Android

Guía de desarrollo de aplicaciones para Smartphones y Tabletas (2a edición)

Verdadera **guía de aprendizaje**, este libro acompaña al lector en el desarrollo de **aplicaciones Android para Smartphones y Tabletas táctiles**. Está dirigido a aquellos desarrolladores que posean unos conocimientos mínimos sobre programación orientada a objetos, lenguaje **Java** y entornos de desarrollo integrados como **Eclipse** o **Android Studio**, y cubre todas las versiones de Android, **hasta la versión 4.4 inclusive**.

El libro presenta el proceso completo de creación de aplicaciones, desde la**preparación del entorno de desarrollo** hasta la **publicación de la aplicación**, y describe una gran selección de funcionalidades provistas por el sistema Android.

A lo largo de este libro descubrirá, en primer lugar, **la plataforma Android**, instalará **el entorno de desarrollo** y creará, sin esperar más, su primera aplicación Android. Estudiará, a continuación, cómo construir la **interfaz de usuario** y conocerá los componentes fundamentales de la aplicación. Aprenderá a desarrollar **interfaces complejas** que se **adapten a las pantallas de Tabletas y Smartphones** y a construir sus propios componentes reutilizables. A continuación, se presentarán la **persistencia de los datos**, la **programación concurrente**, la **seguridad** y la **comunicación de red**. Se dedica un capítulo a explicar cómo **integrar las redes sociales** en sus aplicaciones.

Para publicar aplicaciones con la mayor calidad posible, descubrirá cómo**agregar trazas** y **probar su aplicación**. Por último, se le guiará paso a paso para **publicar sus aplicaciones** a usuarios del mundo entero.

La obra termina presentando cómo determinar la **geolocalización** y el uso de **sensores integrados** en los dispositivos Android. Se abordan, también, con detalle aspectos avanzados tales como la **creación de AppWidget**, la**protección de aplicaciones de pago** (LVL), las **compras in-app** así como las**comunicaciones NFC**. Tras su lectura, será capaz de desarrollar y publicar aplicaciones de calidad, nativas Android (en distintas versiones, hasta la versión 4.4 inclusive) para Smartphones y Tabletas táctiles.

Por último, como complemento y para ilustrar de forma práctica el propósito del libro, el autor proporciona **para su descarga** en el sitio web www.ediciones-eni.com un proyecto completo que reúne todas las nociones presentadas en el libro. **Interfaz de usuario**, **listas**, **procesamientos asíncronos**, **geolocalización**, **NFC**, etc.: todos los módulos del proyecto son funcionales, se pueden explotar directamente y proveen una sólida base para sus desarrollos.

Los capítulos del libro:
Prólogo - El universo Android - Primeros pasos - Descubrir la interfaz de usuario - Los fundamentos - Completar la interfaz de usuario - Componentes principales de la aplicación - La persistencia de los datos - Construir interfaces complejas - Concurrencia, seguridad y red - Redes sociales - Trazas, depuración y pruebas - Publicar una aplicación - Sensores y geolocalización - La tecnología NFC - Funcionalidades avanzadas

---

Sébastien PÉROCHON - Sylvain HÉBUTERNE

Sylvain HÉBUTERNE es Arquitecto Android. Especializado en la programación orientada a objetos tras 15 años de experiencia, diseña aplicaciones Android a título personal y para agencias de comunicación. Estos proyectos, muy diversos, le permiten explotar todo el potencial de la plataforma Android así como las funcionalidades más avanzadas que provee su última versión.

Sébastien PÉROCHON es el fundador de Mobiquité, empresa especializada en el desarrollo y la formación a desarrolladores de aplicaciones móviles (Android, iPhone, iPad). Tras diez años de experiencia en el desarrollo de software, la dirección de proyectos y la gestión de equipos, Sébastien Pérochon está también muy implicado en las comunidades de desarrollo sobre Android.

Introducción

Son pocos los sistemas que han conocido una progresión tan brillante como la que recientemente ha conocido el sistema Android.

Hace todavía poco tiempo, el sistema Android sólo estaba presente en smartphones. Más tarde, era posible encontrarlo, también, en televisores con conexión a Internet. En la actualidad, es el sistema operativo más extendido entre los smartphones.

Muchos son los motivos de este éxito. Uno de ellos es, sin duda, la amplia oferta de aplicaciones disponibles para su descarga (más de un millón en enero de 2014), que permiten a cualquiera personalizar su dispositivo Android.

Para que el lector pueda formar parte de este éxito, este libro pretende acompañarle desde sus primeros pasos en el desarrollo de aplicaciones nativas Android, que podrá publicar a continuación y, eventualmente, vender a usuarios de todo el mundo.

A quién se dirige este libro

Este libro está dirigido a todo aquél que se interese en mayor o menor medida por el universo Android y quiera ir más allá descubriendo cómo crear sus propias aplicaciones nativas Android.

Conocimientos previos necesarios para abordar este libro

El propósito de este libro es el desarrollo de aplicaciones Android en lenguaje Java dentro del entorno de desarrollo integrado Eclipse.

En particular, este libro se centra únicamente en el desarrollo de aplicaciones propias de Android.

Esto supone que se trata de un lector que posee conocimientos previos en programación orientada a objetos, particularmente en Java, y conocimientos previos en el uso del entorno de desarrollo integrado Eclipse.

Es por ello que no se ha considerado oportuno retomar por enésima vez el estudio del lenguaje en este libro para, así, concentrarse únicamente en la parte específica a Android.

La escritura de aplicaciones Android no requiere, para comenzar, un nivel elevado en Java. Basta con que el lector posea algunos conocimientos básicos sobre Java para recorrer este libro sin problema alguno y, sobretodo, para realizar sus propias aplicaciones Android.

En lo que respecta al uso del entorno de desarrollo integrado Eclipse, algunos conocimientos básicos son más que suficientes.

Objetivos a alcanzar

Este libro tiene como objetivo orientar al lector en sus primeros pasos en el desarrollo de aplicaciones Android y guiarle hacia funcionalidades más avanzadas.

Gracias a este libro, el lector, usuario de Android, se convertirá en diseñador de aplicaciones Android. Descubrirá los fundamentos del sistema Android y las bases del desarrollo de aplicaciones Android para smartphones y tabletas táctiles hasta la versión 4.4 incluida. Desde este momento, el lector podrá publicar sus propias aplicaciones Android de modo que estén disponibles para usuarios de todo el mundo.

Para ello, tras la presentación de la plataforma Android, el lector descubrirá cómo implantar un entorno de desarrollo para crear y ejecutar su primer proyecto Android. Le seguirá un primer estudio de la interfaz de usuario y de dos componentes principales como son las intenciones y las actividades. Un estudio complementario de la interfaz de usuario desvelará los elementos más importantes que pueden utilizar las aplicaciones.

A continuación, se abordarán otros componentes principales tales como los fragmentos y los servicios. Posteriormente se describirán varias soluciones de persistencia de datos. Después, el lector descubrirá la implementación de la programación concurrente en Android, noción esencial para producir aplicaciones de calidad, la gestión de la seguridad y la comunicación de red.

Se detallarán a continuación los medios para trazar, depurar y probar las aplicaciones. El lector descubrirá cómo publicar una aplicación, especialmente en Google Play Store.

Por último, este libro termina con un estudio de los sensores y medios de localización geográfica. Se abordan, también, funcionalidades avanzadas tales como la creación de un App Widget, la protección de aplicaciones de pago, el uso de NFC (*Near Field Communication*) así como el pago integrado.

Descarga

Los ejemplos de código ofrecidos a lo largo del libro sirven para ilustrar de manera individual cada una de las propuestas. No se trata de ejemplos de aplicaciones completas.

Por ello se ofrecen de forma complementaria proyectos Eclipse completos y funcionales. Estos proyectos permiten ilustrar de manera concreta las prácticas y los conceptos descritos en este libro, y también nociones más avanzadas, detalladas o no en el libro.

Estos proyectos están disponibles para su descarga en la página Información.

Información complementaria

La cantidad de temas relativos al desarrollo de aplicaciones Android es muy extensa, demasiado extensa para abordarla en un único libro. Se ha realizado una selección respecto a la cantidad de temas abordados en este libro. Por un lado, algunos temas se han tratado con detalle, otros se han abordado de forma más ligera, y algunos se han omitido de forma voluntaria.

La selección de estos temas se ha realizado con el objetivo de cubrir un conjunto coherente de temas que sea suficiente en lo relativo al aprendizaje del desarrollo de aplicaciones Android. Corresponde al lector, una vez haya adquirido los conocimientos básicos, ir más allá y descubrir por sí mismo las funcionalidades complementarias. El libro sugiere en varias ocasiones retomar el estudio de las clases y las funcionalidades que permiten profundizar en los distintos temas abordados.

A su vez, si bien la práctica totalidad de las funcionalidades de desarrollo de aplicaciones Android está disponible tanto en Eclipse como por línea de comandos, se ha optado por centrarse principalmente en el desarrollo de aplicaciones bajo Eclipse.

Recursos

Como complemento al libro y al código fuente disponible para descargar que le acompaña, existen numerosos recursos y documentación como soporte al desarrollador. Los principales son:

- El sitio oficial de desarrollo Android: http://developer.android.com/

  Este sitio Web incluye numerosos recursos, entre ellos, la guía de desarrollo y la documentación de las API (*Application Programming Interface*) del SDK (*Software Development Kit*) que pueden descargarse para consultarlas de forma local, tal y como se explica en el libro.

  Este sitio incluye también diversos recursos tales como proyectos de ejemplo, vídeos y un blog oficial que publica artículos muy útiles e interesantes para los desarrolladores.

- El foro oficial para desarrolladores Android: http://groups.google.com/group/android-developers

- Preguntas/respuestas entre desarrolladores Android:http://stackoverflow.com/questions/tagged/android

- La lista de bugs de Android: http://code.google.com/p/android/issues

- Un proyecto Android oficial ApiDemos que permite descubrir numerosas funcionalidades de Android así como consultar y estudiar el código correspondiente.

¡Feliz lectura!

Introducción

Plataforma integrada por primera vez en un smartphone (teléfono inteligente) aparecido en España en marzo de 2009, Android se ha emancipado rápidamente para conquistar numerosos dispositivos, móviles o no tales como netbooks (mini-ordenadores), tabletas táctiles y televisores con conexión a Internet, hasta el punto de haberse convertido hoy en día en uno de los sistemas operativos más importantes del mundo.

Para comprender bien cómo Android ha llegado aquí tan rápidamente, haremos una retrospectiva de su evolución antes de instalar y preparar el entorno de desarrollo.

Presentación de Android

Antes de sumergirse plenamente en el universo Android, es preciso presentar la plataforma.

Vamos a descubrir aquí los orígenes y la historia de la plataforma Android, sus distintas versiones y su distribución actual y pasada en los dispositivos Android. Descubriremos, por último, la arquitectura de la plataforma Android sobre la que se ejecutan las aplicaciones.

1. Open Handset Alliance™

Comprada por Google en 2005, Android era inicialmente una startup (empresa joven) que desarrollaba un sistema operativo para dispositivos móviles.

Desde entonces, varios rumores anunciaban la aparición de un teléfono Google llamado Gphone. No obstante Google preparaba, de hecho, mucho más que eso.

El 5 de noviembre de 2007 se anunció la creación del OHA (*Open Handset Alliance*): un consorcio creado por iniciativa de Google que reunía en sus inicios a una treintena de empresas. La mayoría de empresas eran operadores móviles, fabricantes de dispositivos, empresas industriales y fabricantes de software. El rol del OHA era favorecer la innovación en los dispositivos móviles proporcionando una plataforma realmente abierta y completa.

El mismo día, el OHA presenta oficialmente Android: la primera plataforma completa y abierta para dispositivos móviles. Esta plataforma incluía un sistema operativo, el middleware (las aplicaciones intermedias), una interfaz de usuario y aplicaciones de referencia.

Algunos días más tarde, el 12 de noviembre de 2007, el OHA anuncia la aparición del primer SDK(*Software Development Kit* o kit de desarrollo de software) Android que permite a los desarrolladores crear sus propias aplicaciones para la plataforma Android.

A principios de 2014, el OHA agrupa a más de ochenta empresas miembros.

2. Historia

El 21 de octubre de 2008, Google y el OHA anuncian la publicación del código fuente de la plataforma Android en open source, bajo licencia Apache 2.0.

Desde entonces era posible descargar el código fuente del sistema Android, compilarlo, instalarlo y ejecutarlo. También era posible contribuir a su evolución. Todo el código fuente y la información se encuentran en el sitio web Android Open Source Project: http://source.android.com/

En noviembre de 2008 apareció el Android Market, la tienda de aplicaciones de Google que permite a los desarrolladores y fabricantes de software poner a disposición de cualquier cliente de Android sus aplicaciones. En marzo de 2012, bajo la óptica de unificar sus servicios, Google renombra su tienda de aplicaciones como "Play Store". Esta tienda está disponible directamente en los dispositivos Android a través de la aplicación Play Store, y también en el sitio de Internet disponible en la siguiente dirección: https://play.google.com/store

Inicialmente la versión web sólo permitía consultar algunas aplicaciones de referencia. Más adelante se han incluido numerosas evoluciones, y en lo sucesivo Play Store permite consultar todas las aplicaciones disponibles, además de presentarle al usuario conectado la lista de aplicaciones instaladas en sus distintos dispositivos. También es posible instalar una aplicación sobre un dispositivo desde la aplicación web, desde un ordenador.

El primer smartphone Android apareció en octubre de 2008 en los Estados Unidos y en marzo de 2009 en España. Fue el HTC Dream G1. Su particularidad consistía en poseer un teclado físico deslizante. Su pantalla ocupaba 8 cm de diagonal y poseía una resolución de 320 x 480 píxeles. Contaba con 192 MB de memoria RAM, 256 MB de ROM, una cámara digital sin flash de 3,1 mega-píxeles, Wi-Fi y 3G.

Más adelante, muchos fabricantes han sacado al mercado un número impresionante de dispositivos que funcionan con Android. A principios de 2014, la empresa Samsung, líder del mercado en dispositivos Android en dicha fecha, poseía más de veinte modelos diferentes de smartphones y once modelos de tabletas. Según las estadísticas presentadas por Play Store existirían, actualmente, más de cinco mil dispositivos diferentes (contando todas las variaciones de cada fabricante) capaces de conectarse a Play Store.

Además de los smartphones y las tabletas (la primera tableta apareció a principios de 2011), existen dispositivos intermedios llamados "tabléfonos" (*phablet* en inglés) en los catálogos de los fabricantes. Estos dispositivos, cuyo nombre es la contracción entre teléfono inteligente (*smartphone*, en inglés) y tableta (*tablet*, en inglés), disponen de pantallas de gran dimensión (típicamente 6,5 pulgadas, unos 16 cm), están equipados con una tarjeta GSM (para utilizar la red del teléfono móvil) y, en general, pueden utilizarse con el dedo o con un lápiz táctil.

Los últimos dispositivos están ahora equipados con la tecnología NFC (*Near Field Communication*) que les permite comunicarse en el corto alcance, como por ejemplo para realizar un pago sin contacto. Estudiamos la tecnología NFC para Android más adelante en este libro.

Se ha recorrido mucho camino tras la aparición de la plataforma Android. Cada semana supone otra tanda de nuevos smartphones y tabletas. ¡Y sigue creciendo!

3. Versiones de Android

Igual que con el imperio romano, Android no se ha construido en un día. Es por ello que la progresión en el número de funcionalidades es similar a la de su cuota de mercado, simplemente impresionante.

Estas funcionalidades, mejoras y correcciones de bugs han ido apareciendo con el tiempo en las sucesivas versiones.

Veamos a continuación cada una de estas versiones desde la aparición oficial de Android y su reparto actual en el conjunto de dispositivos Android.

a. Cronología de versiones

En septiembre de 2008 apareció la primera versión de la plataforma Android, la versión 1.0. A continuación apareció la versión 1.1 en febrero de 2009; versión que se integró en el primer smartphone Android comercializado en España en marzo de 2009.

Algunos meses más tarde, Android pasó directamente a la versión 1.5, versión mayor que apareció en abril de 2009. Le seguirían tres revisiones de esta versión: la 1.5r2, la 1.5r3 y la 1.5r4.

La versión 1.5 aportaba nuevas funcionalidades tales como la detección de la rotación mediante un acelerómetro, la grabación de vídeo y la integración de los App Widgets (véase el capítulo Funcionalidades avanzadas - App Widget).

La versión 1.6 apareció en septiembre de 2009. Esta versión integró en particular el soporte de numerosas resoluciones y tamaños de pantalla distintos. También se integró la síntesis vocal. A continuación aparecieron dos revisiones, la 1.6r2 y la 1.6r3.

En octubre de 2009 apareció la versión mayor 2.0, aportando el soporte de HTML5 y de Exchange.

En diciembre de 2009 apareció la 2.0.1 y, en enero de 2010, la 2.1 que integraba los fondos de pantalla dinámicos. A continuación apareció la revisión 2.1r2.

Estas últimas versiones fueron rápidamente reemplazadas por la versión 2.2 que incluía la funcionalidad de hotspot (punto de acceso Wi-Fi) y supuso una mejora neta del rendimiento de 2 a 5 veces más rápido que la versión 2.1. Esta versión permite a su vez la instalación de aplicaciones en un almacenamiento externo como una tarjeta de memoria extraíble. Todavía apareció una revisión: la 2.2r2.

A finales de 2010 apareció la versión 2.3 que incluyó el NFC, soporte a varias cámaras, una interfaz

mejorada, mejor gestión de la energía, gestión de nuevos sensores, una mejora importante en el rendimiento, de la máquina virtual Dalvik, un teclado virtual más rápido con la funcionalidad copiar/pegar mejorada, soporte para VoIP (*Voice over Internet Protocol*) y SIP (*Session Initiation Protocol*)…

La versión 2.3 sólo estuvo disponible en el smartphone Nexus S de Google.

Fue preciso esperar a la versión 2.3.3 para que los demás teléfonos, el Nexus One entre otros, pudieran situarse al mismo nivel.

En febrero de 2011 apareció la versión 3.0. Esta versión marcó una etapa importante en el mundo Android. Estaba especialmente diseñada para tabletas táctiles con el objetivo de aprovechar sus pantallas de grandes dimensiones. Además de la integración de una nueva interfaz de usuario rica en nuevos componentes que incorporaba un nuevo tema gráfico, la versión 3.0 aportó el soporte de la arquitectura de procesadores de múltiples núcleos.

El año 2011 se publicaron varias versiones 2.3.x para smartphones, así como las versiones 3.1 (mayo) y 3.2 (julio).

En octubre de 2011 se publicó la versión 4.0, que unificaba los sistemas operativos para smartphones y tabletas. Las versiones 4.1 y 4.2 se publicaron, respectivamente, en julio de 2012 y octubre de 2012. La versión 4.3 apareció en julio de 2013, tras varias versiones 4.2.x.

La última versión mayor (a principios de enero de 2014), Android 4.4 Kitkat, se anunció el 3 de septiembre de 2013.

A estas versiones del sistema Android le corresponden niveles de interfaz de programación o simplemente API (*Application Programming Interface*) del SDK. A continuación aparece una tabla que muestra esta correspondencia.

Versión de la plataforma Android	**Fecha**	**Nombre**	**Nivel de API**
1.0	09/2008		1
1.1	02/2009		2
1.5	04/2009	Cupcake	3
1.5r2	05/2009		
1.5r3	07/2009		
1.5r4	05/2010		
1.6	09/2009	Donut	4
1.6r2	12/2009		
1.6r3	05/2010		
2.0	10/2009	Eclair	5
2.0.1	12/2009		6
2.1	01/2010		7
2.1r2	05/2010		
2.2	05/2010	Froyo	8
2.2r2	07/2010		
2.3	12/2010	Gingerbread	9
2.3.3	02/2011		10
3.0	02/2011	Honeycomb	11

3.1	05/2011	Honeycomb	12
3.2	07/2011		13
3.2.1	08/2011		13
3.2.2	09/2011		13
4.0.1	10/2011	Ice Cream Sandwich	14
4.0.2	11/2011		14
4.0.3	12/2011		15
3.2.4	12/2011	Honeycomb	13
3.2.6	02/2012		13
4.0.4	03/2012	Ice Cream Sandwich	15
4.1	07/2012	Jelly Bean	16
4.1.1	07/2012		16
4.1.2	10/2012		16
4.2	10/2012		17
4.2.1	11/2012		17
4.2.2	02/2013		17
4.3	07/2013		18
4.3.1	10/2013		18
4.4	10/2013	KitKat	19

Observe que las versiones mayores están tituladas con nombres de postres en orden alfabético: **C**upcake->**D**onut->**E**clair->**F**royo->**G**ingerbread->**H**oneycomb->**I**ce Cream Sandwich->**J**elly Bean->**K**itKat->...

El sistema Android asegura a las aplicaciones una compatibilidad ascendente. Esto significa que una aplicación pensada para funcionar sobre una versión mínima de Android 1.6 (API 4) funcionará automáticamente sobre todas las versiones de Android 1.6 (API 4) y superiores, y por tanto, por ejemplo, sobre la versión 4.0.1 (API 14).

b. Reparto de las distribuciones Android

Cada quince días, el sitio web Android (http://developer.android.com) proporciona el reparto de las versiones de Android en los sistemas que han accedido a Play Store en las dos últimas semanas. También aparecen los datos históricos que permiten seguir la evolución en este reparto.

Esta información está disponible en la siguiente dirección:

http://developer.android.com/resources/dashboard/platform-versions.html

Estos datos ayudan al desarrollador a la hora de escoger, con toda la información disponible, la versión mínima sobre la que deberá ejecutar su aplicación. Gracias a la compatibilidad ascendente de Android, esta aplicación funcionará automáticamente en las versiones superiores.

Cuanto menor sea la versión mínima de Android requerida por la aplicación, mayor será el número de dispositivos, y en consecuencia el número de usuarios y de compradores potenciales, que podrán ejecutar la aplicación. En contrapartida, la aplicación dispondrá de

menos APIs de SDK que las versiones más recientes. Es, por tanto, tarea del desarrollador encontrar el mejor compromiso entre las funcionalidades requeridas del SDK y la extensión del público objetivo.

En enero de 2014, la versión más extendida de Android es Android Jelly Bean que, contando todas las versiones secundarias, equipa al 60% del parque Android, 21% de los dispositivos ejecutan todavía Android Gingerbread.

Esta profusión de versiones diferentes, así como la cantidad de dispositivos con características diferentes (tamaño de pantalla, resolución de pantalla, memoria, etc.), resumido bajo el término "fragmentación", es el principal problema para los desarrolladores de Android. Veremos, a lo largo de este libro, cómo gestionar de la mejor forma esta fragmentación sin restringir las funcionalidades y el diseño de nuestras aplicaciones.

4. Arquitectura

Tras la versión Jelly Bean, el sistema Android está basado en el núcleo de Linux 3.0, y las primeras versiones estaban basadas en el núcleo 2.6. Este núcleo tiene en cuenta la gestión de las capas inferiores, tales como los procesos, la gestión de la memoria, los permisos de usuario y la capa de hardware.

Sobre este núcleo, se sitúa la capa de bibliotecas principales del sistema proporcionadas por los fabricantes. Éstas, de bajo nivel, están escritas en C y/o C++. Proporcionan los servicios esenciales tales como la gestión de la visualización 2D y 3D, un motor de base de datos SQLite, la reproducción y la grabación de audio y vídeo, un motor de navegador web…

Las funcionalidades ofrecidas por estas bibliotecas las recoge y utiliza la capa superior bajo la forma de bibliotecas Java. Éstas proporcionan bibliotecas y componentes reutilizables específicos a dominios particulares. Encontramos, por ejemplo, bibliotecas de acceso a bases de datos, de telefonía, de localización geográfica, de comunicación sin contacto de corto alcance…

Android proporciona a su vez una multitud de bibliotecas Java básicas estándar tales como las del paquete java.*.

Por último, la capa de más alto nivel es la de las aplicaciones. Estas aplicaciones son las que se incluyen por defecto, tales como la aplicación de *inicio* (llamada a menudo *escritorio*), la aplicación que permite ejecutar otras aplicaciones, el navegador web, la aplicación de telefonía... Pero también son las aplicaciones específicas creadas por desarrolladores, ¡de los que usted formará parte muy pronto!

Si bien es posible desarrollar aplicaciones basadas en código C y/o C++ mediante el NDK (*Native Development Kit*), sobretodo para mejorar el rendimiento, no se abordará este tema en este libro. Utilizaremos únicamente las API Java que proporciona el SDK y que bastan, en la gran mayoría de casos, para crear todo tipo de aplicaciones estándar. Si se tratara de aplicaciones con un gran consumo de recursos gráficos tales como juegos 2D o 3D, el escenario sería distinto.

Por defecto, cada aplicación se ejecuta en una máquina virtual Java alojada en un proceso Linux dedicado. Esta máquina virtual es específica a la plataforma Android y está especializada para entornos embebidos. Se le llama máquina virtual Dalvik.

Entorno de desarrollo

Incluso aunque es posible desarrollar completamente una aplicación mediante un editor de texto básico y una línea de comandos para realizar la compilación, resulta mucho más cómodo utilizar un entorno de desarrollo, que le facilitará la escritura, la compilación y la depuración de sus aplicaciones.

Existen varios entornos de desarrollo, cada uno con sus ventajas e inconvenientes. Veremos, en este capítulo, cómo instalar dos de ellos: Eclipse/ADT (*Android Development Tools*) y Android Studio.

A lo largo del libro, utilizaremos como referencia el entorno de desarrollo Eclipse, al que habremos agregado el plug-in ADT.

Veremos en el siguiente capítulo cómo instalar y configurar un emulador de Android, herramienta esencial para el desarrollador.

1. Requisitos previos

Para poder desarrollar aplicaciones Android, es preciso asegurarse de que el puesto de desarrollo es compatible con los criterios requeridos.

Sistemas operativos soportados:

- Windows XP (32 bits), Windows Vista (32 o 64 bits), Windows 7 (32 o 64 bits).
- Mac OS X 10.5.8 o superior (x86 únicamente).
- Linux (librería GNU C (glibc) 2.11 o superior; las distribuciones de 64 bits deben ser capaces de ejecutar aplicaciones de 32 bits).

Se requiere un mínimo de 200 MB de espacio en disco sólo para la instalación del SDK, de sus herramientas y de alguna versión de la plataforma Android.

Se requiere el JDK (*Java Development Kit*) de Java 6 o superior. Si el puesto de desarrollo todavía no está disponible, es posible descargar el JDK de la plataforma Java SE (*Java Platform, Standard Edition*) en la siguiente dirección:

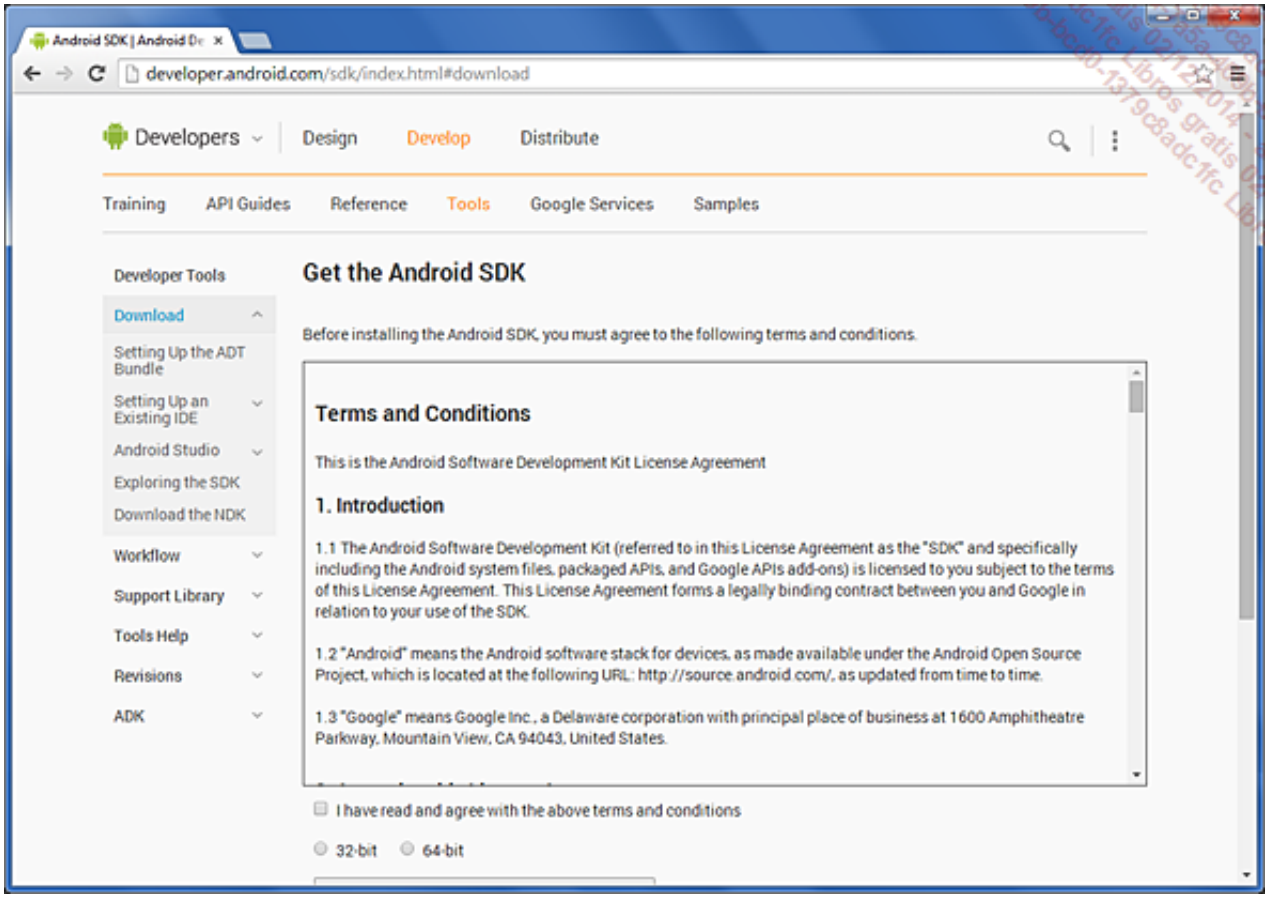
http://www.oracle.com/technetwork/java/javase/downloads/index.html

2. Eclipse y el Plug-in ADT

Eclipse es un entorno de desarrollo bien conocido en el universo Java. Gratuito, dispone de una importante colección de plug-in, y está indicado para el desarrollo de aplicaciones Android. Para ello, es necesario agregar el plug-in ADT, que integra numerosas funcionalidades específicas del desarrollo Android, que van desde el asistente para la creación de un proyecto Android hasta la exportación de la aplicación final.

Google provee una versión "Bundle" de Eclipse, que integra de manera nativa el plug-in ADT, sin requerir una instalación o configuración particular. Esta versión es la que vamos a instalar y utilizaremos a lo largo de este libro.

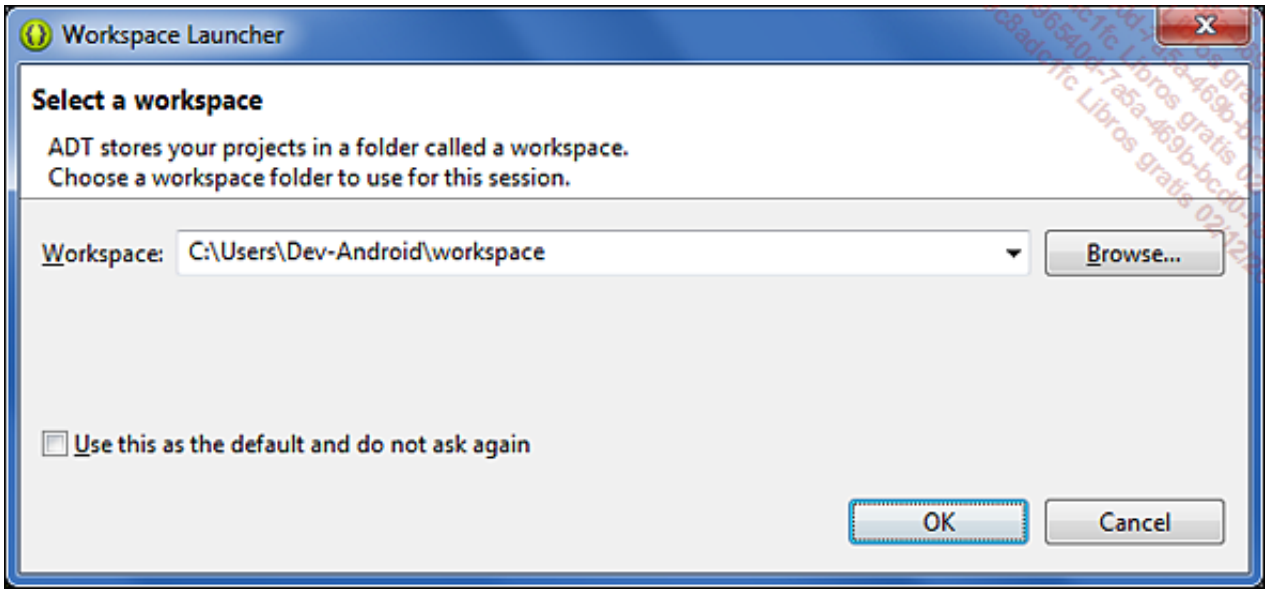
La descarga del bundle puede llevarse a cabo desde la siguiente dirección:http://developer.android.com/sdk/index.html



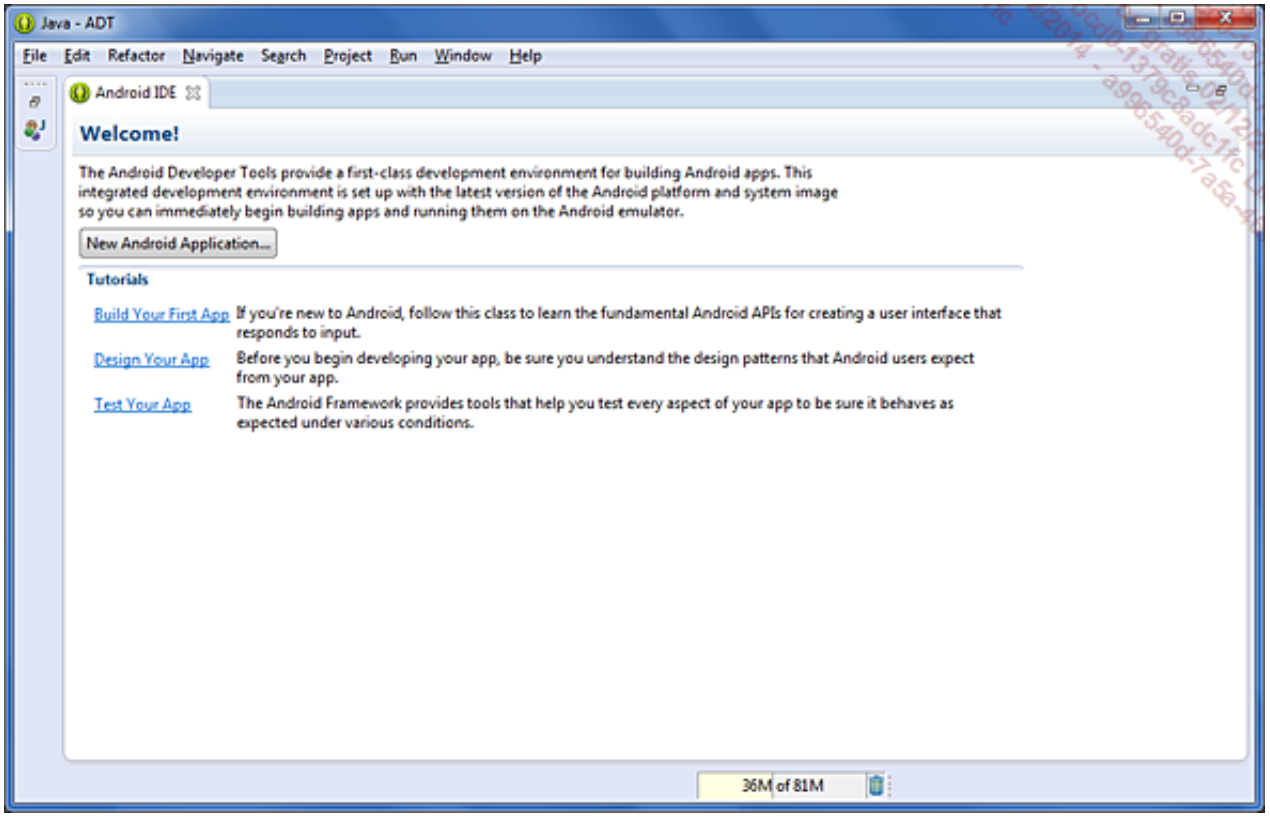
Una vez descargado, basta con descomprimir el archivo distribuido en forma de archivo comprimido en una carpeta de su elección: recomendamos seleccionar una carpeta cuya ruta de acceso no incluya espacios.

La descompresión produce dos carpetas, "eclipse" y "sdk", así como un archivo ejecutable, "SDK Manager". Basta, ahora, con ubicarse en la carpeta "eclipse" y ejecutar la aplicación.

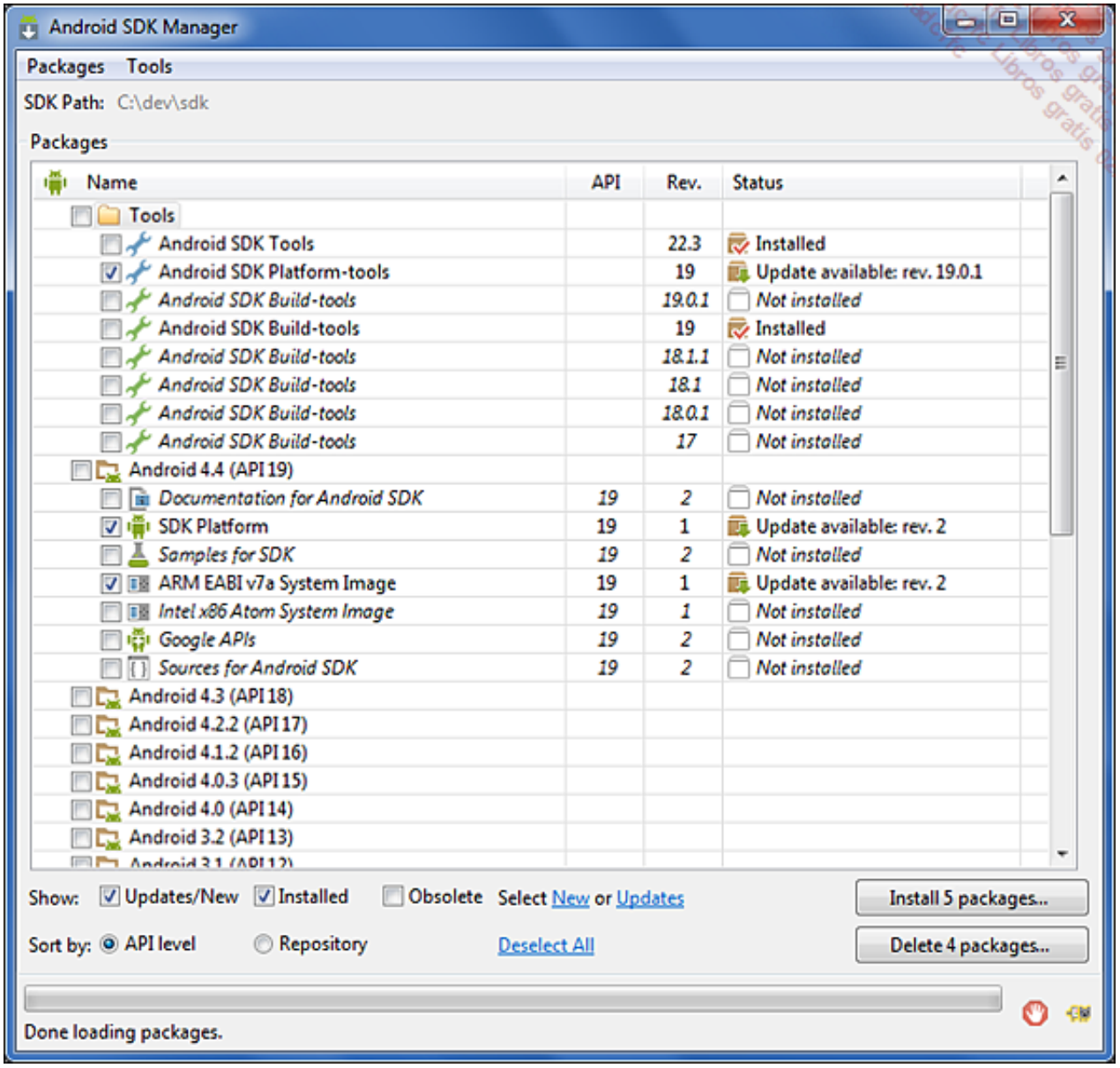
Tras la visualización de la pantalla de inicio "ADT", se abre un cuadro de diálogo que le invita a seleccionar la carpeta de trabajo en la que se almacenarán sus proyectos (workspace). Seleccione la carpeta de su elección, donde los proyectos se crearán como subcarpetas.



Haga clic en **OK**. Se abre el espacio de trabajo por defecto de Eclipse.



Antes de crear un primer proyecto, vamos a ver cómo instalar varias versiones de Android, lo que es indispensable para poder probar una aplicación en diferentes configuraciones. Para ello, basta con ejecutar la aplicación Android SDK Manager desde la carpeta de instalación del bundle (también es posible ejecutar el SDK Manager a través de Eclipse, en el menú **Window** - **Android SDK Manager**, o mediante la barra de herramientas de Eclipse).



La aplicación Android SDK Manager se conecta directamente con los servidores de Google para descargar la lista de paquetes disponibles. La pantalla de la aplicación muestra, a continuación, una lista de paquetes, organizados según la versión de Android afectada. Para instalar un paquete, o un elemento de un paquete, basta con marcar la opción correspondiente.

Es probable que la lista que se muestre sea diferente a la de la captura de pantalla. Esto no tiene importancia.

De arriba a abajo, los paquetes disponibles contienen:

- **Android SDK Tools**: la última versión de las herramientas dependientes de la plataforma Android.
- **Documentation for Android SDK**: la última versión de la documentación del SDK Android. Se trata de una copia local de la documentación disponible en línea.
- **SDK Platform Android...**: SDK de varias plataformas Android desde la versión 1.5 (API 3), cada una en su última versión. Observe que ciertas versiones, como la 2.0 (API 5) y la 2.0.1 (API 6), no están disponibles y han sido reemplazadas por la versión 2.1 (API 7).
- **Samples for SDK API...**: proyectos Android de ejemplo correspondientes a distintas versiones del SDK.

La categoría **Extra** contiene módulos complementarios y opcionales que permiten extender las

funcionalidades del entorno de desarrollo o de las aplicaciones Android.

Destacaremos los siguientes componentes:

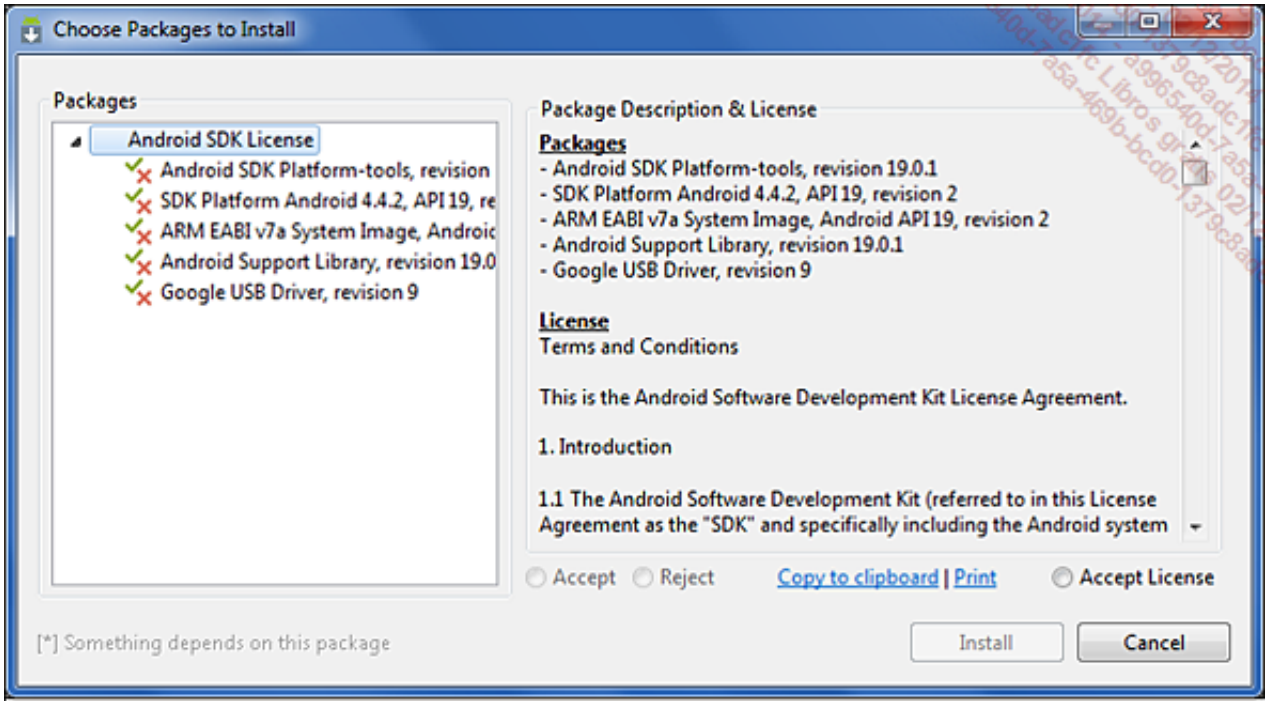
- **Google Play services**: módulos que proporcionan funcionalidades suplementarias ofrecidas por la empresa Google tales como los mapas geográficos de Google Maps (véase el capítulo Sensores y geolocalización - Google Maps), la gestión de cuentas de tipo Gmail, interacciones con Google+, etc.
- **Google Play Licensing Library**: librería que permite proteger las aplicaciones de pago mediante la verificación en línea de la licencia (véase el capítulo Funcionalidades avanzadas - Proteger las aplicaciones de pago). Esta funcionalidad implica que la aplicación esté publicada en la Play Store.
- **Google Play Billing Library**: librería que permite comprar contenido digital o funciones suplementarias, desde una aplicación. Este servicio se llama In-app Billing (pago integrado o in-app). Esta funcionalidad implica que la aplicación esté publicada en Google Play Store. Estudiaremos esta librería en el capítulo Funcionalidades avanzadas.
- **Android Support Library**: librería que ofrece ciertas funcionalidades aparecidas con la versión 4.x a las aplicaciones destinadas a versiones anteriores. Utilizaremos esta librería en el capítulo Componentes principales de la aplicación.

También es posible seleccionar la documentación del SDK para descargarla sobre el puesto local. De este modo, la consulta de esta documentación podrá realizarse directamente en local, mejor y más rápido que en línea. Ocurre lo mismo con los ejemplos. Es posible instalar y desinstalar cada uno de estos componentes en cualquier momento, a excepción de las herramientas **Android SDK Tools** que permiten, entre otros, administrar la instalación de los componentes y deben estar por tanto siempre presentes.

Se desaconseja instalar todos los paquetes disponibles: ralentizará considerablemente el inicio de la aplicación - cada paquete se carga durante el arranque de Eclipse. Es preferible seleccionar los paquetes correspondientes a las versiones principales de Android, e instalar más adelante un paquete concreto si fuera necesario.Recomendamos instalar las versiones Android 4.4 (API 19). Android 4.1 (API 16) así como la versión 2.3.3 (API 10).

- Marque las opciones correspondientes a los paquetes deseados.
- Una vez seleccionados los componentes, haga clic en el botón **Install x packages**.

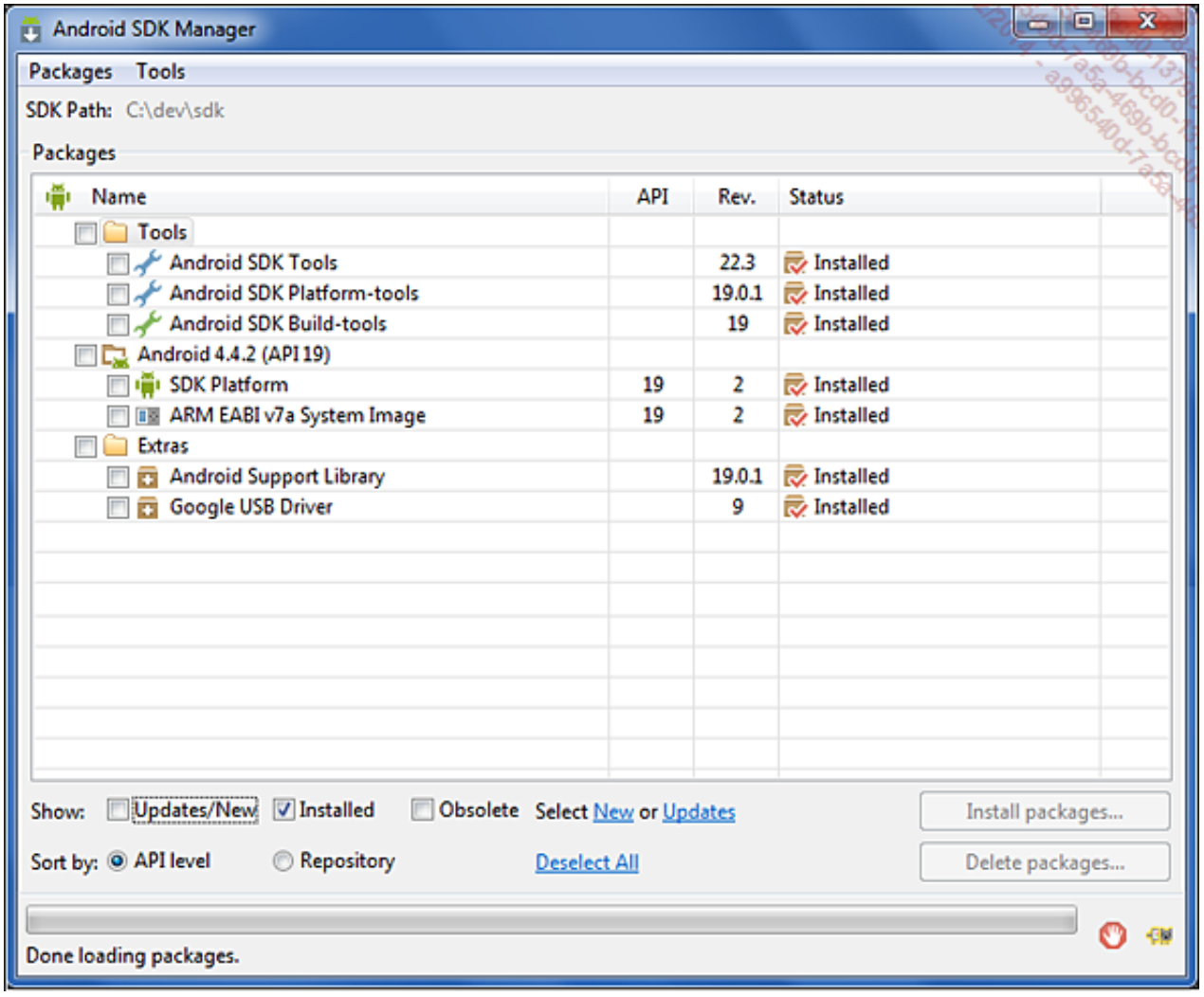
Aparece una nueva ventana detallando los componentes seleccionados.



- Marque la opción **Accept License** y haga clic en el botón **Install** para ejecutar la descarga y la instalación de los componentes. Dependiendo de su conexión a Internet y del número de paquetes seleccionados, esta operación puede resultar algo larga.

Una vez finalizada la instalación, estos componentes deben aparecer en la lista de componentes instalados.

- Para verificarlo, seleccione únicamente **Installed** en la opción **Show** de la ventana de Android SDK Manager para mostrar la lista de componentes instalados.



- Cierre a continuación la ventana.

¡Listo! El entorno de desarrollo Eclipse está preparado.

Instalación de Android Studio

Google ha anunciado, tras la conferencia Google I/O 2013, la aparición de un entorno de desarrollo específico para Android, llamado Android Studio. Este entorno de desarrollo, basado en el IDE IntelliJ IDEA, de la empresa JetBrains, está disponible de manera gratuita, aunque a día de hoy se ofrece en versión "Early Access". Por este motivo, es preferible ser cauto y meticuloso antes de iniciar un proyecto de producción (incluso aunque la tendencia parece mejorar, las primeras actualizaciones de Android Studio han presentado problemas en el pasado).

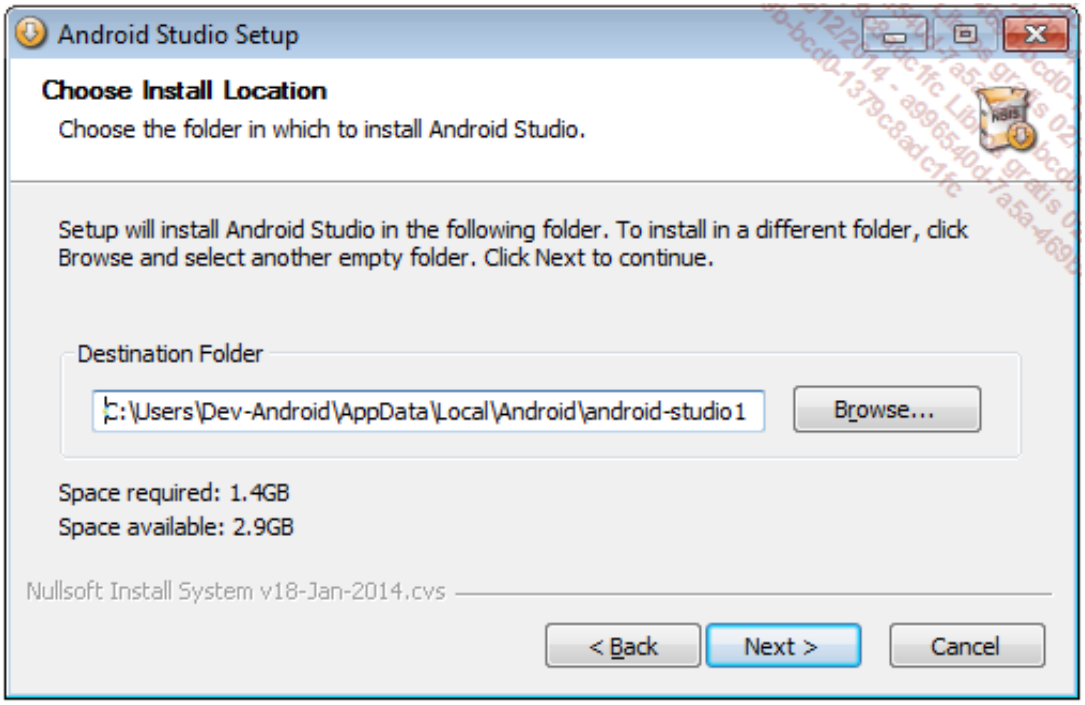
A fecha 1 de febrero de 2014, la versión más reciente es la v0.4.2. El vínculo para descargar el archivo de instalación es el siguiente: http://developer.android.com/sdk/installing/studio.html

Comience leyendo y aceptando la licencia de uso (Terms and Conditions) y, a continuación, descargue la versión propuesta para su sistema operativo.

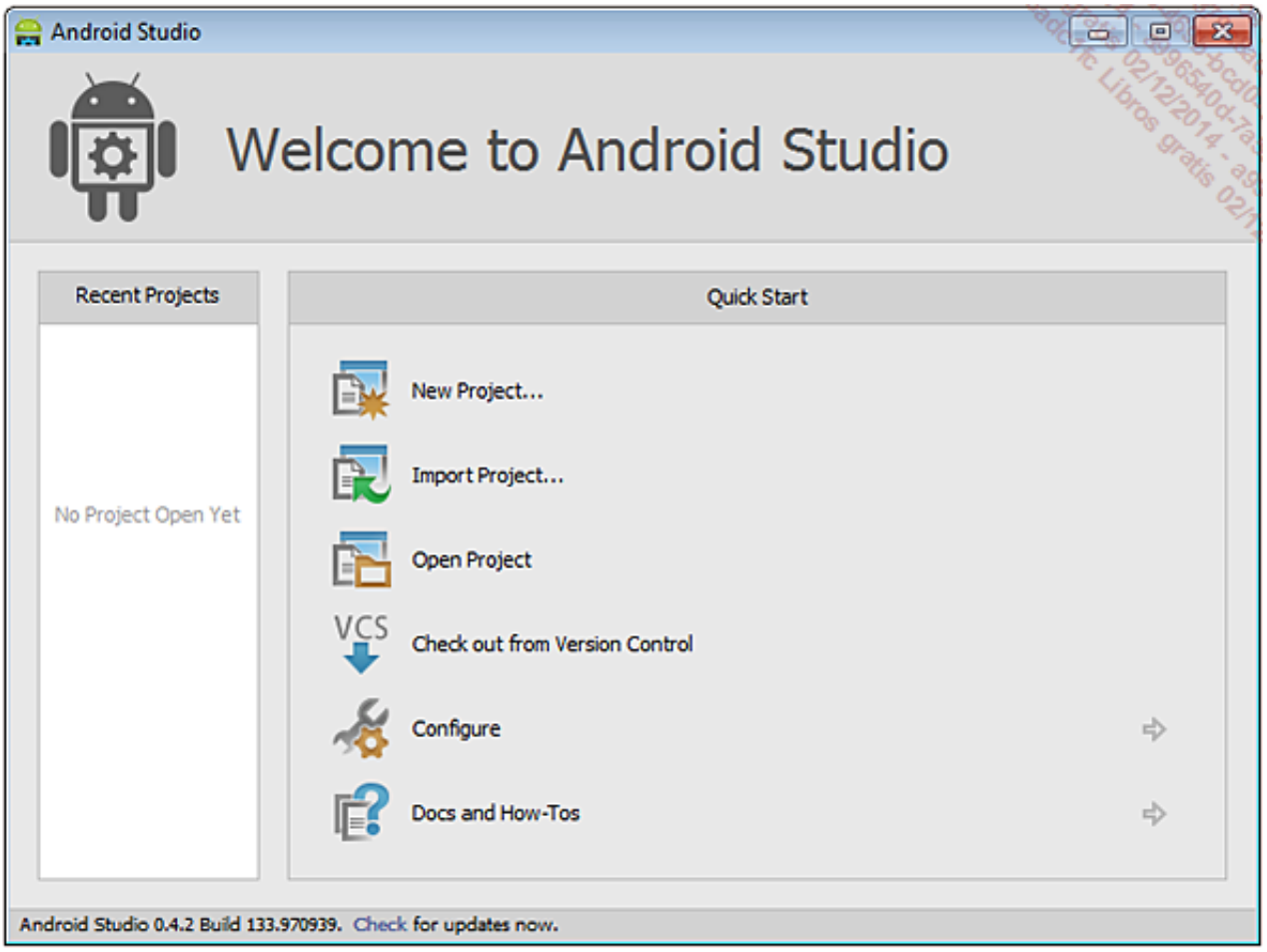
La instalación de Android Studio es muy sencilla:

- Inicie la ejecución del archivo descargado y, a continuación, sigua el asistente de instalación: seleccione la carpeta de instalación que desee y haga clic en **Install**.

La instalación termina y se abre un cuadro de diálogo que le pregunta si desea iniciar la aplicación.



- Seleccione la carpeta de destino y haga clic en **Next**.



La aplicación arranca y ofrece diversas opciones. Seleccione **New Project...**. ¡La instalación de Android Studio ha terminado!

Observe que, como ocurre con el entorno Eclipse, Android SDK Manager se instala al mismo tiempo que la aplicación. Su funcionalidad es idéntica a la versión que hemos estudiado antes en este capítulo.

Primer proyecto Android

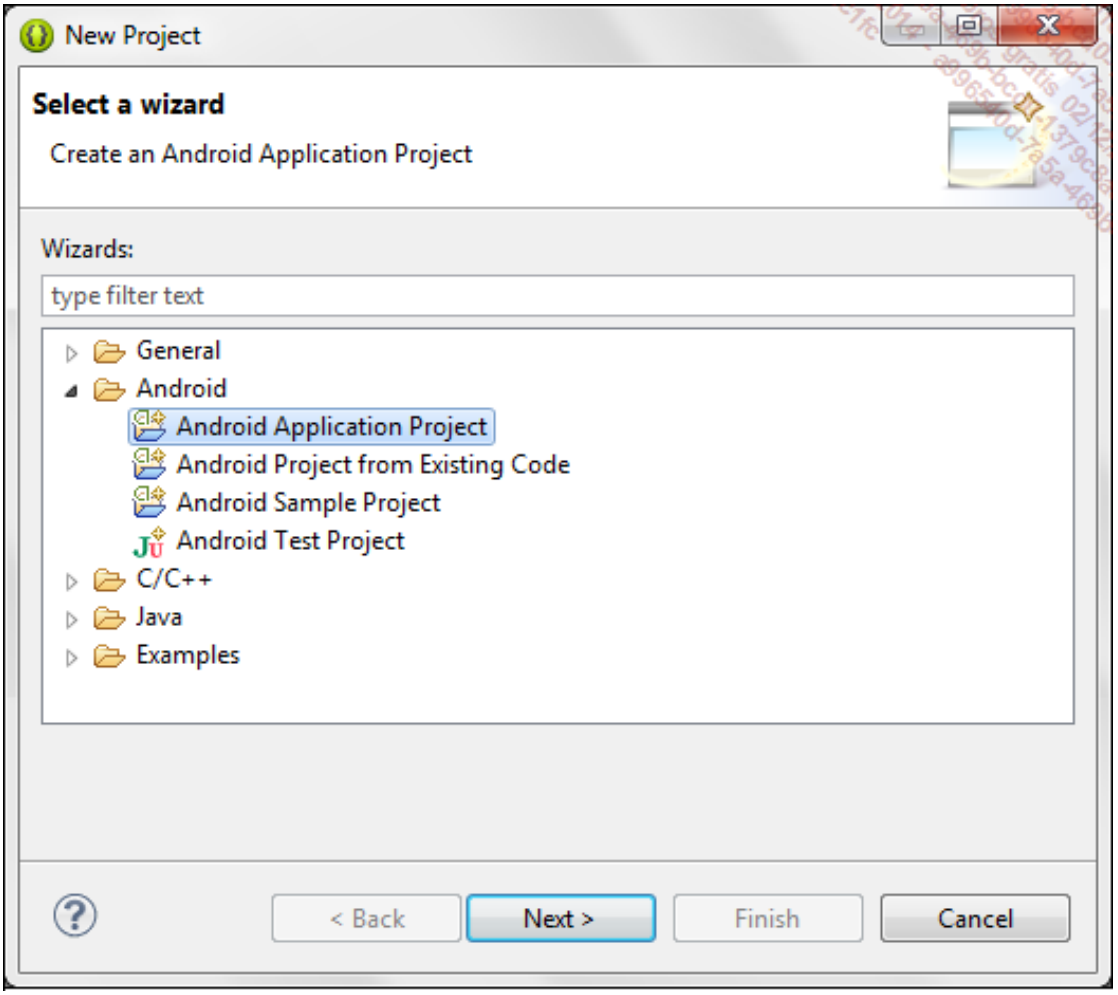
Con el entorno de desarrollo ya instalado y listo para su uso, vamos a crear sin esperar más nuestro primer proyecto Android bajo Eclipse. A continuación, lo ejecutaremos en el emulador de Android y, eventualmente, sobre un dispositivo Android.

A continuación, en la sección siguiente, descubriremos la estructura detallada de un proyecto Android.

1. Creación del proyecto

- En el menú general de Eclipse, seleccione **File - New - Project...**.

Aparece la ventana **New Project**, que muestra los distintos asistentes para la creación de proyectos.



- Despliegue la carpeta **Android**, seleccione **Android Application Project** y haga clic en el botón**Next**.

Aparece a continuación la ventana **New Android Application Project** del asistente de creación de proyectos Android.

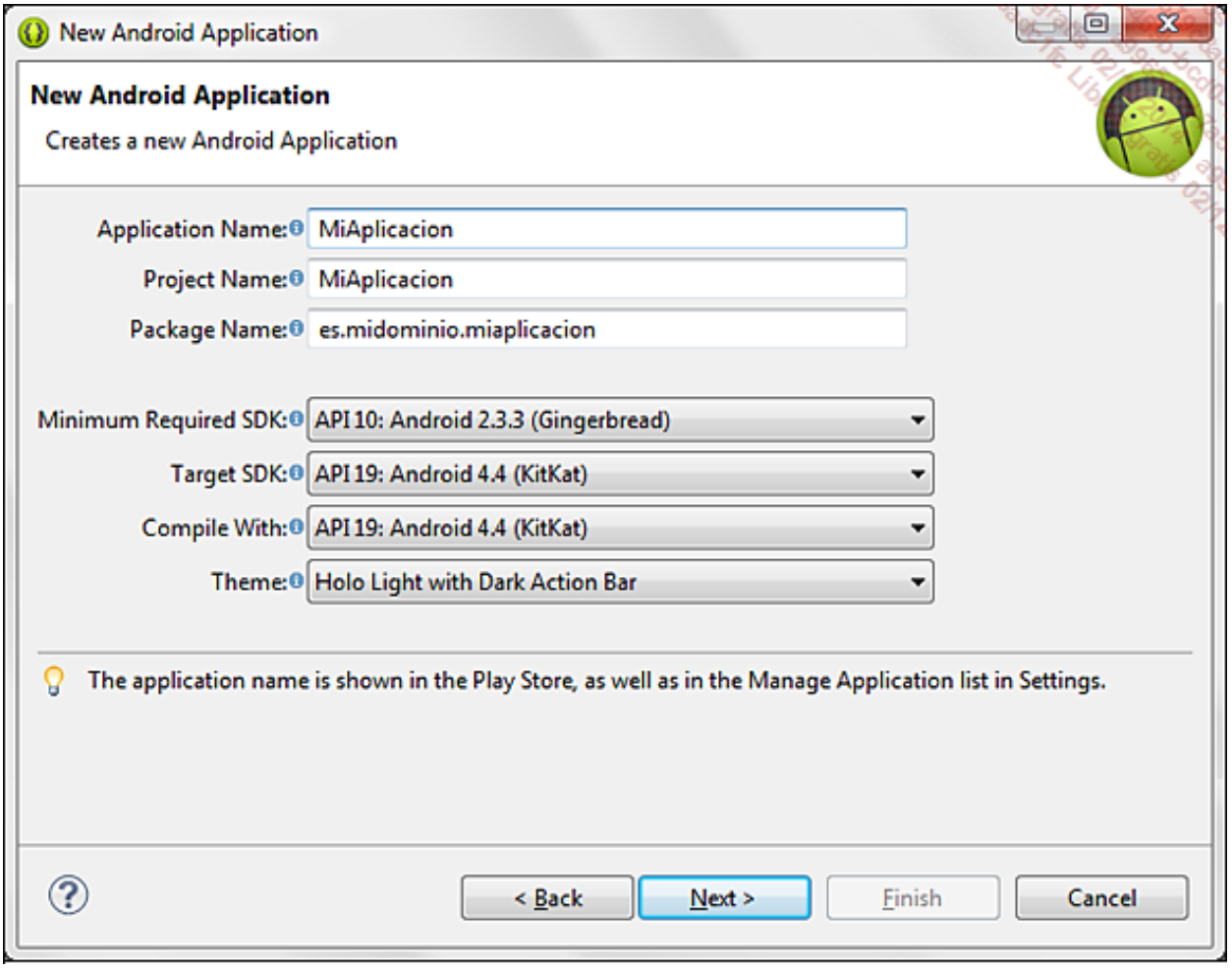
Es preciso informar numerosos campos. He aquí su correspondencia:

- **Application Name**: nombre de la aplicación, tal y como aparecerá en la Play Store y en el título que se muestra en la pantalla del dispositivo. En este primer ejemplo, seleccionamos el nombre `MiAplicacion`.
- **Project Name**: nombre del proyecto Android y de la carpeta que contiene el proyecto. Por defecto, el asistente propone el mismo nombre que el de la aplicación.
- **Package Name**: nombre del paquete Java en el que se generará el código fuente. Éste sirve

a su vez como identificador único de la aplicación para Play Store. No debe cambiar una vez publicada la aplicación en Play Store. Es preciso utilizar un formato estándar de nombre de dominio. En nuestro ejemplo, utilizaremos `es.midominio.miaplicacion`.

- **Minimum Required SDK**: versión mínima requerida para que un dispositivo pueda ejecutar la aplicación. Se recomienda seleccionar el nivel de API más bajo posible, en base a las restricciones impuestas, para poder alcanzar a una audiencia lo más amplia posible. En este primer ejemplo, vamos a seleccionar la versión 2.3.3 (API 10). La selección de una API mínima y sus consecuencias se discutirán en el capítulo Completar la interfaz de usuario.
- **Target SDK**: indica qué nivel máximo de API está comprobado para esta aplicación. Permite asegurar al sistema que no debe preocuparse de la compatibilidad de la aplicación para todos los dispositivos que dispongan de una versión del sistema inferior o igual a la indicada. En este ejemplo, seleccionaremos la versión 4.4 (API 18).
- **Compile With**: permite especificar la versión objetivo de la plataforma Android utilizada para compilar la aplicación. Se recomienda seleccionar el mismo nivel de API que para **Target SDK**.
- **Theme**: aquí indicaremos qué tema base utilizaremos para el diseño de la aplicación (color, tipografía, etc.). Seleccionaremos el tema `Holo Light with Dark Action Bar`, que parece ser el más extendido.

Estas opciones pueden modificarse una vez creado el proyecto, bien sean específicas a Eclipse o a Android.



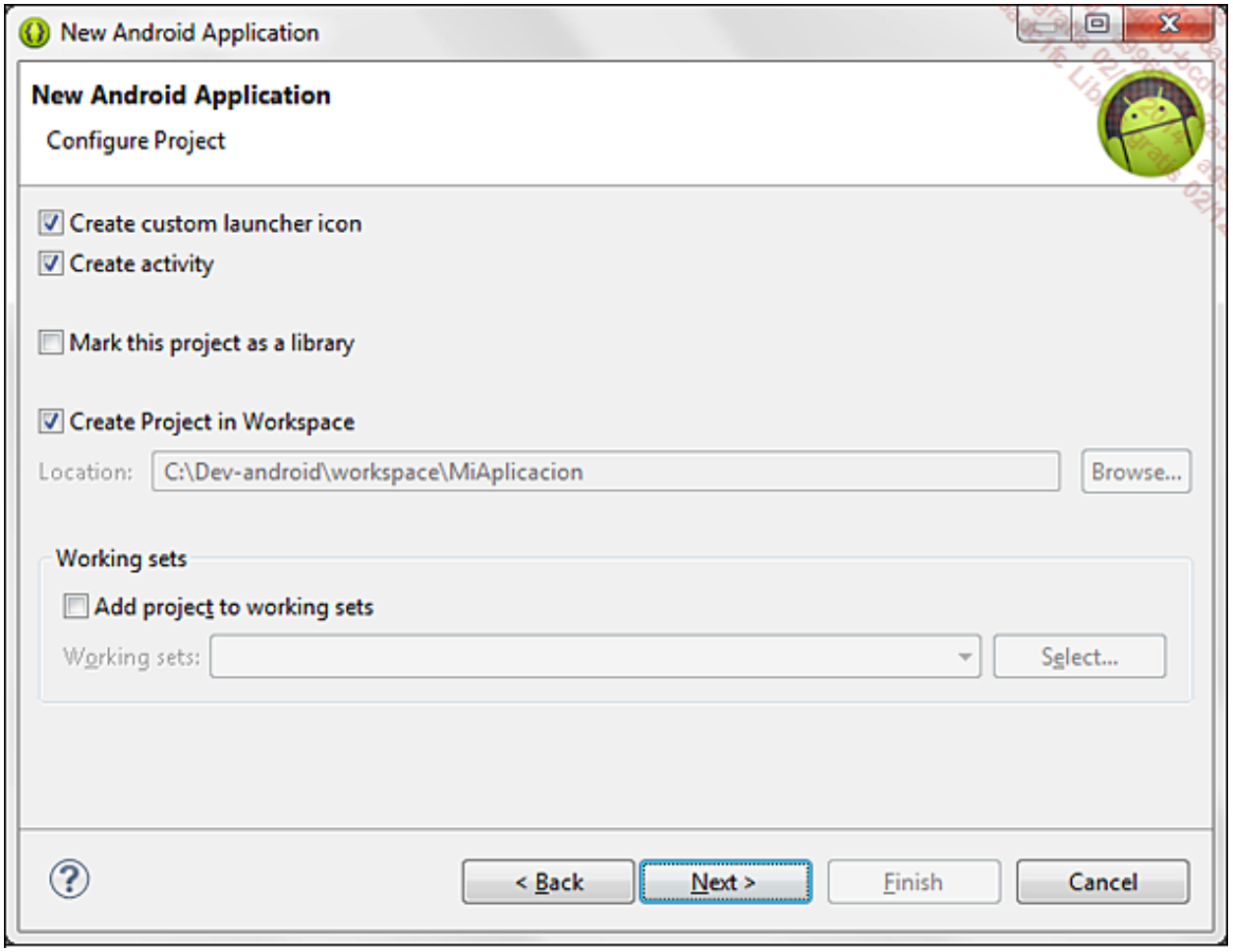
Haga clic en el botón **Next**.

El asistente nos permite, a continuación, definir opciones relativas por un lado a elementos que se integrarán automáticamente en el proyecto y, por otro, a la configuración de Eclipse para este proyecto:

- **Create custom launcher icon**: permite indicar al asistente si debe presentarnos la ventana de creación de un icono de inicio de la aplicación. En un proyecto de producción su interés es

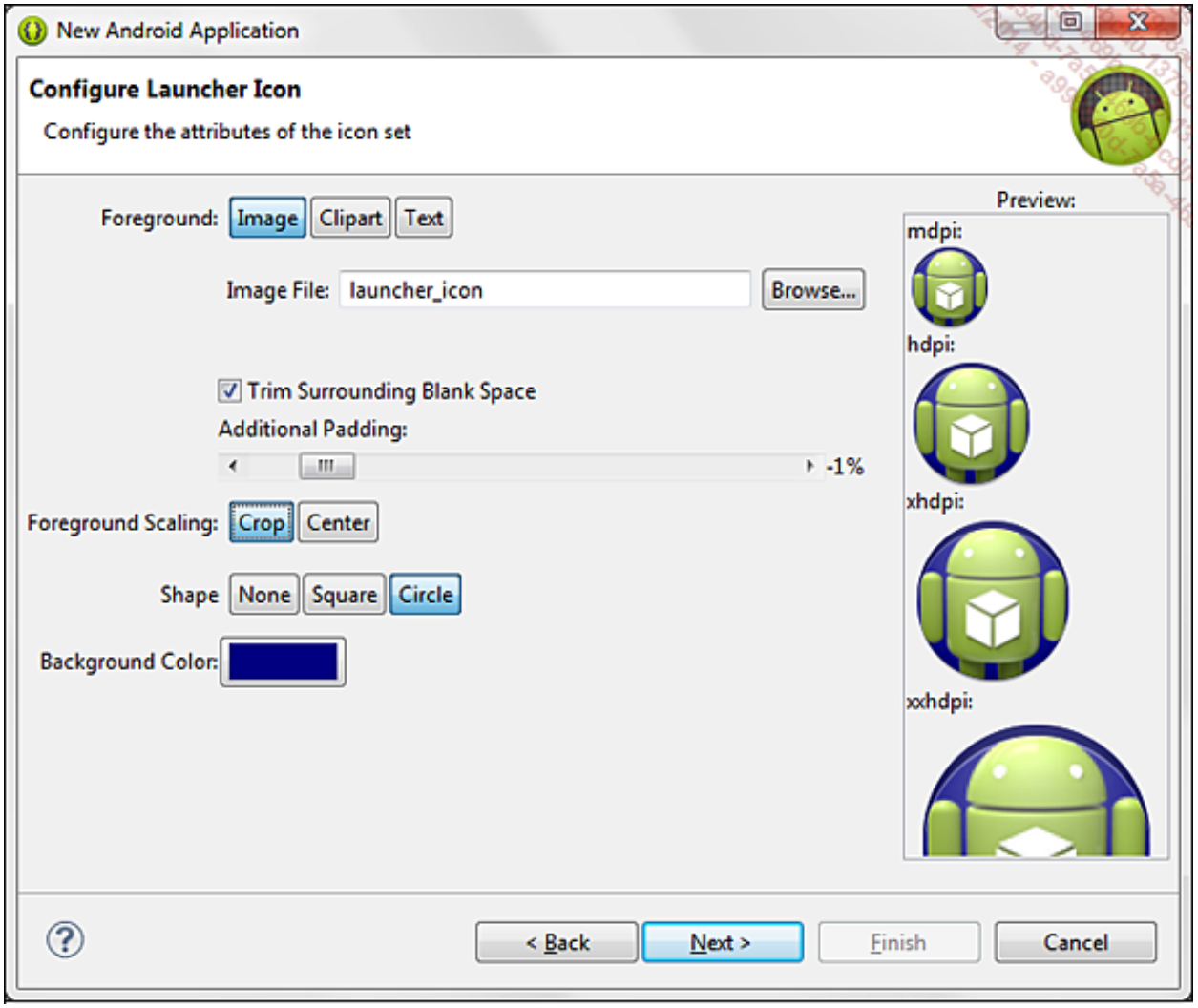
bastante limitado, aunque para un proyecto de ejemplo, supone un ahorro de tiempo apreciable. Por esta vez, marcaremos la opción.

