer la respuesta que vendrá desde el wearable y la mostraremos en caso de que haya sida proporcionada.

```
protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Bundle extras = getIntent().getExtras();
        if (extras != null){
            String voiceResponse = extras.getString("ExtraVoice");
            new AlertDialog.Builder(this)
                    .setTitle("Respuesta")
                    .setMessage(voiceResponse)
                    .show();
        }
    }
```

En el método `sendNotification()` crearemos el RemoteInput y lo asignaremos a la notificación. Además aprovecharemos para salir de la aplicación en el último paso del método y así poder probar mejor la aplicación.

```
...
//segunda página
Notification secondPage =
   new NotificationCompat.Builder(this).setContentTitle("Página 2")
   .setContentText(((EditText) findViewById(R.id.editText2)).getText())
   .build();
//Comando por voz
RemoteInput remoteInput = new RemoteInput.Builder("ExtraVoice")
   .setLabel("Respuestas")
   .setChoices(new String[]{"Carne","Pescado","Sopa"}).build();

Notification completeNotification =
   new WearableNotifications.Builder(notificationBuilder).addPage(secondPage)
   .addRemoteInputForContentIntent(remoteInput).build();
// obtención del servicio
NotificationManagerCompat notificationManager =
   NotificationManagerCompat.from(this);
// notificación.
notificationManager.notify(notificationId, completeNotification);
//salimos
onBackPressed();
}
```

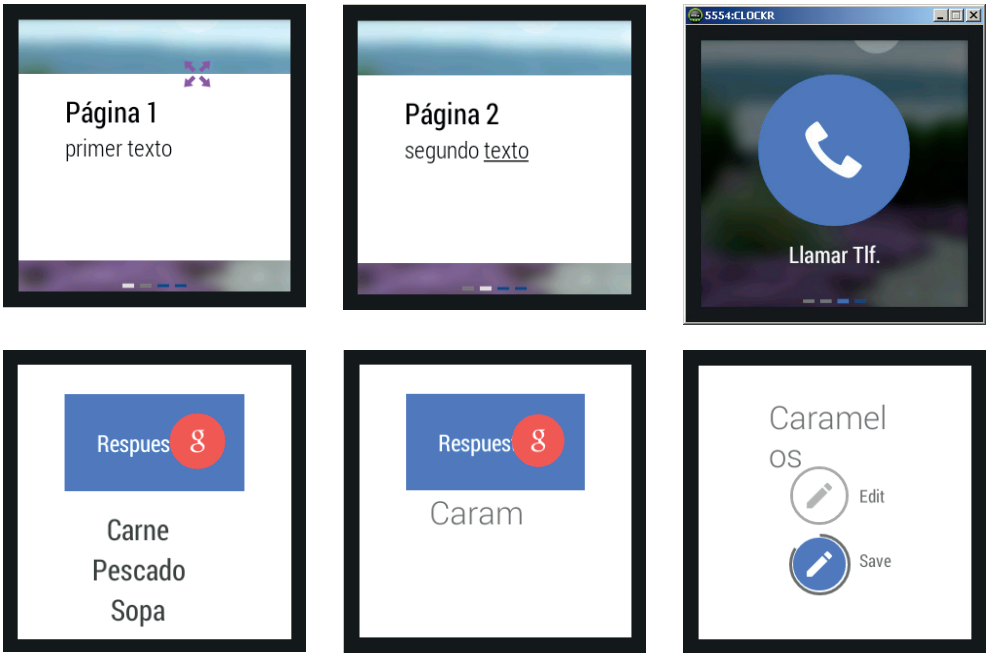

Figura 19.6. Secuencia de tarjetas en wearable cuadrado.

Ahora si probamos la aplicación, tendremos todo igual con las dos páginas conteniendo la información introducida en cada uno de los campos, la acción de llamada y la acción de responder (Reply) que es la que nos lleva a la respuesta por voz con todas sus acciones. Si seleccionamos esta última acción será cuando se nos dé la opción de seleccionar por voz o pulsando alguna de las opciones mostradas. Si se quiere emular la opción por voz, debemos teclear lo que quisiéramos decir, entonces (que no tiene porque ser una de las opciones mostradas), veremos que sale una nueva pantalla indicando si se quiere editar o no lo que ha entendido el dispositivo. Si se envía, veremos que en el dispositivo móvil la aplicación se ha vuelto a lanzar con un mensaje indicando lo seleccionado desde el wearable. Claro está que esto es sólo un ejemplo y lo mostramos en pantalla; en una aplicación real habría que discernir qué realizar con la opción seleccionada.





20 Miscelánea

En este capítulo aprenderá a:

- Comprender el funcionamiento de la barra de acción.
- Controlar los elementos de los menús.
- Usar pestañas en las aplicaciones.
- Compartir de manera estándar datos.

A lo largo del libro se han ido tratando diferentes aspectos de la programación de aplicaciones para Android, más o menos agrupados por conceptos y temas. En este capítulo se verán un par de conceptos que bien no encajaban el algunos capítulos o bien hacían que otros fueran muy largos, pero que no obstante me parecen interesantes de explicar.

Action Bar

La barra de acción o Action Bar es una barra que se encuentra en la parte superior de las pantallas en las versiones superiores a la 3.0 y en las versiones inferiores mediante el paquete de soporte. Se trata de un widget que puede ser utilizado en las Activity y que principalmente está pensado para sustituir la típica barra de título de las pantallas a la vez que darle una funcionalidad mayor, mediante la inclusión de elementos que permitan la interactuación con ellos. Su funcionalidad la podríamos describir como:

- Una zona donde indicar al usuario en qué parte de la aplicación se encuentra, a modo de título de la ventana.
- Permitir la navegación e intercambio de vistas mediante pestañas y desplegables
- Dar acceso rápido a las acciones más comunes.

Cuando trabajemos con ella, hay que diferenciar si se trabaja con versiones de API inferiores a la 11 (Android 3.0) que habrá que utilizar la clase del paquete de compatibilidad `android.support.v7.app.ActionBar` y cuando se trabaja con versiones superiores se debe trabajar con la `import android.app.ActionBar`; que pese a tener el mismo nombre, no son la misma clase.

Figura 20.1. Action Bar de lector de correos.

En una Activity, la barra de acción por defecto está compuesta por el logo de la aplicación a la izquierda, seguido del título de la actividad (dado mediante programación o mediante el archivo `AndroidManifest.xml`) y otro tipo de entradas como pueden ser pestañas o entradas disponibles en el menú de opciones. El menú de opciones para estas aplicaciones se realiza del mismo modo que se ha realizado para las aplicaciones de Android 2.x, bien mediante programación o bien mediante archivo XML y sobrescribiendo el método

`onCreateOptionsMenu()`, pero en este caso, el usuario para poder desplegarlo, tendrá que pulsar sobre un icono que aparecerá a la derecha de la barra de acción  o  .
Además de lo visto, la barra de acción puede contener muchos otros elementos, como por ejemplo:

- Pestañas para navegar entra distintos Fragment.
- Listas desplegables.
- Elementos del menú de opciones, que en lugar de mostrarse contraídos en bajo el icono situado a la derecha de la barra, se muestren directamente sobre ella (lo cual es muy útil para acceder rápidamente a las opciones más utilizadas). A estos elementos se les denomina action items (elementos de acción).
- Vistas completas, denominadas action views (vistas de acción), como pueden ser cuadros de búsqueda semejantes al que se muestra en la figura 20.1.

La barra se muestra por defecto en todas las aplicaciones de superiores a Android 3.0, es decir si se ha incluido la etiqueta

```
<uses-sdk android:minSdkVersion="11" />
```

en el archivo `AndroidManifest.xml`, si se ha indicado en la creación del proyecto que la versión mínima de API es la 11 o se ha añadido la entrada correspondiente en la configuración de Gradle (que realmente lo que hace en todas es añadir la etiqueta anterior al `AndroidManifest.xml`).
Puede ser que en alguna ocasión no interese tener la barra en ciertas pantallas, como podría ser en caso de que se quiera mostrar una imagen en pantalla completa; por ello existen dos modos de ocultar la barra de acción, uno en tiempo de ejecución y otro en tiempo de diseño. En tiempo de ejecución se realiza dinámicamente mediante programación llamando a su método `hide()` tras haber obtenido su instancia a través de `getActionBar()`.

```
ActionBar actionBar = getActionBar();
actionBar.hide();
```

Y en cualquier momento se puede volver a mostrar usando el método `show()`.

```
actionBar.hide();
```

La manera de ocultar la barra en tiempo de diseño es a través del fichero `AndroidManifest.xml` en la definición de la actividad. Realmente lo que se hace en este archivo no es ocultar la barra en sí, sino indicarle que utilice un tema de estilo que no tiene la barra de acción definida. La definición de la actividad sería

```
<Listados<activity … android:theme="@android:style/Theme.Holo.NoActionBar">
```

Veamos cómo sacarle el mayor partido posible al Action Bar incluyendo en ella diferentes elementos que permitan al usuario interactuar con ellos y con la actividad en los que se haya definida. Realizaremos un proyecto donde se irán mostrando las distintas posibilidades de configuración de la barra. Cree un nuevo proyecto con las siguientes propiedades:

- Application name: ActionBar
- Module Name: ActionBar
- Package name: com.acme.actionbar
- Minimum Required SDK: API 14 Android 4.0 (IceCreamSandwich) o superior
- Activity Name: ActionBarTest

Si nada más crear el proyecto ejecuta la aplicación, podrá ver lo que se ha comentado anteriormente, que por defecto encontramos la barra de acción, mostrando el icono por defecto de la actividad (que en este caso coincide con el de la aplicación) y el título de la misma.

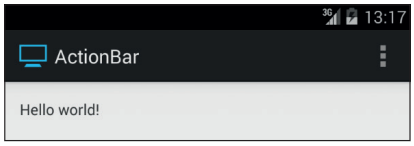

Figura 20.2. Action Bar por defecto la aplicación.

Añadir elementos

La manera más rápida (y ya conocida por el lector) de añadir elementos a la barra es a través del menú de opciones y una gran noticia es que todo lo comentado en capítulos anteriores donde se generaban los menús es válido. Pongamos un pequeño menú a la aplicación. Cree el archivo XML llamado `action_bar_test.xml` en `/src/main/res/menu` o aproveche el existente creado por el asistente y añada cuatro elementos configurados con icono y texto. En nuestro caso podría ser:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android" >
    <item android:id="@+id/edit"
        android:icon="@android:drawable/ic_menu_edit" android:title="@
        string/edit"/>
    <item android:id="@+id/delete"
        android:icon="@android:drawable/ic_menu_delete" android:title="@
        string/delete"/>
    <item android:id="@+id/save"
        android:icon="@android:drawable/ic_menu_save" android:title="@
```

```
        string/save"/>
    <item android:id="@+id/search"
        android:icon="@android:drawable/ic_menu_search" android:title="@
        string/search"/>
</menu>
```

Una vez se tiene definido el archivo con los componentes que formarán parte del menú, se debe sobrescribir el método `onCreateOptionsMenu()` de la actividad ActionBarTest para inflar dicho menú; compruebe que el asistente lo ha hecho correctamente.

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

Añada las cadenas de texto necesarias en el fichero `strings.xml`:

```
<string name="edit">Editar</string>
    <string name="delete">Borrar</string>
    <string name="save">Guardar</string>
    <string name="search">Buscar</string>
```

Y por último habría que dotar de código a la selección de cada uno de los elementos del menú, sobrescribiendo el método `onOptionsItemSelected()`, pero en este caso no vamos a hacer nada en la selección, así que de momento elimínelo del código o comente su contenido (de lo contrario no compilará el proyecto), aunque la discriminación de los elementos y su gestión se haría del mismo modo que se ha realizado en ejercicios anteriores. Si ejecuta la aplicación, podrá observar que a la derecha se dispone de un icono; pulsando sobre él, aparecerán las opciones de menú insertadas en el fichero `action_bar_test.xml`.

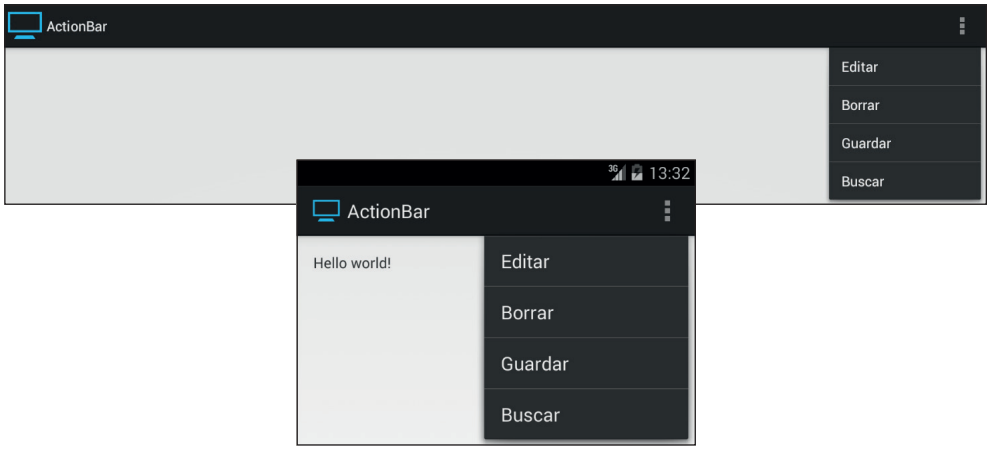

Figura 20.3. Action Bar con el menú desplegado.

Ocultar el Action Bar

Para probar a eliminar la barra puede realizarlo directamente sobre el archivo `AndroidManifest.xml` añadiendo la propiedad `android:theme="@android:style/Theme.Holo.NoActionBar"` a la línea:

```
<activity android:name=".ActionBarTest"
                  android:label="@string/app_name">
```

Pero esto es poco flexible, así que pasemos a ver como se haría dinámicamente por programación. Añada al layout un ToggleButton que utilizaremos para activar y desactivar la barra.

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:orientation="vertical"
   android:layout_width="fill_parent"
   android:layout_height="fill_parent"
   android:background="#777777">
<ToggleButton
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:textOn="@string/hide"
      android:textOff="@string/show"
   android:id="@+id/showHide"/>
</LinearLayout>
```

No hay que olvidar añadir las cadenas de texto necesarias en el archivo `strings.xml`.

```
<string name="hide">Barra oculta</string>
<string name="show">Barra mostrada</string>
```

En la clase ActionBarTest hay que asegurarse que extiende la clase Activity en lugar de la clase ActionBarActivity, ya que si la clase extendida es la ActionBarActivity, estaríamos trabajando con las clases del paquete de compatibilidad. En el método `onCreate()` de la clase ActionBarTest debemos asignar el código para que cuando esté activo se oculte la barra y cuando se desactive se vuelva a mostrar:

```
@Override
public void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_action_bar_test);
      // Activar o desactivar la barra
final ToggleButton showHide = (ToggleButton) findViewById(R.id.showHide);
      showHide.setOnClickListener(new View.OnClickListener(){
      @Override
      public void onClick(View v) {
         ActionBar actionBar = getActionBar();
         if (showHide.isChecked()){
            actionBar.hide();
         }
```

```
            else{
               actionBar.show();
            }
        }
    });
}
```

Pulsando ahora sobre el ToggleButton conseguiremos ir mostrando y escondiendo la barra de acción.

Añadir Action Items

Los Action Items eran los elementos a modo de botones o iconos que podemos encontrarnos en la barra de acción. Si nos fijamos bien, ya hay un icono en ella… efectivamente, el icono de la actividad (o de la aplicación si no se ha especificado ninguno para la actividad en el `AndroidManifest.xml`).
La pulsación sobre este icono se trata como si fuera una pulsación sobre cualquier elemento del menú de opciones, es decir a través del método `onOptionsItemSelected()`. Recordará que para saber el elemento pulsado en este método, lo que se hacía era `switch()` sobre el identificador que había producido el evento, que se obtenía a través del método `getItemId()` del objeto MenuItem pasado como parámetro. El identificador enviado tras una pulsación sobre el icono de la actividad es la constante `android.R.id.home`, por lo que para probar las pulsaciones sobre él podríamos implementar:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
switch (item.getItemId()) {
         case android.R.id.home:
               Toast.makeText(this, "Icono de actividad pulsado!", Toast.
               LENGTH_LONG).show();
                return true;
            default:
                return super.onOptionsItemSelected(item);
}
}
```

Google especifica que el comportamiento por norma general, cuando se pulsa sobre este icono, es que la aplicación ha de volver a su estado original, a su actividad principal, como si hubiera sido recién lanzada. La manera de hacerlo es implementar en el método anterior una llamada con un Intent que tenga como destino la clase correspondiente a la Activity principal de la aplicación. El problema de hacer esto, es que casi con toda seguridad, la actividad principal ya se encuentre en la pila de las Activity que se han ejecutado en la aplicación, es decir aquellas a las que volveríamos si pulsáramos sobre el botón de "volver atrás". La manera de hacer que no se incluya como una más en la pila

sino que comience de nuevo, es informando al objeto Intent que se va a lanzar el flag (la bandera) `FLAG_ACTIVITY_CLEAR_TOP`. Este flag lo que hace es indicar al sistema que revise si ya existe la Activity que se quiere ejecutar dentro de la pila de actividades realizadas y en caso de existir, elimina de la pila todas aquellas que se encuentren por encima de ella y quedándose como última Activity ejecutada, si traducimos todo esto a programación y lo adaptamos a nuestro ejemplo, el resultado del código para el método `onOptionsItemSelected()` sería que en la clase principal deberíamos añadir el método:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
switch (item.getItemId()) {
         case android.R.id.home:
      Intent intent = new Intent(this, ActionBarTest.class);
      intent.addFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP);
                  startActivity(intent);
              return true;
            default:
              return super.onOptionsItemSelected(item);
}
}
```

Pero lo normal es que nos interese añadir otro tipo de elementos de acción, como por ejemplo la acción de guardar, para tenerla a mano y no tener que ir hasta el menú, desplegarlo y seleccionarla de allí.
Para poder tener un acceso rápido a los elementos del menú, simplemente se debe variar su definición y añadirles dentro de la etiqueta `<ítem>` correspondiente, el atributo `android:showAsAction="ifRoom"`, que hará que se muestre (si hay sitio para él) en la barra de acción. En caso de que no hubiera sitio suficiente para mostrarlo, se encontraría accesible dentro del menú desplegable, como cualquier otro elemento más del menú.

Advertencia:

Aunque es posible forzar a que un elemento del menú aparezca como un elemento de acción siempre, no es aconsejable hacerlo ya que el estado de la barra puede cambiar o puede que se usen pantallas pequeñas y entonces no haber sitio para todos los elementos. A modo informativo, el valor del atributo para conseguirlo es `android:showAsAction="always"`.

Por defecto, los action ítem no se muestran con el texto asignado en el menú, para que se muestre tanto el icono como el texto, se debe configurar así en el archivo XML, asignándole al atributo el valor `android:showAsAction="`

`ifRoom|withText"`. Si se quiere tener la opción salvar disponible en la barra de acción, se debe modificar el archivo de definición de menú de modo que el elemento `<ítem>` correspondiente a salvar quedaría:

```
<item android:id="@+id/save"
android:icon="@drawable/save"
android:title="@string/save" android:showAsAction="ifRoom|withText"/>
```

Como se puede ver en la figura 20.4, al no ser igual el tamaño de visualización de una tableta y el de un teléfono, es posible que no se muestren los mismos elementos de acción en ambas, y es que así lo hemos definido al decir que se mostraran si hay espacio, así pues, si no caben en la Action bar, se quedan en modo menú.
Puede ser que en ocasiones interese mostrar u ocultar dinámicamente los elementos de acción. La manera de hacerlo es muy sencilla:

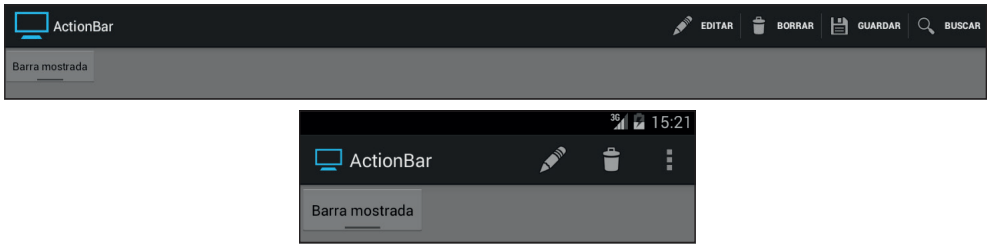

Figura 20.4. Action Bar con elementos de acción.