- **Create activity**: marcaremos esta opción, que permite crear automáticamente una primera pantalla (llamada *activity* en el universo Android) para nuestra aplicación; pantalla que servirá como punto de entrada de la aplicación.
- **Mark this project as a library**: permite indicar que el proyecto no es una aplicación directamente ejecutable, sino una librería que se utilizará en otras aplicaciones. Dejamos la opción desmarcada para crear una aplicación directamente ejecutable.
- **Create Project in Workspace**: esta opción afecta a la configuración de los proyectos en Eclipse. Cada proyecto se integra en un workspace (espacio de trabajo), materializado por una carpeta del dispositivo de almacenamiento correspondiente. El proyecto se crea en una subcarpeta de dicho workspace. Dejamos la opción marcada para mantener la ubicación por defecto propuesta por Eclipse.
- **Add project to working sets**: Eclipse permite definir working sets (grupos de trabajo), que permiten agrupar varios proyectos en un mismo grupo. Esta funcionalidad resulta útil cuando se trabaja con un gran número de proyectos en el mismo workspace y se desea establecer subconjuntos de proyectos para trabajar con una mayor claridad. Dejamos esta opción desmarcada.



Haga clic en el botón **Next** para pasar al asistente de creación de icono.

Esta pantalla nos permite seleccionar una imagen, un clipart o texto como base de nuestro icono de inicio de la aplicación. Es posible redimensionar el elemento seleccionado, ajustar el color de fondo, etc. Dejaremos libertad de creación a este nivel. Observe la sección derecha de la ventana que muestra cómo se generan varias calidades de imagen (las cuales son necesarias) para adaptarse a las distintas configuraciones de pantalla.



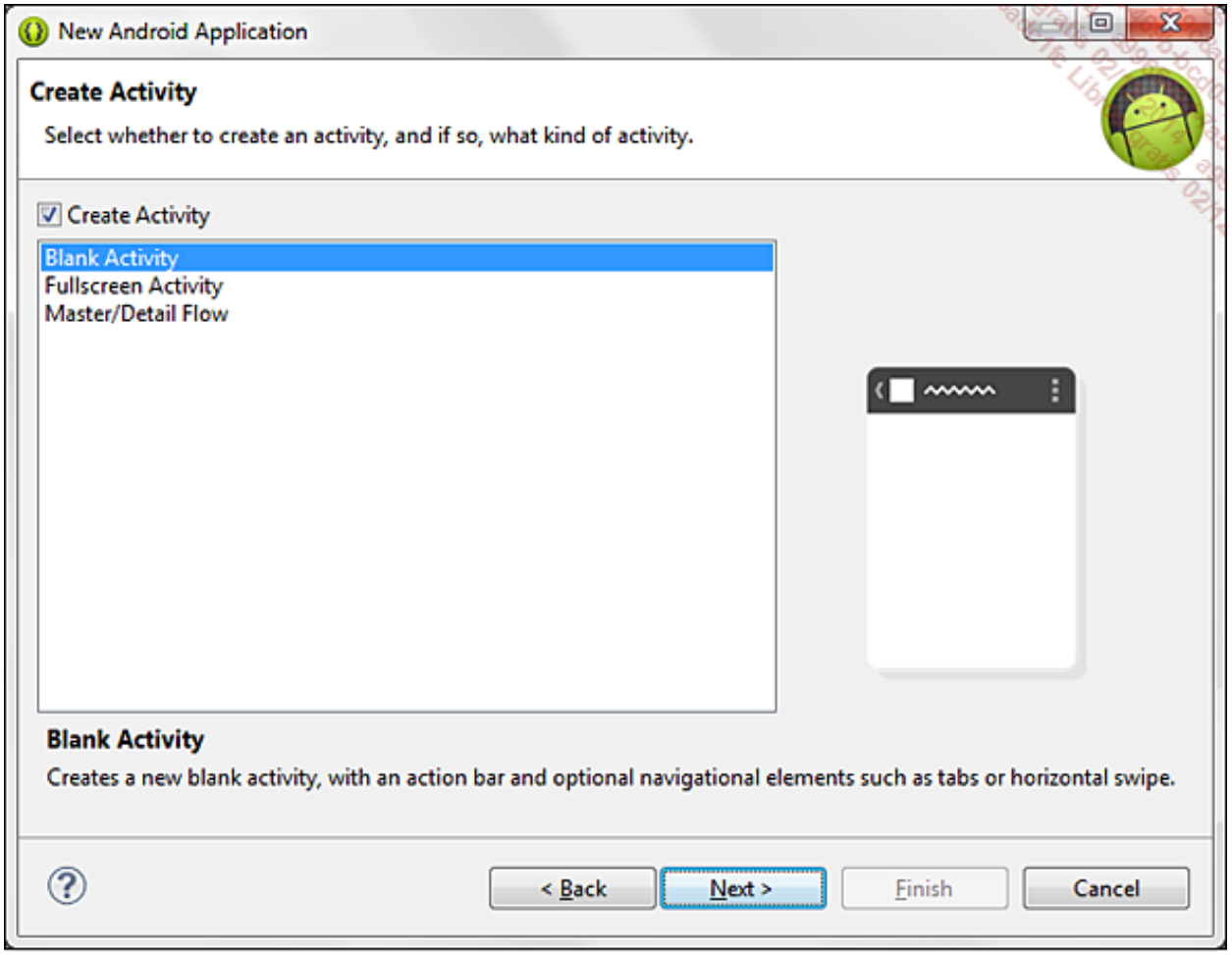
Una vez creado el icono, haga clic en el botón **Next** que permite especificar la actividad que se creará automáticamente.

El asistente le propone tres modelos de pantalla:

- **Blank Activity**: una pantalla vacía, con una barra de acción (action bar), en el formato por defecto de Android.
- **Fullscreen Activity**: una pantalla vacía, sin barra de acción, que ocupa la totalidad de la pantalla del terminal.
- **Master/Detail Flow**: una pantalla dividida en dos partes, la sección de la izquierda presentando una lista de elementos y la sección de la derecha una zona de detalle para el elemento seleccionado. El interés de dicha configuración automática consiste en integrar a la vez los mecanismos de presentación de pantallas de gran dimensión (tabletas) y pantallas de smartphones.

También puede, adicionalmente, desmarcar la opción **Create Activity** para no crear esta primera actividad.

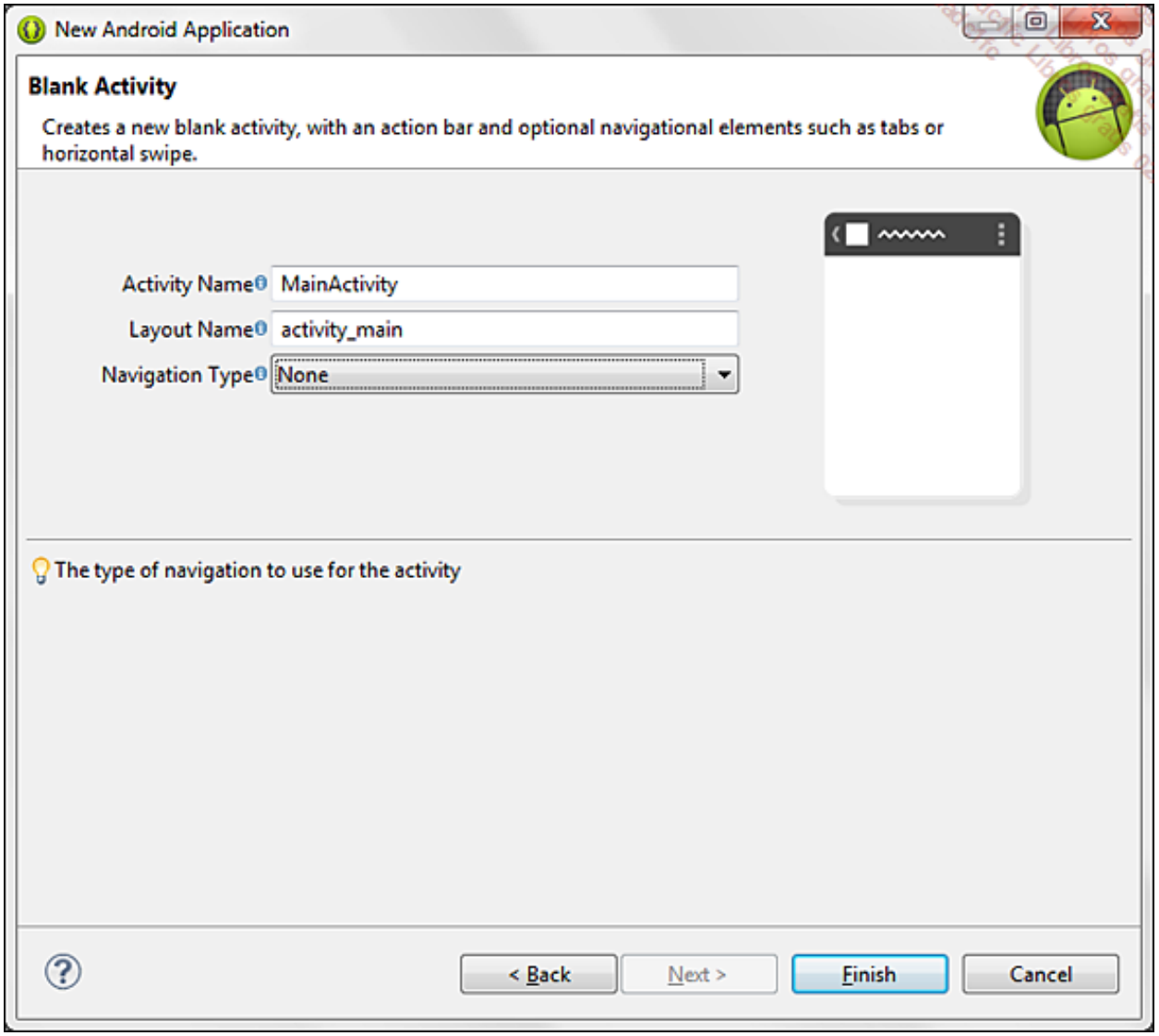
En el marco de nuestro primer ejemplo, seleccionaremos una actividad de tipo **Blank Activity**.



Haga clic en el botón **Next** para indicar el nombre de la actividad que se creará.

La pantalla nos presenta, a continuación, las opciones disponibles para la creación de nuestra activida.

- **Activity Name**: registrará, aquí, el nombre de su actividad. Una actividad es, desde el punto de vista de la programación, una clase java, de modo que aquí indicará el nombre de la clase. Dejaremos la propuesta por defecto, **MainActivity**.
- **Layout Name**: en general, la composición de una pantalla - composición de distintos elementos - se define en un archivo XML, llamado layout. Los layouts se estudian con detalle a lo largo de este libro, de modo que no diremos mucho más en este punto y dejaremos el nombre propuesto por defecto, **activity_main**.
- **Navigation Type**: permite crear automáticamente el mecanismo de navegación que se implementará para la actividad. La lista desplegable permite, a continuación, seleccionar entre varias opciones: **None** (sin navegación prevista), **Fixed Tabs + Swipe** (sistema de pestañas con navegación por movimientos de barrido), **Scrollable Tabs + Swipe** (pestañas deslizantes con navegación por movimientos de barrido) o **Dropdown** (lista desplegable en la barra de acción, que hace las veces de menú). Seleccionaremos **None**, nuestra aplicación es una simple pantalla como primer ejercicio.

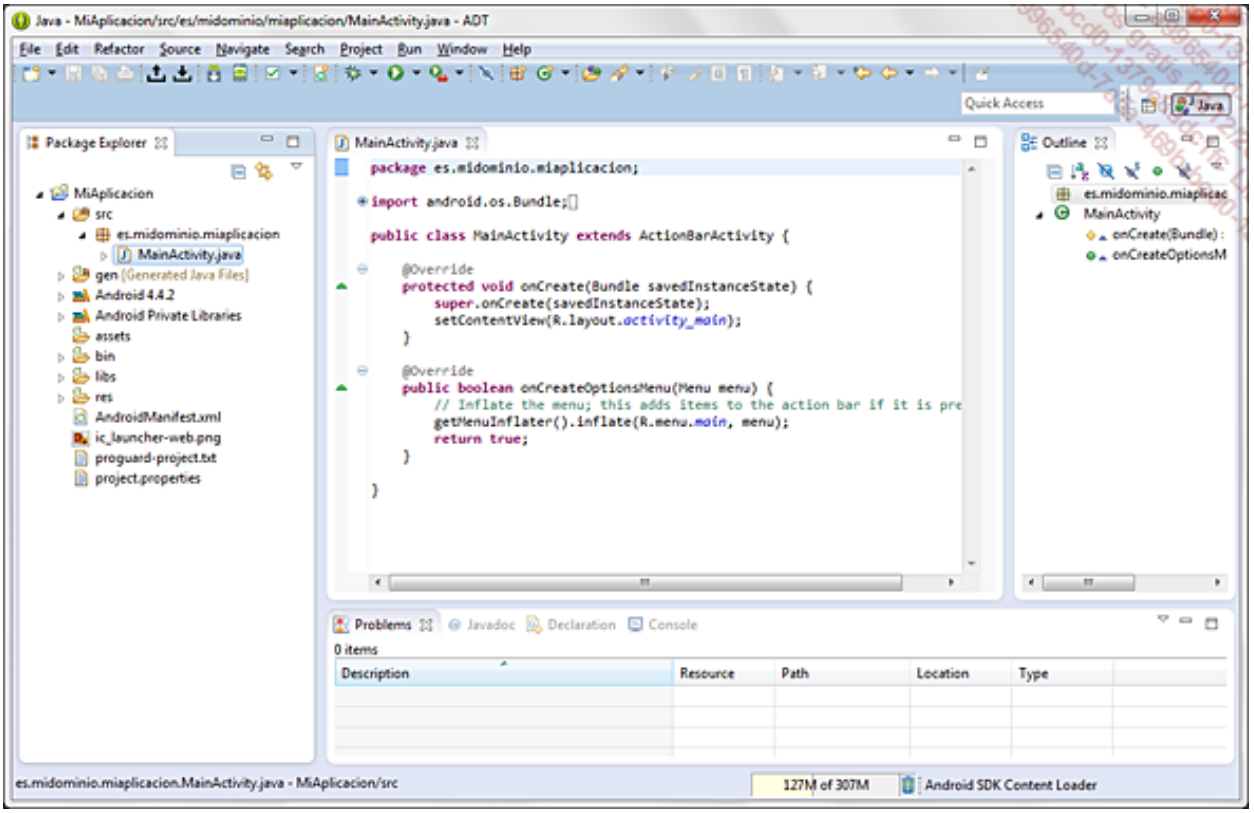


- Haga clic en el botón **Finish** para iniciar la creación del proyecto.

Se nos propone crear un proyecto de prueba. No haremos nada de momento pues este tema se abordará en otro capítulo del libro (véase el capítulo Trazas, depuración y pruebas - Pruebas unitarias y funcionales).

- Haga clic en el botón **Finish**.

Se genera el proyecto en el espacio de trabajo de Eclipse y se agrega a la vista **Package Explorer**(Explorador de paquetes). Se compila automáticamente.



La compilación del proyecto conlleva la creación de un archivo binario `MiAplicacion.apk` en la carpeta `bin` del proyecto.

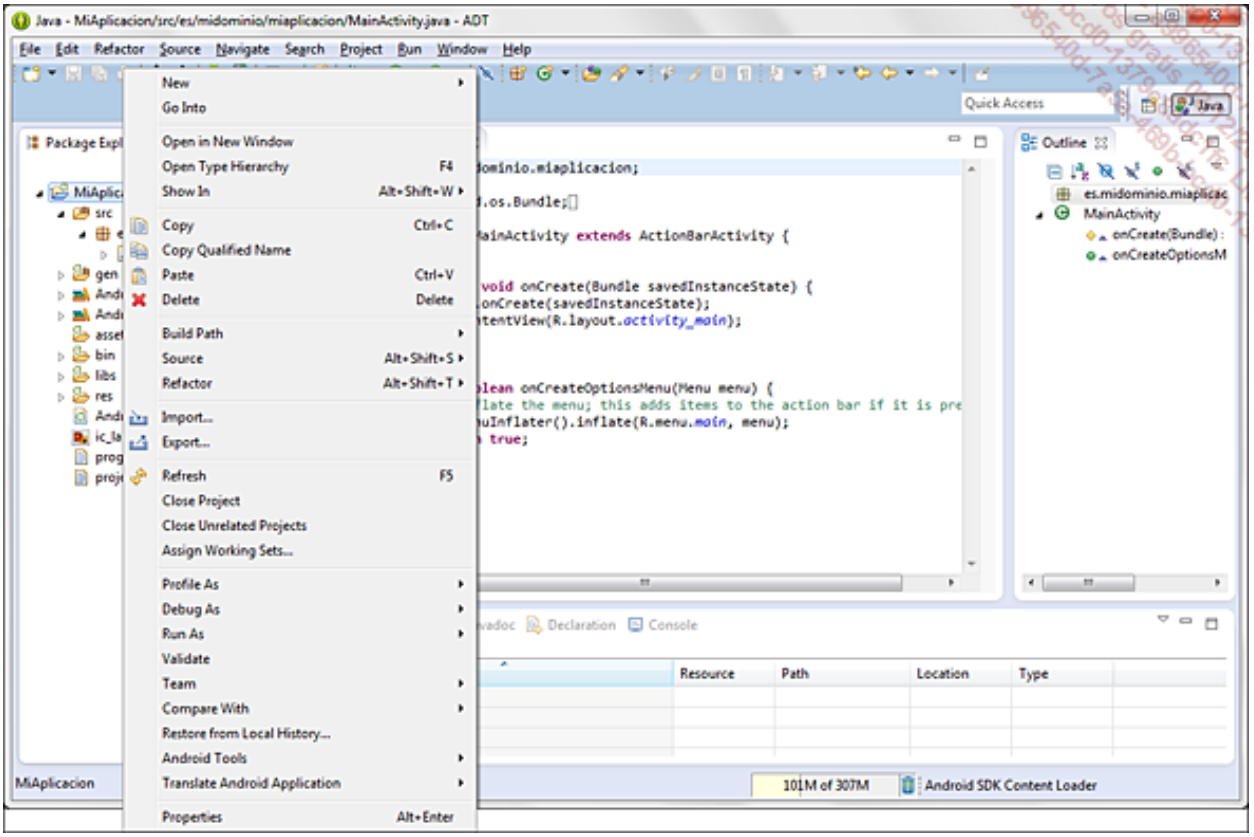
Se trata de un archivo de formato apk (*Android Package*), es decir, un archivo zip que contiene todo el proyecto: código compilado, datos y archivos de recursos.

Debido a que este archivo contiene la aplicación, es habitual referirse a este archivo usando directamente el término aplicación.

2. Ejecución de la aplicación

Ahora que el proyecto ha sido creado con éxito, y que la aplicación ha sido generada, sólo queda instalarla en un sistema Android y ejecutarla. Para ello:

- En la vista **Package Explorer** de Eclipse, haga clic con el botón derecho sobre el proyecto`MiAplicacion`.



A continuación haga clic en **Run As - Android Application**.

La vista Consola permite seguir el avance de carga de la aplicación y de su ejecución.

Si aparece en la consola un mensaje de error del tipo `The connection to adb is down, and a severe error has occured`, reinicie Eclipse e inténtelo de nuevo.

Aparece una ventana de diálogo **Android AVD Error** informándonos de que no existe ningún AVD(*Android Virtual Device*), es decir una configuración del emulador Android o un dispositivo Android conectado que posea una versión igual o superior a la versión mínima indicada durante la creación del proyecto. Se nos propone crear la configuración de un dispositivo a emular.

Haga clic en el botón **No** y cierre la ventana **Android Device Chooser**. Volveremos a ella más tarde.

Existen dos posibilidades a la hora de ejecutar la aplicación:

- Crear un AVD, un periférico Android virtual, y ejecutarlo sobre el emulador.
- Utilizar un dispositivo Android real, como un smartphone o una tableta táctil, conectado al ordenador.

a. En el emulador Android

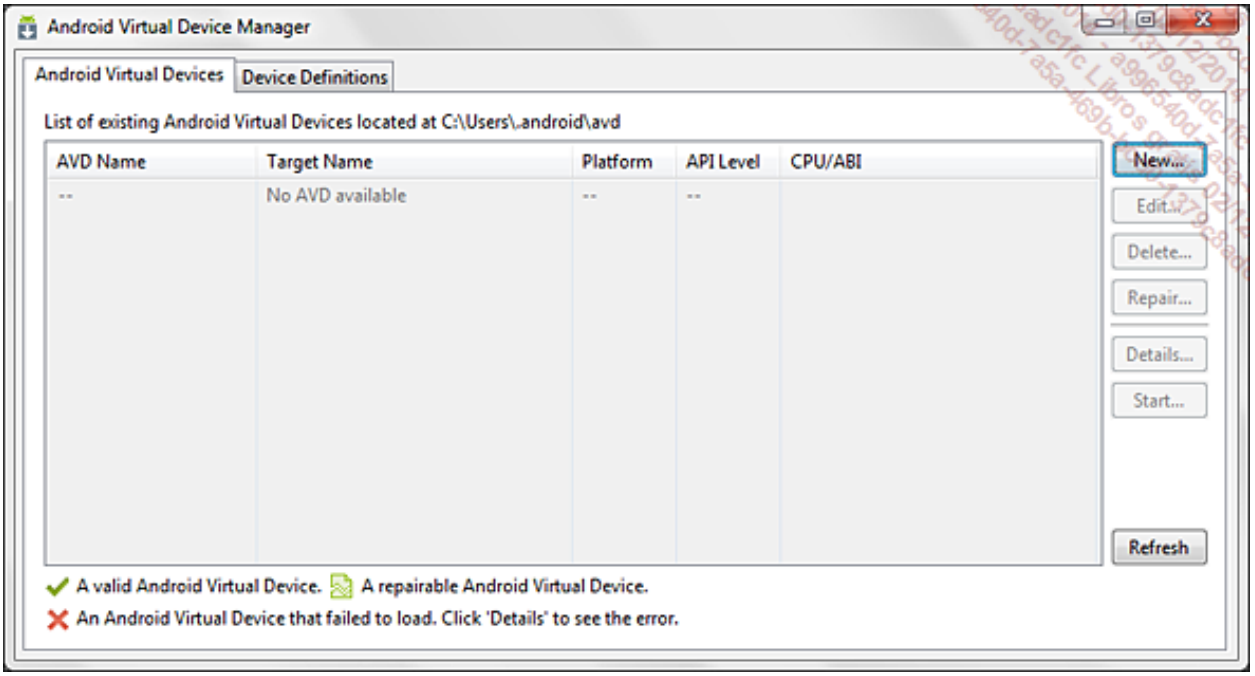
El SDK proporciona un emulador de periféricos Android. Este emulador permite emular numerosas configuraciones de hardware. Un AVD es una configuración de hardware concreta que utilizará el emulador.

Veamos cómo se crea un AVD.

Ejecute la herramienta **Android Virtual and Device Manager (AVD Manager)**.

Recuerde, la herramienta **AVD Manager** puede ejecutarse desde la barra de herramientas haciendo clic en su icono o seleccionado **Window - Android Virtual Device Manager**.

A continuación se abre la herramienta **Android Virtual Device Manager** indicando que no existe ningún AVD.



Haga clic en el botón **New...** situado arriba a la derecha.

Aparece la ventana **Create new Android Virtual Device (AVD)**.

Para crear el AVD es preciso informar numerosos campos:

- **AVD Name**: nombre del AVD. No debe contener caracteres especiales, ni espacios.

Este nombre debe, a su vez, ser corto de cara a tener una mayor comodidad dado que se utiliza por línea de comandos o en una consola, y ser lo suficientemente explícito para caracterizar la configuración del AVD.

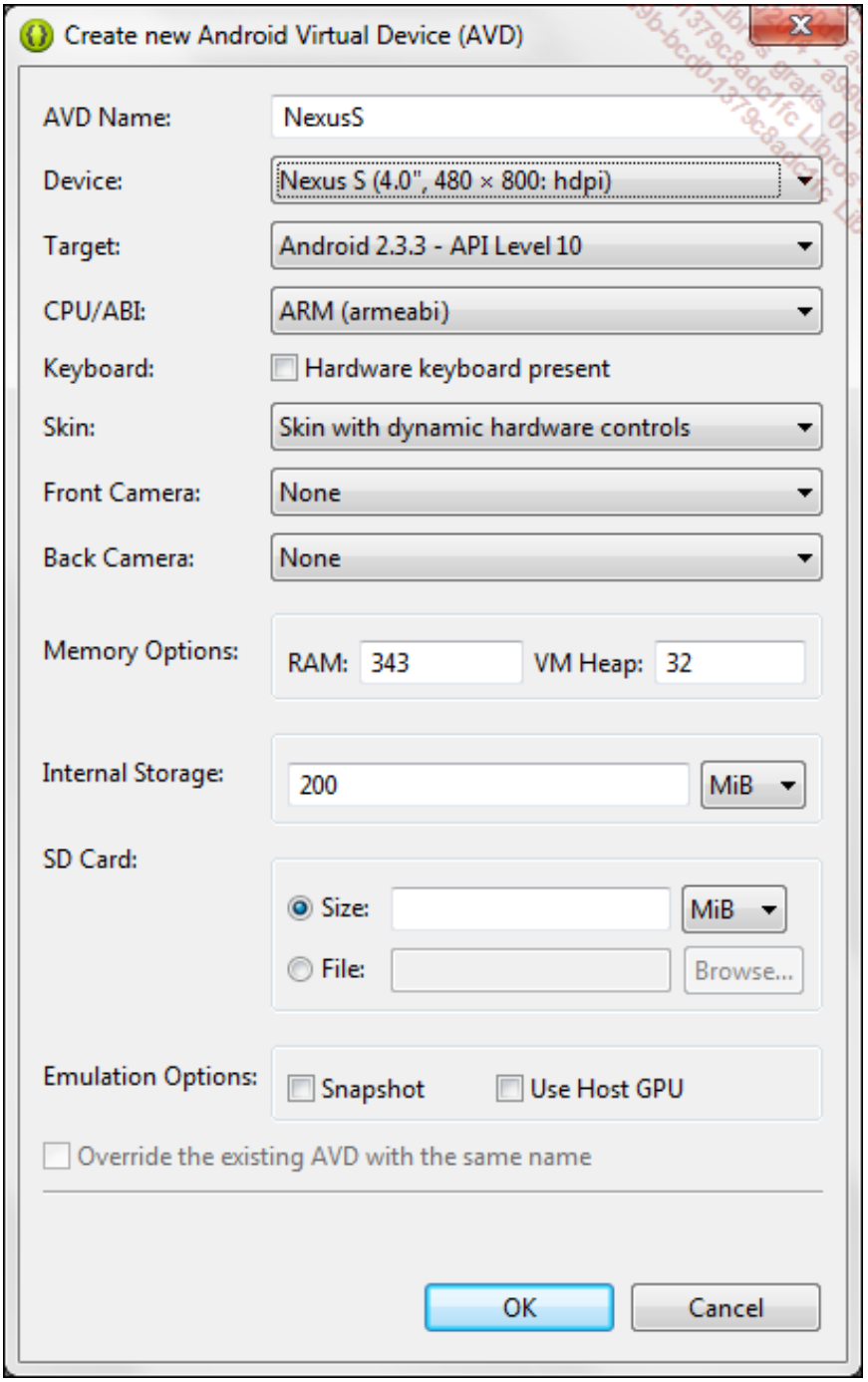
- **Device**: lista de dispositivos Android de configuración estándar. La lista de dispositivos Nexus está presente, así como un conjunto de configuraciones estándar de pantalla, desde QVGA (240 x 320, baja definición) hasta WXGA (1280 x 720, ultra alta definición) y más. La selección de un dispositivo Nexus permite partir de opciones que se corresponden con las especificaciones del dispositivo seleccionado.
- **Target**: lista que proporciona las plataformas disponibles descargadas e instaladas sobre el puesto de desarrollo. Recuerde, la versión de la API de la plataforma escogida deberá ser igual o superior a la versión mínima de la API especificada en la aplicación.
- **CPU/ABI**: lista que permite especificar qué tipo de procesador se emulará. Aunque, en teoría, el desarrollador debería probar su aplicación en todas las configuraciones posibles, en el marco de una aplicación clásica el hecho de seleccionar ARM se admite, comúnmente, como lo más generalista, abarcando el mayor número de dispositivos Android.
- **Keyboard**: permite especificar si el dispositivo simulado dispone de un teclado físico o no. La gran mayoría de dispositivos Android no disponen de teclado, de modo que es preferible desmarcar esta opción.
- **Skin**: seleccionar esta opción indica al emulador que se desea presentar, además de la pantalla del dispositivo, los botones físicos habituales que se encuentran en un terminal Android: botones de volumen, botón Home (inicio), botón de menú y botón back (volver). Es preferible marcar esta opción, ¡pues de lo contrario será preciso utilizar atajos de teclado!
- **Front Camera**: permite especificar si el dispositivo emulado dispone de una cámara frontal

(sobre la cara principal del dispositivo). Puede emular una cámara o bien utilizar la webcam de su equipo. La opción None permite obviar este aspecto de la emulación, en el caso de que la aplicación desarrollada no interactúe con la cámara del dispositivo.

- **Back Camera**: permite especificar si el dispositivo emulado dispone de una cámara sobre la cara trasera. De manera típica, los dispositivos Android disponen de una cámara sobre la cara trasera con una mejor resolución que la disponible en la cara principal.
- **Memory Options**: permite indicar la memoria disponible sobre el dispositivo. Los valores por defecto son los recomendados en la mayoría de casos.
- **Internal Storage**: permite especificar el espacio disponible para la aplicación. Sabiendo que una aplicación Android típica posee un tamaño muy inferior a 50 MB, el valor por defecto propuesto de 200 MB es más que suficiente.
- **SD Card**: permite emular la presencia de una tarjeta SD en el dispositivo virtual especificando un tamaño. Preste atención de no indicar un tamaño excesivamente grande dado que se creará un archivo del tamaño de esta tarjeta SD en el puesto de desarrollo. También es posible reutilizar una tarjeta SD virtual creada anteriormente indicando el nombre de esta tarjeta SD.
- **Snapshot**: cuando se selecciona esta opción, el emulador realiza una copia de seguridad de la RAM tras el arranque para, a continuación, restaurarla en el siguiente inicio, acelerando de este modo el arranque de la AVD.
- **Use Host GPU**: permite especificar que, si una aplicación utiliza comandos OpenGL, utilizará la GPU del ordenador host en lugar de emular una GPU. Esta opción es incompatible con la opción **Snapshot**.

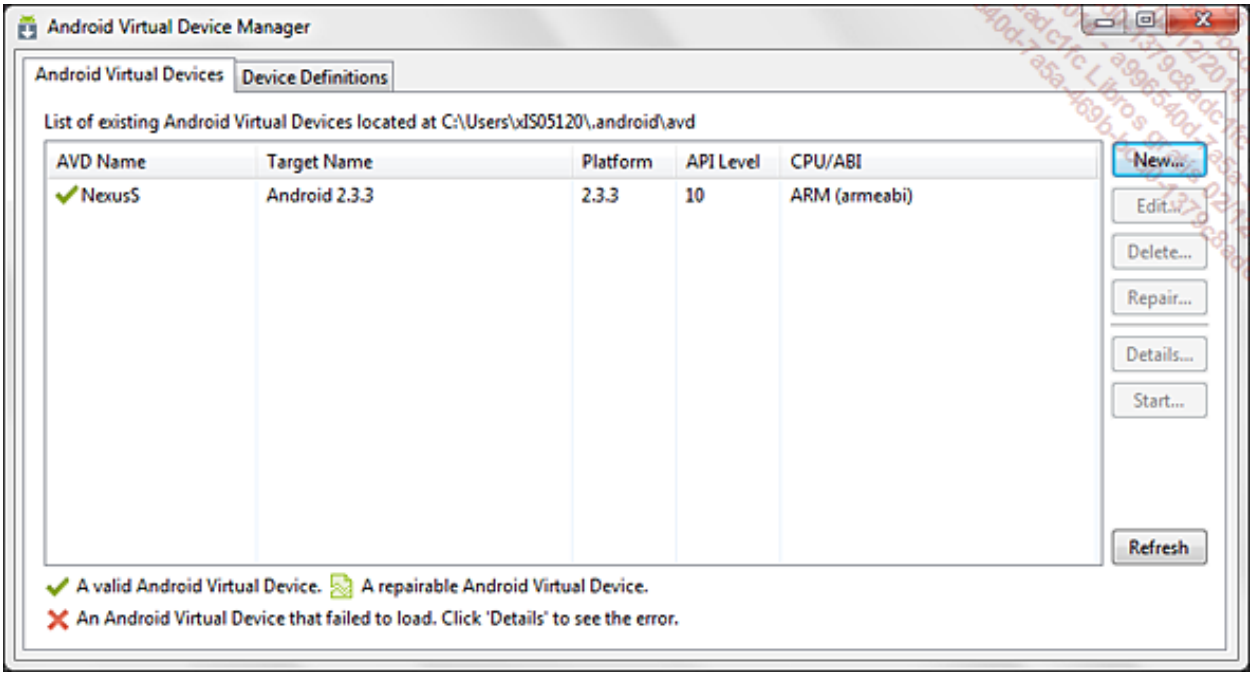
Para nuestro ejemplo, y como primer dispositivo Android, vamos a emular un smartphone Nexus S, que dispone de una pantalla de 480 x 800 en hdpi, Android 2.3.3 y 343 MB de RAM.

Tras la selección del modelo y una vez introducido un nombre para el terminal que se creará, haga clic en el botón **OK**.



A continuación el AVD se agrega a la lista de AVD existentes.

Es posible crear tantos AVD como se quiera. Esto permite probar la misma aplicación sobre numerosas configuraciones de hardware y asegurarse así del buen funcionamiento de ésta sobre el mayor número de configuraciones diferentes.



Los botones **Delete...**, **Repair...** y **Details...** permiten respectivamente suprimir, reparar el AVD seleccionado actualmente y mostrar sus características.

El botón **Start** permite iniciar el emulador con el AVD seleccionado especificando los parámetros de inicio. Es posible modificar el tamaño de la ventana del emulador y borrar los datos de usuario del AVD como si el dispositivo saliera de fábrica. También es posible indicar que se quiere restaurar el AVD tras una copia de seguridad marcando la opción **Launch from snapshot**. La opción **Save to snapshot** permite realizar una copia de seguridad del AVD tras cada cierre de la ventana del emulador.

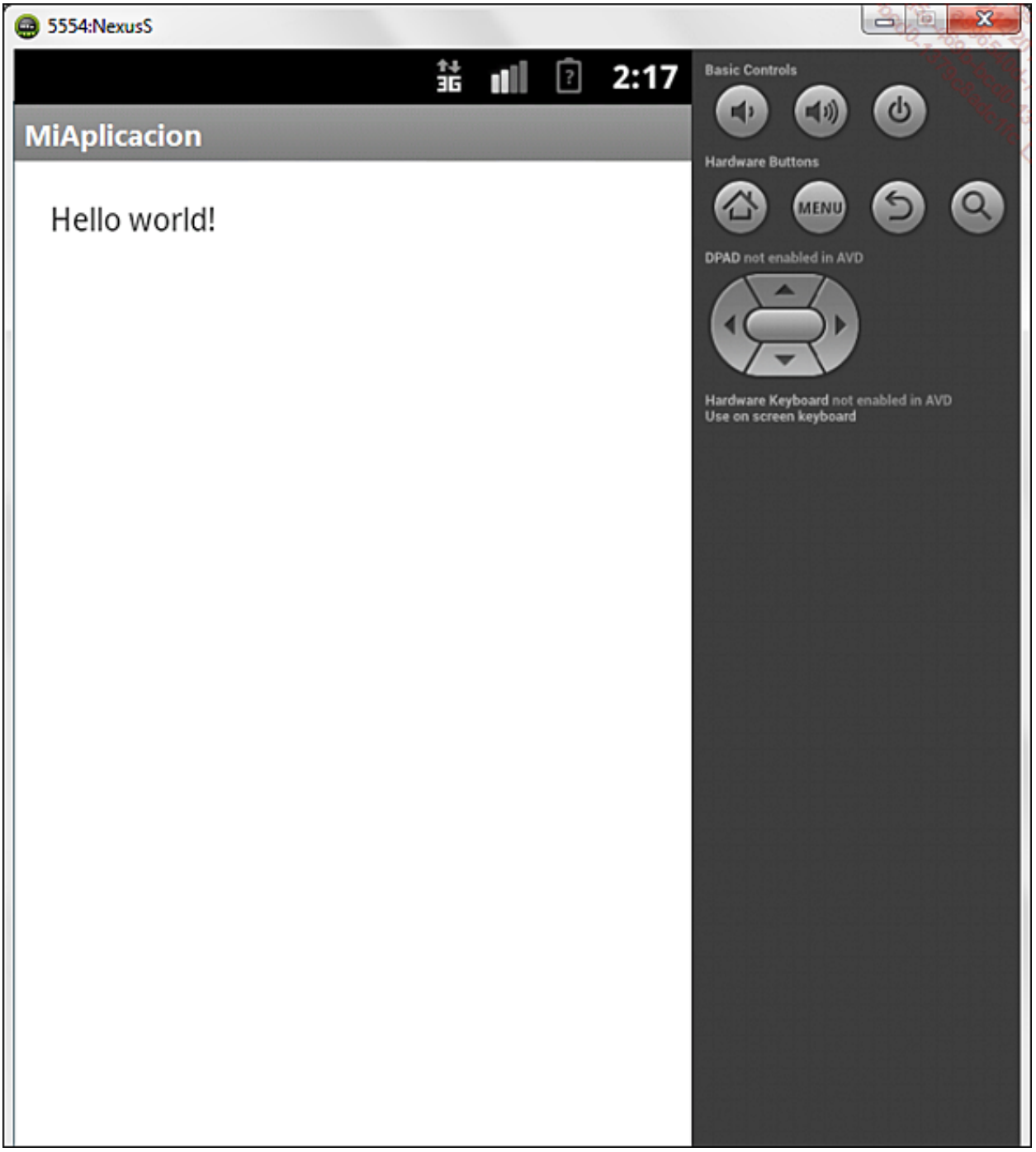
- Cierre la ventana.
- Vuelva a ejecutar la aplicación tal y como hemos visto anteriormente.

El emulador Android se ejecuta a continuación en el puesto de desarrollo. Espere el tiempo de carga del sistema. Esto puede llevar varios minutos según la potencia del puesto de desarrollo. A continuación aparece la pantalla de bienvenida o la pantalla de desbloqueo. A continuación se carga y se ejecuta la aplicación automáticamente.

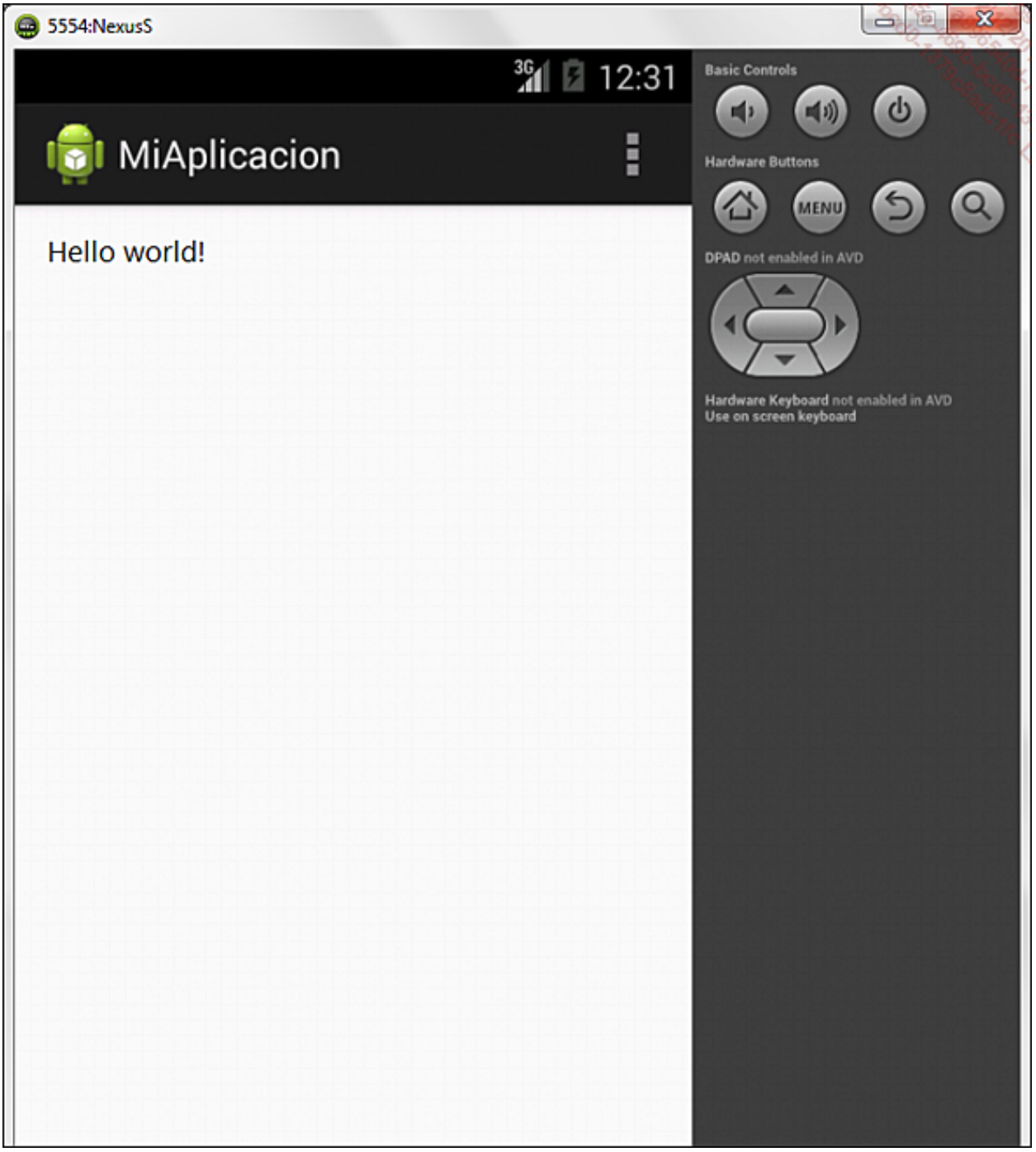
Si aparece la pantalla de desbloqueo, desbloquee para poder ver la aplicación.

La interfaz del emulador y la interfaz de la aplicación son distintas según la versión concreta de Android utilizada por el proyecto, la versión del AVD sobre la que se ejecuta la aplicación, así como las opciones especificadas durante la creación de la actividad.

Con el fin de tener una primera impresión del trabajo de desarrollo de aplicaciones en Android, las dos siguientes capturas de pantalla muestran el aspecto de la aplicación sobre dos dispositivos (emulados) diferentes: incluso para una aplicación tan simple como un proyecto por defecto, las diferencias son visibles.



He aquí la representación de la aplicación sobre el terminal en que acabamos de crear una emulación y, más abajo, el mismo terminal, pero equipado con la versión 4.4. de Android.



Existen numerosos atajos de teclado que permiten interactuar con el emulador. Por ejemplo, la tecla 7 del teclado numérico del ordenador permite cambiar la orientación de la pantalla en un sentido, y la tecla 9 en el otro sentido. Las distintas opciones de ejecución del emulador están disponibles en la siguiente dirección:http://developer.android.com/guide/developing/tools/emulator.html

b. En un dispositivo Android

Si bien el emulador puede adaptarse a numerosas configuraciones de hardware, también es cierto que hay algunas que no puede emular de manera cómoda. Uno de los mayores defectos del emulador es su rendimiento, más bien mediocre. De este modo, el emulador mostrará rápidamente su debilidad si la aplicación que quiere probar requiere un dispositivo potente. Además, probando una aplicación a través del emulador ésta puede parecer correcta mientras que una prueba real va a mostrar en seguida los defectos de la aplicación.

Un ejemplo muy sencillo es la presión táctil sobre la pantalla. En el emulador, basta con hacer clic

con el cursor del ratón, que es muy preciso a nivel de píxel. En un dispositivo real, el dedo del usuario es menos preciso que el cursor del ratón, hasta el punto de que si la aplicación requiere hacer clic sobre una zona pequeña de la pantalla, será mucho más delicado si se hace con los dedos, mientras que no supondrá problema alguno utilizando el ratón en un emulador.

Otro aspecto que genera problemas es la rotación de la pantalla: si bien en un emulador el cambio en la orientación se controla completamente, sobre un dispositivo físico la rotación de la pantalla está vinculada directamente a la manipulación del usuario. El desarrollador, pense a todas las pruebas que pueda hacer, no prevé la respuesta tras una rotación de pantalla que sobreviene en un momento imprevisto. El uso de un dispositivo real permite ajustar cierto realismo a las pruebas, ¡lo cual resulta saludable y ayuda a mejorar la calidad de la aplicación publicada!

Es por ello que es casi obligatorio ejecutar la aplicación en un dispositivo Android real para validar su buen funcionamiento, su reactividad y, sobretodo, su ergonomía. No obstante, dado que esto sólo validará la aplicación en un único tipo de hardware, lo ideal es combinar pruebas en un emulador y pruebas sobre dispositivos reales.

Para probar una aplicación sobre un dispositivo Android, es preciso activar la depuración USB sobre el dispositivo. Para ello, la primera etapa de la manipulación es distinta según el sistema Android utilizado, o según el fabricante - la mayoría de fabricantes y/o distribuidores tienen la (incómoda) costumbre de agregar una capa propietaria por encima de Android.

- En sistemas Android de versiones inferiores a la 3.0 (API 11), vaya a la pantalla de bienvenida si no se muestra directamente y presione a continuación la tecla **Menú**. En sistemas Android de versiones 3.0 (API 11) o superiores, haga clic sobre la hora en la parte inferior derecha y a continuación haga clic en la pequeña ventana que contiene la hora que acaba de aparecer.
- Seleccione **Configuración**.

A continuación se muestra la aplicación de configuración de los parámetros del dispositivo.

- Seleccione **Aplicaciones**, a continuación **Desarrollo**.
- Active la opción **Depuración USB** y valide el mensaje de advertencia haciendo clic en el botón**OK**.

A continuación hay que hacer que el puesto de desarrollo reconozca al dispositivo Android cuando se conecte mediante un cable USB. Siga las instrucciones que aparecen a continuación según el sistema operativo instalado sobre el puesto de desarrollo.

Si apareciesen problemas durante la instalación de los controladores o si el dispositivo no se detectase o reconociese, verifique que el cable USB que está utilizando es el entregado originalmente con el dispositivo Android y que está conectado directamente al puesto de desarrollo.

Mac OS X

- Conecte el dispositivo Android al puesto de desarrollo mediante un cable USB.

El dispositivo Android se detecta automáticamente.

Windows XP

Es preciso instalar un controlador USB sobre el puesto de desarrollo para reconocer dispositivos Android.

- Descárguelo mediante la herramienta Android SDK Manager vista anteriormente. El paquete se llama **Google USB Driver package** y se encuentra en **Third party Add-ons** y a continuación**Google Inc**.
- Conecte el dispositivo Android al puesto de desarrollo utilizando un cable USB. Windows detecta automáticamente el dispositivo y ejecuta el asistente para agregar hardware.

- Seleccione **No, no esta vez** y haga clic en **Siguiente**.
- Seleccione **Instalar a partir de una lista o de una ubicación específica**, y haga clic en**Siguiente**.
- Seleccione **Buscar el mejor controlador en esta ubicación**.
- Desmarque la opción **Buscar en medios extraíbles** y marque **Incluir esta ubicación en la búsqueda**.
- Haga clic en el botón **Examinar**, seleccione la carpeta`sdk\extras\google\usb_driver\` y haga clic en **OK** y a continuación en **Siguiente**.
- Por último, haga clic en el botón **Instalar**.

Se inicia la instalación del controlador.

Windows Vista

Es preciso instalar un controlador USB sobre el puesto de desarrollo para reconocer dispositivos Android.

- Descárguelo mediante la herramienta Android SDK Manager vista anteriormente. El paquete se llama **Google USB Driver package** y se encuentra en **Third party Add-ons** y a continuación**Google Inc**.
- Conecte el dispositivo Android al puesto de desarrollo utilizando un cable USB. Windows detecta automáticamente el dispositivo e intenta instalarlo.
- Abra el menú **Inicio** y seleccione **Panel de control**.
- Seleccione **Hardware y sonido** y a continuación **Administrador de dispositivos**.
- Haga clic con el botón derecho sobre la línea **Periférico USB compuesto** que muestra un pequeño icono de alerta en la lista y seleccione a continuación **Actualizar software del controlador...**.
- Seleccione **Buscar software del controlador en el equipo**.
- Haga clic en el botón **Examinar...**, seleccione la carpeta`sdk\extras\google\usb_driver\` y haga clic en **Siguiente**.
- Por último, haga clic en el botón **Instalar**.

Se inicia la instalación del controlador.

Windows 7

Es preciso instalar un controlador USB sobre el puesto de desarrollo para reconocer dispositivos Android.

- Descárguelo mediante la herramienta Android SDK and AVD Manager vista anteriormente. El paquete se llama **Google USB Driver package** y se encuentra en **Third party Add-ons** y a continuación **Google Inc**.
- Conecte el dispositivo Android al puesto de desarrollo utilizando un cable USB. Windows detecta automáticamente el dispositivo e intenta instalarlo.
- Abra el menú **Inicio** y seleccione **Panel de control**.
- Seleccione **Hardware y sonido** y a continuación **Administrador de dispositivos** (bajo**Dispositivos e impresoras**).
- Haga clic con el botón derecho sobre la línea **Periférico USB compuesto** que muestra un pequeño icono de alerta en la lista y seleccione a continuación **Actualizar software del controlador...**.

- Seleccione **Buscar software del controlador en el equipo**.
- Haga clic en el botón **Examinar...**, seleccione la carpeta`sdk\extras\google\usb_driver\` y haga clic en **Siguiente**.
- Por último, haga clic en el botón **Instalar**.

Se efectuará la instalación del controlador.

Ubuntu

- Desconecte el dispositivo Android del puesto de desarrollo si está conectado.
- Abra una consola y ejecute el comando `lsusb`.

```
$ lsusb
Bus 001 Device 002: ID 80ee:0021
Bus 001 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
```

- Conecte el dispositivo Android al puesto de desarrollo utilizando un cable USB.
- Ejecute de nuevo el comando `lsusb`. Aparece una nueva línea asociada al dispositivo Android.

```
$ lsusb
Bus 001 Device 003: ID 18d1:4e12 Google Inc. Nexus One Phone
Bus 001 Device 002: ID 80ee:0021
Bus 001 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
```

- En esta nueva línea, anote el código de fabricante que se corresponde con los cuatro primeros caracteres situados tras ID. En nuestro ejemplo, este código es *18d1*.
- Desconecte el dispositivo Android del puesto de desarrollo.

Para que Ubuntu pueda reconocer un dispositivo Android, es preciso proporcionar un archivo de reglas que contengan la configuración USB correspondiente a este dispositivo.

- Conéctese como `root`.
- Cree el archivo `/etc/udev/rules.d/##-android.rules` reemplazando **##** por *70*para las versiones de Ubuntu 9.10 y superiores, y por *51* para las versiones inferiores.
- Inserte en el archivo la siguiente línea correspondiente a la versión de Ubuntu del puesto de desarrollo:

Ubuntu 10.10 y superiores:

```
SUBSYSTEM==”usb”, ATTR{idVendor}==”18d1", MODE=”0666"
```

Ubuntu 7.10 a 10.04:

```
SUBSYSTEM==”usb”, SYSFS{idVendor}==”18d1", MODE=”0666"
```

Ubuntu 7.04 e inferiores:

```
SUBSYSTEM==”usb_device”, SYSFS{idVendor}==”18d1", MODE=”0666"
```

- Reemplace en esta línea el código *18d1* por el código de fabricante correspondiente al dispositivo Android anotado anteriormente, o selecciónelo de la siguiente lista:

Fabricante	Código

Acer	0502
Dell	413c
Foxconn	0489
Garmin-Asus	091E
Google	18d1
HTC	0bb4
Huawei	12d1
Kyocera	0482
LG	1004
Motorola	22b8
Nvidia	0955
Pantech	10A9
Samsung	04e8
Sharp	04dd
Sony Ericsson	0fce
ZTE	19D2

- Ejecute el siguiente comando reemplazando `##-android.rules` por el nombre del archivo creado anteriormente.

```
$ chmod a+r /etc/udev/rules.d/##-android.rules
```

- Reinicie el demonio `udev` para que tenga en cuenta este nuevo archivo de reglas utilizando uno de los siguientes comandos:

```
$ sudo reload udev
$ sudo service udev reload
$ sudo restart udev
```

- Vuelva a conectar el dispositivo Android al puesto de desarrollo.

Verificación de la conexión

Llegados a este punto, es importante verificar que el entorno de desarrollo reconoce bien el dispositivo Android conectado.

- Abra una consola y ejecute el siguiente comando:

```
$ adb devices
```

El cliente adb se encuentra en la carpeta `sdk/platform-tools`. Por comodidad, se recomienda agregar esta ruta a la variable PATH del sistema.

Debe aparecer la siguiente respuesta, con un identificador del dispositivo distinto al indicado aquí. Si tal es el caso, el dispositivo Android ha sido detectado correctamente.

```
List of devices attached
HT018PXXXXXX    device
```

Las combinaciones en las configuraciones del entorno de desarrollo y los dispositivos Android son tan numerosas, que es posible que el dispositivo no se reconozca. En este caso, tras verificar que se han seguido correctamente las instrucciones, se recomienda realizar alguna búsqueda en línea. Otros usuarios habrán tenido seguramente el mismo problema y le ofrecerán sin duda alguna solución.

Ejecutar la aplicación

- Lance la ejecución de la aplicación en el dispositivo.

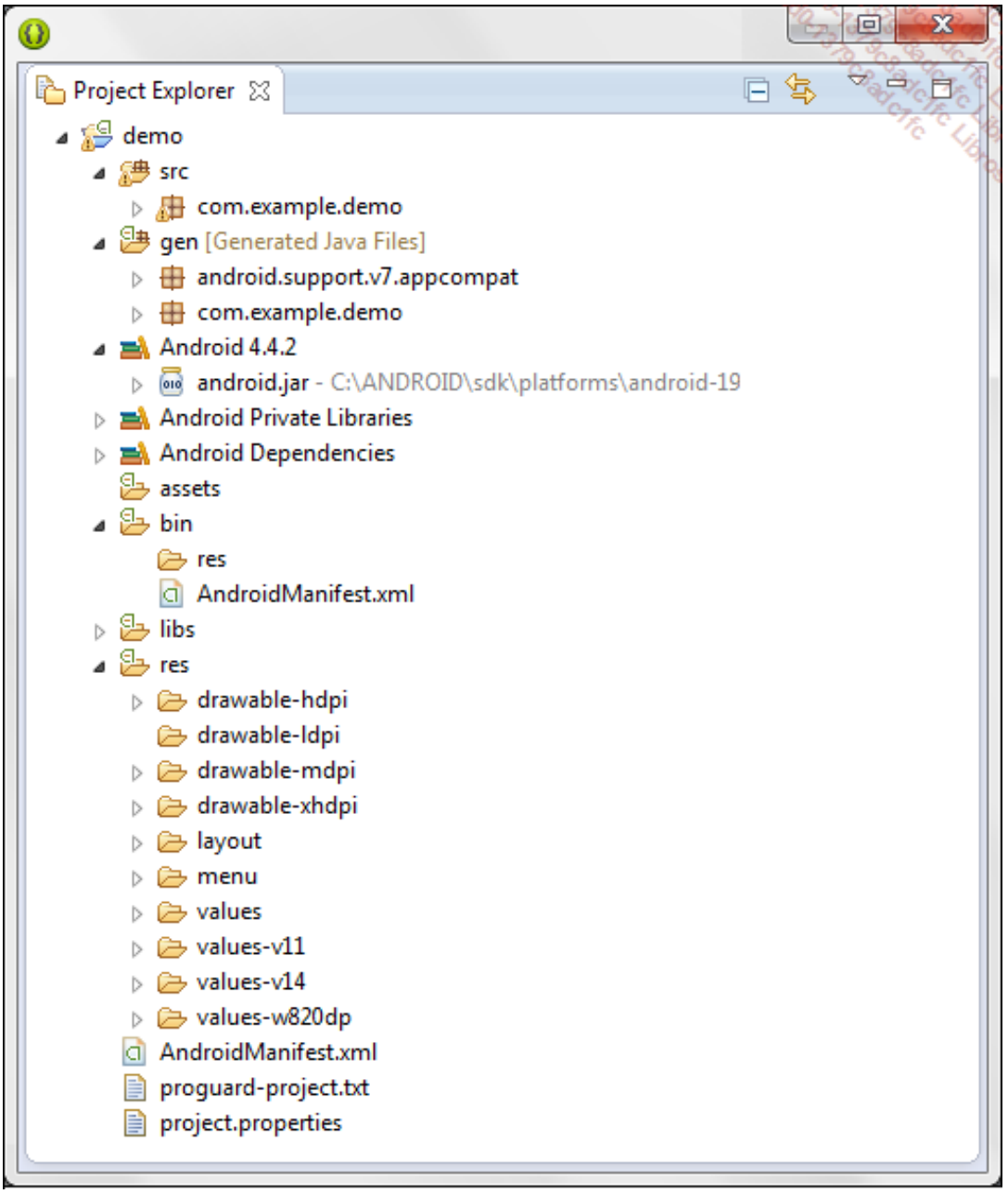
La aplicación se carga rápidamente y se ejecuta automáticamente.

Si aparece la pantalla de desbloqueo, es preciso desbloquear el dispositivo antes de poder ver la aplicación.

Estructura de un proyecto Android

Ahora que hemos configurado un terminal Android, sea emulado o físico, descubriremos más adelante qué contiene el proyecto creado y, de forma más general, qué constituye un proyecto Android.

Un proyecto Android contiene varias carpetas y algunos archivos de configuración en su raíz. En la vista Explorador de Eclipse, tenemos los elementos siguientes:



- La carpeta `src` contiene el código fuente Java del proyecto.
- La carpeta `gen` contiene los archivos generados automáticamente por el plug-in ADT, como elarchivo `R.java`. Este último permite identificar fácilmente los recursos del proyecto desde el código Java asociando una constante con el identificador único de un recurso.
- La carpeta `Android` o `Google APIs` incluye el SDK de la versión concreta de compilación tras la creación del proyecto.
- La carpeta `assets` permite almacenar archivos de datos brutos como, por ejemplo, archivos MP3, vídeos, etc. La aplicación podrá a continuación leer estos archivos en su formato original bajo la forma de flujo de bytes. El formato del archivo se conserva tal cual.
- La carpeta `res` contiene todos los archivos de recursos de la aplicación clasificados en las subcarpetas según su tipo (imágenes, cadenas de caracteres, layouts, etc.). Estos archivos, a

diferencia de los que se almacenan en la carpeta `assets`, pueden modificarse para optimizarse durante la compilación del proyecto.

- El archivo `AndroidManifest.xml` es el archivo de manifiesto del proyecto. Este archivo se describe más adelante en este capítulo.
- El archivo `project.properties` contiene las propiedades del proyecto. Este archivo no debe editarse de forma manual dado que se actualiza automáticamente cuando el desarrollador edita las propiedades del proyecto mediante la interfaz de Eclipse (haciendo clic con el botón derecho en el nombre del proyecto **Android** y, a continuación, **Propiedades**).

Existe también la carpeta `bin`, que contiene los archivos compilados y el archivo binario final de la aplicación. La carpeta `libs` está destinada a contener librerías de terceros.

1. El manifiesto

El archivo `AndroidManifest.xml` es el archivo de manifiesto del proyecto. Contiene la configuración principal de la aplicación.

En lo sucesivo en este libro, nos referiremos a este archivo llamándolo simplemente manifiesto.

El manifiesto es un archivo en formato XML. El desarrollador puede modificarlo directamente o utilizar la vista que proporciona el plug-in ADT.

Este archivo permite, entre otros, indicar los componentes definidos por la aplicación, el punto de entrada principal, los permisos de seguridad necesarios de la aplicación, las bibliotecas requeridas, los entornos de sistema y de hardware compatibles para que funcione la aplicación…

Cualquier componente definido por la aplicación como, por ejemplo, una actividad, un servicio… debe estar declarado en el `manifiesto`; sin él, el sistema Android no lo conocerá y no permitirá por tanto usarlo.

Existen demasiadas etiquetas y atributos como para detallarlos todos en este libro. Muchos atributos son opcionales. Sólo algunos son obligatorios.

Las principales etiquetas y sus atributos se describirán conforme sea necesario en los capítulos dedicados a los componentes o a las funcionalidades que representan.

He aquí los primeros elementos que figuran en este archivo.

Sintaxis

```
<xml version="1.0" encoding="utf-8">
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="cadena de caracteres"
    android:versionCode="entero"
    android:versionName="cadena de caracteres">

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

  <application android:icon="recurso gráfico"
               android:label="recurso texto">
    <activity android:name="cadena de caracteres"
              android:label="recurso texto">
      <intent-filter>
        <action android:name="cadena de caracteres" />
```

```
        <category android:name="cadena de caracteres" />
      </intent-filter>
    </activity>
  </application>

</manifest>
```

La primera línea del archivo describe el formato del archivo y su codificación.

Ejemplo

```
<xml version="1.0" encoding="utf-8">
<manifest
    xmlns:android="http://schemas.android.com/apk/res/android"
    package="es.midominio.android.miaplicacion"
    android:versionCode="1"
    android:versionName="1.0">

  <uses-sdk android:minSdkVersion="10"
            android:targetSdkVersion="18" />

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

En este ejemplo, el contenido de la etiqueta `intent-filter` de la actividad indica que esta actividad es el componente principal de la aplicación, es decir la que hay que ejecutar para iniciar la aplicación, y que puede aparecer en las aplicaciones que contienen el inventario de actividades principales, como la aplicación Lanzador de aplicaciones del sistema Android. Todo esto se describe con más detalle en los capítulos siguientes.

a. Etiqueta manifest

La etiqueta raíz del archivo es la etiqueta `manifest`. Contiene directa o indirectamente a todas las demás etiquetas.

Incluye varios atributos. El primero `xmlns:android` define el espacio de nombres Android que se utilizará en el archivo. Su valor debe ser obligatoriamente`http://schemas.android.com/apk/res/android`.

El atributo `package` contiene el paquete de la aplicación que se solicitó durante la creación del proyecto. Es el identificador único de la aplicación y debe respetar el formato estándar para los nombres de dominio.

A continuación vienen los atributos `android:versionCode` y `android:versionName` que representan en número y letra la versión de la aplicación (véase el capítulo Publicar una aplicación - Preliminares).

b. Etiquetas uses-sdk

La etiqueta `uses-sdk` permite indicar las versiones del SDK sobre las que podrá ejecutarse la aplicación. Android utiliza los valores de los atributos de esta etiqueta para autorizar o no la

instalación de la aplicación. Play Store las utiliza también para filtrar las aplicaciones compatibles con el sistema Android del dispositivo del usuario.

El atributo `android:minSdkVersion` indica el nivel mínimo de API requerido por la aplicación. Este valor depende de las API utilizadas por la aplicación. Es importante tener en cuenta que cuanto menor sea este valor mayor será el número de sistemas Android que podrán ejecutar la aplicación. Aunque en este caso, la aplicación no podrá utilizar las novedades aportadas por las últimas versiones de la API. Es preciso encontrar el equilibrio adecuado entre la elección de la API que se quiere utilizar y las versiones de los sistemas Android candidatos.

Preste atención y verifique bien que el valor del atributo `android:minSdkVersion` no es demasiado bajo respecto a las API utilizadas, pues si la aplicación utiliza una API de alguna versión superior a la especificada en el atributo `android:minSdkVersion`, la aplicación no funcionará en un sistema Android de alguna versión que no contenga la API utilizada.

Si aparece el atributo `android:targetSdkVersion`, indica el nivel de API utilizado durante el desarrollo de la aplicación y sobre el que se ha probado la aplicación. El sistema Android utiliza este atributo para determinar si es preciso o no agregar, en algún caso, mecanismos de compatibilidad. Si el sistema posee la misma versión que este atributo, el sistema Android no hace nada. En caso contrario, puede agregar mecanismos de compatibilidad. Esto permite a nuestra aplicación aplicar por defecto el tema gráfico del sistema de esta versión y ser compatible con los tamaños y densidades de pantalla de esta versión del sistema.

c. Etiqueta application

La etiqueta `application` contiene información sobre la propia aplicación. Existen dos atributos esenciales. Se trata de los atributos `android:icon` y `android:label` que permiten especificar respectivamente el icono y el título de la aplicación. Este icono y este título se utilizarán para representar a la aplicación en diversos lugares, como por ejemplo el propio dispositivo Android o Play Store.

Si bien el atributo `android:label` acepta directamente cadenas de caracteres, se aconseja indicar solamente referencias a recursos. Esto presenta múltiples ventajas: la separación entre código y recursos, o la traducción automática de la cadena de caracteres, etc. como veremos más adelante.

Esta etiqueta contiene etiquetas que describen los componentes de la aplicación. Se trata, entre otros, de las etiqueta `activity`, `service`, `provider`, `receiver` y `uses-library`. Estas etiquetas y sus atributos se estudiarán conforme avancemos a lo largo del libro.

2. Los recursos

Los recursos de una aplicación Android son los datos que utiliza. Puede tratarse de una imagen como un icono de la aplicación, un sonido, un texto como el título de la aplicación...

Estos recursos se incorporan en formato binario a la aplicación. Forman parte integrante del proyecto Android y deben almacenarse en subcarpetas dentro de la carpeta `res` según el tipo de datos que contengan.

- `res/anim`: animaciones de transición definidas por archivos con formato XML.
- `res/color`: colores con varios estados definidos en archivos con formato XML.
- `res/drawable`: imágenes con formatos 9-Patch, PNG, GIF, JPEG y BMP y diseños de tipo`Drawable` definidos en archivos con formato XML.
- `res/layout`: descripciones de interfaces gráficas definidas en archivos con formato XML.

- `res/menu`: menús de distintos tipos definidos en archivos con formato XML.
- `res/raw`: datos brutos, idénticos a los que pueden almacenarse en la carpeta `assets` del proyecto, pero accesibles únicamente mediante su identificador único de recurso.
- `res/values`: valores simples de distintos tipos como, por ejemplo, cadenas de caracteres, valores enteros… contenidos en archivos con formato XML. Cada uno de estos valores es accesible mediante su identificador único de recurso.

Si bien los nombres de los archivos contenidos en la carpeta res/values se dejan a la elección del desarrollador y los archivos que puede contener corresponden a distintos tipos de recursos sencillos, es habitual clasificar los recursos en archivos independientes según su tipo. Los nombres de estos archivos se corresponden con el tipo de recurso que contienen. Por ejemplo,`strings.xml` para recursos de tipo `string`, `arrays.xml` para los conjuntos, `colors.xml`para los colores...

- `res/xml`: archivos XML diversos, en particular archivos de configuración de componentes, que se leen en tiempo de ejecución.

Todas estas carpetas no se crean por defecto. Es preciso crearlas a medida que se van necesitando.

No es posible crear subcarpetas en estas subcarpetas para organizar todavía mejor los recursos... lo cual puede resultar algo molesto en proyectos que utilicen una gran cantidad de recursos del mismo tipo.

Por defecto, un nuevo proyecto incluirá las carpetas `res/drawable-hdpi`, `res/drawable-mdpi` y `res/drawable-ldpi`. De forma general, los términos que aparecen tras el guión determinan la configuración de software o hardware que utilizan específicamente este recurso. Los términos `hdpi`, `mdpi` y `ldpi` en particular, describen la densidad de pantalla, y se estudiarán con mayor detalle en el siguiente capítulo.

Destaquemos no obstante que más allá de los recursos de tipo `Drawable`, cualquier recurso puede verse afectado por una configuración concreta de software o hardware. Para ello, el nombre de la carpeta de recursos debe completarse por el o los calificativos correspondientes separados por guiones. Existe un gran número de calificativos disponibles que representan el tamaño de la pantalla, su densidad, el idioma del sistema, la orientación de la pantalla… Descubriremos estos calificativos en los siguientes capítulos.

Sintaxis

```
tipo_de_recurso[-calificativo_1]...[-calificativo_n]
```

Ejemplo

res/drawable-hdpi

res/drawable-es-hdpi

En este ejemplo, el último caso corresponde con recursos de tipo `Drawable` que se utilizarán en un sistema cuya pantalla esté categorizada como de alta densidad de píxeles y en el que el idioma del sistema sea el español.

Se recomienda que cada recurso disponga de una opción por defecto. Es decir que el recurso debe figurar como mínimo en su carpeta de recurso por defecto, que es la que no lleva ningún calificativo. Por ejemplo, todo recurso de tipo imagen debe figurar como mínimo en la carpeta`res/drawable`. Esto permite asegurar la compatibilidad de la aplicación en las versiones anteriores de Android.

Cada recurso de la carpeta `res` posee su propio identificador único interno de la aplicación. Este valor, un entero, se genera automáticamente cuando se agrega un recurso al proyecto.

Sería laborioso, para el desarrollador, utilizar estos identificadores internos en el código (los valores son, por otro lado, susceptibles de verse modificados conforme se agregan/eliminan recursos), de modo que en su lugar el compilador genera automáticamente una clase Java R (por *Resource*) que realiza la correspondencia entre los identificadores únicos y constantes con un nombre explícito. El desarrollador utilizará, por tanto, estas constantes en lugar de los identificadores únicos generados, bien sea en el código Java o en los archivos XML.

Las constantes, estáticas y públicas, se definen en las subclases de la clase R, donde cada subclase se corresponde con un tipo de recurso - layout, string, drawable, etc.

- En el caso de que el recurso sea un archivo, se utiliza el nombre del archivo, sin extensión.
- Para recursos de tipo values, se utilizará el nombre especificado por el atributo name de la etiqueta.

La notación utilizada difiere según la constante se utilice en el código Java de una actividad o en un archivo XML (un archivo de layout, por ejemplo).

Sintaxis en código Java

```
[paquete.]R.tipo.nombre
```

Ejemplos

R.drawable.icono

R.layout.main

R.string.mensaje

R.color.rojo

Sintaxis en código XML

```
@[paquete:]tipo/nombre
```

Ejemplos

@drawable/icono

@layout/main

@string/mensaje

@color/rojo

En nuestra aplicación podemos ver, por ejemplo, que la cadena de caracteres "Hello world" se define en el archivo de recursos `strings.xml` de la carpeta `/res/values`. El recurso correspondiente posee el nombre `hello_word`, y se utiliza en el archivo que define el `layout` de la actividad, es decir el archivo `/res/layout/activity_main.xml`, bajo la forma `@string/hello_world`.

El código java de la actividad utiliza, él mismo, el recurso `activity_main.xml` (archivo de layout), bajo la forma `R.layout_activity_main`.

Por último, destacaremos algunas restricciones de este sistema de direccionamiento por constantes que aplican a los recursos de un proyecto Android:

- Los nombres de los recursos pueden estar compuestos por caracteres en minúsculas, cifras del 0 al 9 y del carácter _ : `[a-z]`, `[0-9]`, _
- Como las constantes Java, el nombre debe comenzar por un carácter alfabético `[a-z]`.

- Dos recursos de tipo archivo no pueden diferenciarse únicamente por la extensión del archivo (por ejemplo, mi_imagen.jpg y mi_imagen.png).

Introducción

Las aplicaciones Android proporcionan, la mayor parte de ellas, una interfaz de usuario. Ésta va a permitir una interacción bidireccional entre el usuario y la aplicación de forma visual, táctil y sonora.

Este capítulo presenta las bases de la construcción de la interfaz de usuario. Comenzamos analizando el mecanismo de gestión de las distintas resoluciones de pantalla de los dispositivos Android disponibles en el mercado y, a continuación, veremos un primer enfoque de las actividades, pilares de toda aplicación que presente una interfaz de usuario. Veremos, a continuación, con detalle los fundamentos del diseño gráfico de interfaces y estudiaremos, por último, algunos componentes entre los más utilizados en las pantallas de una aplicación. A continuación, se estudiarán con detalle las actividades, en el siguiente capítulo.

Es habitual hablar de clic para designar la acción de presionar y levantar la presión rápidamente sobre un botón de un dispositivo de tipo puntero como, por ejemplo, un ratón. Si bien la mayoría de términos Android pueden describirse sin hacer referencia al ratón, el término clic se utilizará para describir el contacto del dedo sobre la pantalla, por estar lo suficientemente extendido en el vocabulario corriente.

Pantallas

Una de las particularidades del sistema Android es que puede adaptarse a un gran número de configuraciones de hardware diferentes.

El hecho que mejor pone de manifiesto esta diversidad es el tamaño de las pantallas de los dispositivos Android. Esto abarca desde un pequeño smartphone donde la diagonal de la pantalla mide 6,5 cm hasta una tableta táctil de 30 cm, pasando por un gran número de dispositivos de tamaño intermedio, sin contar los televisores que cuentan con el sistema Google TV basado en Android...

No obstante el tamaño es sólo una de las características de la pantalla. La resolución de la pantalla es otra, yendo por lo general de la mano de la primera, una pequeña pantalla presenta poca resolución y una gran pantalla presenta una resolución mucho mayor. Conviven dispositivos Android con una resolución de 240 x 320 píxels (QVGA, *Quarter Video Graphics Array*) y terminales que muestran 2560 x 1600 píxels, la versión 4.3 de Android incluye soporte para pantallas llamadas 4K, ¡que presentan una resolución de 4096 x 2160 píxels!.

En pocas palabras, vemos que el sistema Android puede funcionar en muchas y muy diversas configuraciones de pantalla. En lo sucesivo una aplicación pensada para una pantalla de smartphone de poca resolución tendrá un aspecto mediocre en la inmensidad de una pantalla más grande de alta resolución. Con el objetivo de ofrecer al usuario la mejor experiencia posible, el desarrollador deberá arreglárselas para que la aplicación se adapte también a los distintos tamaños y resoluciones de pantalla de los dispositivos Android sobre los que se puede ejecutar la aplicación.

Android clasifica las pantallas según el tamaño de su diagonal en cuatro categorías: `small`(pequeña), `normal` (normal), `large` (grande) et `xlarge` (muy grande). Esta última categoría existe desde Android 2.3 (API 9). Encontraremos por ejemplo pantallas de unos 6,35 cm (2,5 pulgadas) de tamaño de diagonal en la categoría de pantallas pequeñas, de unos 10 cm (4 pulgadas) en la categoría de pantallas normales, de unos 18 cm (7 pulgadas) para las grandes pantallas y de más de 25 cm (10 pulgadas) para pantallas extragrandes.

En una misma categoría pueden coexistir varias resoluciones. Si bien para una misma superficie de pantalla el número de píxeles puede variar sensiblemente. El número de píxeles de una superficie determina su densidad. Se expresa en puntos por pulgada llamados `dpi` (*dots-per-inch*).

Android clasifica así las pantallas en categorías según su densidad de píxeles: `ldpi` (baja), `mdpi`(media), `hdpi` (alta) y `xhdpi` (muy alta), así como `xxhdpi`, o incluso `xxxhdpi`. Esta última categoría existe desde Android 2.2 (API 8). Encontraremos por ejemplo valores de 120 dpi en la categoría de densidades bajas, 160 dpi en la categoría media, 240 dpi en la categoría alta y 320 dpi en la categoría muy alta. Estos valores de densidad pueden servirnos como referencia.

Categoría de pantalla	Densidad en dpi
ldpi	120
mdpi	160
hdpi	240
xhdpi	320
xxhdpi	480
xxxhdpi	640

Aun así, para un tamaño de pantalla estrictamente igual, un botón de 100 píxeles de largo, por ejemplo, tendrá un tamaño de visualización diferente si las resoluciones son diferentes. Para evitar esto, Android recomienda utilizar una nueva unidad: el `dp` o `dip` (*density-independent pixel*) o píxel independiente de la densidad.

¡No confundir dpi y dip!

El sistema se encarga de convertir un valor determinado en dip en número de píxeles físicos teniendo en cuenta la densidad de la pantalla. La dimensión física resultante será la misma sea cual sea ladensidad de la pantalla. Es decir, el botón tendrá el mismo tamaño físico en todas las pantallas.

En Android, las dimensiones pueden especificarse en distintas unidades. En la siguiente tabla se muestra un inventario de estas unidades:

Unidad	Descripción
`dp` o `dip`	Unidad independiente de la densidad de la pantalla (*density-independent pixel*).
`sp` o `sip`	Unidad dependiente del tamaño preferido para la fuente de caracteres (*scale-independent pixel*). Esta unidad se utiliza para ajustar el tamaño de la letra en cadenas de caracteres.
`px`	Pixel. Unidad gráfica mínima.
`in`	Pulgada. Unidad de medida que vale 2,54 cm.
`mm`	Milímetro.

Con el objetivo de no confundir dip, sip y dpi, es preferible utilizar únicamente dp y sp.

Para que una aplicación pueda adaptarse correctamente a todos los tipos de pantalla, tamaños y resoluciones, se recomienda encarecidamente utilizar únicamente dimensiones independientes de la densidad, es decir especificadas en dp (o dip) y en sp (o sip).

Es posible especificar distintas variantes de una misma imagen según la densidad de la pantalla. Android buscará la versión correspondiente a la categoría de densidad de la pantalla en curso. Si existe, la utilizará. Si no, Android buscará este recursos para otra densidad, de forma prioritaria en las densidades mayores que la actual. Una vez encontrado, el sistema lo redimensionará utilizando cierto factor.

Para conocer el factor a aplicar, el sistema utiliza la siguiente escala indicando los distintos ratios: 3-4-6-8-12-16 que se corresponden respectivamente con las distintas densidades: `ldpi`, `mdpi`, `hdpi`,`xhdpi`, `xxhdpi` y `xxxhdpi`.

Por ejemplo, si la densidad de la pantalla es alta (`hdpi`) y el recurso no existe en esta densidad ni en una densidad superior (`xhdpi`), el sistema puede utilizar la densidad media (`mdpi`) suponiendo que exista. A continuación el sistema modifica automáticamente su tamaño aplicando un factor de 1,5, es decir 6/4, para las densidades altas (`hdpi`). Si la pantalla tiene una densidad baja (`ldpi`), el factor utilizado es 0,75, es decir 3/4.

El desarrollador que quiera aportar imágenes dedicadas a una categoría de densidades específica podrá utilizar los mismos coeficientes multiplicadores para determinar el tamaño que quiere proporcionar para estas densidades, o utilizar la siguiente fórmula:

px=dp*(dpi/160)

Esta fórmula permite confirmar que, para una densidad media, un dp equivale a un píxel.

Por ejemplo, si aporta una imagen de 100x100 píxeles para una densidad normal (`mdpi`) y quiere ofrecer una para una densidad alta, esta imagen deberá tener un tamaño de 150x150 píxeles, aplicando un coeficiente multiplicador de 1,5.

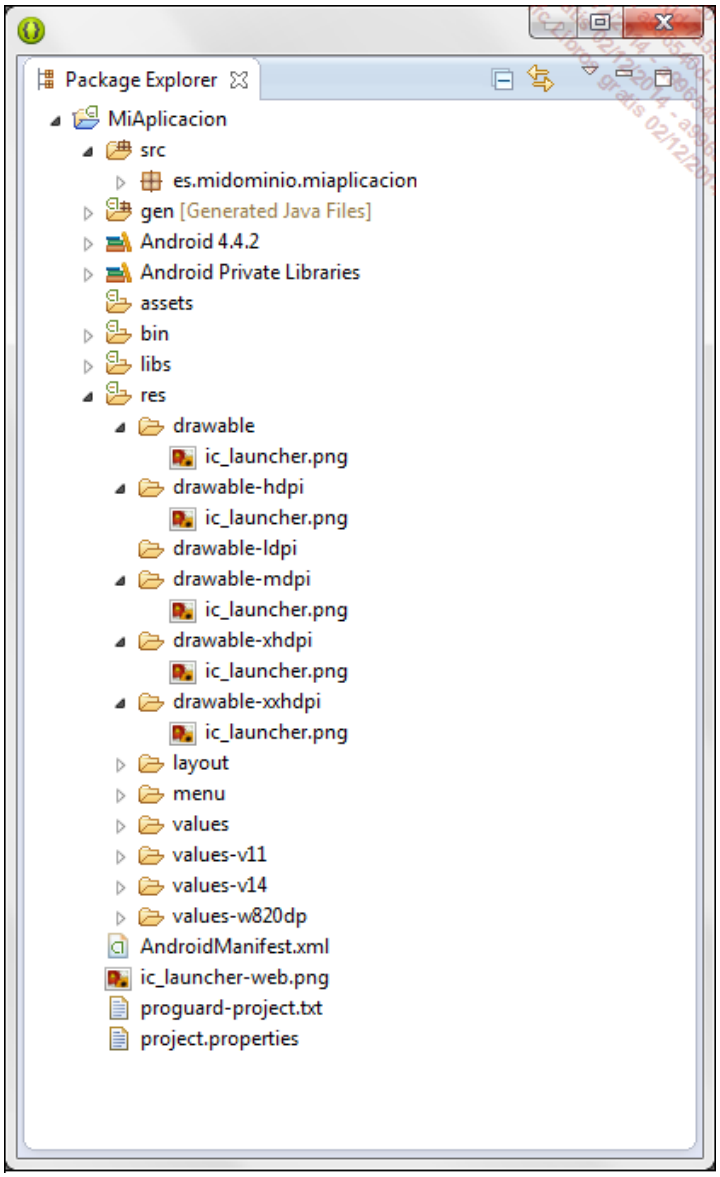
Cada versión de la imagen correspondiente a una densidad específica debe almacenarse en la subcarpeta `drawable` de la carpeta `res` donde el calificativo se corresponda con la categoría de densidad de la pantalla.

Sintaxis

```
drawable-categoría_densidad
```

Ejemplo

En la siguiente captura de pantalla, el recurso ic_launcher está disponible para las resoluciones xxhdpi, xhdpi, hdpi y mdpi. El recurso no existe para la resolución ldpi: en este caso se utilizará la versión mdpi transformada.



Los recursos que se encuentran en la carpeta `res/drawable` están considerados recursos para una densidad media (`mdpi`).

Desde Android 1.6 (API 4), es posible utilizar la etiqueta `supports-screen` en el manifiesto para indicar al sistema, y de forma accesoria en Google Play Store, con qué categorías de tamaño de pantalla es compatible la aplicación.

A cada una de estas categorías le corresponde un atributo de esta etiqueta:`android:smallScreens`, `android:normalScreens`, `android:largeScreens` y`android:xlargeScreens`. Éstas toman un valor booleano que indica si la aplicación soporta la categoría correspondiente o no. Si no es el caso, el sistema busca la mejor solución de visualización compatible posible y convierte automáticamente las dimensiones para adaptarlas a la pantalla utilizada.

Los valores por defecto de estos atributos dependen de los valores de los atributos`android:minSdkVersion` y `android:targetSdkVersion`. En lo sucesivo, lo más sencillo es indicar estos valores explícitamente en el manifiesto para evitar cualquier ambigüedad.

El atributo `android:anyDensity` toma un valor booleano que indica si la aplicación ha sido diseñada de forma independiente a la densidad, o más concretamente si utiliza dimensiones de tipo dp o sp o bien convierte las dimensiones especificadas en píxeles. En caso contrario, el sistema ejecutará la aplicación en un modo de compatibilidad.

Recuerde, se recomienda encarecidamente realizar aplicaciones cuya interfaz sea independiente de la densidad de la pantalla.

Sintaxis

```
<supports-screens
```

```
  android:smallScreens="booleano"
  android:normalScreens="booleano"
  android:largeScreens="booleano"
  android:xlargeScreens="booleano"
  android:anyDensity="booleano"
  ...
/>
```

Ejemplo

```
<xml version="1.0" encoding="utf-8">
<manifest
    xmlns:android="http://schemas.android.com/apk/res/android"
    package="es.midominio.android.miaplicacion"
    android:versionCode="1"
    android:versionName="1.0">

  <uses-sdk android:minSdkVersion="8"
            android:targetSdkVersion="11" />

  <supports-screens
      android:smallScreens="true"
      android:normalScreens="true"
      android:largeScreens="true"
      android:xlargeScreens="true"
      android:anyDensity="true" />

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

Actividades y Layout

En el universo Android, las actividades (*Activity* en inglés) forman parte de los objetos más utilizados. Cada pantalla que ve y manipula el usuario está, en efecto, implementada por una clase que hereda de la clase `Activity`. Es, por tanto, esencial entender perfectamente todos los conceptos que aporta esta clase.

```
public class MiActividadPrincipal extend Activity {
[...]
}
```

Salvo excepciones (principalmente con los servicios), una aplicación incluye como mínimo una clase que hereda de `Activity` y puede, por supuesto, incluir varias. Solamente se ejecuta una actividad (y sólo una) al inicio de una aplicación.

Cada actividad, para poder iniciarse, debe declararse en el archivo `Manifest.xml`, dentro de una etiqueta `<activity>`, etiqueta hija de la etiqueta `<application>`.

Sintaxis

```
<activity android:icon="recurso drawable"
          android:label="recurso texto"
          android:name="cadena de caracteres"
          ... >
    ...
</activity>
```

El siguiente capítulo, Los fundamentos, estudia con detalle esta sección del archivo `Manifest.xml`: de momento bastará con saber que para indicar al sistema qué actividad ejecutar tras el inicio de la aplicación será preciso agregar una etiqueta `<intent-filter>` que contenga las siguientes etiquetas en su interior:

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

La declaración de la actividad `MiActividadPrincipal` como punto de entrada de la aplicación tendrá, por tanto, la siguiente sintaxis:

```
[...]
  <application android:icon="@drawable/icon"
               android:label="@string/app_name">
  <activity android:name=".MiActividadPrincipal"
            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>
[...]
```

La clase `Activity` forma parte del paquete `Android.app`. El punto de entrada de una clase que hereda de `Activity` es el método `onCreate`: se trata del primer método que se invoca tras la creación de la actividad, y es típicamente dentro de este método donde el desarrollador realizará sus inicializaciones si fuera necesario. El método `onCreate` de la clase que hereda de `Activity` debe, obligatoriamente, invocar al método `onCreate` de la clase madre.

La sintaxis del método `onCreate` es la siguiente:

```
@Override
protected void onCreate(Bundle savedInstanceState)
```

El objeto Bundle que se pasa como parámetro del método `onCreate` es un objeto que contiene los datos guardados tras la destrucción por parte del sistema de una instancia previa de la actividad. Este objeto se estudia en el capítulo Los fundamentos, en la sección Actividad - Salvaguarda y restauración del estado.

En resumen, el código mínimo de una actividad es el siguiente:

```
public class MiActividad extends Activity {

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

}
```

Modo programático y modo declarativo

Para diseñar una interfaz de usuario - es decir para ubicar elementos visuales - el desarrollador cuenta con dos técnicas:

- El modo declarativo: la interfaz se construye escribiendo código XML en archivos XML separados, llamados archivos de layout. Este modo permite aislar el código de la interfaz de usuario del código de la aplicación en Java.
- El modo programático: la interfaz se construye completamente en el código Java de la actividad. Este modo permite generar la interfaz dinámicamente pero tiene el inconveniente de mezclar el código de diseño de la interfaz con el código de la aplicación.

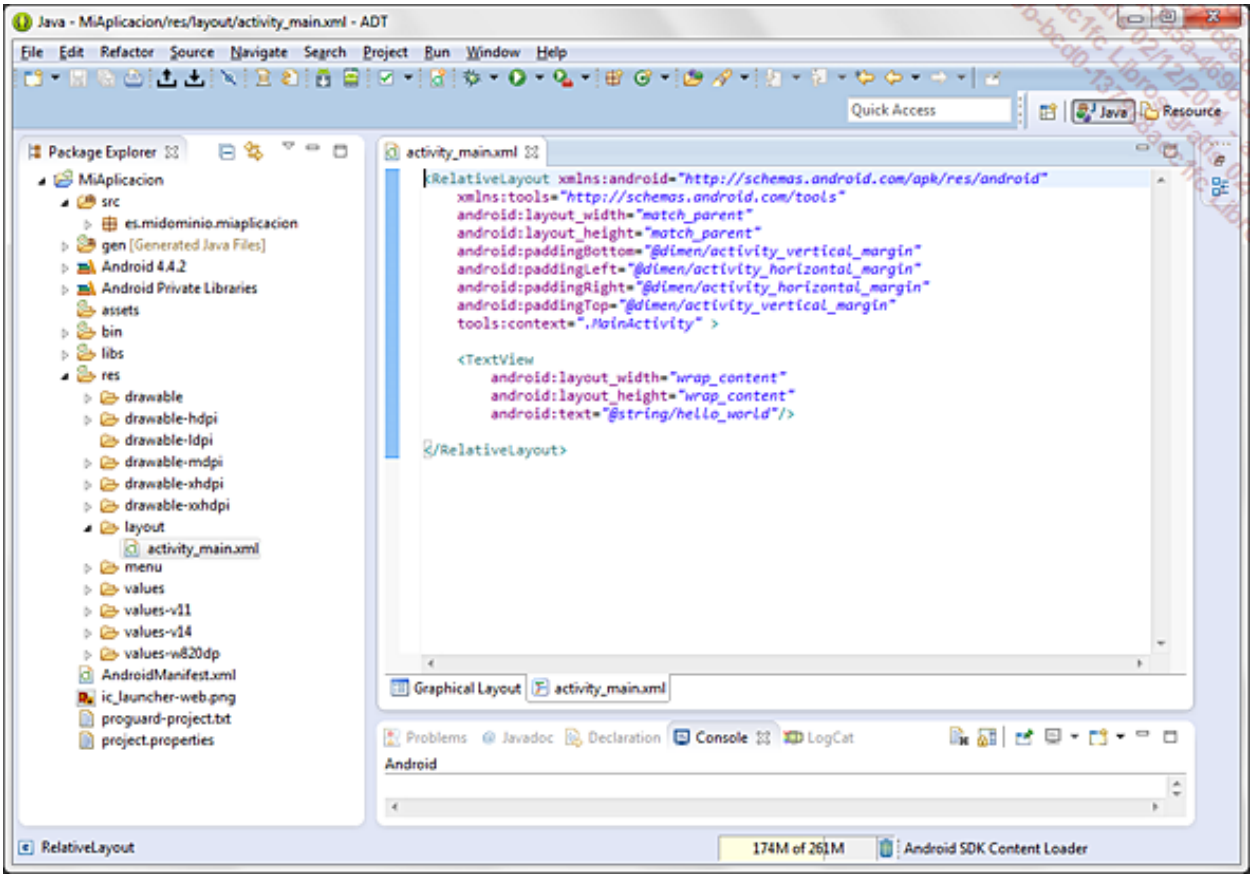
En la mayoría de casos, es habitual construir los layouts en modo declarativo de forma que se pueda separar la interfaz de usuario del código de la aplicación. Se especifican valores por defecto a los atributos de las vistas declaradas que, más adelante, se podrán modificar en tiempo de ejecución de la aplicación mediante el modo programático.

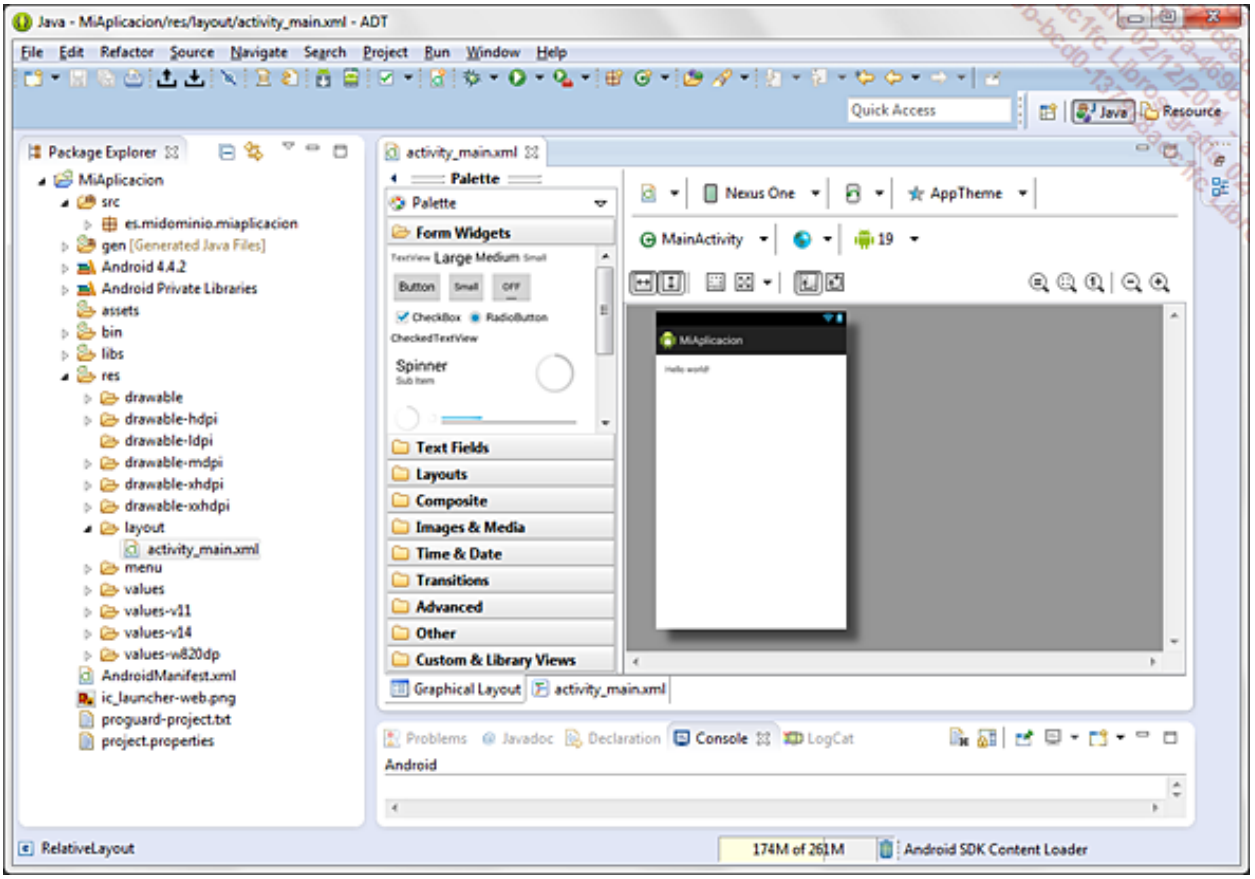
Esto permite tener una visión clara de la jerarquía de vistas establecida en el layout. Esto permite, también, especificar distintos layouts según el tamaño de la pantalla, el idioma utilizado por el usuario... sin tener que modificar el código de la aplicación.

De este modo, se combina lo mejor de ambos métodos. A lo largo de este libro daremos prioridad a esta forma de trabajar para construir los layouts y demás componentes gráficos.

Por otro lado, en Eclipse, el plug-in ADT permite obtener una visualización previa de la apariencia de un archivo de layout, en tiempo de diseño. Android Studio proporciona la misma funcionalidad.

Por ejemplo, al abrir un archivo de layout (véase la sección Layouts en este capítulo) se permite editar el archivo al mismo tiempo mediante un editor de código XML y mediante una interfaz gráfica llamada Graphical Layout.





En el caso de que la interfaz se defina en un archivo de layout, es preciso vincularla a una actividad para que se muestre en tiempo de ejecución. Se utiliza, para ello, el método `setContentView` de la clase `Activity`.

Sintaxis

```
public void setContentView (int layoutResID)
```

Este método recibe como parámetro un número entero que es el identificador único del layout (consulte Estructura de un proyecto Android - Los recursos, en el capítulo Primeros pasos).

Ejemplo

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

En este ejemplo, la actividad en curso utiliza el archivo llamado `/res/layout/layout_principal.xml` para construir la interfaz de usuario que se mostrará tras la creación de la actividad.

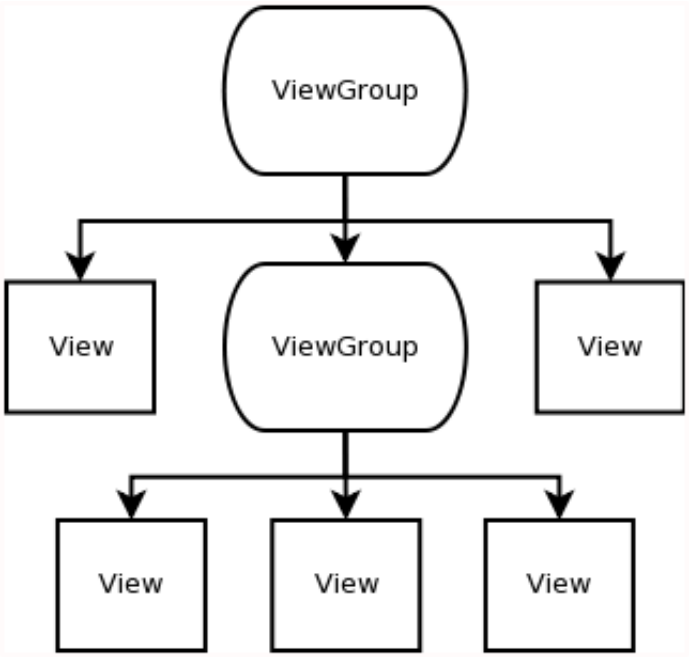
Ahora que se han explicado los conceptos básicos relacionados con la creación de actividades, estudiaremos con más detalle el diseño de layout en modo declarativo.

Vistas

El elemento básico de la interfaz de usuario es la vista (clase `View`).

Todos los demás elementos de la interfaz de usuario heredan de la clase `View`. Este es el caso, en particular, de la clase `ViewGroup` que es una vista que permite contener a otras vistas. Se le denomina contenedor de vistas.

En lo sucesivo, la interfaz de usuario de una actividad estará compuesta por un conjunto de vistas y contenedores de vistas. Dada una actividad, se tendrá por ejemplo una jerarquía de vistas del tipo:



Esta jerarquía permite ordenar la visualización de las vistas. La vista que se encuentra en la raíz de la jerarquía se dibuja en primer lugar. Si se trata de un contenedor de vistas, entonces el contenedor solicita a sus vistas hijas que se dibujen, y así recursivamente hasta haber dibujado todas las vistas.

Una vista posee propiedades. Todas las clases que heredan de la clase `View` heredarán estas propiedades. Veamos las principales:

Propiedad	Descripción
`android:background`	Elemento a utilizar para el fondo de la vista (imagen, color...).
`android:clickable`	Especifica si la vista reacciona a los clics.
`android:id`	Identificador único de la vista.
`android:minHeight`	Altura mínima de la vista.
`android:minWidth`	Longitud mínima de la vista.
`android:onClick`	Nombre del método de la actividad a ejecutar cuando el usuario hace clic en la vista.
`android:padding`	Dimensión del margen interno para los cuatro lados de la vista.
`android:paddingBottom`	Dimensión del margen interno inferior de la vista.
`android:paddingLeft`	Dimensión del margen interno izquierdo de la vista.
`android:paddingRight`	Dimensión del margen interno derecho de la vista.

android:paddingTop	Dimensión del margen interno superior de la vista.
android:tag	Permite asociar un objeto a la vista.
android:visibility	Especifica la visibilidad de la vista.

El atributo android:tag puede presentar distintos roles y revelarse de gran utilidad en un buen número de situaciones. Se trata de una propiedad que se deja al buen uso del desarrollador. Por ejemplo, permite asociar un dato a una vista.

Layouts

Los layouts son contenedores de vistas (`ViewGroup`) predefinidos ofrecidos por Android. Cada layout proporciona un estilo de representación en página distinto permitiendo a las vistas posicionarse las unas respecto a las otras o respecto al contenedor padre.

Los principales layouts predefinidos son `FrameLayout`, `LinearLayout`, `RelativeLayout`,`TableLayout`. Ofrecen respectivamente una estructura de posicionamiento de las vistas en marco, lineal, relativa y en forma de tabla.

Tenga precaución de no confundir los archivos de layout, definidos en la carpeta `/res/layout/`, con los contenedores de vistas: los archivos de layout son archivos que permiten componer la interfaz visual de una actividad en modo declarativo, los contenedores de vista `frameLayout` y`LinearLayout` son objetos que estructuran la presentación en página. Típicamente, un archivo de layout contiene, como primer elemento, un contenedor de vistas (ya sea un `FrameLayout`, un`TableLayout` o, con frecuencia, un `LinearLayout` o un `RelativeLayout`).

FrameLayout

Contenedor reducido a su mínima expresión. Todo lo que contiene se dibujará a partir de la esquina superior izquierda. Los últimos elementos hijos agregados se dibujarán debajo de los más antiguos.

LinearLayout

Contenedor en el que los elementos hijos se disponen en función del valor del atributo de orientación:

- Verticalmente, los unos debajo de los otros, un único elemento por línea.
- Horizontalmente, los unos a continuación de los otros, a la derecha del anterior.

RelativeLayout

Contenedor en el que las posiciones de las vistas hijas están determinadas respecto a la vista padre o respecto a otras vistas hijas. El uso del contenedor `RelativeLayout` se estudia con detalle en el capítulo Construir interfaces complejas, sección Representación en pantalla compleja.

TableLayout

Contenedor en el que los elementos hijos se disponen en forma de tabla. Las vistas hijas son objetos`TableRow` que definen cada uno una línea de la tabla. Los elementos hijos de los objetos`TableRow` son celdas de la tabla. Observe que dichas tablas no disponen, hablando con propiedad, de columnas, como ocurre con HTML.

1. Creación en modo declarativo

El modo declarativo es el modo más sencillo para declarar la interfaz de usuario de una pantalla. Veamos cómo crear un layout en un archivo XML.

La primera etapa consiste en crear un archivo con formato XML, en la carpeta `res/layout` del proyecto.

Recuerde (véase el capítulo Primeros pasos - Estructura de un proyecto Android), el nombre de la carpeta `layout` puede especializarse para filtrar el uso de los layouts según ciertas características. En posible en particular proveer distintos layouts para una misma pantalla, a modo de interfaz, según la categoría del tamaño de la pantalla física. Para ello, el calificador que debe usarse es el nombre de la categoría correspondiente al tamaño de pantalla.

Los recursos que se encuentran en la carpeta `res/layout` están clasificados como recursos

para pantallas de tamaño normal (normal).

Ejemplo

res/layout-small/layout_principal.xml

res/layout-normal/layout_principal.xml

res/layout-xlarge/layout_principal.xml

También es posible crear layouts distintos según la orientación de la pantalla. El calificativo que debe especificar es `port` para el modo vertical y `land` para el modo apaisado. Es posible combinar las distintas categorías de calificadores para afinar al máximo el filtrado.

Ejemplo

res/layout-port/layout_principal.xml

res/layout-xlarge-land/layout_principal.xml

El archivo layout XML debe tener la sintaxis siguiente:

Sintaxis

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

La primera línea describe la versión XML y la codificación del archivo.

La segunda línea especifica el tipo de layout deseado utilizando la etiqueta XML correspondiente. Se especifican a continuación las propiedades de esta etiqueta y, por tanto, del layout.

El layout raíz de la pantalla debe especificar el atributo `xmlns:android` con el valor asignado anteriormente para definir el espacio de nombres android que se utilizará para la declaración de los atributos.

Los atributos `android:layout_width` y `android:layout_height` son obligatorios, y se produce una excepción en caso contrario.

Los posibles valores de estos atributos son:

- `match_parent`: la vista mide igual que la vista padre menos el valor del espaciado (padding). Desde la versión 8 del SDK, este valor reemplaza al valor `fill_parent`.
- `wrap_content`: la vista mide igual que las dimensiones de su contenido más el valor del espaciado (padding). Es decir, el tamaño más pequeño posible pero suficiente como para mostrar su contenido.
- una dimensión: la vista se dimensiona de forma precisa según este valor. Una dimensión descrita por un número, en coma flotante si fuera preciso, seguido de una unidad. Por ejemplo: `42dp`.

Recuerde, para que una aplicación pueda adaptarse a cualquier tipo de pantalla, tamaño y resolución, se recomienda encarecidamente utilizar únicamente dimensiones independientes de la densidad. De este modo, para las dimensiones de un layout, se dará prioridad a los valores`match_parent`, `wrap_content` y a aquellas especificadas en dp.

Ejemplo

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

En este ejemplo, la pantalla aplica un layout relativo cuyas dimensiones son las de la vista padre. Es decir, en este caso la pantalla completa salvo por la barra de estado y la barra de título en la parte superior.

Una buena práctica, que resulta esencial en un entorno de producción, es prefijar los nombres de los archivos XML de layout con el término 'layout_', puesto que en la carpeta layout puede haber, como veremos en próximos capítulos, otros tipos de archivos. Como resulta imposible crear subcarpetas en la carpeta `/res/layout`, es bastante difícil ubicarse si no se adquiere este hábito.

2. Creación en modo programático

El modo programático es el más flexible y el más potente de los dos modos. En contrapartida, también es el más complejo.

Permite generar dinámicamente la interfaz de usuario tras la creación de la actividad. También, y principalmente, permite modificar esta interfaz dinámicamente durante la ejecución de la aplicación, por ejemplo, cuando un botón **OK** deshabilitado debe activarse si el usuario rellena todos los campos de un formulario.

Creación del layout

Para crear un layout, es preciso instanciar un nuevo objeto de la clase del tipo de layout deseado. Tomemos, por ejemplo, un layout relativo.

Sintaxis

```
public RelativeLayout (Context context)
```

Este constructor recibe como parámetro un contexto. De momento, quedémonos con la idea de que un contexto sirve para reagrupar un conjunto de información asociada al entorno de la aplicación, y que la clase `Activity` hereda indirectamente de la clase `Context`.

Una vez creado el objeto, hay que especificar sus dimensiones usando su método`setLayoutParams`.

Sintaxis

```
public void setLayoutParams (ViewGroup.LayoutParams params)
```

Este método recibe como parámetro un objeto de tipo `ViewGroup.LayoutParams`. Es preciso crear uno utilizando uno de sus constructores.

Sintaxis

```
public ViewGroup.LayoutParams (int width, int height)
```

Los parámetros `width` y `height` son números enteros. Como hemos visto anteriormente, puede ser un valor a escoger entre `match_parent`, `wrap_content` o una dimensión.

Para terminar, es preciso definir este layout como vista raíz de la actividad para construir su interfaz

de usuario gracias al segundo método `setContentView` que recibe esta vez como parámetro un objeto de tipo `View`.

Sintaxis

```
public void setContentView (View view)
```

Ejemplo

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

    RelativeLayout layout = new RelativeLayout(this);
    LayoutParams lp = new LayoutParams(LayoutParams.MATCH_PARENT,
      LayoutParams.MATCH_PARENT);
    layout.setLayoutParams(lp);
    setContentView(layout);
}
```

En este ejemplo, el constructor `RelativeLayout` recibe como parámetro `this` dado que el método `onCreate` ejecutado corresponde al de un objeto que hereda de la clase `Activity`, heredando ella misma de la clase `Context`.

Widgets

Un widget es un componente de la interfaz gráfica que puede o no permitir la interacción con el usuario.

Android proporciona un conjunto de componentes básicos que permiten crear interfaces gráficas más o menos complejas.

1. Declaración

La declaración de estos componentes se ve facilitada en los archivos de layouts con formato XML. La sintaxis de su declaración es la siguiente.

Sintaxis

```
<Tipo_de_widget
    android:id="@[+][paquete:]id/nombre_recurso"
    android:layout_width="dimensión"
    android:layout_height="dimensión"
    ...
/>
```

La propiedad `android:id` permite asociar un identificador único al componente. Se trata de una propiedad opcional pero, gracias a este identificador, es sencillo recuperar el objeto correspondiente en el código Java y modificarlo a continuación. Es por este motivo por lo que se indica aquí.

Generalmente, el valor del identificador único tiene el formato `@+id/`*`nombre_recurso`*. El signo más indica que se trata de un nuevo identificador. Se le asignará un nuevo valor de forma automática durante la generación del archivo `R.java`, en la subclase `R.id`.

2. Uso

Para poder utilizar el componente desde el código Java de la aplicación, basta con utilizar el método`findViewById` y pasarle como parámetro el identificador único del componente afectado. Este método devuelve un objeto de tipo `View` que hay que convertir a continuación en el tipo de clase adecuado.

Sintaxis

```
public View findViewById (int id)
```

Ejemplo

```
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.layout_principal);
    Button btn = (Button) findViewById(R.id.button_haga_clic);
}
```

3. Descubrir algunos widgets

Sería demasiado extenso presentar de forma exhaustiva todos los componentes y todas sus propiedades. Ciertos componentes pueden tener más de cincuenta propiedades específicas.

Todos estos componentes forman parte del paquete `android.widget`.

Por ello, vamos a presentar únicamente los principales componentes gráficos que proporciona Android y algunas de sus propiedades principales.

En cada ocasión, mostraremos una captura de pantalla junto al trozo de código utilizado en el ejemplo.

a. TextView (campo de texto)

El componente `TextView` permite visualizar texto.

Las principales propiedades del componente `TextView` son:

Propiedad	Descripción
`android:autoLink`	Permite convertir automáticamente los enlaces de hipertexto y las direcciones de correo electrónico en enlaces sobre los que es posible hacer clic.
`android:ellipsize`	Especifica cómo debe mostrarse el texto cuando es más largo que la vista.
`android:gravity`	Indica dónde debe estar posicionado el texto en la vista cuando es más pequeño que ella.
`android:height`	Altura del componente.
`android:lines`	Número exacto de filas que se desea mostrar.
`android:maxHeight`	Altura máxima del componente.
`android:maxLines`	Número máximo de filas que se desea mostrar.
`android:maxWidth`	Longitud máxima del componente.
`android:minLines`	Número mínimo de filas que se desea mostrar.
`android:text`	Texto que se quiere mostrar.
`android:textColor`	Color del texto.
`android:textSize`	Tamaño del texto.
`android:textStyle`	Estilo del texto.
`android:width`	Longitud del componente.

A saber: el componente TextView posee a su vez cuatro propiedades que permiten mostrar a sus lados un recurso de tipo `Drawable`, como por ejemplo una imagen. Éstas son las propiedades: `android:drawableTop`,`android:drawableBottom`, `android:drawableLeft` y `android:drawableRight`.

Ejemplo

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

    <TextView android:id="@+id/textview_mensaje"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerInParent="true"
        android:background="#FFF"
```

Las propiedades del componente pueden modificarse en el código Java utilizando el método`findViewById` como se ha visto anteriormente.

Ejemplo

En este ejemplo, modificamos el texto del componente antes de mostrarlo.

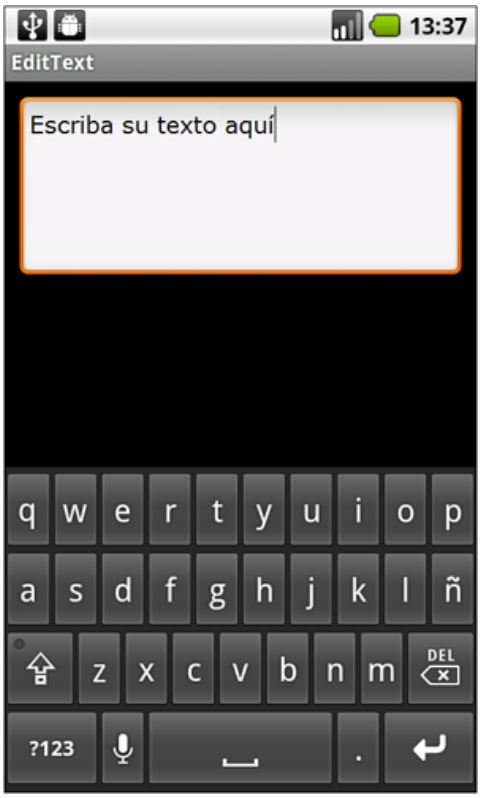
```
        android:textColor="#000"
        android:textSize="14sp"
        android:maxLines="3"
        android:gravity="center"
        android:text="@string/mensaje" />

</RelativeLayout>
```

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

    setContentView(R.layout.textview);
         TextView msg = (TextView) findViewById(R.id.textview_mensaje);
    msg.setText(R.string.nuevo_mensaje);
}
```

El uso de colores para el fondo de las vistas permite verificar rápidamente las dimensiones de las mismas.

b. EditText (campo para introducir texto)

El componente `EditText` permite al usuario introducir texto. Cuando el usuario hace clic sobre el componente, aparece el teclado virtual. El usuario puede utilizar también el teclado físico si el dispositivo posee uno.



Este componente hereda del componente `TextView`. Sus propiedades son las mismas que las del componente `TextView`. Las principales propiedades utilizadas por este componente son:

Ejemplo

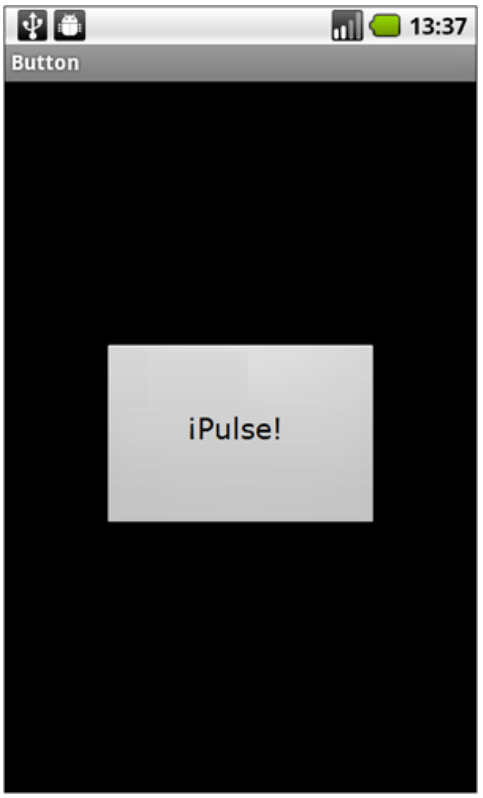
Propiedad	Descripción
`android:inputType`	Permite filtrar el texto introducido por el usuario especificando el tipo de dato que acepta el componente.
`android:scrollHorizontally`	Permite deslizar el texto horizontal cuando es mayor que la longitud del componente.

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

    <EditText android:id="@+id/edit_texto"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_margin="10dp"
        android:background="#FFF"
        android:textColor="#000"
```

```
        android:lines="5"
        android:gravity="top"
        android:inputType="textMultiLine|textAutoCorrect"
        android:text="@string/edit_texto"/>

</RelativeLayout>
```

c. Button (Botón)

El componente `Button` representa un botón sobre el que el usuario puede presionar y hacer clic. Es posible asociar acciones a estos estados.



Este componente hereda del componente `TextView`. Sus propiedades son las mismas que las del componente `TextView`. Las principales propiedades utilizadas por el componente son:

Propiedad	Descripción
`android:text`	Texto a mostrar sobre el botón.
`android:onClick`	Nombre del método de la actividad a ejecutar cuando el usuario hace clic sobre el botón.

El uso de la propiedad `onClick` implica la creación del método correspondiente en la clase de la actividad. Este método debe definirse como `public` y recibir como parámetro un objeto de tipo`View`.

Sintaxis

```
public void nombreMétodo(View vistaAsociada)
```

Ejemplo

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

    <Button android:id="@+id/button1_pulse"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerInParent="true"
        android:padding="50dp"
        android:textSize="20sp"
        android:text="@string/pulse"
        android:onClick="onClickButton" />

</RelativeLayout>
```

```
public void onClickButton(View view) {
    ((Button)view).setText(R.string.pulse);
```

He aquí el método `onClickButton` creado en nuestra actividad.

En este ejemplo, cuando el usuario hace clic sobre el botón, la propiedad `onClick` indica que es preciso ejecutar el método `onClickButton`. Éste recibe como parámetro la vista afectada. Aquí, se trata de un botón y por ello convertimos la vista en un botón para poder modificar su texto.

Existe una alternativa al uso de la propiedad `onClick` que consiste en declarar la acción en el código Java en lugar de en el código XML. El uso cotidiano suele dar

```
}
```

preferencia a esta alternativa, pues permite limitar las dependencias entre layout XML y código Java.

Para ello, se utiliza el método `setOnClickListener` del objeto botón para definir la acción a realizar cuando el usuario haga clic sobre el botón.

Sintaxis

```
public void setOnClickListener (View.OnClickListener l)
```

Este método recibe como parámetro un objeto de tipo `View.onClickListener`. Es preciso crear uno e implementar su método abstracto `onClick`, donde se informará la acción a realizar.

Sintaxis

```
public void onClick (View v)
```

Ejemplo

A continuación aparece la declaración del evento a ejecutar:

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

    <Button android:id="@+id/button2_pulse"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerInParent="true"
        android:padding="50dp"
        android:textSize="20sp"
        android:text="@string/pulse" />

</RelativeLayout>
```

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

    setContentView(R.layout.button2);
    final Button btn =
      (Button) findViewById(R.id.button_pulse);
    btn.setOnClickListener(new View.OnClickListener() {
        public void onClick(View v) {
          btn.setText(R.string.pulse);
        }
    });
}
```

d. Otros widgets

Existen muchos otros componentes, más o menos sofisticados. Hay disponibles, por ejemplo, casillas de opción (`CheckBox`), botones de radio (`RadioButton`), listas desplegables (`Spinner`), barras de progreso (`ProgressBar`), imágenes (`ImageView`)…

En Android 3.0 (API 11) han aparecido nuevos componentes, en particular un calendario (`CalendarView`), una lista de elección en una ventana flotante (`ListPopupWindow`) y un selector de número (`NumberPicker`).

Veremos con detalle varios de estos componentes en los próximos capítulos, en particular en el capítulo dedicado a los Componentes principales de la aplicación, y veremos en el capítulo Construir interfaces complejas cómo crear nuestros propios componentes.

Introducción

Este capítulo presenta con detalle dos nociones que se abordaron en el capítulo anterior: el ciclo de vida de las actividades, que hemos evocado con el método `onCreate` de la clase `Activity`, y la noción de intención, con las etiquetas `<intent-filter>`, utilizadas en el manifiesto.

Comenzaremos estudiando las intenciones, que son el vínculo entre los distintos componentes de la aplicación en el sistema Android. Gracias a ellos, una aplicación Android puede, en particular, aprovechar otras aplicaciones instaladas en el sistema, incluso sin conocerlas. En efecto, las intenciones se resuelven dinámicamente en tiempo de ejecución del sistema.

A continuación, presentamos el ciclo de vida de una actividad, y los métodos disponibles para salvaguardar el estado de una actividad y restaurar una actividad destruida por el sistema: es, en efecto, el sistema el que gestiona el ciclo de vida de las aplicaciones (y, por tanto, de las actividades). En concreto, es el sistema quien decide cuándo se destruye una aplicación, especialmente en función de los recursos del sistema disponibles tales como el procesador, la memoria… o en función de la duración de la inactividad del usuario... Cualquier aplicación desarrollada de manera seria, debe tener en cuenta todo esto.

Intención

Una de las grandes ventajas de Android es que no es preciso vincular entre sí de manera estática los componentes de aplicación, volviéndolos así lo más independientes posible. Este principio se lleva tan lejos que permite incluso poder utilizar componentes de una aplicación desde otra aplicación sin conocerlos en tiempo de desarrollo. Se escogerán en tiempo de ejecución de la aplicación, bien por el sistema, bien por el usuario.

Para poder hacer esto y comunicar de manera dinámica componentes entre ellos, Android proporciona objetos de tipo `Intent` (intención). Como su propio nombre indica, una intención designa una acción a realizar. Pero una intención también puede describir un evento que acaba de producirse, como en el caso de los receptores de eventos (véase el capítulo Componentes principales de la aplicación - Receptor de eventos).

Un objeto de tipo `Intent` encapsula únicamente información, como por ejemplo un mensaje. Es tarea del componente origen y del sistema hacer uso del mismo para explotar la intención que encierra.

Android proporciona nombres genéricos para las acciones, que se pueden utilizar tal cual. Por ejemplo, `android.intent.action.DIAL` para iniciar una llamada telefónica.

El desarrollador puede, a su vez, crear sus propias acciones. Dado que las acciones pueden tener un alcance global en el sistema, es preciso nombrar las acciones creadas precediéndolas del nombre del paquete de la aplicación que las crea.

La acción de un objeto de tipo `Intent` puede especificarse y recuperarse utilizando los métodos`setAction` y `getAction`.

Sintaxis

```
public Intent setAction (String action)
public String getAction ()
```

El método `setAction` devuelve el mismo objeto de tipo `Intent` con el objetivo de poder encadenar llamadas entre distintos métodos.

Para realizarse, la acción especificada en la intención puede requerir datos. También puede necesitar información respecto a los datos que deben utilizarse o modificarse en la acción. Toda esta información puede proporcionarse al objeto de tipo `Intent` bajo la forma de categoría, de datos o de Extras.

La categoría se utiliza para determinar el tipo de componente que debe realizar la acción. Existen varias categorías por defecto. El desarrollador puede, incluso, definir nuevas categorías. Para agregar una categoría a una intención basta con invocar a su método `addCategory` pasándole como parámetro una cadena de caracteres que designe a la categoría.

Sintaxis

```
public Intent addCategory (String category)
```

Los datos reagrupan la URI y el tipo MIME del dato afectado. El formato y el sentido de estos datos son dependientes de la acción especificada en la intención. En general, el tipo MIME puede deducirse de la URI. He aquí los principales métodos disponibles para especificar y recuperar esta información:

Sintaxis

```
public Intent setData (Uri data)
public Uri getData ()
public Intent setType (String type)
public String getType ()
```

El uso de `setData` y el de `setType` son concurrentes. Borran sus datos respectivamente.

Los Extras son otros datos de aplicación que pueden incluirse en la intención utilizando el método`putExtras`, y el método `getExtras` para recuperarlos. Estos datos deben proporcionarse bajo la forma de un objeto de tipo `Bundle`.

La función de la clase `Bundle` es similar a la de la clase `Map`, es decir almacena los datos, asociados entre sí bajo la forma clave-valor. A una clave le corresponde un único valor que puede no obstante ser una tabla de valores del mismo tipo. La clase `Bundle` sólo acepta cadenas de caracteres como clave. Los valores deben implementar la interfaz `Parcelable`, mecanismo de serialización ligera específico de Android.

Preste atención, la interfaz `Parcelable` y el tipo `Parcel` asociado no deben utilizarse para serializar datos persistente, la representación interna de los datos puede modificarse con cada versión del SDK.

Sintaxis

```
public Intent putExtras (Bundle extras)
public Bundle getExtras ()
```

Para mayor comodidad, la clase `Intent` proporciona una multitud de métodos directos para especificar los datos extras sin tener que pasar explícitamente por el objeto `Bundle`.

Sintaxis

```
public Intent putExtra (String name, int value)
public int getIntExtra (String name, int defaultValue)
public Intent putExtra (String name, String value)
public Intent putExtra (String name, Parcelable value)
public String getStringExtra (String name)
...
```

Por último, el método `addFlags` permite agregar flags a un objeto de tipo `Intent`, como veremos más adelante.

Sintaxis

```
public Intent addFlags (int flags)
```

Existen dos formas de escribir una intención: de forma explícita o de forma implícita.

1. Intención explícita

Una intención explícita designa de forma precisa el componente concreto al que está destinada. De hecho, escoge también la acción que quiere realizar. La intención puede contener también la acción a realizar en el caso en que el componente pueda realizar varias.

Uno de los usos más comunes de las intenciones explícitas es la ejecución de componentes de aplicación, en particular actividades, en el interior de una misma aplicación.

El componente de destino se busca entre los componentes declarados en el manifiesto. Como consecuencia, si el componente no aparece en el archivo, no podrá ser invocado.

En concreto, para crear una intención explícita, es preciso crear un objeto de tipo `Intent` e indicarle la clase de destino. Para ello, una solución consiste en utilizar uno de los constructores de la

clase `Intent`. Éstos reciben como parámetro el contexto de la aplicación y el nombre de la clase de destino y devuelven el objeto de tipo `Intent` creado. Otro de los constructores permite, además, especificar la acción y los datos.

Sintaxis

```
public Intent (Context packageContext, Class<> cls)
public Intent (String action, Uri data, Context packageContext,
  Class<> cls)
```

Ejemplo

```
Intent intent = new Intent(this, MiActividadDestino.class);
```

2. Intención implícita

A diferencia de la intención explícita, que designa al componente de destino, la intención implícita indica, en sí misma, la acción a realizar. Encarga al sistema que encuentre el componente aplicativo de destino que mejor se adapte para realizar esta acción, ubicado por lo general en otra aplicación. Si existen varios componentes al mismo nivel, el sistema solicitará al usuario que seleccione uno. Si el sistema no encuentra ninguno, se generará una excepción.

Descubramos en primer lugar cómo crear una intención implícita y a continuación cómo declaran los componentes de aplicación las intenciones implícitas a las que pueden responder.

a. Creación

En particular, para crear una intención implícita, es preciso crear un objeto de tipo `Intent` e indicarle la acción a realizar. Para ello, una solución consiste en utilizar uno de los constructores de la clase `Intent` que reciben directamente la acción como parámetro. Otro de los constructores permite especificar, además, los datos asociados.

Sintaxis

```
public Intent (String action)
public Intent (String action, Uri uri)
```

Ejemplo

```
Intent intent = new Intent(Intent.ACTION_DIAL,
  Uri.parse("tel:654321987"));
```

El sistema buscará a continuación el mejor componente de entre los que hayan declarado la capacidad de realizar la acción solicitada y respondan a los criterios exigidos. Utilizará para ello los valores `action`, `category` y `data` del objeto `intent`.

b. Filtro de intención

Un filtro de intención permite a un componente de aplicación declarar al sistema el tipo de intención implícita a la que puede responder. Sólo aquellas intenciones implícitas que se tengan en cuenta en el filtro se procesarán en el componente afectado.

Los filtros de intención no tienen ninguna incidencia sobre las intenciones explícitas. Estas últimas no tienen nada que hacer con estos filtros y los pasan.

La declaración de las capacidades de un componente de aplicación se realiza utilizando la etiqueta`intent-filter` en el manifiesto. Un componente de aplicación puede tener varios filtros de intención diferentes. Si no tiene ninguno, el componente sólo puede ser ejecutado por

intenciones explícitas.

Cuando debe procesarse una intención implícita, el sistema analiza sus datos y los compara a los de los filtros de los componentes disponibles. El objeto de tipo `Intent` debe cumplir los criterios de uno de los filtros para poder ser procesado. Los criterios son la acción, la categoría y los datos.

De ahí que las etiquetas `intent-filter` puedan incluir etiquetas `action`, `category` y `data`para describir cada uno de estos puntos.

Sintaxis

```
<intent-filter android:icon="recurso_gráfico"
               android:label="recurso_texto"
               android:priority="entero" >
    ...
</intent-filter>
```

Los atributos `android:icon` y `android:label` pueden mostrarse al usuario según las necesidades. Si no se especifican, utilizarán por defecto las del componente padre.

Si una intención se corresponde con varios filtros de actividades o de receptores de eventos distintos, el atributo `android:priority` permite escoger aquél que utilizará el sistema. El valor de este atributo debe ser un número entero. Se dará prioridad al de mayor valor.

La etiqueta `action` de un filtro de intención indica el nombre completo de la acción que el componente puede procesar mediante el atributo `android:name`. El nombre completo está formado por el nombre del paquete de la aplicación seguido del nombre de la acción en mayúsculas separado por un punto. Es posible indicar varias acciones en la misma etiqueta`intent-filter`.

Basta con que la acción del objeto intent se corresponda con una sola de las acciones del filtro para que se cumpla el criterio de la acción.

Sintaxis

```
<action android:name="cadena de caracteres" />
```

Ejemplo

```
<intent-filter>
  <action android:name="android.intent.action.VIEW" />
  <action android:name="es.midominio.android.miaplicacion.ACCION1" />
</intent-filter>
```

La etiqueta `category` de un filtro de intención indica el nombre completo de la categoría que caracteriza al componente. El nombre completo está formado por el nombre del paquete de la aplicación seguido del nombre de la categoría en mayúsculas separado por un punto. Es posible indicar varias categorías en el mismo filtro de intención.

Es preciso que todas las categorías del objeto intent estén incluidas en el mismo filtro para que se cumpla el criterio de la categoría. Poco importa si el filtro aporta más categorías que las solicitadas por el intent.

Sintaxis

```
<category android:name="cadena de caracteres" />
```

Ejemplo

```
<intent-filter ...>
  <category android:name="android.intent.category.DEFAULT" />
  <category
    android:name="es.midominio.android.miaplicacion.CATEGORIA1"/>
</intent-filter>
```

La etiqueta `data` de un filtro de intención permite especificar una URI y/o un tipo MIME. Los atributos de esta etiqueta son numerosos y, para algunos de ellos, dependientes los unos de los otros. De hecho, descubriremos el uso de esta etiqueta y de algunos de sus atributos conforme avancemos en este libro.

Los datos, data y tipo MIME, indicados por la intención y aquellos del filtro deben corresponderse perfectamente para que se cumpla el criterio de los datos.

Sintaxis

```
<data android:host="cadena de caracteres"
      android:mimeType="cadena de caracteres"
      android:path="cadena de caracteres"
      android:pathPattern="cadena de caracteres"
      android:pathPrefix="cadena de caracteres"
      android:port="cadena de caracteres"
      android:scheme="cadena de caracteres" />
```

Los valores de los atributos `android:host`, `android:mimeType` y `android:scheme`deben escribirse en minúsculas.

Android considera que si sólo se indica el tipo MIME entonces el filtro, y por tanto el componente, acepta los datos `content:` y `file:` que descubriremos más adelante.

Ejemplo

```
<intent-filter>
  <data android:mimeType="video/*" />
</intent-filter>
```

3. Intención pendiente

Una intención pendiente indica un objeto de tipo `PendingIntent`. Ésta contiene un objeto de tipo`Intent` que describe una acción a realizar. Esta acción puede realizarse en una fecha posterior por otra aplicación haciéndose pasar por la aplicación que ha creado el objeto de tipo `PendingIntent`y recibiendo los permisos para la ocasión.

Los métodos estáticos `getActivity`, `getService` y `getBroadcast` de la clase`PendingIntent` crean un objeto de tipo `PendingIntent` permitiendo respectivamente ejecutar una actividad, un servicio o difundir un evento. Estos métodos reciben como parámetro el contexto de la aplicación, un código no utilizado, la intención a realizar así como los flags.

Sintaxis

```
public static PendingIntent getActivity (Context context, int
requestCode, Intent intent, int flags)
public static PendingIntent getService (Context context, int
requestCode, Intent intent, int flags)
public static PendingIntent getBroadcast (Context context, int
```

```
requestCode, Intent intent, int flags)
```

Ejemplo

```
Intent intent = new Intent(this, MiActividadDestino.class);
intent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
PendingIntent pendingIntent = PendingIntent.getActivity(this, 0,
  intent, PendingIntent.FLAG_UPDATE_CURRENT);
```

Este ejemplo crea, o actualiza (mediante el flag `PendingIntent.FLAG_UPDATE_CURRENT`), un objeto de tipo `PendingIntent` encargado de ejecutar la actividad `MiActividadDestino` en una fecha posterior.

El hecho de agregar el flag `Intent.FLAG_ACTIVITY_NEW_TASK` es obligatorio para arrancar la actividad en una nueva tarea (véase la sección Actividad).

Actividad

Hemos visto en el capítulo Descubrir la interfaz de usuario que una actividad es un componente independiente que tiene la mayoría de veces una interfaz de usuario. En el caso más representativo, una actividad presenta una pantalla al usuario para interactuar con él. Una aplicación muy simple puede tener una única actividad mientras que una más compleja puede poseer varias. No obstante, solamente una única actividad de la aplicación estará activa a la vez, y la pantalla que la gestiona es la que se muestra al usuario.

Recordemos que para definir una actividad, es preciso crear una clase que herede de la clase`Activity` e implementar, como mínimo, los métodos heredados.

La ejecución de una actividad se produce en el proceso ligero, o thread, principal del proceso de la aplicación. Este thread también se denomina thread de la interfaz de usuario puesto que permite modificar la interfaz de usuario (UIThread). Cualquier modificación en la interfaz desde un thread concurrente genera un error.

Para preservar la experiencia de usuario, una actividad no debería bloquear su thread principal más de algunos segundos (véase el capítulo Concurrencia, seguridad y red - Programación concurrente).

1. Declaración

Para poder utilizarse, una actividad debe declararse en el sistema mediante el manifiesto.

La etiqueta `activity` contiene información propia de una actividad. Como la mayoría de las etiquetas, esta etiqueta contiene muchos atributos. Veremos algunos de ellos conforme descubramos las funcionalidades asociadas.

Una actividad que acepta recibir intenciones implícitas debe declarar obligatoriamente lacategoría `android.intent.category.DEFAULT` en su filtro de intención. Esta categoría se agrega automáticamente al objeto intent cuando se pasa este objeto al método`startActivity` o al método `startActivityForResult`.

Recordemos la sintaxis de esta etiqueta y sus tres atributos principales:

Sintaxis

```
<activity android:icon="recurso gráfico"
          android:label="recurso texto"
          android:name="cadena de caracteres"
          ... >
    ...
</activity>
```

Los atributos `icon` y `label` tienen las mismas funciones que en la etiqueta `application` pero limitadas a la actividad. Si no se especifican, se utilizarán por defecto las de la aplicación.

Observe que el atributo `android:screenOrientation` permite a la actividad indicar la orientación que debe adoptar. Por defecto, la orientación de una actividad depende de cómo el usuario sostenga el dispositivo. Es posible obviar estos cambios de orientación y fijar de manera permanente la orientación de la actividad utilizando por ejemplo el valor `portrait`, `landscape`o `nosensor` correspondiente respectivamente a vertical, apaisado, y el uno o el otro en función de la elección del sistema según el dispositivo. No obstante, se recomienda que una aplicación no lo suponga y fije su orientación. Esto es todavía más evidente si debe funcionar en dispositivos de distintos tipos tales como smartphones y tabletas táctiles, pues los usuarios tienen tendencia a utilizar los primeros principalmente en modo vertical y los segundos en modo apaisado.

Desde Android 3.0 (API 11), es posible demandar que la visualización de las vistas de una actividad o de una aplicación completa utilice la aceleración de hardware si está disponible en el dispositivo utilizado. Esto permite mejorar el rendimiento de la representación de las vistas, las animaciones y, en general, la reactividad del sistema. De este modo, la experiencia del usuario se verá mejorada. Para ello, el desarrollador debe utilizar el atributo`android:hardwareAccelerated` bien en la etiqueta `application` o bien en una o varias etiquetas `activity` del manifiesto según el uso deseado.

El atributo `android:name` permite especificar el nombre de la actividad concreta, es decir su nombre de clase Java precedido por el nombre completo del paquete.

Ejemplo

Para acortarlo, dado que

```
android:name="es.midominio.android.miaplicacion.MiActividadPrincipal"
```

el nombre del paquete se conoce, pues viene especificado por el atributo`package` de la etiqueta `manifest`, es posible indicar el nombre de la clase de la actividad reemplazando el paquete por un punto.

Ejemplo

```
android:name=".MiActividadPrincipal"
```

Ejemplo

En este ejemplo, y a diferencia de lo que se indica en la observación situada más arriba, el filtro de la actividad no declara

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

la categoría `android.intent.category.DEFAULT`. Es el único caso en el que no es obligatorio declarar de forma combinada de la acción`android.intent.action.MAIN` y de la categoría`android.intent.category.LAUNCHER`.

Recuerde, la acción `android.intent.action.MAIN` indica que el componente es el punto de entrada de la aplicación. La categoría `android.intent.category.LAUNCHER` indica además que el compo-nente forma parte de los componentes que puede ejecutar el usuario. La combinación de ambas etiquetas permite que este componente, en este caso una actividad, se agregue a la aplicación **Lanzador de aplicaciones** del dispositivo Android.

La aplicación **Lanzador de aplicaciones** es la aplicación que muestra, en forma de parrilla de iconos con su nombre, la lista de las aplicaciones que puede ejecutar el usuario.

2. Ciclo de vida

El ciclo de vida de una actividad describe los estados en los que la actividad puede encontrarse entre su creación, la instanciación, y su muerte (destrucción de esta instancia).

Cada cambio de estado que se produce invoca a un método específico que puede sobrecargarse en la clase de la actividad.

Cada uno de estos métodos debe invocar a su método padre, en caso contrario se generará una excepción.

El sistema no puede destruir una actividad mientras se encuentre en un estado anterior al estado de pausa. En pausa y más allá, el sistema puede decidir destruir la actividad en cualquier momento.

a. onCreate

El método `onCreate` es el primer método del ciclo de vida que se invoca durante la creación de una actividad. Sólo se invoca una única vez durante todo el ciclo de vida de la actividad. La función de este método es permitir inicializar la actividad, crear las vistas que deben mostrarse tras la creación de la actividad, recuperar las instancias de estas vistas…

Si se produce una llamada al método `finish` dentro de este método para poner fin directamente a la actividad sin que haya podido mostrarse en ningún momento, entonces elmétodo `onDestroy` se invocará inmediatamente después, cortocircuitando la secuencia de llamadas normal: `onStart`, `onResume`...

Recibe como parámetro un objeto de tipo `Bundle`. Éste permite recuperar información, los datos salvaguardados anteriormente tras la última ejecución de esta actividad. Se describe con más detalle más adelante.

Conviene destacar que este parámetro es nulo durante la primera creación de esta actividad.

Sintaxis

```
protected void onCreate (Bundle savedInstanceState)
```

Ejemplo

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

En este ejemplo, el método invoca a su método padre pasándole el objeto de tipo `Bundle` e invoca al

```
    Intent intent = getIntent();
    traitement(intent);
  }
}
```

método `setContentView` para crear la vista inicial de la actividad que se mostrará al usuario. El método `getIntent` permite recuperar la intención que ha solicitado la ejecución de esta actividad.

b. onStart

El método `onStart` es el simétrico del método `onStop`. Se invoca tras el método `onCreate` o tras el método `onRestart` si la actividad se encontraba en el estado detenido y ha vuelto a primer plano.

Precede a la visualización de la actividad, en concreto de su vista.

Sintaxis

```
protected void onStart ()
```

Ejemplo

```
public class MiActividad extends Activity {
  @Override
  public void onStart() {
    super.onStart();
  }
}
```

c. onResume

El método `onResume` es el simétrico del método `onPause`. Se invoca tras el método `onStart` o tras el método `onPause` si la actividad estaba en pausa y ha vuelto a primer plano.

En este estado, la actividad se muestra al usuario. Una vez que se sale de este método, la actividad va a funcionar con normalidad en primer plano y podrá interactuar con el usuario.

Sintaxis

```
protected void onResume ()
```

Ejemplo

```
public class MiActividad extends Activity {
  @Override
  public void onResume() {
    super.onResume();
  }
}
```

d. onPause

El método `onPause` es el simétrico del método `onResume`. Se invoca justo antes de que otra actividad tome el control y pase a primer plano. Es preciso que sea a la vez un método rápido y eficaz de cara a no bloquear a la actividad siguiente, con la función de salvaguardar los datos persistentes de la actividad y parar las tareas que consuman CPU, memoria...

Hasta Android 3.0 (API 11), el sistema Android puede decidir matar la aplicación, en concreto su proceso, a partir de cualquier momento desde la salida de este método. Es el único método del que podemos estar seguros que se invoca antes de suprimir el proceso. Por tanto, es la última ocasión que tenemos de realizar las copias de seguridad y demás acciones importantes que deben realizarse obligatoriamente. Desde Android 3.0 (API 11), es el métodoonStop el que cubre este rol.

Puede ser útil utilizar el método `isFinishing` para determinar si la actividad está tratando de terminar o si simplemente se está poniendo en pausa. En el primer caso, devolverá `true`, en caso contrario devolverá `false`.

Sintaxis

```
public boolean isFinishing ()
```

Sintaxis del método `onPause`

```
protected void onPause ()
```

Ejemplo

```
public class MiActividad extends Activity {
  @Override
  public void onPause() {
    super.onPause();
    if (isFinishing()) {
      procesamiento1();
    } else {
      procesamiento2();
    }
  }
}
```

e. onStop

El método `onStop` es el simétrico del método `onStart`.

Este método permite liberar ciertos recursos, en particular los objetos de tipo `Cursor` queveremos más adelante (véase el capítulo La persistencia de los datos - Bases de datos SQLite).

Hasta Android 3.0 (API 11), este método puede que jamás se invoque si el sistema decide antes terminar con el proceso de la aplicación. Es preciso utilizar en su lugar el método`onPause` para salvaguardar los datos persistentes.

Desde Android 3.0 (API 11), el sistema Android no puede matar el proceso de la aplicación antes de invocar a este método. Por el contrario, puede decidir terminar con la aplicación a partir de cualquier momento desde la salida de este método. Por tanto, es la última ocasión que tenemos de realizar las copias de seguridad y demás acciones importantes que deban realizarse obligatoriamente.

Sintaxis

```
protected void onStop ()
```

Ejemplo

```
public class MiActividad extends Activity {
  @Override
  public void onStop() {
    super.onStop();
  }
}
```

f. onRestart

El método `onRestart` puede invocarse desde el método `onStop` si la actividad vuelve a primer plano. A continuación continúa con una llamada al método `onStart`.

Este método permite solicitar de nuevo ciertos recursos liberados en el método `onStop`.

Sintaxis

```
protected void onRestart ()
```

Ejemplo

```
public class MiActividad extends Activity {
  @Override
  public void onRestart() {
    super.onRestart();
  }
}
```

g. onDestroy

El método `onDestroy` es el simétrico del método `onCreate`. Se invoca después de invocar almétodo `finish` o directamente por el sistema si necesita liberar recursos. Es el último método invocado que se pone a disposición del desarrollador antes de destruir efectiva e irreversiblemente la actividad.

Este método permite liberar recursos ligados a la actividad, por ejemplo un thread que no tiene sentido sin esta actividad.

Este método puede que jamás se invoque si el sistema ha decidido matar antes el proceso de la aplicación. Es preciso utilizar en su lugar el método `onPause` para salvaguardar los datos persistentes.

Sintaxis

```
protected void onDestroy ()
```

Ejemplo

```
public class MiActividad extends Activity {
  @Override
  public void onDestroy() {
    supprimeThread();
    super.onDestroy();
  }
}
```

3. Ejecución

El método `startActivity` permite ejecutar una actividad pasándole como parámetro un objeto de tipo `Intent` (intención), implícita o explícita.

Sintaxis

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

Ejemplo

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

Si la actividad ejecutada debe devolver un resultado una vez haya finalizado, es preciso utilizar losmétodos `startActivityForResult` y `onActivityResult` que funcionan de forma conjunta. Una vez que la actividad de destino termina, se invoca el método `onActivityResult` del componente origen. Se le pasan los siguientes parámetros: el código informado en la llamada`startActivityForResult` que permite identificar el origen del resultado, el resultado de la actividad que acaba de terminar y los datos suplementarios que proporciona la actividad que acaba de terminar en los campos `extra` de un objeto de tipo `intent`.

La ejecución del método `onActivityResult` tiene lugar justo antes del método `onResume`de la actividad origen.

Sintaxis

```
public void startActivityForResult (Intent intent, int requestCode)
protected void onActivityResult (int requestCode, int resultCode,
  Intent data)
```

Ejemplo

```
public class MiActividadPrincipal extends Activity {
  private static final int REQ_MSG_ACTIVIDADDEST = 1;

  private void ejecutaActividadDestino() {
    Intent intent =
      new Intent(this, MiActividadDestino.class);
      startActivityForResult(intent, REQ_MSG_ACTIVIDADDEST);
  }

  @Override
  protected void onActivityResult(int requestCode,
      int resultCode, Intent data) {
    if (requestCode == REQ_MSG_ACTIVIDADDEST) {
      if (resultCode == RESULT_OK) {
        Bundle bundle = data.getExtras();
        procesamiento(bundle);
      }
    } else
      super.onActivityResult(requestCode, resultCode, data);
  }
}
```

La actividad de destino indica el resultado devuelto y, si es preciso, el objeto `intent` lo acompaña utilizando el método `setResult`. Este método modifica precisamente el valor de retorno, borrando el valor anterior llegado el caso. No termina la actividad en curso. Se utilizará el método `finish`para ello.

Sintaxis

```
public final void setResul
public final void setResul
```

Ejemplo

```
public class MiActividadSecundaria extends Activity {

  private void devuelveCancel() {
    setResult(RESULT_CANCELED);
    finish();
  }

  private void devuelveOK() {
```

```
    Intent intent = new Intent();
    intent.putExtra("clave", "valor");
    setResult(RESULT_OK, intent);
    finish();
  }
}
```

4. Salvaguarda y restauración del estado

Como vimos en la descripción del método `onCreate`, un objeto de tipo `Bundle` permite salvaguardar el estado de la actividad cuando ésta es destruida por el sistema (y únicamente en este caso) para poder recrearla a continuación con idéntico estado.

El usuario que cierra voluntariamente una actividad, por ejemplo presionando la tecla **Volver** del dispositivo, no provocará la copia de seguridad de este objeto de tipo `Bundle`, que se realizará por el contrario cuando el sistema necesite recursos o, con mayor frecuencia, cuando el usuario cambie la orientación del dispositivo.

El sistema destruye entonces la actividad y sus vistas en curso para ser recreadas en un formato compatible con la nueva orientación. Para ello, el SDK proporciona dos métodos complementarios:`onSaveInstanceState` y `onRestoreInstanceState`.

El método `onSaveInstanceState` permite salvaguardar los datos en el objeto de tipo `Bundle`que se pasa como parámetro. Este será el objeto pasado, llegado el momento, al método`onCreate` y al método `onRestoreInstanceState`. Este método se invoca antes del método`onStop`. Esto puede ser antes o después del método `onPause` pero, en cualquier caso, la actividad no es destruida por el sistema mientras no se haya invocado a este método.

Existen casos en los que se invoca el método `onPause` mientras que el método`onSaveInstanceState` no. Es el caso descrito anteriormente en el que el usuario cierra la actividad en curso. Hay que prestar atención y utilizar el método `onSaveInstanceState` en lugar del método `onPause`, en particular para salvaguardar los datos persistentes.

El método `onRestoreInstanceState` recibe como parámetro el objeto de tipo `Bundle`salvaguardado anteriormente. Gracias a esta información, la actividad puede restaurar sus datos y sus vistas en el estado en que estuvieran anteriormente. También es posible hacerlo en el método`onCreate`. El método `onRestoreInstanceState` se invoca después del método `onStart` y antes que el método `onResume`.

Sintaxis

```
protected void onSaveInstanceState (Bundle outState)
protected void onRestoreInstanceState (Bundle savedInstanceState)
```

Ejemplo

```
public class MiActividad extends Activity {
  private static final String CLAVE_1 =
    "es.midominio.android.miaplicacion.clave1";
  private static final String CLAVE_2 =
    "es.midominio.android.miaplicacion.clave2";

  private String mValor1;
  private int mValor2;

  @Override
  protected void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState);
    outState.putString(CLAVE_1, mValor1);
    outState.putInt(CLAVE_2, mValor2);
  };

  @Override
  protected void onRestoreInstanceState(Bundle savedState) {
    mValor1 = savedState.getString(CLAVE_1);
    mValor2 = savedState.getInt(CLAVE_2);
    super.onRestoreInstanceState(savedState);
  };
}
```

5. Pila de actividades

A cada aplicación el sistema le asocia una pila de actividades de tipo FIFO (*First In, First Out*) o el primero en entrar es el primero en salir. Ésta apila las actividades de la aplicación que se ejecutan las unas a continuación de las otras. Esto aplica también a las actividades ejecutadas desde la aplicación pero proporcionadas por otras aplicaciones, como veremos en los próximos capítulos.

Esta secuencia de actividades almacenadas en la pila representa una tarea, en el sentido funcional.

Cuando el usuario desea volver a la actividad anterior a la actividad en curso apretando por ejemplo el botón **Volver**, el sistema desapila la actividad en curso, la destruye y retoma la ejecución de la actividad anterior. La operación puede repetirse hasta vaciar por completo la pila, que se destruye a continuación.

Cuando el usuario vuelve a la pantalla de bienvenida de Android, el escritorio, presionando la tecla**Inicio**, el sistema salvaguarda en memoria la pila de la aplicación en curso. Si el usuario vuelve a ejecutar la aplicación, por defecto, se restaurará su pila y la actividad en curso estará situada en la parte superior de la pila.

- Si se estima la necesidad, el sistema Android puede destruir en cualquier momento las actividades que no estén en curso para liberar recursos del sistema. Por ello se aconseja encarecidamente salvaguardar el estado de las aplicaciones antes de su destrucción de modo que el sistema pueda restaurarlas en el estado en el que las hubiera dejado el usuario.

- Por defecto, el sistema puede vaciar una pila que no se esté usando tras cierto lapso de tiempo y así guardar solamente la primera actividad. El sistema considera en efecto que el usuario no quiere retomar una acción anterior sino comenzar una nueva. Es posible modificar este comportamiento haciendo uso de los atributos `android:alwaysRetainTaskState`,`android:clearTaskOnLaunch` y `android:finishOnTaskLaunch` de la etiqueta activity de la actividad de entrada en el manifiesto.

Como veremos más adelante con la barra de acción (véase el capítulo Completar la interfaz de usuario - Barra de acción), el usuario puede presionar el icono de la aplicación de esta barra para volver justo a la actividad anterior. Esto se realiza fácilmente utilizando el método `finish` que hemos visto.

No obstante el hecho de presionar el icono de la aplicación en esta barra de acción también puede indicar que el usuario quiere volver a la actividad de entrada. En el estado actual, si la actividad de entrada se ejecuta de nuevo usando un intent, se situará en la parte superior de la pila de actividades. El usuario no comprenderá que si vuelve atrás para salir de la aplicación, se encuentre con una actividad antigua en su lugar... La acción correcta consiste en hacer que todas las actividades salvo la primera se desapilen de la pila. La única actividad que queda, la primera, que se corresponde por lo general con la actividad de entrada, será la actividad en curso.

Para hacer esto posible de manera sencilla, Android permite modificar el funcionamiento por defecto de la pila.

Por ejemplo, basta con agregar el flag `Intent.FLAG_ACTIVITY_CLEAR_TOP` al objeto `intent`utilizado para ejecutar la actividad que permite desapilar las actividades de la pila hasta aquella afectada por el intent. En nuestro caso, se trata de la actividad de entrada. Esto provoca que todas las actividades desde la parte superior de la pila hasta la actividad de entrada se desapilen.

Ejemplo

```
Intent intent = new Intent(this, MiActividadPrincipal.class);
intent.addFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP);
startActivity(intent);
```

- Existen muchos otros flags de tipo `Intent.FLAG_ACTIVITY_*` que permiten modificar el comportamiento por defecto.

- Android permite a su vez a las actividades especificar su modo de agregación a la pila directamente en el manifiesto. Para ello, se puede utilizar el atributo `android:launchMode`de la etiqueta `activity`. Llegado el momento, los flags especificados en los intents serán prioritarios a los definidos en el manifiesto.

Introducción

La presentación de la interfaz de usuario nos ha permitido abordar los temas generales y descubrir los widgets. No obstante, la interfaz de usuario de una aplicación no se limita a esto. Existen otros elementos que complementan a los ya expuestos anteriormente y que aportan nuevas funcionalidades a las aplicaciones.

En este capítulo, descubriremos cómo utilizar los nuevos estilos y temas incluidos en Android 3.0, cómo crear un menú o una barra de acción según el sistema utilizado y veremos los distintos métodos para notificar al usuario. Por último, terminaremos describiendo cómo programar una aplicación multiidioma.

Estilos y temas

De forma similar a las hojas de estilo CSS (*Cascading Style Sheets*) usadas en las páginas Web, los estilos en Android permite separar las propiedades de diseño de una vista de su contenido. Esta separación permite tener un código más claro para las vistas, una reutilización de estilos y facilita enormemente la construcción de interfaces homogéneas.

Un estilo es un recurso definido en un archivo XML de la carpeta `res/values`. Es habitual reagrupar todos los estilos en un único archivo llamado `styles.xml`. Los estilos se definen utilizando laetiqueta `style`. Ésta permite darle nombre al estilo y, eventualmente, especificar el estilo padre de propiedades, del que hereda. A continuación se definen los distintos elementos que componen el estilo utilizando etiquetas `item`.

Los estilos se aplican únicamente sobre una vista individual utilizando el atributo `style` de la etiqueta vista. El valor de este atributo designa el estilo deseado.

Por ello si queremos aplicar un estilo a toda la aplicación, resulta algo molesto asignar el estilo a todas las vistas de la aplicación. Para evitar tener que hacer esta manipulación, Android permite usar temas.

Los temas no son más que estilos aplicados a actividades o a la aplicación entera.

Si una aplicación no define su propio tema utilizando el atributo `android:theme` en su etiqueta`application` del manifiesto, utiliza en ese caso el tema del sistema Android sobre el que se ejecute.

Un nuevo tema gráfico, llamado holográfico (o de forma abreviada holo), ha hecho aparición con Android 3.0 (API 11). Éste introduce en concreto un nuevo elemento de la interfaz: una barra de acción (véase la sección Barra de acción).

Cualquier aplicación que quiera utilizar el nuevo tema holográfico cuando se ejecute sobre un sistema con versión 3.0 o superior debe especificar un nivel de API igual o superior a 11 en el atributo`android:targetSdkVersion` de la etiqueta `uses-sdk` del manifiesto.

Esto sólo es necesario en el caso de una aplicación hecha para sistemas Android 3.0 o superiores, es decir cuando el valor de `android:minSdkVersion` es igual o superior a 11.

De este modo, si la aplicación se ejecuta sobre un sistema Android de versión inferior a 3.0 (API 11), utilizará el tema del sistema de esta versión. Y si la aplicación se ejecuta sobre un sistema 3.0 (API 11) o superior, utilizará el nuevo tema holográfico.

Recuerde, las API que puede utilizar una aplicación son aquellas que existen en la versión especificada por el atributo `android:minSdkVersion`. De ello se desprende que si el valor del atributo `android:minSdkVersion` es inferior a 11, entonces la aplicación no podrá utilizar los nuevos elementos tales como la barra de acción dado que en estas versiones no existen las clases correspondientes a estos elementos.

Menús

Elementos importantes de la interfaz de usuario, en Android hay disponibles dos tipos de menús: los menús de actividad y los menús contextuales. Su declaración se realiza de forma idéntica.

1. Declaración

Recuerde, daremos prioridad aquí al método declarativo si bien es posible crear o modificar los menús directamente desde el código Java.

La primera etapa consiste en crear un archivo de formato XML en la carpeta `res/menu` del proyecto e incluir en él la etiqueta `menu`.

Sintaxis

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    ...
</menu>
```

Ejemplo

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

Las opciones del menú - o entradas - se agregan, a continuación, una a una utilizando la etiqueta `item`. Sus atributos permiten parametrizar la opción. He aquí las principales:

Propiedad	Descripción
`android:id`	Identificador del elemento.
`android:title`	Título de la opción a mostrar.
`android:icon`	Imagen con el icono asociado a la opción.
`android:onClick`	Nombre del método de la actividad a ejecutar cuando el usuario hace clic en la vista. Existe desde Android 3.0 (API 11).
`android:showAsAction`	Permite especificar si esta opción debe figurar en la barra de acción y de qué forma. Existe desde Android 3.0 (API 11).
`android:checked`	Permite preseleccionar la opción cuando forma parte de un grupo de opciones a marcar o de botones radio.