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
MenuInflater inflater = getMenuInflater();
inflater.inflate(R.menu.menu, menu);
MenuItem item = menu.findItem(R.id.copy);
   item.setShowAsAction(MenuItem.SHOW_AS_ACTION_IF_ROOM);    return true;
}
```

Como podrá observar, del mismo modo que mediante configuración XML, por defecto sólo se muestra el icono, no el texto. Para que se muestre el texto también, puede utilizar el flag `SHOW_AS_ACTION_WITH_TEXT` a la hora de llamar al método `setShowAsAction()`.

Añadir pestañas

Otro elemento interesante que se puede añadir a la barra de acción son las pestañas. Mediante el uso de pestañas y Fragments, es posible crear actividades perfectamente ordenadas a la vez que amigables al usuario.

La manera de trabajar con las pestañas en la barra de acción, es crear una clase que recibirá los eventos de las pestañas y se encargará de ir mostrando el Fragment que corresponda en cada momento. Esta clase debe implementar la interfaz `ActionBar.TabListener`, que a su vez obliga a la implementación de los métodos:

- onTabSelected(): Se produce cuando una pestaña se selecciona. Se debe registrar el Fragment y añadirlo al FragmentTransaction para poder navegar después por ellos.
- onTabUnselected(): Se produce cuando una pestaña deja de estar seleccionada. Lo normal es eliminar su Fragment del FragmentTransaction .
- onTabReselected (): Se produce cuando una pestaña ya está seleccionada por el usuario, pero éste vuelve a seleccionarla. En un principio no se tendría que hacer nada en referencia al FragmentTransaction.

Dentro de estos métodos se reciben como parámetros dos objetos, uno de tipo ActionBar.Tab que contiene información referente a la pestaña que ha recibido el evento y otro de tipo FragmentTransaction donde se podrán añadir y eliminar los Fragment para poder navegar por ellos en cualquier momento.
Para ver un ejemplo de funcionamiento vamos a añadir a la aplicación dos pestañas que no harán nada pero que servirán al lector para entender los pasos a realizar a la hora de trabajar con pestañas en la barra de acción.
Lo primero es crear una clase listener de tipo ActionBar.TabListener. Nosotros en este ejemplo no vamos a añadir ningún tipo de lógica en ella (aunque se deberían registrar al menos las transacciones entre distintos *Fragment*), simplemente vamos a implementar los métodos necesarios para poder mostrar las pestañas en pantalla. Dentro de la clase *ActionBarTest*, implemente la clase privada *TabListener* que se encargará de gestionar los eventos recibidos por cada pestaña dentro de la clase principal de la aplicación:

```
private class TabListener implements ActionBar.TabListener{
  public TabListener(Fragment fragment) {
     // TODO Auto-generated constructor stub
  }
  @Override
  public void onTabReselected(ActionBar.Tab tab, FragmentTransaction ft) {
     // TODO Auto-generated method stub
  }
  @Override
  public void onTabSelected(ActionBar.Tab tab, FragmentTransaction ft) {
     // TODO Auto-generated method stub
  }
  @Override
  public void onTabUnselected(ActionBar.Tab tab, FragmentTransaction ft) {
     // TODO Auto-generated method stub
  }
}
```

Dentro del método `onCreate()` configuraremos la barra para que pueda almacenar pestañas como contenido mediante su método `setNavigationMode()` y después crearemos, configuraremos y añadiremos las pestañas a la barra.
Seguido del código existente en el método `onCreate()` añada las líneas necesarias para obtener la barra de acción y configurarla para que acepte las pestañas:

```
// Preparación de la barra de acción para poder ser usada como pestañas
final ActionBar actionBar = getActionBar();
actionBar.setNavigationMode(ActionBar.NAVIGATION_MODE_TABS);
```

Tras ello creamos las instancias de los Fragment a utilizar. En caso de una aplicación normal, se debería crear una clase que extendiera la clase Fragment con la lógica necesaria para su funcionamiento y un layout que lo acompañara. Nosotros en este caso simplemente usaremos la clase Fragment de modo directo. Una vez tenido el objeto Fragment, se añade una nueva pestaña a la barra de acción mediante `actionBar.addTab(actionBar.newTab())`, pero hay que añadir también un identificador de la pestaña para que el usuario sepa dónde está pulsando; estos identificadores pueden ser textos y/o iconos.
Para este ejemplo y ya que no vamos a añadirle nada de contenido a los Fragment, al menos vamos a dejar un poco vistosas las pestañas, para lo cual añadiremos un texto y un icono mediante sendos métodos `setText()` y `setIcon()` del objeto Tab creado. El código resultante para las dos pestañas sería:

```
// Instancia del Fragment1
      Fragment firstFragment = new Fragment();
      // Añadir el tab configurando etiqueta, icono y listener
      actionBar.addTab(actionBar.newTab().setText(R.string.edit)
            .setIcon(android.R.drawable.ic_menu_edit)
            .setTabListener(new TabListener(firstFragment)));

      // Instancia del Fragment2
      Fragment secondFragment = new Fragment();
      // Añadir el tab configurando etiqueta, icono y listener
      actionBar.addTab(actionBar.newTab().setText(R.string.delete)
            .setIcon(android.R.drawable.ic_menu_delete)
            .setTabListener(new TabListener(secondFragment)));
```

El resultado puede verse en la figura 20.5.
En el caso de que tuviéramos múltiples elementos visibles en la barra de acción, es posible que el nombre de la actividad moleste; en cualquier momento se puede ocultar mediante la llamada `setDisplayShowTitleEnabled(false)` del objeto ActionBar.

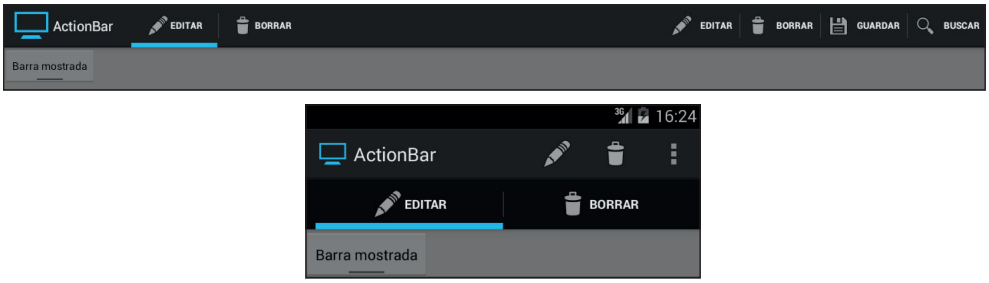

Figura 20.5. Action Bar mostrando pestañas.

Compartiendo información simple

Imaginemos la siguiente situación; tenemos que realizar una aplicación que muestre las noticias relacionadas con un colegio, donde además de simplemente texto, se incorporan de vez en cuando fotografías de los alumnos. Cuando los padres vayan recibiendo noticias y sobre todo si aparece en alguna foto sus hijos, querrán rápidamente compartirla con la familia y amigos por correo, o por Twitter, o por Google+ o ... vaya usted a saber qué nueva red social saldrá en un año, a la que iremos todos como zombis. Son muchas las opciones que tenemos hoy en día para compartir información, no todos usamos las mismas (gracias a Dios) y nuevas surgen cada día; es imposible codificar algo usando la API de cada una de ellas. Android ofrece un mecanismo estandarizado para comunicar datos entre aplicaciones y poder así compartir información a través del medio que más nos guste.

Envío

Realmente este mecanismo no difiere en exceso de lo visto hasta ahora, es decir, transmitir información entre aplicaciones a través de Intent. La llamada a la actividad se realizará de forma implícita, indicando que la acción a realizar es `Intent.ACTION_SEND`, detallando el tipo de dato que se envía y por supuesto enviando también el dato que se quiere pasar a la nueva actividad. Por ejemplo, para transmitir un texto sería:

```
 Intent intent = new Intent();
intent.setAction(Intent.ACTION_SEND);
intent.setType("text/plain");
intent.putExtra(Intent.EXTRA_TEXT, "Texto para enviar muy lejos.");
startActivity(intent);
```

En este momento el sistema se encargaría de mirar cual de las aplicaciones se ha registrado para atender texto plano (tipo `text/plain`) y lanzaría su ejecución, encargándose desde ese momento de procesar los datos enviados. Si hubiera más de una posibilidad , se nos mostraría un selector para elegir con cuál de las aplicaciones queremos completar la acción.
El problema que se nos puede plantear en este caso, es que el usuario haya definido ya una aplicación para manejar textos y siempre se dirija a esa aplicación en concreto, no pudiendo usar el resto, o que no tenga ninguna aplicación instalada que cumpla con las especificaciones del Intent, perdiéndose la petición... para estos casos, podemos forzar la aparición de este selector, pudiendo así mostrar todas las opciones posibles, indicar un texto para el título del selector y además, si no hubiera ninguna aplicación capaz de atender la aplicación, nos los indicaría.
Para probar estas opciones, variaremos el programa para añadir un par de botones de prueba. El layout existente, lo modificamos de la siguiente manera:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:background="#777777"
    android:orientation="vertical">
<ToggleButton
        android:id="@+id/showHide"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textOff="@string/show"
        android:textOn="@string/hide" />

<LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:background="#007878">
   <EditText
   android:id="@+id/editText"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:ems="10"
   android:inputType="textMultiLine" />
   <Button
   android:id="@+id/button2"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:onClick="sendText"
   android:text="Enviar texto" />
    </LinearLayout>

<LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginTop="20dip"
        android:background="#789078">
```

```
    <ImageView
    android:id="@+id/imageView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:src="@drawable/ic_launcher" />
    <Button
    android:id="@+id/button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:onClick="sendImage"
    android:text="Enviar  Imagen" />
</LinearLayout>
</LinearLayout>
```

En este caso usaremos otra manera de informar los eventos `onClick()` de los botones; hasta ahora hemos indicado las acciones a realizar en las pulsaciones de los botones desde el código Java, obteniendo una referencia del objeto en pantalla mediante `findViewById()` y asignando el listener mediante `setOnClickListener()`. En este caso, en el XML hemos añadido el atributo `android:onClick` a cada uno de los botones. Este atributo tiene el nombre de un método en la clase que utilice esta vista y este método tiene que tener como parámetro una vista. Con estos datos vamos a implementarlos en la clase ActionBarTest, añadiendo los métodos:

```
public void sendText(View v){
}

public void sendImage(View v){
}
```

Ahora lo que se codifique dentro de cada uno de estos métodos será lo que se ejecute en la pulsación de cada uno de los botones.
Para conseguir que se muestre el selector de aplicaciones con las que terminar la acción del Intent, debemos llamar al método `Intent.createChooser()`, donde le tenemos que pasar como parámetros, el Intent que se quiere atender y una cadena de texto que hará las funciones de título del selector. Completaremos el método `sendText()` con:

```
Intent intent = new Intent();
intent.setAction(Intent.ACTION_SEND);
intent.setType("text/plain");
String text = ((EditText)findViewById(R.id.editText)).getText().toString();
intent.putExtra(Intent.EXTRA_TEXT,text);
startActivity(Intent.createChooser(intent, "Seleccione opción"));
```

Sencillo ¿no?
En cuanto a la imagen se refiere, debemos tener en cuenta que el tipo de dato a enviar es `image/jpeg`, por lo que el método `sendImage()` sería (véase figura 20.6):

```
Intent intent = new Intent();
intent.setAction(Intent.ACTION_SEND);
intent.setType("image/jpeg");
Uri path = Uri.parse("android.resource://com.acme.actionbar /" + R.drawable.
ic_launcher);
intent.putExtra(Intent.EXTRA_STREAM, path);
startActivity(Intent.createChooser(intent, "Seleccione opción"));
```

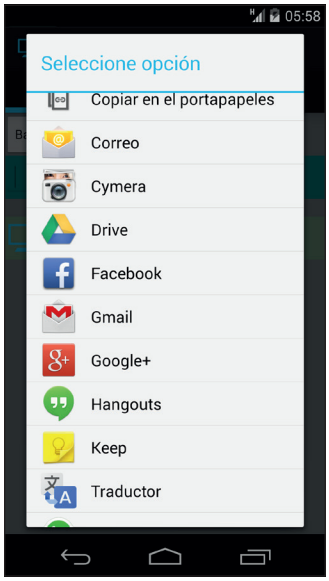

Figura 20.6. Selector de aplicación para compartir datos.

En este caso en lugar de recuperar la imagen de la vista, lo que hacemos es recuperar la Uri (Uniform Resource Identifier, Identificador de recurso uniforme) para acceder a ella. Esto es así, porque vamos a hacer que una aplicación externa acceda a un dato que está en nuestra aplicación y la manera que tiene de trabajar es esta. La Uri de la imagen se crea mediante una cadena con `android.resource://`, seguido del nombre del paquete de la aplicación y el identificador del recurso.

Claro está que aquí hemos puesto los parámetros justos en el Intent para que pueda ser procesado, pero por ejemplo si se va a enviar por mail, se podrían informar otros como por ejemplo

```
intent.putExtra(android.intent.extra.SUBJECT,"Tema del mail");
```

Si se quisiera enviar más de un objeto a la vez, se haría de modo semejante, salvo que la acción a informar sería `Intent.ACTION_SEND_MULTIPLE` y los datos se informarían mediante un ArrayList, por ejemplo:

```
ArrayList<Uri> imageList = new ArrayList<Uri>();
// se obtienen las Uri de las imágenes
...
// se envían
intent.setAction(Intent.ACTION_SEND_MULTIPLE);
intent.putParcelableArrayListExtra(Intent.EXTRA_STREAM, imageList);
```

Dado que en este tema se ha tratado la barra ActionBar, vamos a ver un mecanismo para implementar lo visto anteriormente de manera más sencilla aún. A partir de la versión 4.0 (API 14) podemos utilizar un ActionProvider para usar en nuestra barra. El ActionProvider es un botón que se puede incrustar en la barra de modo que se integra en ella y simplemente tenemos que encargarnos de suministrar el Intent a compartir; veamos cómo.
Modificaremos el fichero de menú para añadir la entrada de la acción. Añadimos el elemento:

```
<item
        android:id="@+id/share"
        android:showAsAction="ifRoom"
        android:title="Compartir"
        android:actionProviderClass=
            "android.widget.ShareActionProvider" />
```

En él, podemos ver el atributo nuevo `android:actionProviderClass`, donde se le informa qué clase atenderá la acción (aunque aquí estemos usando un ActionProvider genérico, podemos crear los nuestros propios) .
En la clase ActionBarTest, crearemos una propiedad para mantener el provider:

```
private ShareActionProvider mShareActionProvider;
```

En el método `onCreateOptionsMenu()`, obtenemos el ActionProvider del elemento de menú `R.id.share` y asignamos el Intent que se debe enviar cuando se pulse sobre el botón de la barra.

```
public boolean onCreateOptionsMenu(Menu menu) {
   getMenuInflater().inflate(R.menu.action_bar_test, menu);
   MenuItem item = menu.findItem(R.id.share);
   mShareActionProvider = (ShareActionProvider) item.getActionProvider();
   setShareIntent()
   return true;
}
```

En el método `setShareIntent()` simplemente se debe indicar al ActionProvider el Intent que se quiere enviar.

```
if (mShareActionProvider != null) {
   Intent intent = new Intent();
   intent.setAction(Intent.ACTION_SEND);
   intent.setType("text/plain");
   String text = ((EditText)findViewById(R.id.editText)).getText().toString();
   intent.putExtra(Intent.EXTRA_TEXT,text);
   mShareActionProvider.setShareIntent(intent);
}
```

En este ejemplo hemos colocado la llamada al método `setShareIntent()` tras la creación del menú, de modo que se llamará una sola vez, pero por ejemplo podríamos tener una aplicación de textos y videos, y dependiendo del elemento marcado en cada ocasión, pues se llamaría a un método semejante al visto para configurar el Intent para el envío de texto o vídeo según el caso.

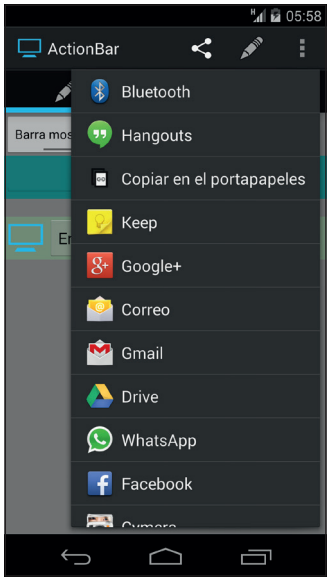

Figura 20.7. Compartición de datos desde un ActionProvider.

Recepción

Del mismo modo que desde nuestra aplicación enviamos datos hacia otras, podemos necesitar recibir datos. Supongamos que nuestra aplicación permite a los padres escribir en un foro y compartir noticias; sería bueno que nuestra aplicación apareciera en el selector de aplicaciones para compartir datos y de esta forma que cualquier aplicación fuera capaz de interactuar con la nuestra. Para ello debemos registrar la actividad que atenderá la petición con los filtros `android.intent.action.SEND` y/o `android.intent.action.SEND_MULTIPLE` dependiendo de las opciones soportadas por nuestra aplicación; el registro, como no podría ser de otra forma, se realiza en el `AndroidManifest.xml`. Por ejemplo para que nuestra actividad fuera capaz de atender envíos de textos, imágenes e incluso múltiples imágenes, la entrada de la actividad quedaría:

```
<activity  android:name="com.acme.actionbar.ActionBarTest"
           android:label="@string/app_name" >
   <intent-filter>
      <action android:name="android.intent.action.MAIN" />
      <category android:name="android.intent.category.LAUNCHER" />
   </intent-filter>
   <intent-filter>
      <action android:name="android.intent.action.SEND" />
      <category android:name="android.intent.category.DEFAULT" />
      <data android:mimeType="text/plain" />
   </intent-filter>
   <intent-filter>
      <action android:name="android.intent.action.SEND" />
      <category android:name="android.intent.category.DEFAULT" />
      <data android:mimeType="image/*" />
   </intent-filter>
   <intent-filter>
      <action android:name="android.intent.action.SEND_MULTIPLE" />
      <category android:name="android.intent.category.DEFAULT" />
      <data android:mimeType="image/*" />
   </intent-filter>
</activity>
```

Una vez registrada la actividad para que aparezca en el selector, debemos preparar el código para ser capaz de recibir y procesar los datos enviados por la aplicación que la llame. Este proceso se realiza como unos pasos más dentro del método `onCreate()` de la actividad registrada. En él se debe obtener el Intent que ha producido el lanzamiento de al Activity, y mirar qué acción y qué tipo de dato lleva asociado y actuar en consecuencia con los datos anexados.

```
// obtención de la acción y tipo de datos
Intent intent = getIntent();
String action = intent.getAction();
String type = intent.getType();
//proceso dependiendo de la acción y tipo de dato
if (Intent.ACTION_SEND.equals(action) && type != null) {
   if ("text/plain".equals(type)) {
      //proceso del texto
                  processText(intent.getStringExtra(Intent.EXTRA_TEXT));
   } else if (type.startsWith("image/")) {
      //proceso de la imagen
      processImage(intent.getParcelableExtra(Intent.EXTRA_STREAM));
   }
} else if (Intent.ACTION_SEND_MULTIPLE.equals(action) && type != null) {
   if (type.startsWith("image/")) {
      //proceso de las imágenes
      processMultipleImages(intent.getParcelableArrayListExtra(Intent.EXTRA_
STREAM));
   }
} else {
   //otras acciones
}
...
```

Y ya sólo quedaría implementar cada uno de los tres métodos que realmente procesan los datos:

```
private void processText (String text) {
...
}

private void processImage (Parcelable image) {
...
}

private void processMultipleImages(ArrayList<Parcelable> imageList) {
...
}
```





21 Mejorando el aspecto

En este capítulo aprenderá a:

- Animar objetos de pantalla.
- Combinar efectos para que se produzcan de modo coordinado.
- Crear animaciones a partir de distintos gráficos.
- Cambiar el aspecto de las pantallas.
- Generar estilos reutilizables a lo largo de la aplicación.
- Usar temas personalizados.

El diseño de pantallas que se ha hecho hasta ahora ha sido aprovechando los estilos proporcionados por la plataforma Android, pero aunque en las últimas versiones y gracias al tema *Holo*, el diseño de los elementos de pantalla está mucho más cuidado, seguramente el desarrollador quiera darle un aspecto más personal y añadirle detalles para hacerlo más atractivo o incluso añadir efectos de transición entre elementos que hagan la aplicación mucho más amigable al usuario. En esta unidad nos centraremos en conocer cómo realizar animaciones y cambiar el aspecto de los componentes de pantalla.

Animaciones

Como parte de las mejoras que se han introducido a lo largo de las versiones Android ha sido una mejora de la capacidad de interactuar con el usuario, creando nuevas APIs para ayudar a los programadores a implementar de modo más sencillo las animaciones; tanto es así, que gracias a ellas es posible animar cualquier objeto de pantalla para conseguir interfaces amigables con efectos impactantes que hagan que el usuario se sienta a gusto y disfrute de la aplicación. Desde Android 4.1 y gracias al proyecto *Butter*, las animaciones han sido aún más refinadas, de modo que las transiciones son mucho más suaves y agradables.
Cuando se habla de animar un objeto de pantalla, lo que se quiere decir es que vamos a poder modificar su posición, aspecto, tamaño… en relación a un tiempo determinado. Mediante la API de animación se tiene control de diferentes aspectos:

- Tipo de animación (rotación, traslación…)
- Duración y tiempos de espera entre animaciones
- Número de repeticiones
- Uso de interpoladores
- Uso de conjuntos de animaciones para crear animaciones más complejas
- …

Cualquier atributo de los objetos de pantalla que esté representado por un tipo de dato `int`, `float` o `hexadecimal`, puede ser tomado como parte de una animación, como el tamaño o la rotación, que son de tipo `float` y pueden ser animados sin dificultad y crear efectos muy llamativos. Para otro tipo de animaciones como podrían ser los textos de una etiqueta, deberíamos basarnos en la interfaz *TypeEvaluator*, donde se debería implementar el

método `evaluate(float fraction, Object startValue, Object endValue) ()` con tal de devolver el objeto deseado dependiendo del valor tomado por los parámetros.

La manera de realizar las animaciones de los objetos se basa en la clase *ValueAnimator*, quien es capaz de realizar cálculos de variables teniendo en cuenta el tiempo transcurrido; por ejemplo mediante el método `setIntValues()` se le puede informar la lista de valores enteros entre los que tiene que estar la animación, a través de `setDuration()` darle la duración total de la animación y la clase ya calculará cada cuanto tiempo debe actualizar los valores, usando por defecto el interpolador *AccelerateDecelerateInterpolator*.

Para aquellas animaciones en las que el interpolador por defecto no se ajuste, podemos cambiarlo usando el método `setInterpolator()` y utilizar uno propio. El problema de utilizar esta clase para realizar la animación de los objetos de pantalla es que hay que implementar a mano toda la lógica propia de la animación, es decir, simplemente realiza los cálculos, pero somos nosotros quienes tenemos que implementar una escucha para detectar los cambios y procesar los valores obtenidos para animar el objeto.

Para ayudarnos a estas tareas, Android ofrece una clase que no sólo hace los cálculos del mismo modo que la clase *ValueAnimator* sino que también permite indicarle el objeto y propiedad a animar. Esta clase es la *ObjectAnimator* (que es una subclase directa de *ValueAnimator*) y que más adelante utilizaremos en un ejemplo para comprender mejor su uso. Para conseguir una animación mediante *ObjectAnimator* solamente hay que indicarle el objeto a animar, la propiedad concreta que recibirá la animación, los valores que puede tomar la propiedad y comenzar la animación, así sin más; la clase se encargará de realizar los cálculos de los valores a entregar a la propiedad a lo largo del tiempo y asignarlos. Claro está que se pueden informar otros muchos parámetros con el fin de conseguir la animación esperada.

Otra clase que ayuda en gran medida a la realización de interfaces muy vistosas en cuanto a efectos visuales de transición es *LayoutTransition*. Con el uso de esta clase es posible habilitar animaciones automáticas dentro de los elementos de un *ViewGroup* cada vez que se realice un cambio en su *layout*, por ejemplo que se eliminen elementos y que haya que reordenar el *layout*. Para utilizarla, se debe llamar al método `setLayoutTransition()` del objeto *ViewGroup* al que se le quiera añadir la animación, y pasarle como parámetro un objeto *LayoutTransition*, donde se puede dejar las animaciones por defecto o definir cada uno los comportamientos de éstas mediante el método `setAnimator()`.

Dentro de los *ViewGroup* se pueden producir dos tipos de cambios que son los que dispararán el evento que iniciará la animación correspondiente. El primero de los cambios en el *layout* es que se añadan nuevos elementos

mientras que el otro cambio posibles es (como habrá podido adivinar) que se eliminen elementos. Realmente no hace falta crear o eliminar el objeto como tal, sino que el hecho de cambiar la visibilidad por ejemplo de `GONE` a `VISIBLE` también disparará el evento para la reproducción de la animación. Estos dos tipos de cambio llevan consigo cuatro tipos distintos de animaciones:

- La que se aplica al elemento que aparece.
- La que se aplica al elemento que desaparece.
- La que se aplica al resto de elementos que ya estaban en el *layout* cuando un elemento aparece.
- La que se realizará sobre los elementos del *layout* que permanecen cuando un elemento desaparece.