Sintaxis

```
<item android:id="@[+][paquete:]id/nombre_recurso"
  android:icon="recurso_gráfico"
  android:title="recurso_texto"
  android:onClick="nombre_método"
  android:showAsAction="[always|ifRoom|never][|][withText]"
  android:checked="booleano"
  ...
/>
```

Ejemplo

```
<xml version="1.0" encoding="utf-8">
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item android:id="@+id/menú_principal_opción1"
        android:icon="@drawable/menú_principal_opción1"
        android:title="@string/menú_principal_opción1"
        android:onClick="procesamientoOpción1" />
</menu>
```

Es posible crear un nivel suplementario de submenús. Se trata de menús que se muestran cuando el usuario ha seleccionado una opción en el primer menú.

Esto permite crear una arborescencia de menús con dos niveles como máximo. No es posible tener submenús de submenús.

Para crear un submenú, basta con insertar un nuevo menú en un ítem del primer menú utilizando de nuevo la etiqueta menu. Los ítems del submenú deben informarse dentro de esta nueva etiqueta`menu`.

Sintaxis

```
<item ...>
  <menu>
    <item ...>
    ...
  </menu>
</item>
```

Ejemplo

```
<xml version="1.0" encoding="utf-8">
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item android:id="@+id/menú_principal_opción1"
        android:icon="@drawable/menú_principal_opción1"
        android:title="@string/menú_principal_opción1">
    <menu>
      <item android:id="@+id/menú_secundario_opción1"
            android:icon="@drawable/menú_secundario_opción1"
            android:title="@string/menú_secundario_opción1" />
    </menu>
  </item>
</menu>
```

Es posible agrupar parte de las opciones de un menú para aplicarles las mismas propiedades. También es posible agregar casillas para marcar o botones de radio a todos los elementos de un menú. Para estos dos casos, las opciones deben estar agrupadas en la misma etiqueta `group`. Si fuera necesario, el atributo `android:checkableBehavior` permite especificar el tipo de agrupación.

Sintaxis

```
<group android:id="@[+][paquete:]id/nombre_recurso"
  android:checkableBehavior="all|none|single" >
  ...
</group>
```

Ejemplo

```
<xml version="1.0" encoding="utf-8">
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <group android:id="@+id/group"
         android:checkableBehavior="single">
    <item android:id="@+id/menú_contextual_opción1"
```

```
        android:icon="@drawable/icon"
        android:title="@string/menú_config"
        android:checked="true" />
    <item android:id="@+id/menú_contextual_opción2"
        android:icon="@drawable/icon"
        android:title="@string/menú_config" />
  </group>
</menu>
```

2. Uso

La transformación de la descripción del menú desde el formato XML en una instancia del objeto de tipo `Menu` se realiza desde la actividad asociada utilizando un objeto de tipo `MenuInflater`.

Éste es devuelto por el método `getMenuInflater` de la actividad.

Sintaxis

```
public MenuInflater getMenuInflater ()
```

Ejemplo

```
MenuInflater conversorMenu = getMenuInflater();
```

El método `inflate` de este objeto permite leer el archivo de menú XML pasado como parámetro mediante su identificador y agregar los elementos y submenús al objeto de tipo `Menu` pasado como parámetro.

Sintaxis

```
public void inflate (int menuRes, Menu menu)
```

Ejemplo

```
Menu menu = new Menu();
conversorMenu.inflate(R.menu.principal, menu);
```

3. Menú de actividad

El menú de actividad es un menú que permite escoger en función de la pantalla mostrada actualmente. Este menú puede contener iconos, pero no puede contener casillas de opción a marcar ni botones de radio.

Para las aplicaciones que corran sobre una versión inferior a Android 3.0 (API 11), cuando el usuario presione la tecla **Menú** de su dispositivo, éste aparecerá. O, más bien, aparecerán los seis primeros elementos del menú. Si existen más de seis, el sexto elemento se reemplaza por la opción **Más**. Si el usuario la selecciona, aparece un nuevo menú con forma de lista y muestra el resto de elementos del menú, desde el sexto hasta el último.

Este menú de actividad desaparece cuando el usuario selecciona una opción o cuando presiona la tecla **Volver**.

Para las aplicaciones que corran sobre una versión de Android 3.0 (API 11) o superior, las opciones del menú se reagrupan por defecto en el submenú de la barra de acción (véase la sección Barra de acción) representadas por un icono de menú situado en la parte derecha de la barra de acción. Su presencia y su apariencia pueden modificarse mediante su atributo `android: showAsAction`.

a. Creación

Dado que este tipo de menú está asociado a una actividad, debe estar definido por la actividad en su método `onCreateOptionsMenu`. Este método recibe como parámetro el objeto de

tipo `Menu`que debe construir. Devuelve el valor `true` para permitir la visualización de este menú.

Sintaxis

```
public boolean onCreateOptionsMenu (Menu menu)
```

Ejemplo

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

b. Uso

Desde Android 3.0 (API 11), el atributo `android:onClick` de la etiqueta `item` permiteespecificar directamente el nombre de la actividad que se quiere ejecutar cuando el usuario selecciona la opción correspondiente. Este método debe estar definido en la actividad afectada, estar declarado como público y recibir como parámetro el objeto de tipo `MenuItem` seleccionado.

Sintaxis

```
public void nombreMétodo(MenuItem item)
```

Ejemplo

```
public void onMenuOption1(MenuItem item) {
  procesamiento1();
}
```

El objeto de tipo `MenuItem` contiene información del elemento seleccionado, recuperable mediante sus distintos accesores. Uno de ellos, el método `getItemId`, permite recuperar el identificador de la opción seleccionada por el usuario.

Sintaxis

```
public int getItemId ()
```

Ejemplo

```
int id = item.getItemId();
```

Existe otra forma de responder a la selección del usuario de una opción de menú y que es la única disponible en las versiones inferiores a Android 3.0 (API 11). Consiste en implementar el método`onOptionsItemSelected` en la actividad o el fragmento asociado. Recibe como parámetro el objeto de tipo `MenuItem` seleccionado. Debe devolver `true` si procesa la opción seleccionada, o`false` en caso contrario.

Sintaxis

```
public boolean onOptionsItemSelected (MenuItem item)
```

Ejemplo

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
  switch (item.getItemId()) {
  case R.id.menu_option1 :
```

```
    procesamiento1();
    return true;
  }
  return super.onOptionsItemSelected(item);
}
```

En este ejemplo, al finalizar el método `onOptionsItemSelected`, éste invoca a su método padre en el caso en el que este último haya definido opciones de menú suplementarias. Es el caso, por ejemplo, de actividades que deseen compartir menús comunes. Heredan de una misma clase madre que especifica las opciones de menú comunes y las acciones correspondientes.

Para modificar dinámicamente el menú, una vez creado, hay que sobrecargar el método`onPrepareOptionsMenu`. Desde Android 3.0 (API 11), es preciso también invocar de forma explícita al método `invalidateOptionsMenu` para forzar la reconstrucción del menú.

El capítulo Redes sociales, en la sección Integración estándar, propone otro método de gestión de elementos de menú que utiliza los atributos ActionProvider.

4. Menú contextual

Un menú contextual es un menú que depende del contexto en el que se activa. El usuario activa este menú presionando durante cierto tiempo sobre una vista. Si la vista ha declarado un menú contextual, éste aparecerá bajo la forma de una ventana en primer plano. Un menú así no puede contener iconos.

Las opciones de menú se muestran en forma de lista en un pop-up. Este menú contextual desaparece cuando el usuario selecciona una opción o presiona sobre la tecla **Volver**.

a. Creación

En primer lugar, hay que permitir a la vista gestionar cuando el usuario presiona durante un tiempo sobre la pantalla. Para ello es preciso utilizar el método `registerForContextMenu`pasándole como parámetro la vista.

Sintaxis

```
public void registerForContextMenu (View view)
```

Ejemplo

```
View vista = findViewById(R.id.layout_principal);
registerForContextMenu(vista);
```

Un menú contextual debe definirlo la actividad de la vista afectada en el método`onCreateContextMenu`. Ésta recibe como parámetro el objeto menú de tipo `ContextMenu`que debe construir, la vista correspondiente al menú contextual y un objeto de tipo`ContextMenuInfo` que incluya los datos suplementarios según el tipo de vista utilizada.

Sintaxis

```
public void onCreateContextMenu (ContextMenu menu, View v,
  ContextMenu.ContextMenuInfo menuInfo)
```

Ejemplo

```
@Override
public void onCreateContextMenu(ContextMenu menu, View v,
    ContextMenuInfo menuInfo) {
  super.onCreateContextMenu(menu, v, menuInfo);
  switch (v.getId()) {
  case R.id.layout_principal:
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.contextuel, menu);
    break;
  }
}
```

b. Uso

El método `onContextItemSelected` debe sobrecargarse para procesar la elección del usuario entre las opciones del menú contextual. Este método recibe como parámetro el objeto de tipo`MenuItem` seleccionado. Devuelve `true` si procesa la opción seleccionada, o `false` en caso contrario.

Sintaxis

```
public boolean onContextItemSelected (MenuItem item)
```

Ejemplo

```
@Override
public boolean onContextItemSelected(MenuItem item) {
  switch (item.getItemId()) {
  case R.id.menu_contextual_opcion1 :
    procesamientoContextual1();
    return true;
  }
  return super.onContextItemSelected(item);
}
```

Barra de acción

Nuevo elemento aparecido con Android 3.0 (API 11) a través del tema holográfico, la barra de acción reemplaza a la barra de título así como al menú de actividad tal y como aparecen en las versiones anteriores.

Por defecto, de izquierda a derecha, la barra incluye el icono de la actividad o por defecto de la aplicación, su título así como las opciones del menú de actividad que se muestran directamente en la barra, mediante un submenú representado por un icono situado a la derecha del todo.

También es posible insertar directamente widgets en la barra de acción como, por ejemplo, una barra de búsqueda, pestañas con fragmentos o una lista de navegación. También es posible modificar la apariencia de la barra de acción y reemplazar el icono por una imagen.

1. Opciones de menú

Visibles directamente o accesibles mediante el submenú, las opciones de menú se gestionan tal y como se explica en la sección Menús.

Por defecto, las opciones de menú aparecen en el submenú de la barra de acción, submenú accesible mediante el icono situado a la derecha del todo. Es posible situar directamente una de estas opciones de menú sobre la barra de acción, hay que utilizar para ello el atributo`android:showAsAction` de la etiqueta `item` correspondiente en la descripción XML del menú. Las posibles opciones para el atributo `android:showAsAction` son las siguientes:

- `always`: el elemento se mostrará siempre en la barra de acción.
- `ifRoom`: el elemento se mostrará si el espacio disponible en la barra de acción lo permite.
- `never`: el elemento no se mostrará jamás directamente en la barra de acción.

Existen dos atributos suplementarios que pueden ajustarse a los siguientes valores:

- El valor `withText` permite indicar que también es preciso mostrar el título de la opción.
- El valor `collapseActionVue` (disponible a partir de la API 14) permite indicar al sistema que puede reducir el elemento a un icono, y mostrar la totalidad del elemento cuando el usuario haga clic en el menú (en el caso de que el elemento sea un elemento compuesto, como una zona de búsqueda, por ejemplo).

Ejemplo

```
<xml version="1.0" encoding="utf-8">
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item android:id="@+id/menu_principal_opcion1"
        android:icon="@drawable/menu_principal_opcion1"
        android:title="@string/menu_principal_opcion1"
        android:onClick="traitementOption1"
        android:showAsAction="ifRoom|withText" />
</menu>
```

El método `getActionBar` de una actividad permite recuperar la instancia de su barra de acción de tipo `ActionBar`.

Sintaxis

```
public ActionBar getActionBar ()
```

Ejemplo

```
ActionBar actionBar = getActionBar();
```

2. Icono de la aplicación

En una barra de acción, el icono de la actividad o por defecto de la aplicación forma también parte del botón de acción. Cuando el usuario hace clic sobre él, se recomienda que la actividad en curso vuelva a su estado inicial o que la aplicación vuelva a la pantalla de bienvenida (véase el capítulo Los fundamentos - Actividad).

Para detectar el clic sobre el botón, hay que implementar el método `onOptionsItemSelected`visto anteriormente (véase la sección Menús). Éste recibe un objeto de tipo `Item` cuyo identificador único es, en este caso, `android.R.id.home`. A continuación es preciso implementarlo e iniciar la acción correspondiente.

El método `setDisplayHomeAsUpEnabled` permite agregar una marca al icono para indicar al usuario que al hacer clic sobre él es posible volver, no al inicio, sino a la actividad anterior. Hay que pasarle como parámetro un valor booleano que active o no esta posibilidad.

Este método sólo permite modificar la visualización del icono, es tarea del desarrollador procesar el clic del botón como se ha descrito anteriormente.

Sintaxis

```
public abstract void setDisplayHomeAsUpEnabled(boolean showHomeAsUp)
```

Ejemplo

```
actionBar.setDisplayHomeAsUpEnabled(true);
```

Notificaciones

Una aplicación puede necesitar advertir y notificar al usuario. Para ello, Android proporciona varias soluciones dependiendo de si la aplicación se ejecuta en primer plano o como tarea de fondo. Estas tres soluciones son: toast (mensaje temporal), la caja de diálogo y la barra de estado.

1. Toast

En Android, un toast es un mensaje que se muestra durante varios segundos en primer plano en una ventana independiente hecha a medida del mensaje.

Esta ventana no acepta la interacción del usuario. Android se encarga de hacerla aparecer y desaparecer a continuación.

Es el sistema de notificación perfecto para mensajes de carácter puramente informativo.

Es posible que el usuario no vea los mensajes toast si desvía su vista durante algunos segundos de la pantalla. La importancia de estos mensajes es muy baja y no deben suponer ninguna incidencia.

Para crear un objeto de tipo `Toast` muy sencillo, basta con utilizar uno de los métodos estáticos`makeText` de la clase `Toast`. Este método recibe como parámetro el contexto de la aplicación, el identificador único del mensaje que se quiere visualizar o directamente el mensaje en forma de cadena de caracteres y, por último, la duración de la visualización del mensaje en pantalla. Esta duración puede ser la constante `Toast.LENGTH_SHORT` para indicar una duración muy corta, del orden de algunos segundos, o la constante `Toast.LENGTH_LONG` para una duración algo más prolongada. El método `makeText` devuelve el objeto `Toast` creado.

Sintaxis

```
public static Toast makeText (Context context, int resId,
  int duration)
public static Toast makeText (Context context, CharSequence text,
  int duration)
```

Ejemplo

```
Toast toast = Toast.makeText(this, "Mensaje",
  Toast.LENGTH_SHORT);
Toast toast = Toast.makeText(this, R.string.message,
  Toast.LENGTH_LONG);
```

Para mostrar el mensaje, basta con invocar a continuación al método `show`.

Sintaxis

```
public void show ()
```

Ejemplo

```
toast.show();
```

Un pequeño truco, para evitar tener que declarar el objeto toast, consiste en encadenar directamente la llamada al método `show` tras la llamada al método `makeText`.

Ejemplo

```
Toast.makeText(this, "Mensaje", Toast.LENGTH_SHORT).show();
```

La clase `Toast` proporciona otros métodos que permiten posicionar el mensaje en cualquier lugar de la pantalla o incluso personalizar la vista del mensaje proporcionando su propio layout.

2. Caja de diálogo

Una caja de diálogo es una ventana flotante modal, es decir, que aparece en primer plano e impide realizar cualquier otra interacción con las ventanas de fondo. El usuario sólo puede interactuar con la caja de diálogo mediante, por ejemplo, el uso de botones **Aceptar** y **Cancelar**.

En términos de notificación, una caja de diálogo así puede utilizarse, por ejemplo, para hacer esperar al usuario mostrando una barra de progreso. También puede utilizarse para realizar una pregunta al usuario y solicitarle su aprobación (**Aceptar**) o no (**Cancelar**).

a. Generalidades

Es posible crear todo tipo de cajas de diálogo pasando un layout personalizado a la clase`Dialog`. No obstante, la mayor parte de las aplicaciones tienen prácticamente las mismas necesidades en cuanto a cajas de diálogo. Por ello, Android proporciona cajas de diálogo preformateadas, simplificando el trabajo al desarrollador. Éstas heredan de la clase `Dialog`.

La caja de diálogo más extendida es la caja de diálogo de alerta, que estudiaremos aquí. Otra es la caja de diálogo de progreso, representada por la clase`ProgressDialog`, que permite informar al usuario que un procesamiento está en curso y le permite además seguir el nivel de progreso.

Otras cajas de diálogo permiten al usuario introducir una fecha o una hora. Se representan respectivamente por las clases `DatePickerDialog` y `TimePickerDialog`.

Observe que sobre ciertos dispositivos y sobre ciertas versiones de Android, la fecha inicial de una caja de diálogo de tipo `DatePickerDialog` que se muestra al usuario no puede ser inferior al año 2000 (incluso aunque el desarrollador necesite configurar una fecha anterior en el atributo `android:startYear` y la compilación se ejecute sin problemas).

b. Caja de diálogo de alerta

Como su propio nombre indica, la caja de diálogo de alerta permite alertar al usuario. Para ello, permite mostrar un título, un icono, entre cero y tres botones así como un mensaje o una lista de elementos que incluyen por lo general casillas de opción o botones de radio.

Una caja de diálogo de alerta es una instancia de tipo `AlertDialog`. Para ayudar a su construcción, Android proporciona la clase `AlertDialog.Builder`. El uso de una instancia de este tipo permite generar dicha caja de diálogo.

La creación de una instancia de tipo `AlertDialog.Builder` se realiza mediante uno de sus constructores. Éstos reciben como parámetro el contexto de la aplicación y para uno de ellos el tema gráfico.

Sintaxis

```
public AlertDialog.Builder (Context context)
public AlertDialog.Builder (Context context, int theme)
```

Ejemplo

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

Los distintos métodos de la clase `AlertDialog.Builder` permiten componer a continuación la caja de diálogo. Por ejemplo, los método `setTitle` y `setMessage` permiten especificar respectivamente el título y el mensaje bien pasando como parámetro una secuencia de caracteres, o bien indicando el identificador del recurso que se quiere mostrar. Estos métodos devuelven la misma instancia de tipo `AlertDialog.Builder` de modo que sea posible encadenar las llamadas a los métodos.

Sintaxis

```
public AlertDialog.Builder setTitle (CharSequence title)
public AlertDialog.Builder setTitle (int titleId)
public AlertDialog.Builder setMessage (CharSequence message)
public AlertDialog.Builder setMessage (int messageId)
```

Ejemplo

```
builder.setTitle(R.string.title).setMessage(R.string.message);
```

Los métodos `setIcon` permiten especificar el icono que se quiere mostrar. Reciben como parámetro un objeto de tipo `Drawable`, o bien el identificador del recurso. Devuelven el objeto de tipo `AlertDialog.Builder`.

Sintaxis

```
public AlertDialog.Builder setIcon (Drawable icon)
public AlertDialog.Builder setIcon (int iconId)
```

Ejemplo

```
builder.setIcon(android.R.drawable.ic_dialog_alert);
```

Es posible agregar botones en una caja de diálogo para interactuar con el usuario permitiéndole, por ejemplo, anular o confirmar una acción. La clase `AlertDialog.Builder` define tres botones, todos opcionales. Se corresponden con una respuesta negativa, neutra o positiva del usuario.

Estos botones se agregan a la caja de diálogo utilizando respectivamente los métodos`setNegativeButton`, `setNeutralButton` y `setPositiveButton`. Estos métodos reciben como primer parámetro el texto que se muestra sobre el botón al que representan bien mediante una cadena de caracteres, o bien mediante el identificador del recurso. Su segundo parámetro es un objeto de tipo `DialogInterface.OnClickListener` cuyo método `onClick` se invoca cuando el usuario hace clic sobre el botón correspondiente. Estos métodos devuelven, también, el objeto de tipo `AlertDialog.Builder`.

El botón positivo se posiciona automáticamente en la parte izquierda, el neutro en el medio y el negativo a la derecha. Siendo opcionales, existen múltiples combinaciones en la visualización de estos botones.

Los términos negativos, neutros o positivos sirven únicamente para ayudar al desarrollador. No imponen ninguna restricción. Si bien no se recomienda, es posible anular una acción desde un botón positivo y validar una acción desde un botón negativo...

Sintaxis

```
public AlertDialog.Builder setNegativeButton (CharSequence text,
  DialogInterface.OnClickListener listener)
```

```
public AlertDialog.Builder setNegativeButton (int textId,
  DialogInterface.OnClickListener listener)
public AlertDialog.Builder setNeutralButton (int textId,
  DialogInterface.OnClickListener listener)
public AlertDialog.Builder setNeutralButton (CharSequence text,
  DialogInterface.OnClickListener listener)
public AlertDialog.Builder setPositiveButton (int textId,
  DialogInterface.OnClickListener listener)
public AlertDialog.Builder setPositiveButton (CharSequence text,
  DialogInterface.OnClickListener listener)
```

Ejemplo

```
builder.setPositiveButton(android.R.string.yes,
  new DialogInterface.OnClickListener() {
    public void onClick(DialogInterface dialog, int id) {
      finish();
    }
  })
.setNegativeButton(android.R.string.no,
  new DialogInterface.OnClickListener() {
    public void onClick(DialogInterface dialog, int id) {
      dialog.cancel();
    }
  }
);
```

En vez de un mensaje, es posible informar una lista de elementos que incluyan eventualmente casillas de opción o botones de radio. Es posible hacer esto utilizando uno de los métodos`setItems`. Reciben como parámetro bien el identificador de un recurso de tipo tabla, o bien una tabla de secuencias de caracteres. Su segundo parámetro es un objeto de tipo`DialogInterface.OnClickListener` cuyo método `onClick` se invoca cuando el usuario presiona sobre el botón correspondiente. Estos métodos devuelven el objeto de tipo`AlertDialog.Builder`.

Esta lista puede incluir botones de radio o casillas de opción sobre cada uno de los elementos para componer una lista de selección única o múltiple. Para ello, en vez del método`setItems`, se utilizará respectivamente los métodos `setSingleChoiceItems` y`setMultiChoiceItems`.

Sintaxis

```
public AlertDialog.Builder setItems (int itemsId,
  DialogInterface.OnClickListener listener)
public AlertDialog.Builder setItems (CharSequence[] items,
  DialogInterface.OnClickListener listener)
```

Ejemplo

```
final String[] seleccion = { "Selección 1", "Selección 2", "Selección 3" };

builder.setItems(seleccion, new DialogInterface.OnClickListener() {
  public void onClick(DialogInterface dialog, int item) {
    Toast.makeText(MiActividadPrincipal.this, seleccion[item],
      Toast.LENGTH_SHORT).show();
  }
});
```

Si bien la clase `AlertDialog.Builder` proporciona un método `show` que permite crear y mostrar la caja de diálogo, se recomienda dejar su gestión a la actividad o al fragmento del que

depende.

De este modo, una vez se hayan definido los distintos elementos que componen la caja de diálogo, sólo queda generar la instancia de tipo `AlertDialog` correspondiente invocando almétodo `create` del objeto de tipo `AlertDialog.Builder` asociado. Este método devuelve una instancia de tipo `AlertDialog` que podrá explotarse en la actividad o el fragmento correspondiente.

Sintaxis

```
public AlertDialog create ()
```

Ejemplo

```
AlertDialog dialogo = builder.create();
```

Para una actividad, es necesario implementar el método `onCreateDialog` que devolverá la instancia de tipo `AlertDialog` creada aquí. La visualización de la caja de diálogo se realiza mediante el método `showDialog` de la actividad.

Para un fragmento, es necesario crear una nueva clase que herede de la clase`DialogFragment` e implemente el método `onCreateDialog` que devolverá la instancia de tipo `AlertDialog` creada aquí. La creación y la visualización del fragmento, y por tanto de la caja de diálogo, se realizan desde la actividad, como para cualquier fragmento.

3. Barra de notificación

La barra de notificación o barra de estado es el sistema que proporciona Android para permitir a las aplicaciones que se ejecuten como tarea de fondo advertir al usuario sin perturbar su uso actual del dispositivo. Esta barra recibe y almacena las notificaciones que se le envían desde las aplicaciones.

Dependiendo del dispositivo - y de la versión de Android instalada en el mismo se ubica bien en la parte superior o bien debajo del todo en la pantalla. En su lado izquierdo muestra los iconos de las notificaciones que ha recibido. Estirándola hacia abajo aparece el detalle de cada una de ellas.

Es posible eliminar, en un aplicación, esta barra de notificación de la pantalla. La forma de realizarlo difiere en función de la versión de Android.

La visualización de una notificación en la barra de notificación se realiza utilizando un layout por defecto. Desde la API 14 de Android (Android 4.0.1), existen dos formatos de notificación posibles: el formato normal y el formato extendido. En las versiones anteriores, sólo está disponible el formato llamado normal.

Las notificaciones con formato normal presentan un icono, un título en una primera línea, un mensaje en una segunda línea y una fecha. Es posible especificar una acción a realizar cuando el usuario haga clic sobre la notificación.

En el modo extendido, se agrega una zona suplementaria, entre el título y la línea del mensaje. Con esta configuración, la zona suplementaria puede contener una imagen (con un alto máximo de 256 dp) o varias líneas de texto.

Tras la recepción de la notificación es posible iniciar otras funcionalidades solas o combinadas entre sí tales como la visualización de un mensaje en la parte superior de la barra de notificación, la reproducción de un sonido, la vibración del dispositivo o incluso el parpadeo de la luz del dispositivo.

También es posible utilizar un layout personalizado para mostrar la notificación en la barra de notificación en lugar del layout por defecto.

a. Creación de una notificación

La clase `Notification.Builder` hace su aparición con Android 3.0 (API 11) con el objetivo de facilitar la creación de notificaciones representadas por la clase `Notification`. La clase`Notification.Builder` proporciona todo un conjunto de métodos que permiten definir los elementos de una notificación sin utilizar directamente el objeto de tipo `Notification`. No detallaremos aquí más que las características esenciales de una notificación estándar.

Para las versiones inferiores a Android 3.0 (API 11), que no pueden utilizar la clase`Notification.Builder`, hay que crear directamente un objeto de tipo `Notification` y pasarle los elementos que se quieren utilizar en sus variables públicas.

Para crear una notificación con la clase `Notification.Builder`, basta con utilizar su constructor. Éste recibe como parámetro el contexto de la aplicación.

Sintaxis

```
public Notification.Builder (Context context)
```

Ejemplo

```
Notification.Builder notificationBuilder =
  new Notification.Builder(this);
```

Los métodos `setContentTitle`, `setContentText` y `setContentInfo` permiten especificar respectivamente el título, el mensaje y la información de la notificación. Reciben como parámetro una secuencia de caracteres de tipo `CharSequence` y devuelven el objeto `Builder` en curso con el objetivo de encadenar las sucesivas llamadas.

La llamada al método `setContentInfo` puede reemplazarse por la llamada al método`setNumber` si la información que se quiere visualizar sólo afecta a un número entero. La visualización del resultado será más apropiada.

Sintaxis

```
public Notification.Builder setContentTitle (CharSequence title)
public Notification.Builder setContentText (CharSequence text)
public Notification.Builder setContentInfo (CharSequence info)
```

Ejemplo

```
notificationBuilder.setContentTitle("Título de la notificación")
.setContentText("Mensaje de la notificación")
.setContentInfo("Información");
```

Los métodos `setSmallIcon` permiten especificar el icono que se quiere mostrar en la barra de notificación y en la propia notificación. Reciben como parámetro el identificador único de la imagen y un índice para identificar la imagen si forma parte de un conjunto de imágenes de tipo`LevelListDrawable`.

Se recomienda encarecidamente especificar un icono que permita al usuario ver rápidamente la presencia de la notificación.

Sintaxis

```
public Notification.Builder setSmallIcon (int icon)
public Notification.Builder setSmallIcon (int icon, int level)
```

Ejemplo

```
notificationBuilder.setSmallIcon(R.drawable.icon);
```

El método `setLargeIcon` permite agregar una imagen en la notificación además de la del icono. Este método recibe como parámetro el objeto de tipo `Bitmap` que se quiere mostrar y devuelve el objeto `Builder` en curso.

Sintaxis

```
public Notification.Builder setLargeIcon (Bitmap icon)
```

Ejemplo

```
notificationBuilder.setLargeIcon(bitmap);
```

Para agregar una zona de texto extendido a la notificación (API 16 o superior), es preciso instanciar un objeto de tipo `Notification.InboxStyle`, agregarle líneas mediante el método`addLine` y, por último, asociar la instancia de `Notification.InboxStyle` al objeto`Notification.Builder`.

Sintaxis

```
public Notification.Builder setStyle(Notification.InboxStyle inboxStyle)
```

Ejemplo

```
Notification.Builder builder = new Notification.Builder(this);
// [...]
Notification.InboxStyle inbox = new InboxStyle();
for(int i =1;i<6;i++)
   inbox.addLine("Línea número " + i);
builder.setStyle(inbox);
```

El método `setContentIntent` permite especificar una acción, por ejemplo la ejecución de una actividad, que se ejecutará cuando el usuario haga clic sobre la notificación. Para ello, este método recibe como parámetro un objeto de tipo `PendingIntent` (consulte el capítulo Los fundamentos - Intención).

Sintaxis

```
public Notification.Builder setContentIntent (PendingIntent intent)
```

Ejemplo

```
notificationBuilder.setContentIntent(pendingIntent);
```

El método que permite crear la notificación previamente configurada cambia en función de la versión de Android. Antes de la API 16, se utiliza el método `getNotification`. Desde la versión 16 (Android 4.1), este método se reemplaza por el método build. En cambos casos, se devuelve un objeto de tipo `Notification`.

Sintaxis

```
public Notification getNotification ()
```

```
public Notification build ()
```

Ejemplo

```
Notification notification = null ;
if (Build.VERSION.SDK_INT < 16)
     notification = notificationBuilder.getNotification();
else
     notification = notficationBuilder.build() ;
```

b. Envío de una notificación

La gestión y la ejecución de las notificaciones las realiza el gestor de notificaciones. Se trata de un servicio del sistema que gestiona las notificaciones.

Como para todo servicio del sistema, es posible recuperar su instancia utilizando el método`getSystemService`. Éste recibe como parámetro el nombre del servicio y devuelve la instancia de tipo `Object` que quiere convertir en el tipo de servicio concreto.

En este caso, el servicio deseado es el gestor de notificaciones que viene indicado por la constante`Context.NOTIFICATION_SERVICE` de tipo `NotificationManager`.

Sintaxis

```
public abstract Object getSystemService (String name)
```

Ejemplo

```
NotificationManager notificationManager =
  (NotificationManager) getSystemService(NOTIFICATION_SERVICE);
```

Cada notificación debe estar identificada de forma única en la aplicación. Para ello, el identificador se representa por la combinación de un entero y de una cadena de caracteres. Si la cadena es`null`, el valor entero representa por sí solo al identificador. En ese caso, debe ser único en la aplicación.

Si se utiliza un identificador existente, la notificación a la que reenvía se modificará con los nuevos parámetros indicados.

El envío de una notificación se realiza a continuación invocando simplemente a uno de los métodos`notify`. Éstos reciben como parámetro el identificador de la notificación y la notificación correspondiente.

Sintaxis

```
public void notify (int id, Notification notification)
public void notify (String tag, int id, Notification notification)
```

Ejemplo

```
notificationManager.notify(1, notificacion);
```

Internacionalización

Una de las principales claves del éxito de los smartphones y tabletas Android es la oferta de cientos de miles de aplicaciones a través de Play Store. Como veremos en otro capítulo (véase el capítulo Publicar una aplicación - Publicación de la aplicación en Play Store), Play Store permite descargar aplicaciones en distintos países del mundo. Por ello, si el desarrollador quiere satisfacer de la mejor forma a los usuarios, tendrá que proveer aplicaciones en su idioma. Tendrá no sólo que traducir los textos sino también adaptar el formato de los números, las monedas, los gráficos, los archivos de sonido...

Android tiene en cuenta la internacionalización de forma nativa, de modo que es relativamente sencillo programar una aplicación multiidioma.

A menudo se abrevia el vocablo inglés internacionalización (internationalisation) utilizando el término i18n, dado que se trata de una i, seguida de 18 caracteres, y termina con una n.

Durante la ejecución de una aplicación, Android utiliza automáticamente los datos de la aplicación que mejor se corresponden con el dispositivo y su configuración.

Cuando el dispositivo está configurado para utilizar parámetros regionales distintos a los que propone la aplicación, Android utiliza los parámetros regionales por defecto de la aplicación.

El archivo `strings.xml` debe estar ubicado por defecto en la carpeta `res/values/`.

En este archivo se indican los valores de las cadenas de caracteres en la lengua que se utilizará por defecto.

Sintaxis

```
<string name="nombre recurso">"cadena de caracteres"</string>
```

Las comillas que agrupan la cadena de caracteres no son obligatorias pero permiten utilizar caracteres especiales tales como el apóstrofe...

Ejemplo

```
<xml version="1.0" encoding="utf-8">
<resources>
  <string name="nombre_aplicacion">"MyApplication"</string>
  <string name="aceptar">"Ok"</string>
  <string name="bienvenida">"Welcome!"</string>
  <string name="confirmacion">"Are you sure"</string>

</resources>
```

En este ejemplo, el idioma inglés se utiliza como idioma por defecto.

El archivo de traducción por defecto, `strings.xml`, debe existir obligatoriamente y contener todas las cadenas de caracteres utilizadas en la aplicación. Si no fuera el caso, la aplicación no funcionará y aparecerá un mensaje de error.

A continuación bastará con traducir todo o parte de este archivo en los idiomas deseados creando nuevos archivos `strings.xml` en las ubicaciones correspondientes:

Ejemplo de traducción española - archivo res/values-es/strings.xml

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

```
<resources>
  <string name="nombre_aplicacion">"MiAplicacion"</string>

  <string name="bienvenida">"¡Bienvenido!"</string>
  <string name="confirmacion">"¿Está seguro/a?"</string>

</resources>
```

También es posible restringir la aplicación de una traducción a un país además de a una lengua.

Por ejemplo, se creará el archivo `res/values-es-rAR/strings.xml` para la traducción española utilizada en Argentina.

Para un recurso concreto, Android buscará siempre en primer lugar el recurso en el parámetro regional correspondiente a la configuración del dispositivo. Si el recurso no existe, se buscará en los archivos por defecto.

Por ejemplo, supongamos que los parámetros regionales del dispositivo estén configurados para España y que el recurso buscado es la traducción de la palabra `Aceptar`. Android buscará a continuación el recurso en el archivo `res/values-es/strings.xml`. No lo encontrará. Lo buscará a continuación en el archivo por defecto `res/values/strings.xml`. Allí el valor sí aparece. Android utilizará por tanto este valor.

Si fuera necesario, también es posible internacionalizar los demás tipos de recursos tales como layouts, recursos gráficos y demás recursos especializando del mismo modo las carpetas correspondientes.

Ejemplo

res/drawable/...

res/drawable-es/...

res/drawable-es-AR/...

Es posible recuperar la secuencia de caracteres correspondiente a un recurso utilizando uno de losmétodos `getString`. Éstos permiten parametrizar el identificador del recurso. Uno de ellos permite también pasar como parámetro argumentos complementarios para reemplazarlos en la cadena de caracteres. Estos métodos devuelven la cadena de caracteres construida de esta manera.

Sintaxis

```
public final String getString (int resId)
public final String getString (int resId, Object... formatArgs)
```

Ejemplo

```
String msg = getString(R.string.mensajes_no_leidos, "Usuario",  5);
```

Introducción

En realidad, las actividades abordadas en un capítulo anterior no son los únicos componentes de la aplicación. Existen otros que responden a necesidades distintas.

Por ello, en este capítulo descubriremos los fragmentos, una especie de mini-actividades, que permiten explotar plenamente pantallas de gran tamaño. Estudiaremos a continuación los servicios, que permiten ejecutar en segundo plano una tarea sin interfaz gráfica. A continuación llegará el turno a los receptores de eventos que se activan solamente tras la recepción de un mensaje.

Por último, si bien no son un componente principal de la aplicación hablando con propiedad, ilustraremos la implementación de una lista, pues se trata de uno de los elementos más utilizados en las aplicaciones.

Fragmento

La integración de Android en dispositivos con pantallas extra-grandes, tales como las tabletas táctiles, ha revelado nuevas necesidades, en particular a la hora de utilizar plena, eficaz y fácilmente las grandes resoluciones y el espacio disponible que proporcionan estas pantallas.

Los fragmentos, introducidos con la API 11 (Android 3.0) y generalizados para los smartphones en la API 14 (Android 4.0.1), ofrecen un método sencillo y económico en tiempo para adaptar la representación a este espacio disponible.

Un fragmento puede verse como… un fragmento de actividad: que posee su propio layout e implementa el código que gestiona los elementos presentes en dicho layout. De este modo, una actividad puede estar compuesta por uno o varios fragmentos, según el espacio disponible en la pantalla.

El caso de uso típico de los fragmentos es el esquema *Master/Detail*, que puede traducirse como Vista principal/Vista de detalle:

- Existe un primer componente que presenta una lista de elementos.
- La selección de un elemento - haciendo clic en él - ejecuta la representación de otro componente que presenta los datos detallados acerca de dicho elemento.

En una tableta táctil con una pantalla lo suficientemente grande, ambos componentes pondrán visualizarse en la misma pantalla, mientras que para un smartphone con una pantalla algo más pequeña cada componente se mostrará uno a continuación del otro.

Los fragmentos proporcionan todas las funcionalidades necesarias para implementar este esquema.

Recuerde, si bien se introdujo con Android 3.0 (API 11), el concepto de fragmento puede encontrarse en las versiones de Android 1.6 (API 4) y superiores importando la librería de compatibilidad descendente de Android en el proyecto (consulte el capítulo El universo Android - Entorno de desarrollo).

La clase madre de los fragmentos es la clase `Fragment`. De forma similar a las clases hijas ofrecidas por el SDK, especializando la clase madre `Activity`, existen varias clases hijas de la clase`Fragment` que son `DialogFragment`, `ListFragment` y `PreferenceFragment`,representando fragmentos especializados en funciones particulares.

Para definir un fragmento hay que crear una clase que herede, directa o indirectamente, de la clase`Fragment` e implementar, llegado el caso, los métodos heredados.

1. Integración del fragmento

La integración de un fragmento con una actividad puede realizarse de forma declarativa o de forma programática. Si no posee layout, el fragmento sólo puede agregarse de forma programática.

a. Modo declarativo

Recuerde, el modo declarativo permite describir el layout directamente en el código XML.

Para incluir un fragmento en el layout de una actividad, basta con utilizar la etiqueta `fragment` y especificar en su atributo `android:name` el nombre de la clase del fragmento que se quiere instanciar. Una vez instanciado y habiendo invocado al método `onCreateView`, la vista que se recupera se insertará en el layout de la actividad en el lugar de la etiqueta `fragment` específica.

Como con cualquier otro widget, las dimensiones se configuran mediante los atributos`android:layout_width` y `android:layout_height`. También es posible asignar un identificador mediante el atributo `android:id` o una cadena de caracteres mediante el atributo`android:tag`. Si no se le asigna ninguno de estos atributos, el sistema utilizará el identificador interno de la vista del fragmento.

En lo que queda de capítulo, aparecen varios métodos que hacen referencia al identificador único de la vista contenedora. En modo declarativo, este identificador corresponde simplemente con el valor del atributo `android:id` especificado en la etiqueta `fragment`.

Sintaxis

```
<fragment android:name="nombre completo de la clase"
  android:id="@[+][paquete:]id/nombre_recurso"
  android:tag="recurso_texto"
  android:layout_width="dimensión"
  android:layout_height="dimensión"
  ... />
```

Ejemplo

```
<fragment
  android:name="es.midominio.android.miaplicacion.MiFragmento"
  android:id="@+id/mifragmento"
  android:layout_width="match_parent"
  android:layout_height="match_parent" />
```

b. Modo programático

En modo programático, un fragmento debe ubicarse en un componente (widget) ViewGroup de la actividad padre. El gestor de fragmentos permite a la actividad padre realizar las operaciones relativas a los fragmentos: carga, reemplazo, etc.

La instancia del gestor de fragmentos, de tipo `FragmentManager`, la devuelve el método`getFragmentManager` invocado desde la actividad que lo alberga.

Sintaxis

```
public FragmentManager getFragmentManager ()
```

Ejemplo

```
FragmentManager fragmentManager = getFragmentManager();
```

Las operaciones relativas a los fragmentos deben realizarse dentro de transacciones. Una misma transacción puede contener una o varias operaciones secuenciales. Esta transacción, o serie de operaciones, se realiza de forma atómica. Es decir, que todas las operaciones contenidas en la transacción forman un todo indivisible, y se realizarán todas prácticamente al mismo tiempo.

Para crear una transacción, hay que invocar al método `beginTransaction` del gestor de fragmentos recuperado anteriormente. Éste devuelve la transacción bajo la forma de un objeto de tipo `FragmentTransaction`.

Sintaxis

```
public abstract FragmentTransaction beginTransaction ()
```

Ejemplo

```
FragmentTransaction fragmentTransaction =
  fragmentManager.beginTransaction();
```

La transacción puede recibir a continuación una lista de operaciones relativas a los fragmentos, como por ejemplo agregar o suprimir fragmentos.

Los métodos `add` del objeto de tipo `FragmentTransaction` permiten agregar un fragmento en una vista contenedora. Estos métodos reciben como parámetro un objeto de tipo `Fragment` y bien el

identificador único de la vista contenedora que contiene el fragmento, o bien la etiqueta asociada al fragmento, o bien ambas. Estos métodos devuelven el objeto de tipo`FragmentTransaction` de cara a poder encadenar las llamadas a estos métodos.

- El uso de una etiqueta es el único medio para identificar un fragmento que no posee vistas.

- Las vistas de los fragmentos se agregan al contenedor en el orden de llamada al método`add`.

Sintaxis

```
public abstract FragmentTransaction add (int containerViewId,
  Fragment fragment)
public abstract FragmentTransaction add (Fragment fragment,
  String tag)
public abstract FragmentTransaction add (int containerViewId,
  Fragment fragment, String tag)
```

Ejemplo

```
MiFragmento fragmento = new MiFragmento();
fragmentTransaction.add(R.id.vista_contenedora, fragmento);
```

Los métodos `replace` permiten reemplazar el o los fragmentos agregados a la vista contenedora por un nuevo fragmento. Estos métodos reciben como parámetro el identificador único de la vista contenedora, el nuevo fragmento de tipo `Fragment`, y eventualmente la etiqueta asignada al fragmento. Este método devuelve el objeto de tipo `FragmentTransaction` en curso.

Sintaxis

```
public abstract FragmentTransaction replace (int containerViewId,
  Fragment fragment)
public abstract FragmentTransaction replace (int containerViewId,
  Fragment fragment, String tag)
```

Ejemplo

```
fragmentTransaction.replace(R.id.vista_contenedora, fragmento);
```

El método `remove` permite suprimir un fragmento y sus eventuales vistas de la vista contenedora. Este método recibe como parámetro la instancia del fragmento que se quiere suprimir y devuelve el objeto de tipo `FragmentTransaction` en curso.

Sintaxis

```
public abstract FragmentTransaction remove (Fragment fragment)
```

Ejemplo

```
fragmentTransaction.remove(fragmento);
```

La transacción así preparada debe ejecutarse invocando a su método `commit`. Si se ha solicitado agregar esta transacción en la pila de las transacciones de fragmentos que detallamos más adelante, este método devuelve un valor entero positivo identificando la transacción. En caso contrario, devuelve un valor entero negativo.

Sintaxis

```
public abstract int commit ()
```

Ejemplo

```
fragmentTransaction.commit();
```

El gestor de fragmentos permite buscar la instancia de un fragmento particular utilizando bien el identificador único de la vista contenedora o bien la etiqueta del fragmento. Para ello, el gestor proporciona respectivamente los métodos `findFragmentById` y `findFragmentByTag`. Estos métodos devuelven la instancia de tipo `Fragment` o `null` si ningún fragmento se corresponde con el identificador.

Sintaxis

```
public abstract Fragment findFragmentById (int id)
public abstract Fragment findFragmentByTag (String tag)
```

Ejemplo

```
fragmentManager.findFragmentById(R.id.vista_contenedora);
```

2. Fragmentos y representación adaptativa

Combinando todas las nociones que hemos abordado en el libro hasta este punto, resulta bastante sencillo construir un esquema de implementación de representación adaptativa utilizando fragmentos. El objetivo es, como se indicaba al comienzo de este capítulo, construir una interfaz de tipo Vista principal/Vista de detalle.

Para dispositivos de tipo tableta, la pantalla debe mostrar al mismo tiempo una vista principal (típicamente un `ListView`, elemento que se describe en la sección Lista de este capítulo) y una vista detallada del elemento seleccionado en la vista principal.

Para los smartphones, una pantalla - una actividad - muestra la vista principal, que se reemplaza por la vista de detalle cuando se selecciona un elemento de la vista principal.

Cada vista, principal y de detalle, será un fragmento. En el caso de una tableta, una actividad incluirá ambos fragmentos, declarados en el layout XML de la actividad. En el caso de un smartphone, sólo se mostrará un fragmento, y se remplazará por otro cuando sea necesario.

Basta, entonces, al final con crear un layout XML dedicado a las tabletas y otro layout para los smartphones. Es el sistema el encargado de determinar, en tiempo de ejecución, qué layout se utilizará en función del dispositivo: cada archivo de layout se almacenará en la carpeta de layout correspondiente (consulte la sección Layouts - Creación en modo declarativo del capítulo Descubrir la interfaz de usuario).

En el código de la actividad, determinar con qué configuración se ejecutará la aplicación es relativamente sencillo: basta con comprobar la presencia de la ubicación del fragmento vista de detalle mediante el método `findViewById` de la clase `Activity`. Si dicho fragmento está presente, la aplicación se ejecuta en una tableta, en caso contrario, sobre un smartphone: el código debe, únicamente, gestionar en el primer caso la configuración de la vista de detalle (pasándole un identificador del elemento seleccionado, por ejemplo). En el segundo caso, es preciso utilizar una instancia de `FragmentManager` para remplazar el fragmento vista principal por una instancia del fragmento vista de detalle.

3. Ciclo de vida

Los métodos `onCreate`, `onStart`, `onResume`, `onPause`, `onStop` y `onDestroy` son similares a los de una actividad pero sólo aplican al fragmento. Se invocan al mismo tiempo que los mismos métodos de la actividad host. Esto es así puesto que el ciclo de vida de un fragmento depende también del de la actividad que lo contiene.

Existen otros métodos disponibles, específicos del ciclo de vida de un fragmento. Se trata de los métodos `onAttach`, `onCreateView`, `onActivityCreated`, `onDestroyView` y `onDetach`.

a. onAttach

El método `onAttach` es el primer método del ciclo de vida del fragmento que se invoca tras su creación. Recibe como parámetro la instancia de la actividad host que lo contiene.

La instancia de la actividad host que se recibe como parámetro del método `onAttach` puede recuperarse en el fragmento en cualquier momento mediante su método `getActivity`.

Sintaxis

```
public void onAttach (Activity activity)
```

Ejemplo

```
@Override
public void onAttach(Activity activity) {
  super.onAttach(activity);
  procesamiento(activity);
}
```

b. onCreateView

El método `onCreateView`, opcional, permite pasar la vista raíz del layout del fragmento así como toda la jerarquía de vistas correspondiente que se utilizará para diseñar el fragmento. Puede considerarse como el equivalente del método `onCreate` de una actividad (dejando a un lado que, para una actividad, `onCreate` es el primer método invocado, lo que no ocurre aquí).

Este método `onCreateView` se invoca tras el método `onCreate` de la actividad o después del método `onDestroyView`. Recibe como parámetros un objeto de tipo `LayoutInflater` que permite agregar vistas al fragmento, la vista contenedora de tipo `ViewGroup`, en la que se insertará el fragmento, o `null` si no existe la vista contenedora, y un objeto de tipo `Bundle` que contiene los datos salvaguardados de un estado que se pasa al fragmento. Debe devolver la vista raíz de tipo `View` del layout del fragmento. Por defecto, este método devuelve `null`, lo que significa que el fragmento no tiene interfaz gráfica.

Sintaxis

```
public View onCreateView (LayoutInflater inflater,
  ViewGroup container, Bundle savedInstanceState)
```

Ejemplo

```
@Override
public View onCreateView(LayoutInflater inflater,
    ViewGroup container, Bundle savedInstanceState) {
  return inflater.inflate(R.layout.mifragmento, container,
    false);
}
```

La asociación entre la vista y la vista contenedora la realiza el sistema automáticamente. Esto es así porque, en este ejemplo, se especifica que la vista no debe estar asociada, una segunda vez, a su vista contenedora especificando el valor `false` en el último parámetro del método `inflate`.

c. onActivityCreated

El método `onActivityCreated` se invoca cuando la actividad host y su jerarquía de vistas están

creadas. Esta jerarquía comprende en particular, llegado el caso, la vista del fragmento en curso. En concreto, esta llamada se realiza tras la ejecución del método `onCreate` de la actividad host. Este método permite recuperar las instancias de las vistas y restaurarlas si fuera necesario mediante el uso del objeto de tipo `Bundle` que contiene los datos salvaguardados de un estado que se pasa al fragmento.

Sintaxis

```
public void onActivityCreated (Bundle savedInstanceState)
```

Ejemplo

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

d. onDestroyView

El método `onDestroyView` es el simétrico del método `onCreateView`. Se invoca justo antes de que la jerarquía de vistas del fragmento se suprima de la vista contenedora. Este método se invoca incluso si el fragmento no ha proporcionado ninguna vista raíz en el método`onCreateView`.

Sintaxis

```
public void onDestroyView ()
```

Ejemplo

```
@Override
public void onDestroyView() {
  super.onDestroyView();
  limpiaVista();
}
```

e. onDetach

El método `onDetach` es el simétrico del método `onAttach`. Se invoca cuando se libera el fragmento de la actividad. Es el último método puesto a disposición del desarrollador que se invoca antes de la destrucción efectiva e irreversible del fragmento.

Este método permite liberar recursos ligados al fragmento.

Sintaxis

```
public void onDetach ()
```

Ejemplo

```
@Override
public void onDetach() {
  super.onDetach();
  procesamiento();
}
```

4. Salvaguarda y restauración del estado

De forma similar a una actividad, un fragmento también puede salvaguardar su estado cuando se destruye para poder reconstruirlo de forma idéntica más adelante.

La salvaguarda de este estado se realiza en el método `onSaveInstanceState` de la clase`Fragment`. Es la actividad host del fragmento la que decide cuándo salvaguardar el estado del

fragmento y, por tanto, cuándo invocar a este método. La sintaxis y el principio son los mismos que para el método `onSaveInstanceState` de la clase `Activity`.

La restauración del fragmento puede realizarse en los método `onCreate`, `onCreateView` y`onActivityCreated` que reciben todos ellos el objeto de tipo `bundle` anteriormente salvaguardado.

5. Pila de fragmentos

De forma similar a la pila de actividades, existe una pila de transacciones de fragmentos llamadaBack Stack que permite al usuario volver atrás o, en particular, volver a transacciones de fragmentos anteriores. Una transacción consiste en una serie de operaciones que se realizan de forma atómica, de modo que la vuelta atrás, o desapilamiento, de una transacción afecta al conjunto de operaciones contenidas en esta transacción.

A diferencia de las actividades, esta pila la gestiona completamente el desarrollador, en particular para las transacciones de fragmentos y el gestor de fragmentos.

Por ejemplo, cada una de las transacciones de fragmentos puede indicar que tiene que ser incluida en la pila utilizando el método `addToBackStack`. Este método debe invocarse antes del método`commit`. Recibe como parámetro una cadena de caracteres opcional que identifica a la transacción, y devuelve la transacción en curso.

Tras el commit, el fragmento reemplazado o suprimido se agrega a la pila y se invoca a sus método`onPause`, `onStop` y `onDestroyView`.

En el caso en que el fragmento reemplazado o suprimido no sea agregado a la pila, es decir, si el método `addToBackStack` no se utiliza, se invocan sus métodos `onDestroy` y`onDetach` y éste se destruye a continuación.

Sintaxis

```
public abstract FragmentTransaction addToBackStack (String name)
```

Ejemplo

```
OtroFragmento otroFragmento = new OtroFragmento();
fragmentTransaction.replace(R.id.fragmento, otroFragmento)
.addToBackStack(null).commit();
```

En cualquier momento, el usuario puede volver atrás, y desapilar las transacciones de fragmentos presionando la tecla **Volver**. La aplicación también puede desapilar las transacciones de fragmentos de la pila utilizando una de las múltiples variantes de los métodos `popBackStack` o`popBackStackImmediate` del gestor de fragmentos.

La diferencia entre los métodos `popBackStack` y `popBackStackImmediate` reside en el hecho de que la operación de desapilar se realiza de forma asíncrona en el primero, o bien inmediatamente en las llamadas al segundo.

Las variantes más simples no reciben ningún parámetro y desapilan la última transacción agregada a la pila. Los demás reciben como parámetro el identificador de la transacción hasta la que se desea desapilar. Este identificador es bien el identificador de la transacción devuelto anteriormente por el método `commit`, o bien el nombre de la transacción que se ha pasado como parámetro al método`addToBackStack`.

Por último, es posible pasar un flag como parámetro con el valor `POP_BACK_STACK_INCLUSIVE`para indicar que la transacción designada también debe desapilarse, o indicando el valor cero en caso contrario.

El fragmento que se encuentra en la parte superior de la pila es, en lo sucesivo, el fragmento

activo. Retoma su ciclo de vida tras el método `onCreateView`.

Sintaxis

```
public abstract void popBackStack ()
public abstract void popBackStack (int id, int flags)
public abstract void popBackStack (String name, int flags)
public abstract boolean popBackStackImmediate ()
public abstract boolean popBackStackImmediate (int id, int flags)
public abstract boolean popBackStackImmediate (String name, int flags)
```

Ejemplo

```
fragmentManager.popBackStack();
```

El gestor de fragmentos ofrece otras funcionalidades tales como especificar animaciones para las transiciones, un título de tipo breadcrumb o incluso la posibilidad de agregar un listener que reaccione a modificaciones en la pila. Para obtener más información, consulte respectivamente losmétodos `setTransition`, `setBreadCrumbTitle` y`addOnBackStackChangedListener`.

Servicio

Un servicio es un componente de aplicación independiente que no tiene interfaz gráfica y que se ejecuta en segundo plano. Como su propio nombre indica, este componente de aplicación representa un servicio, en el sentido estricto del término, que está disponible para la aplicación que lo contiene y/o para otras aplicaciones.

Un servicio proporciona una interfaz que permite a los demás componentes de la aplicación comunicarse con él.

Para definir un servicio, es preciso crear una clase que herede de la clase `Service` e implementar, llegado el caso, los métodos heredados.

Como con la actividad, la ejecución de un servicio se realiza en el thread principal del proceso de la aplicación de la que forma parte.

Un servicio no se ejecuta en un proceso separado, ni en un thread concurrente del thread principal. Por tanto, no debe bloquear el thread principal durante más de diez segundos, igual que ocurre con las actividades. En el caso de tener que realizar un procesamiento largo, el servicio puede crear un thread concurrente para ello (véase el capítulo Concurrencia, seguridad y red - Programación concurrente). O, para mayor comodidad, el servicio puede heredar de la clase `IntentService` que facilita la gestión de procesamientos asíncronos.

Es posible utilizar un servicio de varias formas: directamente, estableciendo una conexión con él o bien combinando ambos modos.

1. Declaración

Para poder utilizarse, un servicio debe estar declarado en el sistema mediante el manifiesto.

La etiqueta `service` contiene la información propia de un servicio. Como la mayoría de etiquetas del manifiesto, proporciona multitud de atributos. Descubriremos algunos de ellos a medida que vayamos descubriendo las funcionalidades correspondientes. He aquí, como ejemplo, la sintaxis de esta etiqueta y de sus tres atributos principales:

Sintaxis

```
<service android:icon="recurso_gráfico"
         android:label="recurso_texto"
         android:name="cadena de caracteres"
         ... >
    ...
</service>
```

Los atributos `android:icon` y `android:label` tienen las mismas funciones que los de la etiqueta `application` pero limitándose al servicio. Si no se especifican, se utilizarán por defecto los de la aplicación.

El atributo `android:name` permite especificar el nombre del servicio afectado, es decir, su nombre de clase Java precedido del nombre completo del paquete. De forma similar a la actividad, este nombre puede empezar por un punto, queriendo decir que se utilizará el nombre del paquete indicado por el atributo `package` de la etiqueta `manifest` para componer el nombre completo.

Ejemplo

```
<service android:name=".MiServicio"/>
```

2. Uso directo

Es posible ejecutar un servicio invocando al método `startService` y pasándole como parámetro un objeto intención (intent), implícita o explícita, que permita identificar el servicio que se va a ejecutar.

Si el servicio no existe, el método devuelve `null`, en caso contrario devuelve un objeto de tipo`ComponentName`, objeto que permite simplemente identificar un componente de aplicación mediante su paquete y el nombre de su clase.

Sintaxis

```
public abstract ComponentName startService (Intent service)
```

Ejemplo

```
Intent intent = new Intent(this, MiServicio.class);
startService(intent);
```

Una vez iniciado, este mismo servicio puede recibir nuevos `intent` y, por tanto, nuevas instrucciones que procesar, llamando de nuevo al método `startService`. El servicio procesará a continuación estas intenciones como si se tratase del primer `intent` recibido durante el arranque del servicio. Este truco permite, en ciertos casos, evitar tener que establecer una conexión con el servicio tal y como se describe más adelante.

Un servicio puede detenerse en cualquier momento. Sea cual sea el número de llamadas al método`startService` realizadas, una única llamada al método `stopService` detiene el servicio pasado como parámetro. Este método devuelve `true` si se ha encontrado y detenido algún servicio en curso, `false` en caso contrario.

Sintaxis

```
public abstract boolean stopService (Intent service)
```

Ejemplo

```
Intent intent = new Intent(this, MiServicio.class);
stopService(intent);
```

El servicio también puede detenerse a sí mismo invocando los métodos `stopSelf` o`stopSelfResult`. Este último método recibe como parámetro un identificador único designando la última consulta recibida por el método `onStartCommand`. Esto permite asegurar que no existen otras consultas pendientes por procesar antes de autorizar la detención del servicio. El método`stopSelfResult` devuelve `true` si el servicio se detiene, o `false` en caso contrario, particularmente si existen consultas por procesar. El método `onStartCommand` se detalla más adelante.

Sintaxis

```
public final void stopSelf ()
public final boolean stopSelfResult (int startId)
```

Ejemplo

```
public void errorDeProcesamiento() {
  stopSelf();
}
```

3. Uso estableciendo una conexión

Un componente de aplicación también puede establecer una conexión con un servicio, si se lo

permite, sin tener que pasar por el uso directo descrito anteriormente.

Para ello, el servicio proporciona una interfaz a los demás componentes, que pueden utilizarlo para establecer un enlace y comunicarse con el servicio.

En concreto, la conexión se realiza invocando al método `bindService`. Este método recibe como parámetro un objeto intent que designa el servicio a ejecutar, un objeto que implementa la interfaz`ServiceConnection` y unos flags. La interfaz `ServiceConnection` proporciona dos métodos,`onServiceConnected` y `onServiceDisconnected`, que se invocan respectivamente cuando se establece y cuando se pierde la conexión con el servicio. Este método devuelve `true` si la conexión al servicio se ha establecido con éxito, o `false` en caso contrario.

Si es preciso, se ejecuta el servicio si no está ya en ejecución mediante la llamada al método`bindService`.

Sintaxis

```
public abstract boolean bindService (Intent service,
  ServiceConnection conn, int flags)
```

Ejemplo

```
Intent intent = new Intent(this, MiServicio.class);
bindService(intent, conexion, Context.BIND_AUTO_CREATE);
```

La desconexión con el servicio se realiza invocando al método `unbindService` que recibe como parámetro el mismo objeto que se ha pasado durante el establecimiento de la conexión mediante el método `bindService`.

Sintaxis

```
public abstract void unbindService (ServiceConnection conn)
```

Ejemplo

```
unbindService(conexion);
```

4. Ciclo de vida

El ciclo de vida de un servicio describe los estados en los que se puede encontrar el servicio entre su creación, la instanciación y su muerte (la destrucción de dicha instancia). Este ciclo de vida depende de la elección de uso del servicio.

Como se ha comentado antes, es posible combinar el uso directo del servicio y la implementación de una conexión con el servicio. En tal caso, la llamada al método `stopService` puede que no se lleve a cabo si se ha realizado una conexión con el servicio después de haber ejecutado la actividad mediante el método `startService`.

A cada cambio de estado le corresponde un método que puede sobrecargarse en la clase del servicio.

Cada uno de estos métodos debe invocar a su método padre, en caso contrario se generará una excepción.

a. onCreate

El método `onCreate` se invoca automáticamente tras la creación del servicio, bien sea mediante el método `startService` o el método `bindService`.

Sintaxis

```
public void onCreate ()
```

Ejemplo

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

b. onStartCommand

El método `onStartCommand` se invoca automáticamente cada vez que un cliente ejecuta el servicio utilizando el método `startService`.

Este método recibe como parámetro el intent que se proporciona al método `StartService`, los flags y un identificador único que designa al procesamiento de este intent. Este identificador puede pasarse al método `stopSelfResult` llegado el caso. El método `onStartCommand` devuelve un valor que indica cómo reiniciar el servicio si se ha destruido prematuramente.

Sintaxis

```
public int onStartCommand (Intent intent, int flags, int startId)
```

Ejemplo

```
@Override
public int onStartCommand(Intent intent, int flags, int startId) {
    mStartId = startId;
    return super.onStartCommand(intent, flags, startId);
}
```

c. onBind

El método `onBind` se invoca automáticamente tras cada petición de establecimiento de conexión. Este método recibe como parámetro el objeto intent que se proporciona al método `bindService`y devuelve un objeto que implementa la interfaz `IBinder` y que permite comunicarse con el servicio.

Preste atención, la intención recibida se desliga de los datos `Extras` que se le pudieran haber proporcionado.

Sintaxis

```
public abstract IBinder onBind (Intent intent)
```

Ejemplo

```
private final IBinder mBinder = new LocalBinder();

@Override
public IBinder onBind(Intent intent) {
  return mBinder;
}
```

d. onUnbind

El método `onUnbind` se invoca automáticamente cada vez que se cierra una conexión. Este método recibe como parámetro el objeto intent que se proporciona al método `bindService`. Por defecto, devuelve `false`. Devolver `true` permite llamar al método `onRebind` en futuras

reconexiones.

Sintaxis

```
public boolean onUnbind (Intent intent)
```

Ejemplo

```
@Override
public boolean onUnbind(Intent intent) {
  procesamiento(intent);
  return super.onUnbind(intent);
}
```

El método `onRebind` se invoca únicamente si el método `onUnbind` devuelve `true`. Este método es similar al método `onBind`.

Sintaxis

```
public void onRebind (Intent intent)
```

Ejemplo

```
@Override
public void onRebind(Intent intent) {
  procesamiento(intent);
  super.onRebind(intent);
}
```

e. onDestroy

El método `onDestroy` es el simétrico del método `onCreate`. Lo invoca el sistema automáticamente antes de suprimir un servicio. Es el último método que se pone a disposición del desarrollador antes de destruir efectiva e irreversiblemente el servicio.

Este método permite liberar los recursos ligados al servicio, por ejemplo un thread.

Sintaxis

```
public void onDestroy ()
```

Ejemplo

```
@Override
public void onDestroy() {
  suprimeThread();
  super.onDestroy();
}
```

Receptor de eventos

Un receptor de eventos es un componente de aplicación cuyo rol consiste únicamente en recibir eventos y procesarlos como debe.

De forma similar a un servicio, un receptor de eventos no posee una interfaz gráfica. Cuando recibe un mensaje y se quiere informar al usuario puede, por ejemplo, utilizar la barra de notificaciones o incluso ejecutar una actividad.

Como para la actividad y el servicio, la ejecución de un receptor de eventos se opera en el thread principal del proceso de la aplicación de la que forma parte.

Un receptor de eventos no debe bloquear el thread principal durante más de diez segundos (véase el capítulo Concurrencia, seguridad y red - Programación concurrente).

Para definir un receptor de eventos, es necesario crear una clase que herede de la clase`BroadcastReceiver` e implementar únicamente el método `onReceive`.

1. Evento

Los eventos los produce bien el sistema, o bien las propias aplicaciones. Se envían a todos los receptores de eventos filtrando el evento concreto.

En particular, estos eventos son objetos de tipo `Intent` que designan la acción que acaba de ejecutarse o el evento que acaba de producirse.

Para enviar tal evento, el componente emisor dispone de varios métodos, de los que el más sencillo es el método `sendBroadcast`. Este método recibe como parámetro el objeto intent que se difundirá a los receptores.

El envío de la intención se realiza de forma asíncrona de cara a no bloquear el componente que envía el evento.

Sintaxis

```
public abstract void sendBroadcast (Intent intent)
```

Ejemplo

```
Intent intent = new Intent(es.midominio.android.miaplicacion.EVT_1);
sendBroadcast(intent);
```

Invocando este método, la propagación de la intención hacia todos los componentes de destino se realiza de forma simultánea, o casi. Es no obstante posible propagar la intención receptor a receptor, uno cada vez, y esto de forma ordenada o no. El receptor de eventos puede entonces decidir pasar un resultado al receptor de eventos siguiente e incluso detener la propagación del intent.

Para ello, es preciso utilizar el método `sendOrderedBroadcast`. Este método recibe como parámetros el objeto intent que se quiere difundir a los receptores de eventos y un permiso opcional que debe poseer el receptor, `null` en el caso de ninguno.

El orden de llamada a los receptores de eventos se determinará según el atributo`android:priority` de su `intent-filter` (véase el capítulo Los fundamentos - Intención).

Sintaxis

```
public abstract void sendOrderedBroadcast (Intent intent,
  String receiverPermission)
```

Ejemplo

```
Intent intent = new Intent(es.midominio.android.miaplicacion.EVT_1);
sendOrderedBroadcast(intent, null);
```

2. Declaración

Para crearlo de forma estática, un receptor de eventos debe declararse en el sistema a través del manifiesto.

La etiqueta `receiver` contiene información propia de un receptor de eventos. Como la mayoría de etiquetas, esta etiqueta proporciona varios atributos. Los descubriremos a medida que vayamos descubriendo las funcionalidades correspondientes. He aquí la sintaxis de esta etiqueta y de sus principales atributos:

Sintaxis

```
<receiver android:icon="recurso_gráfico"
          android:label="recurso_texto"
          android:name="cadena de caracteres"
          ... >
    ...
</receiver>
```

Los atributos `android:icon` y `android:label` realizan las mismas funciones que los de la etiqueta `application` pero limitadas al receptor de eventos. Si no se especifican, se utilizarán por defecto las de la aplicación.

El atributo `android:name` permite especificar el nombre del evento correspondiente, es decir su nombre de clase Java precedido del nombre entero del paquete. De forma similar a la actividad, este nombre puede comenzar con un punto queriendo decir que se agregará al paquete indicado por el atributo `package` de la etiqueta `manifest` para componer el nombre completo.

3. Ciclo de vida

El ciclo de vida del componente receptor de eventos es muy simple.

Antes de recibir su primer evento, el componente está inactivo. Se activa cuando recibe un evento como parámetro de su método `onReceive`. Tras la salida de este método, el componente se vuelve inactivo de nuevo.

En efecto, para el sistema, el hecho de salir del método `onReceive` significa que el componente ha terminado de procesar el evento, incluso si ha ejecutado un thread asíncrono que sigue ejecutándose tras haber finalizado el método `onReceive`.

En este caso, si ningún otro componente de la aplicación está activo, el sistema puede decidir matar el proceso de la aplicación y, por tanto, el thread en curso.

También, para evitar este problema, se aconseja lanzar la ejecución del thread desde un servicio. El sistema detectará entonces el servicio como componente activo y no matará el proceso, salvo en aquellos casos extremos en que se necesiten recursos.

El sistema gestiona de forma automática la creación y la destrucción de la instancia del receptor de eventos.

El método `onReceive` se invoca automáticamente tras la recepción de un evento. Este método recibe como parámetros el contexto de la aplicación y el evento de tipo `Intent` recibido.

Sintaxis

```
public abstract void onReceive (Context context, Intent intent)
```

Ejemplo

```
public class ReceptorDeEventos extends BroadcastReceiver {
  @Override
  public void onReceive(Context context, Intent intent) {
  }
}
```

Lista

Debido a su pequeña pantalla, los smartphones deben adoptar una interfaz de usuario con una cierta ergonomía específica. Para ello, y como hemos visto en el capítulo Completar la interfaz de usuario, la plataforma Android proporciona múltiples componentes gráficos. Uno de los componentes más representativos de la forma de navegar en una aplicación es la lista de elementos. Se trata de una lista vertical que posee una barra de desplazamiento. El usuario puede desplazar la lista de arriba a abajo e inversamente para seleccionar un elemento.

Esta lista vertical se ha convertido, con el paso de los años, en un elemento prácticamente imprescindible de toda aplicación Android.

La vista que representa una lista es de tipo `ListView`. La implementación de una lista implica, a su vez, proveer dos elementos: un layout para visualizar cada elemento de la lista y un adaptador (*Adapter*, en inglés) para el origen de datos.

Incluso aunque una lista pueda estar integrada en cualquier actividad, el sistema Android provee un tipo de actividad específica para el uso de elementos `ListView`: la clase `ListActivity`, que hereda de la clase `Activity`. El uso de esta clase evita al desarrollador tener que realizar parte del trabajo de implementación propio de `ListView`.

Las siguientes secciones presentan ambos enfoques: la sección Implementación estándar estudia el uso de una lista en el marco de un uso estándar normal de los componentes proporcionados por Android, mientras que la sección Implementación específica presenta el conjunto de nociones necesarias para implementar un objeto `ListView` completamente específico.

Observe que con la llegada de Android 3.0 (API 11), adaptado a las grandes pantallas de las tabletas táctiles, aparece una variación de la clase `ListActivity` como fragmento de tipo `ListFragment`.

Sea en el marco de una actividad o de un fragmento, el funcionamiento general de una lista es el mismo. Utilizaremos por tanto el término lista para denominar a la vez a las listas de tipo`ListActivity` y aquellas de tipo `ListFragment`.

1. Implementación estándar

La implementación estándar se basa en el uso de la clase `ListActivity` como soporte de la lista. Esta clase utiliza un layout por defecto para la composición de la actividad, que presenta una lista única, centrada en la pantalla.

La vista `ListView` que representa la lista es de hecho una vista de tipo `ViewGroup`. Cada línea de la lista es una vista hija que se encarga de la representación gráfica de los datos de un elemento de la lista.

a. Layout de los elementos de la lista

En el marco de una implementación estándar, Android proporciona layouts predefinidos para los elementos de la lista. Estos layouts son dos:

- El layout `android.R.layout.simple_list_item_1` comprende únicamente un objeto de tipo `TextView`.
- El layout `android.R.layout.simple_list_item_2` comprende dos objetos de tipo`TextView`; el segundo debajo del primero y con un tipo de letra menor.

b. Adaptadores

Los datos de los elementos de una lista puede tener distintos orígenes. Este puede ser el caso, por ejemplo, de datos almacenados directamente en el código de la aplicación, datos que provienen de una base de datos o datos provistos por un proveedor de contenidos (véase el capítulo La persistencia de los datos).

Para poder soportar cualquier tipo de origen de datos, una lista consulta los datos que debe representar mediante una interfaz de tipo `ListAdapter`. Este adaptador realiza el enlace entre la vista que debe mostrar los datos y los propios datos.

Android proporciona varios tipos de adaptadores de listas que implementan la interfaz`ListAdapter` y se corresponden con los orígenes y tipos de datos más comunes:

- La clase `ArrayAdapter<T>` representa un adaptador de datos almacenados en forma de lista genérica.
- La clase `CursorAdapter`, abstracta, representa el adaptador de datos almacenados en base de datos o provistos por un gestor de contenidos. Es preciso implementar los métodos`bindView` y `newView`.
- La clase `SimpleCursorAdapter`, en sí misma, es una implementación del cursor Adapter que permite gestionar fácilmente datos de tipo cadena de caracteres y de tipo imagen.

Por defecto, la clase `CursorAdapter` y, por tanto, la clase `SimpleCursorAdapter`también, realizan la carga de los datos en el thread principal. Esto no está aconsejado, de cara a no bloquear la aplicación (véase el capítulo Concurrencia, seguridad y red - Programación concurrente). Es preferible, por tanto, realizar estas cargas de datos en un thread secundario. Desde la versión 3.0 (API 11), Android proporciona las clases complementarias`LoaderManager` y `CursorLoader` que permiten realizar esto de forma más sencilla.

c. Implementación

La implementación se realiza en dos etapas: en primer lugar es preciso instanciar un adaptador y, a continuación, pasarle este adaptador a la lista.

La instanciación del adaptador se realiza en función de su tipo.

Instanciar un ArrayAdapter<T>

Por defecto, la clase `ArrayAdapter<T>` se utiliza con el layout de elementos de lista`android.R.layout.simple_list_item_1`. El widget `TextView`, que compone este layout, se referencia en el sistema invocando al método `toString()` de la clase T. Los datos se almacenan en un objeto tabla de <T> y se proveen al adaptador en el constructor.

Sintaxis

```
public ArrayAdapter (Context context, int textViewResourceId,
  T[] objects)
```

Ejemplo

```
ArrayAdapter<String> adaptatador =
  new ArrayAdapter<String>(this,
    android.R.layout.simple_list_item_1,
    new String[] { "Fila 1", "Fila 2", "Fila 3" });
```

Es posible modificar el comportamiento por defecto para mostrar más información para cada elemento. Para ello, es preciso sobrecargar el método `getView` de la clase `ArrayAdapter<T>`. Esta sobrecarga se estudia en el marco de la implementación específica de una lista.

Instanciar un CursorAdapter

La clase `CursorAdapter` es una clase abstracta: para poder instanciarla es preciso implementar los métodos `bindView (View view, Context context, Cursor cursor)` y`newView(Context context, Cursor cursor, ViewGroup parent)`.

El método `bindView` se invoca para vincular los datos que se quieren visualizar con la vista que los representa (*to bind*: vincular). Su implementación trata de obtener, a partir de la vista `view` que se pasa como parámetro, los widgets de representación (`TextView`, por ejemplo) utilizando el método que ya hemos estudiado `View.findViewById(int id)`, e informar el valor que se quiere mostrar a partir del cursor. Consulte el capítulo La persistencia de los datos, sección Bases de datos SQLite, para una representación de la clase `Cursor`.

El método `newView` debe devolver la vista que se utilizará como layout para cada elemento de la lista. Su implementación consiste, por tanto, en invocar al método `inflate()` a partir de un objeto de tipo `LayoutInflater`. Un `LayoutInflater` se obtiene a partir de un objeto de tipo `Context`.

A continuación se muestra un ejemplo de implementación. Observe que este ejemplo no está optimizado: sería conveniente, en un entorno de producción, evitar instanciar un objeto de tipo`LayoutInflater` con cada llamada al método `newView`. Al final de este capítulo se muestra un enfoque con un mejor rendimiento.

Ejemplo

```
CursorAdapter cursorAdapter = new CursorAdapter(this, miCursor) {
   @Override
   public View newView(Context context, Cursor cursor, ViewGroup
parent) {
      LayoutInflater mInflater = (LayoutInflater)
context.getSystemService(Context.LAYOUT_INFLATER_SERVICE);

      return mInflater.inflate(android.R.layout.simple_list_item_2,
parent);
   }
   @Override
   public void bindView(View view, Context context, Cursor cursor) {
      TextView text1 = (TextView)view.findViewById(android.R.id.text1);
      TextView text2 = (TextView)view.findViewById(android.R.id.text2);

      text1.setText(cursor.getString(0));
      text2.setText(cursor.getString(1));
   }
};
```

Instanciar un SimpleCursorAdapter

La clase `SimpleCursorAdapter` es una implementación de `CursorAdapter`. El vínculo entre los datos y los componentes encargados de mostrarlos se especifica en el constructor: además de los parámetros recibidos por la clase `CursorAdapter`, el constructor recibe como parámetro una tabla de `String` que indican el nombre de las columnas del cursor que se mostrarán y una tabla de valores enteros que se corresponden con los identificadores de los widgets utilizados para mostrar cada columna.

Sintaxis

```
SimpleCursorAdapter(Context context, int layout, Cursor c, String[]
from, int[] to)
```

Ejemplo

```
SimpleCursorAdapter simpleCursorAdapter =
   new SimpleCursorAdapter(this,
   android.R.layout.simple_list_item_2,
   miCursor,
   new String[] {"Nombre_Columna1", "Nombre_Columna2"},
   new int[] {android.R.id.text1, android.R.id.text2});
};
```

El ejemplo anterior instancia un `SimpleCursorAdapter` que muestra los datos Nombre_Columna1 y Nombre_Columna2 del cursor miCursor, utilizando un layout provisto por la plataforma que posee dos `TextView`.

Proveer el adaptador a la lista

La segunda etapa de la implementación se realiza invocando al método `setListAdapter` de la clase `ListActivity`, que recibe como parámetro el adaptador de tipo ListAdapter: este método realiza el vínculo entre la lista de la actividad y el adaptador definido antes.

Sintaxis

```
public void setListAdapter (ListAdapter adapter)
```

Ejemplo

```
public class MiActividadPrincipal extends ListActivity {
  private static final String[] ETIQUETAS = {
    "Fila 1", "Fila 2", "Fila 3", "Fila 4"
  };

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setListAdapter(new ArrayAdapter<String>(this,
      android.R.layout.simple_list_item_1, ETIQUETAS));
  }
}
```

2. Implementación específica

Incluso aunque se requiera el uso de un layout específico, es posible utilizar la claseListActivity, siempre que se respete una condición: en el layout de la actividad, el objetoListView debe tener como identificador el valor `@android:id/list`.

En el caso de que no pueda utilizarse la clase `ListActivity`, es posible vincular el adaptador a la lista invocando al método `setAdapter` de la clase `ListView`. La vista `ListView` se recupera de forma clásica, utilizando el método `findViewById` de la clase `Activity`.

Sintaxis

```
public setAdapter (ListAdapter adapter)
```

Ejemplo

```
public class MiActividadPrincipal extends Activity {

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

  ListView miLista =(ListView)findViewById(R.id.listView);
  //[...]: definición del adaptador

  miLista.setAdapter(miAdaptador);
  }
}
```

a. Layout de los elementos de la lista

El interés de una implementación específica es, principalmente, extender los layouts propuestos

para visualizar los elementos de una lista. El layout puede tomar cualquier forma deseada por el desarrollador, y se construye como cualquier otro layout de la plataforma. El archivo XML de layout debe ubicarse, como el resto de archivos de layout, en la carpeta /res/layout (y eventualmente en una de las carpetas dedicadas a la especialización). Se recomienda prefijarlo con un identificador explícito (`listitem_`, por ejemplo).

b. Adaptador

Sea cual sea el marco de uso de una `ListActivity` o de una `Activity`, una implementación específica de `ListView` provee un adaptador específico. Este adaptador, según la naturaleza de los datos que deba tratar, hereda de una de las clases de la plataforma que implementa`ListAdapter`: típicamente `ArrayAdapter<T>` o, si se desea una implementación más potente, `BaseAdapter`, con `ArrayAdapter<T>` como implementación de `BaseAdapter`.

La clase abstracta `BaseAdapter` presenta los siguientes métodos abstractos, que deben implementarse:

```
public View getView(int p, View convertView, ViewGroup parent)
```

Devuelve una vista que realiza la visualización del elemento ubicado en la posición p en el origen de datos. El parámetro `convertView` contiene, eventualmente, una antigua vista reciclada. Puede ser nula, si no existe ninguna vista reciclada disponible. El parámetro `parent` es la vista a la que la vista en curso puede vincularse, eventualmente.