Cuando se configuran las animaciones a través del objeto *LayoutTransition*, lo que se hace es crear una animación patrón que será aplicada a cada elemento del *layout*, pero modificando ciertos parámetros, es decir se están informando datos como la duración o el tiempo que debe tardar en iniciarse la animación, pero cuando se aplica a cada elemento del *ViewGroup*, se realiza una copia de esta animación y se ajustan dinámicamente otros valores de modo automático, por ejemplo el elemento sobre el que se va a aplicar la animación o los valores de inicio y fin de ésta para que cuadre con cada elemento. Cada animación es una copia que se modifica dependiendo del objeto sobre el que se va a aplicar y el tipo de animación configurada.

Advertencia:

No se debe utilizar este tipo de animaciones en elementos que estén incluidos en una vista con desplazamiento, ya que si mientras se están realizando las animaciones se produce un desplazamiento, es muy probable que la colocación de los elementos no quede de modo correcto, porque las posiciones finales se calculan al iniciar la animación, si se desplaza el contenedor, las posiciones también habrán variado, pero no en la animación.

Para probar las animaciones crearemos un pequeño proyecto donde haremos aparecer y desaparecer botones a la vez que les aplicaremos una animación. La aplicación consistirá en una línea de siete botones que desaparecerán al pulsar sobre ellos, generando ciertas animaciones. Para la gestión de las animaciones pondremos un botón para controlar si queremos que reproduzcan

las animaciones o no, otro para controlar si queremos que se oculte el botón o queremos que desaparezca del *layout* y una barra de desplazamiento para indicar el tiempo que debe invertir el sistema en realizar las animaciones. Por último también pondremos un botón para poder hacer aparecer de nuevo a todos los botones. Cree un nuevo proyecto con las siguientes propiedades:

- Application name: Animaciones
- Module Name: Animaciones
- Package name: com.acme.animations
- Minimum Required SDK: API 11: Android 3.0 (Honeycomb)
- Activity name: Animations

Modifique el fichero de *layout* con tal de satisfacer las necesidades que se han explicado anteriormente. El contenido podría ser:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:orientation="vertical" android:layout_width="fill_parent"
   android:layout_height="fill_parent">
<LinearLayout android:orientation="vertical"
android:layout_width="fill_parent" android:layout_height="wrap_content"
      android:id="@+id/parent" >
      <Button android:layout_width="wrap_content"
android:layout_height="wrap_content" android:text="@string/buttonGeneration"
         android:id="@+id/add" />
      <LinearLayout xmlns:android="http://schemas.android.com/apk/res/
      android"
android:orientation="horizontal" android:layout_width="fill_parent"
      android:layout_height="wrap_content">

      <ToggleButton android:layout_width="wrap_content"
android:layout_height="wrap_content" android:id="@+id/animation"
android:textOff="@string/noAnimation" android:textOn="@string/animation" />

      <ToggleButton android:layout_width="wrap_content"
android:layout_height="wrap_content" android:id="@+id/invGone"
android:textOff="@string/hide" android:textOn="@string/gone" />
      <LinearLayout xmlns:android="http://schemas.android.com/apk/res/
      android"
android:orientation="vertical" android:layout_width="fill_parent"
android:layout_height="wrap_content" android:paddingLeft="10dp">
      <TextView android:layout_width="wrap_content"
android:layout_height="wrap_content" android:id="@+id/duration" />

      <SeekBar android:layout_width="fill_parent"
android:layout_height="wrap_content" android:id="@+id/animLength"
      android:max="10000" android:progress="500" />
      </LinearLayout>
      </LinearLayout>
</LinearLayout>
</LinearLayout>
```

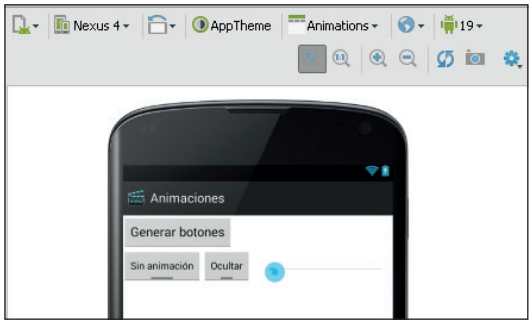

Figura 21.1. Aspecto de la pantalla en edición.

Para la selección de el tipo de visibilidad que aplicaremos, si `GONE` o `INVISIBLE`, al igual que para la selección de si se quieren utilizar animaciones o no, se usan un par de botones del tipo *ToggleButton*; estos botones tienen la característica de una vez pulsados, mantenerse seleccionados hasta que se vuelve a pulsar sobre ellos. Como tienen dos estados (seleccionado o no), se les debe configurar con un par de cadenas de texto, una para cada estado: `android:textOff` como propiedad para el texto a mostrar cuando el botón no está seleccionado y `android:textOn` para cuando está seleccionado. Si no se informan estas propiedades se mostrará el texto *On* cuando este seleccionado y *Off* cuando no, mientras que si se informa la propiedad `android:text`, ésta será ignorada.

Usaremos el *LinearLayout* llamado `parent` como contenedor para introducir en él de modo dinámico los botones que luego haremos aparecer y desaparecer. Ya como código de la clase `Animations.java`, cree las siguientes variables de tipo miembro en la clase:

```
ViewGroup layout = null;
// animación al aparecer un elemento
ObjectAnimator animIn = null;
// animación al desaparecer un elemento
ObjectAnimator animOut = null;
//animación del resto al aparecer un elemento
ObjectAnimator animRestOut = null;
//animación del resto al desaparecer un elemento
ObjectAnimator animRestIn = null;
```

Estas variables nos servirán para tener la referencia de tanto el *ViewGroup* que aglutinará los botones que haremos desaparecer y aparecer, como de las animaciones que les acompañen.

Dentro del método `onCreate()`, se asigna el código a cada uno de los elementos de la pantalla. Comenzaremos obteniendo ciertas referencias y creando un *LinearLayout* horizontal para más adelante añadirle los botones:

```
@Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout. activity_animations);
        // boton de ocultar eliminar
        final ToggleButton invGone = (ToggleButton) findViewById(R.
        id.invGone);
        // tiempo de animación
        final SeekBar animLength = (SeekBar) findViewById(R.id.animLength);
        // etiqueta de tiempo
        final TextView duration = (TextView) findViewById(R.id.duration);

        // layout para los botones
        layout = new LinearLayout(this);
```

Crearemos después un *LayoutTransition* para gestionar las animaciones.

```
// efectos
final LayoutTransition layoutTrans = new LayoutTransition();
//   layout.setLayoutTransition(layoutTrans);
```

Como puede ver, existe una línea de código comentada; se trata de la asignación del objeto *LayoutTransition* a *ViewGroup* al que debe controlar las animaciones. Déjela así para que más adelante pueda apreciar la diferencia entre asignar este *LayoutTransition* y no asignarlo.

Para el tiempo de duración de las animaciones, se ha añadido un *SeekBar* y una etiqueta que muestra el valor actual del *SeekBar*. Para que la etiqueta muestre realmente este valor, hay que llamar al método `OnSeekBarChangeListener()`. Se debe añadir también una línea de código para que actualice la etiqueta en el momento de creación de la vista, ya que de lo contrario al iniciar la aplicación aparecería la etiqueta en blanco hasta que se actualizara el valor variando la barra.

```
// si cambia la duración de la animación, actualizar etiqueta
animLength.setOnSeekBarChangeListener(
new SeekBar.OnSeekBarChangeListener() {
// si cambia la barra de duración de la animación, //actualiza la etiqueta
public void onProgressChanged(SeekBar seekBar, int progress, boolean
fromUser) {
   duration.setText(String.format(getResources().getText(
            R.string.duration).toString(),animLength.getProgress()));

}

public void onStartTrackingTouch(SeekBar seekBar) {
   // TODO Auto-generated method stub
}


   public void onStopTrackingTouch(SeekBar seekBar) {
   // TODO Auto-generated method stub
   }
});
```

```
// poner la duracción actual durante la creación
duration.setText(String.format(getResources().getText(
   R.string.duration).toString(),animLength.getProgress()));
```

El botón de añadir elementos no los añadirá realmente creándolos de nuevo, sino que recorrerá los elementos existentes en el contenedor, volviendo a poner su propiedad de visibilidad a visible mediante `view.setVisibility(View.VISIBLE)`, y es que no olvidemos que aunque digamos que vamos a eliminar los botones del *layout*, lo único que se va a hacer es modificar su visibilidad a `GONE` o `INVISIBLE`, por lo que no hace falta volver a crearlos.

```
//botón de añadir elementos
Button addButton = (Button) findViewById(R.id.add);
addButton.setOnClickListener(new View.OnClickListener() {
public void onClick(View v) {               layoutTrans.
setDuration(animLength.getProgress());
for (int i = 0; i < layout.getChildCount(); ++i) {
         View view = (View) layout.getChildAt(i);
            view.setVisibility(View.VISIBLE);
      }
   }
});
```

Pero los botones que ocultaremos y haremos aparecer de nuevo no los hemos creado en el fichero de *layout*, por lo que hay que hacerlo mediante programación. A cada botón se le pondrá como texto "Púlsame (n)" siendo n el número de elemento introducido, para poder diferenciarlos más adelante en la ejecución del programa y entender mejor como se recolocan los elementos. También hay que añadir lógica para que cuando se pulse uno de ellos, cambie su visibilidad teniendo en cuenta el estado del *ToggleButton* que dicta si ha de ocultarse o ha de sacarse del *layout*.

```
// añadir botones
Button btn = null;
for (int i = 0; i < 6; ++i) {
   btn = new Button(this);          btn.setText(String.format(getResources().
getText(R.string.clickme).toString(),i));
   btn.setOnClickListener(new View.OnClickListener() {
      public void onClick(View v) {
      layoutTrans.setDuration(animLength.getProgress());
          v.setVisibility(invGone.isChecked() ? View.GONE : View.INVISIBLE);
      }
         });
   layout.addView(btn);
}
```

Por último y para acabar con el método `onCreate()` se ha de añadir el *LinearLayout* con los botones a la vista principal, más concretamente lo añadimos al *ViewGroup* con identificador `R.id.parent` .

```
//añadir el layout a la vista
ViewGroup parent = (ViewGroup) findViewById(R.id.parent);
parent.addView(layout);
```

Antes de poder probar la aplicación, se debe haber completado el archivo `strings.xml` con los valores que hemos ido utilizando:

```
<string name="buttonGeneration">Generar botones</string>
    <string name="noAnimation">Sin animación</string>
    <string name="animation">Con animación</string>
    <string name="hide">Ocultar</string>
    <string name="gone">Eliminar</string>
    <string name="duration">Duración %d ms</string>
    <string name="clickme">Púlsame (%d)</string>
```

Si prueba la aplicación, verá que al pulsar sobre el *ToggleButton* de Quitar/Ocultar, el comportamiento de la aplicación varía, y al pulsar sobre los botones, estos desaparecerán ordenándose en caso de que esté Quitar activo y simplemente desapareciendo si lo está Ocultar y podemos hacerlos visibles de nuevo mediante el botón Genera botones. Pero nada de animaciones. Quite el comentario a la línea.

```
layout.setLayoutTransition(layoutTrans);
```

Si prueba de nuevo la aplicación, verá que ya hay unas pequeñas animaciones a la hora de ocultar los elementos, y es que en este caso ya hemos asignado el objeto *LayoutTransition* al *ViewGroup;* también puede probar a modificar el tiempo a invertir en la animación para ver que ésta varía su velocidad de ejecución. No obstante aún no hay diferencias entre tener seleccionado el *ToggleButton* de animaciones o no tenerlo. Modifiquemos el código para añadir unas animaciones más interesantes.

Al final del método `onCreate()`, llamaremos al método `createAnimations()`, donde se configurarán cada una de las animaciones a utilizar.

```
// dependiendo de si el botón esta pulsado o no, usamos unos efectos u otros
// configuración de las animaciones
createAnimations();
```

Para asignar las animaciones, nos valdremos del método `setOnCheckedChangeListener()` del *ToggleButton*, de manera que cuando se detecte que se ha activado, se asignen todas las animaciones a cada uno de los eventos; por ejemplo para que cuando se produzca una aparición de un elemento, los que ya existían tomarán la animación `animRestIn`, se debería realizar el código:

```
layoutTrans.setAnimator(LayoutTransition.CHANGE_APPEARING, animRestIn);
```

Los distintos tipos de animaciones que se pueden asignar son:

- APPEARING: cuando un elemento nuevo se añade o se hace visible dentro del contenedor; es la animación que se debe aplicar al nuevo elemento mientras se añade.

- DISAPPEARING: cuando un elemento se elimina o se hace invisible dentro del contenedor; es la animación que se debe aplicar al elemento que desaparece mientras se quita.
- CHANGE_APPEARING: cuando un elemento nuevo se añade o se hace visible dentro del contenedor; es la animación que se debe aplicar a los elementos que ya existían en el contenedor mientras el nuevo se añade.
- CHANGE_DISAPPEARING: cuando un elemento se elimina o se hace invisible dentro del contenedor; es la animación que se debe aplicar a los elementos que siguen estando visibles en el contenedor, mientras el elemento se quita.

Para establecer el tiempo de espera entre los comienzos de las animaciones, nos podemos valer del método `setStagger()` del objeto *LayoutTransition*, al que se le informa como parámetros, el tipo de animación y el tiempo que debe esperar para reproducirla, por ejemplo:

```
layoutTrans.setStagger(LayoutTransition.CHANGE_DISAPPEARING, 50);
```

Para asignar todas las animaciones (y eliminar la asignación cuando no se quieran animaciones) mediante el *ToggleButton* añada al final del método `onCreate()`:

```
//activar desactivar animaciones
ToggleButton animations = (ToggleButton) findViewById(R.id.animation);
animations.setOnCheckedChangeListener(new CompoundButton.
OnCheckedChangeListener() {
public void onCheckedChanged(CompoundButton buttonView, boolean isChecked) {
if (isChecked) {
   layoutTrans.setAnimator(LayoutTransition.APPEARING,
   animIn); layoutTrans.setAnimator(LayoutTransition.DISAPPEARING,
   animOut);   layoutTrans.setAnimator(LayoutTransition.CHANGE_APPEARING,
   animRestIn);   layoutTrans.setAnimator(LayoutTransition.CHANGE_
   DISAPPEARING, animRestOut);
   layoutTrans.setStagger(LayoutTransition.CHANGE_APPEARING, 50);
   layoutTrans.setStagger(LayoutTransition.CHANGE_DISAPPEARING, 50);
} else {
   //no animaciones
   layoutTrans.setAnimator(LayoutTransition.APPEARING, null);
   layoutTrans.setAnimator(LayoutTransition.DISAPPEARING, null);
   layoutTrans.setAnimator(LayoutTransition.CHANGE_APPEARING, null);
   layoutTrans.setAnimator(LayoutTransition.CHANGE_DISAPPEARING, null);
   layoutTrans.setStagger(LayoutTransition.CHANGE_APPEARING, 0);
   layoutTrans.setStagger(LayoutTransition.CHANGE_DISAPPEARING, 0);
      }
}
});
```

Para finalizar, se debe implementar el método `createAnimations()`, que se encargará de configurar cada una de las cuatro animaciones.

A la hora de asignar animaciones propias, lo más sencillo es animar una sola propiedad del objeto, por ejemplo la rotación en el eje X. Mediante el objeto *ObjectAnimator* se puede indicar quien es el receptor de la animación, la propiedad a animar y entre que valores se debe hacer la animación:

```
ObjectAnimator anim = ObjectAnimator.ofFloat(Animations.this, "rotationX",
0f, 180f);
```

Entre otras muchas opciones del objeto *ObjectAnimator*, podemos establecer mediante el método `setDuration()` la duración de cada animación, al igual que cambiar el elemento sobre el que aplicar la animación mediante `setTarget()`.
Lo normal es añadir un *listener* para que cuando la animación termine se ajusten bien los parámetros que realmente debe tener el elemento al acabar la animación y así nos aseguramos que la interfaz quede en todo momento como se espera.

```
anim.addListener(new AnimatorListenerAdapter() {
    public void onAnimationEnd(Animator anim) {
    View view = (View) ((ObjectAnimator) anim).getTarget();
      view.setRotationX(0f);
      }
    });
```

En ocasiones puede ser más que suficiente ver cómo gira sobre sí mismo un elemento, pero las animaciones más vistosas son aquellas que modifican varios parámetros a la vez, como puede ser una rotación con una traslación. Para estos casos se debe utilizar la clase *PropertyValuesHolder*. Esta clase está pensada para crear animaciones exclusivamente con objetos *ValueAnimator* o *ObjectAnimator*. Un objeto *PropertyValuesHolder* almacena información sobre una propiedad y los valores que debe tomar durante la animación. Posee distintos métodos donde informar valores que sean enteros, flotantes, objetos genéricos u objetos *Keyframe*. Por ejemplo si se quiere que la propiedad `"scaleX"` tome valores entre el cien por ciento, el cero y nuevamente el cien por ciento durante la animación su *PropertyValuesHolder* sería:

```
PropertyValuesHolder scaleX =  PropertyValuesHolder.ofFloat("scaleX", 1f,
0f, 1f);
```

Una vez conocido el funcionamiento a groso modo de las animaciones, vamos a ponerlas en práctica para el ejemplo que estamos realizando.
Para la animación de entrada de un elemento, haremos que se vaya acercando mientras va girando, es decir realizaremos un escalado del elemento a la vez que una rotación. Añada al método `createAnimations()`:

```
// a realizar cuando se añade
PropertyValuesHolder inScaleX =
PropertyValuesHolder.ofFloat("scaleX", 0f, 1f);
PropertyValuesHolder inScaleY =
      PropertyValuesHolder.ofFloat("scaleY", 0f, 1f);
Keyframe kfi0 = Keyframe.ofFloat(0f, 0f);
Keyframe kfi1 = Keyframe.ofFloat(.9999f, 360f);
Keyframe kfi2 = Keyframe.ofFloat(1f, 0f);
PropertyValuesHolder inRotation =
      PropertyValuesHolder.ofKeyframe("rotation", kfi0, kfi1, kfi2);
animIn = ObjectAnimator.ofPropertyValuesHolder(
         Animations.this, inScaleX, inScaleY, inRotation);
//listener
animIn.addListener(new AnimatorListenerAdapter() {
   public void onAnimationEnd(Animator anim) {
         View view = (View) ((ObjectAnimator) anim).getTarget();
                view.setScaleX(1f);
                view.setScaleY(1f);
                view.setRotation(0f);
   }
});
```

Para la eliminación de un elemento, lo que haremos es un escalado tanto de la coordenada X como de la Y, primero aumentando el elemento, haciendo un zoom hacia él, y luego alejándolo hasta que no se vea, es decir a escala 0.

```
// a realizar cuando se elimina
PropertyValuesHolder outScaleX =
PropertyValuesHolder.ofFloat("scaleX", 1f, 2f, 1f, 0f);
PropertyValuesHolder outScaleY =
PropertyValuesHolder.ofFloat("scaleY", 1f, 2f, 1f, 0f);
animOut = ObjectAnimator.ofPropertyValuesHolder(
         Animations.this,  outScaleX, outScaleY);
//listener
animOut.addListener(new AnimatorListenerAdapter() {
   public void onAnimationEnd(Animator anim) {
         View view = (View) ((ObjectAnimator) anim).getTarget();
            view.setScaleX(0f);
            view.setScaleY(0f);
      }
});
```

Como ya se explicó, no sólo existen animaciones para los elementos que aparecen y desaparecen de pantalla, sino también para los que ya se encuentran en ella. Para el caso que nos ocupa, si hemos eliminado el botón 2, y lo volvemos a hacer aparecer, se debe abrir un hueco para que pueda aparecer éste. Los botones que se desplazan para dejar hueco al nuevo elemento pueden ser también animados.

Para los elementos que quedan en pantalla cuando un botón se elimina, lo que haremos será trasladarlos a su nueva ubicación, a la vez que se escalan, dando la sensación de que se alejan para volver de nuevo a acercarse.

```
// a realizar el resto cuando uno se elimina
PropertyValuesHolder left =PropertyValuesHolder.ofInt("left", 0, 1);
PropertyValuesHolder top =PropertyValuesHolder.ofInt("top", 0, 1);
PropertyValuesHolder right =PropertyValuesHolder.ofInt("right", 0, 1);
PropertyValuesHolder bottom =PropertyValuesHolder.ofInt("bottom", 0, 1);
PropertyValuesHolder scaleX =PropertyValuesHolder.ofFloat("scaleX", 1f, 0f, 1f);
PropertyValuesHolder scaleY =PropertyValuesHolder.ofFloat("scaleY", 1f, 0f, 1f);
animRestOut = ObjectAnimator.ofPropertyValuesHolder(
   Animations.this, left, top, right, bottom, scaleX, scaleY);
//listener
animRestOut.addListener(new AnimatorListenerAdapter() {
   public void onAnimationEnd(Animator anim) {
         View view = (View) ((ObjectAnimator) anim).getTarget();
         view.setScaleX(1f);
            view.setScaleY(1f);
   }
});
```

Para los elementos que ya están en pantalla, cuando un elemento aparece, lo que haremos es desplazarlos hacia la derecha a su nueva posición, a la vez que rotan sobre ellos mismos.