```
public long getItemId(int p)
```

Devuelve el identificador del elemento que se muestra en la posición p.

```
public Object getItem(int p)
```

Devuelve el elemento en la posición p en el origen de datos.

```
public int getCount()
```

Devuelve el número de elementos en el origen de datos.

Si la sobrecarga de los métodos `getItemId`, `getItem` y `getCount` es sencilla, la sobrecarga del método `getView` resulta algo más compleja. Esta sobrecarga condiciona la correcta visualización de la lista, lo cual puede influir en el rendimiento de la aplicación.

La dificultad, en todo caso, radica en la gestión del parámetro `convertView`. La plataforma Android, para optimizar la memoria durante la visualización de una lista, recicla las vistas que ya no están visibles en la lista (cuando el usuario navega por la lista). Para obtener un mejor rendimiento, el desarrollador debe tener en cuenta esta optimización, y no generar una nueva vista salvo si `convertView` tiene valor nulo (es decir, en el caso de que no exista ninguna vista reciclada disponible).

Si `convertView` tiene valor nulo, el método `getView` debe instanciar una vista a partir del archivo XML de layout de su elección. Esta instanciación se realiza utilizando un objeto de tipo`LayoutInflater`, e invocando al método `inflate` de dicho objeto.

Sintaxis

```
public View inflate(int resource, ViewGroup root, boolean attachToRoot)
```

Se obtiene un objeto `LayoutInflater` a partir de una instancia de `Context`, invocando al método `getSystemService`.

Sintaxis

```
LayoutInflater layoutInflater =
context.getSystemService(Context.LAYOUT_INFLATER_SERVICE);
```

Cuando se instancia la vista, bien sea una vista reciclada u obtenida mediante un`LayoutInflater`, la siguiente etapa consiste en recuperar los widgets del layout que se utilizarán para visualizar los datos. Esta etapa se realiza, sencillamente, invocando al método`findViewById` de la vista instanciada.

A continuación, hay que asociar los valores a los componentes widgets. El elemento del origen de datos que se debe mostrar se obtiene invocando al método `getItem` de `BaseAdapter` (método que, por lo general, está sobrecargado).

La última etapa consiste en devolver la vista rellena.

A continuación se muestra un ejemplo de implementación del método `getView`. Es válida tanto para una sobrecarga de `BaseAdapter` como de `ArrayAdapter<T>` (en cuyo caso hay que remplazar en el código el tipo Ejemplo por el tipo representado por `<T>`).

```
@Override
public View getView(int position, View convertView, ViewGroup
parent) {
  View vista;
  if(convertView==null) {
     LayoutInflater inflater = (LayoutInflater)
getContext().getSystemService(Context.LAYOUT_INFLATER_SERVICE);
     vista = inflater.inflate(R.layout.listitem__especifico,
parent, false);
  }
  else
     vista = convertView;
  TextView titulo =
(TextView)vista.findViewById (R.id.listitem_Titulo);
  TextView descripcion =
(TextView)vista.findViewById(R.id.listitem_Descripcion);

  Ejemplo elemento = (Ejemplo)getItem(position);
  titulo.setText(elemento.getTitulo());
  descripcion.setText(elemento.getDescripcion());

  return vista;
}
```

Observe que es posible mejorar varios aspectos del ejemplo anterior para obtener un mejor rendimiento:

- Es preciso, obligatoriamente, invocar `getSystemService` con cada elemento de la lista. La instancia de `LayoutInflater` utilizada debe almacenarse en una variable externa al método.
- Es preferible no invocar a `findViewById` con cada elemento de la lista. Una posible optimización consiste en almacenar, mediante el método `setTag` de la vista, las instancias de los widgets utilizados, cuando la vista se instancia mediante el método `inflate`. Cuando la vista no es nula (se recicla), basta entonces con invocar al método `getTag` para recuperar las instancias salvaguardadas.

Ejemplo

```
static class SalvaguardaWidget {
   TextView titulo ;
   TextView descripcion ;
}
@Override
public View getView(int position, View convertView, ViewGroup
```

```
parent) {
   SalvaguardaWidget salvaguardaWidget;

   if(convertView==null) {
      LayoutInflater inflater =
(LayoutInflater)
getContext().getSystemService(Context.LAYOUT_INFLATER_SERVICE);
      convertView =
inflater.inflate(R.layout.listitem__especifico, parent, false);
      salvaguardaWidget = new SalvaguardaWidget ();
      salvaguardaWidget.titulo =
(TextView)convertView.findViewById(R.id.listitem_Titulo);
      salvaguardaWidget.descripcion =
(TextView)convertView.findViewById(R.id.listitem_Descripcion);
  convertView.setTag(salvaguardaWidget);
  }
  else
     salvaguardaWidget =(SalvaguardaWidget)convertView.getTag();

  Ejemplo elemento = (Ejemplo)getItem(position);
  salvaguardaWidget.titulo.setText(element.getTitulo());
  salvaguardaWidget.descripcion.setText(element.getDescripcion());
  return convertView;
}
```

3. Selección de un elemento

La clase `ListView` expone varios métodos que permiten capturar las acciones del usuario. Haremos referencia, aquí, al más utilizado, que permite capturar el clic sobre un elemento de la lista: el método `setOnItemClickListener`.

Sintaxis

```
public void setOnItemClickListener (AdapterView.OnItemClickListener listener)
```

El objeto que se pasa como parámetro es de tipo `AdapterView.OnItemClickListener`, que expone un método abstracto `onItemClick`. Este método, una vez sobrecargado, permite gestionar la acción posterior al clic del usuario.

Sintaxis

```
Void onItemClick(AdapterView<> parent, View view, int position, long id)
```

El método recibe como parámetro el identificador del elemento seleccionado por el usuario, id, que contiene el valor devuelto por el método `getItemId` de la clase `BaseAdapter`.

Ejemplo

```
miLista.setOnItemClickListener(new OnItemClickListener() {
```

Introducción

Este capítulo tiene como objetivo presentar la persistencia de los datos en Android.

Los datos persistentes de una aplicación son los datos salvaguardados antes del cierre de la aplicación de tal forma que puedan ser restaurados posteriormente.

Android proporciona varios mecanismos que permiten gestionar la persistencia de los datos, en función de la naturaleza de los mismos. Descubriremos los archivos de preferencias, los archivos estándar y las bases de datos.

Terminaremos viendo los proveedores de contenido que, más allá de la persistencia de los datos, proporcionan un mecanismo de compartición de datos entre las aplicaciones.

Archivos de preferencias

Android proporciona un framework simple para salvaguardar y restaurar los datos de tipos primitivos. Estos datos se salvaguardan en archivos con formato XML bajo la forma de pares clave-valor. Estos archivos se denominan archivos de preferencias.

Estudiaremos en primer lugar cómo preparar un archivo de preferencias, a continuación cómo leer y escribir datos en él. Terminaremos viendo cómo borrar todos los datos, o parte de ellos, de estos archivos.

El sistema Android permite visualizar y salvaguardar las preferencias generales del usuario. Toda aplicación puede adoptar la misma funcionalidad y la misma representación. La jerarquía de preferencias propuesta puede realizarse directamente en un archivo XML. La implementación de tal pantalla de preferencias se realiza derivando la clase `PreferenceActivity`. Desde Android 3.0 (API 11), esta clase funciona de forma conjunta con los fragmentos de tipo`PreferenceFragment` para poder, entre otros, mostrar de lado a lado los títulos de las secciones y de las preferencias que proponen.

1. Preparar el archivo

Por defecto, un archivo de preferencias está asociado a la actividad que lo crea. Este archivo lleva automáticamente el nombre completo de la actividad correspondiente, por ejemplo: `es.midominio.android.miaplicacion.prefsArch1.xml`.

La creación y la gestión del archivo de preferencias se realizan a través de un objeto de tipo`SharedPreferences` devuelto por el método `getPreferences` de la clase `Activity`.

Sintaxis

```
public SharedPreferences getPreferences (int mode)
```

Este método recibe como parámetro el modo de acceso que se quiere asignar al archivo tras su creación. Los valores disponibles para este parámetro son:

- `Context.MODE_PRIVATE`: modo privado. Es el modo por defecto. El archivo sólo puede leerse desde la aplicación en curso, o una aplicación que comparta el mismo identificador de usuario.
- `Context.MODE_WORLD_READABLE`: las demás aplicaciones pueden leer el archivo.
- `Context.MODE_WORLD_WRITEABLE`: las demás aplicaciones pueden modificar el archivo.

Ejemplo

```
SharedPreferences prefs = getPreferences(Context.MODE_PRIVATE);
```

También es posible especificar explícitamente otro nombre de archivo. Esto permite crear varios archivos de preferencias. Para ello, hay que utilizar el método `getSharedPreferences`especificando el nombre del archivo como primer parámetro.

Sintaxis

```
public abstract SharedPreferences getSharedPreferences (String name,
int mode)
```

Ejemplo

```
SharedPreferences prefs =
```

```
  getSharedPreferences("nombreArchivoPrefs1.xml",
    Context.MODE_PRIVATE);
```

2. Lectura

Los datos contenidos en un archivo de preferencias se almacenan bajo la forma de pares clave-valor. Tales asociaciones están compuestas:

- por una clave que es una cadena de caracteres de tipo `String`.
- por un valor de tipo primitivo: `boolean` (booleano), `float` (número de coma flotante), `int`o `long` (enteros) o `String` (cadena de caracteres).

Para leer los datos contenidos en un archivo de preferencias se utiliza el objeto de tipo`SharedPreferences` recuperado anteriormente. Se invoca a continuación a ciertos accesores que permiten leer individualmente un dato según su tipo.

Sintaxis

```
public abstract boolean getBoolean (String key, boolean defValue)
public abstract float getFloat (String key, float defValue)
public abstract int getInt (String key, int defValue)
public abstract long getLong (String key, long defValue)
public abstract String getString (String key, String defValue)
```

El primer parámetro es el nombre de la clave. El segundo parámetro es el valor por defecto que se quiere devolver si no existe la clave.

Ejemplo

```
boolean modoWifi = prefs.getBoolean("modoWifi", false);
int contador = prefs.getInt("contador", 0);
String comentario = prefs.getString("comentario", "");
```

También es posible recuperar todos los datos de golpe utilizando el método `getAll`.

Sintaxis

```
public abstract Map<String, ?> getAll ()
```

Ejemplo

```
Map<String, ?> valores = prefs.getAll();
Boolean modoWifi = (Boolean)valores.get("modoWifi");
```

El método `contains` del objeto `SharedPreferences` permite verificar la existencia de una clave dada que se le pasa como parámetro.

Sintaxis

```
public abstract boolean contains (String key)
```

Ejemplo

```
if (prefs.contains("modoWifi")) {
  procesamiento();
}
```

3. Escritura

La escritura de datos en un archivo de preferencias se realiza a través de un objeto

de tipo`SharedPreferences.Editor`. Este objeto lo devuelve el método `edit` invocado sobre el objeto de tipo `SharedPreferences` recuperado anteriormente.

Sintaxis

```
public abstract SharedPreferences.Editor edit ()
```

Ejemplo

```
SharedPreferences.Editor editor = prefs.edit();
```

El objeto `Editor` anterior permite especificar nuevos datos o modificar los datos existentes borrándolos con los nuevos. Estos métodos permiten escribir individualmente un par clave-valor. De forma similar a los métodos de lectura, existe un método de escritura por cada tipo primitivo. Estos métodos reciben como parámetro el nombre de la clave y el valor del dato.

Sintaxis

```
public abstract SharedPreferences.Editor putBoolean (String key,
  boolean value)
public abstract SharedPreferences.Editor putFloat (String key,
  float value)
public abstract SharedPreferences.Editor putInt (String key,
  int value)
public abstract SharedPreferences.Editor putLong (String key,
  long value)
public abstract SharedPreferences.Editor putString (String key,
  String value)
```

Ejemplo

```
editor.putBoolean("modoWifi", true);
editor.putInt("contador", 42);
editor.putString("comentario", "Esto es un comentario");
```

La escritura de los datos no será efectiva en el archivo hasta que se ejecute el método `commit` delobjeto `Editor`.

Sintaxis

```
public abstract boolean commit ()
```

Ejemplo

```
editor.commit();
```

Preste atención a no olvidarse de llamar al método `commit`. En efecto, si no realiza esta llamada, el objeto `Editor` no sirve de nada; no se registrarán las modificaciones que contiene.

4. Borrado

El borrado de los datos contenidos en un archivo de preferencias se hace utilizando el objeto`editor` de tipo `SharedPreferences.Editor` igual que para escribir los datos.

El método `remove` del objeto `Editor` permite suprimir un par clave-valor. Se pasa el nombre de la clave como parámetro.

Sintaxis

```
public abstract SharedPreferences.Editor remove (String key)
```

Ejemplo

```
editor.remove("modoWifi");
```

El método `clear` permite borrar todos los datos, es decir, todos los pares clave-valor.

Sintaxis

```
public abstract SharedPreferences.Editor clear ()
```

Ejemplo

```
editor.clear();
```

Como ocurre con la escritura, es necesario invocar al método `commit` para registrar los cambios.

También es posible encadenar las modificaciones, puesto que los métodos del objeto `Editor`devuelven este mismo objeto.

Ejemplo

```
editor.clear().putBoolean("modoWifi", modoWifi).commit();
```

Cuando se invoca al método `commit`, el método `clear` se ejecuta en primer lugar, sea cual sea la posición de su llamada.

Es posible, por ejemplo, reescribir la línea anterior sin modificar el resultado.

Ejemplo

```
editor.putBoolean("modoWifi", modoWifi).clear().commit();
```

Archivos

Como acabamos de ver en la sección anterior, los archivos de preferencias son la solución ideal para salvaguardar valores de tipos primitivos de manera simple. Pero si lo que se quiere es almacenar datos más complejos o datos brutos sin un formato adaptado al formato XML como, por ejemplo, la copia de una imagen con formato PNG, esta solución no es adecuada.

Necesitamos poder crear, escribir y leer archivos directamente.

Android permite guardar archivos en el almacenamiento interno del dispositivo o en un almacenamiento externo como, por ejemplo, una tarjeta SD. Proporciona también las API para guardar archivos temporales o archivos de caché en ubicaciones definidas.

Vamos, por tanto, a ver en primer lugar la gestión de archivos en el almacenamiento interno, y en segundo lugar sobre un almacenamiento externo. Por último, veremos la gestión de los archivos temporales.

1. Almacenamiento interno

Por defecto, los archivos se almacenan en el almacenamiento interno del dispositivo. El acceso a estos archivos está restringido a la aplicación. Ni el usuario ni las demás aplicaciones pueden acceder a él.

Vamos a ver cómo crear un archivo en el almacenamiento interno, cómo leerlo y, por último, cómo borrarlo.

a. Escritura

La creación de un archivo se realiza invocando al método `openFileOutput` de la clase`Context`. Este método espera recibir dos parámetros: el nombre del archivo sin arborescencia y el modo de acceso que se quiere asignar a este archivo. El método devuelve un flujo de tipo`FileOutputStream`.

Sintaxis

```
public abstract FileOutputStream openFileOutput (String name,
  int mode)
```

Ejemplo

```
FileOutputStream flujo = openFileOutput("miArchivo.png",
  Context.MODE_PRIVATE);
```

El flujo devuelto permite escribir los datos brutos en el archivo utilizando los distintos métodos`write`.

Sintaxis

```
public void write (byte[] buffer)
public void write (int oneByte)
public void write (byte[] buffer, int offset, int count)
```

Ejemplo

```
flujo.write(contenido.getBytes());
```

Sólo queda cerrar el flujo para escribir definitivamente el archivo sobre el soporte invocando al método `close`.

Sintaxis

```
public void close ()
```

Ejemplo

```
flujo.close();
```

b. Lectura

Para poder leer los datos contenidos en un archivo, hay que abrirlo en modo lectura. Para ello, podemos utilizar el método `openFileInput` de la clase `Context`. Este método recibe como parámetro el nombre del archivo sin arborescencia y devuelve un flujo de tipo`FileInputStream`.

Sintaxis

```
public abstract FileInputStream openFileInput (String name)
```

Ejemplo

```
intnb = FileInputStream flujo = openFileInput("miArchivo.png");
```

El flujo devuelto permite leer los datos del archivo utilizando los distintos métodos `read`. Estos métodos devuelven el número de bytes leídos o -1 si se ha alcanzado el final del archivo.

Sintaxis

```
public int read ()
public int read (byte[] buffer)
public int read (byte[] buffer, int offset, int count)
```

Ejemplo

```
flujo.read();
```

Sólo queda el flujo invocando al método `close`.

Sintaxis

```
public void close()
```

Ejemplo

```
flujo.close();
```

c. Eliminar un archivo

El método `deleteFile` de la clase `Context` permite eliminar un archivo. Hay que pasarle el nombre del archivo como parámetro. Devuelve `true` si el archivo se ha eliminado correctamente, o`false` en caso contrario.

Sintaxis

```
public abstract boolean deleteFile (String name)
```

Ejemplo

```
boolean exito = deleteFile("miArchivo.png");
```

2. Almacenamiento externo

Además del almacenamiento interno, los dispositivos Android proporcionan un almacenamiento externo. Los archivos almacenados en este soporte son públicos. El usuario y todas las demás aplicaciones pueden, por tanto, leerlos, modificarlos y suprimirlos en cualquier momento.

Al contrario de lo que su propio nombre pudiera indicar, el almacenamiento externo puede ser interno al dispositivo. También puede ser extraíble como, por ejemplo, una tarjeta SD. En este caso, el usuario puede retirarlo en cualquier momento. La aplicación debe, por tanto, tenerlo en cuenta durante su etapa de diseño, y prestar atención en particular a cómo se quiere gestionar la ausencia del soporte con cada intento de escritura o de lectura.

Vamos a ver en primer lugar cómo verificar la presencia y la disponibilidad del almacenamiento externo. Veremos, a continuación, dónde se almacenan los archivos. Por último, veremos la existencia de una arborescencia común a todas las aplicaciones que permite compartir archivos los que contiene.

a. Disponibilidad del soporte

Para asegurar la disponibilidad del soporte de almacenamiento externo, hay que invocar al método`getExternalStorageState` de la clase `Environment`.

Sintaxis

```
public static String getExternalStorageState ()
```

Ejemplo

```
String estadoSoporteExterno = Environment.getExternalStorageState();
```

Este método devuelve una cadena de caracteres que indica la disponibilidad del soporte de almacenamiento, que habrá que comparar con los estados predefinidos.

Ejemplo

```
if (Environment.MEDIA_MOUNTED.equals(estadoSoporteExterno)) {
  procesamiento();
}
```

La siguiente tabla muestra las constantes declaradas en la clase `Environment` que representan los distintos estados de disponibilidad.

Estado	Descripción
`MEDIA_BAD_REMOVAL`	El soporte se ha retirado antes de haberse desmontado por completo.
`MEDIA_CHECKING`	El soporte está presente, en estado de verificarse.
`MEDIA_MOUNTED`	El soporte está presente, montado con permisos de lectura/escritura.
`MEDIA_MOUNTED_READ_ONLY`	El soporte está presente, montado con permisos de sólo lectura.
`MEDIA_NOFS`	El soporte está presente, pero el formato del sistema de archivos no está soportado.
`MEDIA_REMOVED`	El soporte no está presente.
`MEDIA_SHARED`	El soporte está presente, no montado y

	compartido como periférico de almacenamiento USB.
MEDIA_UNMOUNTABLE	El soporte está presente, pero no se ha podido montar.
MEDIA_UNMOUNTED	El soporte está presente, pero no está montado.

b. Accesos y ubicaciones

Cada aplicación tiene su carpeta en el soporte externo. La ruta es:

```
/Android/data/paquete_aplicación/files/
```

Los archivos de datos se reparten, a continuación, en diversas subcarpetas según su tipo. He aquí la lista de subcarpetas:

Carpeta	Constante	Descripción
Music/	DIRECTORY_MUSIC	Contiene archivos de música.
Podcasts/	DIRECTORY_PODCASTS	Contiene podcasts.
Ringtones/	DIRECTORY_RINGTONES	Contiene tonos de llamada.
Alarms/	DIRECTORY_ALARMS	Contiene sonidos de alarma.
Notifications/	DIRECTORY_NOTIFICATIONS	Contiene sonidos de notificaciones.
Pictures/	DIRECTORY_PICTURES	Contiene fotografías.
Movies/	DIRECTORY_MOVIES	Contiene vídeos.
Download/	DIRECTORY_DOWNLOADS	Contiene otros tipos de archivo.

El método `getExternalFilesDir` permite obtener un objeto de tipo `File` que representa la subcarpeta correspondiente al tipo de dato a salvaguardar.

Este método, y de forma global todas las operaciones de escritura sobre el almacenamiento externo, requieren permisos de escritura sobre este almacenamiento. Para obtenerlo, es necesario incluir la siguiente fila en el archivo manifiesto (véase el capítulo Concurrencia, seguridad y red - Seguridad y permisos) :

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

Sintaxis

```
public abstract File getExternalFilesDir(String type)
```

El parámetro indica el tipo de dato que se quiere salvaguardar. Por ejemplo, para un archivo de música, se indicará:

```
Environment.DIRECTORY_MUSIC
```

Para obtener la carpeta raíz, hay que pasar `null` como parámetro.

La carpeta raíz, así como todas las subcarpetas y sus datos, se suprimirán tras desinstalar la aplicación.

c. Archivos comunes

Android pone a disposición una arborescencia común a todas las aplicaciones sobre el almacenamiento externo. Esta arborescencia permite compartir archivos entre todas las aplicaciones. Los archivos creados en esta arborescencia no se suprimirán tras desinstalar la aplicación que los haya creado.

La arborescencia de las subcarpetas es similar a la arborescencia privada de cada aplicación. La carpeta raíz es la carpeta raíz del soporte de almacenamiento externo.

El método `Environment.getExternalStoragePublicDirectory` permite acceder a estas subcarpetas.

Sintaxis

```
public static File getExternalStoragePublicDirectory(String type)
```

El parámetro indica el tipo de dato que se quiere salvaguardar. Por ejemplo, para un archivo de música, se indicará:

```
Environment.DIRECTORY_MUSIC
```

3. Archivos temporales

Una aplicación puede necesitar archivos temporales o archivos de caché. Cada aplicación tiene reservada una ubicación específica para almacenar estos archivos tanto en el almacenamiento interno como en el almacenamiento externo en el caso de que exista. El uso de uno u otro espacio de almacenamiento dependerá del tamaño de estos archivos; la capacidad de almacenamiento externo suele ser más conveniente que la del almacenamiento interno.

Vamos a ver cómo recuperar estas carpetas tanto en el almacenamiento interno como en el almacenamiento externo.

a. Almacenamiento interno

La llamada al método `getCacheDir` de la clase `Context` devuelve la carpeta en la que la aplicación debe almacenar sus archivos temporales sobre el almacenamiento interno.

Sintaxis

```
public abstract File getCacheDir()
```

El sistema puede suprimir estos archivos en cualquier momento, en particular cuando no existe más espacio en el almacenamiento. No obstante esto no es obligatorio. Por ello, la aplicación debe velar por que estos archivos no ocupen demasiado espacio, más de un megabyte en total. Y debería borrarlos una vez ya no fueran necesarios.

Si la aplicación requiere más espacio para sus archivos temporales, podrá utilizar el almacenamiento externo.

Todo el contenido de la carpeta de caché de la aplicación se borrará tras la desinstalación de la aplicación.

b. Almacenamiento externo

Además de la ubicación de caché específica a la aplicación sobre el almacenamiento interno, el

sistema proporciona también una ubicación de caché específica a cada aplicación en el almacenamiento externo.

Esta ubicación permite a una aplicación crear archivos temporales de gran tamaño.

La llamada al método `getExternalCacheDir` de la clase `Context` devuelve esta carpeta.

De forma similar al método `getExternalFilesDir`, este método requiere permisos de escritura sobre el almacenamiento externo`android.permission.WRITE_EXTERNAL_STORAGE`.

Sintaxis

```
public abstract File getExternalCacheDir()
```

La aplicación debe suprimir sus archivos de caché cuando ya no los necesite para liberar espacio.

Todo el contenido de la carpeta de caché de la aplicación se borrará tras la desinstalación de la aplicación.

Bases de datos SQLite

Una aplicación puede necesitar una base de datos para almacenar y realizar consultas sobre sus datos. Android permite crear bases de datos con formato SQLite.

Una aplicación puede crear varias bases de datos. Estas bases de datos son privadas a la aplicación; solamente ella tiene acceso.

Android no proporciona un framework de alto nivel de mapping objeto-relacional. Es preciso utilizar directamente consultas SQL para crear, gestionar y ejecutar consultas sobre las bases de datos.

En esta sección, veremos cómo crear una base de datos y agregar una tabla. A continuación veremos cómo ejecutar consultas. Por último, veremos cómo modificar una base de datos existente.

1. Creación de una base de datos

Android proporciona la clase abstracta `SQLiteOpenHelper` que permite gestionar la creación y la actualización de las bases de datos.

Esta clase es abstracta, de modo que hay que crear una clase derivada que herede de ella. Esta clase derivada debe invocar al constructor padre y pasarle como parámetros el nombre y la versión de la base de datos. El número de versión se utiliza durante la actualización de la base de datos a una nueva versión como veremos más adelante.

Heredando de la clase padre `SQLiteOpenHelper`, la clase derivada debe sobrecargar, entre otros, el método `onCreate`. Este método permite especificar las consultas de creación de las tablas de la base de datos.

Sintaxis

```
public abstract void onCreate (SQLiteDatabase db)
```

Ejemplo

```
public class BDDAssistant extends SQLiteOpenHelper {
  private static final int VERSION_BDD = 1;
  private static final String NOMBRE_BDD = "miBDD";

  public BDDAssistant(Context context) {
    super(context, NOMBRE_BDD, null, VERSION_BDD);
  }

  @Override
  public void onCreate(SQLiteDatabase db) {
  db.execSQL("CREATE TABLE miTabla ( _id INTEGER PRIMARY KEY
AUTOINCREMENT, nombre TEXT );");
  }
}
```

La clase `SQLiteOpenHelper` proporciona el método `getWritableDatabase` para crear y abrir la base de datos. Esta clase devuelve un objeto de tipo `SQLiteDatabase` accesible en modo de escritura que permite modificar la base de datos.

Sintaxis

```
public synchronized SQLiteDatabase getWritableDatabase ()
```

Una vez terminados la creación y el uso de la base de datos, es preciso cerrarla invocando a

losmétodos `close` en el objeto de tipo `SQLiteDatabase` y, a continuación, en el objeto de tipo`SQLiteOpenHelper`.

Sintaxis

```
public void close ()
```

Ejemplo

```
BDDAssistant bddAss = new BDDAssistant(this);
SQLiteDatabase bdd = bddAss.getWritableDatabase();
procesamiento(bdd);
bdd.close();
bddAss.close();
```

La llamada a `getWritableDatabase` puede devolver un objeto en modo sólo lectura si existe un problema como por ejemplo un disco lleno. Una vez corregido el problema, una nueva llamada al método `getWritableDatabase` cerrará el objeto en modo sólo lectura y devolverá un objeto accesible en lectura/escritura.

Es tras la primera llamada al método `getWritableDatabase` cuando la base de datos se crea realmente. Este método ejecutará sucesivamente los método `onCreate`, `onUpgrade` y`onOpen` de la instancia `bddAss`. Una vez abierta la base de datos, se pone en caché de modo que las sucesivas llamadas a este método devuelven directamente la base de datos sin invocar de nuevo a los métodos anteriores.

Para abrir una base de datos en modo sólo lectura, se reemplazará la llamada al método`getWritableDatabase` por el método `getReadableDatabase`.

Sintaxis

```
public synchronized SQLiteDatabase getReadableDatabase ()
```

Ejemplo

```
SQLiteDatabase bdd = bddAss.getReadableDatabase();
```

2. Procedimientos y consultas SQL

El objeto de tipo `SQLiteDatabase` recuperado en la sección anterior permite ejecutar sentencias y consultas SQL: `CREATE TABLE`, `DELETE`, `INSERT`… La ejecución de sentencias SQL la realizan los métodos `execSQL`.

Sintaxis

```
public void execSQL (String sql)
public void execSQL (String sql, Object[] bindArgs)
```

Ejemplo

```
bdd.execSQL("DROP TABLE IF EXISTS miTabla");
```

Sólo está permitido una sentencia por llamada a estos métodos. El uso de ';' para separar las sentencias está, por tanto, prohibido.

Las consultas SQL pueden ejecutarse con los distintos métodos `query` que reciben distintos parámetros de entrada como el nombre de la tabla, los nombres de los campos…

Cada uno de estos métodos devuelve un objeto de tipo `Cursor` que permite navegar por los resultados de la consulta y extraer los datos. El cursor devuelto por el método query se posiciona**antes** del primer registro.

Sintaxis

```
public Cursor query (String table, String[] columns,
  String selection, String[] selectionArgs,String groupBy,
  String having, String orderBy)

public Cursor query (String table, String[] columns,
  String selection, String[] selectionArgs,String groupBy,
  String having, String orderBy, String limit)

public Cursor query (boolean distinct, String table,
  String[] columns, String selection,String[] selectionArgs,
  String groupBy, String having, String orderBy, String limit)
```

Ejemplo

```
Cursor cursor = bdd.query("miTabla", new String[] { "_id", "nombre" },
null, null, null, null, "_id desc", 10);
```

a. Navegar los resultados

La interfaz `Cursor` provee todos los métodos que permiten navegar un juego de resultados de una consulta. El acceso a los registros puede realizarse de forma secuencial (método`moveToNext`) o directamente indicando un número de registro (método `moveToPosition`).

Observe que no se debe olvidar invocar al método `moveToFirst` antes de recorrer secuencialmente una colección de registros, pues el método `query` posiciona el cursor antes del primer registro.

Los principales métodos utilizados para recorrer un juego de registros se enumeran en la tabla a continuación.

Nombre del método	Acción
`int getCount()`	Devuelve el número de registros contenidos en el cursor.
`boolean moveToFirst()`	Posiciona el cursor sobre el primer registro.
`boolean moveToNext()`	Posiciona el cursor sobre el registro siguiente. Si el cursor ya estuviera posicionado en el último registro, el método devolverá `false`.
`boolean isLast()`	Devuelve `true` si el registro en curso es el último, o `false` en caso contrario.
`boolean isFirst()`	Devuelve `true` si el registro en curso es el primero, o `false` en caso contrario.
`boolean moveToPosition (int posicion)`	Posiciona el cursor en la posición que se indica como parámetro. El primer registro se encuentra en la posición 0. El rango de valores aceptados por el parámetro `posicion` es -1<= posicion <=count, o desde la posición`isBeforeFirst()` hasta la posición`isAfterLast()`.

b. Lectura de datos

Para leer los datos de un registro - las columnas del resultado de una consulta SQL, la interfaz`Cursor` expone tantos métodos como tipos de datos posibles existen en una base de datos SQLite.

En estos métodos, la columna que se quiere leer se indica como parámetro, mediante su índice. La primera columna tiene el índice 0, y la última columna tiene el índice `getColumnCount() - 1`.

A continuación se muestra una tabla resumen de los métodos más utilizados para la lectura de un registro.

Nombre del método	Acción
`int getColumnCount()`	Devuelve el número de columnas para el registro.
`int getColumnIndex(String nombreColumna)`	Devuelve el índice de la columna cuyo nombre se pasa como parámetro.

`double getDouble(int indiceColumna)`
`float getFloat(int indiceColumna)`
`long getLong(int indiceColumna)`
`int getInt (int indiceColumna)`
`short getShort(int indiceColumna)`
`String getString(int indiceColumna)`
`byte[] getBlob(int indiceColumna)`
	Devuelve el valor de la columna cuyo índice se pasa como parámetro. Produce una excepción si el dato no es del tipo indicado, o si el valor es `null`.

`boolean isNull(indexindiceColumna)`	Devuelve `true` si el valor de la columna es`null`.

3. Actualizaciones

Tras la creación de una nueva versión de la aplicación, es posible que la base de datos necesite evolucionar, creando por ejemplo nuevas tablas, o modificando tablas existentes. Es preciso, en estos casos, realizar una actualización de la base de datos.

Android integra un mecanismo específico que ayuda a actualizar la base de datos: en primer lugar se invoca a uno de los métodos `getReadableDataBase` y `getWritableDataBase`, el sistema compara el número de versión en curso (que se pasa como parámetro en la llamada del constructor padre `SQLiteOpenHelper`) con el valor devuelto por el método `getVersion`. Si ambos números de versión difieren, se invoca al método `onUpgrade`.

Basta, por tanto, con sobrecargar el método `onUpgrade` cuya sintaxis es la siguiente:

Sintaxis

```
public abstract void onUpgrade (SQLiteDatabase db, int oldVersion,
  int newVersion)
```

Ejemplo

```
public class BDDAssistant extends SQLiteOpenHelper {
  private static final int VERSION_BDD = 2;
  ...
public BDDAssistant(Context context) {
   super(context, NOMBRE_BDD, null, VERSION_BDD);
  }
```

```
  @Override
  public void onUpgrade(SQLiteDatabase db, int oldVersion,
      int newVersion) {
          ...
  }
}
```

Los usuarios no actualizan, necesariamente, las aplicaciones instaladas en su dispositivo con cada nueva publicación, por lo que es importante gestionar, mediante el método `onUpgrade`, las posibles actualizaciones desde cualquier número de versión.

Una de las formas de gestionar estas actualizaciones incrementales es escribir un bucle que vaya desde `oldVersion` hasta `newVersion`, y gestionar cada etapa con la ayuda de una instrucción`switch`. Un posible esquema del método `onUpgrade` podría ser el siguiente:

```
@Override
public void onUpgrade(SQLiteDatabase db, int oldVersion, int
newVersion) {
   for(int indiceVersion =oldVersion ; indiceVersion <newVersion;
indiceVersion ++) {
      int nextVersion = indiceVersion+1;
      switch(nextVersion) {
         case 2 :
            // actualización para la versión 2
            break;
         case 3 :
            // actualización para la versión 3
            break;
         ...
      }
}
```

Por último, para asegurar que la base de datos se mantiene en un estado conocido en caso de error de actualización, se recomienda ejecutar el conjunto de instrucciones de actualización dentro de la misma transacción.

Una transacción se inicia mediante el método `beginTransaction` del objeto `SQLiteDatabase`. En caso de éxito, una llamada al método `setTransactionSuccessful` del mismo objeto permitirá indicar al sistema que las transacciones pueden confirmarse. El método`endTransaction` finaliza la transacción: si `setTransactionSuccessful` se hubiera invocado anteriormente, los cambios se aplicarán (acción commit). En caso contrario, no se aplicará ningún cambio a la base de datos (acción rollback).

La llamada al método `onUpgrade` se realiza tras la ejecución de los métodos`getReadableDatase` y `getWritableDatabase`. El tiempo de actualización puede ser largo. En este caso, es preciso lanzar esta operación desde un thread distinto al thread principal (véase el capítulo Concurrencia, seguridad y red - Programación concurrente).

Proveedor de contenidos

Los proveedores de contenidos permiten compartir públicamente datos entre todas las aplicaciones. Es el caso, por ejemplo, de los datos relacionados con los contactos personales que pueden compartirse entre todas las aplicaciones. Cada aplicación puede, si tiene los permisos necesarios, leer, agregar o modificar los contactos personales.

Existen, por defecto, varios proveedores de contenidos para los datos de audio, de vídeo, las imágenes, los contactos personales… Podemos utilizarlos tal cual para almacenar nuevos registros. Esto permite no tener que crear nuevos.

Vamos a describir la interfaz común a los proveedores de contenidos, y después veremos cómo realizar consultas y modificaciones sobre estos proveedores de contenidos. Por último, veremos cómo borrar registros en los proveedores de contenidos.

1. Interfaz y URI

Todos los proveedores de contenidos exponen la misma interfaz. De forma interna, no obstante, cada uno es libre de salvaguardar sus datos utilizando la o las soluciones de almacenamiento que estime oportunas.

Sea cual sea el modo de almacenamiento interno, los proveedores de contenidos devuelven sus datos bajo la forma de tablas de base de datos. Cada fila es un registro y cada columna un valor correspondiente al campo asociado.

Cada registro posee un campo numérico `_ID` que identifica de forma única el registro en la tabla.

Cada proveedor de contenidos provee una URI única que corresponde con una tabla de sus datos. Habrá, por tanto, tantas URI como tablas de datos.

Sintaxis de la URI

```
content://es.midominio.nombreProveedor
content://es.midominio.nombreProveedor/id
content://es.midominio.nombreProveedor/ruta
content://es.midominio.nombreProveedor/ruta/id
```

Fragmento	Descripción
`content://`	Indica que los datos están controlados por un proveedor de contenidos.
`es.midominio.nombreProveedor`	Fragmento autorizado de la URI.
`ruta`	Permite al proveedor de contenidos determinar el tipo de dato. Está vacío si gestiona un único tipo de dato.
`id`	El identificador único del registro solicitado, es decir, el valor del campo `_ID`. Está vacío si la consulta devuelve más de un registro.

Android proporciona URI para los proveedores de contenidos incluidos en el sistema.

Ejemplo de URI para obtener los datos de los contactos

```
ContactsContract.Contacts.CONTENT_URI
```

El acceso a ciertos datos requiere algunos permisos. Aquí, se requiere el permiso`android.permission.READ_CONTACTS` para poder acceder a los datos de los

contactos. A su vez, se requiere el permiso `android.permission.WRITE_CONTACTS` para poder agregar, modificar o borrar datos de los contactos.

2. Consultas

La clase `ContentResolver` contiene los métodos que permiten realizar consultas para recuperar los datos almacenados en los proveedores de contenidos. El método `getContentResolver` de la clase `Context` permite recuperar una instancia de tipo `ContentResolver`.

Sintaxis

```
public abstract ContentResolver getContentResolver ()
```

Ejemplo

```
ContentResolver resolver = getContentResolver();
```

El método `query` de la clase `ContentResolver` permite ejecutar una consulta pasándole la URI como parámetro. Los demás parámetros permiten precisar los nombres de los campos que se desea extraer, una cláusula SQL `WHERE`, los argumentos de la cláusula `WHERE` y una cláusula SQL `ORDER BY`.

Sintaxis

```
public final Cursor query (Uri uri, String[] projection,
  String selection, String[] selectionArgs, String sortOrder)
```

Ejemplo

```
Cursor cursor = resolver.query(
  ContactsContract.Contacts.CONTENT_URI, null, null, null, null);
```

Para una mayor comodidad, en las versiones de Android inferiores a la 3.0 (API 11), es posible utilizar el método `managedQuery` de la clase `Activity` que, a diferencia del método anterior, gestiona el ciclo de vida del cursor automáticamente en función del estado de la actividad que la contiene. Por ejemplo, cuando la actividad está en pausa, el cursor libera la memoria usada y se recarga automáticamente cuando se retoma la actividad.

Sintaxis

```
public final Cursor managedQuery (Uri uri, String[] projection,
  String selection, String[] selectionArgs, String sortOrder)
```

Ejemplo

```
Cursor cursor = managedQuery(
  ContactsContract.Contacts.CONTENT_URI, null, null, null, null);
```

Desde la versión 3.0 (API 11), el método `managedQuery` se ha deprecado en beneficio de la nueva clase `CursorLoader` que permite ejecutar fácilmente consultas de forma asíncrona.

Estas llamadas devuelven un cursor que apunta sobre todos los registros del resultado. Para devolver un único registro, es preciso agregar el identificador único del registro solicitado a la URI. Para ello, se utiliza el método `withAppendedId` de la clase `ContentUris` que devuelve la URI así formada.

Sintaxis

```
public static Uri withAppendedId (Uri contentUri, long id)
```

Ejemplo

```
Uri uri = ContentUris.withAppendedId(
  ContactsContract.Contacts.CONTENT_URI, 42);
Cursor cursor = resolver.query(uri, null, null, null, null);
```

El cursor obtenido permite leer los valores de los registros. Para ello, la clase `Cursor` provee, entre otros, el método `getColumnIndex`. Este método recibe como parámetro el nombre del campo solicitado y devuelve el índice correspondiente.

Sintaxis

```
public abstract int getColumnIndex (String columnName)
```

Conociendo el tipo de valor del campo, es posible recuperar el valor utilizando el método correspondiente de entre los métodos `getInt`, `getFloat`... Por ejemplo, para recuperar un valor de tipo `String`, se utilizará el método `getString`.

Sintaxis

```
public abstract String getString (int columnIndex)
```

Ejemplo

```
int indiceCol =

 cursor.getColumnIndex(ContactsContract.Contacts.DISPLAY_NAME);
String nombre = cursor.getString(indiceCol);
```

3. Agregar un registro

Para agregar nuevos registros a un proveedor de contenidos, hay que crear un objeto de tipo`ContentValues` que contendrá los datos del registro bajo la forma de pares clave-valor. Para agregar datos en este objeto se utiliza uno de sus métodos `put` que reciben como primer parámetro el nombre de la clave y como segundo parámetro el valor. Por ejemplo, el método `put`correspondiente a un valor de tipo `String` posee la sintaxis siguiente:

Sintaxis

```
public void put (String key, String value)
```

Ejemplo

```
ContentValues registro = new ContentValues();
registro.put(Phone.RAW_CONTACT_ID, 42);
registro.put(Phone.NUMBER, "000-000");
registro.put(Phone.TYPE, Phone.TYPE_MOBILE);
```

Para agregar el registro en los datos del proveedor de contenidos, basta con invocar a continuación al método `insert` del objeto de tipo `ContentResolver` y pasarle como parámetro la URI y el registro. Este método devuelve la URI correspondiente al registro.

Sintaxis

```
public final Uri insert (Uri url, ContentValues values)
```

Ejemplo

```
Uri uri = resolver.insert(Phone.CONTENT_URI, registro);
```

4. Borrado de registros

El borrado de uno o varios registros se realiza invocando al método `delete` de la clase`ContentResolver`.

Sintaxis

```
public final int delete (Uri url, String where,
  String[] selectionArgs)
```

Si el parámetro `url` contiene el identificador de un registro, entonces sólo se borrará este registro.

Si, por el contrario, el parámetro `url` no contiene el identificador, entonces se utilizarán los parámetros `where` y `selectionArgs` para especificar la cláusula SQL `WHERE` y sus argumentos.

Ejemplo

```
Uri uri = ContentUris.withAppendedId(
  ContactsContract.Contacts.CONTENT_URI, 42);
resolver.delete(uri, null, null);
```

Copia de seguridad en la nube

Desde la versión 2.2 (API 8), Android permite realizar la copia de seguridad de los datos persistentes de la aplicación en la nube, es decir, en línea, sobre un servidor. Se trata de un servicio complementario al de la copia de seguridad de los datos en local.

Cuando el usuario reinicializa el dispositivo Android, cuando cambia de dispositivo o utiliza un nuevo dispositivo suplementario, como por ejemplo una tableta como complemento a un smartphone, puede solicitar (re)instalar una aplicación. El hecho de tener que reconfigurar la aplicación y volver a insertar ciertos datos de forma manual son tareas que pueden resultar poco atractivas al usuario.

Es, en este momento, cuando entra en juego la copia de seguridad en la nube. Ésta permite evitar tareas de configuración largas y tediosas. Es posible descargar e insertar los datos, salvaguardados previamente en línea, durante una nueva instalación. El usuario encontrará automáticamente la aplicación y los datos que la acompañan sin tener que hacer nada por su parte.

Los datos están asociados a la cuenta de Google del usuario principal configurado en el dispositivo. El usuario tendrá que configurar a continuación la misma cuenta principal sobre sus demás dispositivos para aprovechar plenamente esta funcionalidad.

Cabe observar que, si bien esta funcionalidad está presente en la mayoría de dispositivos Android, algunos no la poseen. También puede darse el caso de que el usuario desactive esta funcionalidad en los parámetros generales del sistema. En tales casos, la aplicación funcionará de manera normal; simplemente los datos persistentes no se salvaguardarán ni se recuperarán. Será tarea del usuario reinsertarlos manualmente.

La aplicación no envía, en sí misma, los datos persistentes al servidor. Se comunica con el Gestor de copia de seguridad sobre el sistema Android. Éste se encargará de preparar los datos a enviar, comunicarse con el servidor en línea y transmitirle los datos. También se encarga de recibir los datos desde el servidor en el caso de una nueva instalación de la aplicación.

La aplicación debe indicar al Gestor de copia de seguridad que quiere utilizar el servicio de copia de seguridad en línea, dado que el sistema permite utilizar distintos servicios de copia de seguridad en línea. Estudiaremos aquí el servicio de copia de seguridad propuesto por Google: Android Backup Service.

Corresponde a los responsables de la aplicación, sea un particular, una empresa u otros verificar si deben declarar la copia de seguridad de los datos sobre sus servidores, según el tipo de dato afectado y, en especial, si se trata de datos personales, debido a la LOPD (Agencia Española de Protección de Datos - Ley Orgánica de Protección de Datos - https://www.agpd.es/). Si usa el servicio Android Backup Service de Google, el responsable deberá, sin duda, completar su declaración indicando una posible transferencia de datos fuera de la Unión Europea.

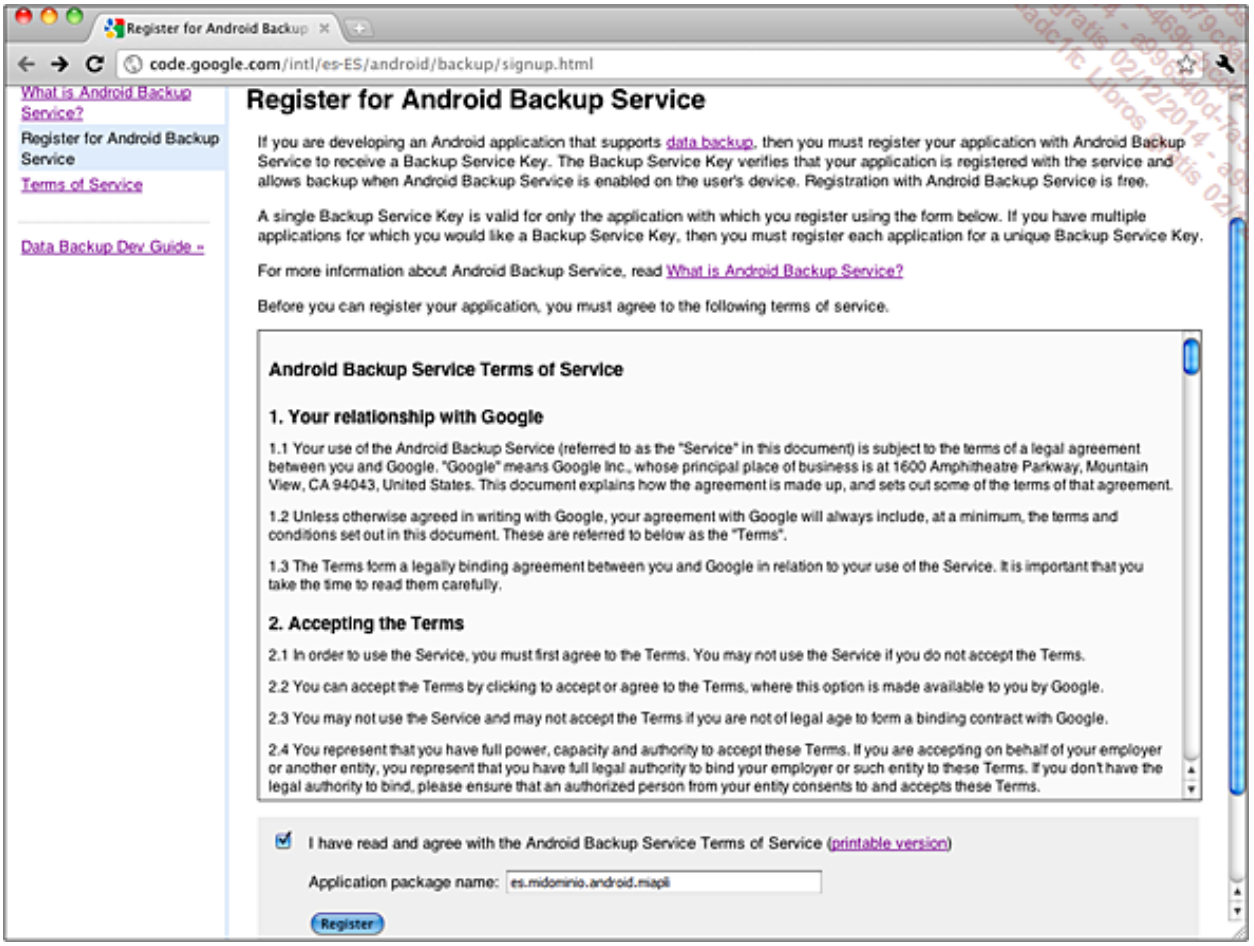
1. Suscribirse a Android Backup Service

La inscripción de la aplicación en el servicio de Android Backup Service es una etapa obligatoria para que pueda salvaguardar y restaurar los datos. La inscripción es gratuita.

En la etapa de inscripción, se genera una clave única. Esta clave permite identificar la aplicación desde el servicio en las fases de copia de seguridad y de restauración.

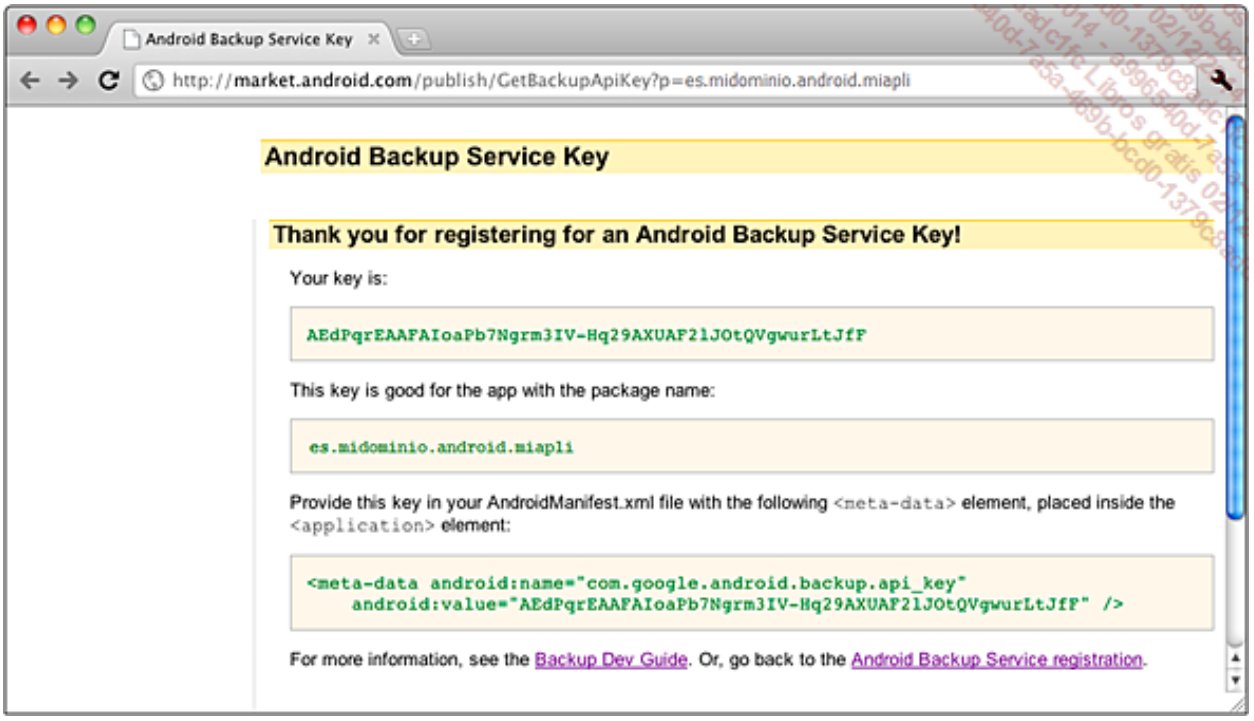
El servicio requiere que cada aplicación posea su propia clave. Será preciso, por tanto, inscribir cada aplicación, cada paquete, para poder generar tantas claves como aplicaciones quieran utilizar el servicio.

- Diríjase a la siguiente dirección: http://developer.android.com/google/backup/signup.html
- Lea las condiciones de uso del servicio Android Backup Service que describen, en particular, las condiciones y las restricciones de uso del servicio. Marque la opción **I have read and agree with the Android Backup Service Terms of Service**.
- Introduzca el nombre del paquete de la aplicación que quiera utilizar el servicio en el campo**Application package name**.
- Haga clic en el botón **Register**.



Aparece la siguiente pantalla, que incluye la clave única generada y asociada al nombre del paquete que se ha indicado.

- Guarde la página web o, al menos, la clave anotando a qué paquete está asociada. Si fuera necesario, es posible generar la misma clave varias veces.



2. Configuración de la clave

Una vez generada la clave, es preciso agregarla en el manifiesto insertando una etiqueta `meta-data` dentro de la etiqueta `application`. Esta etiqueta `meta-data` debe incluir el atributo`android:name` identificando al dato, en este caso `com.google.android.backup.api_key`, y el atributo `android:value` especificando el valor de la clave generada.

Sintaxis

```
<meta-data android:name="com.google.android.backup.api_key"
           android:value="cadena de caracteres" />
```

Ejemplo

```
<?xml version="1.0" encoding="utf-8"?>
<manifest ...>
  <application ...>
    ...
  <meta-data android:name="com.google.android.backup.api_key"
    android:value="AEdPqrEAAFAIoa...Hq29AXUAFOtQVgwurLtJfF" />
  </application>
</manifest>
```

3. Agente de copia de seguridad

La aplicación debe proveer un agente de copia de seguridad encargado de transmitir la información a salvaguardar y a restaurar al Gestor de copia de seguridad.

Para ello, hay que crear una clase que herede de la clase `BackupAgent`. La clase creada se comunica con el Gestor de copia de seguridad y debe, en particular, implementar los métodos`onBackup` y `onRestore` para realizar respectivamente las operaciones de salvaguarda y de restauración.

Android provee la clase `BackupAgentHelper` que hereda de la clase `BackupAgent`. Esta clase utiliza asistentes capaces de salvaguardar de manera sencilla ciertos tipos de datos y, en particular, archivos binarios almacenados en local. Por ello, no es necesario implementar los

métodosonBackup y onRestore. Por el contrario, heredar de la clase BackupAgentHelper entraña ciertas restricciones como, por ejemplo, la imposibilidad de salvaguardar datos almacenados en una base de datos SQLite. Para ello, será preciso heredar directamente de la clase BackupAgent.

Aquí trataremos solamente la herencia de la clase BackupAgentHelper, antes que de la clase básica BackupAgent, con el objetivo de aprovechar los servicios que ofrece aquélla.

a. Configuración

Una vez se ha escogido el nombre de la clase, es preciso indicarla en el manifiesto utilizando el atributo android:backupAgent de la etiqueta application. Esto permite al Gestor de copia de seguridad conocer y poderse comunicar con el agente correspondiente.

Sintaxis

```
android:backupAgent="cadena de caracteres"
```

De forma similar a los demás componentes, este nombre puede comenzar por un punto, de modo que se utilizará el nombre del paquete indicado en el atributo package de la etiqueta manifestpara componer el nombre completo.

Ejemplo

```
<?xml version="1.0" encoding="utf-8"?>
<manifest ...>
    <application android:backupAgent=".MiAgenteDeCopiaDeSeguridad" >
        ...
    </application>
</manifest>
```

b. BackupAgentHelper

La clase BackupAgentHelper utiliza los asistentes de copia de seguridad, cada uno especializado en la salvaguarda de un tipo de datos. Android proporciona dos asistentes representados por las clases SharedPreferencesBackupHelper y FileBackupHelper.

La clase SharedPreferencesBackupHelper está especializada en la copia de seguridad y la restauración de archivos de preferencias. Basta con instanciar esta clase para obtener el asistente correspondiente. El constructor recibe como parámetros el contexto de la aplicación así como un número indefinido de nombres de archivos de preferencias que se quiere salvaguardar.

Sintaxis

```
public SharedPreferencesBackupHelper (Context context,
   String... prefGroups)
```

Ejemplo

```
SharedPreferencesBackupHelper asistenteArchivosPrefs =
  new SharedPreferencesBackupHelper(this, "nombreArchivoPrefs1",
    "nombreArchivoPrefs2");
```

La clase FileBackupHelper está especializada en la copia de seguridad y la restauración de archivos binarios de pequeño tamaño. Incluso en este caso, basta con instanciar esta clase para obtener el asistente correspondiente. El constructor recibe como parámetros el contexto de la aplicación así como un número indefinido de archivos binarios de almacenamiento interno a salvaguardar.

Sintaxis

```
public FileBackupHelper (Context context, String... files)
```

Ejemplo

```
FileBackupHelper asistenteArchivosBin =
  new FileBackupHelper(this, "nombreArchivoBin1", "nombreArchivoBin2");
```

Como se ha indicado, la clase `MiAgenteDeCopiaDeSeguridad` se crea sobrecargando la clase`BackupAgentHelper`. Ésta debe indicar los asistentes que utiliza, es decir, las instancias de los objetos `SharedPreferencesBackupHelper` y `FileBackupHelper`. Para ello, debe utilizar su método `addHelper`. Este método recibe como parámetros una clave en forma de cadena de caracteres así como el objeto asistente asociado.

Sintaxis

```
public void addHelper (String keyPrefix, BackupHelper helper)
```

Ejemplo

```
addHelper("archivosPrefs", asistenteArchivosPrefs);
```

Esta declaración de asistentes debe realizarse en el método `onCreate` de la clase`MiAgenteDeCopiaDeSeguridad`. Este método se invoca antes de realizar la copia de seguridad o la restauración. Permite, por tanto, inicializar el proceso.

Sintaxis

```
public void onCreate ()
```

Ejemplo

```
public class MiAgenteDeCopiaDeSeguridad extends BackupAgentHelper {

  @Override
  public void onCreate() {
    SharedPreferencesBackupHelper asistenteArchivosPrefs =
      new SharedPreferencesBackupHelper(this,
        "nombreArchivoPrefs1", "nombreArchivoPrefs2");
    addHelper("archivosPrefs", asistenteArchivosPrefs);

    FileBackupHelper asistenteArchivosBin =
      new FileBackupHelper(this, "nombreArchivoBin1",
        "nombreArchivoBin2");
    addHelper("archivosBin", asistenteArchivosBin);
  }

}
```

Dado que la copia de seguridad de los datos puede tener lugar en cualquier momento, hay que asegurar que no ocurre durante una actualización de los datos en local por la propia aplicación. En caso contrario, los datos almacenados en línea podrían verse corrompidos.

Una de las soluciones propuestas para responder a este problema consiste en volver atómicas las fases de escritura y de lectura de datos en los archivos. En concreto, esto consiste en sincronizar el acceso a los datos mediante el uso de un cerrojo, por ejemplo.

El acceso a los archivos de preferencias mediante una interfaz de tipo `SharedPreferences` es, en lo sucesivo, thread-safe. Es decir, esta interfaz asegura la atomicidad de las operaciones de escritura y de lectura. No es necesario hacer nada para estos archivos.

Por el contrario, este no es el caso de los archivos binarios. Es preciso implantar tal mecanismo en el

uso de estos archivos por los componentes de la aplicación y por el agente de copia de seguridad. La implementación de este mecanismo debe realizarse en los métodos`onBackup` y`onRestore`que es preciso implementar.

Sintaxis

```
public abstract void onBackup (ParcelFileDescriptor oldState,
  BackupDataOutput data, ParcelFileDescriptor newState)

public abstract void onRestore (BackupDataInput data,
  int appVersionCode, ParcelFileDescriptor newState)
```

Ejemplo

```
public static final Object sCerrojo = null;

@Override
public void onBackup(ParcelFileDescriptor oldState,
    BackupDataOutput data, ParcelFileDescriptor newState)
    throws IOException {
  synchronized (sCerrojo) {
    super.onBackup(oldState, data, newState);
  }
}

@Override
public void onRestore(BackupDataInput data, int appVersionCode,
    ParcelFileDescriptor newState) throws IOException {
  synchronized (sCerrojo) {
    super.onRestore(data, appVersionCode, newState);
  }
}
```

Todos los accesos de escritura a los archivos salvaguardados deberán realizarse utilizando este bloqueo en el conjunto de la aplicación.

4. Gestor de copia de seguridad

Corresponde a la aplicación advertir al Gestor de copia de seguridad que sus datos se han modificado y deben, por tanto, sincronizarse en línea. El Gestor de copia de seguridad decide a continuación en qué momento se realizará realmente esta sincronización en línea. Incluso si en ese tiempo la aplicación ha vuelto a realizar su petición varias veces, el Gestor mutualizará estas peticiones para realizar la copia de seguridad en línea una única vez. Utilizará, por tanto, el Agente de copia de seguridad especificado por la aplicación.

a. Solicitar una copia de seguridad

Para realizar una petición de copia de seguridad, la aplicación debe, en primer lugar, crear una instancia del Gestor de copia de seguridad. Para ello, debe utilizar simplemente el constructor de la clase `BackupManager`.

Sintaxis

```
public BackupManager (Context context)
```

Ejemplo

```
BackupManager backupManager = new BackupManager(this);
```

La aplicación debe, a continuación, invocar al método `dataChanged` del Gestor de copia de seguridad para informar que sus datos se han visto modificados y que puede realizar la copia de seguridad en línea.

Sintaxis

```
public void dataChanged ()
```

Ejemplo

```
backupManager.dataChanged();
```

Los datos salvaguardados en línea los restaura automáticamente el sistema en el dispositivo Android tras la instalación de la aplicación y, por tanto, antes de su primera ejecución. Para ello es necesario que el sistema disponga de esta funcionalidad, que el usuario no la haya desactivado y que el servicio en línea esté disponible.

La aplicación puede, no obstante, forzar una restauración de datos en cualquier momento invocando al método `requestRestore` del Gestor de copia de seguridad.

b. Probar el servicio

Para poder probar de forma sencilla la implementación del sistema de copia de seguridad en línea por la aplicación, es posible forzar manualmente las distintas etapas de copia de seguridad y de restauración.

Esto se realiza desde una consola. La herramienta `adb` permite abrir un shell en el dispositivo Android o el emulador. Desde este shell, hay que utilizar el comando `bmgr` pasándole las opciones deseadas. Las opciones más utilizadas son `backup`, `run` y `restore` que se corresponden respectivamente con la solicitud de copia de seguridad, la realización efectiva de la copia de seguridad en línea y la restauración. Basta con no especificar ninguna opción para obtener la lista completa de opciones disponibles.

Sintaxis

```
adb shell bmgr opciones
```

Ejemplo

```
$ adb shell bmgr enable true
$ adb shell bmgr backup es.midominio.android.miaplicacion
$ adb shell bmgr run
$ adb uninstall es.midominio.android.miaplicacion
```

En este ejemplo, se activa el servicio de copia de seguridad en el emulador o el dispositivo. A continuación, se solicita y se fuerza la ejecución de la copia de seguridad. Por último, se suprime la aplicación.

Sólo queda reinstalar la aplicación para verificar que los datos se restauran correctamente.

Introducción

En un contexto profesional, la interfaz de una aplicación Android es un elemento que no hay que descuidar: no debemos olvidar que, en un primer momento, es el diseño de la aplicación lo que va a dar a los usuarios de Play Store una primera impresión y permitirles prejuzgar la calidad de nuestra aplicación.

De hecho, el diseño de la interfaz gráfica supone una carga de trabajo en consecuencia: además de las restricciones vinculadas con los múltiples tamaños de pantalla y a las diferentes versiones del sistema Android, las normas de ergonomía así como las restricciones impuestas por los equipos de diseño gráfico (por lo general responsables de la integridad del diseño de la aplicación) incluyen su lote de dificultades para el desarrollador.

Veremos, en este capítulo, cómo reducir la carga de trabajo maximizando el uso de componentes complejos de la interfaz y, además, veremos cómo implementar los elementos de navegación sugeridos por Google. A continuación, veremos cómo, utilizando elementos gráficos creados mediante XML, limitar las variaciones gráficas para los pictogramas, botones y fondos de pantalla. Por último, terminaremos el capítulo con algunos trucos que simplifican el diseño de las representaciones utilizadas con mayor frecuencia.

Crear sus propios componentes

Una de las reglas más importantes en términos de diseño de aplicación es la unicidad: si una misma funcionalidad está presente en varias pantallas de una aplicación, debe presentar el mismo diseño para todas las pantallas.

Para evitar al desarrollador tener que producir el mismo código en distintas ubicaciones, el sistema Android le ofrece la posibilidad de diseñar sus propios componentes de interfaz, que podrán utilizarse con la misma facilidad que los componentes nativos de la plataforma como, por ejemplo, `editText`,`ListView`, etc.

1. Sobrecargar un componente existente

Si el componente que queremos crear es parecido a algún otro componente ya existente, y si lo que queremos, principalmente, es extender sus funcionalidades, es recomendable sobrecargar dicho componente en lugar de crear un componente nuevo partiendo de una hoja en blanco.

Todos los componentes, como hemos visto en el capítulo Los fundamentos, forman parte del paquete `android.widget` y es posible sobrecargarlos.

a. Extender una clase del paquete android.widget

Por regla general, el layout del componente sobrecargado no se modifica en sí mismo, sino que las modificaciones realizadas en este contexto son por lo general reducidas. Basta, por tanto, con crear una nueva clase, que extenderá de la clase del componente que se ha seleccionado, y agregar los métodos que deseemos.

```
package es.midominio.miAplicacion;

import android.widget.AutoCompleteTextView;
[...]
public class MiCustomAutoComplete extends AutoCompleteTextView{
...
}
```

Esta clase deberá proveer dos constructores: uno que se utiliza cuando se declara una instancia de la clase en el código y otra que se utiliza específicamente cuando el componente se declara en un archivo de recursos (un archivo de layout de una actividad, por ejemplo).

En ambos casos, resulta obligatorio invocar al constructor correspondiente del padre, llamada que debe ser la primera instrucción del constructor.

El primer constructor, para una declaración directa en el código, recibe como único parámetro un objeto `Context`.

```
public MiCustomAutoComplete(Context context) {
   super(context);
}
```

El segundo constructor, que se utiliza cuando el componente se declara en un archivo XML de layout, recibe, además de un objeto `Context`, un objeto de tipo`android.util.AttributeSet`, que permite recuperar y utilizar de manera simplificada los atributos declarados en XML.

```
public MiCustomAutoComplete (Context context, AttributeSet attr){
   super(context, attr);
   ...
}
```

En el caso de un componente creado por varias piezas, este último constructor es el más complicado de definir, puesto que debe tener en cuenta todas las propiedades que pueden haberse declarado en el archivo XML. En el caso de una sobrecarga de un componente nativo, la llamada al constructor permite evitar este trabajo.

Según la personalización deseada, una vez definidos los constructores, basta con sobrecargar los métodos correspondientes, consultando la documentación del componente nativo extendido.

b. Integrar el nuevo componente en un layout

La integración de un componente personalizado en un layout se realiza de la misma manera que para un componente nativo. Es preciso, no obstante, tener en cuenta el empaquetado al que pertenece el componente personalizado e indicar el nombre completo del paquete de la clase creada en la etiqueta XML del componente.

```
<es.midominio.miAplicacion.MiCustomAutoComplete
       android:layout_width="fill_parent"
       android:layout_height="wrap_content"/>
```

El uso del componente en el código de la actividad no difiere del uso de un componente nativo: la referencia al componente se realiza utilizando el método `findViewById`.

c. Agregar atributos personalizados

Uno de los principales intereses para sobrecargar componentes nativos del sistema reside en la posibilidad que esto ofrece para definir atributos personalizados. Estos atributos pueden, a continuación, recibir un valor igual que los atributos nativos mediante la etiqueta XML correspondiente del componente.

En primer lugar, veremos cómo declarar estos atributos personalizados. A continuación, estudiaremos cómo vincular sus valores (y explotarlos).

La declaración de atributos personalizados se realiza en un archivo de recursos XML específico, ubicado en la carpeta `values` de los recursos del proyecto. Este archivo se denomina, clásicamente, `attrs.xml` (por *attributes*), aunque es posible asignarle el nombre que se desee.

El archivo `attrs` es un archivo de recursos: la etiqueta XML de primer nivel es, por tanto, una etiqueta `<resources>`. Cada conjunto de atributos personalizados se define en una etiqueta`<declare-styleable>`, cuya propiedad `name` debe estar obligatoriamente informada. Cada atributo se define en una etiqueta `<attr>`, con las propiedades `name` y `format`.

```
<?xml version="1.0" encoding="utf-8"?>
 <resources>
   <declare-styleable name="misAtributosCustoms">
      <attr name="miAtributoCustom_1" format="string"/>
      <attr name="miAtributoCustom_2" format="boolean"/>
   </declare-styleable>
</resources>
```

Para utilizar estos atributos personalizados en un archivo de `layout`, es preciso declarar el espacio de nombres correspondiente (el nombre del paquete de la solución) en el archivo XML del`layout`, de forma análoga al tradicional espacio de nombres "`android`".

```
<RelativeLayout
   xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:capitulo8="http://schemas.android.com/apk/res/
es.midominio.MiAplicacion"
```

```
   android:layout_width="match_parent"
   android:layout_height="match_parent">

   <es.midominio.miAplicacion.MiCustomAutoComplete
       android:layout_width="fill_parent"
       android:layout_height="wrap_content"
       capitulo8:miAtributoCustom_1="valor_1"
       capitulo8:miAtributoCustom_2="true"/>
</RelativeLayout>
```

Es posible recuperar los atributos definidos por el componente en la etiqueta XML a nivel de la clase del componente, típicamente en el constructor. Esta recuperación se realiza en dos tiempos:

- Recuperación de un objeto de tipo `TypedArray` gracias al método `obtainStyledAttributes()` del objeto `Context`.
- A continuación, recuperación del valor de cada uno de los atributos a partir del objeto `TypedArray`.

La versión del método `obtainStyledAttributes` que utilicemos aquí presenta la siguiente firma:

```
public final TypedArray obtainStyledAttributes(AttributeSet set,
int[] attrs)
```

- El primer parámetro es de tipo `AttributeSet`, y se corresponde con el que se recibe como parámetro del constructor del componente.
- El segundo parámetro, una tabla de valores enteros, representa la lista de atributos que se desean recuperar.

Este segundo parámetro debe explicitarse: se trata de hecho de una tabla con los identificadores generados en tiempo de compilación de la solución, y que podemos encontrar en el archivo `R.java`, dentro de la carpeta `gen` de la solución.

Para cada conjunto de atributos personalizados (cada etiqueta `declare-styleable`), el compilador genera los identificadores y una tabla de valores enteros correspondiente. Las siguientes reglas se aplican en la nomenclatura de los identificadores y de la tabla:

- La tabla lleva el nombre indicado en su etiqueta `declare-styleable`.
- Cada atributo se denomina [nombre_de_la_etiqueta_xml]_[nombre_del_atributo].
- Estos elementos son accesibles mediante la clase `R.styleable`.

En nuestro ejemplo, tendremos:

```
public static final int
   misAtributosCustoms_miAtributoCustom_1 = ...;

public static final int
   misAtributosCustoms_miAtributoCustom_2 = ...;

   public static final int[]misAtributosCustoms=...;
```

Y típicamente, es la tabla `misAtributosCustoms` la que se utilizará como segundo argumento del método `obtainStyledAttributes`.

```
public MiCustomAutoComplete(Context context, AttributeSet attr){
   super(context, attr);

   TypedArray customAttributes =
       context.obtainStyledAttributes(attr,
```

```
            R.styleable.misAtributosCustoms);
    ...
}
```

Por último, resulta muy sencillo obtener los valores de los distintos atributos utilizando el objeto`TypedArray` que devuelve el método: en función del tipo definido, basta con utilizar el método correspondiente.

```
String atributo_1 = customAttributes.getString(
    R.styleable.misAtributosCustoms_miAtributoCustom_1);

boolean atributo_2 = customAttributes.getBoolean(
    R.styleable.misAtributosCustoms_miAtributoCustom_2,
    true);
```

2. Reunir un conjunto de componentes

De la misma forma que podemos extender un componente nativo del framework, puede resultar interesante reunir un conjunto de componentes en uno solo y, de este modo, encapsular toda la lógica de procesamiento para una mejor reutilización.

En este caso, en lugar de sobrecargar directamente un componente, resulta más adecuado extender uno de los componentes layout - los componentes `linearlayout` o `relativelayout`, por ejemplo - que realizarán la misma representación del componente, y exponer los métodos necesarios para el funcionamiento del componente.

En esta arquitectura, los componentes puede declararse bien directamente en el código o bien utilizando un archivo XML de layout.

En el caso de componentes declarados por programación, el funcionamiento es estrictamente idéntico al de una actividad: es preciso instanciar los componentes y agregarlos al layout padre mediante una de las variantes del método `addView(View)`.

En el caso de preferir utilizar un layout declarado en XML, para una representación compleja, por ejemplo, basta con utilizar un objeto de tipo `LayoutInflater` para invocar la creación del layout, en lugar de la clásica llamada al método `setContentView` de la clase `Activity`.

El objeto `LayoutInflater` se obtiene a partir de una instancia de la clase `Context`, invocando al método `getSystemService()`.

```
public void inflate(int layout) {
    LayoutInflater li =
(LayoutInflater)getContext().getSystemService
(Context.LAYOUT_INFLATER_SERVICE);
    li.inflate(layout, this);
}
```

3. Construir completamente un componente

Para ir todavía más allá y extender (casi) completamente los componentes nativos de la plataforma, es posible crear un componente completamente nuevo. Evidentemente, la tarea resulta mucho más compleja, y dependerá en gran medida de las interacciones con el usuario.

En cualquier caso, se recomienda encarecidamente basarse en un objeto de tipo `View`, que aportará los procesamientos de bajo nivel, siendo lo más genérico posible. Además de la declaración de los constructores vistos anteriormente, la parte esencial del trabajo será, en este caso, la escritura de los métodos `onDraw()` y `onMeasure()`: el método `onDraw()` se encarga de "dibujar" la interfaz gráfica por pantalla, el método `onMeasure()` es responsable de determinar las dimensiones (ancho y alto) del componente creado.

a. Implementar onDraw()

La firma completa del método es la siguiente:

```
protected void onDraw (Canvas canvas)
```

El objeto de tipo `Canvas` que se pasa como parámetro es la instancia que incluirá la llamada a los métodos encargados de dibujar las formas que deseemos integrar en la vista. Estas formas podrán ser primitivas geométricas (línea, rectángulo, arco, punto, etc.), pero también objetos de tipo bitmap así como texto.

Por último, para dibujar el componente, es necesario definir al menos un objeto de tipo `Paint`, que se corresponde con un pincel virtual. La clase `Paint` permite definir el estilo del pincel, el color, los atributos de texto, así como la presencia de sombra (¡automática!), el antialiasing, etc.

El método `onDraw()` se invoca la primera vez que la vista debe dibujarse. Para forzar, a continuación, que se invoque al método `onDraw()`, es preciso invocar al método `invalidate()` de la vista. En el caso de que la demanda se realice en un thread separado (es decir, en un thread diferente a `UIThread`), es preciso invocar a `postInvalidate()`; la llamada será, entonces, asíncrona.

b. Implementar onMeasure()

La sobrecarga del método `onMeasure()`, en el marco de la creación de un nuevo componente, resulta seguramente la parte más compleja de implementar para un desarrollador que desee cubrir todos los casos de uso del componente creado.

El objetivo de este método es calcular el ancho y alto de la vista e informar al padre de la vista en cuestión. El hecho de informar al padre de la vista, una vez realizado el cálculo, es obligatorio, y su omisión generará una excepción de tipo `IllegalStateException`.

La firma de `onMeasure()` es la siguiente:

```
protected void onMeasure (int widthMeasureSpec, int
heightMeasureSpec)
```

Los parámetros de entrada al método están en el formato especificado por la clase estática`View.MeasureSpec`. Cada parámetro ofrece dos datos necesarios para la medida del componente:

- Un componente indica la dimensión asignada por la vista padre. Este componente se obtiene utilizando el método `MeasureSpec.getSize(int)`.
- Un componente indica la restricción impuesta sobre la dimensión. El valor de dicho componente la devuelve el método `MeasureSpec.getMode(int)`.

Este segundo componente tendrá uno de los siguientes valores:

- `UNSPECIFIED`: no existe ninguna restricción sobre las dimensiones del componente.
- `EXACTLY`: el padre impone las dimensiones indicadas mediante el método`getSize(int)`.
- `AT_MOST`: el padre indica que la vista debe tener una dimensión igual o inferior a la que se indica en el método `getSize()`.

Una vez determinadas las dimensiones (ancho y alto) deseadas del componente, es obligatorio invocar al método `setMeasuredDimension()`, cuya firma es la siguiente:

```
setMeasuredDimension (int measuredWidth, int measuredHeight);
```

Si este método no se invoca, el sistema generará una excepción del tipo`IllegalStateException`.

c. Obtener las dimensiones de la pantalla

Con el objetivo de dimensionar correctamente el control, resulta indispensable conocer las dimensiones de la pantalla del dispositivo sobre el que se ejecuta la aplicación.

Para dispositivos que ejecuten una versión de Android anterior a la versión 3.2 (API 13), es preciso invocar a los métodos `getWidth()` y `getHeight()` del objeto `Display`.

Para los sistemas que trabajan con Android 3.2 o superior, el método `getSize()` del objeto`Display` reemplaza a los métodos `getWidth()` y `getHeight()`.

```
private void measureScreen() {
        WindowManager wm = (WindowManager)
this.getContext().getSystemService(Context.WINDOW_SERVICE);

        Display display = wm.getDefaultDisplay();

        if (android.os.Build.VERSION.SDK_INT >=
android.os.Build.VERSION_CODES.HONEYCOMB_MR2) {
                Point size = new Point();
                display.getSize(size);
                screenWidth = size.x;
                screenHeight = size.y;
        }
        else {
                screenWidth = display.getWidth();
                screenHeight = display.getHeight();
        }
}
```

Utilizar el Navigation Drawer

Aparecido progresivamente con la versión 4 de Android, el "Navigation drawer" se ha impuesto como un elemento imprescindible en la interfaz de las aplicaciones Android. Este componente, literalmente "el cajón de navegación", es de hecho un panel que contiene el acceso a las distintas pantallas de una aplicación - lo denominaremos *panel de navegación* en lo sucesivo. Este panel se posiciona sobre la pantalla principal, sin cubrirla en su totalidad, cuando el usuario realiza un movimiento de barrido sobre la pantalla (de izquierda a derecha) o cuando hace clic en el icono ubicado en la zona superior izquierda de la barra de acción.

El rol del panel de navegación es presentar al usuario el conjunto de pantallas de una aplicación accesibles sin un contexto particular, cuando existen varios: en efecto, si una aplicación posee dos o tres pantallas de primer nivel, resulta más eficaz utilizar pestañas.

Figura 1: panel de navegación en la aplicación Gmail (Google)

La noción de pantalla sin contexto resulta esencial para integrar correctamente este componente en una aplicación, el panel de navegación resulta accesible por lo general desde todas las pantallas de la aplicación, y su contenido no puede, y no debe, depender del contenido de la pantalla mostrada, o de algún elemento seleccionado por el usuario. Recuerde que los elementos de navegación contextuales deben mostrarse en la barra de acción rápida, o en el menú contextual accesible desde esta misma barra.

1. Implementar el panel de navegación

El componente que permite mostrar un panel de navegación no es nativo de la plataforma Android, sino que está integrado en la librería android-support-v4: es preciso, en primer lugar, integrar esta librería al proyecto.

El componente se declara habitualmente en el archivo de layout principal (suponiendo que por lo general va de la mano del uso de los fragmentos), empleando la etiqueta `<DrawerLayout>`.

El primer punto a destacar es que este objeto `DrawerLayout` es, de hecho, un contenedor (que extiende de `ViewGroup`, igual que un componente `LinearLayout` o `ScrollView`).

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.v4.widget.DrawerLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/drawer_layout"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
...
</android.support.v4.widget.DrawerLayout>
```

El `DrawerLayout` debe contener dos vistas: la vista que utiliza el contenido del panel de navegación, y la vista que contiene los elementos de la página/pantalla. Es necesario respetar algunas reglas, en caso contrario el componente no funcionará:

Es preciso declarar la vista que incluye el contenido de la página en primer plano: los componentes declarados en primer plano se posicionan "sobre" los componentes declarados a continuación.

Es preciso que la vista que incluye el contenido del panel de navegación declare la propiedad`layout_gravity`, en caso contrario no se considerará como un panel de navegación. En teoría, esta propiedad debe tener el valor "left", aunque, para ser compatible con todas las formas de escritura, se recomienda utilizar el valor "`start`": si el dispositivo del usuario estuviera configurado para escribir de derecha a izquierda, el panel de navegación se situará de este modo a la derecha de la pantalla.

Por último, el ancho del panel - cuando es visible - debe indicarse en `dp`. Google recomienda no utilizar un ancho superior a 320 dp, con el objetivo de dejar siempre una parte de la página visible para el usuario.

El sistema, que se basa en la presencia de la propiedad `layout_gravity` para identificar el panel de navegación, necesita que sólo la vista que forma este panel incluya esta propiedad`layout_gravity`.

Incluso aunque clásicamente los desarrolladores utilizan una `listView`, la vista que forma el panel de navegación puede ser cualquier componente que herede de `View` (`LinearLayout`,`ScrollView`, `GridView`, etc.).

El esqueleto del archivo XML de layout tiene el siguiente aspecto:

La

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.v4.widget.DrawerLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/drawer_layout"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <FrameLayout
        android:id="@+id/content_layout"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical"/>

    <ListView
        android:id="@+id/drawer_content"
        android:layout_width="260dp"
        android:layout_height="match_parent"
```

```
        android:background="#efedef"
            android:layout_gravity="start"/>

</android.support.v4.widget.DrawerLayout>
```

obtención de una referencia al objeto `DrawerLayout` a nivel del código de la actividad no difiere de otros widgets clásicos de la plataforma, y se realiza naturalmente mediante el método `findViewById()`.

Por defecto, el panel de navegación no integra el efecto de sombra que se observa de manera general cuando el panel está abierto. Para agregarlo, es preciso invocar al método `setDrawerShadow(Drawable shadowDrawable, int gravity)`, pasándole como parámetros el recurso `drawable` que genera la sombra y la constante que indica el lado hacia el que debe abrirse el panel.

```
drawerLayout = (DrawerLayout)findViewById(R.id.drawer_layout);

drawerLayout.setDrawerShadow(R.drawable.drawer_shadow, Gravity.START);
```

Google provee un conjunto de iconos estándar para vestir el panel de navegación, que puede descargarse de la siguiente dirección:http://developer.android.com/downloads/design/Android_Navigation_Drawer_Icon_20130516.zip

En este paquete se provee también una imagen `drawer_shadow.9.png`.

2. Usar el panel de navegación

Las tareas de apertura y cierre del panel de navegación se gestionan desde la librería android-support, y no se requiere ningún código adicional específico.

Los dos únicos puntos a tener en cuenta son relativos al comportamiento de la barra de acción, otro componente prácticamente indispensable en una interfaz moderna que respete los estándares de diseño.

Observe que no abordamos en esta sección el clic sobre un elemento del panel de navegación: este punto está directamente vinculado con la elección del desarrollador en cuanto al contenido del panel (un `ListView` o cualquier otro layout), y el procesamiento se realiza de manera idéntica a los mecanismos clásicos de respuesta a un clic sobre un elemento de una vista (esta noción se ha abordado durante el estudio de los principales widgets de la plataforma).

a. Detectar los eventos de apertura/cierre

La API proporciona una interfaz `DrawerListener` que permite detectar los eventos de apertura y cierre del panel de navegación.

Es

```
DrawerListener drawerListener = new DrawerListener() {
    @Override
    public void onDrawerStateChanged(int arg0) {
    }

    @Override
    public void onDrawerSlide(View arg0, float arg1) {
    }

    @Override
    public void onDrawerOpened(View arg0) {
    }

    @Override
    public void onDrawerClosed(View arg0) {
    }
```

```
};
```

posible vincular un objeto de tipo `DrawerListerner` invocando al método `setDrawerListener()` del objeto `DrawerLayout`.

Es

```
 DrawerLayout drawerLayout;
[...]
drawerLayout.setDrawerListener(drawerListener);
```

preciso, no obstante, tener en cuenta que es extraño sobrecargar directamente la interfaz `DrawerListener`: es mucho más habitual utilizar un objeto `ActionBarDrawerToggle`, que estudiaremos en las siguientes secciones.

b. Navigation Drawer y ActionBar

Por regla general, estos dos elementos de navegación que son el panel de navegación y la barra de acción se utilizan de manera conjunta. Google, a través de sus propias aplicaciones y en los consejos de diseño que propone, nos da algunas directivas acerca de la manera de gestionar eficazmente estos dos componentes:

- El icono de acción overflow debe estar visible incluso aunque el panel de navegación esté visible.
- Cuando el panel de navegación se abre, el título que se presenta sobre la barra de acción debe cambiar para reflejar el contenido del panel.
- El usuario debe poder abrir el panel de navegación haciendo clic sobre el icono correspondiente, ubicado en la zona superior izquierda de la barra de acción.

No olvide que el icono que permite abrir el panel de navegación es, a priori, la única pista tras iniciar la aplicación de la presencia de este panel.

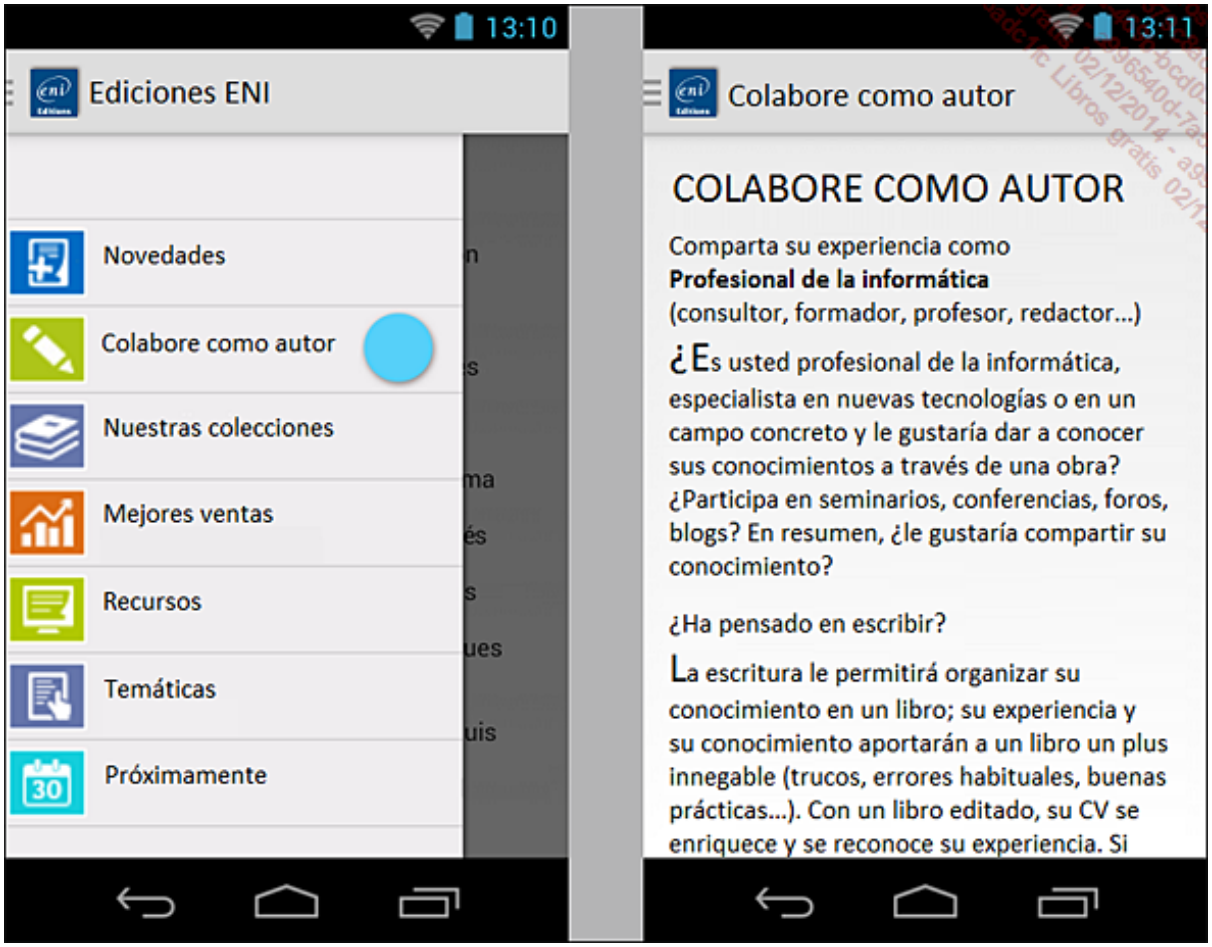


Figura 2: el título cambia cuando se selecciona un elemento

Para especificar que se desea mostrar el botón de apertura/cierre del panel, es preciso asignar el valor `true` a la propiedad `DisplayHomeAsUpEnabled`.

```
@Override
public void onCreate(Bundle savedInstanceState) {
   [...]
   getSupportActionBar().setDisplayHomeAsUpEnabled(true);
   [...]
}
```

c. Integrar el botón de apertura/cierre

La librería Android-Support provee un objeto `ActionBarDrawerToggle` que simplifica la gestión de este botón de apertura/cierre.

El

```
public ActionBarDrawerToggle (Activity activity,
         DrawerLayout drawerLayout,
         int drawerImageRes,
         int openDrawerContentDescRes,
         int closeDrawerContentDescRes)
```

constructor recibe como parámetro la actividad que declara el panel de navegación, el objeto de tipo `DrawerLayout` correspondiente, el identificador del recurso `drawable` utilizado como icono, y dos cadenas de caracteres utilizados en el caso de una visualización en modo de accesibilidad.

Este objeto, como hemos visto antes, implementa también la interfaz `DrawerListener` que permite responder a la apertura y al cierre del panel de navegación. Basta, entonces, con sobrecargar los métodos, tal y como hemos visto antes (típicamente, los métodos`onDrawerOpened()` y `onDrawerClosed()`), y asignarlos como `DrawerListener` al`DrawerLayout` de la actividad.

El

```
ActionBarDrawerToggle actionBarDrawerToggle =
      new ActionBarDrawerToggle(this,
            drawerLayout,
            R.drawable.drawer,
            R.string.open,
            R.string.close) {
   @Override
   public void onDrawerOpened(View arg0) {
      Toast.makeText(MainActivity.this, "Opened",
      Toast.LENGTH_SHORT).show();
   }

   @Override
   public void onDrawerClosed(View arg0) {
      Toast.makeText(MainActivity.this, "Closed",
      Toast.LENGTH_SHORT).show();
   }
};

drawerLayout.setDrawerListener(actionBarDrawerToggle);
```

método que se ejecuta al hacer clic sobre este botón es, como para los demás elementos de la barra de acción, el método `onOptionItemSelected(MenuItem)`.

Para facilitar el procesamiento, la clase `ActionBarDrawerToggle` expone su propio método`onOptionItemSelected(MenuItem)`, que devuelve `true` si el evento se ha procesado, o`false` en caso contrario: el evento se procesará si el `MenuItem` que se pasa como parámetro corresponde al objeto `ActionBarDrawerToggle`, si existe un panel de navegación y si el botón está declarado como disponible.

El método `onOptionItemSelected()` de la actividad se programa, con apenas código, de la

siguiente manera:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
   if (actionBarDrawerToggle.onOptionsItemSelected(item))
            return true;

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

Queda por tratar la inicialización del botón en función del estado del panel de navegación (abierto o cerrado) mediante el método `syncState()` del objeto `ActionBarDrawerToggle`. Esto se realiza, de forma clásica, a través del método `onPostCreate(Bundle savedInstanceState)` de la actividad.

```
@Override
protected void onPostCreate(Bundle savedInstanceState) {
   super.onPostCreate(savedInstanceState);
    actionBarDrawerToggle.syncState();
}
```

d. Forzar la apertura del panel cuando se inicia la actividad

En algunos casos puede resultar interesante abrir el panel de navegación cuando se inicia por primera vez la actividad. Para ello, basta con invocar al método `openDrawer(int gravity)` del objeto `DrawerLayout`.

Del mismo modo, el método `closeDrawer(int gravity)` permite forzar el cierre del panel de navegación.

En ambos casos, el parámetro, un entero, indica, como en el archivo XML de layout, el sentido de apertura y cierre del panel. Habitualmente, se utiliza el valor de la constante `Gravity.START`.

```
[...]
drawerLayout = (DrawerLayout)findViewById(R.id.drawer_layout);

drawerLayout.openDrawer(Gravity.START);
[...]
```

Crear imágenes redimensionables

De manera habitual, el desarrollador de la aplicación Android se enfrenta al problema de los elementos visuales redimensionables, tanto para fondos de pantalla, dibujos o fondos de los widgets (botones y zonas de texto, principalmente). La problemática es siempre la misma: conciliar eficacia - no multiplicar inútilmente las variaciones gráficas - y estética - no mostrar componentes amorfos.

La plataforma Android proporciona dos soluciones que permiten producir imágenes que se adaptan automáticamente a las dimensiones de los controles a los que están vinculadas. Veremos, en esta sección, dos técnicas, una orientada al aspecto gráfico y otra orientada a los desarrolladores.

1. Las imágenes 9-patch

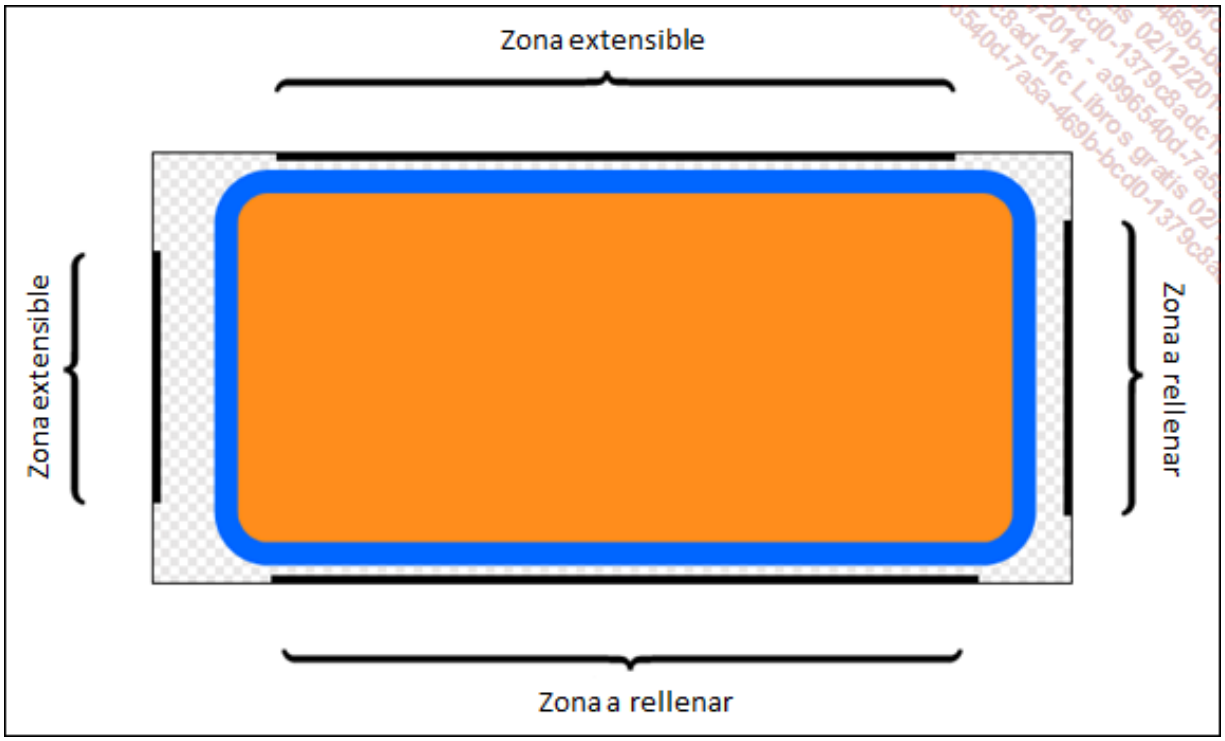
a. Presentación

Las imágenes nine-patch (a menudo descritas como 9-patch) son elementos visuales con formato PNG que contienen, además, información sobre las zonas que pueden ser estiradas por el sistema. Estas imágenes tienen como extensión de archivo "`.9.png`".

El principio de funcionamiento de las imágenes 9-patch es el siguiente: se definen en la imagen zonas horizontales y verticales que pueden estirarse para adaptarse a las dimensiones deseadas, así como las zonas que pueden rellenarse - en el caso de un botón o de una zona de texto, por ejemplo.

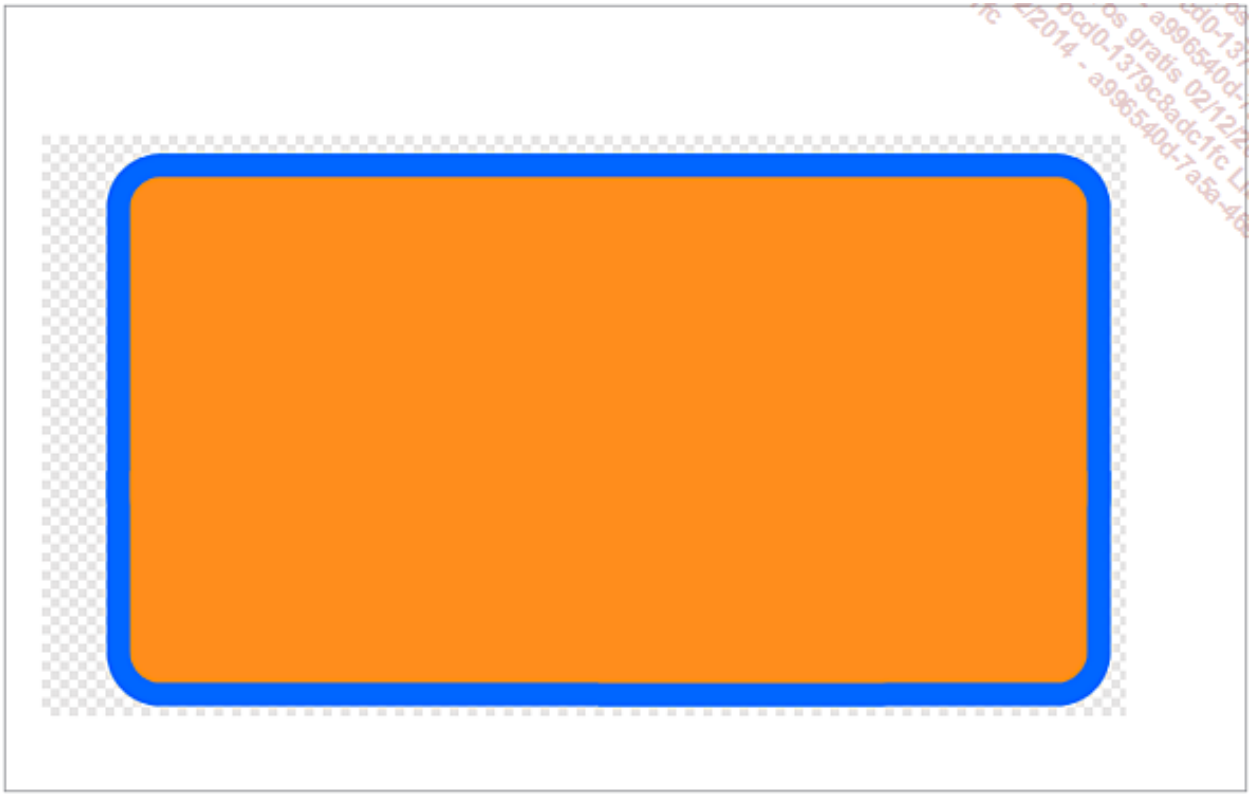
La definición de distintas zonas se realiza integrando un espacio suplementario de 1 pixel de cada lado de la imagen original, indicando en este espacio, mediante un pixel negro, que la zona puede agrandarse o rellenarse. Los márgenes de 1 pixel a la izquierda y arriba de la imagen permiten definir las zonas ampliables, los márgenes a la derecha y abajo indican las zonas que pueden rellenarse (mediante texto u otros elementos visuales).

El siguiente esquema presenta estas zonas para un objeto de fondo de una zona de texto:

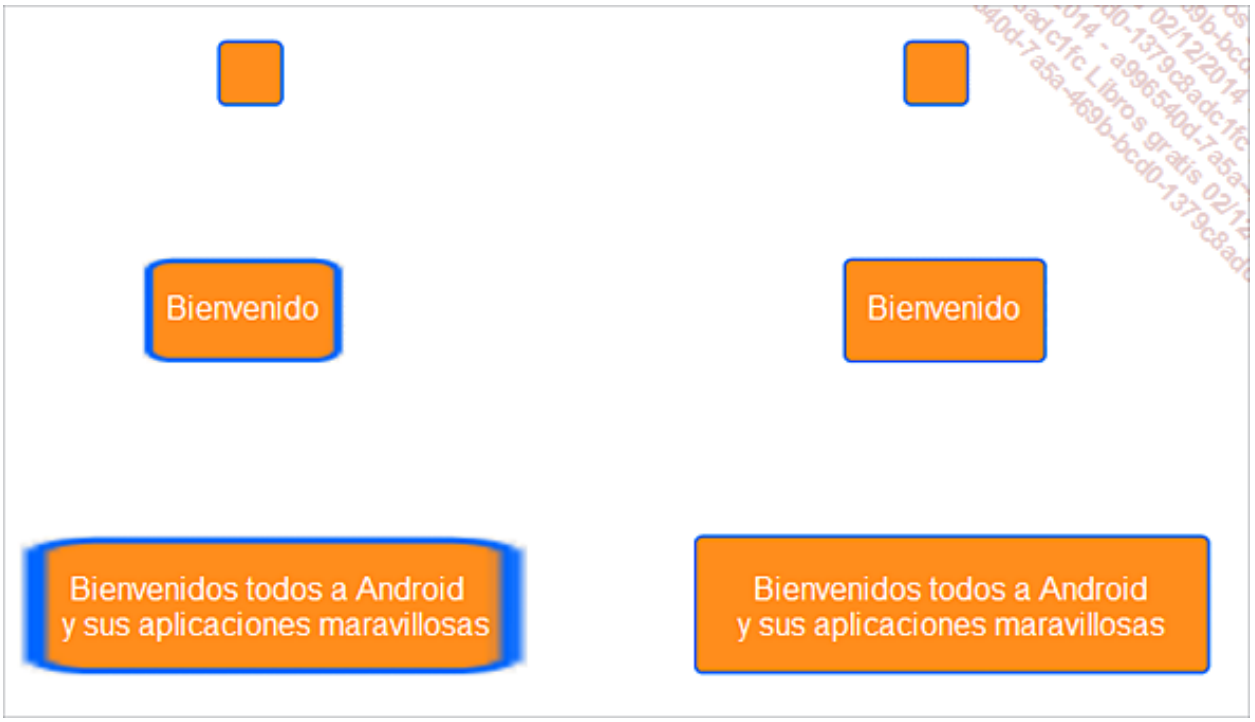


Como puede verse en el esquema, hemos definido una zona extensible, a la izquierda y arriba, que no integra el redondeo del borde del rectángulo: estas zonas no se verán afectadas por el redimensionamiento y, por tanto, no serán estiradas.

Cuando este elemento se agrande tendrá como resultado el siguiente aspecto visual:



La representación en Android muestra de forma evidente la diferencia de procesamiento de un fondo con formato 9-patch respecto a un simple archivo PNG.



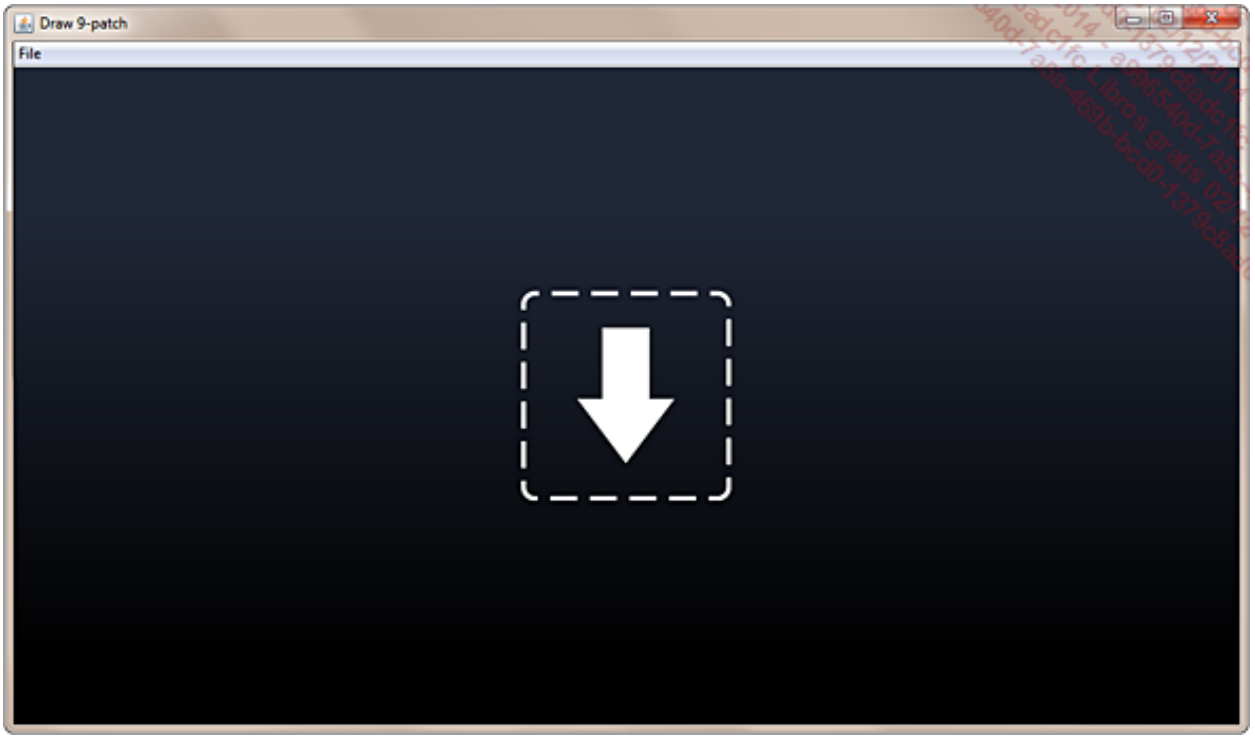
A la izquierda se muestran las zonas de texto que tienen un fondo con formato `PNG`, mientras que aquellas que aprovechan un fondo definido en formato `9-patch` se muestran a la derecha.

b. Crear imágenes 9-patch

La creación de imágenes 9-patch no presenta, a priori, ninguna dificultad: a partir de una imagen en formato PNG basta con ampliar la imagen dos píxeles en anchura y altura y, a continuación, posicionar los píxeles negros según convenga.

No obstante, en la práctica, utilizar un programa clásico de procesamiento de imágenes no da buenos resultados: la mayoría de programas realizan, en efecto, procesamientos sobre la imagen que la vuelven inutilizable en 9-patch: modificaciones (inapreciables) en el color de los píxeles que sirven para indicar las zonas, aplicación de filtros antialiasing, etc. El formato 9-patch impone, en efecto, que los píxeles que delimitan las zonas sean de estricto color negro ((0,0,0) en RGB), y que estén situadas exactamente en el borde de la imagen.

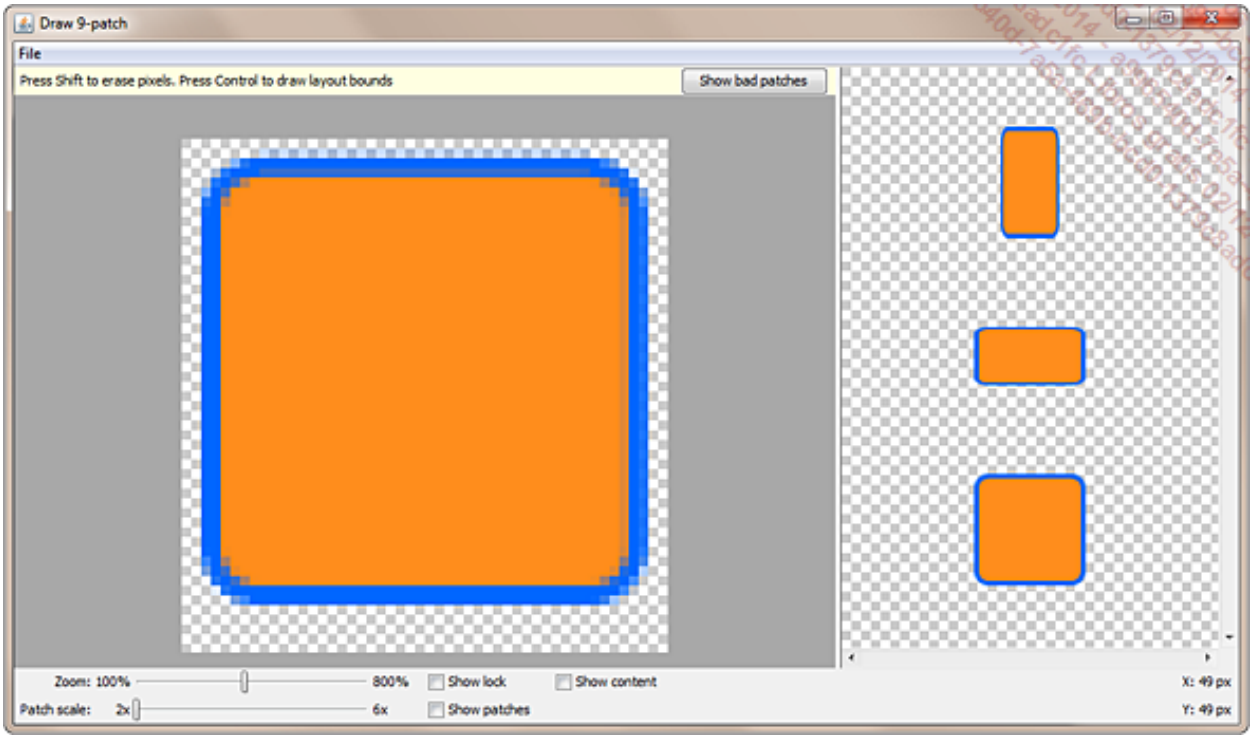
Para producir imágenes 9-patch es preferible utilizar una herramienta específica, disponible de manera gratuita, llamada Draw 9-patch, que se incluye con el SDK Android, en la carpeta`sdk/tools/`. Para iniciar el programa basta con ejecutar el archivo `draw9patch.bat`.



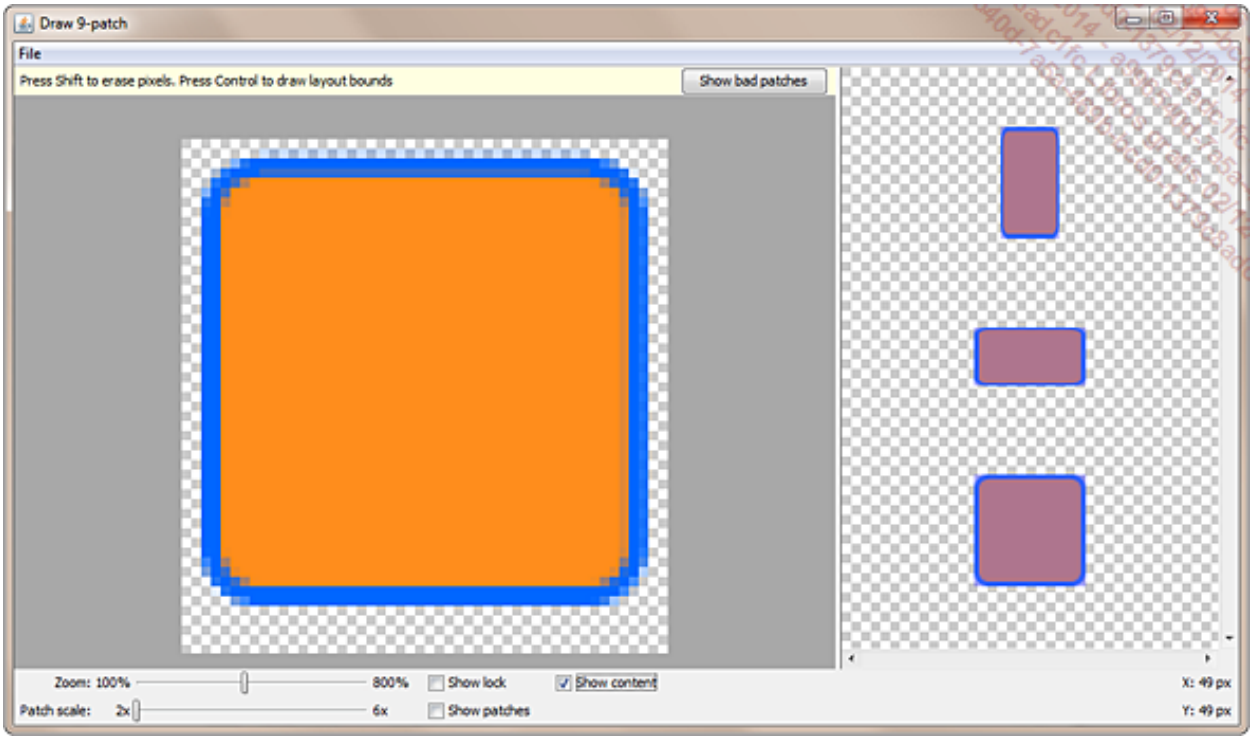
El uso de Draw 9-patch es muy sencillo:

- Seleccione **File** - **Open 9-patch** y seleccione su archivo de origen (en formato PNG).

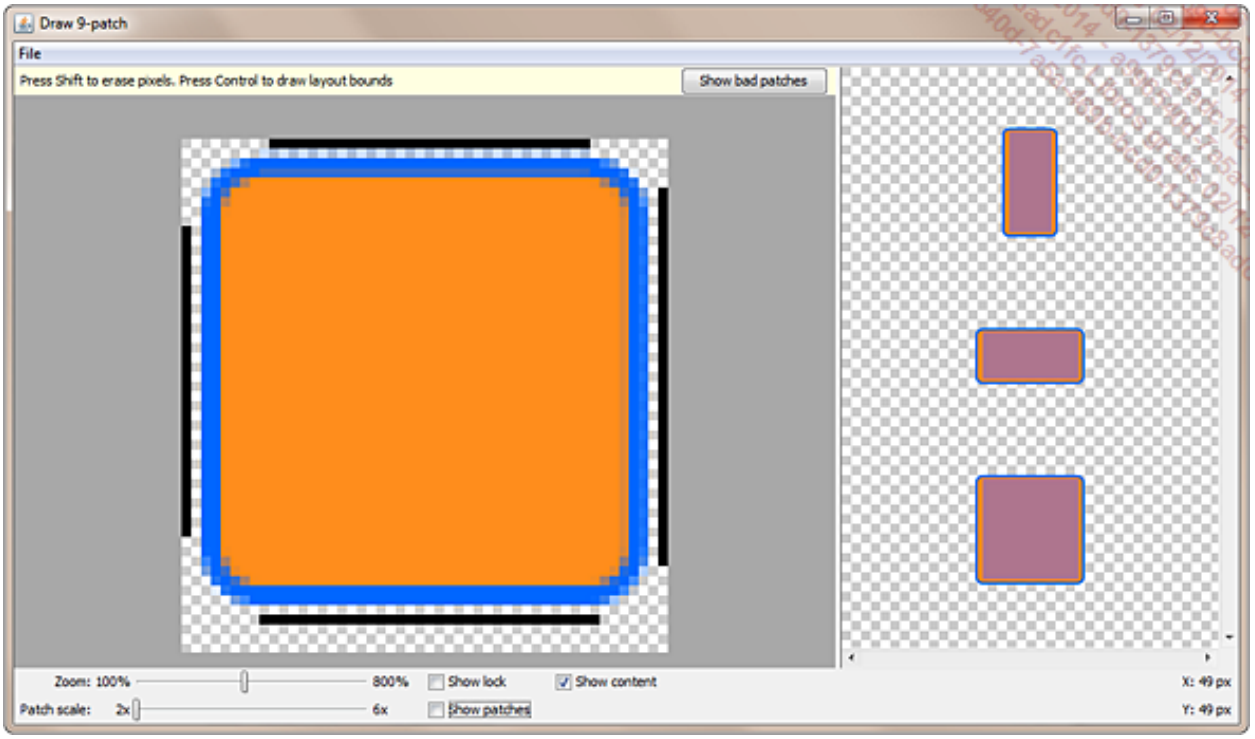
La ventana de la aplicación presenta dos zonas diferenciadas: la zona de la izquierda es la zona de trabajo, en la que puede trazar las zonas de ampliación y de relleno, mientras que la zona derecha muestra una visualización previa.



- Marcando la opción **Show content** es posible visualizar, en color morado en la parte derecha de la aplicación, la zona dedicada al contenido.



- Para definir las zonas basta con trazar, con ayuda del puntero del ratón, las marcas de las zonas en los bordes izquierdo e inferior (para la zona de ampliación) y derecho y superior (para la zona de relleno).



- Tan solo queda guardar el elemento visual creado de este modo (**File** - **Save as 9-patch**) e importarlo en Eclipse.

2. Los drawables XML

Para los elementos visuales simples, tales como los que hemos utilizado en la sección anterior, existe otra técnica que podríamos llamar "orientada al desarrollador", que permite generar gráficos redimensionables: los recursos `drawable` definidos en XML.

En efecto, además de los archivos gráficos clásicos (JPG, PNG, etc.), también es posible definir un recurso `drawable` mediante lenguaje XML. La plataforma Android permite crear formas sencillas (rectángulos, óvalos, líneas y anillos) que pueden colorearse de distintas formas (relleno uniforme o degradado).

En esta sección veremos cómo definir una forma sencilla y, a continuación, repasaremos las distintas opciones disponibles para las formas. Por último, veremos cómo combinar varias formas entre sí.

a. Definir una forma en XML

El archivo de definición de la forma es un archivo XML almacenado en la carpeta `drawable` - como cualquier otro recurso de tipo imagen.

El elemento raíz del archivo es la etiqueta `<shape>`. La sintaxis es la siguiente:

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

Para visualizar las distintas formas, es indispensable definir un color de relleno: por defecto, las formas son transparentes. Como toda modificación en las formas, tal y como veremos en la siguiente sección, la definición del color se realiza mediante una etiqueta hija de la etiqueta`<shape>`.

En primer lugar definiremos un color sólido, mediante la etiqueta `<solid>`. La propiedad `color`de dicha etiqueta permite especificar un color.

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android"
    android:shape="rectangle">
    <solid
            android:color="#888888"/>
</shape>
```

En este primer ejemplo no definiremos dimensiones para la forma: se utilizarán las dimensiones especificadas en la etiqueta `<ImageView>` del archivo de `layout` que utiliza el elemento visual que indique las dimensiones del rectángulo.

Observe que existen dos formas que requieren parámetros específicos, sin los que no pueden mostrarse correctamente:

- La forma "line". Es obligatorio informar una etiqueta `<stroke>` (trazo) para esta forma. Su ausencia provocará una excepción en tiempo de ejecución. El color, si es único, se especifica directamente como propiedad de dicha etiqueta en lugar de encapsularla en la etiqueta`<solid>`.
- En el caso de que la forma seleccionada sea un anillo `(ring)`, existen parámetros suplementarios disponibles para definir las dimensiones del anillo. Estos parámetros permiten definir el ancho del anillo (thickness) y el radio del círculo interior (inner radius). Ambos parámetros pueden expresarse bajo la forma de un porcentaje de la dimensión del elemento visual, o como valor absoluto:
  - `android:innerRadiusRatio`: permite especificar el radio del círculo interior respecto a la dimensión de la imagen. Por ejemplo, para una imagen de `120 dp` de ancho, un anillo cuya propiedad `innerRadiusRatio` valga `6` tendrá un círculo interior de `20dp`(120 / 6).
  - `android:innerRadius`: permite especificar el radio del círculo interior en valor absoluto.
  - `android:thicknessRatio`: permite especificar el ancho del anillo respecto a la dimensión de la imagen. Por ejemplo, para una imagen de `120dp` de ancho, un anillo cuya propiedad `thicknessRatio` valga `4` tendrá un ancho de `40dp`.
  - `android:thickness`: permite especificar el ancho del anillo en valor absoluto.

Preste atención: en el caso de un anillo, la documentación recomienda informar una propiedad suplementaria, `useLevel`, con el valor `false`, pues en caso contrario no se mostrará la forma. ¡No existe ninguna explicación satisfactoria a este respecto!

Antes de ver con más detalle las distintas posibilidades de modificación de una forma, repasaremos algunos ejemplos, con los parámetros más básicos.

```
<shape
   xmlns:android="[...]"
   android:shape="line">
   <stroke
      android:width="2dp"
      android:color="#888888"/>
</shape>
```

```
<shape
   xmlns:android="[...]"
   android:shape="rectangle">
   <solid
```

```
      android:color="#888888"/>
</shape>
```

```
<shape
   xmlns:android="[...]"
   android:shape="oval">
   <solid
      android:color="#888888"/>
</shape>
```

```
<shape
   xmlns:android="[...]"
   android:shape="oval">
   <solid
      android:color="#888888"/>
</shape>
```

No existe ninguna diferencia en la forma respecto a la forma anterior: son las dimensiones de la vista las que determinan si la forma es un óvalo o un círculo.

```
<shape
   xmlns:android="[...]"
   android:shape="ring"
   android:innerRadiusRatio="9"
   android:thicknessRatio="3"
   android:useLevel="false">
   <solid
      android:color="#888888"/>
</shape>
```

```
<shape
   xmlns:android="[...]"
   android:shape="ring"
   android:innerRadiusRatio="6"
   android:thicknessRatio="3"
   android:useLevel="false">
   <solid
      android:color="#888888"/>
</shape>
```

```
<shape
   xmlns:android="[...]"
   android:shape="ring"
   android:innerRadiusRatio="4"
   android:thicknessRatio="4"
   android:useLevel="false">
   <solid
      android:color="#888888"/>
</shape>
```

Para conseguir una mejor legibilidad, la etiqueta `XML` así como la definición del `namespace`se han omitido en la transcripción de los archivos XML de forma. Deben, evidentemente, estar presentes en los archivos XML.

b. Modificar la forma inicial

La etiqueta `<shape>` acepta, como etiqueta hija, las siguientes etiquetas:

- `<corners>`: especifica la forma de las esquinas (únicamente para la forma "rectangle").
- `<gradient>`: especifica un relleno degradado.
- `<padding>`: especifica los márgenes interiores de la vista, para posicionar la forma.
- `<size>`: permite especificar las dimensiones de la forma.
- `<solid>`, que ya hemos visto, permite indicar un color único para el relleno.
- `<stroke>`, que ya hemos visto, permite indicar que la forma posea un contorno.

Veremos con detalle los atributos de cada una de estas etiquetas, y algunos ejemplos de su aplicación.

Etiqueta <corners>

La etiqueta `<corners>` permite especificar, para un rectángulo, el redondeo que se aplicará a sus esquinas.

El atributo `android:radius` indica, en dimensión, el radio del redondeo para el conjunto de esquinas del rectángulo. El valor informado en este atributo debe ser superior a 0. Existe un conjunto de atributos `"android:topLeftRadius"`, `"android:bottomLeftRadius"`, `"android:bottomLeftRadius"` y `"android:bottomRightRadius"` que permiten sobrecargar este valor global para cada una de las esquinas del rectángulo. Estos atributos son dimensiones, y aceptan el valor 0 (que se corresponde con un ángulo recto para la esquina afectada).

El siguiente elemento visual muestra un rectángulo cuya propiedad `android:radius` vale `8dp`, y sobrecargada a `0dp` para las esquinas inferiores derecha e izquierda. El `ImageView` que muestra la forma posee un ancho de `100dp` y una altura de `60dp`.

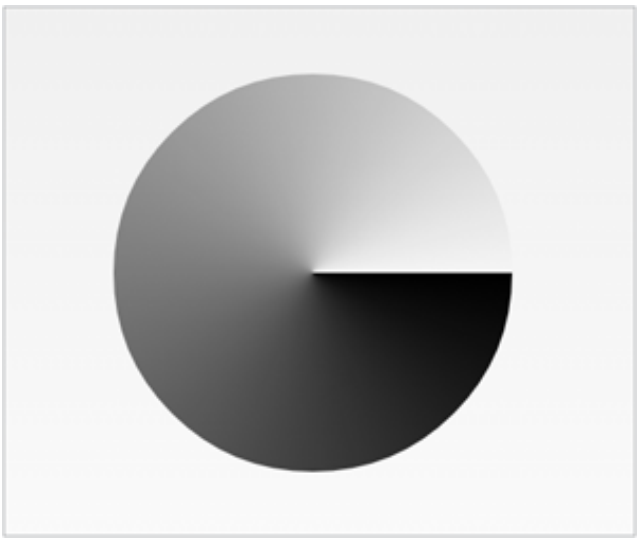
Etiqueta <gradient>

La etiqueta <gradient>, remplazando a <solid>, indica que el relleno debe ser de tipo degradado. La plataforma Android autoriza tres tipos de degradado: lineal, radial y sweep (de tipo circular), y ofrece la posibilidad de definir dos o tres colores para el efecto degradado (color de inicio - startColor, de fin - endColor, y de transición - centerColor).

Según el tipo de degradado, existen varias opciones disponibles:

- android:angle: degradado lineal únicamente. Define el ángulo que se desea utilizar para representar el degradado. El valor debe ser un múltiplo de 45. Un ángulo definido de 0 - valor por defecto - indica un degradado que va de la izquierda de la forma hacia la derecha (startColor situado a la izquierda, endColor a la derecha). Con un valor de 45, el degradado se realizará desde la esquina inferior izquierda de la forma hasta la esquina superior derecha. Un valor de 90 indica un degradado de abajo a arriba.
- android:gradientRadius: degradado radial únicamente. Define el radio del degradado.
- android:centerX y android.centerY: degradado radial y sweep. Define el centro del degradado, en posición relativa (0: origen, 1: fin). Los valores por defecto son 0.5 para ambos atributos, lo que corresponde con el centro de la forma.

El siguiente elemento visual utiliza un gradiente de tipo sweep, cuyo color de inicio es "negro", y el color de fin es "transparente".



Etiqueta <padding>

La etiqueta <padding> es algo particular en el sentido de que no se aplica directamente sobre la forma, sino al contenido de la vista (en el caso de que el elemento visual definido en XML se utilice como imagen de fondo, por ejemplo). Los atributos de esta etiqueta son, clásicamente,android:left, android:right, android:top y android:bottom, y permiten especificar el espacio entre el contenido y el borde de la vista.

Etiqueta <size>

Esta etiqueta permite especificar las dimensiones de la forma. Sus dos atributos son, por tanto,android:width y android:height.

Es preciso, no obstante, tener en mente que, por defecto, el elemento visual definido se adapta a las dimensiones de la vista que lo incluye: las dimensiones indicadas en la etiqueta <size> sirven, por tanto, para indicar la relación ancho/alto de la forma. Si desea, en el marco de una vistaImageView, que las dimensiones especificadas en la forma se utilicen de manera estricta, es preciso declarar el atributo scaleType con el valor "center" en la etiqueta del imageView.

Etiqueta <solid>

Ya hemos mencionado la etiqueta <solid>, que incluye un único atributo, android:color, que permite indicar el color de relleno de la forma.

Etiqueta <stroke>

La etiqueta <stroke> permite indicar que la forma tendrá los bordes con dibujo. Existen otros dos atributos, android:width y android:color, que permiten indicar respectivamente el ancho y el color del borde, y posee dos atributos que permiten obtener efectos difícilmente realizables de otra forma: android:dashWidth, que permite especificar que el borde esté dibujado de forma punteada (dashWidth indica el ancho de cada punto) y android:dashGap, que indica la distancia entre dos puntos.

El siguiente elemento visual muestra un componente TextView declarado con un fondo formado por un rectángulo con un borde punteado (y un relleno "solid" transparente). También se utiliza una etiqueta <padding> para dejar espacio entre el texto y el borde.

El archivo XML correspondiente al fondo es el siguiente:

```
<?xml version="1.0" encoding="utf-8"?>
<shape
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:shape="rectangle">
    <padding
        android:left="8dp"
        android:right="8dp"
        android:top="4dp"
        android:bottom="4dp"/>
    <corners
        android:radius="6dp"/>
    <solid
            android:color="@android:color/transparent"/>
    <stroke
        android:width="2dp"
        android:color="#888888"
        android:dashWidth="5dp"
        android:dashGap="2dp"/>
</shape>
```

El interés resulta inmediato: un fondo definido con 9-patch no podría adaptarse correctamente a las dimensiones de la vista, y el borde punteado no se mostraría correctamente.

c. Combinar varias formas

Para combinar varias formas en un único elemento visual, vamos a utilizar un objetoLayerDrawable, que se define también en un archivo XML. La etiqueta XML correspondiente es<layer-list>, y su sintaxis es la siguiente:

```
<?xml version="1.0" encoding="utf-8"?>
<layer-list
    xmlns:android="http://schemas.android.com/apk/res/android" >
    <item>
      [...]
   </item>
```

```
    <item>
      [...]
   </item>
    <item>
      [...]
   </item>
</layer-list>
```

Cada etiqueta `<item>` hija de `<layer-list>` representa un recurso `drawable`, bien sea un`drawable` XML o cualquier otro `drawable` (incluso recursos PNG).

Los elementos `item` se dibujan los unos a continuación de los otros, en orden: el primer elemento`item` se dibujará en primer lugar, a continuación debajo el segundo, y así sucesivamente.

Para definir cada capa de manera fina, la etiqueta `item` posee los atributos `android:left`,`android:right`, `android:top` y `android:bottom`. Estos atributos aceptan únicamente valores absolutos, definidos típicamente en `dp`, y definen, según la documentación oficial, el desfase ("offset") del ítem respecto a los bordes del conjunto del elemento visual.

No obstante, esta noción de desfase resulta algo equivocada: en teoría un desfase no modifica la forma. O, en el caso que nos ocupa, el desfase produce una modificación de la forma, que adapta sus dimensiones al espacio disponible. Es, por ello, más conveniente ver estos atributos `left`,`right`, `top` y `bottom` como los que permiten definir la zona rectangular en la que se dibujará la forma.

Existen otros atributos disponibles: `android:drawable`, para indicar qué recurso `drawable`utilizar (aunque, en el caso de una forma XML, es posible definir una etiqueta hija de `<item>`), y`android:id`, para definir un identificador específico para cada ítem (que podrá, de este modo, manipularse en el código java de manera independiente a los demás).

Representación en pantalla compleja

Para terminar este capítulo acerca de las técnicas disponibles para reducir la carga de desarrollo de una aplicación profesional, vamos a repasar algunas nociones que simplifican la representación en pantalla.

En efecto, puede resultar algo frustrante, para el desarrollador que se encuentra en situación de construir una aplicación profesional, darse cuenta de que, si bien maneja de manera global todas las nociones de programación, no llega a obtener el diseño esperado por el cliente o, directamente, por el equipo que ha concebido este diseño.

Evidentemente, no existe ninguna regla absoluta a este respecto: cada desarrollador adquiere su propia técnica, basada en su propia experiencia y sus aspiraciones. Hay quien prefiere diseñar una pantalla específica para cada resolución, otros trabajan principalmente manipulando vistas mediante código java. Nosotros hemos dado preferencia al diseño de vistas únicas, que se aplican para todas las configuraciones de pantalla, definidas por completo en XML (el famoso archivo de `layout` de una actividad).

1. Seleccionar el layout

El primer punto a tener en cuenta, cuando se diseña una pantalla, es la naturaleza del `layout`sobre el que se basará el diseño. Habitualmente, se plantea la pregunta de si utilizar un`linearLayout` o un `relativeLayout`. Si bien el primero tiene como ventaja su enorme simplicidad a la hora de implementarlo, no permite diseñar interfaces que se adaptan correctamente a cada tamaño de pantalla.

El `layout` relativo, en sí mismo, es más complicado de implementar: el posicionamiento relativo de cada elemento de la pantalla requiere más trabajo que el hecho de sencillamente enumerar una lista de controles en un orden definido.

No obstante, esta experiencia demuestra que rara vez se tiene una pantalla que muestra una lista de controles, todos posicionados unos debajo de los otros. Por este motivo, una interfaz compleja es, en la mayoría de casos, sinónimo de `relativeLayout`, por motivos obvios: la posibilidad de situar una vista de manera relativa a su padre, la capacidad del `relativeLayout` para solapar vistas (el término "vista" hace referencia, aquí, al objeto `view`, del que heredan todos los componentes visuales de Android).

2. Posicionamiento relativo

Hemos indicado rápidamente la característica del `relativeLayout` para situar las vistas unas respecto a las otras. Veamos en detalle su uso.

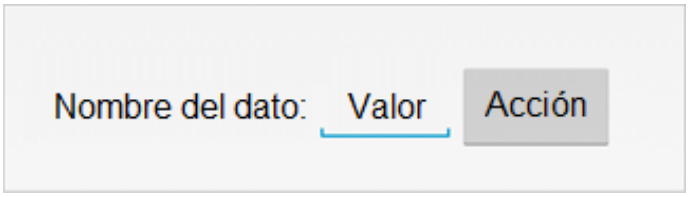
Cuando se sitúa en un `relativeLayout` una vista, bien sea un `TextView`, un `ImageView`, o vistas más complejas como un `listView` o incluso un `relativeLayout`, resulta indispensable indicar su posicionamiento respecto a otra vista, o respecto a su elemento padre. Los principales atributos que permiten especificar esta característica son los siguientes:

`layout_toLeftOf="[id]"`	La vista se posiciona a la izquierda del objeto cuyo identificador es [id].
`layout_toRightOf="[id]"`	La vista se posiciona a la derecha del objeto cuyo identificador es [id].
`layout_below="[id]"`	La vista se posiciona debajo del objeto cuyo identificador es [id].
`layout_above="[id]"`	La vista se posiciona encima del objeto cuyo identificador es [id].

layout_alignParentLeft	Indica si la vista debe alinearse a la izquierda en la vista padre (true|false).
layout_alignParentRight	Indica si la vista debe alinearse a la derecha en la vista padre (true|false).
layout_alignParentBottom	Indica si la vista debe posicionarse en la parte inferior de la vista padre (true|false).
layout_alignParentTop	Indica si la vista debe posicionarse en la parte superior de la vista padre (true|false).

Observe que si alguno de estos atributos de posicionamiento no se informa, la vista se situará en la parte superior izquierda de la vista padre (el objeto que contiene a esta vista).

La ilustración más simple de estas características es un clásico de los formularios de información de datos: una etiqueta (un textView) seguida de una zona de introducción de texto (un EditText). Por ejemplo:



El archivo de layout correspondiente es el siguiente:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_margin="16dp" >
   <TextView
       android:id="@+id/label"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content"
       android:layout_alignParentLeft="true"
       android:layout_centerVertical="true"
       android:text="Nombre del dato: "
       android:textSize="18sp"/>

   <EditText
       android:id="@+id/edit"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content"
       android:layout_toRightOf="@+id/label"
       android:layout_centerVertical="true"
       android:text="Valor"/>

   <Button
       android:id="@+id/bouton"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content"
       android:text="Acción"
       android:layout_toRightOf="@+id/edit"
       android:layout_centerVertical="true"/>

</RelativeLayout>
```

Como puede observarse, el control EditText se posiciona a la derecha del control TextView, y el control Button está ubicado a su vez a la derecha del control EditText. El control TextViewestá posicionado de manera relativa al control padre, RelativeLayout en el ejemplo.

Este ejemplo de implementación sirve, principalmente, a modo ilustrativo. En la realidad, no se aconseja aplicar este tipo de representación, ¡que es una simple extrapolación de los formularios propios de aplicaciones de oficina!

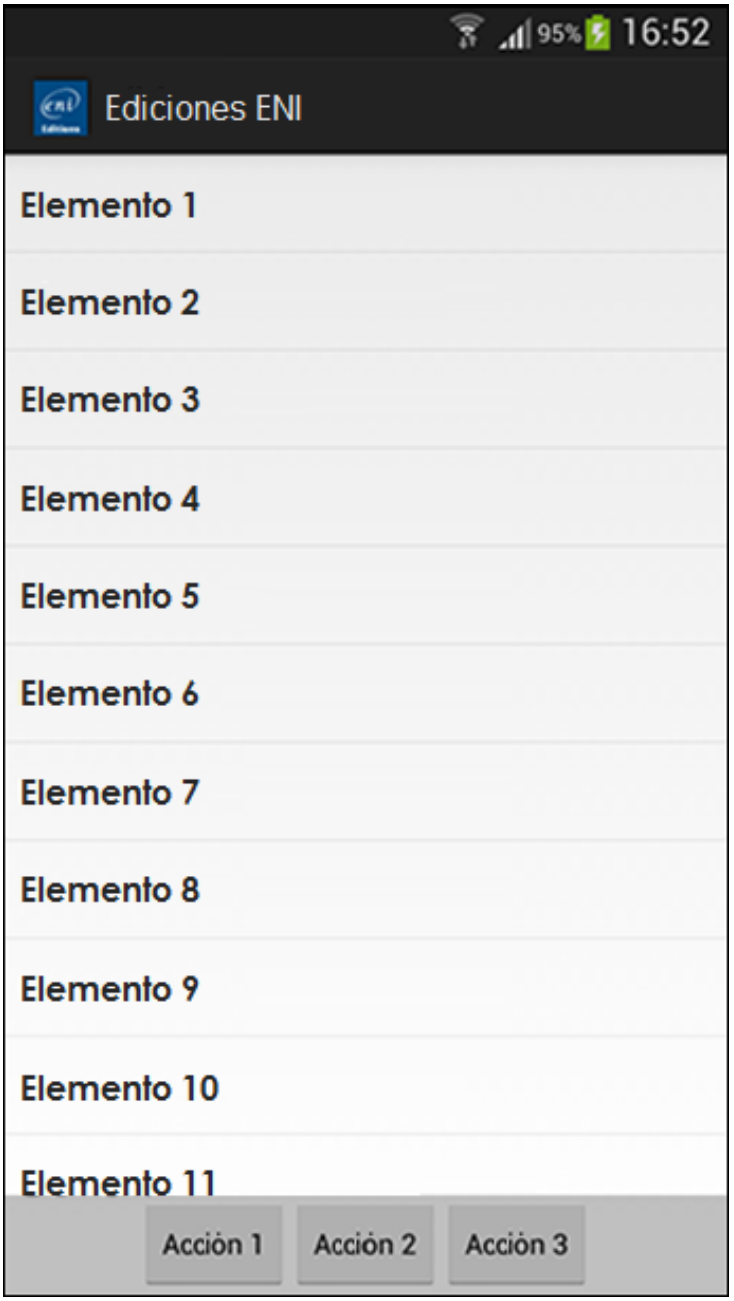
3. Superposición de vistas

A priori, al desarrollador no le interesa, jamás, la superposición de vistas: ¡es la mejor forma de producir una interfaz poco legible! No obstante es, en algunos casos concretos, la manera más sencilla, aunque no la más elegante, de diseñar una interfaz sin tener que recurrir a un posicionamiento absoluto.

El caso más típico que presenta este problema a los desarrolladores es la situación en la que la pantalla requiere una vista que utiliza la totalidad de la pantalla, a excepción de una zona, en general en la parte inferior de la pantalla, que está reservada a un panel con botones de acción.

Este diseño, si bien no es una recomendación de Google en términos de ergonomía, es típico de aplicaciones iPhone: en efecto, los iPhones no poseen tecla "menú", de modo que las acciones más corrientes se sitúan en la zona inferior de la pantalla.

La siguiente imagen ilustra este tipo de representación.



El problema es el siguiente: la altura de la pantalla no se conoce de antemano, de modo que resulta imposible asignar una dimensión fija al componente `ListView`. Si el desarrollador declara una altura correspondiente a la de la pantalla para esta lista, no existe espacio para la barra de acción.

Aquí, la solución más sencilla consiste en utilizar la superposición, permitida por el`relativeLayout`.

El truco consiste, por tanto, en declarar que la `listView` ocupe todo el alto de la pantalla (`android:layout_height="match_parent"`) y declarar un margen que ocupe exactamente la altura de la barra de acción para la zona inferior con la lista (`android:layout_marginBottom`).

La barra de acción se declara alineada en la zona inferior de su elemento padre (`android:layout_alignParentBottom=true`).

De este modo, todos los elementos de la lista estarán visibles, ¡y la representación se vuelve independiente del tamaño de la pantalla!

El código del `layout` correspondiente es el siguiente:

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

```
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >
   <ListView
       android:id="@+id/list"
       android:layout_width="match_parent"
       android:layout_height="match_parent"
       android:layout_marginBottom="50dp"/>
   <LinearLayout
       android:id="@+id/panel_Action"
       android:layout_width="match_parent"
       android:layout_height="50dp"
       android:gravity="center_horizontal"
       android:layout_alignParentBottom="true"
       android:orientation="horizontal"
       android:background="#88888888">
       <Button
           android:layout_width="wrap_content"
           android:layout_height="wrap_content"
           android:textSize="14sp"
           android:text="Acción 1"/>
       <Button
           android:layout_width="wrap_content"
           android:layout_height="wrap_content"
           android:textSize="14sp"
           android:text="Acción 2"/>
      <Button
           android:layout_width="wrap_content"
           android:layout_height="wrap_content"
           android:textSize="14sp"
           android:text="Acción 3"/>
   </LinearLayout>
</RelativeLayout>
```

Observe también el uso conjunto de un `relativeLayout` para la representación global de la vista y de componentes `linearLayout` para las zonas que requieren un posicionamiento complejo (en este caso, los botones de la barra de acción).

4. Un último detalle

Los puristas podrán encontrar un defecto en el último ejemplo de la sección anterior: la dimensión de la barra de acción y, por tanto, del margen aplicado a la lista, recibe valor directamente en el código del `layout`.

Esto puede resultar problemático. Imaginemos que, para los dispositivos que tengan una pantalla poco ancha, el conjunto de botones puede no caber en una sola "línea". Será preciso, en estos casos, prever dos líneas de botones, y aumentar por tanto la altura de la barra de acción. Resultará no obstante algo incómodo, y contrario al mantenimiento sencillo de la aplicación, ¡definir un`layout` para cada tamaño de pantalla!

Por este motivo, entre otros, se recomienda encarecidamente no declarar directamente una dimensión directamente en el código del `layout`, sino pasar los valores definidos en un archivo de recursos de tipo `<dimen>`, que definirá las dimensiones utilizadas por la aplicación (noción que se estudia en la sección Estructura de un proyecto Android - Los recursos del capítulo Primeros pasos).

A continuación se muestra un extracto de código de definición de dimensiones:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <dimen name="altura_barraAccion">50dp</dimen>
</resources>
```

El uso de esta dimensión en el `layout` que se presentó en el ejemplo anterior de la sección Superposición de vistas será como se indica a continuación:

```
<ListView
    android:id="@+id/list"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_marginBottom="@dimen/altura_barraAccion"/>
```

De este modo, gracias al mecanismo de recursos específicos para cada tamaño de pantalla, basta con modificar, para cada tamaño de pantalla que requiera un procesamiento particular, un archivo de recursos con los valores adecuados.

Introducción

Android está basado en el sistema Linux. Por tanto, retoma las principales características y, en concreto, su funcionamiento.

En este capítulo, vamos a descubrir la noción de proceso aplicado a las aplicaciones Android. A continuación, detallaremos cómo crear y ejecutar threads secundarios dedicados a procesamientos costosos. Veremos cómo pueden comunicarse con el thread principal para actualizar la interfaz de usuario. Por último, estudiaremos la implementación de la seguridad en Android y, en especial, el uso de los permisos de las aplicaciones y los componentes.

Procesos

Por defecto, toda aplicación Android se ejecuta en su propio proceso Linux. Para ser más precisos, cuando el sistema debe ejecutar por primera vez un componente de una aplicación, crea un nuevo proceso Linux. Ejecuta, en este proceso, una máquina virtual Dalvik, y carga la aplicación y ejecuta el componente deseado en un sólo y único thread, el thread principal.

Como se ha indicado en varias ocasiones a lo largo de este libro, el sistema puede decidir, en cualquier momento, matar el proceso entero para liberar recursos del sistema para las demás aplicaciones. El fin del proceso provoca el fin de la aplicación, es decir, de todos sus componentes, sin distinción. Se ejecutará un nuevo proceso cuando se tenga que volver a usar alguno de los componentes de la aplicación.

El nombre del usuario Linux creado para ejecutar este proceso es del estilo *app_id* donde id es un número único por aplicación. Por defecto, el sistema crea un usuario por proceso y, por tanto, por aplicación.

1. android:process

El proceso toma el nombre del paquete de la aplicación. Su nombre puede modificarse especificando el atributo `android:process` de la etiqueta `applicacion` del manifiesto. Se recomienda respetar el formato utilizado para la nomenclatura de los paquetes.

Sintaxis

```
<application
  android:process="[:]cadena de caracteres"
  ... >
    ...
</application>
```

Ejemplo

```
<application
  android:icon="@drawable/icon"
  android:label="@string/app_name"
  android:process="es.midominio.android.miaplicacion.miproceso">
</application>
```

Por defecto, todos los componentes de la misma aplicación se ejecutarán en este mismo proceso, bajo el mismo nombre, y en el mismo thread principal. No obstante, cada componente puede saltarse esta regla fácilmente. Para ello, basta con que utilice el atributo `android:process` en su etiqueta para indicar el nombre del nuevo proceso en el que se ejecutará el componente. Si el nombre viene precedido por un signo de dos puntos, el proceso será privado a la aplicación. Es decir, su nombre estará precedido por el nombre del paquete de la aplicación seguido del nombre especificado.

Ejemplo

```
<activity android:name=".MiActividadSecundaria"
          android:process=":otroProceso" />
```

En este ejemplo, la actividad secundaria se ejecuta en su propio proceso llamado`es.midominio.android.miaplicacion:otroProceso`.

Dado que Android está basado en el sistema Linux, es posible acceder a un shell utilizando la herramienta `adb`. Ésta permite ejecutar distintos comandos.

Sintaxis

```
adb shell [comando]
```

Ejemplo

```
$ adb shell
$ ps
USER   PID   PPID  VSIZE  RSS    WCHAN     PC         NAME
root   1     0     224    208    ffffffff  00000000 S /init
root   2     0     0      0      ffffffff  00000000 S kthreadd
root   3     2     0      0      ffffffff  00000000 S ksoftirqd/0
...
app_76 2009  58    129020 20156 ffffffff  00000000 S
es.midominio.android.miaplicacion
app_76 2017  58    136976 20112 ffffffff  00000000 S
es.midominio.android.miaplicacion:otroProceso
```

En este ejemplo, una vez conectado al shell, el comando `ps` permite listar los procesos actuales y, en especial, aquellos correspondientes a una aplicación o, más específicamente, a un componente de una aplicación. Este es el caso del proceso `2009` correspondiente a la aplicación, y a la actividad principal, `es.midominio.android.miaplicacion` con el proceso `2017` correspondiente a la actividad secundaria.

2. Compartición de proceso

Una de las ventajas de especificar el nombre del proceso que se quiere crear y utilizar es poder compartirlo, no sólo entre distintos componentes de una misma aplicación, sino también, y sobretodo, entre distintas aplicaciones. Es decir, varias aplicaciones que especifiquen el mismo nombre de proceso se ejecutarán en el mismo proceso, permitiéndoles acceder a él fácilmente y compartir los mismos recursos: memoria, archivos, preferencias…

Para ello, estas aplicaciones deben no sólo especificar el mismo nombre de proceso mediante el atributo `android:process` sino que también deben compartir el mismo identificador de usuario Linux y estar firmadas con el mismo certificado digital (véase el capítulo Publicar una aplicación - Firma digital de la aplicación).

El identificador de usuario Linux es un identificador único atribuido por el sistema a cada aplicación. Es posible, no obstante, especificar uno de modo que varias aplicaciones usen el mismo. Para ello, hay que utilizar el atributo `android:sharedUserId`. El nombre que se pasa como valor debe contener, al menos, un punto.

Sintaxis

```
<manifest
    android:sharedUserId="cadena de caracteres"
    ... >
  ...
</manifest>
```

Ejemplos

Se muestra a continuación el archivo AndroidManifest.xml de la aplicación MiAplicacion.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
    xmlns:android="http://schemas.android.com/apk/res/android"
    package="es.midominio.android.miaplicacion"
    android:sharedUserId="es.midominio.android.usuario1" >
  <application android:icon="@drawable/icon"
    android:label="@string/app_name"
    android:process="es.midominio.android.proceso.compartido" >
```

```
  </application>
</manifest>
```

Y he aquí el archivo AndroidManifest.xml de la aplicación MiAplicacion2.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
    xmlns:android="http://schemas.android.com/apk/res/android"
    package="es.midominio.android.miaplicacion2"
    android:sharedUserId="es.midominio.android.usuario1" >
  <application android:icon="@drawable/icon"
    android:label="@string/app_name"
    android:process="es.midominio.android.proceso.compartido" >
  </application>
</manifest>
```

En este ejemplo, las dos aplicaciones comparten el mismo identificador de usuario`es.midominio.android.usuario1` y el mismo proceso`es.midominio.android.proceso.compartido`. Estas aplicaciones se ejecutan entonces en el mismo proceso del sistema y pueden compartir de forma abierta sus recursos, como si fueran una única aplicación.

Una verificación desde el shell nos confirma que ambas aplicaciones corren en el mismo proceso.

Ejemplo

```
$ ./adb shell ps
USER     PID   PPID  VSIZE  RSS     WCHAN    PC         NAME
root      1     0     224    208   ffffffff 00000000 S /init
root      2     0     0      0     ffffffff 00000000 S kthreadd
...
app_85    2470  58    142400 20608 ffffffff 00000000 S
es.midominio.android.proceso.compartido
```

Programación concurrente

Por defecto, un proceso sólo comprende un thread, el thread principal. Todos los componentes de la aplicación se ejecutan en este thread. Para mejorar la experiencia de usuario, Android considera que una aplicación está bloqueada cuando no responde tras diez segundos. El usuario puede destruirla.

Para evitar tal bloqueo de la aplicación, cualquier procesamiento largo como, por ejemplo, una descarga web o un cálculo intensivo deben realizarse en un thread secundario dedicado, liberando al thread principal. Este último puede entonces dedicarse al funcionamiento global de la aplicación y a su representación gráfica, tarea que se realiza de forma obligatoria en el thread principal.

Es posible crear tantos threads secundarios como se desee. Se recomienda encarecidamente crear threads antes que procesos, pues estos últimos consumen más recursos.

Si bien no es obligatorio, en muchos casos puede ser oportuno crear servicios para ejecutar los threads secundarios. En efecto, llegado el caso, el sistema matará de manera prioritaria aquellos procesos que no muestren nada al usuario o los receptores de eventos sin actividad antes que los servicios. La clase `IntentService` puede ser otra gran idea en este sentido. En efecto, permite crear rápidamente servicios que integran un thread secundario, y gestionarlos fácilmente. Es, por tanto, una solución suplementaria a tener en cuenta además de las que se detallan en este capítulo.

Es común ejecutar procesamientos largos y querer visualizar sus resultados una vez terminados. O bien, el procesamiento largo debe realizarse en un thread secundario y su visualización a través la interfaz de usuario en el thread principal. Es preciso, por tanto, que estos threads puedan comunicarse mutuamente.

Para ello, existen distintas técnicas a nuestra disposición.

1. AsyncTask

Como su propio nombre indica, la clase `AsyncTask` permite realizar una tarea de forma asíncrona. Es, sin duda, la forma más sencilla de crear y ejecutar un thread secundario y, a continuación, mostrar el resultado en el thread principal.

Esta clase permite abstraerse de la manipulación de threads, y no tener que preocuparse más que del procesamiento de fondo a realizar y de la gestión de resultados. La creación del thread secundario se realiza de forma interna y la visualización de resultados se realiza en el thread principal. El desarrollador no tiene que preocuparse de la creación del thread secundario, de la gestión de estos threads ni de la comunicación entre ellos.

La clase `AsyncTask` proporciona métodos que se invocarán automáticamente en cada etapa de la tarea: la inicialización, la ejecución del procesamiento, el progreso y la finalización. El desarrollador no tiene que invocar a estos métodos directamente.

Es preciso, por tanto, crear una clase que herede de la clase `AsyncTask` e implementar los métodos deseados. La clase `AsyncTask` recibe tres tipos genéricos:

- `Params`: tipo de los parámetros pasados a la entrada de la tarea.
- `Progress`: tipo de la unidad de progreso del procesamiento.
- `Result`: tipo del resultado del procesamiento.

Sintaxis

```
class MiTarea extends AsyncTask<Params, Progress, Result> {
    ...
}
```

Ejemplo

```
public class MiActividadPrincipal extends Activity {
  private class NumerosPrimos extends
      AsyncTask<Integer, Integer, Integer> {
    ...
  }
}
```

La declaración de la clase que hereda de la clase `AsyncTask` se realiza generalmente como clase privada interna al componente utilizado, como ocurre en nuestro ejemplo.

La primera etapa de la realización de la tarea es su inicialización, que se lleva a cabo implementando el método `onPreExecute`. Este método se invoca desde el thread principal y puede, por tanto, modificar la interfaz de usuario.