```
// a realizar el resto cuando uno se añade
        Keyframe kfri0 = Keyframe.ofFloat(0f, 0f);
        Keyframe kfri1 = Keyframe.ofFloat(.5f, 360f);
        Keyframe kfri2 = Keyframe.ofFloat(1f, 0f);
        PropertyValuesHolder rotation =
                PropertyValuesHolder.ofKeyframe("rotationX", kfri0, kfri1,
                kfri2);
        animRestIn = ObjectAnimator.ofPropertyValuesHolder(
                        Animations.this, left, top, right, bottom, rotation);
        //listener
        animRestIn.addListener(new AnimatorListenerAdapter() {
            public void onAnimationEnd(Animator anim) {
                View view = (View) ((ObjectAnimator) anim).getTarget();
                view.setRotation(0f);
            }
        });
```

Ahora ya puede probar la aplicación, donde todos los botones son funcionales y configurar las velocidades de animación así como su comportamiento. Le invito además a que juegue variando el método `createAnimations()` y verá que de modo sencillo puede hacer animaciones vistosas.

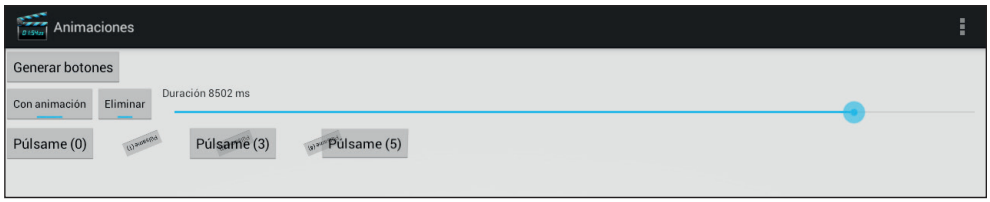

Figura 21.2. Aspecto de las animaciones en ejecución.

Pero como hemos dicho, es posible animar cualquier tipo de elemento en pantalla, vamos a añadir un poco más de código al ejemplo para mostrarle como hacer una vista que rote sobre si misma a la vez que cambia su contenido.

Lo que se quiere conseguir es que parezca que la vista gire y que por cada una de las caras tenga un contenido distinto. Tendremos dos vistas, una girada 90 grados y otra 0. Conseguiremos el efecto haciendo rotar la vista visible de 0 a 90 grados, mientras que la segunda lo haremos de -90 grados a 0. Para que en este ejemplo queden bien las animaciones, hay que tener en cuenta que una de las imágenes no se verá, mientras que la otra si, y al ocultar la primera se debe mostrar la segunda, pero no a la vez, sino que se debe comenzar a mostrar cuando la primera haya acabado, así dará la sensación que se está girando la vista y que por el otro lado tiene la nueva imagen.

Modifique el *layout* del ejemplo de modo que quede:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:orientation="vertical" android:layout_width="fill_parent"
   android:layout_height="fill_parent">
<LinearLayout android:orientation="vertical"
android:layout_width="fill_parent" android:layout_height="wrap_content"
      android:id="@+id/parent">
      <Button android:layout_width="wrap_content"
android:layout_height="wrap_content" android:text="@string/buttonGeneration"
         android:id="@+id/add" />

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
android:orientation="horizontal" android:layout_width="fill_parent"
      android:layout_height="wrap_content">
      <ToggleButton android:layout_width="wrap_content"
android:layout_height="wrap_content" android:id="@+id/animation"
android:textOff="@string/noAnimation" android:textOn="@string/animation" />

      <ToggleButton android:layout_width="wrap_content"
android:layout_height="wrap_content" android:id="@+id/invGone"
android:textOff="@string/hide" android:textOn="@string/gone" />

      <LinearLayout xmlns:android="http://schemas.android.com/apk/res/
      android"
android:orientation="vertical" android:layout_width="fill_parent"
android:layout_height="wrap_content" android:paddingLeft="10dp">
      <TextView android:layout_width="wrap_content"
android:layout_height="wrap_content" android:id="@+id/duration" />
      <SeekBar android:layout_width="fill_parent"
android:layout_height="wrap_content" android:id="@+id/animLength"
      android:max="10000" android:progress="500" />
      </LinearLayout>
      </LinearLayout>
</LinearLayout>

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

```
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:id="@+id/images">
    <ImageView
        android:id="@+id/img1"
        android:layout_width="match_parent"
        android:layout_weight="1.0"
        android:layout_height="0dip"
        android:src="@drawable/image1"/>
    <ImageView
        android:id="@+id/img2"
        android:layout_width="match_parent"
        android:layout_weight="1.0"
        android:layout_height="0dip"
        android:src="@drawable/image2"
        android:visibility="gone"/>
</LinearLayout>
</LinearLayout>
```

Como lo que queremos hacer es que las imágenes roten al tocar sobre ellas para dar la sensación de que se les da la vuelta, en el código se tiene que asignar al *LinearLayout* llamado "`images`" la lógica para ello, esto lo realizaremos mediante el método `setOnClickListener()`.
Al iniciar la aplicación debemos mantener una de las imágenes giradas y asignar el código para lanzar los giros al *LinearLayout* correspondiente. Modifique para ello el método `onCreate()`, añadiéndole al final las líneas:

```
final ImageView image1 = (ImageView) findViewById(R.id.img1);
final ImageView image2 = (ImageView) findViewById(R.id.img2);
//inicializar girada
image2.setRotationY(-90f);
//añadir la lgógica para los giros
LinearLayout images = (LinearLayout) findViewById(R.id.images);
images.setOnClickListener(new View.OnClickListener() {
public void onClick(View v) {
         flip(image1,image2, animLength.getProgress());
   }
 });
```

El método `flip()` dentro de la clase *Animations* será el encargado de controlar las animaciones. Lo primero que se debe hacer es conocer cuál de las dos imágenes es la que se está mostrando para saber qué animación se debe asignar a cada una de ellas. Una vez conocido el estado de visibilidad de cada una de las imágenes, se asigna a las variables correspondientes (`visibleImage` e `invisibleImage`). Se deben crear también las animaciones para cada una de las vistas; una de las animaciones será para pasar de invisible a visible, y será una rotación de -90 grados a 0 en el eje Y, mientras que la animación para pasar de visible a invisible será de 0 grados a 90 también en el eje Y.

En este caso las el inicio animaciones las controlaremos nosotros mismos mediante código, a diferencia del caso anterior que se disparaban mediante la modificación de los elementos del *layout*. Para iniciar una animación mediante programación se debe llamar al método `start()` del objeto *ObjectAnimator* que interese en cada momento. Como se explicó unas líneas antes, en este caso hay que sincronizar las animaciones, de modo que comience la animación de mostrar la vista cuando acabe la de ocultar la vista. Para ello configuraremos la clase *listener* de tipo *AnimatorListenerAdapter* sobre la animación que oculta la imagen, llamando a su método `onAnimationEnd()` de modo que podamos saber cuando ésta acaba y comenzar la animación de mostrar. Todo esto se ha de realizar antes de que comience la animación de ocultar, así pues, esta animación será la última sentencia a ejecutar dentro de este método.
El código del método `flip()` quedaría:

```
<Listado>private void flip(ImageView image1,ImageView image2, int duration) {
        final ImageView visibleImage;
        final ImageView invisibleImage;
        //mirar la visibilidad que tiene la imagen1, para saber cuál de
        ellas tiene que tener qué animación
        if (image1.getVisibility() == View.GONE) {
         visibleImage = image2;
         invisibleImage = image1;
        } else {
         invisibleImage = image2;
            visibleImage = image1;
        }
        // rotación 90 grados para pasar  a visible
        final ObjectAnimator goneToVisible = ObjectAnimator.
        ofFloat(invisibleImage, "rotationY",
                -90f, 0f);
   //ajustar duración
        goneToVisible.setDuration(duration);

        //rotacion 90 grados para pasar  a invisible
        ObjectAnimator visibleToGone = ObjectAnimator.ofFloat(visibleImage,
        "rotationY", 0f, 90f);
//ajustar duración
        visibleToGone.setDuration(duration);

        //listener
        visibleToGone.addListener(new AnimatorListenerAdapter() {
            @Override
            public void onAnimationEnd(Animator anim) {
               visibleImage.setVisibility(View.GONE);
                //cuando se acabe de ocultar, comenzar a mostrar la nueva
                goneToVisible.start();
                invisibleImage.setVisibility(View.VISIBLE);
            }
        });
        // comenzar a ocultar
        visibleToGone.start();
    }
```

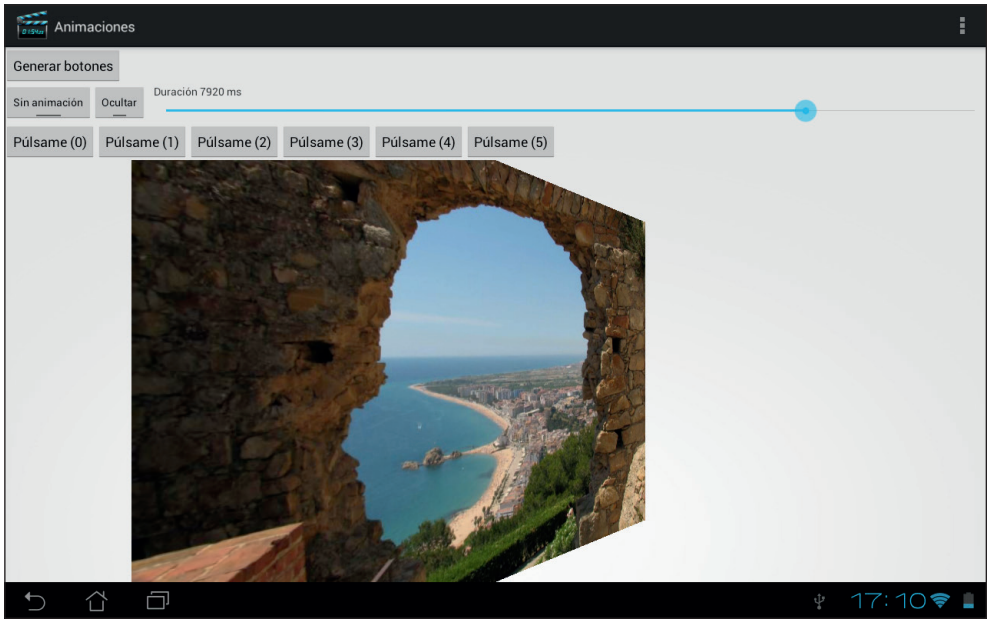

Figura 21.3. Imágenes rotando.

Animaciones tipo frame

Las animaciones denominadas *frame* son aquellas que van mostrando diferentes imágenes con pequeñas modificaciones de modo que parezca que existe movimiento por parte de estas, es como hacer dibujos animados. Para hacer este tipo de animaciones, lo primero que se necesita son las imágenes que se reproducirán una detrás de otra, para este ejemplo se utilizarán cuatro imágenes que generarán la animación final. La definición de la animación se hace a través de un fichero XML, que se guarda en el directorio `/src/main/res/drawable/` (si quiere puede guardarlo en `/src/main/res/anim/` aunque este directorio suele usarse para las animaciones de *layout*), y su estructura es:

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

    android:oneshot=["true" | "false"] >
    <item
        android:drawable="@drawable/resource_name"
        android:duration="integer" />
</animation-list>
```

Donde `<animation-list>` es siempre el elemento raíz que puede contener uno o más elementos `<item>` y el atributo `android:oneshot` que si es verdadero indica si se tiene que reproducir una sola vez y en caso de ser falso, se reproduce indefinidamente. El elemento `<item>` se utiliza para definir el dibujo a mostrar y su comportamiento; el atributo `android:drawable` lleva informado el gráfico en sí y en `android:duration` se informa la duración que tiene que estar en pantalla medida en milisegundos.
Para el ejemplo cree un fichero llamado `animation.xml` con el siguiente contenido:

```
<?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/frame0" android:duration="50"/>
    <item android:drawable="@drawable/frame1" android:duration="50"/>
    <item android:drawable="@drawable/frame2" android:duration="50"/>
    <item android:drawable="@drawable/frame3" android:duration="50"/>
</animation-list>
```

Modificaremos el *layout* principal para añadir la animación recién creada. Inserte la nueva vista entre el *ToggleButton* llamado `android:id="@+id/invGone"` y el *LinearLayout* que va a continuación.

```
<ToggleButton
                android:id="@+id/invGone"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:textOff="@string/hide"
                android:textOn="@string/gone" />
            <View
                android:id="@+id/imgAnim"
                android:layout_width="45dip"
                android:layout_height="45dip"/>
            <LinearLayout
                xmlns:android="http://schemas.android.com/apk/res/android"
                android:layout_width="fill_parent"
                android:layout_height="wrap_content"
                android:orientation="vertical"
                android:paddingLeft="10dp" >
```

Para animarlo y detenerlo usaremos las imágenes que giran, de modo que se inicie la animación cuando éstas se animan, y se detenga con ellas.
Antes del método `onCreate()` y debajo de todas las propiedades de clase de tipo *ObjectAnimator*, añada una para mantener la referencia a la animación, de modo que se pueda acceder a ella desde distintos métodos

```
// animación del resto al desaparecer un elemento
   ObjectAnimator animRestIn = null;
   //animacíón frame
   AnimationDrawable frameAnimation;
```

Dentro del método `onCreate()` añada al final las siguientes líneas, que sirven para obtener la referencia a la vista que albergará la animación, asignarle como fondo la animación y obtener la referencia de ésta.

```
//animación frame
View frameAnimationView = findViewById(R.id.imgAnim);
frameAnimationView.setBackgroundResource(R.drawable.animation);
frameAnimation = (AnimationDrawable) frameAnimationView.getBackground();
```

Ya sólo queda generar el código que inicie y detenga la animación. Al final del método flip(); modifique las líneas para que queden:

```
// listener
visibleToGone.addListener(new AnimatorListenerAdapter() {
   @Override
   public void onAnimationEnd(Animator anim) {
      visibleImage.setVisibility(View.GONE);
      // cuando se acabe de ocultar, comenzar a mostrar la nueva
      goneToVisible.start();
      invisibleImage.setVisibility(View.VISIBLE);
      frameAnimation.stop();
   }
});
// comenzar a ocultar
visibleToGone.start();
frameAnimation.start();
```

Figura 21.4. Aplicación con el elemento de animación frame en un móvil.

Como habrá deducido, mediante el método `start()` se inicia la animación y se detiene con el método `stop()`. Ya puede probar la aplicación y disfrutar de la animación mientras las imágenes giran. Una cosa a tener en cuenta es que la velocidad de esta animación no está dada por el selector de velocidades de la aplicación, sino directamente desde la definición de la animación en el XML por lo que no variará la velocidad como se ha hecho con los otros componentes; queda por parte del lector, hacer que la velocidad de la animación varíe también con el *Slider* tal y como lo hacen el resto de objetos de la aplicación.

Temas y estilos

Cada una de las subclases de tipo *View* que se utilizan para definir las vistas en Android, tienen ciertos atributos para cambiar su aspecto, tales como fondo de la imagen, tipo de letra, etc, y dependiendo del tipo de *View* que sea se pueden definir ciertos aspectos dependientes de estado, como por ejemplo en los botones el aspecto que deben tener cuando se pulsan. Pero imagine que está haciendo una aplicación con todos los botones azules y llega un momento que el diseñador le dice que no, que no pueden ser azules y deben ser rojos. Si se han definido los colores de fondo de los botones uno a uno sería una tarea tediosa de realizar. Para esto existen los estilos (*style*), que son una serie de propiedades definidas mediante un archivo XML que varían el aspecto de una vista o ventana. De algún modo se asemejan bastante a los CSS en diseños web. Así, a la hora de definir el XML del *layout* se pueden extraer todos los atributos de un componente que sean correspondientes con el aspecto y encapsularlos en un estilo, y para darle el aspecto simplemente hay que asignar el estilo al componente del *layout*.

Por otro lado se tienen los temas (*theme*) que realmente son estilos que se aplican a toda la actividad en lugar de a una sola vista como es el caso de los estilos. Para definir los estilos se utiliza un fichero XML que debe guardarse en el directorio `/src/main/res/values` y que puede tener cualquier tipo de nombre (incluso se pueden usar varios ficheros) pero lo que es mandatorio es que el elemento raíz del XML de este fichero sea de tipo `<resources>`. Por ejemplo, actualmente al crear un proyecto ya genera por defecto un archivo de estilos llamado `styles.xml`, Dentro de este elemento raíz, se irán añadiendo los distintos estilos mediante la etiqueta `<style>` y dentro de cada una de ellas, se añadirán los atributos a modificar mediante la etiqueta `<item>`. Recuerde que un tema no es más que un estilo que se aplica a toda una actividad, y por lo tanto se debe definir como un estilo.

Al ser un recurso, durante la compilación del proyecto se generará la conveniente entrada en la clase *R* de manera automática, de modo que podamos utilizarlos cómodamente en el diseño. Para acceder a los estilos desde el XML simplemente hay que utilizar `@style/` delante del nombre del estilo a utilizar. Por ejemplo, podríamos definir un estilo para una caja de texto del siguiente modo:

```
<resources>
<style name=" EditTextNormal" >
        <item name="android:layout_width">fill_parent</item>
        <item name="android:layout_height">wrap_content</item>
        <item name="android:textColor">#0000FF</item>
        <item name="android:typeface">sans</item>
        <item name="android:maxLines">1</item>
        <item name="android:background">@android:drawable/editbox_background</
item>
    </style>
</resources>
```

Dentro de la etiqueta de estilo se define el atributo `name="EditText"` que servirá para identificarlo y poderlo asignar como recurso. Como puede ver, para cara uno de los atributos que se quieren cambiar del View hay que crear una entrada `<item>`, donde el atributo `name` indica el atributo del *View* a cambiar a definir y como valor de esta etiqueta se da el valor que debe tomar el atributo en el *View*. Para asignar este estilo en el *layout* valdría con:

```
<EditText style="@style/EditTextNormal"/>
```

Con esto se está haciendo que el elemento *EditText* tenga de tamaño todo el ancho posible y ajustado en alto, con una letra *sans* de color azul, que ocupe una línea y que presente el fondo `@drawable/editbox_background`, pero como ve, en la definición del *layout* propiamente dicho, no hay nada de esto, lo que hace que queden mucho más limpios y fáciles de mantener. El hecho que hayamos llamado al estilo `"EditTextNormal"` no quiere decir que solamente se pueda utilizar en elementos de ese tipo; los nombres son en cierto modo arbitrarios (ya veremos que si queremos hacer herencia no lo son), y se puede asignar el nombre que se quiera como estilo para utilizarlo en cualquier tipo de vista, incluso compartir estilos, ya que los atributos se aplicarán igualmente, es decir, podríamos asignar:

```
<TextView style="@style/EditTextNormal"/>
```

Anteriormente se ha hablado de la herencia, y es que es posible que nos encontremos que por ejemplo tenemos unas etiquetas de texto con unas propiedades pero que justamente cuando se usan de cabecera son todas las propiedades iguales excepto el tamaño de fuente. Esto se soluciona utilizando herencia en los estilos. Para usar herencia dentro del mismo paquete, vale

con poner como nombre del estilo, el nombre del estilo a heredar, un punto y el nombre que le queramos dar al estilo hijo, por ejemplo:

```
<resources>
<style name=" EditTextNormal" >
        <item name="android:layout_width">fill_parent</item>
        <item name="android:layout_height">wrap_content</item>
        <item name="android:textColor">#0000FF</item>
        <item name="android:typeface">sans</item>
        <item name="android:maxLines">1</item>
        <item name="android:background">@android:drawable/editbox_background</
item>
    </style>
    <style name=" EditTextNormal.Red" >
        <item name="android:textColor">#FF0000</item>
    </style>
</resources>
```

Con lo que habríamos creado un estilo exactamente igual a `"EditTextNormal"` sólo que con la fuente en color rojo. A la hora de heredar, no existen restricciones en el número de herencias, es decir, el estilo1 puede heredar del estilo2 que a su vez hereda del estilo3 y así las veces que haga falta. Las herencias pueden hacerse también sobre los estilos propios de Android, para ello hay que indicar dentro del elemento `<item>` el atributo `parent` indicando el estilo del que heredar. Esta forma de trabajar (utilizando el atributo `parent`) es también posible con estilos propios, si bien se aconseja no utilizarla por claridad a la hora de distinguir estilos heredados de propios de los heredados del sistema. Un ejemplo de estilo heredado del sistema sería:

```
<style name="EditText" parent="@android:style/TextAppearance.Medium">
        <item name="android:layout_width">fill_parent</item>
        <item name="android:layout_height">wrap_content</item>
        <item name="android:textColor">#000000</item>
        <item name="android:typeface">sans</item>
        <item name="android:maxLines">1</item>
        <item name="android:background">@drawable/editbox_background</item>
    </style>
```

Para conocer todos los atributos modificables podemos dirigirnos a `http://developer.android.com/reference/android/R.attr.html` pero siempre teniendo en cuenta que no todos los atributos están disponibles en todos las vistas, por lo que es buena práctica dirigirse a la documentación de cada *View* concreta para conocer exactamente los atributos disponibles.

Advertencia:

Si al definir un View *en el* Layout *se informa un estilo que por ejemplo tenga un tamaño de letra, pero en la propia definición del* View *se especifica otra distinta, la de la definición prevalece sobre la del estilo.*

Vamos a hacer un pequeño ejercicio para poner todo esto en práctica. Cree un proyecto con los siguientes parámetros:

- Application name: Styles
- Module Name: Styles
- Package name: com.acme.styles
- Minimum Required SDK: API 11: Android 3.0 (Honeycomb) o superior
- Activity name: Styles

Abra el archivo de estilos que acompaña al proyecto, se encuentra dentro de la carpeta `/src/main/res/values/` y se llama `styles.xml`. Verá que ya existe contenido, se trata de una entrada para la generación del tema base de la aplicación. Vamos a comenzar cambiando el aspecto de un botón, creando varios estilos, algunos heredados y otros no, y además cambiaremos su estilo dependiendo de su estado (si lo presionan o no).
Los estilos los iremos añadiendo dentro del fichero `styles.xml` y como hijos directos del elemento `<resources>`, es decir entre sus etiquetas XML.
Comenzaremos creando un estilo simple, heredando de uno ya creado por Android.

```
<!-- Estilo simple -->
    <style name="Button" parent="@android:style/Widget.Button">
        <item name="android:gravity">center_vertical|center_horizontal</item>
        <item name="android:layout_width">wrap_content</item>
        <item name="android:layout_height">wrap_content</item>
        <item name="android:paddingLeft">20dip</item>
        <item name="android:paddingRight">20dip</item>
        <item name="android:textColor">#00ff00</item>
    </style>
```

Se puede ver que el nuevo estilo creado, lo hemos llamado `"Button"` y hereda el `@android:style/Widget.Button` de Android; sobre la herencia, se han definido unas variaciones que corresponden a la gravedad, los tamaños, los *padding* y el color del texto que en este caso hemos usado el verde. El color del texto lo hemos definido como un RGB mediante su valor hexadecimal, pero también podríamos haber creado una constante color o utilizar algunas de las que vienen definidas, por ejemplo para poner un rojo dependiente del tema de Android, sería

```
<item name="android:textColor">@android:color/holo_red_dark</item>
```

Añadamos otro estilo:

```
<!-- fondo de botón, hereda estilo simple -->
    <style name="Button.Background">
        <item name="android:background">@drawable/bg_green</item>
        <item name="android:textStyle">bold</item>
        <item name="android:textSize">20sp</item>
    </style>
```

En este caso el estilo hereda directamente del estilo `"Button"` y le añade un fondo gráfico al botón y modificaciones en el texto, haciéndolo negrita y de tamaño 20. Creemos ahora un estilo sin herencia:

```
<!-- fondo de botón NO hereda estilo simple -->
    <style name="Background">
        <item name="android:background">@drawable/bg_green</item>
        <item name="android:textStyle">bold</item>
        <item name="android:textSize">20dip</item>
        <item name="android:layout_width">wrap_content</item>
        <item name="android:layout_height">wrap_content</item>
    </style>
```

Este estilo al no heredar del "Button" necesita que se le defina el ancho y alto de la vista (o se podría haber puesto en la propia vista, pero por comodidad cuando creemos el *layout* lo definimos aquí) de lo contrario la aplicación se detendrá al ejecutarla. La diferencia más notable que habrá entre este estilo y el anterior cuando se pruebe la aplicación es el color del texto, que en este caso aparecerá en negro en lugar de verde. Añadimos estilos para el texto

```
<!-- Texto heredando estilo simple -->
    <style name="Button.text">
        <item name="android:textColor">#FF0000</item>
    </style>
```

Con este estilo se cambiará el color del texto a rojo del estilo heredado por `"Button"`. Y en caso de querer añadir una nueva herencia con el color seleccionado a todo lo heredado hasta ahora sería:

```
<!-- Texto heredando estilo fondo y estilo simple -->
    <style name="Button.Background.text">
        <item name="android:textColor">#FF0000</item>
    </style>
```

Por parte del *layout* se añadirá un *LinearLayout* con tantos botones como estilos para ver cómo se modifican, indicando en el texto, el estilo que utilizan para que pueda comprobar con más claridad los cambios de cada uno de ellos:

```
<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="match_parent"   android:layout_height="match_parent">
<LinearLayout     xmlns:tools="http://schemas.android.com/tools"
   android:layout_width="match_parent"   android:layout_height="wrap_content"
    android:orientation="vertical">
   <Button android:layout_width="wrap_content"
   android:layout_height="wrap_content"
        android:text="Sin estilo" />

   <Button  style="@style/Button" android:text="Button" />

   <Button style="@style/Button.Background" android:text="Button.Background" />
```

```
    <Button style="@style/Background" android:text="Background" />

    <Button style="@style/Button.text" android:text="Button.text" />

    <Button style="@style/Button.Background.text" android:text="Button.
    Background.text" />
</LinearLayout>
</ScrollView>
```

Para el recurso `bg_green` puede usar cualquier gráfico de color verde y colocarlo en los directorios `drawable` correspondientes (ya veremos más adelante como generarlos por coódigo XML y usar 9-patch, pero por el momento es válido).

Si prueba la aplicación verá cómo los botones han cambiado su aspecto con los distintos estilos. Pero vamos a dar un poco más de vistosidad a los botones y vamos a hacer que de alguna manera hagan partícipe al usuario de que han sido pulsados. En este ejemplo, lo que se hará es que cuando no esté pulsado aparezca el fondo verde y cuando se pulse pase a rojo; pero como tenemos el texto en color rojo, también cambiaremos su color para que se distinga, en este caso usaremos azul.

Comenzaremos por el fondo del botón. Para ello crearemos un archivo XML con el contenido que debe mostrar dependiendo del estado en el que se encuentre, aunque sea un archivo XML internamente se trata como un *drawable* más (como las animaciones) y podrá ser utilizado en cualquier lugar donde se pudiera usar un gráfico al uso. Estos archivos que definen estados tienen un elemento raíz llamado `<selector>` y dentro de él, aparecen tantos elementos `<item>` como estados se quieran definir. El elemento `<item>` es el que se encarga de definir los cambios, por ejemplo para poder cambiar el fondo, cree un nuevo archivo XML dentro del directorio `/src/main/res/drawable` y llámelo `button_bg.xml` su contenido será:

```
<?xml version="1.0" encoding="utf-8"?>
<selector xmlns:android="http://schemas.android.com/apk/res/android">
    <!-- presionado -->
    <item android:state_pressed="true" android:drawable="@drawable/bg_red" />
    <!-- defecto -->
    <item android:drawable="@drawable/bg_green" />
</selector>
```

En la primera entrada `<item>` se define el elemento gráfico `@drawable/bg_red` (use cualquier gráfico rojo) cuando el estado presionado sea positivo y en la segunda entrada se define el ele-mento gráfico `@drawable/bg_green` cuando la vista se encuentra sin ningún modificador activo, el estado por defecto. Existen otros modificadores como `state_checked` o `state_focused` con los que jugar y ajustar el compor-tamiento del elemento gráfico a lo esperado.

Para hacer uso de este nuevo *drawable,* diríjase al fichero de estilos `styles.xml` y modifique las dos entradas

```
<item name="android:background">@drawable/bg_green</item>
```

por:

```
<item name="android:background">@drawable/button_bg</item>
```

Si prueba ahora la aplicación, podrá ver cómo al pulsar sobre los botones que heredan el fondo de pantalla, éste cambia mientras se mantiene la pulsación. Vamos a modificar el texto para que varíe también con la pulsación su color, cree contraste y permita una mejor lectura. Cree un nuevo fichero XML dentro del directorio `/src/main/res/drawable` con el nombre `text_blue.xml` y dele el contenido:

```
<?xml version="1.0" encoding="utf-8"?>
<selector xmlns:android="http://schemas.android.com/apk/res/android">
    <!-- pulsado -->
    <item android:state_focused="false" android:state_pressed="true"
android:color="#00bfda"/>
    <!-- defecto -->
    <item android:color="#ffffff"/>
</selector>
```

De la misma forma que se hizo con el fondo gráfico del botón, se crean dos estados para el texto, uno para cuando esté pulsado con un color azulado y otro estado que se mostrará por defecto, con una letra blanca. Crearemos ahora un nuevo estilo y un nuevo botón para utilizar este color de texto. Dentro del fichero `styles.xml` añada el estilo (véase pantallas de la figura 21.5.):

```
 <!-- completo -->
    <style name="Button.Background.text.dinamic">
   <item name="android:textColor">@drawable/text_blue</item>
            <item name="android:background">@drawable/button_bg</item>
            <item name="android:shadowColor">#000000</item>
    <item name="android:shadowDx">3</item>
    <item name="android:shadowDy">3</item>
    <item name="android:shadowRadius">2</item>
    </style>
```

El nuevo estilo además de utilizar el fondo de vista y color de texto recién creados, añade un efecto de sombra negra a la letra para hacerlo un poco más vistoso. En el archivo de *layout* se añade un nuevo botón con este nuevo estilo y la aplicación está lista para probar de nuevo:

```
<Button
        style="@style/Button.Background.text.dinamic"
        android:text="Button.Background.text.dinamic" />
```

Por último vamos a asignar un estilo como tema de la actividad. Generaremos un nuevo estilo de modo semejante a los anteriores y lo llamaremos `AppThemeNew`.

```
<style name="AppThemeNew" parent="AppTheme">
        <item name="android:textSize">40sp</item>
        <item name="android:typeface">monospace</item>
    </style>
```

En él se modifica el tamaño de letra y el tipo de fuente. Para asignarlo a toda la actividad, se debe hacer desde el archivo `AndroidManifest.xml`, modificando la definición:

```
 <activity
            android:name="com.acme.styles.Styles"
            android:label="@string/app_name" android:theme="@style/
AppThemeNew">
```

Figura 21.5. Aplicación con el botón pulsado y sin pulsar

En la nueva versión se verá que ciertos botones han cambiado el tamaño de letra pero todos tienen el mismo estilo; dado que ninguno de los estilos aplicados tiene definido el tipo de fuente, se utilizará el definido en este estilo, pero en cuanto al tamaño, prevalece el aplicado mediante estilo de vista. Si se desea, también puede usare el tema para todas las actividades de la aplicación, simplemente hay que asignarle `android:theme="@style/AppThemeNew"` en la etiqueta `<application>` (véase figura 21.6.).

Para acabar la unidad, habíamos comentado que se podrían hacer imágenes mediante código XML. Supongamos que queremos hacer un fondo de uno de los botones que sea un gradiente entre una especie y rojo. Podemos valernos de las opciones que nos dan los *drawables* para generarlo; crearemos un

nuevo dichero llamado gradient.xml dentro del directorio `src/main/res/drawable` y le daremos como código, por ejemplo:

```
<shape xmlns:android="http://schemas.android.com/apk/res/android"
    android:shape="rectangle">
    <gradient
        android:startColor="#FFFF0000"
        android:endColor="#8000FFFF"
        android:angle="45"/>
    <padding android:left="10dp"
        android:top="10dp"
        android:right="10dp"
        android:bottom="10dp" />
    <corners android:radius="60dp" />
</shape>
```

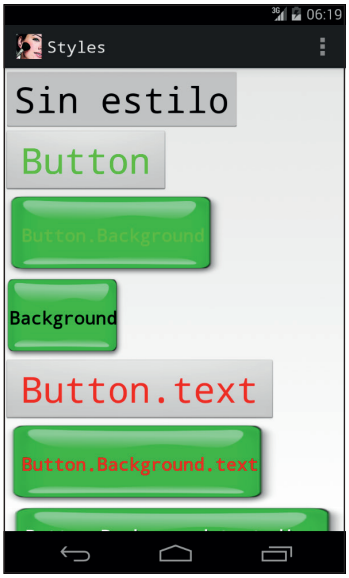

Figura 21.6. Tema aplicado a la actividad

Donde se define un gradiente entre dos colores con un ángulo de 45% e incluso se redondean los bordes mediante `android:radius`. Pero es que también podemos hacer composiciones con esta técnica, por ejemplo genere un nuevo fichero de nombre layers.xml y con el contenido:

```
<layer-list xmlns:android="http://schemas.android.com/apk/res/android">
    <item>
        <bitmap android:src="@drawable/ic_launcher"
            android:gravity="center" />
    </item>
    <item android:top="10dp" android:left="10dp">
        <bitmap android:src="@drawable/ic_launcher"
```

```
            android:gravity="center" />
    </item>
    <item android:top="20dp" android:left="20dp">
        <bitmap android:src="@drawable/ic_launcher"
            android:gravity="center" />
    </item>
</layer-list>
```

En este caso se ha generado una composición de tres imágenes (el icono de la aplicación) desplazadas entre ellas. Para ver el resultado modificaremos el *layout* para añadir estas imágenes:

```
<Button  style="@style/Button" android:text="Gradiente"
android:background="@drawable/gradient" />
<Button  style="@style/Button" android:text="Capas" android:background="@
drawable/layers" />
```

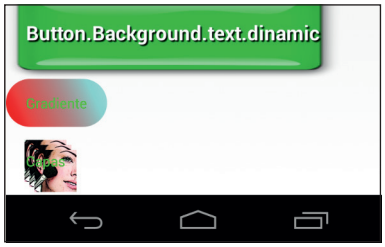

Figura 21.7. Botones generados mediante código

Como puede comprobar, si no es bueno manejándose con programas de dibujo, parte de la magia la puede conseguir también mediante diferentes archivos *Drawables*; le invito a que pruebe con diferentes valores y distintas etiquetas para ver los resultados. Existen demasiados modificadores en estos archivos para verlos en este capítulo, y versión a versión van apareciendo nuevos. Más información sobre creación de distintos tipos de *drawables* puede encontrarse en `http://developer.android.com/guide/topics/resources/drawable-resource.html`

22 Herramientas

En este capítulo aprenderá a:

- Ejecutar las herramientas de línea de comando.
- Usar los diferentes monitores disponibles.
- Mejorar los gráficos de las aplicaciones a través de las herramientas.
- Emular llamadas y posiciones para el emulador.
- Controlar la memoria y los procesos de la aplicación.
- Ver el tráfico de red del dispositivo.

Durante la instalación de Android Studio y sus complementos, se han instalado en el ordenador una serie de herramientas que ayudan a la hora de realizar los programas, algunas de ellas se utilizan de forma indirecta durante el desarrollo de la aplicación en el entorno de desarrollo y otras mediante llamadas desde la línea de comando. A lo largo de este capítulo descubriremos la utilidad y uso de las más importantes.
Si bien es cierto que en la primera instalación de Android Studio descargamos dichas herramientas, hay que tener en cuenta que Google va liberando nuevas versiones y es muy importante mantener estas herramientas al día para poder disfrutar de todas las funcionalidades que ofrecen. A lo largo de las versiones de Android, se ha cambiado la forma de actualizar las herramientas e incluso el directorio en el que se instalan. Para acceder a la actualización de las herramientas (y otros componentes) vale con pulsar sobre el icono correspondiente al SDK Manager desde el propio Android Studio  .

Herramientas de línea de comando

En las primeras versiones del *SDK*, las herramientas las podíamos encontrar en el directorio `<directorio_del_SDK>/tools` y actualmente las encontramos divididas en tres directorios dentro del directorio de instalación del Android Studio:

- `sdk/tools` : es donde se encuentran las utilidades que ayudan al diseño y depuración de la aplicación.
- `sdk/platform-tools` que contiene las aplicaciones para la comunicación con el dispositivo.
- sdk/build-tools: que se encarga de almacenar las herramientas que son dependientes de la plataforma y que ofrecen utilidades para las características que van apareciendo en las nuevas versiones de Android. Dentro de este directorio encontraremos un nuevo directorio por cada versión de desarrollo que tengamos; dentro de ellos es donde realmente se encuentran las herramientas. Normalmente no entraremos en él, puesto que ya lo hace Gradle por nosotros.

Otras herramientas como el *Android SDK Manager* (encargada de mantener al día nuestra instalación del *SDK*) y el *Android AVD Manager* (encargada de gestionar los dispositivos Android virtuales) que antes las podíamos encontrar en el directorio raíz de la instalación, ahora las encontramos dentro de `sdk/tools/lib`.

Aunque no haya sido consciente, el lector ya ha ido utilizando parte de estas herramientas (como el *adb* o *dx* por ejemplo) a lo largo de los ejercicios de este libro, siendo invocadas desde los programas Gradle de forma transparente para el programador.
A lo largo de los puntos siguientes entraremos un poco más en detalle en estas herramientas, viendo su utilidad y modo de invocación. Comenzaremos en orden alfabético por las herramientas disponibles dentro del directorio `sdk/platform-tools`, continuaremos por `sdk/build-tools` para acabar con `sdk/tools` pero el lector debe tener en cuenta que durante las diferentes versiones, se ha ido modificando la localización de cada una de las herramientas y es posible que lo siga haciendo, por lo que si no se encuentra en un directorio, se debe buscar en los otros.

adb (Android Debug Bridge)

Es una de las herramientas más importantes dado que permite la comunicación con los dispositivos, tanto virtuales como físicos. Es una utilidad con una estructura diseñada de modo cliente-servidor mediante tres componentes:

- Un servidor que se ejecuta en proceso de fondo en la máquina donde se desarrolla, se encarga de la comunicación entre el cliente y el demonio que se ejecuta en el dispositivo de desarrollo, tanto físico como virtual.
- Un cliente que se ejecuta en la máquina de desarrollo. Este cliente puede ser ejecutado desde la línea de comando o invocado de modo indirecto desde el entorno de desarrollo.
- Un demonio que se ejecuta en cada dispositivo.

El servidor *adb* se inicia de modo automático, y cada vez que se crea un cliente *adb*, este mira si el servicio *adb* está ejecutándose, en caso de que se encuentre disponible, comienza a realizase la comunicación, en caso contrario, se inicia el servidor *adb* y tras ello se comienza la comunicación, es decir, no debemos preocuparnos de arrancar el servicio. Una vez arrancado, el servicio escucha los comandos enviados por los clientes a través del puerto 5037 de la máquina de desarrollo. La comunicación con los demonios de los dispositivos se realiza mediante escaneo por parte del servicio *adb* de todos los puertos impares entre 5555 y 5585, y allí donde encuentra un demonio establece la conexión con el dispositivo, todo de modo transparente al desarrollador. La conexión se realiza a través de un par de puertos consecutivos, el puerto impar del *adb* y el puerto par inmediatamente inferior para las consolas, es

decir, si el demonio adb se ha conectado en el 5559, la consola de ese dispositivo se encuentra en el 5558.
Una vez que se ha establecido la comunicación, es posible utilizar los comandos de *abd* para controlar los dispositivos conectados. Gracias a las herramientas disponibles en Android Studio, la mayor parte de estos comandos pueden ser invocados directamente de forma gráfica desde el entorno de desarrollo, sencilla y transparente; no obstante, también es posible ejecutar comandos desde la línea de comando como veremos a continuación.
Abd soporta una multitud de comandos que van desde un simple listado de los dispositivos conectados a la máquina de desarrollo, hasta comandos para reiniciarlos. Los comandos se invocan mediante la estructura

```
abd [-d|-e|-s <numserie>] <comando>
```

Pero se ha comentado que se pueden tener tantos dispositivos conectados a la máquina de desarrollo como se quiera, así pues ¿sobre qué dispositivo se ejecuta el comando?, sobre el que se le indique mediante los parámetros -d, -e o -s. Se utiliza el parámetro –d para indicar que se ejecute sobre el único dispositivo físico que esté conectado (en caso de que sólo haya uno), -e para que se ejecute sobre el único emulador (en caso de que sólo haya uno) y –s para indicar el número de serie del dispositivo sobre el que se quiere ejecutar el comando; el número de serie de cada dispositivo se verá cómo obtenerlo más adelante.
La lista de todos los comandos disponibles se puede consultar mediante:

```
abd –h
```

no obstante revisaremos alguno de los más interesantes

- Listar dispositivos.

  abd devices nos muestra la lista de los números de serie de los dispositivos conectados a la máquina de desarrollo junto con sus estados

  ```
  abd devices
  ```

- Copiar ficheros y directorios al dispositivo.

  ```
  adb push <ruta_local> <ruta_destino>
  ```

La copia se realiza de modo recursivo, es decir si lo indicado es un directorio, se copiará el directorio y todo su contenido. Por ejemplo:

```
adb push img1.jpg /sdcard/miimagen.jpg
```

Copiar ficheros y directorios del dispositivo.Este comando es muy semejante al visto anteriormente, si bien el origen y destino se intercambian.

```
abd pull <ruta_destino> <ruta_local>
```

- Linea de comando del dispositivo.

Existe un comando muy especial que permite abrir una línea de comando (*shell*) en el propio dispositivo sobre el que ejecutar nuevamente comandos. Los comandos disponibles desde esta *shell*, se encuentran dentro del directorio /system/bin del dispositivo. Los comandos sobre la *shell* se pueden ejecutar uno a uno sobre ella y abandonándola tras cada ejecución:

```
adb [-d|-e|-s {<numSerie>}] shell <comando>
```

Por ejemplo:

```
adb -s emulator-5558 shell ls
```

O abrir una Shell interactiva sobre la que ir escribiendo cada comando, donde aparecerá un *prompt* donde escribir cada comando. Para abandonar este modo y volver a la línea de comando del sistema operativo de la máquina de desarrollo se debe pulsar la combinación de teclas `ctrl + d`.

```
adb [-d|-e|-s {<serialNumber>}] shell
```

Un ejemplo sería:

```
adb -s emulator-5558 shell
```

Una vez dentro de la shell, podremos ejecutar comandos directamente sobre el dispositivo. Dentro de los posibles comandos a ejecutar en la *shell*, merece la pena destacar uno que nos ayudará a conocer los contenidos de las bases de datos utilizadas en nuestras aplicaciones, se trata de:

```
sqlite3 <ruta_de_la_base_de_datos>
```

Al ejecutar este comando conectaremos con la base de datos proporcionada, permitiendo ejecutar comandos SQL directamente sobre ella y así gestionar tanto el diseño como el contenido de dicha base de datos.
Si el lector es un programador experimentado, se habrá encontrado en ocasiones en las que un programa que funciona correctamente llega a manos de alguien que consigue colgarlo en cuestión de minutos por haber realizado operaciones o procesos no tenidos en cuenta en la aplicación. En el mundo de las aplicaciones móviles estos circuitos "a controlar" son muchos y variados, por ejemplo que pasa si el usuario hace dobles pulsaciones en lugar de simples, o las hace muy rápido, o apoya tres dedos, o se va de la aplicación a media operación y vuelve... son operaciones a tener en cuenta. Para probar estas casuísticas podemos aporrear el dispositivo sin ton ni son como un mono.. o usar el mono que viene incorporado con las herramientas. Mediante el comando:

```
$ adb shell monkey [opciones] <número_eventos>
```

Se pueden enviar tanto al emulador como al dispositivo físico una serie de toques y eventos del sistema pseudo aleatorios para estresar la aplicación y ver si realmente soporta una interacción "alocada". Toda la información sobre el comando se puede obtener mediante

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

Un ejemplo de su uso sería

```
adb -d shell monkey -p com.acme.MyApp -v --throttle 500 --pct-touch 50 500
```

Donde -p sirve para indicar el paquete de la aplicación a probar, -v indica el nivel de detalle que queremos en la salida de información (cuantos más -v se añadan mayor nivel se obtiene), --throttle el tiempo en milisegundos de espera entre eventos, --pct-touch el porcentaje de eventos que tienen que ser pulsaciones y el último número es el número de eventos a ejecutar.
Aunque para la mayor parte de pruebas puede valer algo como:

```
adb -d shell monkey -p nombre.del.package.de.la.app -v 500
```

Si nos interesa probar una pantalla de la aplicación en particular, podemos navegar en el dispositivo o en el emulador hasta la pantalla deseada y ejecutar el comando, de modo que se comenzarán las pulsaciones sobre la pantalla actual.
Como ya se ha comentado anteriormente, `adb` es una de las herramientas más importantes que ofrece el SDK de Android e invito al lector que juegue con sus comandos ya que ofrece mucha ayuda a la hora de depurar aplicaciones.