Sintaxis

```
protected void onPreExecute ()
```

Ejemplo

```
@Override
protected void onPreExecute() {
  super.onPreExecute();
  Toast.makeText(MiActividadPrincipal.this,
    "¡Cálculo de los números primos iniciado!",
    Toast.LENGTH_SHORT)
  .show();
}
```

A continuación se invoca al método `doInBackground` que se encarga de realizar el procesamiento de la tarea. Este método se invoca desde el thread secundario. Esto permite ejecutar el procesamiento largo y no bloquear el thread principal. Este método devuelve el resultado del procesamiento.

Sintaxis

```
protected abstract Result doInBackground (Params... params)
```

Ejemplo

```
@Override
protected Integer doInBackground(Integer... arg0) {
  int n = 0;
  int nivel=0;
  int step = (arg0[1] - arg0[0]) / 10;
  for (int i = arg0[0]; i <= arg0[1]; i++) {
    if (isPrime(i)) {
      n++;
    }
    if ((i > arg0[0]) && (i % step == 0))
      publishProgress(++nivel);
  }
  return n;
}
```

El procesamiento contenido en el método `doInBackground` puede invocar en cualquier momento al método `publishProgress` para actualizar la interfaz de usuario según el avance actual del procesamiento. Esta llamada provocará que se invoque el método `onProgressUpdate` desde el thread principal.

Sintaxis

```
protected void onProgressUpdate (Progress... values)
```

Ejemplo

```
@Override
protected void onProgressUpdate(Integer... values) {
  super.onProgressUpdate(values);
  Toast.makeText(MiActividadPrincipal.this,
    values[0]+"0% completado", Toast.LENGTH_SHORT).show();
}
```

Una vez realizado el procesamiento, se invoca el método `onPostExecute` para procesar el resultado. Este método se invoca desde el thread principal y puede, por tanto, modificar la interfaz de usuario.

Sintaxis

```
protected void onPostExecute (Result result)
```

Ejemplo

```
@Override
protected void onPostExecute(Integer result) {
  super.onPostExecute(result);
  Toast.makeText(MiActividadPrincipal.this,
    result + " números primos encontrados en total.",
    Toast.LENGTH_SHORT).show();
}
```

Es posible detener la tarea invocando a su método `cancel`. Para tener en cuenta esta anulación lo más rápidamente posible, el procesamiento contenido en el método`doInBackground` deberá verificar de forma regular el valor de retorno del método`isCancelled`. En tal caso, se invocará al método `onCancelled` en lugar de al método`onPostExecute`.

Sólo queda por utilizar esta clase para ejecutar el procesamiento. Para ello, hay que instanciar la clase e invocar a su método `execute` para ejecutar la tarea. Este método acepta como entrada uno o varios parámetros del tipo genérico `Params`.

Una tarea sólo puede procesarse una única vez. Es preciso crear una nueva instancia para ejecutar un nuevo procesamiento.

Sintaxis

```
public final AsyncTask<Params, Progress, Result>
  execute (Params... params)
public final boolean cancel (boolean mayInterruptIfRunning)
```

Ejemplo

```
public class MiActividadPrincipal extends Activity {

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

```
    NumerosPrimos tarea = new NumerosPrimos();
    tarea.execute(0, 10000000);
  }
}
```

En este ejemplo, la tarea se crea y ejecuta desde la creación de la actividad.

2. Thread

Si bien es práctica, la clase `AsyncTask` puede que no nos convenga en ciertos casos. Otra solución consiste en utilizar la clase `Thread`. Esta clase permite definir y ejecutar un procesamiento. Es posible ejecutar varios threads concurrentes y sincronizarlos entre ellos.

La comunicación entre este thread y el thread principal puede realizarse mediante el uso de objetos de tipo `Handler`.

a. Creación

Existen dos formas de crear un thread.

La primera forma de crear un thread consiste en instanciar la clase `Thread` y pasarle como parámetro un objeto de tipo `Runnable`, este último permite especificar el procesamiento en su método `run`.

Sintaxis

```
public Thread (Runnable runnable)
```

Sintaxis del método run de la clase Runnable

```
public abstract void run ()
```

Ejemplo

```
Thread thread = new Thread(new Runnable() {
  @Override
  public void run() {
    Log.d(TAG, "Ejecución del thread");
  }
});
```

La otra forma de crear un thread consiste en instanciar una clase que herede de la clase `Threade` implementando el método `run`.

Sintaxis

```
public void run ()
```

Ejemplo

```
public class MiThread extends Thread {
  private static final String TAG = "MiThread";

  @Override
  public void run() {
    super.run();
    Log.d(TAG, "Ejecución del procesamiento");
  }
```

```
}
```

En ambos casos de creación del thread, la ejecución del procesamiento se realiza invocando almétodo `start` del objeto de tipo `Thread`.

Sintaxis

```
public synchronized void start ()
```

Ejemplo

```
MiThread thread = new MiThread();
thread.start();
```

b. runOnUIThread

Si un thread secundario quiere ejecutar código sobre el thread principal, por ejemplo para actualizar directamente la interfaz de usuario, puede hacerlo utilizando el método`runOnUIThread`.

Este método recibe como parámetro un objeto de tipo `Runnable` que contiene el código a ejecutar en el thread principal. Esto evita tener que implementar un sistema de comunicación entre ambos threads cuando se trata únicamente de modificar la interfaz de usuario.

Sintaxis

```
public final void runOnUiThread (Runnable action)
```

Ejemplo

```
runOnUiThread(new Runnable() {
  public void run() {
    Toast.makeText(MiActividadPrincipal.this, "blablabla",
      Toast.LENGTH_SHORT)
    .show();
  }
};
```

c. Comunicación interthread

La clase `Handler` permite a distintos threads comunicarse entre ellos. En concreto, un objeto de tipo `Handler` puede enviar mensajes al thread una vez lo ha creado.

La clase `Handler` es también un medio sencillo para enviar mensajes desde y hacia un mismo thread. Esto permite planificar el procesamiento de ciertos mensajes o procesamientos en el tiempo sin tener por qué recurrir a un sistema de alarmas o algún equivalente más adecuado.

Estos mensajes son bien objetos de tipo `Message`, o bien objetos de tipo `Runnable`. Un objeto de tipo `Message` puede, entre otros, comportar los siguientes datos: el sujeto o tipo de mensaje mediante un entero, dos enteros y un objeto de tipo `Object`.

La creación de un objeto de tipo `Handler` se realiza instanciando directamente a la clase`Handler` utilizando el constructor por defecto. Este handler está asociado al thread que lo ha creado. Esto provoca que, si la instanciación tiene lugar directamente en la clase de un componente de aplicación, el thread del handler es el thread principal. En este caso, esto permite a un thread secundario comunicarse con el thread principal que puede modificar la interfaz de usuario.

La recepción de los mensajes enviados al objeto de tipo `Handler` se realiza implementando elmétodo `handleMessage`. Este método recibe como parámetro el objeto de tipo `Message`recibido.

Sintaxis

```
public Handler ()
public void handleMessage (Message msg)
```

Ejemplo

```
public class MiActividadPrincipal extends Activity {
  private Handler handler = new Handler() {
    @Override
    public void handleMessage(Message msg) {
      switch (msg.what) {
      case MSG_PROCESAMIENTO_A:
        procesamientoA();
        break;
      case MSG_PROCESAMIENTO_B:
        procesamientoB();
        break;
      }
    }
  };
}
```

La creación de un objeto de tipo `Message` se realiza indirectamente mediante el objeto de tipo`Handler` utilizando uno de sus métodos `obtainMessage`. Estos métodos permiten especificar como parámetros todos los datos o una parte de ellos.

Sintaxis

```
public final Message obtainMessage ()
public final Message obtainMessage (int what)
public final Message obtainMessage (int what, Object obj)
public final Message obtainMessage (int what, int arg1, int arg2)
public final Message obtainMessage (int what, int arg1, int arg2,
  Object obj)
```

Ejemplo

```
Message msg = handler.obtainMessage(what, obj);
```

Una vez creado el mensaje, basta con enviarlo al handler.

También en este punto, se proveen varios métodos. El primero `sendMessage`, envía el mensaje y lo procesa inmediatamente. El segundo `sendMessageAtTime`, envía el mensaje y lo procesará en el momento indicado en milisegundos desde el arranque del sistema. Por último, el tercer método, `sendMessageDelayed`, envía el mensaje y lo procesa tras un tiempo de espera especificado en milisegundos a partir de la recepción del mensaje.

Sintaxis

```
public final boolean sendMessage (Message msg)
public boolean sendMessageAtTime (Message msg, long uptimeMillis)
public final boolean sendMessageDelayed (Message msg,
  long delayMillis)
```

Ejemplo

```
Message msg = handler.obtainMessage(what, obj);
handler.sendMessage(msg);
```

En este ejemplo, el mensaje lo envía y recibe directamente el método `handleMessage` del

handler.

Si los mensajes que se quiere enviar son sencillos y sólo contienen el asunto, la clase `Handler`proporciona para ello métodos que tienen en cuenta la creación de estos mensajes. La variante de los métodos propuestos es del mismo tipo que el de los métodos `sendMessage`.

Sintaxis

```
public final boolean sendEmptyMessage (int what)
public final boolean sendEmptyMessageAtTime (int what, long uptimeMillis)
public final boolean sendEmptyMessageDelayed (int what, long delayMillis)
```

Ejemplo

```
Message msg = handler.obtainMessage(what, obj);
handler.sendEmptyMessageDelayed(msg, 5000);
```

En este ejemplo, el mensaje lo recibirá el método `handleMessage` del handler en cinco segundos.

Por último, como se ha indicado antes, la clase `Handler` permite a su vez enviar objetos de tipo`Runnable` como mensaje. La variante de los métodos propuestos es del mismo tipo que el de los métodos `sendMessage`.

Sintaxis

```
public final boolean post (Runnable r)
public final boolean postDelayed (Runnable r, long delayMillis)
public final boolean postAtTime (Runnable r, long uptimeMillis)
```

Ejemplo

```
handler.post(new Runnable() {
  public void run() {
    Log.d(TAG, "Código enviado al handler");
    Toast.makeText(MiActividadPrincipal.this,  "Este código ha
sido enviado al handler para la ejecución en su thread.",
    Toast.LENGTH_SHORT).show();
  }
});
```

En este ejemplo, se supone que el handler ha sido creado en el thread principal. Puede, por tanto, mostrar directamente un mensaje de tipo `Toast`.

Seguridad y permisos

Por defecto, Android asigna un identificador de usuario Linux a cada aplicación durante su instalación en el sistema.

Es, por tanto, el sistema de permisos estándar de Linux el que se utiliza para controlar el acceso al espacio de almacenamiento y, por tanto, a las aplicaciones y sus datos. Cada una de las aplicaciones dispone de su propio espacio privado: proceso y espacio de almacenamiento. Puede, no obstante, compartir ciertos datos y archivos especificándolo explícitamente (véase el capítulo La persistencia de los datos - Proveedor de contenidos).

Por defecto, una aplicación no posee ningún permiso que pueda, entre otros, alterar las demás aplicaciones, el sistema Android o que permita acceder a los datos privados del usuario.

Si una aplicación trata de realizar una acción que le está prohibida, se registra un mensaje de error en los logs. También es posible que se eleve una excepción Java de seguridad, según el caso.

El acceso a ciertos datos sensibles, el uso de componentes de hardware del dispositivo Android, el acceso a la red, la posición GPS del usuario o bien otras funcionalidades requieren obligatoriamente que la aplicación declare explícitamente cada uno de los permisos que desea poseer.

Esta lista de permisos se le proporcionará al usuario de forma previa a la descarga de la aplicación, por ejemplo en Play Store. El usuario podrá, entonces, consultarla y decidir, con total conocimiento, si instalar la aplicación o no según los permisos requeridos.

El usuario debe aceptar todos los permisos solicitados por la aplicación para instalarla. Por ello, es preciso no solicitar más permisos que los estrictamente necesarios para el buen funcionamiento de la aplicación. Los usuarios podrán no querer instalar la aplicación si no comprenden por qué se solicitan permisos que parecen injustificados respecto a las funcionalidades que ofrece la aplicación.

Una vez instalada, la aplicación dispone de los permisos solicitados para siempre, hasta la próxima actualización de la aplicación que implique una actualización en la lista de permisos.

1. Declaración de los permisos

La declaración de los permisos se realiza en el manifiesto. La etiqueta `uses-permission` permiteespecificar un permiso requerido. Se utiliza dentro de la etiqueta `manifest`. Es preciso, por tanto, insertar tantas etiquetas `uses-permission` como permisos se soliciten.

Sintaxis

```
<uses-permission android:name="cadena de caracteres">
```

Ejemplo

```
<?xml version="1.0" encoding="utf-8"?>
<manifest ...>
  <uses-permission android:name="android.permission.VIBRATE" />
  <uses-permission android:name="android.permission.SEND_SMS" />
</manifest>
```

En este ejemplo, la aplicación solicita dos permisos: uno para poder hacer vibrar el dispositivo y otro para poder enviar SMS.

La lista completa de permisos proporcionados por el sistema pueden consultarse directamente

en la vista Eclipse Permissions del manifiesto. También es posible consultarla en la siguiente dirección: http://developer.android.com/reference/android/Manifest.permission.html

Observe que una aplicación también puede crear sus propios permisos utilizando la etiqueta`permission` en el manifiesto.

2. Restricción de uso

Cada componente de una aplicación puede solicitar que pueda ser utilizado sólo por aquellos componentes que posean ciertos permisos.

A partir de Android 3.0 (API 11), existe una excepción. Incluso si un componente requiere un permiso específico para poder ser ejecutado, otro componente de la misma aplicación puede utilizarlo sin poseer, él mismo, los permisos solicitados por el componente.

Para ello, debe especificarse el atributo `android:permission` dentro de la etiqueta del componente que quiera controlar su acceso.

Observe que el componente de tipo `ContentProvider` permite especificar de forma más fina el acceso proporcionando otros dos atributos `android:readPermission` y`android:writePermission`. Estos atributos permiten respectivamente indicar quién puede leer y escribir la información en el proveedor de contenidos.

Se elevará una excepción Java de seguridad si un componente cliente utiliza un componente para el que no posee permisos, salvo en el caso del receptor de eventos.

En el caso del envío no autorizado de un evento al receptor de eventos, no se generará ninguna excepción, aunque el receptor de eventos no recibirá este evento. También es posible especificar un permiso requerido que el receptor de eventos tendrá que poseer obligatoriamente para recibir la intención tras su envío (consulte el capítulo Componentes principales de la aplicación - Receptor de eventos).

Sintaxis

```
<activity android:permission="cadena de caracteres" ...>
<service android:permission="cadena de caracteres" ...>
<receiver android:permission="cadena de caracteres" ...>
<provider android:permission="cadena de caracteres" ...>
<provider android:readPermission="cadena de caracteres" ...>
<provider android:writePermission="cadena de caracteres" ...>
```

Ejemplo

```
<?xml version="1.0" encoding="utf-8"?>
<manifest ...>
  <application ...>
    <activity android:name=".MiActividadPrincipal"
              android:permission="android.permission.VIBRATE" >
      <intent-filter>
```

Red

Una de las funcionalidades habituales de la que quiere disponer una aplicación es la capacidad de intercambiar información con un servidor remoto, bien sea sobre una red privada o sobre Internet.

Según se trate de un particular, profesional, …, corresponde al responsable verificar, llegado el caso, si es necesario declarar el uso de datos personales de los usuarios recogidos en los servidores (como por ejemplo sus coordenadas geográficas). Para más información visite la siguiente dirección: http://www.aepd.es/

La duración de un intercambio con un servidor, es decir, el tiempo entre la construcción / envío del mensaje, y el retorno / análisis de la respuesta, depende de muchos factores:

- La construcción y el envío de la consulta, que puede llevar más o menos tiempo según la potencia del dispositivo Android y la complejidad de la consulta.
- La calidad de la red y su velocidad, en especial cuando la conexión se establece mediante la red de telefonía móvil.
- El tiempo de respuesta del propio servidor que puede tardar varios segundos, o decenas de segundos, antes de enviar una respuesta.

El desarrollador es totalmente dependiente de factores cuya duración no puede controlar. Deberían tenerse en cuenta en la etapa del desarrollo de la comunicación con el servidor. En particular, la tarea de comunicación puede ser larga, y tendrá que realizarse en un thread secundario para no bloquear la aplicación.

Para poder comunicarse con un servidor remoto, la primera etapa consiste en dotar a la aplicación de los permisos para abrir sockets de red. Para ello, hay que agregar el permiso`android.permission.Internet` en el manifiesto.

Ejemplo

```
<?xml version="1.0" encoding="utf-8"?>
<manifest ...>
  <uses-permission android:name="android.permission.INTERNET" />
</manifest>
```

Android incluye la librería HTTP Apache que permite utilizar el protocolo HTTP de forma sencilla. Esta librería contiene numerosas clases y métodos. Dado que no es específica de Android, no detallaremos más que algunas de sus clases y ciertos métodos.

Desde la versión 2.2 (API 8), Android proporciona también la clase`AndroidHttpClient` que implementa la interfaz `HttpClient`, configurada especialmente para Android. Esta clase prohíbe, en concreto, el envío de consultas HTTP desde el thread principal, elevando una excepción.

Ejemplo

```
java.lang.RuntimeException: This thread forbids HTTP requests
```

En versiones inferiores a Android 2.2 (API 8), es posible reemplazar la clase`AndroidHttpClient` por la clase `DefaultHttpClient`. El firewall que impide su uso en el thread principal no existe, por lo que habrá que velar por ejecutarla sobre un thread secundario.

Veamos cómo utilizar las clases y los métodos más utilizados de estas librerías para enviar una consulta a un servidor remoto y recuperar la respuesta.

1. Agente usuario

El user-agent (agente usuario) es una cadena de caracteres incluida en el encabezado de una consulta HTTP enviada por un cliente a un servidor. Permite identificar el sistema del cliente indicando, por ejemplo, el nombre de la aplicación cliente, su versión…

El servidor puede entonces utilizar esta información para adaptar mejor su respuesta. Por ejemplo, un servidor web podrá detectar que el cliente es un navegador de un dispositivo móvil. Podrá, entonces, redirigir automáticamente la petición hacia la versión móvil del sitio web.

En el caso en que el desarrollador Android gestione también la parte servidora, puede ser interesante para él construir un agente usuario proporcionando la máxima información posible sobre el dispositivo, el sistema y la aplicación Android. Esto le permitirá ver en detalle qué dispositivos, qué sistemas y versiones del sistema, y qué versiones de las aplicaciones poseen los usuarios.

Para obtener la información asociada al sistema, podemos utilizar algunas de las constantes que proporciona la clase `android.os.Build`. Los valores de estas constantes son cadenas de caracteres.

La constante `Build.MODEL` proporciona el modelo del dispositivo Android en forma de cadena de caracteres. Los valores devueltos por esta constante pueden ser, por ejemplo: `DROID2`, `Nexus One`,`GT-I9000`, `T-Mobile G2`, `X10i`, `ZTE-RACER`, `MB525`, `GT-I5700`...

Sintaxis

```
Build.MODEL
```

Ejemplo

```
String modelo = Build.MODEL;
```

La constante `Build.VERSION.RELEASE` proporciona el número de versión del sistema en forma de cadena de caracteres. Los valores devueltos por esta constante pueden ser, por ejemplo: `3.0`,`2.3.3`, `2.2.2`, `2.1-update1`, `1.5`...

Sintaxis

```
Build.VERSION.RELEASE
```

Ejemplo

```
String versionSistema = Build.VERSION.RELEASE;
```

La constante `Build.DISPLAY` proporciona el nombre de código del sistema en forma de cadena de caracteres. Los valores devueltos por esta constante pueden ser, por ejemplo: `VZW`, `FRG83G`,`ECLAIR`, `FRF91`...

Sintaxis

`Build.DISPLAY`

Ejemplo

```
String codigoSistema = Build.DISPLAY;
```

El código del idioma así como el código regional configurado por el usuario en el sistema Android también pueden recuperarse. Para ello, hay que usar la clase `Locale` y su método `getDefault`. Este método devuelve una instancia de tipo `Locale`. El uso del método `toString` permite recuperar el código del idioma así como el código regional concatenados en forma de cadena de caracteres. Los valores devueltos pueden ser, por ejemplo: `en_US`, `es_ES`, `fr_FR`, `bg_BG`, `zh_TW`…

Sintaxis

```
public static Locale getDefault ()
```

Ejemplo

```
String paramsReg = Locale.getDefault().toString();
```

Android permite obtener información sobre las aplicaciones. En nuestro caso, esto permite recuperar esta información de forma dinámica en lugar de incluirla directamente en nuestro código. El código no se habrá modificado tras la actualización de la aplicación y podrá reutilizarse para otras aplicaciones.

La identificación de una aplicación se realiza mediante el nombre de su paquete. Este nombre se utilizará en los métodos detallados más adelante. Para recuperar dinámicamente el nombre del paquete en curso, hay que usar el método `Context.getPackageName`.

Sintaxis

```
public abstract String getPackageName ()
```

Ejemplo

```
String nombrePaquete = context.getPackageName()
```

La clase `PackageManager` permite recuperar información asociada a las aplicaciones instaladas en el sistema. Es posible recuperar una instancia de esta clase invocando al método estático`getPackageManager`.

Sintaxis

```
public abstract PackageManager getPackageManager ()
```

Ejemplo

```
PackageManager manager = context.getPackageManager();
```

El método `getPackageInfo` permite obtener toda la información contenida en el manifiesto. Este método recibe como parámetros el nombre de la aplicación identificada por su paquete y un flag opcional que aquí no se utiliza.

Devuelve un objeto de tipo `PackageInfo`. Puede elevar una excepción`NameNotFoundException` si el nombre del paquete proporcionado no se encuentra entre las aplicaciones instaladas.

Sintaxis

```
public abstract PackageInfo getPackageInfo (String packageName,
  int flags)
```

Ejemplo

```
PackageInfo info = null;
try {
  info = manager.getPackageInfo(context.getPackageName(), 0);
} catch (NameNotFoundException e1) {
}
```

Queda por recuperar la información del objeto de tipo `PackageInfo` accediendo directamente a sus variables miembros `packageName` y `versionName` proporcionando, respectivamente, el nombre del paquete y el número de versión de la aplicación.

Ejemplo

```
String nombrePaquete = info.packageName;
String nombreVersion = info.versionName;
```

La cadena de caracteres agente usuario puede, así, estar constituida por toda esta información.

Ejemplo

```
final String AGENTE_USUARIO = "%s/%s (Android/%s/%s/%s/%s)";
String agenteUsuario = String.format(AGENTE_USUARIO,
  info.nombrePaquete, nombreVersion, modelo, versionSistema,
  codigoSistema, paramsReg);
```

2. AndroidHttpClient

Para describir el envío de una consulta y la recepción de la respuesta del servidor, supondremos que el servidor remoto acepta una petición HTTP `GET` y envía una respuesta bajo la forma de una cadena de caracteres como, por ejemplo, una respuesta con formato JSON.

Android incluye la librería `org.json` que contiene lo necesario para leer respuestas con formato JSON. Para más información sobre el formato JSON: http://www.json.org/.

Desde Android 3.0 (API 11), la librería `org.json` proporciona las nuevas clases `JsonReader`y `JsonWriter` que permiten, respectivamente, leer y escribir de forma sencilla un flujo JSON.

La petición `GET` se crea fácilmente con la clase `HttpGet` y uno de sus constructores recibe como parámetro la dirección completa del sitio.

Sintaxis

```
public HttpGet (String uri)
```

Ejemplo

```
HttpGet httpGet = new HttpGet("http://...");
```

La clase `AndroidHttpClient` permite utilizar un cliente HTTP específicamente adaptado a Android. Para utilizarla no hay que instanciarla directamente, sino invocar a su método estático`newInstance` que devuelve una instancia. Este método recibe como parámetro una cadena de caracteres que describe el agente usuario.

Sintaxis

```
public static AndroidHttpClient newInstance (String userAgent)
```

Ejemplo

```
AndroidHttpClient httpClient =
  AndroidHttpClient.newInstance(agenteUsuario);
```

El envío de la consulta y la recepción de la respuesta se realizan invocando al método `execute` del cliente HTTP. Este método está disponible en distintas formas. Una de ellas, en concreto, recibe como parámetros la consulta y un objeto que implementa la interfaz `ResponseHandler`. Puede elevar excepciones de tipo `IOException` si ha tenido algún problema o si la conexión ha sido anulada y

excepciones de tipo `ClientProtocolException` su ha habido un error en el protocolo HTTP.

Es preciso, evidentemente, que el dispositivo Android esté conectado a la red mediante Wi-Fi o mediante la red de telefonía móvil, por ejemplo, o en caso contrario se elevará una excepción de tipo `java.net.UnknownHostException`, que hereda del tipo `IOException`.

La clase `BasicResponseHandler` implementa la interfaz `ResponseHandler` y, en concreto, su método `handleResponse`. Este método recibe como entrada la respuesta del servidor bajo la forma de un objeto de tipo `HttpResponse`, extrae el contenido de la respuesta y lo devuelve en forma de cadena de caracteres.

Se elevará una excepción de tipo `HttpResponseException` si el código de retorno del servidor es igual o superior a 300.

Sintaxis

```
public String handleResponse (HttpResponse response)
```

De este modo, el método `execute` devuelve la respuesta del servidor en forma de cadena de caracteres si recibe como parámetro un objeto de tipo `BasicResponseHandler`, y eleva las excepciones llegado el caso.

Sintaxis

```
public abstract T execute (HttpUriRequest request,
  ResponseHandler<? extends T> responseHandler)
```

Ejemplo

```
String reponse = null;
try {
  reponse = httpClient.execute(httpGet,
                               new BasicResponseHandler());
} catch (HttpResponseException e1) {
  int errno = e1.getStatusCode();
  procesarExcepcion(e1);
} catch (ClientProtocolException e2) {
  procesarExcepcion(e2);
} catch (IOException e3) {
  procesarExcepcion(e3);
}
```

Una vez finalizada la comunicación, el cliente debe cerrar las conexiones y liberar los recursos reservados invocando a su método `close`.

Sintaxis

```
public void close ()
```

Ejemplo

```
try {
  ...
} finally {
  httpClient.close();
}
```

La llamada a este método puede, en particular, realizarse en el bloque `finally` de la gestión de las excepciones. De este modo, siempre se invocará y los recursos siempre se liberarán, se haya elevado o no una excepción.

Introducción

La integración de las redes sociales es una funcionalidad indispensable en cualquier aplicación Android: permite a los usuarios compartir su experiencia con sus amigos y, para el desarrollador, aporta cierta publicidad suplementaria para la aplicación, lo cual nunca viene mal.

Veremos en este capítulo dos formas que permiten integrar el famoso botón Compartir (*share*, en inglés) en una aplicación Android.

El primer método puede calificarse como integración estándar: resulta muy rápida de implementar y provee lo esencial, aunque es muy dependiente del entorno en el que se ejecuta.

El segundo método, que propone una integración completa con las principales redes sociales, utiliza una librería externa - open source - que simplifica el trabajo al ser compatible con las especificidades de cada red social.

Integración estándar

Este primer método que se propone para integrar la forma de compartir contenido en las principales redes sociales se basa en una filosofía muy sencilla: el usuario dispone, en su dispositivo, de las aplicaciones de las principales redes sociales que utiliza. Por ello, es posible compartir información a través de estas aplicaciones, y no desarrollando una pieza de software específica, que no sería más que una versión limitada de lo que ya proporciona cada aplicación de red social.

La problemática es la siguiente: exponer la información a compartir, y dejar que el sistema y el usuario seleccionen la mejor solución para compartir el contenido.

1. Con Android 2.x y 3.x

Los objetos que permiten compartir información en un dispositivo Android ya son conocidos y se han estudiado en el capítulo Los fundamentos, sección Intención: se trata de las intenciones.

Recordemos que el funcionamiento de las intenciones es el siguiente: hay que crear una intención especificando la acción que se desea ejecutar y, eventualmente, agregar datos adicionales.

Aquí, la acción que nos interesa es la acción `ACTION_SEND`: indica que la aplicación desea enviar información. El destino del envío no se especifica, éste es el principio de funcionamiento de las intenciones, sino que es el usuario quien, en tiempo de ejecución, entre las opciones capaces de realizar la solicitud selecciona el modo de compartir el contenido.

El contenido que se quiere compartir en el marco de la acción seleccionada debe ser bien un texto o bien un archivo enviado en formato binario.

En el primer caso (texto), el contenido se incluirá en el campo extra `Intent.EXTRA_TEXT`, y el tipo MIME será `text/plain`.

En el segundo caso (archivo binario), se utilizará el campo extra `Intent.EXTRA_STREAM`, y el tipo MIME dependerá del tipo de archivo binario (`image/jpeg` en el caso de una imagen en formato JPEG, por ejemplo, o de forma más genérica `image/*`).

Es posible informar campos extra suplementarios, aunque sin estar seguros de si los tendrá en cuenta cada aplicación: `EXTRA_EMAIL`, `EXTRA_CC`, `EXTRA_BCC`, `EXTRA_HTML_TEXT` y`EXTRA_SUBJECT`.

Una vez creada la intención y asignados los datos "extra", únicamente queda iniciar la actividad que incluirá el objeto `Intent`.

En el marco de una función diseñada para compartir contenido, la mejor solución consiste en encapsular la llamada en un objeto de tipo `Chooser` (seleccionador), que presenta una interfaz específica para la selección de la aplicación. El objeto `Chooser` no se utiliza directamente sino a través del método `Intent.createChooser`.

Sintaxis

```
Static Intent.createChooser(Intent destino, CharSequence titulo)
```

Ejemplo

```
Intent shareIntent=
   new Intent(android.content.Intent.ACTION_SEND);

shareIntent.setType("text/plain");

shareIntent.putExtra(android.content.Intent.EXTRA_SUBJECT,
      "Ejemplo compartir contenido");

shareIntent.putExtra(android.content.Intent.EXTRA_TEXT,
```

```
      "Texto del mensaje compartido");

startActivity(
   Intent.createChooser(shareIntent, "Compartir en..."));
```

2. Con Android 4.x

Con la versión 14 (Android 4.0.1, Ice Cream Sandwich), la forma de compartir contenido en redes sociales está, por lo general, integrada en la barra de acción.

Para integrar en esta barra de acción el icono específico para compartir contenido y las acciones asociadas basta con agregar un ítem en el archivo XML del menú, configurándole un atributo de tipo`actionProviderClass` específico.

El atributo `ActionProviderClass` se incluye con la API 14 y permite asociar una clase que se encargará de gestionar la acción cuando se seleccione el elemento de menú.

La clase que se especifique en el atributo de tipo `ActionProviderClass` debe heredar de la clase abstracta `ActionProvider`.

El sistema Android provee, desde la API 14, un `ActionProvider` específico para compartir contenido en redes sociales: la clase `ShareActionProvider` (del paquete `android.widget`). Es esta clase la encargada de crear los submenús que presentan una lista de redes sociales y los métodos subyacentes.

Es preciso, por tanto, indicar estas clases como valor para el atributo `actionProviderClass` del ítem de menú.

A continuación se muestra un ejemplo de declaración de menú para mostrar un icono "compartir" en la barra de acción. Este archivo de menú debe almacenarse en la carpeta `/res/menu` del proyecto.

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item
     android:id="@+id/menu_item_compartir"
     android:showAsAction="ifRoom"
     android:title="Compartir"
  android:actionProviderClass="android.widget.ShareActionProvider" />
</menu>
```

Para vincular el menú con su actividad es preciso, como se ha visto en el capítulo Completar la interfaz de usuario, sección Menús, sobrecargar el método `onCreateOptionMenu` de la clase`Activity`.

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.menu_compartir, menu);
   [...]
}
```

A continuación, en el método `onCreateOptionsMenu`, hay que recuperar el `ActionProvider`del item de menú declarado en el archivo XML de menú y vincularle un objeto de tipo`ShareActionProvider`. Para ello, la clase `MenuItem` dispone del método`getActionProvider`, que devuelve el `ActionProvider` especificado en el atributo.

```
ShareActionProvider shareActionProvider =
   (ShareActionProvider) item.getActionProvider();
```

Sólo queda por definir, tal y como hemos visto anteriormente, una intención que asociaremos al objeto de tipo `ShareActionProvider`, invocando al método `setShareIntent` de la clase`ShareActionProvider`.

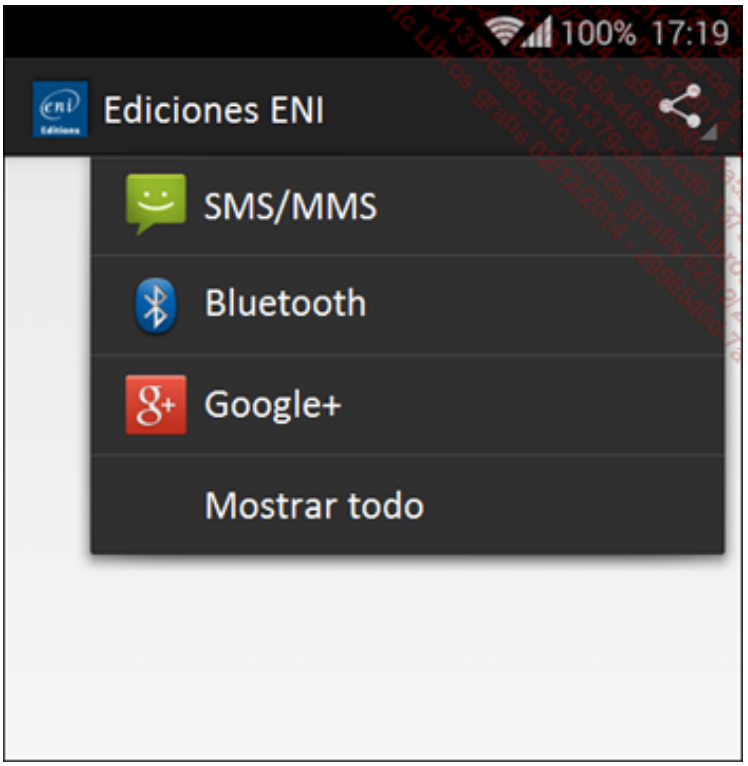
Sintaxis

```
void setShareIntent (Intent shareIntent)
```

Ejemplo

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
   getMenuInflater().inflate(R.menu.menu_cap10, menu);
   MenuItem item = menu.findItem(R.id.menu_item_share);

   ShareActionProvider shareActionProvider =
       (ShareActionProvider) item.getActionProvider();

   shareActionProvider.setShareIntent(createShareIntent());

    return true;
}

private Intent createShareIntent() {
   Intent shareIntent=
      new Intent(android.content.Intent.ACTION_SEND);

   shareIntent.setType("text/plain");
   shareIntent.putExtra(android.content.Intent.EXTRA_SUBJECT,
      "Ejemplo compartir contenido");
   shareIntent.putExtra(android.content.Intent.EXTRA_TEXT,
      "Texto del mensaje compartido");
   return shareIntent;
}
```

Cuando se ejecute la actividad, el botón para compartir se agregará automáticamente en la zona superior derecha de la barra de acción. Haciendo clic en el botón, se abrirá el menú "Compartir".



En resumen, este método de compartir contenido presenta numerosas ventajas:

- Resulta muy sencillo de implementar.
- No requiere ni claves ni API ni procesos de autenticación.

- Utiliza únicamente elementos propios de la plataforma.
- No existe el riesgo de depender de una API externa (del estilo de las API que proveen las redes sociales), lo cual limita y reduce el mantenimiento.
- Todas las redes sociales instaladas en el dispositivo del usuario, y únicamente estas redes, se mostrarán y podrán seleccionarse.

No obstante, existen ciertas limitaciones que pueden resultar algo frustrantes: por un lado, el mensaje que se envía es bastante sencillo y no tiene ningún formato, por otro lado, puede que las aplicaciones de redes sociales no sean compatibles con la intención.

Integración completa

La mayoría de redes sociales, si no todas, proporcionan una API para Android que permite compartir contenido. No obstante, sería complicado integrar, separadamente, cada una de estas API y asegurar su mantenimiento para cada evolución de la API. Una solución alternativa, que se estudia en esta sección, consiste en recurrir a una API externa que es compatible con la mayoría de redes sociales.

Existen varias API de este tipo en el mercado; algunas son de pago pero la mayoría son gratuitas. La elección de la API debe realizarse en base a ciertos criterios a los que cabe prestar atención:

- La API debe presentar un buen rendimiento, y ofrecer todas las funcionalidades para compartir contenido propias de cada red social.
- La API debe actualizarse con regularidad por su(s) autor(es), de cara a evolucionar de la mano de cada red social.
- Por último, el trabajo de integración debe ser lo más ligero posible.

Nuestra elección, en base a estos criterios, es la API `socialauth-android`, versión para Android de una API Java bastante popular. Esta sección presenta todas las etapas necesarias para integrar la API así como la integración que hay que realizar desde cada red social afectada.

1. Obtener las claves de API

Incluso aunque cada red social funcione a su propia manera, existen algunos esquemas de uso que son comunes a todas; es el caso, por ejemplo, de la conexión con aplicaciones de terceros. Todas las redes sociales exigen, para autorizar la publicación de datos en sus sistemas, que la aplicación que envía la solicitud haya sido referenciada previamente en la propia red social.

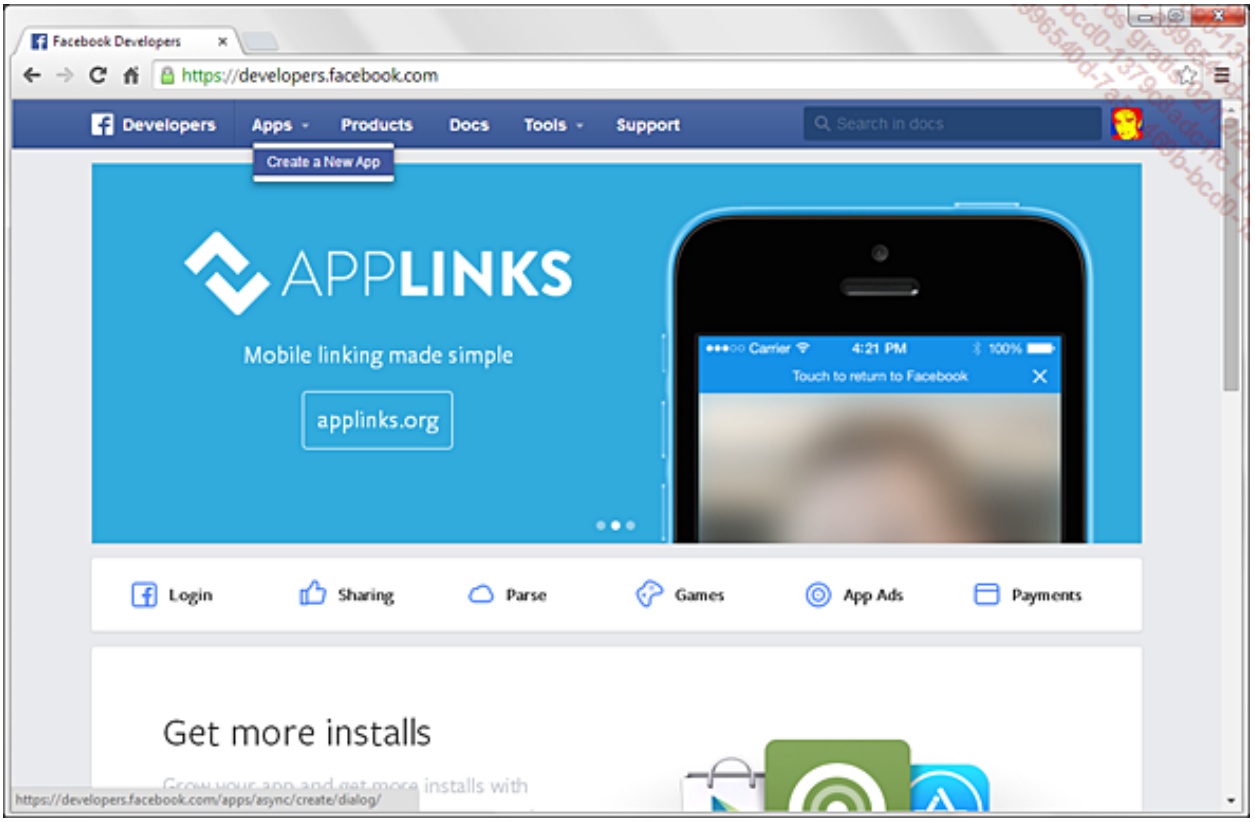
La primera etapa consiste, por tanto, en referenciar nuestra aplicación desde cada una de las redes sociales. Esto nos devolverá un par (clave API / clave secreta) que permite identificar a la aplicación que envía el contenido.

Veremos cómo realizarlo para Facebook, sabiendo que el funcionamiento general es idéntico para el resto de medios. Observe que conviene poseer una cuenta de usuario en cada una de estas redes sociales con la que desee integrarse.

a. Crear una aplicación Facebook

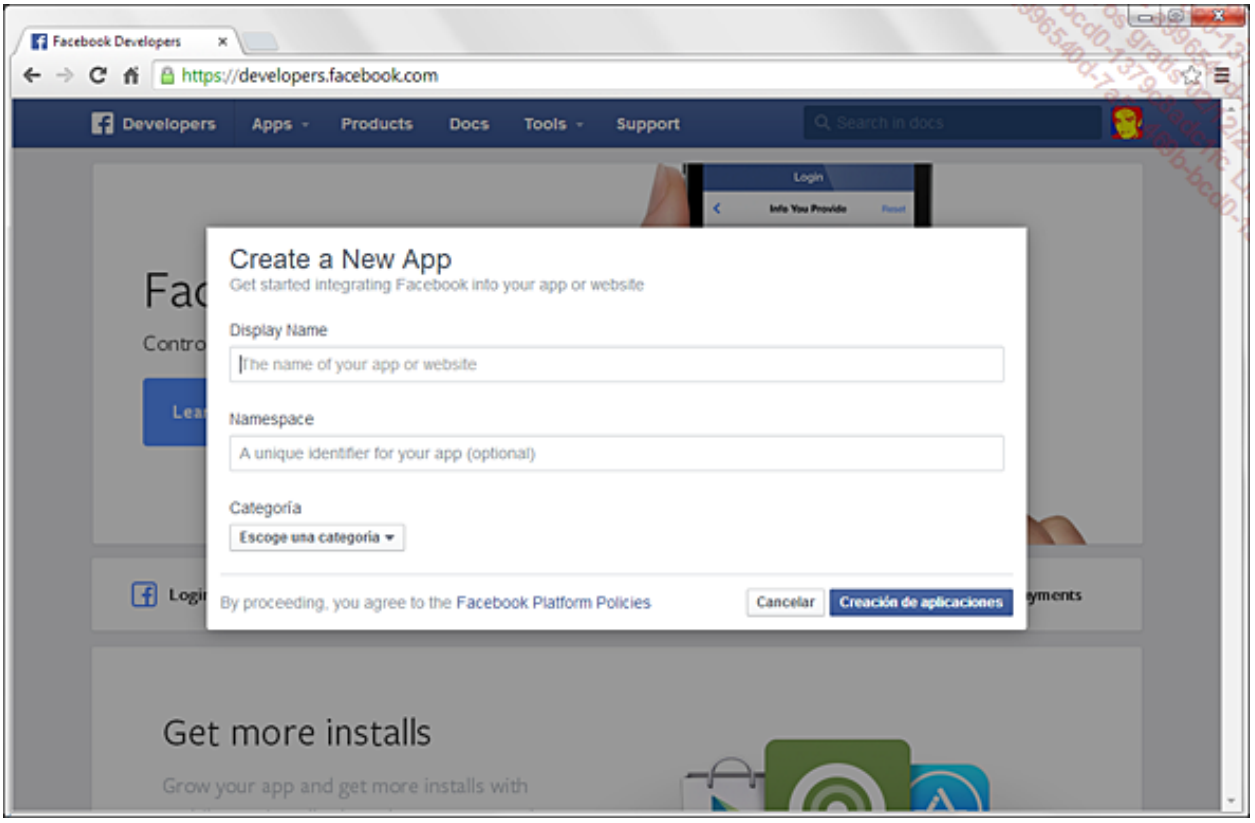
Para poder crear una aplicación Facebook, hay que conectarse con el sitio https://developers.facebook.com, e identificarse con una cuenta de usuario de Facebook.

- En la barra de menú, seleccione **Apps** - **Create a New App**.



Se abre un formulario en una ventana emergente, que le permite introducir un nombre para su aplicación e informar un nombre de `namespace` (que puede dejarse vacío, pues aplica únicamente a aplicaciones web).

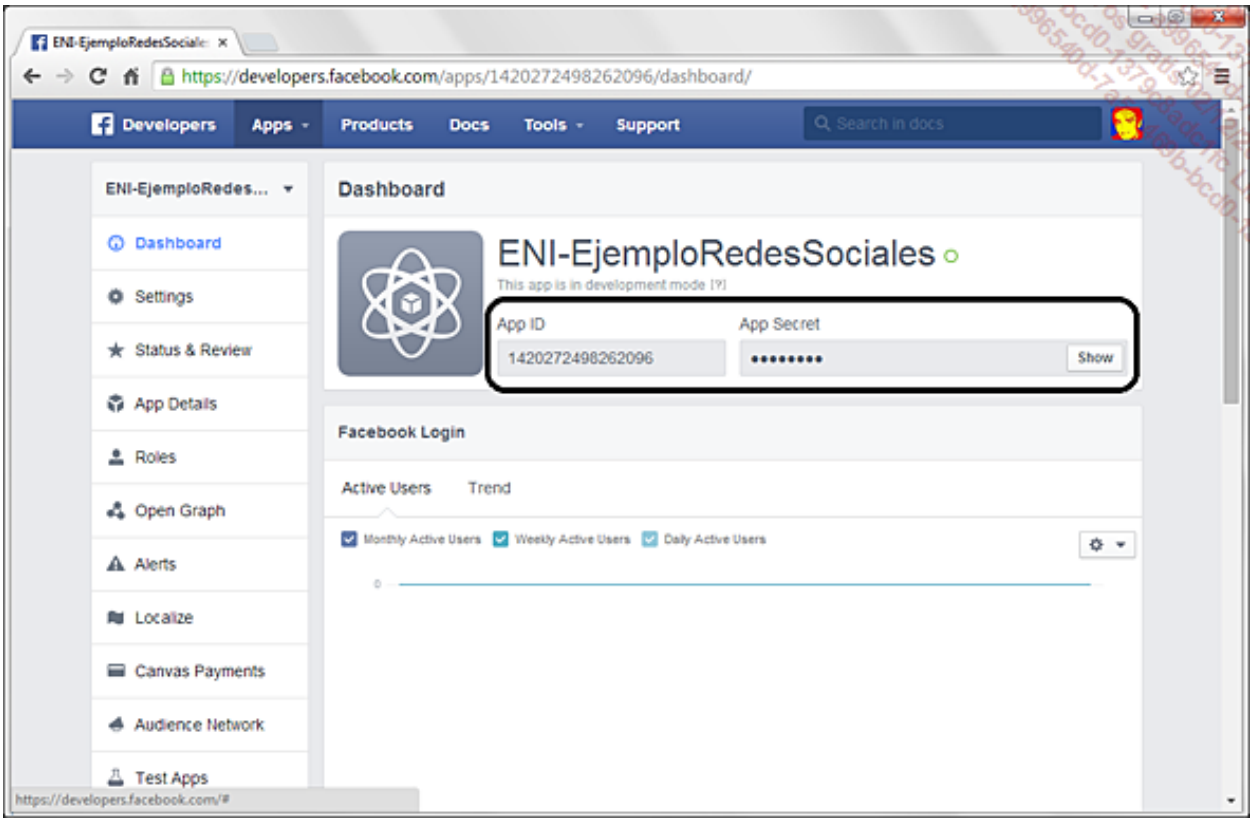
- Una vez relleno el formulario, haga clic en el botón **Creación de aplicaciones**.



Tras un breve instante (no tiene por qué ser inmediato), la página web se actualiza y le muestra el cuadro de mando de su aplicación.

La información que nos interesa, de momento, se ubica en la zona superior de la página:

identificador de la aplicación (se trata de la clave API) así como la clave secreta (oculta por defecto, es preciso hacer clic en el botón **Show** para verla). Esta información debe conservarse, pues se incluirá en un archivo de configuración posteriormente.



La creación de la aplicación para Facebook ha terminado: puede explorar la configuración propuesta por Facebook para personalizar la visualización de los mensajes en el foro de actualidad de los usuarios.

Como con la mayoría de redes sociales, Facebook no proporciona un modo de "desarrollo" o "depuración" para las aplicaciones: todos los mensajes que escriba se integrarán directamente en el hilo de actualidad del usuario que utilice para las pruebas. Estos mensajes pueden, no obstante, borrarse.

b. Otras redes sociales

Dado que el mecanismo de creación de cuentas de desarrollador es propio de cada sitio (y, en algunos casos, con continuas modificaciones), no detallaremos aquí el procedimiento para cada una de estas redes sociales. La siguiente tabla muestra las direcciones donde puede iniciarse el proceso de creación de una cuenta de desarrollador para las principales redes sociales.

Facebook	https://developers.facebook.com/
Foursquare	https://developer.foursquare.com/
Google+	https://developers.google.com/+/api/oauth#apikey
LinkedIn	http://developer.linkedin.com/
Twitter	https://dev.twitter.com/

2. Instalar el SDK SocialAuth-Android

Hemos escogido trabajar con la API `SocialAuth-Android`, proyecto open-source iniciado por la empresa 3Pillar Labs (http://labs.3pillarglobal.com), que permite administrar varias redes sociales de manera global.

La primera etapa consiste en descargar el SDK (*Software Development Kit*) de SocialAuth-Android, de la dirección http://sourceforge.net/projects/socialauth-android/ y, a continuación, descomprimir el archivo zip obtenido. Se crean las carpetas `assets`, `dist`, `examples`, `javadoc`, `libs` y `src`.

a. Integración de las librerías con el proyecto

Para utilizar las librerías de una API en una aplicación, es preciso agregar los archivos de la librería en la carpeta `/libs` del proyecto en Eclipse.

Aquí, necesitamos dos librerías: `socialauth` y `socialauth-android`. Las librerías se proveen en formato java .jar. El archivo `socialauth-4.4.jar` se encuentra en la carpeta`/libs` del archivo descomprimido, el archivo `socialauth-android-3.1.jar` se encuentra en la carpeta `/dist` (los números de versión se indican a título informativo, son susceptibles de cambiar regularmente).

Cuando se agregan ambos archivos a la carpeta `/libs` del proyecto Eclipse, el plug-in ADT los agrega automáticamente en el java `build-path` (ruta de construcción) del proyecto: el build-path hace referencia a todos los archivos necesarios para realizar la compilación del proyecto.

A continuación hay que agregar, en la carpeta `/assets` del proyecto Eclipse, el archivo`oauth_consumer.properties`, que se encuentra en la carpeta `/assets` del archivo descomprimido. Este archivo contiene las claves de la API y las claves secretas para todas las redes sociales soportadas por la API.

Modificando este archivo con el editor Eclipse (o con cualquier otro editor de texto), podremos agregar las claves de las redes sociales que queramos integrar en nuestra aplicación.

Para finalizar la integración de las librerías SocialAuth-Android, queda por agregar las autorizaciones `INTERNET` y `ACCESS_NETWORK_STATE` en el manifiesto del proyecto: la liberaría necesita, en efecto, acceso a Internet en el dispositivo del usuario.

b. Uso de la API

Una vez realizada la instalación y configuración de la API, puede utilizarse directamente en el proyecto.

La API expone diversas opciones para integrar la manera de compartir contenido en las redes sociales: mediante un botón "compartir", mediante el menú de la barra de acción, etc. Vamos a estudiar cómo crear un botón "compartir" en una aplicación.

El procedimiento para compartir contenido se desarrolla en varias etapas:

- Petición de compartir contenido por parte del usuario.
- Selección de la red social que se quiere utilizar para compartir el contenido.
- Conexión del usuario a la red social seleccionada.
- Publicación del contenido a compartir.
- Confirmación del contenido compartido.

La API `SocialAuth-Android` se encarga de gestionar todas las etapas.

La clase `SocialAuthAdapter` provee todos los métodos necesarios para compartir el contenido. El constructor de la clase recibe como parámetro un objeto de tipo `DialogListener`, definido por la API, que se encarga de gestionar el proceso de `callback` (llamada de retorno) que sigue al procesamiento de un cuadro de diálogo.

La interfaz `DialogListener` obliga a implementar los métodos siguientes:

- `onBack`: se invoca cuando el usuario cierra el cuadro de diálogo haciendo clic en el botón volver de su dispositivo.

- onCancel: se invoca cuando el usuario anula la acción.
- onComplete: se invoca cuando el usuario finaliza la acción del cuadro de diálogo.
- onError: se invoca cuando se produce un error desde el cuadro de diálogo.

Ejemplo

```
SocialAuthAdapter adapter =
   new SocialAuthAdapter(new DialogListener() {

   @Override
   public void onError(SocialAuthError arg0) {

   }

   @Override
   public void onComplete(Bundle arg0) {

   }

   @Override
   public void onCancel() {

   }

   @Override
   public void onBack() {

   }
});
```

El objeto de tipo DialogListener que se pasa como parámetro al constructor de la claseSocialAuthListener tiene la función de gestionar el retorno tras la conexión del usuario. En este momento, debe publicarse el mensaje para la red social, como respuesta a la llamada al método onComplete.

La instancia de tipo SocialAuthAdapter permite especificar cuáles son las redes sociales que se mostrarán en el cuadro de diálogo cuando el usuario haga clic en el botón "Compartir".

Para agregar una red social, se utiliza el método addProvider de la claseSocialAuthAdapter.

Sintaxis

```
void addProvider(SocialAuthAdapter.Provider provider, int logo)
```

El parámetro provider permite indicar la red social. SocialAuthAdapter.Provider es una enumeración que contiene la lista de redes sociales soportadas por la API.

El parámetro logo permite indicar el identificador del recurso que se utilizará como logotipo para la red social de la lista.

Ejemplo

```
adapter.addProvider(SocialAuthAdapter.Provider.FACEBOOK,
R.drawable.facebook);

adapter.addProvider(SocialAuthAdapter.Provider.TWITTER,
R.drawable.twitter);
```

Para declarar la acción de compartir contenido cuando el usuario haga clic en el botón compartir de una actividad, hay que declararlo en el objeto SocialAuthAdapter, utilizando el métodoenable.

Sintaxis

```
void enable(android.widget.Button botonCompartir)
```

Ejemplo

```
Button btnCompartir = (Button)findViewById(R.id.boton_compartir);
adapter.enable(btnCompartir);
```

Observe que, a diferencia de lo que ocurre en un botón clásico, no es necesario invocar al método`setOnClickListener` en el botón que sirve para compartir contenido: el método `enable` se encarga de configurar el botón.

Cuando el usuario selecciona una red social y se conecta, se invoca al método `onComplete` de la instancia de `DialogListener` que se pasa como parámetro al constructor`SocialAuthAdapter`: el mensaje puede publicarse.

Para publicar un mensaje, debemos utilizar uno de los siguientes métodos: `updateStatus`,`updateStory` (sólo en Facebook) o `uploadImageAsync` (Facebook y Twitter). La lista de todas las acciones soportadas por la API se describe más adelante en esta sección.

Sintaxis

```
void updateStatus(java.lang.String message,
   SocialAuthListener<java.lang.Integer> listener,
   boolean shareOption)

void updateStory(java.lang.String message,
   java.lang.String name,
   java.lang.String caption,
   java.lang.String description,
   java.lang.String link,
   java.lang.String picture, SocialAuthListener<java.lang.Integer>
listener)


void uploadImageAsync(java.lang.String message,
   java.lang.String fileName,
   android.graphics.Bitmap bitmap,
   int quality,
   SocialAuthListener<java.lang.Integer> listener)
```

Ejemplo

```
private class ResponseListener implements DialogListener {

   @Override
   public void onBack() {
   }

   @Override
   public void onCancel() {
   }

   @Override
   public void onComplete(Bundle arg0) {
      adapter.updateStatus("Mensaje que se mostrará en el flujo
de actualidad",new MessageListener(),false );
   }

   @Override
   public void onError(SocialAuthError arg0) {
```

```
    }

}
```

Los métodos `updateStatus`, `updateStory`, etc., requieren un objeto de tipo`SocialAuthListener<java.lang.Integer>` como parámetro: este objeto permite gestionar el retorno tras la publicación. El valor entero que se pasa como parámetro contiene el código devuelto por la publicación, se trata del código de respuesta http/1.1 devuelto por la red social.

La interfaz `SocialAuthListener<java.lang.Integer>` requiere implementar los métodos`onError` y `onExecute` (el primero se invoca en caso de error, mientras que el segundo se invoca cuando el servidor web de la red social devuelve algún mensaje).

Sintaxis

```
void onError(SocialAuthError e)
void onExecute(java.lang.String provider, T t)
```

Ejemplo

```
class MessageListener implements SocialAuthListener<Integer> {
   @Override
   public void onExecute(String arg0,Integer t) {
      Integer status = t;
      if (status.intValue() == 200 ||
         status.intValue() == 201 ||
         status.intValue() == 204)
         Toast.makeText(getContext(),
         "Mensaje publicado",Toast.LENGTH_LONG).show();
      else
         Toast.makeText(getContext(),
          "Mensaje no publicado",Toast.LENGTH_LONG).show();
      finishActivity();
   }

   public void onError(SocialAuthError e) {

   }
}
```

Puede ser necesario diferenciar la acción que queremos realizar en función de la red social seleccionada por el usuario: la API gestiona las especificidades de cada red social, estando algunas funcionalidades reservadas a alguna red en particular: es el caso, por ejemplo, del método `updateStory` (que sólo está disponible para Facebook).

Para hacer esta distinción, el objeto `SocialAuthAdapter` dispone del método`getCurrentProvider`, que devuelve un objeto de tipo`org.brickred.socialauth.AuthProvider`, que se corresponde con la red social seleccionada por el usuario.

Para identificar una red fácilmente, el método más sencillo consiste en invocar al método`getProviderId`, que devuelve el nombre de la red social en forma de cadena de caracteres.

Ejemplo

```
if (adapter.getCurrentProvider().getProviderId().equalsIgnore
Case("facebook")) {
// Procesamiento específico para Facebook
}
```

```
else {
// Procesamiento para las demás redes sociales
}
```

La siguiente tabla muestra la lista de métodos soportados por la API y sus eventuales restricciones.

Métodos	Función	Red social soportada
getAlbums	Obtener las fotos del usuario.	Facebook, Twitter
getCareerAsync	Obtener información acerca del puesto, los estudios, etc.	Linkedin
getContactList	Obtener una lista de los contactos del usuario.	Todas
getFeeds	Obtener el flujo de información.	Facebook, Twitter, Linkedin
getUserProfile	Obtener el perfil del usuario.	Todas
updateStatus	Actualizar el estado.	Facebook, Twitter, Linkedin, MySpace, etc.
updateStory	Compartir un mensaje y un vínculo con visualización previa del contenido.	Facebook
uploadImage	Subir una imagen.	Facebook, Twitter

El listado anterior es susceptible de cambiar, en función de la evolución de la API. La lista actualizada se encuentra en la siguiente dirección (en inglés):https://code.google.com/p/socialauth-android/, sección Existing Features.

Introducción

Incluso en las aplicaciones más sencillas, es raro poder escribir el código sin incurrir en algún error técnico o funcional. También puede haberse omitido el procesamiento de algún caso funcional particular.

Además, conforme más compleja es la aplicación, más difícil se hace modificarla sin correr el riesgo de agregar bugs o regresiones funcionales.

Para evitar esto, cada cual tiene su técnica. Hay quien sigue ciclos de desarrollo en V o en Y, otros adoptan metodologías de desarrollo más o menos ágiles como XP (*eXtreme Programming*) o TDD (*Test Driven Development*), y a menudo una mezcla de todas ellas...

No obstante, cada uno define su estrategia particular en cuanto a la realización de pruebas unitarias y/o de integración y/o de validación y/o de conformidad funcional. En ocasiones, la estrategia es simple: no se realizan pruebas, si bien esto no es lo más adecuado.

Sea cual sea la necesidad del desarrollador, Android proporciona distintas soluciones para ayudar a mejorar la calidad de su código y eliminar la mayor cantidad de errores.

Comenzaremos estudiando el sistema de registro de eventos. A continuación, veremos la depuración paso a paso de una aplicación para concentrarnos en la perspectiva Android de depuración y de trazas llamada DDMS (*Dalvik Debug Monitor Server*). Por último, veremos las pruebas unitarias y las pruebas de integración.

Registro de eventos

Un registro de eventos contiene los logs (trazas de ejecución de una aplicación). En Android, existen varios registros de eventos: uno principal y varios secundarios específicos a un dominio, por ejemplo los mensajes asociados a la telefonía. Estos registros se almacenan en forma de logs circulares.

Cada uno de estos eventos está constituido por los siguientes elementos:

- Fecha y hora de la aparición del evento.
- Categoría del evento.
- Identificador único del proceso, llamado pid (*Process IDentifier*), que ha generado el evento.
- Una etiqueta, llamada tag, que permite designar el origen del evento: nombre del proceso, del componente, de la actividad, de la clase…
- Un mensaje detallando el evento.

Existen varias categorías que permiten clasificar los eventos. La siguiente tabla muestra las cinco clases principales, clasificadas de la más detallada a la menos detallada:

Categoría	Nivel	Descripción
Verboso	V	Categoría más detallada. Si es muy grande la cantidad de información a mostrar, esta categoría puede ralentizar sensiblemente el funcionamiento de la aplicación. Por ello, se recomienda no utilizarla salvo en tiempo de desarrollo.
Depuración	D	Normalmente, los elementos de esta categoría se muestran si la aplicación se ha compilado en modo debug. En modo release, estos elementos se ignoran y, por tanto, no se muestran.
Información	I	Categoría con carácter informativo. Es conveniente usarla con tranquilidad dado que siempre se muestra.
Advertencia	W	Categoría que agrupa las alertas, eventos a los que el lector debe prestar una atención especial y tratar de resolver antes de que se genere un error.
Error	E	Nivel más elevado de los cinco presentes. Caracteriza generalmente un evento que ha generado un error grave y puesto en peligro el funcionamiento de la aplicación.

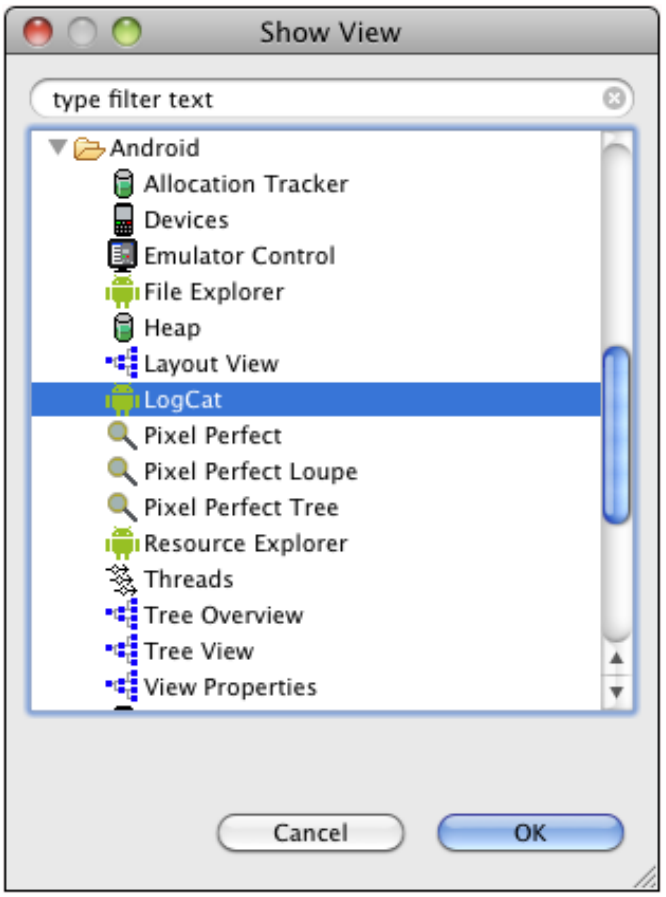
1. Consultar los eventos

En Eclipse, el desarrollador puede consultar el histórico de eventos del registro por defecto así generado mostrando la vista **LogCat**. Ésta ya viene activa en la perspectiva **DDMS** descrita más adelante. Para activarla en la vista **Java**, he aquí el procedimiento a seguir:

- En el menú general de Eclipse, seleccione **Window - Show View - Other**.

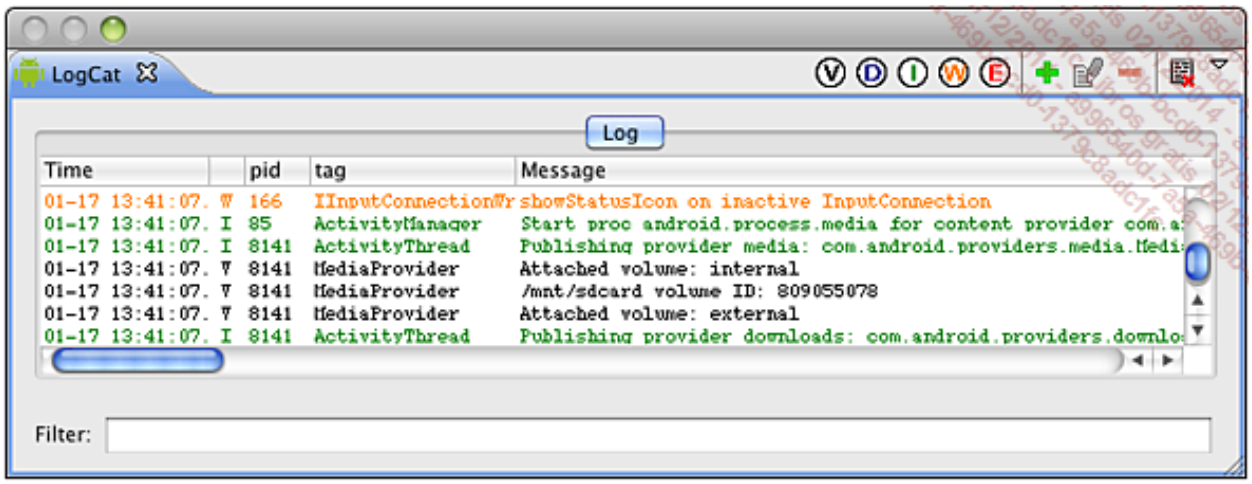
Se muestra la ventana **Show View**.

- Abra la carpeta **Android**.



- Seleccione **LogCat** y haga clic en el botón **OK**.

La vista **LogCat** se agrega a las vistas de Eclipse.



En la parte superior derecha de la ventana figuran los filtros preconfigurados que permiten mostrar únicamente los mensajes de nivel igual o superior al filtro. El uso del botón **+** permite crear nuevos filtros.

También es posible obtener los logs directamente en la consola utilizando la herramienta `adb`seguida del comando `logcat`. Es posible usar numerosas opciones, por ejemplo, indicar otro formato de visualización de los eventos o incluso filtrar los eventos que se quieren mostrar según su etiqueta y su categoría.

Sintaxis

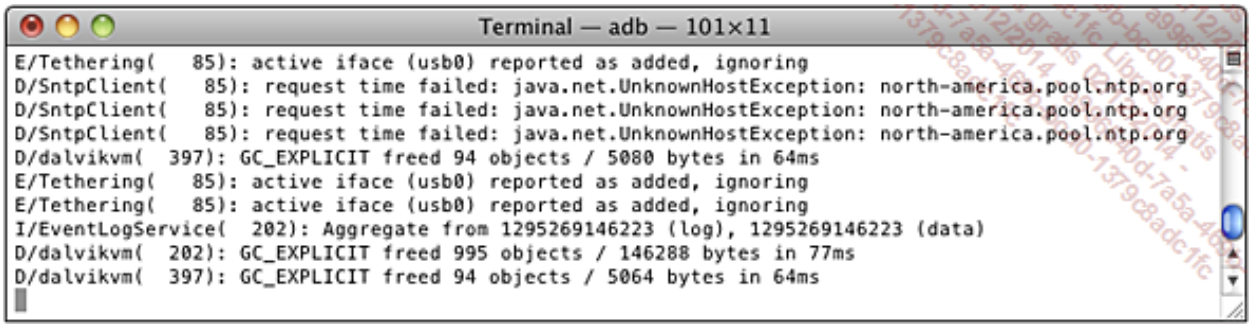
```
adb logcat [opción] ... [filtro] ...
```

Para descubrir todas las opciones y formatos de los filtros disponibles, especifique la opción –help.

Ejemplo

```
$ adb logcat
```

Este comando produce la siguiente visualización:



Por defecto, la visualización contiene la misma información que la vista **LogCat** en Eclipse pero con un formato ligeramente distinto.

En Eclipse, en ocasiones ocurre que la vista **LogCat** no muestra los logs. Es necesario reiniciar Eclipse. Para evitar esto, se puede utilizar la herramienta `adb` como se ha visto anteriormente para obtener la visualización de los logs en una consola.

2. Escribir eventos

Los eventos registrados en el registro son los del sistema, pero también los de las aplicaciones. En efecto, cualquier aplicación puede generar tales eventos.

Para ello, el SDK provee la clase `Log` del paquete `android.util`. Esta clase contiene como mínimo dos métodos para cada una de las categorías de eventos. Estos métodos reciben como parámetro una etiqueta y el mensaje. Uno de los métodos recibe un tercer parámetro de entrada que es un objeto de tipo `Throwable` que contiene un complemento de la información, principalmente el mensaje de la pila de llamadas a los métodos previos a que se provocase el evento.

La categoría Advertencia proporciona un tercer método que recibe como parámetros una etiqueta y un objeto de tipo `Throwable`.

Sintaxis

```
public static int v (String tag, String msg)
public static int v (String tag, String msg, Throwable tr)
public static int d (String tag, String msg)
public static int d (String tag, String msg, Throwable tr)
public static int i (String tag, String msg)
public static int i (String tag, String msg, Throwable tr)
public static int w (String tag, String msg)
public static int w (String tag, Throwable tr)
public static int w (String tag, String msg, Throwable tr)
public static int e (String tag, String msg)
public static int e (String tag, String msg, Throwable tr)
```

Es habitual definir una cadena de caracteres constantes propia a cada clase de tipo actividad, servicio... para usarla como etiqueta en el resto de la clase. Esto permite filtrar y encontrar más

rápidamente los mensajes en el histórico. De manera recíproca, esto permite encontrar fácilmente el origen de los mensajes.

Ejemplo

```
public class MiActividadPrincipal extends Activity {
  private static final String TAG = "MiActividadPrincipal";

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    Log.d(TAG, "Desde el método onCreate");
  }
}
```

Depuración

La depuración es la fase consistente en buscar y corregir los bugs. Para ayudar al desarrollador en esta tarea, Eclipse, el plug-in ADT y el SDK Android ponen a disposición del desarrollador un conjunto de herramientas que vamos a descubrir en las siguientes secciones.

1. Depuración paso a paso

La depuración paso a paso no es específica de Android. Este modo está disponible en todos los programas Java en Eclipse. Utiliza una herramienta llamada depurador que ejecuta el programa en un modo concreto. El desarrollador tiene, entonces, acceso a numerosas funcionalidades que le permiten establecer puntos de ruptura en el código fuente, analizar paso a paso el desarrollo del programa y conocer, entre otros, la pila de llamadas de los métodos y los valores de las variables.

Supondremos que se maneja bien el uso de este modo de depuración en Eclipse dado que no es algo específico de la plataforma Android sino del desarrollo en Java.

Hasta la versión 9 de las Android SDK tools (herramientas del SDK Android), para permitir este tipo de depuración, una aplicación Android necesitaba poner obligatoriamente a `true` el atributo`android:debuggable` de la etiqueta `application` del manifiesto, que por defecto valía`false`. Sin ello, la ejecución en modo de depuración no tenía efecto alguno.