Fastboot

Permite escribir una imagen del sistema operativo en el dispositivo mediante USB desde un ordenador. Por defecto esta opción es para los dispositivos de desarrollo de Google, ya que el USB está deshabilitado por defecto en el arranque de los dispositivos comerciales (aunque es posible habilitarlo para instalar lo que se conocen como ROMS "cocinadas").

aapt

Se llama indirectamente desde Android Studio y encarga de compilar los recursos de la aplicación como pueden ser el `AndroidManifest.xml` y los recursos XML haciéndolos accesibles a través de la clase R.java que se utiliza durante la generación del programa.

aidl (Android Interface Definition Language)

Esta utilidad permite crear las interfaces de comunicación entre cliente y servidor cuando se trabaja mediante comunicaciones entre procesos (IPC, *interprocess communication*). Está principalmente dirigida a programas que van a ser utilizados por clientes de otras aplicaciones y dado que Android no permite la comunicación directa entre ellas, se debe hacer mediante apoyo del sistema operativo, y el paso a través de él tiene que ser hacer bajo un formato de datos concreto, aquí es donde `aidl` entra en juego para facilitarnos la vida. No obstante, normalmente será llamada automáticamente por el entorno de desarrollo cuando sea necesario.

arm-linux-androideabi-ld

Es utilizado en compilaciones cruzadas. Básicamente es una herramienta para utilizar con el NDK de Android. Es semejante a i686-linux-android-ld y mipsel-linux-android-ld pero para diferente plataforma.

bcc_compat

Compilador de ficheros bytecode. Utilizado internamente por Android.

dexdump

Permite desensamblar y obtener información de las clases ya compiladas en formato DEX. Cuando se exporta una aplicación Android se obtiene un fichero `.apk` que es el instalable de las aplicaciones en este sistema operativo; este fichero no es más que un conjunto de archivos y directorios convenientemente comprimidos en formato `.zip`; de modo que para extraer la información, vale con renombrar de `.apk` a `.zip` y usar nuestro descompresor favorito. Esto generará una estructura de directorios semejante a:

- META-INF\
- res\
- AndroidManifest.xml
- classes.dex
- resources.arsc

Dentro del directorio res encontraremos todos los recursos de la aplicación, como por ejemplo los gráficos y los *layouts* (éstos últimos en formato *wbxml*, es decir, si se abrieran con un editor de textos no se vería el esperado XML). El fichero clases . `dex` es el que contiene las clases compiladas de la aplicación. Para desensamblar las clases el comando sería:

```
dexdump -d -f -h classes.dex >> salida.txt
```

Con ello tendremos la salida del desensamblado. Lo obtenido será *Dalvik ByteCode*, que dista bastante de las clases java compiladas en un primer instante, pero mediante herramientas de terceros (en estos momentos no existe ninguna oficial ni creo que llegue a existir nunca) como *dex2jar*, podemos estar un poco más cerca de conseguirlo.
Dexdump es otra de las herramientas que normalmente no llamaremos desde la línea de comando de modo explícito, sino que serán el propio entorno de programación quien se encargue de llamarla. El comando existe también dentro del sistema Android de modo que se puede ejecutar mediante `adb shell` para facilitar el desensamblado de clases ya incorporadas a los dispositivos.

dx

Es una utilidad que permite el paso de `.class` a *Android Byte Code*, que se usa internamente por Gragle. En caso de querer hacer la conversión manualmente, el comando a ejecutar sería:

```
dx --dex --output=dex_clases.dex clases.jar
```

Donde `dex_clases.dex` es el fichero a crear en *Android Byte Code* y `clases.jar` serían las clases compiladas en *Java Byte Code*. Lo normal es no tener que utilizarla nunca si se trabaja desde un entorno de desarrollo integrado como Android Studio.

i686-linux-android-ld

Es utilizado en compilaciones cruzadas. Básicamente es una herramienta para utilizar con el NDK de Android. Es semejante a arm-linux-androideabi-ld y mipsel-linux-android-ld pero para diferente plataforma.

Llvm-rs-cc

Es uno de los componentes que hacen posible le ejecución de scripts en Android. Su función es compilar y optimizar de modo offline los scripts, transformando así el script en *Byte Code* eficiente que puede ser llamado desde

Java. Los otros componentes que hacen posible el *RenderScript* de Android son el compilador online que realiza su función en cada momento que se necesita la ejecución del script (*libbcc*) y ajusta el *bytecode* al dispositivo sobre el que se esté ejecutando y la librería de ejecución (*libRS*) que da soporte a funciones matemáticas, i/o, etc...

mipsel-linux-android-ld

Es utilizado en compilaciones cruzadas. Básicamente es una herramienta para utilizar con el NDK de Android. Es semejante a arm-linux-androideabi-ld y i686-linux-android-ld pero para diferente plataforma.

android

Es una de las herramientas centrales del conjunto, está especializada en la gestión de proyectos Android , dispositivos virtuales y SDKs. Conocemos alguna de sus pantallas porque ya han sido utilizadas anteriormente. Su sintaxis es

```
android [opciones globales] action [opciones de acción]
```

Alguna de las acciones que podemos realizar son:

```
android avd -> abrir el gestor de dispositivos virtuales.
android sdk -> abrir el gestor de dispositivos virtuales.
android create project -> para crear un proyecto Android (con sus parámetros
requeridos).
android create avd -> para crear un nuevo dispositivo virtual (con sus
parámetros requeridos).
```

Toda la información sobre los parámetros requeridos para cada acción puede encontrarse en android –h, aunque recomendamos delegar su uso a Android Studio

ddms (Dalvik Debug Monitor Server)

Es el monitor de dispositivos desde donde podremos obtener datos de los procesos que se estén ejecutando en cada uno de los dispositivos conectados al ordenador, por ejemplo información de consumo de red o de memoria. *DDMS* también lo podemos encontrar encapsulado en la herramienta monitor, que comprende otros monitores y que se verá más adelante.

dmtracedump

Es una herramienta que permite realizar una representación gráfica, en forma de diagrama, de las llamadas realizadas en una aplicación usando como datos un fichero de traza. Para la creación del diagrama es necesario tener instalado el programa *Graphviz*, descargable de forma gratuita desde su web `www.graphviz.org`. Más adelante se aprenderá cómo obtener los ficheros de traza y cómo generar otro tipo de gráfico de llamadas.

draw9patch

Para ajustar los archivos gráficos a los distintos tamaños y densidades de pantalla, Android proporciona un método de creación de los gráficos llamado *Nine-patch* y que consiste en marcar de una cierta forma los archivos *png* a mostrar, de modo que se crean unas zonas susceptibles de ser estiradas y unas zonas de contenido.
Esta herramienta ayuda al programador a preparar y previsualizar cómo quedarían los gráficos en diferentes pantallas. No se entrará muy en profundizar a examinar la herramienta puesto que se hará en un apartado propio dentro de este capítulo.

emulator

Como habrá podido deducir el lector, `emulator` es un comando para controlar y gestionar todo lo referente al emulador de dispositivos virtuales. La sintaxis del comando es:

```
emulator -avd <nombre_de_avd> [<opciones>]
```

Donde `<nombre_de_avd>` es el nombre del dispositivo virtual a ejecutar en modo emulador, de lo que se deduce que es necesario tener creado previamente dicho dispositivo mediante el *AVD Manager* ya visto anteriormente.
Las opciones permiten modificar el comportamiento en cuestiones como imagen de disco a usar como *sdcard*, activar la aceleración hardware, o incluso cambiar el *kernel* de la imagen cargada. Casi la totalidad de las opciones están disponibles de modo gráfico a través de Android Studio. Tiene tres variantes `emulator-arm`, `emulator-mips` y `emulator-x86`, dependiendo del tipo de procesador que se quiera tener en el emulador.

etc1tool

Permite codificar y descodificar archivos *PNG* a formato estándar *ETC1(Ericsson Texture Compression*), que sirve para comprimir texturas, especialmente indicado para *OpenGL*. Las texturas en formato *ETC1* se salvan con extensión `.pkm`.

hierarchyviewer

Es una herramienta pensada para depurar y optimizar la interfaz de usuario. Mediante una representación gráfica a modo de diagrama de la jerarquía de la interfaz y una "lupa" para aumentar la representación de la interfaz para poder observar pequeños detalles a través de la utilidad *Pixel Perfect View* accesible desde el *HierarchyViewer*. Para abrir el HierarchyViewer desde línea de comando vale con ejecutar `hierarchyviewer`. Más adelante se comentará con más profundidad esta herramienta con interfaz gráfica.

hprof-conv

Se utiliza para convertir ficheros del formato *HPROF* (que es generado por Android a un formato estándar para poder utilizar en otras herramientas de *profiling*. Si se quiere utilizar la salida o entrada estándar, hay que indicar "-"como nombre de archivo. Su uso es:

```
hprof-conv <fichero_entrada> <fichero salida>
```

jobb

Es una utilidad para crear archivos OBB (*Opaque Binary Blob*, Blob binario opaco, (Blob es un objeto grande binario, *Binary Large Object*)). Se utiliza para generar archivos cifrados con recursos adicionales para la aplicación, como pueden ser imágenes, sonidos, vídeos, que pueden ser descargados aparte de la aplicación. Si realizamos un juego que ocupe mucho por su calidad gráfica y animaciones, lo que se debe hacer es crear toda la lógica del juego en el archivo `.apk` y dejar todo el tema multimedia (lo que ocupa) en archivos externos que se descargarán una vez instalada la aplicación. Estos archivos adicionales son los OBB. Por ejemplo para generar el archivo OBB de los ficheros y directorios contenidos en `/tmp/assets` de manera recursiva, con un nivel de API mínimo 14, con clave "secret_key" y para el paquete `com.acme.package` sería:

```
jobb -d /tmp/assets/ -o app_external_assets.obb -k secret_key -pn com.acme.
package -pv 14
```

lint

Esta herramienta escanea los proyectos Android en busca de posibles fallos o mejoras tanto en código como en el *layout* de las pantallas y utilización de recursos. Algunas de las comprobaciones que hace son:

- Búsqueda de problemas de rendimiento en los *layouts*.
- Problemas de accesibilidad.
- Errores del *Manifest*.
- Traducciones no hechas o cadenas de texto incrustadas en el código.
- Recursos no utilizados.
- …

Como argumentos, este comando toma uno o varios directorios donde encontrar proyectos Android. En caso de ser varios directorios, se deben separar por espacios. Si existe un directorio con varios proyectos se puede indicar el directorio padre y se chequeará todos los proyectos encontrados recursivamente en ese directorio.
Por ejemplo:

```
lint /projects/android
```

La salida obtenida será semejante a:

```
lint \workspace\ListDetail

Scanning ListDetail: ...................................................
.........................................................................
src\com\acme\listdetail\DriverListFragment.java:5: Warning: Don't include
android.R here; use a fully qualified name for each usage instead
[SuspiciousImport]
import android.R;
 ^
res\layout\activity_driver_detail.xml:1: Warning: This <FrameLayout> can be
replaced with a <merge> tag [MergeRootFrame]
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
^
0 errors, 2 warnings
```

Para proporcionar una lectura posterior del informe más cómoda, existe una opción para guardar la salida en formato HTML, para ello simplemente hay que añadir el modificador `--html` entre el comando y el directorio al escribir la orden en la línea de comando.
El comando `lint` se encuentra actualmente integrado en Android Studio (y otros IDE) para facilitar el uso por parte del usuario. Más adelante se verá más en profundidad esta opción.

mksdcard

Permite generar imágenes de disco con formato FAT32 para utilizar como tarjetas SD en el emulador. Hay que tener en cuenta que el tamaño de esta imagen ha de estar entre los 9 megas y las 1023 gigas por restricciones del propio emulador. Su sintaxis es:

```
mksdcard [-l etiqueta] <tamaño> <nombre_fichero>
```

Donde el tamaño puede expresarse en bytes simplemente expresando el tamaño como un número, en kilobytes si se añade una `K` tras el tamaño, en megas añadiendo una `M` o en Gigas si se añade una `G`.

monitor

Es una aplicación que aúna varias utilidades y que se encuentra disponible directamente desde Android Studio. Desde ella podemos acceder al *HierarchyViewer* (ya visto anteriormente), al *Pixel Perfect* y al *DDMS* (*Dalvik Debug Monitor Server*), que es una de las herramientas más potentes que ofrece el SDK de Android y que se verá con profundidad más adelante en este mismo capítulo.

monkeyrunner

Mediante esta utilidad y la API que proporciona, se permite crear código externo para controlar la aplicación. A través de *Python*, se pueden crear programas que lancen pulsaciones en pantalla, instalen aplicaciones, tomen capturas de pantalla… todo de modo automatizado. Aunque se parecen en nombre, `monkeyrunner` y `monkey` (vista anteriormente), no son la misma herramienta. Mientras que `monkey` se ejecuta en una *Shell* de `adb` y lanza las directamente sobre el dispositivo o emulador las pulsaciones y eventos del sistema, `monkeyrunner` trabaja desde el ordenador tomando el control del dispositivo a través de una API. Entre otras posibilidades, `monkeyrunner` ofrece:

- Comprobación de flujos de trabajo. Incluyendo en el script las pulsaciones y datos a introducir en cada elemento de la pantalla se pueden cubrir de principio a fin los flujos de trabajo, incluyendo capturas de pantalla por pasos.
- Comparar estabilidad y flujo de procesos mediante comparativa de capturas de pantalla

- Control de múltiples dispositivos. Se pueden conectar varios dispositivos y/o emuladores y lanzar procesos al unísono en todos ellos. Los emuladores se pueden lanzar también desde el script.
- Extensible mediante módulos *Python*. Es posible crear nuestros propios módulos de test en forma de *plugin* para ser reutilizados por varios scripts.

Su uso es:

```
monkeyrunner [opciones] SCRIPT_FILE
```

Donde en las opciones se pueden indicar aspectos como el nivel de información o puerto de conexión.

sqlite3

Sirve para gestionar bases de datos del tipo *Sqlite*.
Esta herramienta es semejante a la que se vio anteriormente como comando *Shell* del `adb`, salvo que la anterior era para bases de datos alojadas directamente en el dispositivo, mientras que estas son tratadas desde el ordenador.
La forma de invocar este comando es:

```
sqlite3 <opciones> fichero_base_datos <sentencia>
```

Si el fichero no existe, entonces se creará uno nuevo. En caso de que se informe la sentencia SQL, se ejecutará en el fichero indicado, de lo contrario se abrirá una *shell* donde introducir los comandos SQL. Toda la información sobre dichos comandos puede obtenerla de la página Web de Sqlite en `http://www.sqlite.org/sqlite.html`

traceview

Esta herramienta es en cierto modo sustitutoria de la ya vista `dmtracedump`. Realiza una visualización de los *logs* realizados por la clase *Debug*, para el depurado y mejora de rendimiento de la aplicación. La información la muestra en dos paneles, uno describe cuando se arranca y detiene cada método y *thread*, y el otro panel se encarga de mostrar datos sobre lo que ha pasado en cada método.
Para utilizar la herramienta hay que proveerla de un fichero de traza. Para obtener el fichero podemos operar de dos modos:

- Utilizar la característica de perfilado de la vista `DDMS` de la herramienta `monitor`. Más adelante en este capítulo se verá más en profundidad, Este método no es tan preciso como el siguiente.

- Incluir la clase *Debug* en el código e ir llamando a los métodos de comienzo y finalización de modo "manual". Este método es mucho más preciso que el anterior, puesto que se controlan exactamente los tiempos.

Se ejecuta:

```
traceview fichero_de_traza
```

zipalign

Se utiliza para optimizar el archivo de aplicación `.apk` resultante. Lo que hace es poner los datos no comprimidos en una dirección concreta a partir del comienzo del fichero, de modo que puedan leerse de una sola vez, y se ajusten mejor a la memoria del dispositivo, lo que hará que gasten menos RAM cuando se ejecute la aplicación.
El comando `zipalign` es utilizado de modo interno por Android Studio, pero si se trabaja haciendo todo el proceso de creación el *apk* de modo manual, hay que (frente a lo que dicta la lógica) utilizar esta herramienta después de haber firmado la aplicación, de lo contrario se perdería el alineamiento. Si lo que se quiere es utilizarla para alinear el fichero, su comando es:

```
zipalign [-f] [-v] <alineamiento> fichero_entrada.apk fichero_salida.apk
```

Donde `<alineamiento>` es el alineamiento en *bytes* y su valor por defecto es 4, `-f` permite la sobreescritura del fichero de salida y `-v` es para obtener información en pantalla.
Y para comprobar la alineación:

```
zipalign -c [-v] < alineamiento > fichero_entrada.apk
```

En este caso no se modifica el fichero

Herramientas gráficas

Ya se ha visto que alguna de las herramientas de línea de comando, derivaba a una interfaz gráfica para comunicarse con el usuario, en este apartado se entrará más en profundidad en estas herramientas a la vez que se descubren algunas nuevas.

Draw9patch

Para ajustarse a distintas densidades y tamaños de pantalla, Android propone un sistema de directorios en el cual podemos indicar que use unos gráficos para ciertos tamaños/resoluciones y otros distintos para otras, pero aún así,

es imposible ajustar los gráficos a todas las combinaciones posibles ni a todos los idiomas, porque por ejemplo si hubiera un botón cuyo texto en es "muñeca", en inglés es "doll", pero en alemán es "Schneiderpuppe", por lo que el botón en alemán se deberá estirar para adaptarse a su texto. Al estirar una imagen ésta pierde calidad, y dependiendo de cómo se estire, puede perder incluso su aspecto, por ejemplo un rectángulo con las esquinas redondeadas puede pasar de tener el aspecto A al aspecto B sólo por estirarlo,

Figura 22.1. Imagen normal y estirada con distorsión y sin ella.

Para ello Android proporciona una técnica llamada *Nine-Patch graphic* (gráfico de nueve recuadros) y consiste en dividir el gráfico en nueve porciones, dejando sin escalar las porciones de las esquinas, escalando en una dirección las porciones laterales y escalando en dos direcciones la porción central.
Realmente se trata de un archivo `.png`, al cual se le añade un pixel extra en cada uno de los lados creando así un borde alrededor de la imagen donde se indicarán las zonas escalables. De los cuatro laterales del borde creado el superior y el izquierdo servirán para delimitar las zonas escalables para ello se debe dibujar en la banda con pixeles negros, dejando transparentes las zonas que deben permanecer inalterables. Aunque se denomine *NinePatch*, se pueden marcar tantas zonas como se desee, simplemente hay que tener en cuenta que se escalarán en proporción a la sección marcada, así si tenemos una zona de dos pixeles y otra de cuatro, la zona de cuatro se estirará el doble que la de dos. Por otra parte, las bandas derecha e inferior, sirven para delimitar las zonas dentro de la imagen en las que podrá haber contenido, por ejemplo en los botones donde se utilice la imagen como fondo, indicar la zona donde debe colocarse el texto. Estas líneas son opcionales, y si no están marcadas, Android usará el alto y ancho de la imagen como espacio disponible. Para clarificar un poco las ideas vamos a ver un ejemplo.
Para realizar estas imágenes podemos ir a nuestro editor favorito, abrir la imagen, agrandar dos pixeles de alto y dos de ancho (centrando la imagen de modo que quede un pixel libre a lo largo de toda la imagen) a modo de marco y pintar sobre ese marco las zonas que deseamos que se puedan escalar y tener contenido, salvarlo en formato `.png` teniendo en cuenta que hay que poner de extensión `.9.png`.

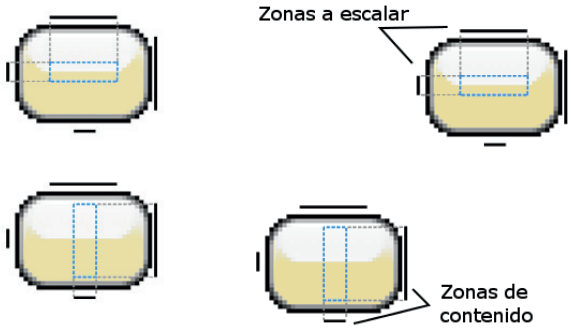

Figura 22.2. Imagen NinePatch con las zonas de marcas.

Atención:

La extensión tiene que ser `.9.png`*, de lo contrario en la aplicación aparecerá el gráfico y las bandas negras dibujadas en lugar de la imagen escalada.*

En estos momentos ya tenemos disponible el recurso para guardarlo en la carpeta `drawable` correspondiente, pero no sabemos a ciencia cierta cómo va a verse en la aplicación. Para facilitar la tarea existe la herramienta vista anteriormente llamada `draw9patch`. Al ejecutar la aplicación se muestra una pantalla donde se trabajará con la imagen que se desee. Para abrir la imagen se puede arrastrar sobre la pantalla de la aplicación o ir a través del menú File>Open 9-patch... Dependiendo de si la imagen es un `.png` el programa añadirá un borde de un pixel alrededor de la imagen para poder dibujar sobre él, mientras que si es un `.9.png`, aprovechará el de la imagen.
Una vez que la imagen ya está disponible en la aplicación, la pantalla se divide en dos secciones, a la izquierda está la zona de edición, donde se pueden dibujar las zonas escalables y de contenido, y en la derecha se puede ver el resultado de aplicar las zonas creadas. Para añadir los pixeles que delimiten las zonas vale con pulsar sobre el borde de 1 pixel de ancho definido alrededor de la imagen, y para borrar los píxeles se debe pulsar con el botón derecho en Windows o manteniendo apretado el *Shift* en caso de estar en Mac.
En la derecha se irán reflejando los cambios que se realicen en la imagen y así tener un control mayor del resultado final. Para ajustar más el resultado, existen otras opciones dentro de la aplicación, como `Show Patches` que marca sobre la imagen los recuadros definidos como áreas de trabajo, `Show Bad`

`Patches` para resaltar zonas que pueden ser distorsionadas debido a las zonas de estiramiento o `Show content` para mostrar sobre qué zona de la imagen se publicaría el contenido.
Una vez acabadas las marcas de las zonas, se debe guardar el archivo, la aplicación lo guardará con la extensión `.9.png` de modo automático para que el usuario no tenga que preocuparse de hacerlo, ya que recuerde que si no tiene esta extensión, durante la ejecución del programa Android no se ajustarán los tamaños.

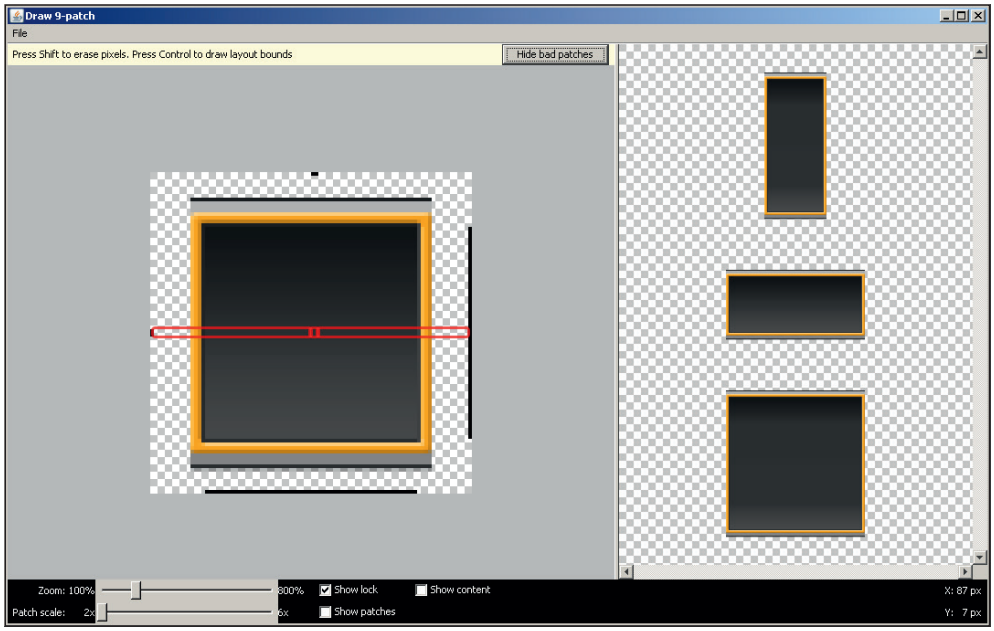

Figura 22.3. draw9patch en ejecución.

HierarchyViewer

Es una herramienta pensada para depurar y optimizar la interfaz de usuario. Mediante una representación gráfica a modo de diagrama de la jerarquía de la interfaz y una "lupa" para aumentar la representación de la interfaz y poder observar pequeños detalles a través de la utilidad *Pixel Perfect View* accesible desde el *HierarchyViewer*. Además *Pixel Perfect View* permite cargar gráficos como una capa adicional a la de la interfaz (por defecto al 50% de transparencia) y así poder comparar si la interfaz está quedando como debería, en el caso de que haya sido previamente diseñada sobre un *bitmap*.

Para abrir el *HierarchyViewer* desde línea de comando vale con ejecutar `hierarchyviewer`.

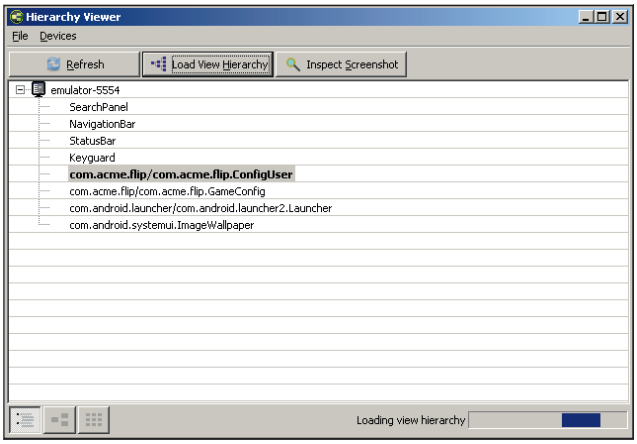

Figura 22.4. Hierarchyviewer mostrando la lista de dispositivos.

En ese momento se mostrará una pantalla con la lista de los dispositivos disponibles para trabajar. No hay que ponerse nerviosos si no aparece el dispositivo físico o al seleccionarlo no puede accederse a él, no es que se haya estropeado el USB ni nada de eso, simplemente que esta herramienta está pensada para trabajar con dispositivos de desarrollo Google (con versiones especiales de Android como el G1) o con el emulador por cuestiones de seguridad, y dado que el lector posiblemente tenga un dispositivo de venta al público general, deberá trabajar con el emulador. En la parte inferior de la pantalla se encuentran tres botones, el primero de ellos corresponde a la pantalla de lista de dispositivos y componentes de actividad, el segundo de ellos es la vista de jerarquía (*HierarchyViewer*) y el tercero es para ver la interfaz de cerca (*Pixel Perfect*), pero previamente a acceder a las pantallas mediante estos botones es necesario cargar las jerarquías mediante los botones superiores. Pulsando sobre View Hierarchy obtendremos los datos de jerarquía de la actividad que se esté mostrando en ese momento en el dispositivo o emulador, con datos de todos sus componentes y estructura. La pantalla se divide en cuatro secciones:

- A la izquierda una vista en árbol donde se puede ver la relación de cada elemento de la interfaz gráfica con el resto de elementos. Para buscar un nodo en concreto podemos valernos de la intuición y buscarlo en todo el árbol, o hacer uso de los filtros que están disponibles en la parte inferior

de la pantalla. Al introducir un filtro, se oscurecerán las clases cuya clase e identificador no coincidan con el filtro y se pondrán en azul las que coincidan, por lo que resultará más fácil hallarlas.

- En la parte superior derecha se encuentra una visión completa del árbol de vistas, para hacer más sencilla la navegación por él. El cuadro rojo mostrado puede ser arrastrado para moverse de una manera rápida por el árbol.
- En la parte central de la derecha se muestran las propiedades del objeto *View* que esté seleccionado. Las propiedades aparecen agrupadas por categoría y para acceder a ellas vale con desplegar pulsando sobre el nombre de la categoría.
- En la parte inferior de la derecha se muestra una representación esquemática del *layout*. Pulsando sobre cada una de las vistas en el árbol, se iluminará en esta representación, y viceversa, cuando se pulsa sobre alguna vista en la representación esquemática, se ilumina en el árbol de vistas.

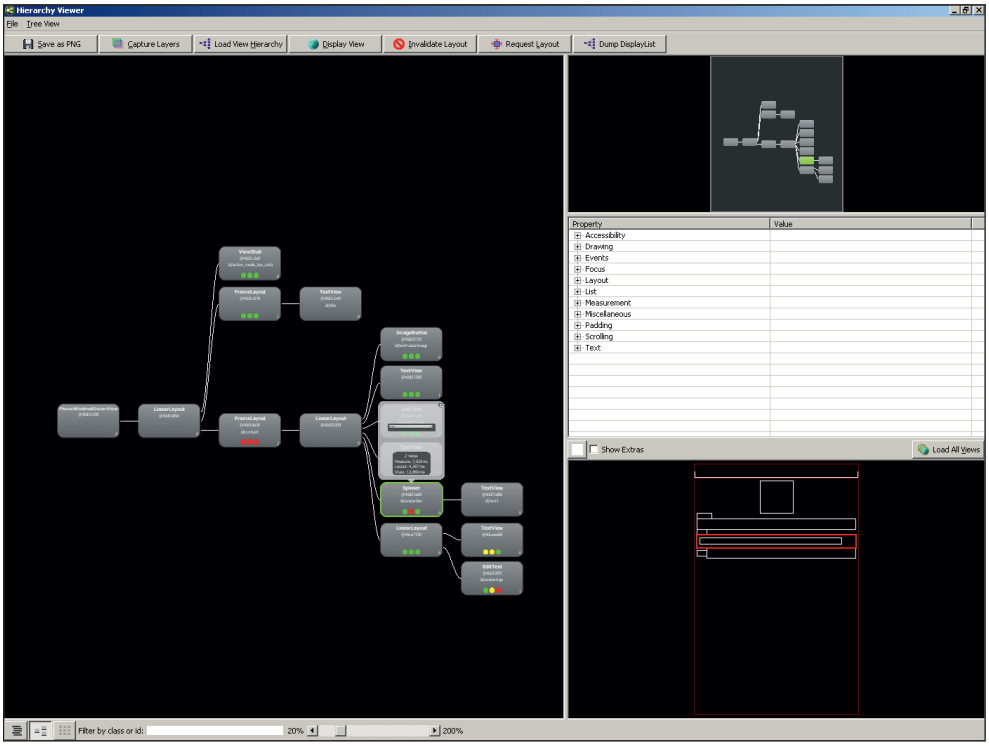

Figura 22.5. Hierarchyviewer en ejecucción mostrando la pantalla de usuario del ejemplo flip.

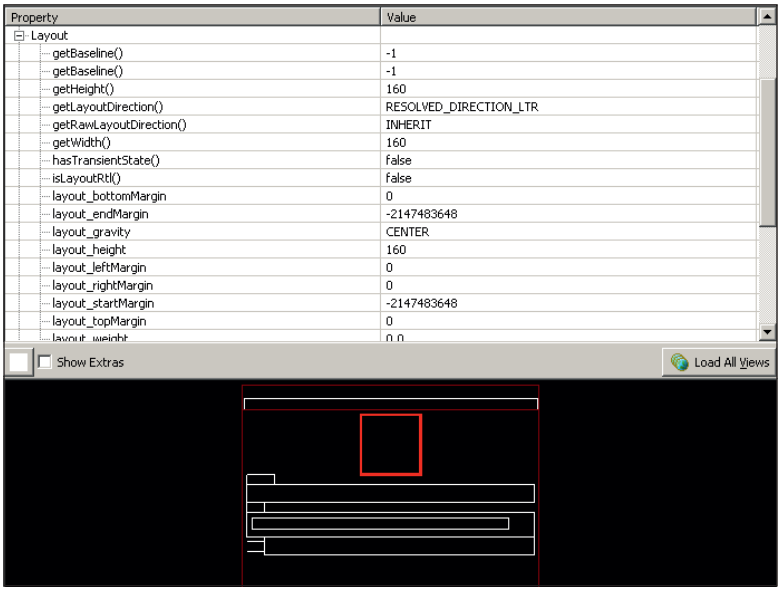

Figura 22.6. Esquema del diseño de la pantalla y propiedades.

Vamos a centrarnos ahora en cada una de las vistas que conforman el árbol, y en la información que se puede extraer de ellas. Cada una de las vistas del árbol viene representada por una ventana, que refleja la siguiente información.

- Nombre de la clase de la vista
- Dirección del objeto de la vista
- Valor del identificador de la vista en el *layout*, el `android:id`
- Indicadores de rendimiento a modo de semáforo. Existe un indicador para el tiempo de cálculo de la medida, del *layout* y del dibujado. Este indicador se calcula a partir del tiempo total de carga de la vista de la actividad, y dependiendo del tiempo de cada vista individual mostrará los distintos resultados:
  - Rojo: La vista es la más lenta del árbol, son las vistas que consumen más tiempo
  - Amarillo: está en el 50% de las más lentas del árbol.
  - Verde: está en el 50% de las más rápidas
- Índice de vista indicando que número de hijo es en su vista padre. Los índices comienzan en 0.

Para poder actualizar una parte del árbol y así poder recalcular los indicadores de rendimiento, hay que seleccionar la rama del árbol que interese refrescar, pulsar sobre el botón Invalidate Layout y volver a obtener de nuevo la jerarquía mediante el botón Load View Hierarchy.
Al pulsar sobre las ventanas de las vistas, aparece una nueva ventana en la parte superior con información adicional:

- La imagen de la vista (y de las vistas hijas) tal y como aparecen dibujadas en el dispositivo o en el emulador.
- Número de vistas representadas por el nodo incluyéndose a sí misma, es decir uno más que el número de hijos que tenga.
- Tiempos invertidos en su dibujado en pantalla, desglosados en tiempos de cálculo de las medidas, posicionamiento en el *layout* y dibujado propiamente dicho. Son los valores utilizados por los indicadores de rendimiento vistos previamente.

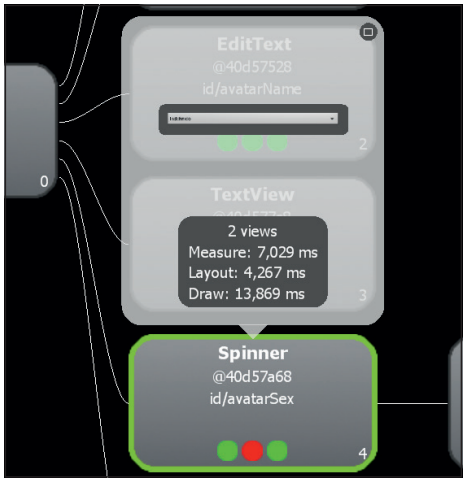

Figura 22.7. Detalle del elemento seleccionado en el árbol de jerarquía.

Hay que tener en cuenta que si la pantalla del dispositivo o emulador varía o se cambia de actividad, la aplicación no es capaz de detectar estos cambios y hay que refrescar los datos mediante el botón Load View Hierarchy.
También desde esta pantalla podemos exportar la información de la estructura a formato `.png`, mediante Save as PNG, y lo que es más interesante, la estructura gráfica de la vista a formato `.psd`, donde se guardará cada elemento gráfico en una capa Photoshop distinta y poder así jugar con las capas con alguna aplicación gráfica que soporte este formato.

La pantalla de *Pixel Perfect* (botón Inspect Screenshot) muestra una imagen aumentada de lo que se está viendo actualmente en la pantalla del dispositivo o emulador sobre el que se esté tratando, y además permite la carga de ficheros gráficos para sobre ponerlos en la representación de la vista de la pantalla, y así poder ver como quedaría un nuevo elemento o (en caso de que se haya hecho el ejercicio de "dibujar" primero cómo debería quedar la interfaz) comprobar si la interfaz está quedando según lo planificado (véase figura 22.8).

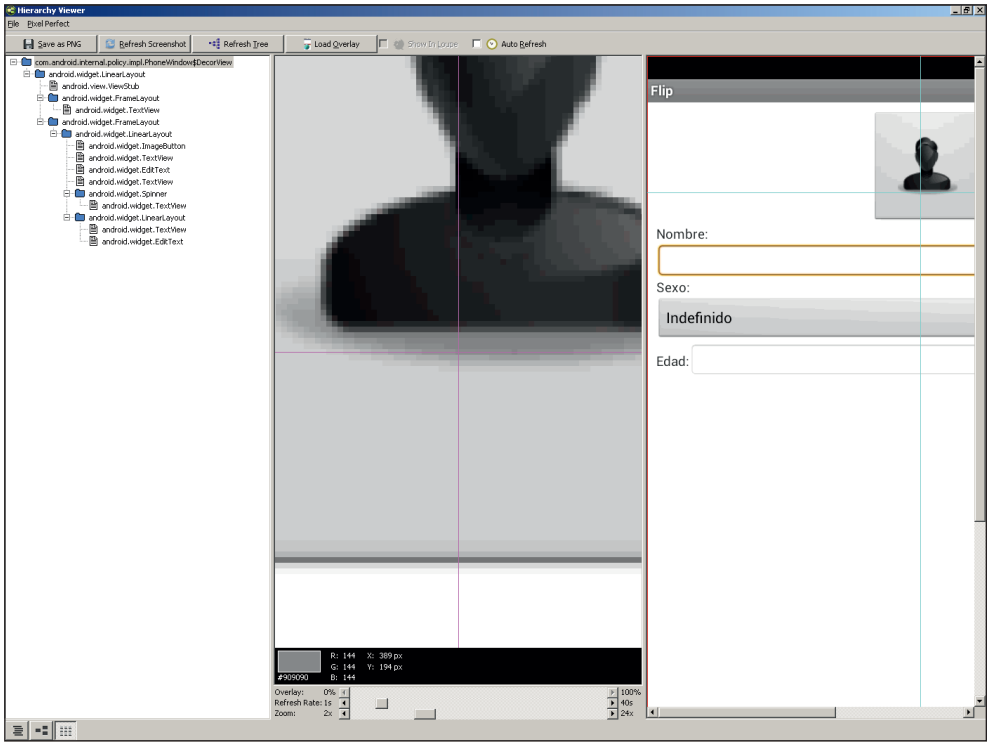

Figura 22.8. Pantalla de Pixel Perfect (Inspect Screenshot).

Esta pantalla muestra tres áreas de trabajo:

- A la izquierda se muestra la lista en forma de árbol con todos los componentes gráficos mostrados en pantalla, así como su interrelación. Si se pulsa sobre cualquiera de ellos, aparecerá iluminado en el panel de la derecha.
- Panel central muestra una imagen ampliada del panel de la derecha, junto a una serie de elementos visuales para facilitar su uso. Uno de estos elementos es la cruz magenta que se corresponde con la cruz verde del panel de la derecha, sirve para situarse en que parte de la pantalla se encuentra.

Existe también una cuadrícula cuyo tamaño se corresponde con el tamaño de los píxeles y pulsando sobre cualquiera de ellos se obtiene la información de él en la parte inferior de la pantalla; esta información incluye las coordenadas, el color del pixel tanto en RGB como en hexadecimal y una muestra visual del mismo color que el pixel seleccionado.

- El panel de la derecha es una representación gráfica del la vista que se esté mostrando en el emulador o dispositivo. La imagen del objeto seleccionado en el panel de la izquierda se muestra en un rojo intenso, mientras que las vistas hermanas y padres se muestran en un rojo pálido y las que no son nada se muestran en blanco. Es posible que se muestren otro tipo de rectángulos de otros colores, en caso de ser así, el blanco o negro (dependiendo el fondo de pantalla que esté activo) muestra el *padding* de la vista en caso de ser internos y el *margin* en caso de ser externos, el verde o violeta muestra el recuadro delimitador de la vista.

Del mismo modo que en la jerarquía, la previsualización de la vista del emulador no se refrescará de modo automático cuando se modifique la vista en el dispositivo conectado, sino que debemos ser el usuario quien se lo pida pulsando el botón Refresh Screenshot situado en la parte superior de la pantalla; a diferencia de la jerarquía, aquí podemos hacer que se refresque cada cierto tiempo mediante el *checkbox* situado arriba a la derecha. También es posible exportar la vista a un fichero `.png` mediante el botón Save as PNG.

Lint

Aunque ya se ha visto como ejecutar *Lint* desde línea de comando, siempre es más sencillo y cómodo hacerlo desde el propio entorno de desarrollo; para ello se encuentra integrado en Android Studio. Entre las características que ofrece `lint` podemos destacar:

- Arreglos automáticos para muchas de las alertas
- Ir a la línea de código del problema
- Configuración de severidades
- Obtener información en los distintos editores de Android Studio (Java, XML…)

En Android Studio, se tiene la particularidad que Lint se ofrece al programador junto con una serie de herramientas de inspección de código propia del entorno IntelliJ, consiguiendo un informe completo de avisos y puntos de mejora en el proyecto. Para ejecutar el análisis y obtener el reporte, se debe

pulsar sobre el menú Analyze > Inspect code…, donde podremos seleccionar qué se quiere comprobar, si todos los proyectos o unidades sueltas.
Tras la ejecución del análisis (y de *Lint* como parte del mismo), se pueden ver los resultados obtenidos en el panel Inspection.

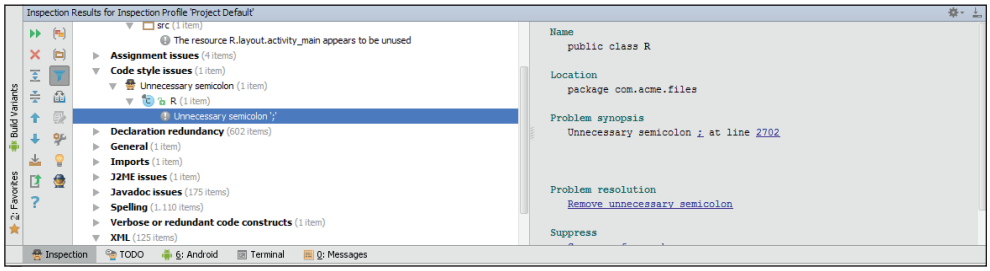

Figura 22.9. Ejecución y panel con los errores y avisos de Lint.

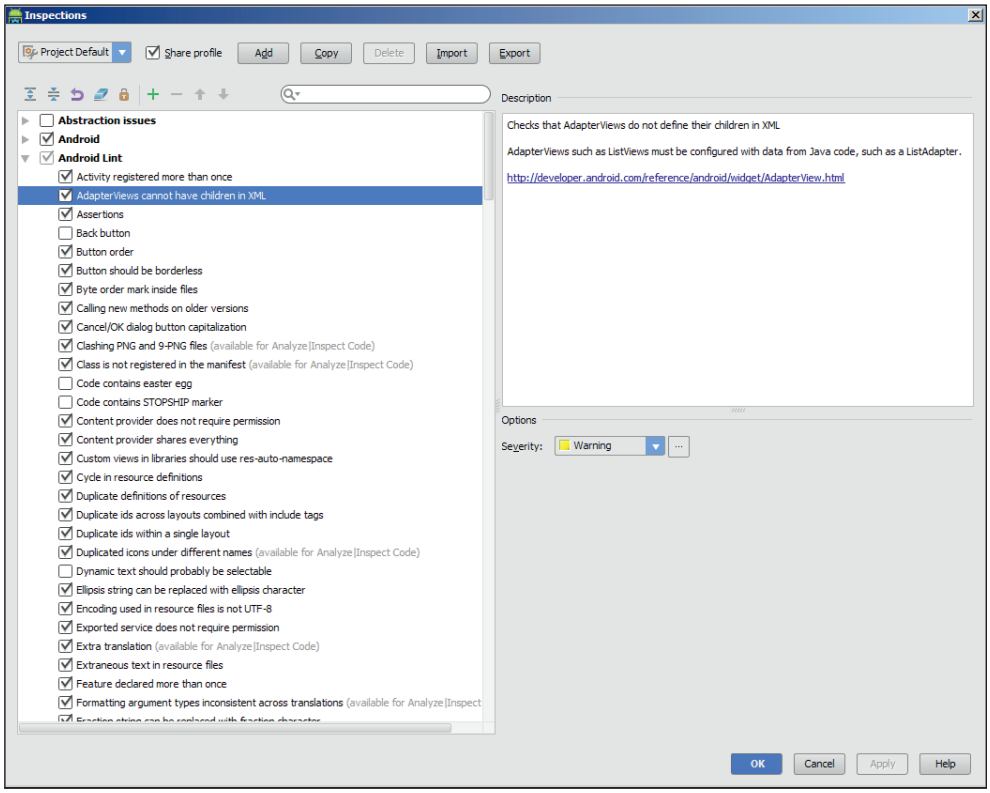

Figura 22.10. Pantalla de configuración de análisis del panel Inspection.

Dentro del panel Inspection, encontramos a su izquierda una botonera con diferentes acciones, como pueden ser opciones de filtro. En el árbol podemos ver las alertas agrupadas por categoría de la misma. Cada entrada se especifica mediante una pequeña descripción sobre la causa que ha generado la advertencia, el tipo de severidad se muestra mediante el icono situado a su izquierda. Si se selecciona una entrada, se obtiene una descripción más detallada, indicando la localización donde se ha producido el error o advertencia, una explicación de la causa, posibles maneras de suprimir el aviso…
Para saltar a la línea de código donde se ha producido la alerta, simplemente se ha de hacer click en el enlace disponible en el detalle de la entrada correspondiente.
Dentro de los botones de la izquierda, cabe mención especial a dos, el primero de ellos es que permite hacer un *quickfix* (arreglo rápido) del problema detectado. Por ejemplo en el caso de la figura 22.9, al aplicar el *quickfix*, se eliminaría el ";" sobrante que indica el problema. El otro de los botones que quería comentar, es que nos lleva a la pantalla de configuración de las opciones disponibles para el análisis del código, tanto de *Lint* como del testo de herramientas de análisis.

DDMS

La utilidad *DDMS* (Dalvik Debug Monitor Server) es en sí una recopilación de varias utilidades, de las cuales algunas ya se han visto y otras no; de hecho, muchas de las herramientas vistas anteriormente dan un mensaje de aviso de que está obsoleta esa llamada si se ejecutan individualmente, que se debe hacer desde *DDMS*. Se puede ver como panel de control de la "nave desarrollo", ya que desde aquí se puede controlar casi todo el emulador y el dispositivo físico que esté enchufado por USB. Para abrirlo desde línea de comando ya conoce como hacerlo, pero es mucho más cómodo tenerlo integrado en Android Studio ¿verdad? Para abrir la ventana diríjase a Tools > Android> Monitor (DDMS incuded) o pulse sobre el icono .
Cuando se ejecuta *DDMS*, conecta con `adb` y crean un monitor que se encarga de conocer cuando un dispositivo se conecta o se desconecta. Cuando recibe que se ha conectado un nuevo dispositivo, *DDMS* se encarga de recuperar toda la información necesaria sobre el dispositivo, y se comunica con el mediante `adb`.
Aunque las ventanas de *DDMS* varían ligeramente si se han accedido desde línea de comando o desde Android Studio, su funcionalidad es muy semejante y tiene un aspecto parecido al de la figura 22.11

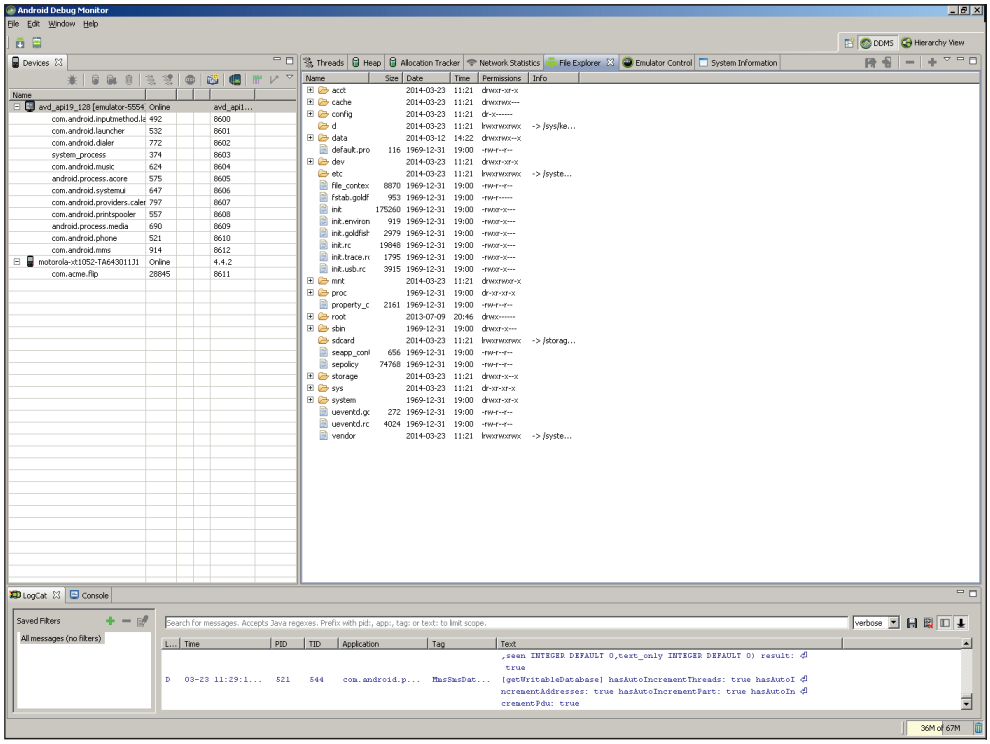

Figura 22.11. Pantalla de DDMS.

Como verá, está compuesta de múltiples vistas (y otras que se le pueden añadir) y diferentes botones que trataremos de explicar.
En la vista situada arriba a la izquierda, llamada Devices, se tiene una lista de dispositivos conectados mediante `adb`, con sus respectivos procesos. En esta vista además aparecen una serie de botones para actuar sobre el dispositivo/proceso seleccionado en la lista de la vista.

Figura 22.12. Botones de la vista Devices.

La función de cada uno de los botones es:

- Permite activar el depurador sobre un proceso que haya sido ejecutado sin el depurador.
- Controlan la gestión de memoria del proceso, respectivamente, el primero puede actualizar el *heap* de memoria del proceso (memoria ocupada por la máquina virtual en el proceso), el segundo sirve para

obtener un fichero *hprof* con información sobre memoria y procesos para *profiling* y el tercero genera un proceso de limpieza de memoria del proceso forzando el *garbage collector*.

- Gestionan los *threads* del proceso. Respectivamente el primero de ellos habilita el seguimiento de los *threads* en la vista de `Threads` y el segundo controla el comienzo y fin de la captación de datos para hacer un *profiling* desde el propio monitor, la primera pulsación lo inicia, la segunda lo detiene y muestra el resultado.
- Detiene el proceso seleccionado.
- Permite hacer capturas de pantalla.
- Obtiene un volcado de la jerarquía de objetos de la interfaz de modo semejante a lo realizado con `hierarchyview`. Este botón sólo es válido para obtener información de dispositivos con API superior a la 16.
- Genera un fichero de traza del sistema dado por unos parámetros de configuración.

Una vez visto de manera genérica la utilidad de estos botones, seguiremos viendo un poco más en profundidad alguno de ellos a la vez que el resto de las vistas restantes.

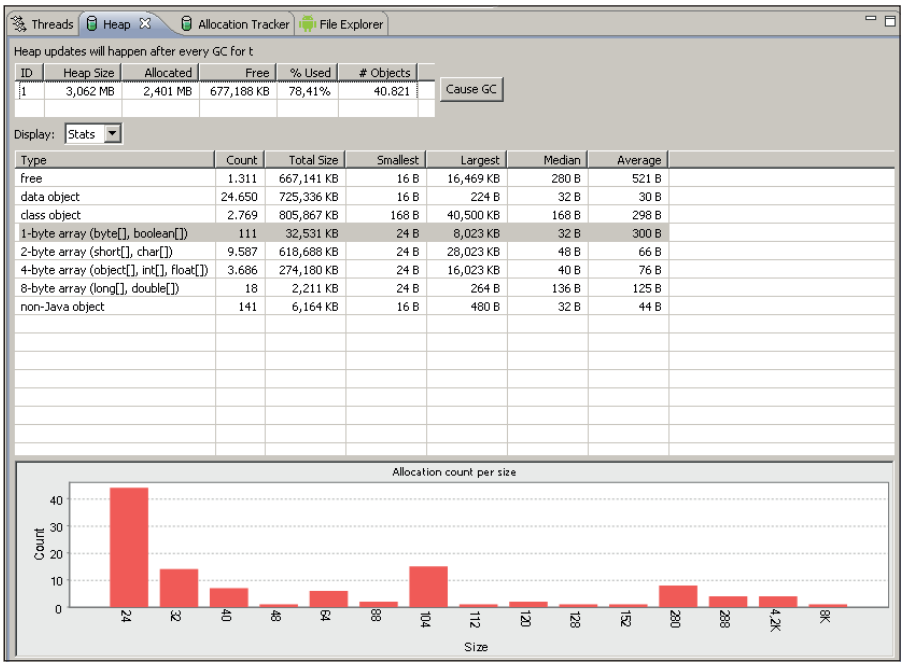

Figura 21.13. Vista `Heap` con información sobre la memoria.

La memoria disponible en los dispositivos móviles es limitada, y como tal debe controlarse su uso; para poder conocer la memoria *heap* que se está consumiendo por un proceso en un momento determinado, se utilizará el botón visto anteriormente  , con ello habilitaremos el acceso a la memoria *heap* del proceso, tras ello iremos a la vista Heap y al pulsar sobre el botón Cause GC se genera una llamada al *garbage collector* (recolector de objetos "basura" de la máquina virtual), y tras su ejecución se obtiene una lista de objetos con su tipo, y la memoria utilizada por cada uno de ellos. Para refrescar los datos no hay más que volver a pulsar sobre Cause GC. Pulsando sobre cada uno de los tipos de objeto de la lista, se visualiza un gráfico con el número de objetos asignados.

Para ver la información de los *threads* (hilos) de cada proceso, se debe seleccionar el proceso a analizar y pulsar sobre el botón  , que hará que se actualice dicha información en la vista `Threads`, para que no se actualicen más los *threads*, vale con volver a pulsar el botón.

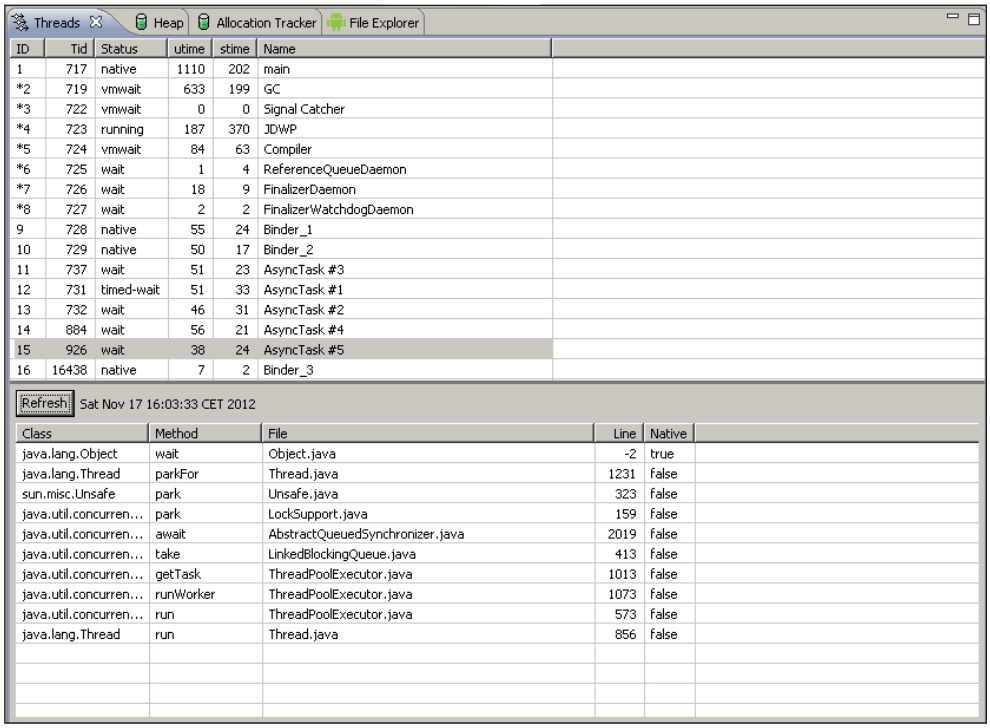

Figura 22.14. Vista `Threads` con información sobre los hilos del proceso.

Para un *profiling* de los métodos de la aplicación podemos utilizar el botón  , al pulsar sobre él se comenzará a obtener datos para el perfilado; es el momento en el que realizar las tareas de la aplicación que se quieran perfilar. Al terminar de hacer las tareas que se desean analizar se vuelve a pulsar sobre el botón y aparecerá una nueva ventana con la información sobre los métodos realizados entre la primera y segunda pulsación del botón. Entre la información obtenida, se encuentran los consumos de CPU y memoria por objeto de ejecución, tiempos consumidos...Cada uno de los nodos mostrados en el árbol de información puede ser expandido para obtener información adicional.

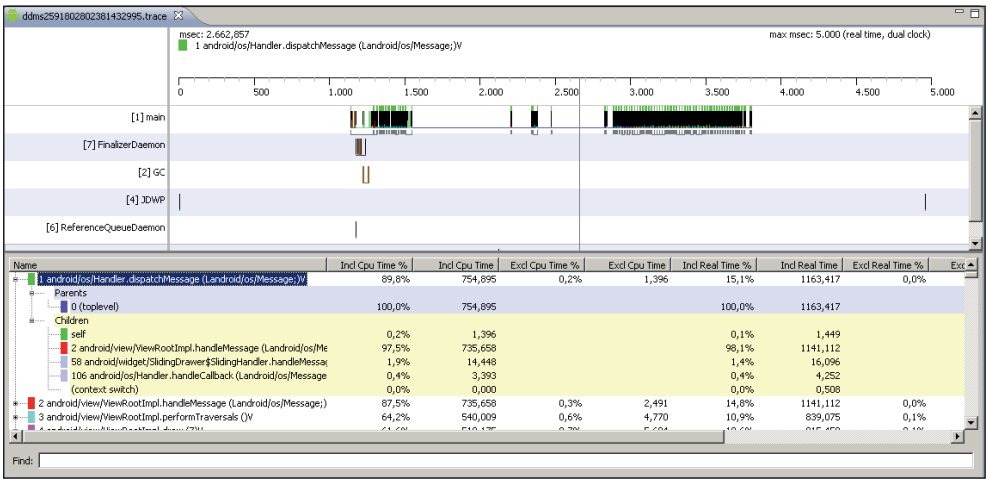

Figura 22.15. Información de perfilado recogida durante un proceso

El botón  permite obtener un volcado de la información jerárquica de la vista de una aplicación. En la ventana que se crea se muestra información sobre el *layout* de dicha vista, incluyendo clases implicadas y propiedades de cada objeto. Pulsando sobre el árbol de objetos se selecciona en la visualización el objeto mediante una caja roja, del mimo modo. si se pulsa sobre la visualización, también se selecciona el objeto (véase figura 22.16.).
El botón  genera un fichero de traza del sistema, cuya información depende de lo configurado en la pantalla de selección de la traza, permite exportar datos de traza a otros programas (véase figura 22.17.).
Para el control de los objetos instanciados se tiene la vista Allocation Tracker, que permite obtener información de los objetos que se instancian, tamaño que ocupan y *thread* donde son llamados. Para aplicaciones con problemas excesivo consumo de memoria, es una herramienta esencial. Para su uso, simplemente hay que pulsar sobre el botón Start Tracking de la vista Allocation Tracker, uti-

lizar la aplicación a analizar (concretamente la zona donde se tienen los problemas de memoria), y pulsar sobre el botón Stop Tracking para detener la obtención de datos. Se mostrará una lista con los objetos instanciados, la cantidad, el orden en el que se han llamado... y pulsando sobre cada uno de ellos se obtiene más información, como la clase y línea donde se instancia (véase figura 22.18.).

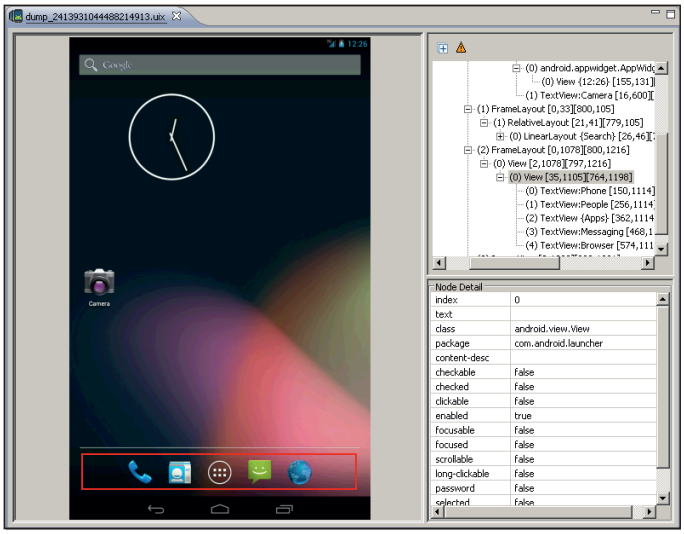

Figura 22.16. Volcado de la jerarquía de la pantalla principal.

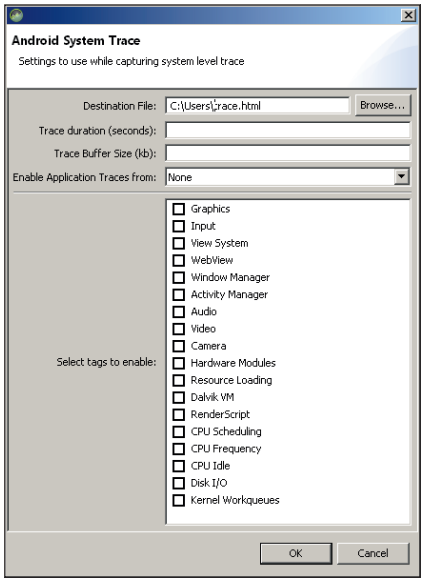

Figura 22.17. Pantalla de configuración de traza.

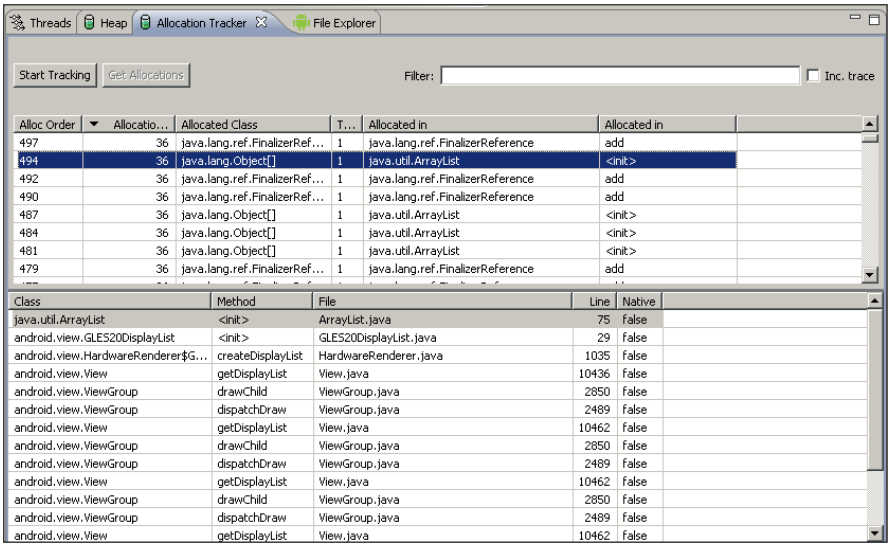

Figura 22.18. Vista de objetos asignados.

La vista File Explorer como su propio nombre indica, es un explorador de archivos de dispositivos, tanto virtuales como reales. Desde aquí se pueden ver el sistema de archivos, navegar por los directorios, e incluso extraer y añadir ficheros al mismo mediante los icono situados arriba a la derecha.

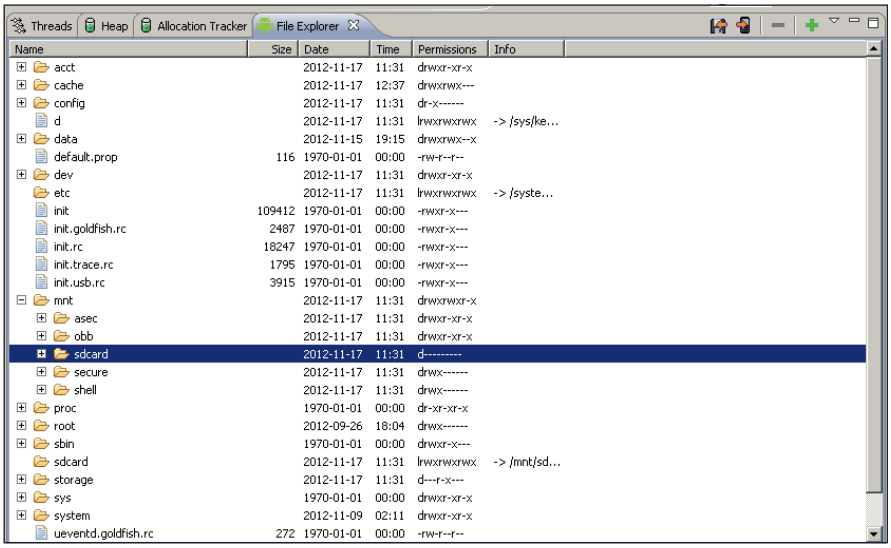

Figura 22.19. Explorador de archivos de DDMS.

La siguiente vista no suele estar disponible de forma estándar, pero en aplicaciones que utilizan las comunicaciones para obtener datos externos es muy útil, porque permite obtener información de la cantidad y tiempo de descarga de los mismos. Para hacerla visible en caso de que no lo sea, debe añadirla a la perspectiva mediante el menú Window > Show View y seleccionar la entrada. `Network Statistics`. Para comenzar la captura de datos transmitidos pulse sobre el botón Start. Es posible variar la frecuencia de muestreo de datos mediante el selector speed.

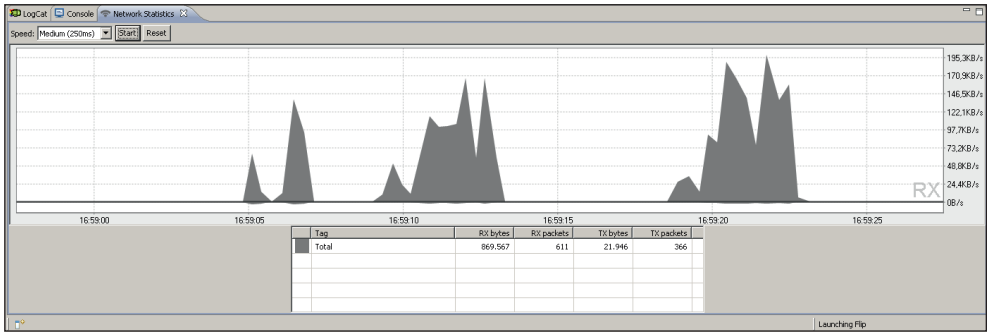

Figura 22.20. Estadísticas de red.

La vista `LogCat` permite recibir mensajes del dispositivo que esté conectado en ese momento al monitor.

Existen otra muchas vistas como las de control de openGL o de información del sistema, pero acabaremos de repasar las vistas de *DDMS* con la vista de control del emulador llamada Emulator Control. Desde esta vista se pueden controlar ciertos aspectos del emulador. La vista se halla dividida en tres secciones, la primera de ella es referente al estado de las comunicaciones del dispositivo, donde se puede variar la velocidad de conexión, latencia o tipo de conexión del que se dispone, así se pueden emular distintas circunstancias de uso del dispositivo.

La segunda sección está dirigida a acciones de telefonía para emular llamadas y mensajes de un número de teléfono dado. Para probarlo sólo tiene que introducir el número de teléfono del emisor, y seleccionar si se desea que sea llamada o SMS (en cuyo caso hay que escribir el mensaje) y pulsar sobre el botón Call, una vez terminada la llamada, se cancela con el botón Hang Up.

El último apartado es para el control de la localización en el emulador, donde se pueden cargar distintas coordenadas y en distintos formatos, con tal de emular que el dispositivo recibe esa localización por parte del GPS.

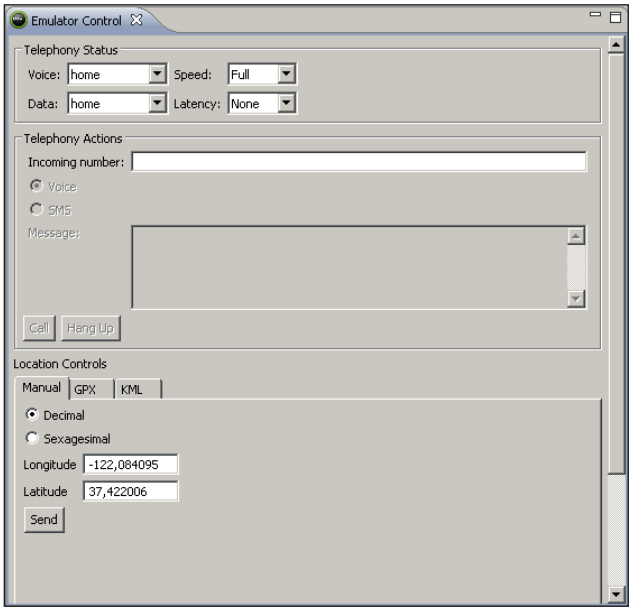

Figura 22.21. Control del emulador