Ejemplo

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
    xmlns:android="http://schemas.android.com/apk/res/android"
    package="es.midominio.android.miaplicacion"
    android:versionCode="1"
    android:versionName="1.0">
  <application android:debuggable="true"
               android:icon="@drawable/icon"
               android:label="@string/app_name">
  </application>
  <uses-sdk android:minSdkVersion="8" />
</manifest>
```

Desde la versión 9 de las herramientas del SDK, el desarrollador ya no necesita agregar este atributo para posicionarlo a `true`. En Eclipse y el plug-in ADT, esto se realiza automáticamente mediante compilaciones incrementales que se consideran, por defecto, compilaciones de debug. No es sino una exportación de una compilación firmada en modo release que no tendrá este atributo.

Esta funcionalidad está, de hecho, incluida en la versión 8, pero un bug impide que funcione correctamente. Este bug ha sido corregido en la versión 9.

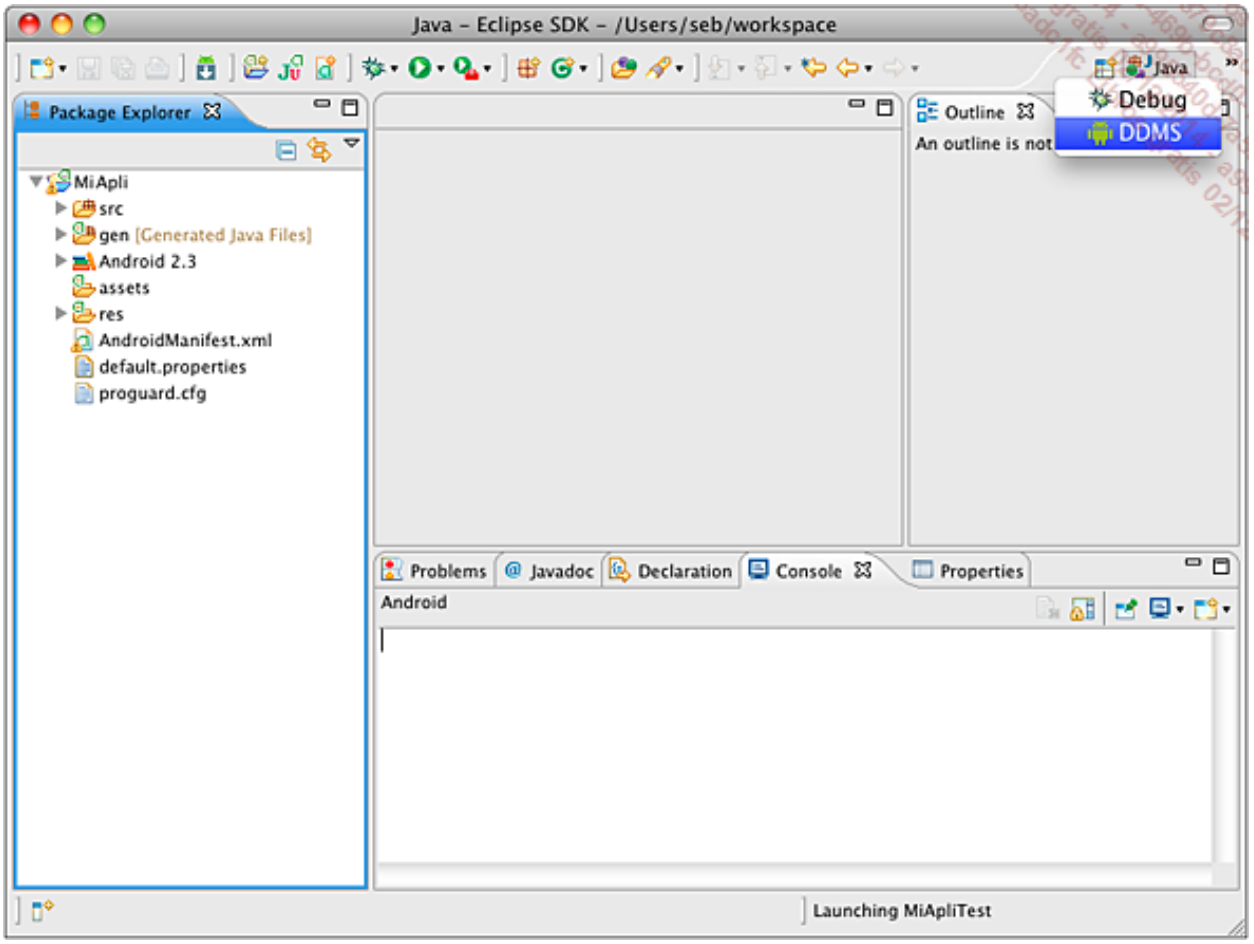
Se recuerda que el desarrollador siempre debe utilizar la versión más reciente de las herramientas del SDK.

2. DDMS

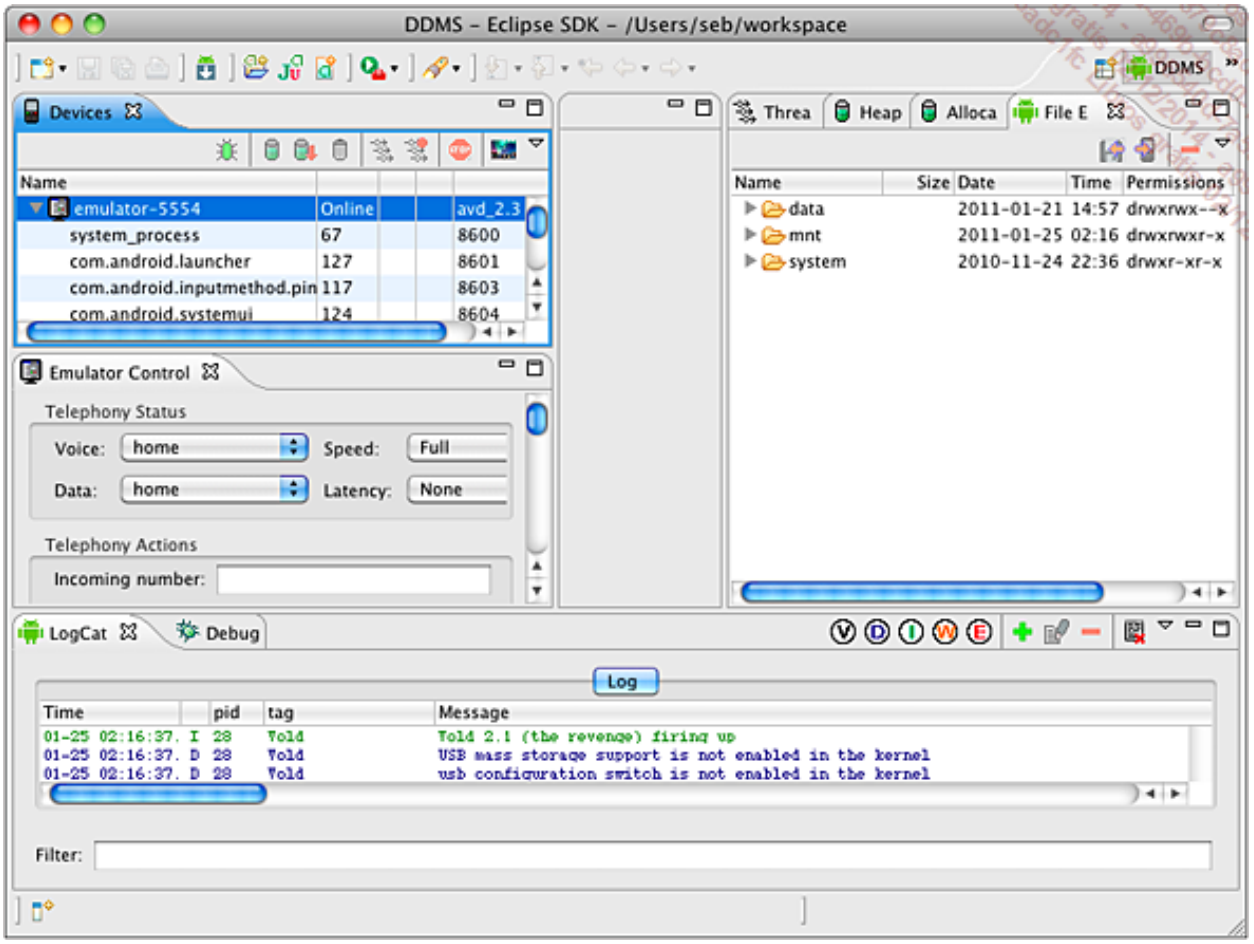
El plug-in ADT proporciona una perspectiva Eclipse dedicada especialmente a la depuración de la aplicación Android: la perspectiva DDMS (*Dalvik Debug Monitor Server*). Ésta utiliza de forma interna la herramienta del mismo nombre: `ddms`. Esta herramienta ofrece bastantes funcionalidades. Vamos a ver las principales, que se acceden desde la perspectiva Eclipse.

Observe que si se ejecuta desde una línea de comandos, esta herramienta ofrece todavía más funcionalidades avanzadas.

- Si la perspectiva **DDMS** no está visible en la parte superior de la ventana general de Eclipse, haga clic en el botón **»** para mostrarla.

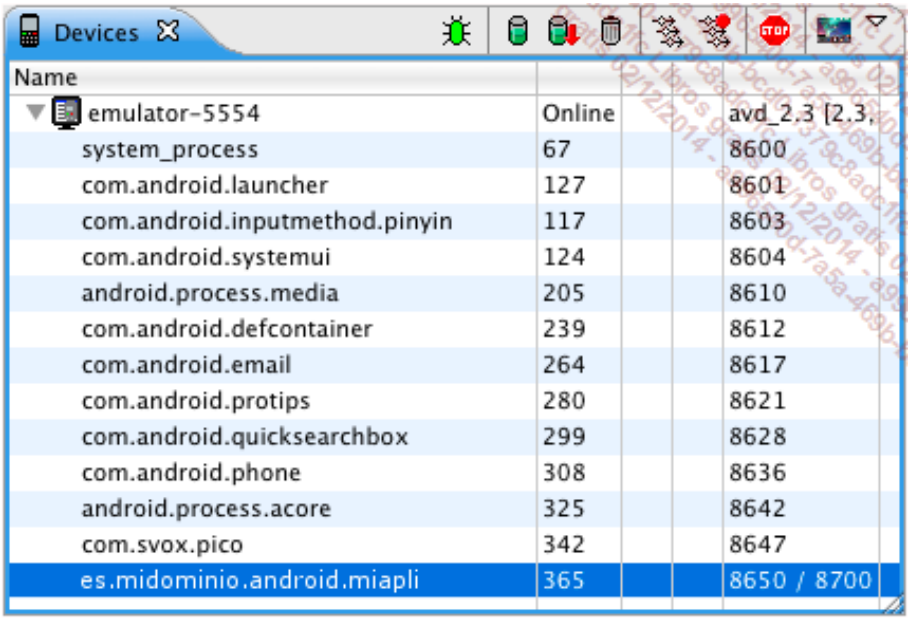


- Haga clic en el botón **DDMS** para mostrar la perspectiva.



La perspectiva **DDMS** contiene varias vistas nuevas: **Devices**, **Emulator Control**, **Threads**, **Heap**,**Allocation Tracker** y **File Explorer** que vamos a descubrir con detalle más adelante.

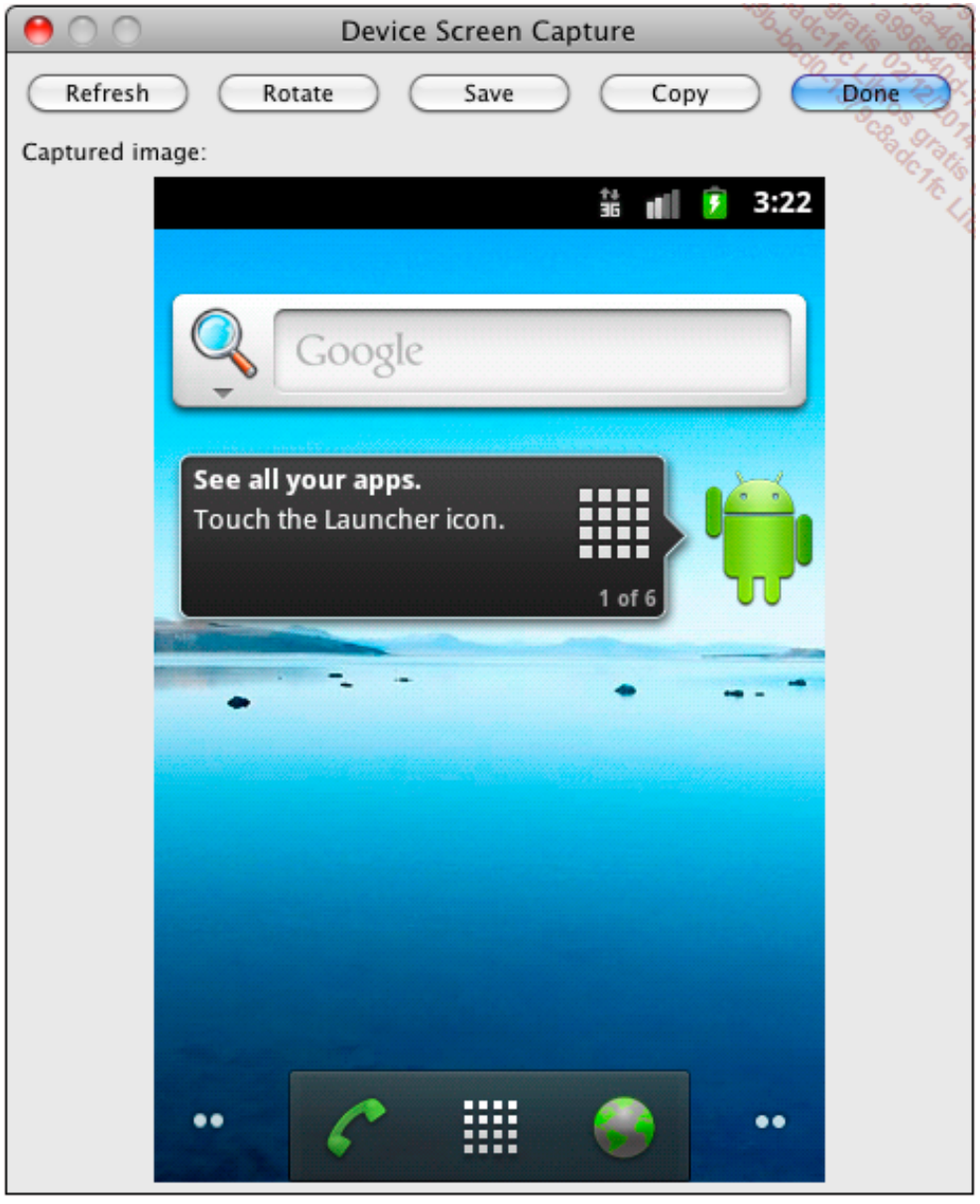
a. Vista Devices



La vista **Devices**, situada por defecto en la parte superior izquierda, proporciona la lista de emuladores y dispositivos conectados y, para cada uno de ellos, la lista de ciertos procesos existentes identificados por el nombre del paquete de la aplicación que los alberga. A continuación se indica, en las columnas de la derecha, el identificador del proceso y el puerto de conexión para el depurador.

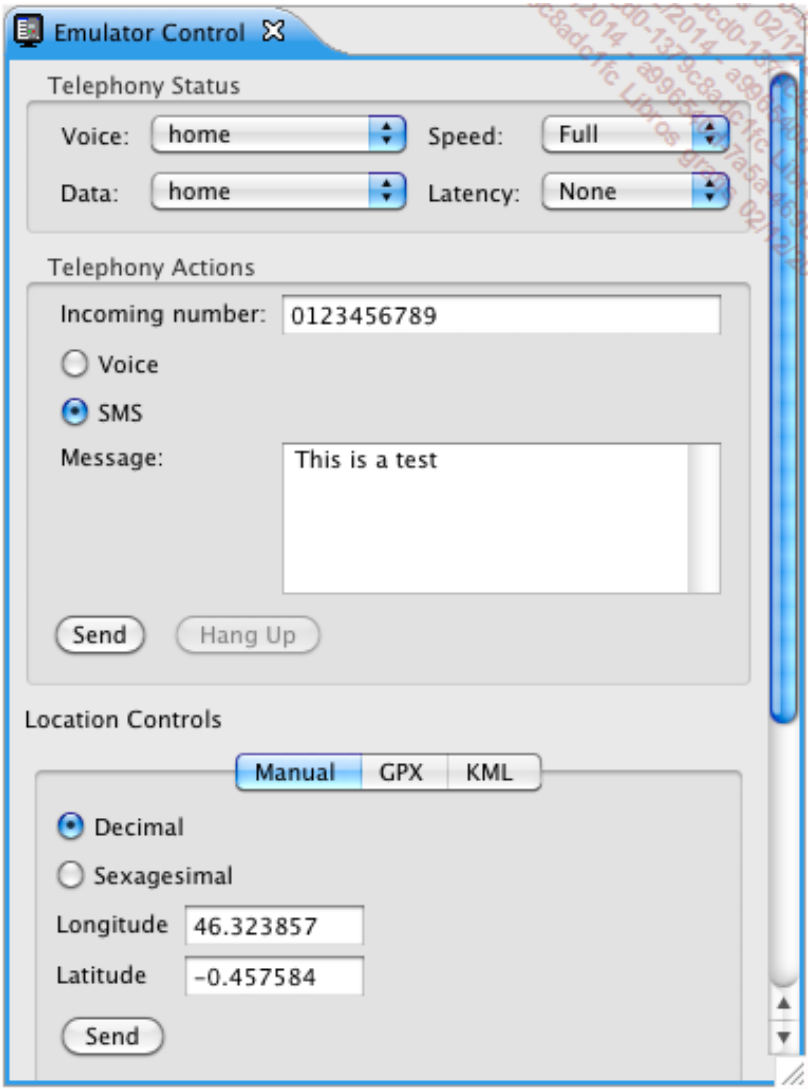
La selección de un proceso de la lista activa los botones situados en la parte superior derecha correspondientes a las funcionalidades siguientes, descritas de izquierda a derecha:

- **Debug the selected process**: pone la aplicación en modo debug. Requiere que el proyecto correspondiente esté abierto en el espacio de trabajo de Eclipse.
- **Update Heap**: activa las actualizaciones de la información de ejecución de la aplicación. Esta información se indica en la vista **Heap** descrita más adelante.
- **Dump HPROF file**: recupera un archivo con formato HPROF. Este archivo es una imagen de la memoria de la máquina virtual. Permite, en concreto, buscar las fugas de memoria estudiándola con las herramientas especializadas como Mat, jmap, jhat.
- **Cause GC**: fuerza la ejecución del Garbage Collector (recolector de basura).
- **Update Threads**: activa la actualización de la información acerca de los threads de la aplicación. Esta información está indicada en la vista **Threads** descrita más adelante.
- **Start Method Profiling**: el primer clic inicia el registro de información asociada a la ejecución de los métodos. El segundo clic detiene el registro y muestra los resultados en una ventana dedicada.
- **Stop Process**: destruye el proceso y, por tanto, la máquina virtual, seleccionado.
- **Screen Capture**: abre una nueva ventana **Device Screen Capture** que permite realizar capturas de pantalla de la aplicación seleccionada y salvaguardarlas. Un clic sobre el botón**Refresh** permite actualizar la copia de pantalla.



Todavía en la vista **Devices**, en el extremo derecho de las funcionalidades descritas anteriormente, aparece una flecha verde orientada hacia abajo. Un clic sobre esta flecha permite descubrir otra funcionalidad indicada debajo de la lista: **Reset adb**. Esto permite reiniciar la herramienta `adb` utilizada por la herramienta `ddms` para conectarse al emulador o al dispositivo Android si la situación exige tener que reiniciar el emulador/dispositivo y/o Eclipse.

b. Vista Emulator Control

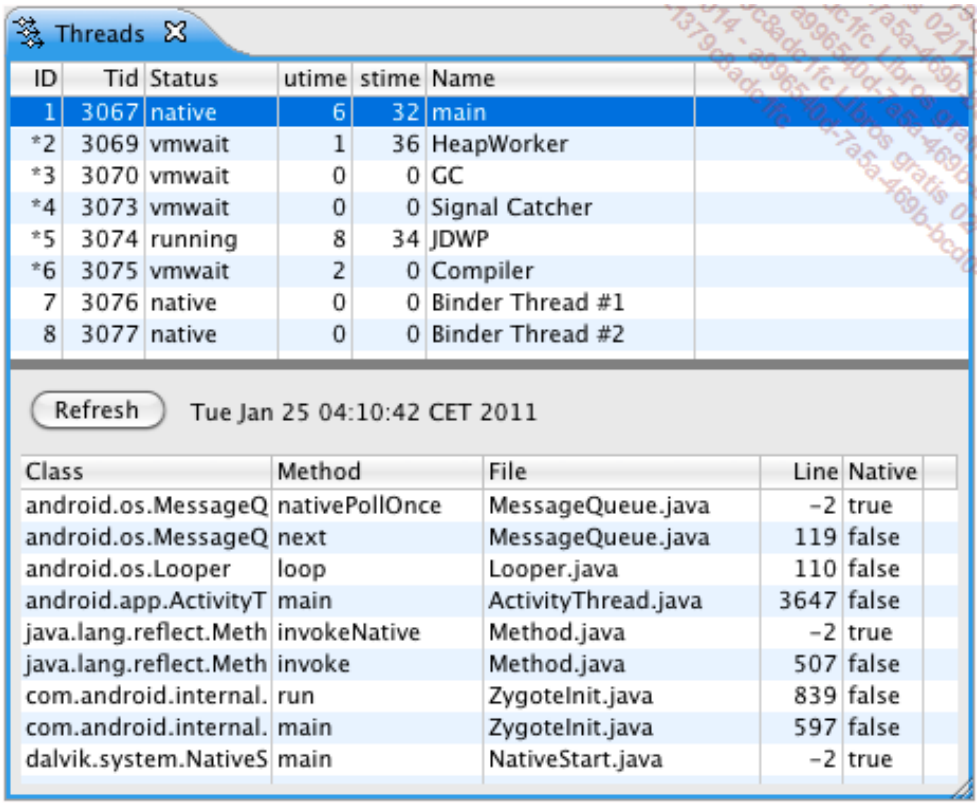


La vista **Emulator Control**, situada a la izquierda, debajo de la vista **Devices**, permite simular el estado de la red telefónica, las llamadas telefónicas, el envío de SMS y las coordenadas geográficas en distintos formatos: manual, GPX (*GPS Exchange Format*) y KML (*Keyhole Markup Language*).

Las coordenadas geográficas emuladas se reciben como provenientes exclusivamente del sistema GPS del dispositivo emulado. No es posible emular posiciones que provengan de la red de telefonía móvil o de redes Wi-Fi. El permiso correspondiente al sistema GPS debe, por tanto, figurar en la aplicación (véase el capítulo Sensores y geolocalización - Localización geográfica).

Observe la presencia de un bug en las versiones 2.3.x del emulador que provoca un crash en el emulador cuando se le proporciona una posición GPS. Para más información, consulte la siguiente página: http://code.google.com/p/android/issues/detail?id=13015

c. Vista Threads



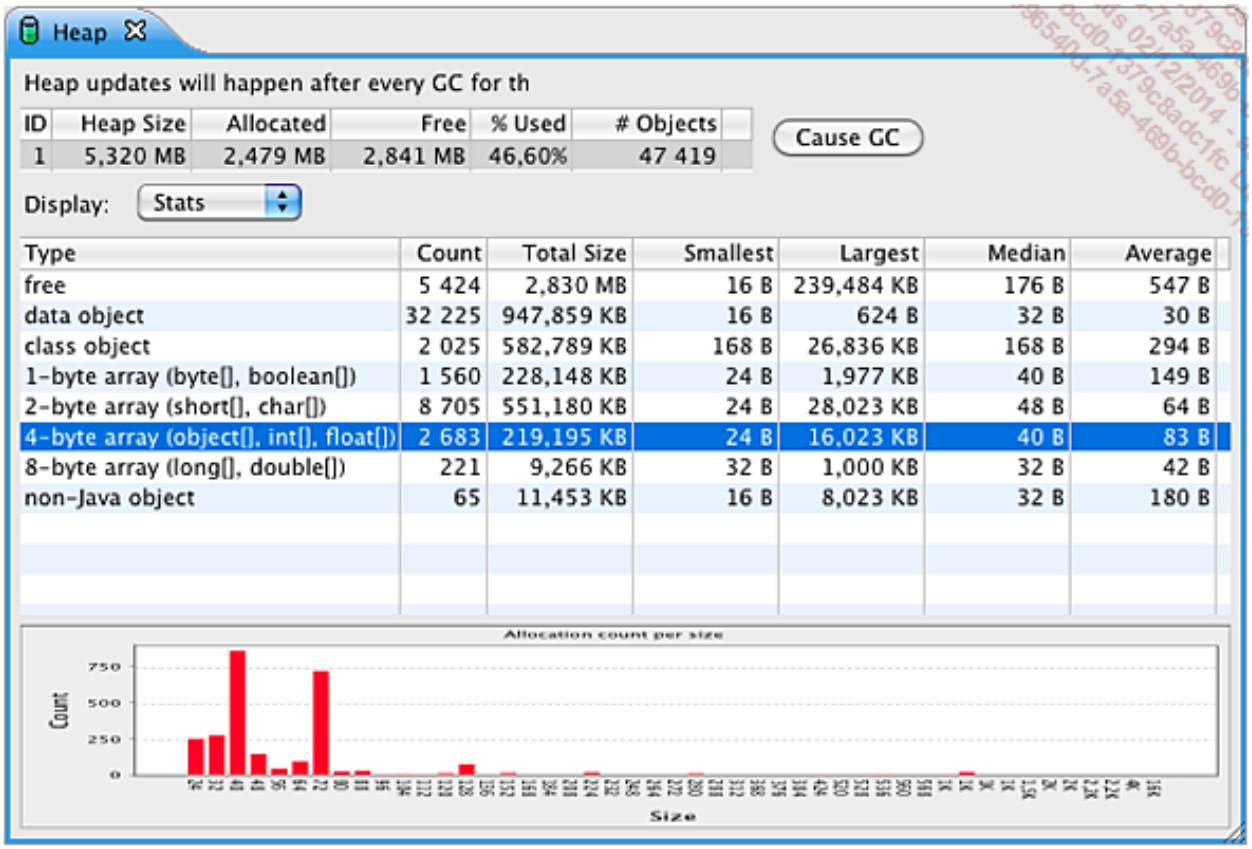
La vista **Threads**, como las siguientes vistas, está situada en la parte derecha.

- Para activar esta vista, seleccione una aplicación en la vista **Devices** y haga clic en el botón**Update threads**.

En la parte superior figura, de izquierda a derecha: un identificador único (ID) del thread atribuido por la máquina virtual, el identificador Linux del thread, el estado del thread, el tiempo acumulador hasta ejecutar el código de usuario, en unidades de 10 ms, y el nombre del thread.

En la parte inferior figura la pila de llamadas de los métodos del thread seleccionado en la parte superior.

d. Vista Heap



Situada a la derecha de la vista **Threads**, la vista **Heap** debe estar activa para recibir información.

- Para ello, seleccione una aplicación en la vista **Devices** y haga clic en el botón **Update Heap** .

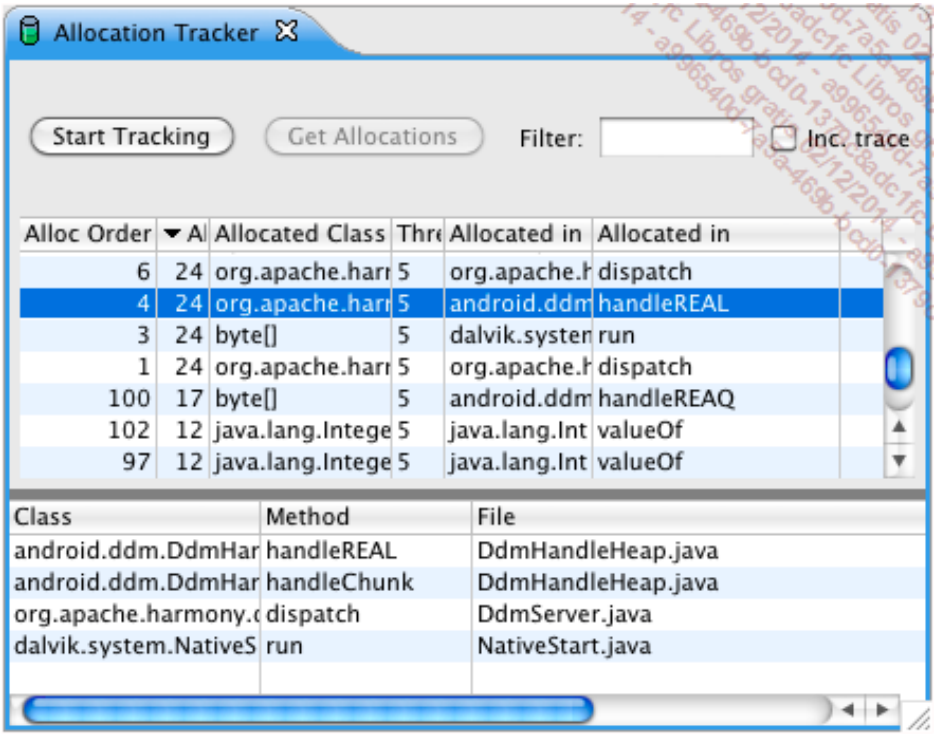
Aparece una indicación mientras que la vista nos informa de que el refresco de la información tendrá lugar tras cada ejecución del recolector de basura. Un clic sobre el botón **Cause GC** permite forzar manualmente la ejecución del recolector de basura y, por tanto, refrescar la información.

En la parte superior figura la información general asociada a la memoria: el tamaño total, el tamaño alojado, el tamaño libre, la proporción de memoria utilizada y el número de objetos alojados.

En la parte central se muestra información y datos estadísticos repartidos en diversas categorías.

En la parte inferior se dibuja un gráfico que representa el número de emplazamientos por tamaño. Si se hace clic con el botón derecho es posible modificar las propiedades, salvaguardar los datos, o imprimirlos.

e. Vista Allocation Tracker



Situada a la derecha de la vista **Heap**, la vista **Allocation Tracker** proporciona información sobre los elementos alojados en memoria.

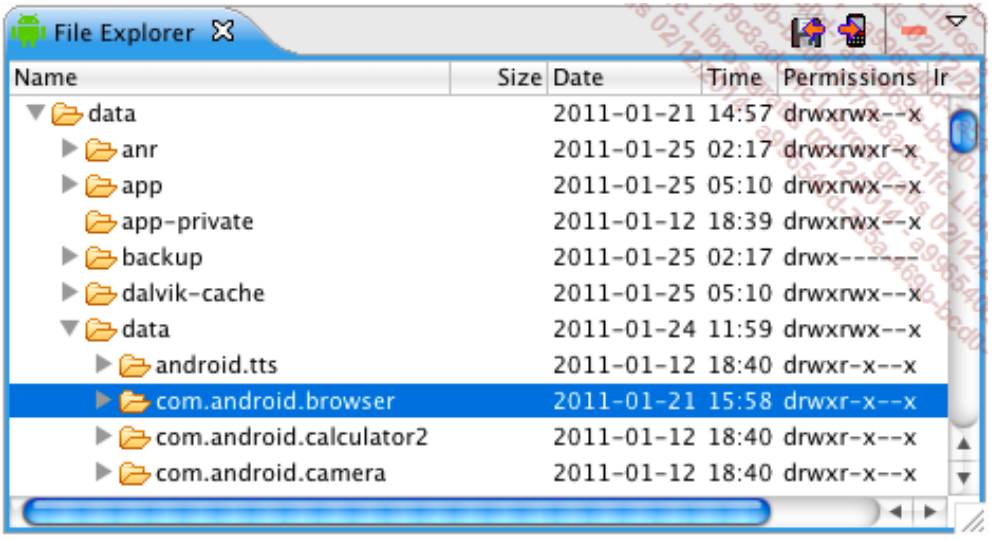
- Para ello, seleccione una aplicación en la vista **Devices** y, a continuación, haga clic sobre el botón **Start Tracking** de la vista **Allocation Tracker**.

Comienza entonces la recolección de datos sobre el consumo de memoria. Cada clic en el botón**Get Allocations** refresca los datos de la ventana.

Los datos proporcionan información sobre las reservas de memoria realizadas desde el inicio de la colecta. Haciendo clic en el botón **Stop Tracking** se detiene la recolección de datos.

Es posible ordenar las filas de la tabla haciendo clic sobre el título de la columna.

f. Vista File Explorer



Situada en la parte derecha de la vista **Allocation Tracker**, la vista **File Explorer** es un explorador del sistema de archivos del emulador o del dispositivo Android.

Es posible transferir archivos desde o hacia el sistema Android y suprimir archivos en el sistema Android haciendo clic en los respectivos iconos situados en la parte superior derecha.

Pruebas unitarias y funcionales

Las pruebas unitarias en Android están basadas en la biblioteca JUnit 3. Retoman, por tanto, la misma filosofía adaptándola a la plataforma Android.

Los API de pruebas en Android no son compatibles con JUnit 4.

Las pruebas también pueden crearse y ejecutarse en Eclipse mediante el plug-in ADT, tanto por línea de comandos como mediante la consola utilizando las herramientas SDK Android. Recuerde, en este libro nos centraremos únicamente en el uso de Eclipse y del plug-in ADT.

Una aplicación Android se prueba mediante otra aplicación Android: una aplicación de pruebas. Crearemos, por tanto, un proyecto de pruebas, le agregaremos las clases de pruebas que contengan las propias pruebas y finalizaremos ejecutando estas pruebas.

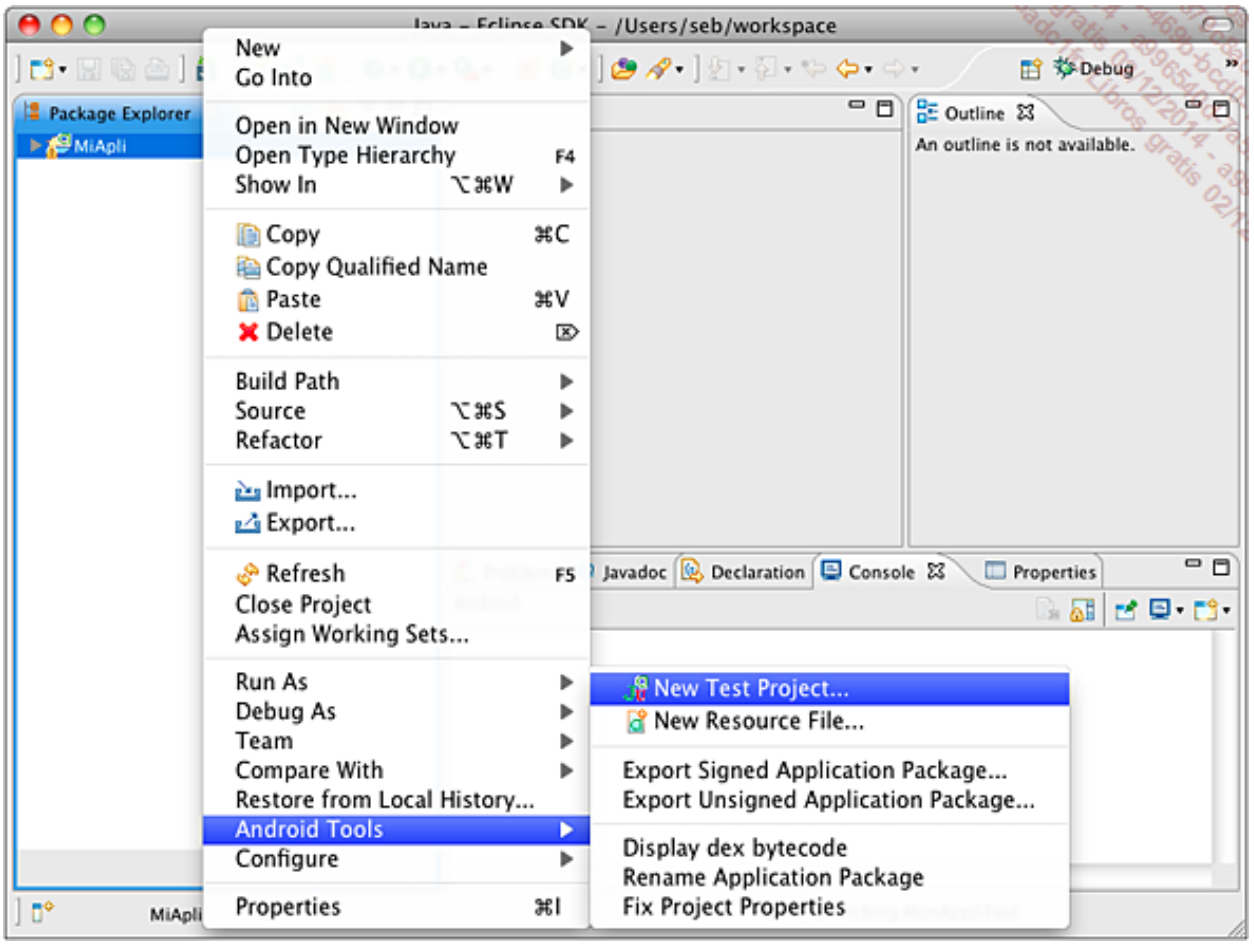
1. Creación de un proyecto de pruebas

Existen dos posibilidades para crear un proyecto de pruebas Android en Eclipse. La primera consiste en crearlo al mismo tiempo que el propio proyecto de la aplicación Android. La segunda consiste en crearlo posteriormente, como puede ser el caso para un proyecto que ya existe y que se quiere probar.

En ambos casos, la ventana Eclipse que se proporciona para crear el proyecto de test es sensiblemente igual.

Descubramos aquí cómo crear un proyecto de pruebas para un proyecto ya existente.

- En la vista **Package Explorer** (Explorador de paquetes) de Eclipse, haga clic con el botón derecho sobre el proyecto que quiera probar, a continuación haga clic sobre **Android Tools**.

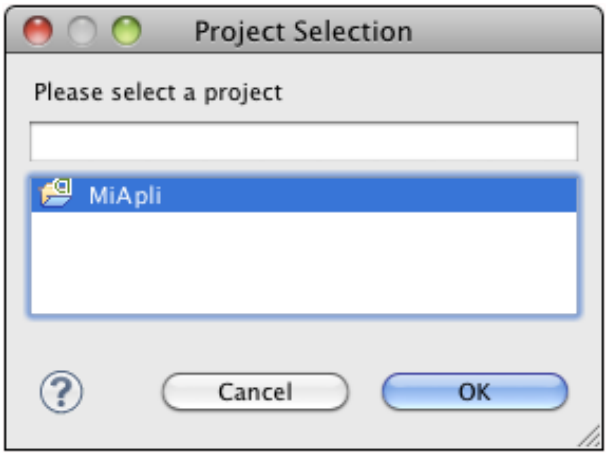


- Seleccione **New Test Project**.

Aparece la ventana **New Android Test Project**.

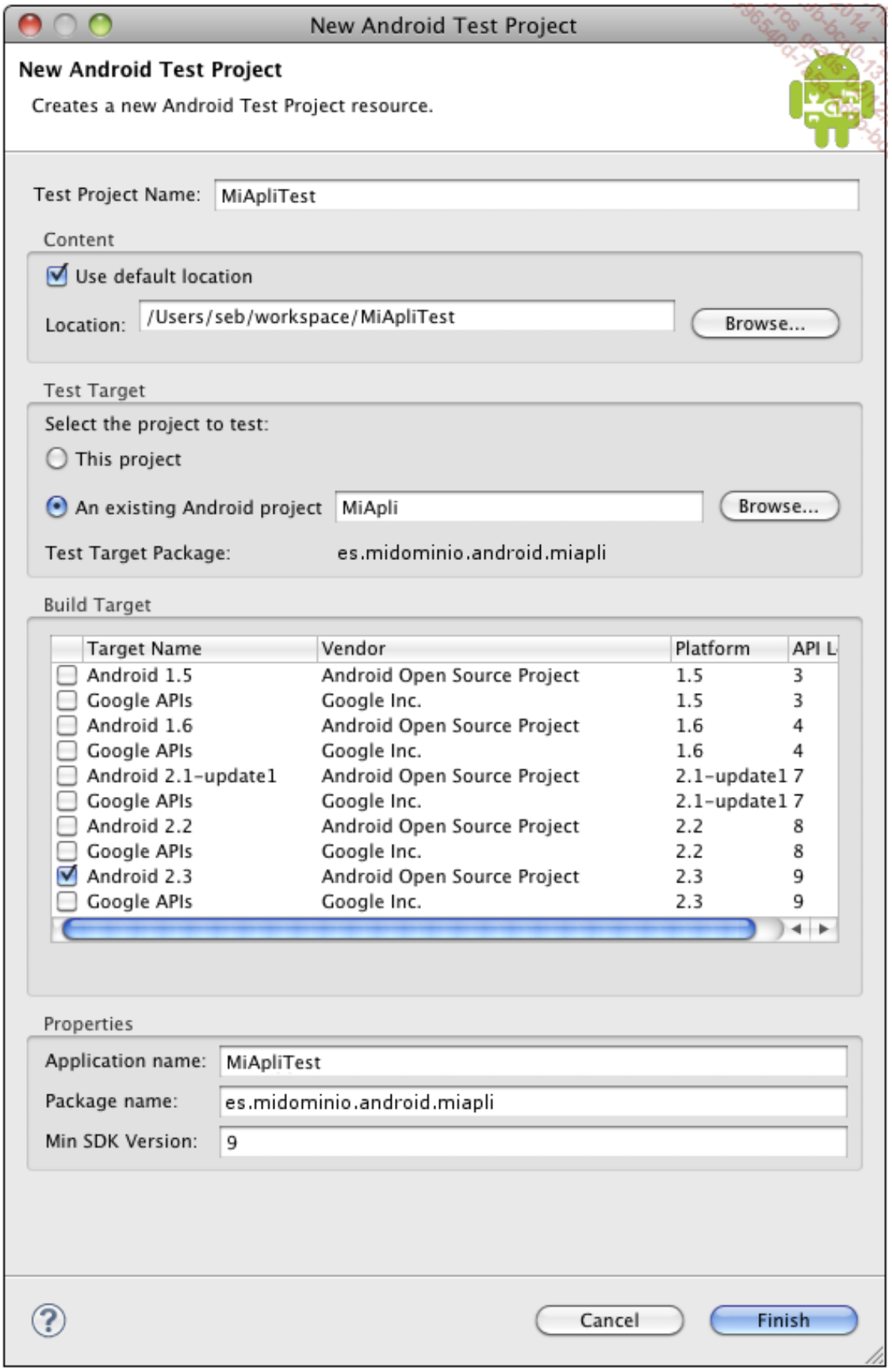
Antes de rellenar uno a uno los campos, seleccione un proyecto a probar y verá cómo se rellenan automáticamente la mayoría de los campos.

- En la zona **Test Target**, haga clic en el botón **Browse...** situado a la derecha del campo **An existing Android project**.



- Seleccione el proyecto a probar y haga clic sobre el botón **OK**.

Los campos se rellenan automáticamente.



También es posible rellenar los campos de forma manual o modificar los que se han rellenado de forma automática. He aquí sus correspondencias:

Test Project Name: nombre del proyecto de pruebas. Si bien el desarrollador tiene total libertad a la hora de seleccionar este nombre, se recomienda enlazarlo con el nombre del proyecto que se quiere probar. Para ello, es habitual retomar el nombre del proyecto que se quiere probar y agregarle la palabra `Test`. Por ejemplo, si el nombre del proyecto que se quiere probar es`MiAplicacion`, se utilizará `MiAplicacionTest`.

Content: permite indicar la ubicación del proyecto de `tests`. Se proporciona automáticamente una ubicación por defecto en el campo **Location** si la opción **Use default location** está marcada. Es posible especificar otro emplazamiento desmarcando esta opción. En este caso, utilizaremos la ubicación por defecto, lo que tendrá como consecuencia disociar físicamente la ubicación del proyecto de pruebas de la del proyecto que se quiere probar.

Para aquellos desarrolladores que prefieran tener el código fuente de las pruebas físicamente dentro del proyecto que se quiere probar, es habitual utilizar como carpeta raíz del proyecto de pruebas una carpeta llamada `tests` situada en la raíz del proyecto que se quiere probar. Es decir, en el mismo nivel que las carpetas `src` y `res`.

Test Target: permite indicar el proyecto que se quiere probar, bien el que está seleccionado o bien otro proyecto Android existente en el espacio de trabajo de Eclipse.

Build Target: como con la creación de un proyecto Android, es preciso seleccionar la versión del SDK Android que se va a utilizar.

Application name: nombre de la aplicación. Este nombre puede incluir espacios.

Package name: nombre del paquete Java. También en este caso, la elección es libre. No obstante es habitual escoger el mismo nombre de paquete que el que tiene el proyecto que se quiere probar y agregarle `.test` al final. Por ejemplo, si el proyecto que se quiere probar tiene como nombre de paquete `es.midominio.android.miaplicacion`, se utilizará`es.midominio.android.miaplicacion.test`.

Min SDK Version: versión mínima del SDK de destino. Corresponde con la versión utilizada en el campo **Build Target**. De hecho, el número de versión se especifica automáticamente en este campo. Una modificación en su valor modificará el valor del campo **Build Target**.

Haga clic sobre el botón **Finish**.

Si el botón **Finish** sigue de color gris, es porque uno o varios campos están mal informados. Para ayudarle a resolver esto, aparecen indicaciones en la parte superior de la ventana. También es posible que aparezcan mensajes de advertencia sin por ello bloquear el botón **Finish**.

Se ha creado y agregado el proyecto al espacio de trabajo. La estructura del proyecto es la misma que la de una aplicación Android (¡pues se trata de una aplicación Android!).

No obstante, el manifiesto creado automáticamente incluye una serie de etiquetas suplementarias:`instrumentation` y `uses-library`.

Archivo AndroidManifest.xml del proyecto de pruebas

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
    xmlns:android="http://schemas.android.com/apk/res/android"
    package="es.midominio.android.miaplicacion.test"
    android:versionCode="1"
    android:versionName="1.0">
  <application android:icon="@drawable/icon"
               android:label="@string/app_name">
    <uses-library android:name="android.test.runner" />
  </application>
  <uses-sdk android:minSdkVersion="9" />
  <instrumentation
    android:targetPackage="es.midominio.android.miaplicacion"
    android:name="android.test.InstrumentationTestRunner" />
</manifest>
```

La etiqueta `instrumentation` incluye dos atributos informados:

- `android:targetPackage`: identifica al paquete de la aplicación que se quiere probar.
- `android:name`: indica la herramienta, la clase, que ejecuta las pruebas.

Preste atención, verifique bien que el valor del campo `android:targetPackage` se corresponde con el paquete de la aplicación que se quiere probar y no con el de la aplicación de pruebas, es decir que no contiene la terminación `.test`. Esto puede ocurrir si se ha escogido la opción `This project` en la zona **Test Target** durante la selección del proyecto que se quiere probar.

Por defecto, el campo `android:name` contiene la clase básica del lanzador de pruebas Android`android.test.InstrumentationTestRunner`. Esta clase no se incluye en el código Android; forma parte de la biblioteca compartida `android.test.runner`. Por ello, es preciso indicar explícitamente que el proyecto la necesita utilizando la etiqueta `uses-library` y su atributo`android:name`.

2. Creación de una clase de caso de prueba

Ahora que se ha creado el proyecto, hay que agregarle los casos de prueba mediante la creación de clases de casos de prueba. Esta clase extiende de una de las numerosas clases de prueba de Android. El paquete en el que se crea esta clase de prueba representa un conjunto de pruebas.

Es posible utilizar la clase `TestCase` de JUnit para realizar pruebas no específicas a Android.

Si no, la clase de caso de prueba específica Android más sencilla es la clase `AndroidTestCase`. Esta clase extiende de las clases `TestCase` y `Assert` de JUnit. Por ello, hereda los métodos estándar de aserción de JUnit tales como `assertEquals`, `assertNull`, `fail`… Android proporciona otros métodos de aserción contenidos en las clases `MoreAsserts` y `ViewAsserts`.

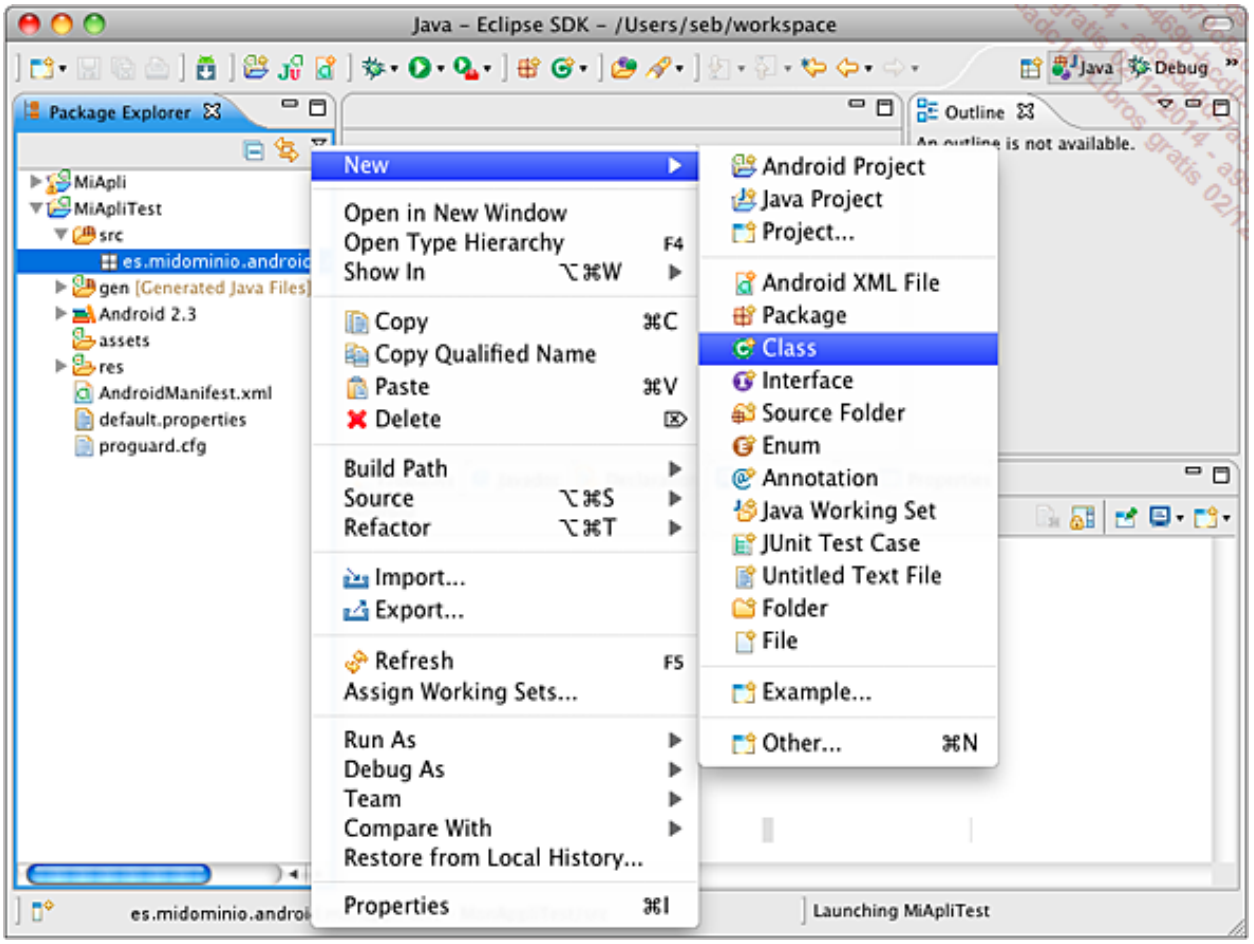
La clase `AndroidTestCase` proporciona los métodos básicos que se debe utilizar y/o extender para realizar las pruebas. Es costumbre definir de forma suplementaria un método`testPreconditions`. He aquí su descripción:

- `testPreconditions`: este método se invoca una única vez antes de ejecutar cualquier prueba. Permite verificar que la aplicación que se quiere probar se ha inicializado correctamente y está lista para ser probada. Si este método falla, entonces no se pueden tener en cuenta los resultados de las siguientes pruebas, pasen o no con éxito.
- `setUp`: este método se invoca antes de cada una de las pruebas de la clase. Permite inicializar las variables y el entorno de pruebas y, en especial, arrancar la actividad o el servicio que se quiere probar, por ejemplo. Este método debe invocar, obligatoriamente, a su método padre utilizando `super.setUp()`.
- `tearDown`: este método es el simétrico al método `setUp`. Se invoca desde cada una de las pruebas. Permite restaurar los parámetros del entorno.

Android proporciona otras clases de casos de prueba a medida para los principales componentes. Esto permite realizar pruebas específicas a la plataforma Android. Estas clases de prueba permiten probar específicamente las actividades, los servicios y los proveedores de contenido.

Veamos cómo crear una clase de caso de prueba para una actividad.

- En la vista **Package Explorer** de Eclipse, haga clic con el botón derecho sobre el paquete Java. En nuestro caso: `es.midominio.android.miaplicacion.test`. A continuación, haga clic sobre **New**.



- Seleccione **Class**.

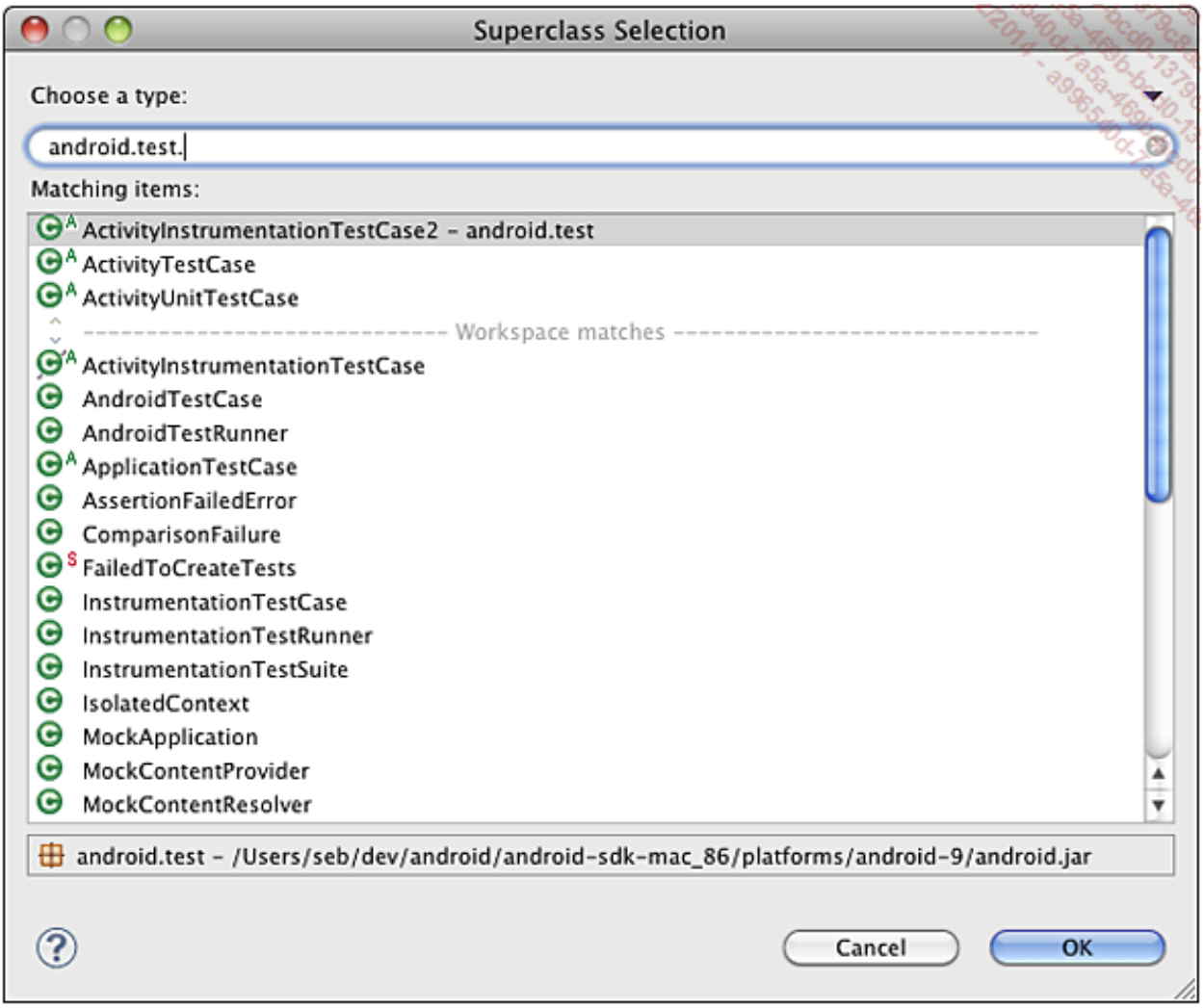
Aparece la ventana **New Java Class**.

- Informe los siguientes campos:

Name: nombre de la clase de pruebas. Por convención, se utilizará el nombre de la clase que se quiere probar y se le agregará la terminación `Test`. Por ejemplo, utilizaremos`MiActividadPrincipalTest` para probar la clase `MiActividadPrincipal` del proyecto principal.

Superclass: clase madre de la clase de prueba. Se corresponde con una clase de caso de prueba Android que se puede escoger entre las que se proporciona para probar las actividades, los servicios y los proveedores de contenidos. Pertenecen todas ellas al paquete `android.test`.

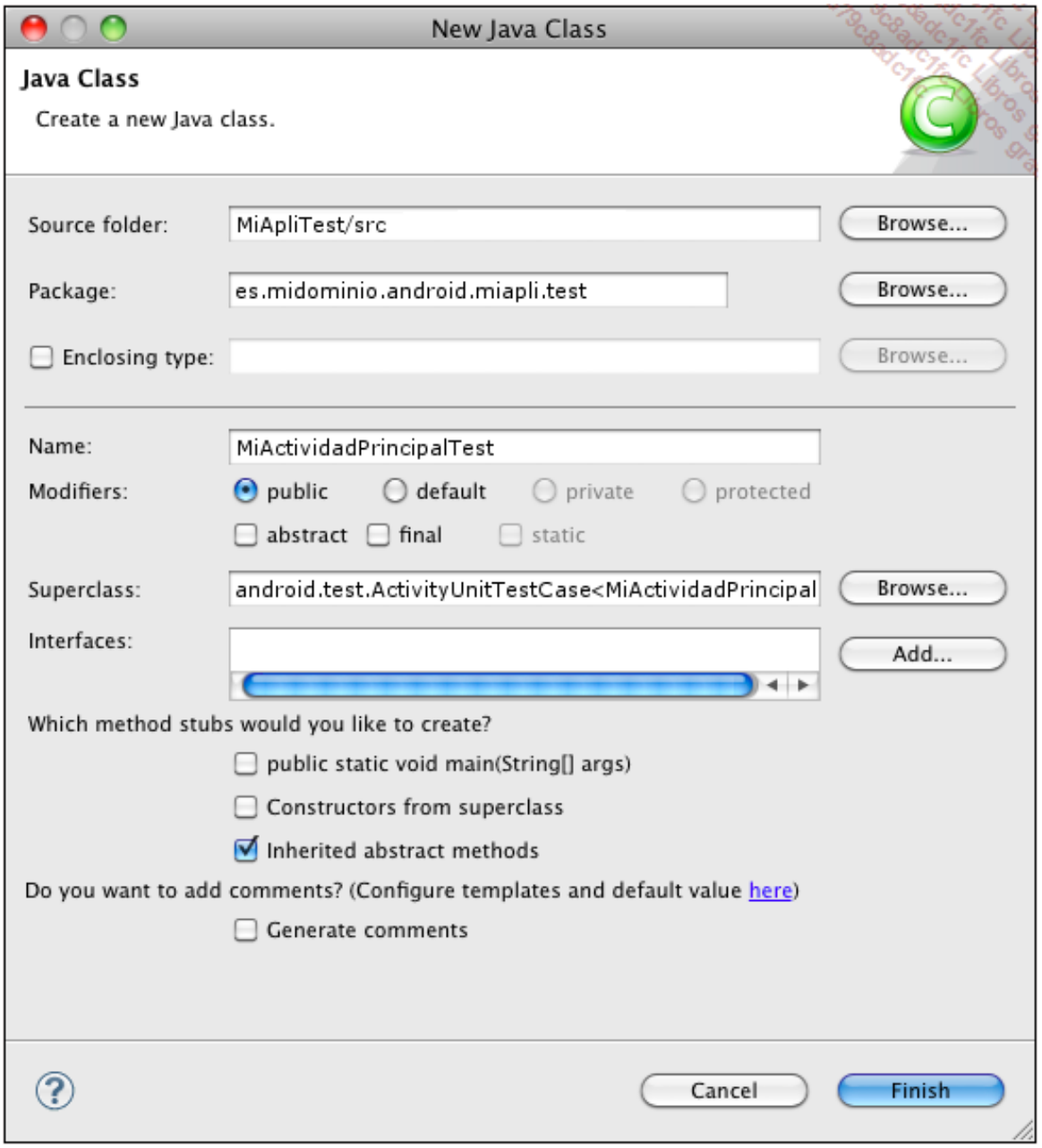
Para mostrarlas, basta con hacer clic en el botón **Browse**... e introducir `android.test`. en el campo **Choose a type** de la ventana **Superclass Selection**.



A continuación hay que hacer doble clic sobre la clase madre que se desee.

Por ejemplo, se utilizará la clase `android.test.ActivityUnitTestCase` para probar de forma aislada una única actividad. Y se remplazará el parámetro genérico `T` por el nombre de la clase que se quiere probar. En nuestro caso, se indicará:`android.test.ActivityUnitTestCase<MiActividadPrincipal>`.

Todos los demás campos pueden dejarse tal cual.



- Haga clic en **Finish**.

Se crea, a continuación, la clase correspondiente que hereda de la clase madre especificada.

Aparecen errores de compilación. En primer lugar, hay que importar la clase que se quiere probar. En nuestro caso se trata de:

```
import es.midominio.android.miaplicacion.MiActividadPrincipal;
```

Si Eclipse no puede resolver la importación es, sin duda, porque el proyecto que se quiere probar no se ha seleccionado de forma explícita durante la creación del proyecto de pruebas. Esto tiene como consecuencia que no se ha incluido la ruta de este proyecto en la ruta de compilación del proyecto a probar. Veamos cómo incluir explícitamente el proyecto a probar en la ruta de compilación del proyecto de pruebas:

- Haga clic con el botón derecho sobre el proyecto de pruebas en el **Explorador de paquetes** y, a continuación, seleccione **Build Path - Configure Build Path**.
- Si el proyecto que se quiere probar no figura en la lista, seleccione la pestaña **Projects**, haga clic en **Add...**, seleccione el proyecto que se quiere probar y haga clic en el botón **OK**.

Debería haberse resuelto el import.

Segunda etapa, JUnit requiere un constructor sin parámetro. Éste debe invocar en primer lugar al constructor padre pasándole como único parámetro la clase que se quiere probar.

Sintaxis del constructor padre

```
public ActivityUnitTestCase (Class<T> activityClass)
```

Ejemplo

```
package es.midominio.android.miaplicacion.test;

import android.test.ActivityUnitTestCase;
import es.midominio.android.miaplicacion.MiActividadPrincipal;

public class MiActividadPrincipalTest extends
    ActivityUnitTestCase<MiActividadPrincipal> {

  public MiActividadPrincipalTest() {
    super(MiActividadPrincipal.class);
  }

}
```

Llegados a este punto, el proyecto no debería contener ningún error de compilación.

Veremos, a continuación, en detalle las particularidades ligadas a las clases de casos de prueba correspondientes a las actividades, los servicios y los receptores de eventos.

a. Probar una actividad

Por su naturaleza y su buen funcionamiento particular, una actividad no puede probarse únicamente mediante la clase `AndroidTestCase`. Es por ello por lo que Android proporciona la clase de caso de prueba `InstrumentationTestCase`. Ésta es la clase básica de los casos de prueba de una actividad. Permite, en particular, controlar el ciclo de vida de la actividad que se está probando: arranque, pausa, destrucción, etc. Permite a su vez controlar el entorno de pruebas creando prototipos (objetos simulados) o incluso simular una interacción con la interfaz de usuario. Esta clase extiende de las clases `TestCase` y `Assert` de JUnit.

Varias clases extienden de esta clase para especializar todavía más los casos de prueba. He aquí algunas de ellas:

- `ActivityUnitTestCase`: esta clase de caso de prueba permite probar una única actividad en un entorno aislado. Es decir, las pruebas se realizan con objetos de tipo`Context` y `Application` simulados. No es posible enviar objetos de tipo `Intent`simulados a la actividad que se está probando excepto mediante la llamada al método`startActivity(Intent intent)`.
- `ActivityInstrumentationTestCase2`: esta clase de caso de prueba permite realizar pruebas funcionales de una o varias actividades en un entorno normal. Es decir, que las pruebas se realizan con objetos de tipo `Context` y `Application` reales. Los objetos de tipo `Intent` simulados permiten probar estas actividades.
- `SingleLaunchActivityTestCase`: esta clase permite probar las actividades que se ejecutan en un modo distinto al modo estándar. Para simular el funcionamiento de estos otros modos, esta clase invoca a los métodos `setUp` y `tearDown` una única vez para el conjunto de las pruebas. Por ello, el entorno de pruebas no cambia entre cada prueba. Por último, no es posible pasarle ningún objeto simulado: `Context`, `Application`, `Intent`…

Los objetos simulados se definen en el paquete `android.test.mock`.

b. Probar un servicio

Por su naturaleza independiente, un servicio puede probarse muy fácilmente de forma aislada utilizando la clase `ServiceTestCase`. Esta clase permite, en particular, inyectar objetos simulados de tipo `Context` y `Application` antes incluso de arrancar el servicio.

c. Probar un receptor de eventos

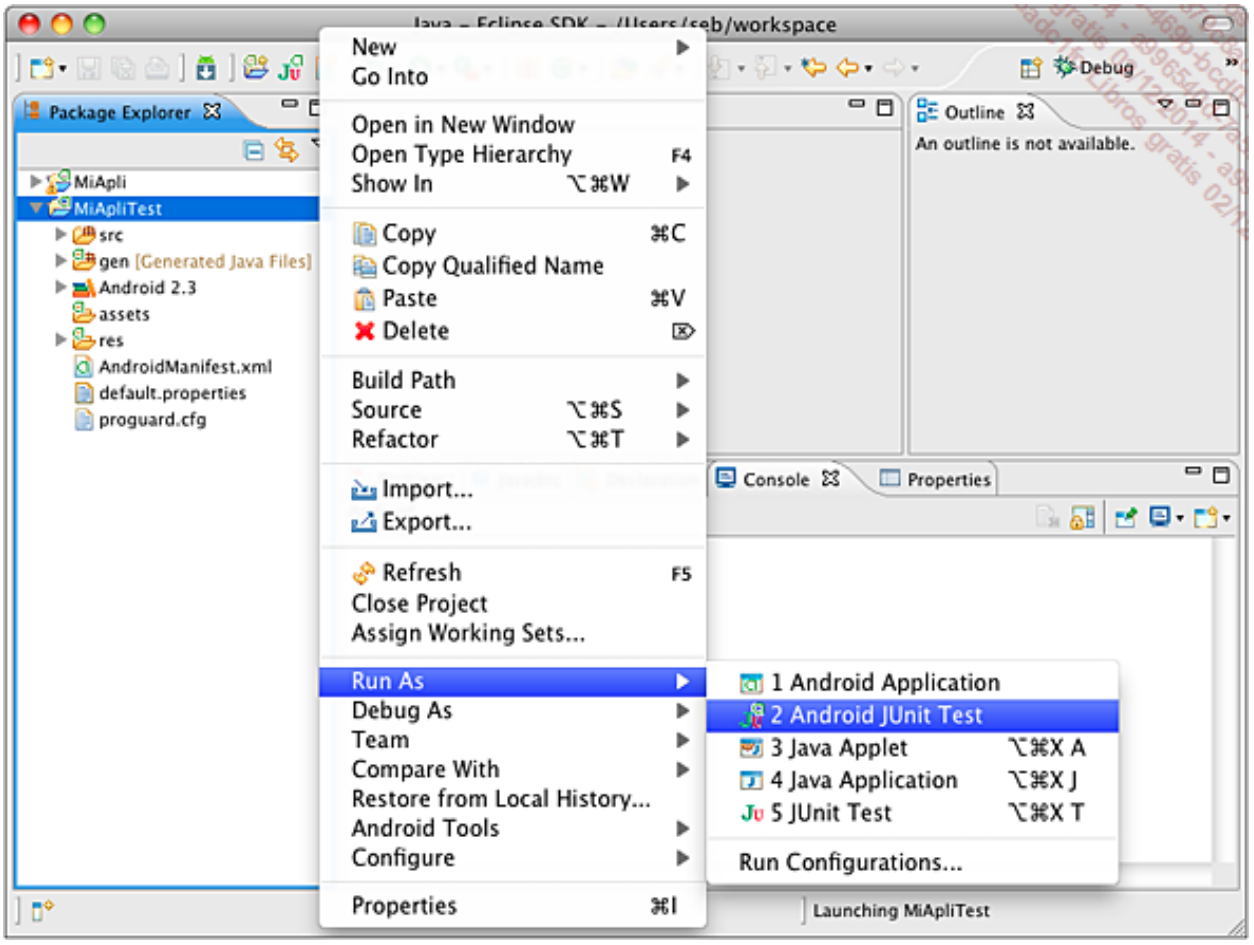
No existen clases de prueba previstas de las que se pueda heredar para probar un receptor de eventos. La razón es simple: para probarlo, basta con probar el objeto que se comunica con este receptor de eventos mediante el envío de objetos de tipo `Intent`.

3. Ejecución de las pruebas

Como hemos visto a lo largo de la creación del proyecto de pruebas, es la herramienta indicada en la etiqueta `instrumentation` del manifiesto la que va a ejecutar las pruebas. Y son estos casos de prueba los que van a ejecutar la aplicación que se quiere probar pasándole objetos simulados o no.

Como para cualquier aplicación Android, ésta sólo puede ejecutarse desde su emulador o desde un dispositivo físico. En ambos casos, la versión del sistema Android debe ser como mínimo igual a la versión correspondiente al SDK especificada en la aplicación de pruebas.

- En la vista **Package Explorer** de Eclipse, haga clic con el botón derecho sobre el proyecto de pruebas y, a continuación, haga clic en **Run As**.



- Seleccione **Android JUnit Test**.

El plug-in ADT ejecuta de forma interna la herramienta `adb`, que ejecuta el lanzador de pruebas indicado en el manifiesto. A continuación, se cargan en memoria la aplicación de pruebas y la aplicación que se quiere probar. Una vez cargadas, el lanzador realiza las pruebas.

Los resultados aparecen en la vista **JUnit** que se encuentra en una pestaña situada junto a la vista **Package Explorer**.

Prueba del mono

Como hemos visto anteriormente, las pruebas unitarias permiten realizar casos de prueba precisos informados por el desarrollador. Pero, si bien el propio desarrollador habrá tenido en cuenta el describir la máxima cantidad de casos de prueba, en un entorno real, el usuario podrá interactuar con la aplicación de una forma que no esté prevista por el desarrollador. Es en este punto donde interviene el concepto de prueba del mono para ayudar al desarrollador a reducir el campo de interacciones de usuario que no se hayan probado.

La prueba del mono es una prueba aleatoria de la aplicación mediante su interfaz gráfica. Esta prueba simula un uso de la interfaz gráfica de usuario mediante la pulsación de teclas, realización de gestos táctiles, clics y demás eventos que cualquier usuario puede realizar normalmente de forma completamente aleatoria. Esto permitirá probar numerosos casos de uso en los que el desarrollador no haya pensado como, por ejemplo, la validación de un formulario mientras sus campos siguen estando vacíos.

Android pone a disposición del desarrollador la herramienta Monkey (Mono) para realizar este tipo de prueba. Esta herramienta se ejecuta desde el shell de la plataforma Android, emulador o periférico.

Sintaxis

```
adb shell monkey [opciones] [número de eventos]
```

Para mostrar la lista completa de opciones disponibles, especifique la opción `–help`.

Ejemplo

```
$ adb shell monkey -p es.midominio.android.miaplicacion -v 100
:Monkey: seed=0 count=100
:AllowPackage: es.midominio.android.miaplicacion
:IncludeCategory: android.intent.category.LAUNCHER
:IncludeCategory: android.intent.category.MONKEY
// Event percentages:
//   0: 15.0%
//   1: 10.0%
//   2: 15.0%
//   3: 25.0%
//   4: 15.0%
//   5: 2.0%
//   6: 2.0%
//   7: 1.0%
//   8: 15.0%
:Switch:
#Intent;action=android.intent.action.MAIN;category=android.intent.
category.LAUNCHER;launchFlags=0x10000000;component=es.midominio.
android.miaplicacion/.MiActividad;end
    // Allowing start of Intent { act=android.intent.action.MAIN
cat=[android.intent.category.LAUNCHER]
cmp=es.midominio.android.miaplicacion/.MmiActividad } in package
es.midominio.android.miaplicacion
:Sending Pointer ACTION_MOVE x=-4.0 y=2.0
:Sending Pointer ACTION_UP x=0.0 y=0.0
:Sending Pointer ACTION_DOWN x=47.0 y=122.0
:Sending Pointer ACTION_UP x=29.0 y=129.0
:Sending Pointer ACTION_DOWN x=255.0 y=259.0
:Sending Pointer ACTION_UP x=255.0 y=259.0
:Sending Pointer ACTION_DOWN x=295.0 y=223.0
:Sending Pointer ACTION_UP x=290.0 y=213.0
:Sending Pointer ACTION_MOVE x=-5.0 y=3.0
:Sending Pointer ACTION_MOVE x=0.0 y=-5.0
    // Rejecting start of Intent { act=android.intent.action.MAIN
```

```
cat=[android.intent.category.HOME]
cmp=com.android.launcher/com.android.launcher2.Launcher } in
package com.android.launcher
:Sending Pointer ACTION_DOWN x=74.0 y=41.0
:Sending Pointer ACTION_UP x=74.0 y=41.0
:Sending Pointer ACTION_MOVE x=3.0 y=-2.0
:Sending Pointer ACTION_UP x=0.0 y=0.0
:Sending Pointer ACTION_MOVE x=-4.0 y=2.0
Events injected: 100
:Dropped: keys=0 pointers=0 trackballs=0 flips=0
## Network stats: elapsed time=1061ms (0ms mobile, 0ms wifi,
1061ms not connected)
// Monkey finished
```

En este ejemplo, la herramienta ha ejecutado la aplicación identificada mediante el paquete`es.midominio.android.miaplicacion`, le ha enviado 100 eventos mostrando la progresión de los resultados.

Sin una indicación específica por parte del usuario de las opciones correspondientes, la herramienta Monkey se detendrá cuando ocurra un problema tal como una excepción no capturada, el bloqueo de la aplicación o un crash.

La herramienta Monkey permite reproducir de forma idéntica y varias veces la misma prueba, la misma secuencia de eventos, pasándole el mismo número (semilla) como entrada mediante la opción `-s`. Esto permite verificar la corrección de un error descubierto antes por la misma prueba.

Introducción

La publicación de una aplicación consiste en ponerla a disposición de terceros. Puede ser de carácter privado como, por ejemplo, la publicación de una aplicación interna de una empresa para su uso en los smartphones y tabletas táctiles de sus empleados. O puede ser de carácter público.

Esta aplicación puede distribuirse de forma gratuita y, en este caso, todos los medios de distribución son buenos: sitios de venta de aplicaciones, como Play Store de Google (antes Android Market) y lugares de venta alternativos implementados por ciertos operadores, o incluso sitios web que proporcionan la descarga directa del archivo apk.

La aplicación también puede ponerse a la venta a una tarifa fija. Incluso en este caso, existe una gran cantidad de puntos de venta para vender aplicaciones. Con el objetivo de protegerse frente a la copia ilegal, la aplicación tendrá que usar algún sistema de protección (véase el capítulo Funcionalidades avanzadas - Proteger las aplicaciones de pago).

En este capítulo detallaremos únicamente la publicación en Play Store, el lugar de venta de las aplicaciones Android. Play Store, del lado del cliente, es una aplicación instalada por defecto en la mayoría de sistemas Android que permite al usuario, entre otros, descargar e instalar aplicaciones gratuitas y de pago.

La publicación de una aplicación requiere una preparación previa. Es necesario, como requisitos previos:

- Especificar un número de versión.
- Agregar, llegado el caso, un Contrato de Licencia de Usuario Final o CLUF específico a la aplicación. Este contrato detalla la responsabilidad de cada uno, el desarrollador y el usuario. Permite, por tanto, proteger al desarrollador y sus derechos de autor. Este contrato se muestra, por lo general, al usuario durante la primera ejecución de la aplicación. Éste debe leer y aceptar los términos del contrato para poder continuar y utilizar la aplicación.
- Utilizar, llegado el caso, el sistema de licencia de aplicación provisto por Play Store. Éste permite, en el caso de las aplicaciones de pago, verificar si la aplicación es una copia legal o no, es decir, si realmente la ha comprado el usuario en Play Store. Si no fuera el caso, el desarrollador puede modificar el comportamiento de la aplicación y, en particular, impedir su uso al usuario.
- Suprimir los logs de la aplicación durante la compilación de la versión final y, llegado el caso, borrar, o poner a `false` si estuviera explícitamente informado a `true`, el atributo`android:debuggable` de la etiqueta `application` especificada en el manifiesto.

La documentación oficial de Android indica que los mensajes que utilizan la clase `Log` se compilan pero no se utilizan durante la ejecución de la aplicación final. Desgraciadamente, éste no es necesariamente el caso.

- Probar la aplicación final lo más exhaustivamente posible (véase el capítulo Trazas, depuración y pruebas - Pruebas unitarias y funcionales y Prueba del mono). Es muy importante probar la aplicación, de forma prioritaria, sobre dispositivos Android físicos en condiciones de uso reales. Por defecto, o para configuraciones de hardware especiales, el emulador será una gran ayuda.

Retomando algunas de estas etapas, descubriremos en primer lugar cómo especificar un número de versión de aplicación. A continuación, detallaremos el procedimiento a seguir para proteger, llegado el caso, las aplicaciones de pago utilizando el sistema de licencia de aplicación provisto por Play Store. Compilaremos, a continuación, la aplicación en versión final y la firmaremos digitalmente. Para terminar, publicaremos la aplicación en Play Store.

Preliminares

Antes de realizar el empaquetado final de la aplicación, hay que verificar algunos puntos tales como la versión de la aplicación y los filtros configurados en la aplicación que permiten ponerla a disposición solamente de aquellos dispositivos compatibles.

1. Versión de la aplicación

Es costumbre, en informática, adjuntar un número de versión a todo software. Este número de versión debe ser un identificador único, de otro modo no tendría mucho interés.

En Android, existen dos campos en el manifiesto que tienen como objetivo determinar este número de versión:`android:versionCode` y `android:versionName`

a. android:versionCode

`android:versionCode` es un número entero. Este número no se muestra al usuario. Permite a las demás aplicaciones y a Play Store poder comparar dos versiones de la aplicación para determinar cuál de las dos se corresponde con la actualización, a saber, la que tenga un número de versión mayor.

Es habitual comenzar la numeración por 1 e incrementar el valor con cada actualización oficial de la aplicación, sea mayor o menor.

El sistema Android no verifica este valor, y deja total libertad al desarrollador a la hora de especificar este valor. Corresponde al desarrollador velar por que la última versión disponible posea, siempre, el valor mayor.

Sintaxis

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    android:versionCode="entero" >
  ...
</manifest>
```

Ejemplo

```
<manifest
    xmlns:android="http://schemas.android.com/apk/res/android"
    package="es.midominio.android.miaplicacion"
    android:versionCode="42" >
</manifest>
```

b. android:versionName

`android:versionName` es el número de versión que se muestra al usuario en Play Store. Es un elemento clave de carácter informativo que sirve tanto al desarrollador como al usuario.

Para el desarrollador, este número de versión permite especificar las versiones mayores, las versiones menores, las versiones correctoras… Permite a su vez, cuando la aplicación se ha publicado bajo diversas versiones, asociar bugs a una u otra o versión.

Para el usuario, el número de versión permite verificar que se está utilizando la última actualización de la aplicación. Le permite juzgar la conveniencia de una actualización según la importancia del cambio de versión, mayor o menor. Por último, le permite también distinguir la versión en las conversaciones con otros usuarios o con el desarrollador.

El desarrollador tiene libertad en cuanto a la sintaxis que quiera adoptar para la numeración de las

versiones. El valor debe ser una cadena de caracteres.

Este número de versión debe ser único y diferente, como mínimo, con cada publicación de la aplicación. En caso contrario nadie sabrá de qué variante de la aplicación se trata.

Sintaxis

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    android:versionName="cadena de caracteres" >
    ...
</manifest>
```

Ejemplo

```
<manifest
  xmlns:android="http://schemas.android.com/apk/res/android"
  package="es.midominio.android.miaplicacion"
  android:versionName="13.3.7" >
</manifest>
```

En este ejemplo, la sintaxis utilizada respeta la siguiente convención:*número_mayor.número_menor.número_build*.

2. Filtros para el mercado

Pensemos, por ejemplo, en una aplicación de fotografía que, por definición, requiere la presencia de una cámara fotográfica instalada en el dispositivo Android. Si esta aplicación se le propusiera a un usuario cuyo dispositivo Android no poseyera cámara, correría el riesgo de no funcionar correctamente.

En este caso, el usuario podría verse motivado a expresar su descontento en forma de comentario y de nota mediocre, los cuales son públicos en Play Store y pueden ser consultados por los demás usuarios antes de decidirse a la hora de instalar una aplicación. Es, por tanto, muy importante no permitir la instalación de la aplicación sobre un dispositivo incompatible.

Para evitar este tipo de disgustos, Play Store filtra las aplicaciones para proponer solamente aquellas que son compatibles con el dispositivo del usuario. Para hacer esto, se basa en la información proporcionada por la aplicación indicando las características de hardware y software de las que depende la aplicación.

Esta información se provee en el manifiesto en las etiquetas `uses-feature` y `uses-configuration`.

Esta información tiene un carácter meramente informativo. El sistema Android no las utiliza, ni siquiera para filtrar la instalación de una aplicación o para autorizar su uso en el dispositivo. Se utilizan, principalmente, en programas específicos como Play Store para filtrar los dispositivos compatibles. No reemplazan, en ningún caso, a las solicitudes de permisos necesarios para utilizar algún hardware o componente de software.

Play Store utiliza también la etiqueta `uses-sdk` y su atributo `android:minSdkVersion`, las etiquetas `uses-library` y la etiqueta `supports-screen` para filtrar las aplicaciones compatibles con el dispositivo del usuario.

a. uses-feature

La etiqueta `uses-feature` permite indicar las características de hardware y software de las que depende la aplicación. Se recomienda informarlas todas. Los valores son de tipo`android.hardware.*` y `android.software.*`.

El atributo `android:required` permite indicar si la característica correspondiente es obligatoria o no. Por defecto, lo es. No obstante, puede ser interesante que la aplicación gestione el caso en que la característica no esté presente. Esto permite abrir la aplicación a más usuarios, en concreto a aquellos que utilicen tabletas táctiles y que no dispongan, necesariamente, de las mismas capacidades de hardware que, por ejemplo, los teléfonos móviles.

Play Store utiliza las declaraciones de permisos requeridos por la aplicación para verificar la lista de características requeridas obligatoriamente y completarla automáticamente. Por ejemplo, si una aplicación solicita el permiso `ACCESS_FINE_LOCATION`, Play Store deducirá lascaracterísticas `android.hardware.location` y `android.hardware.location.gps` y, por tanto, la presencia de un componente de hardware de localización GPS, con carácter obligatorio.

Sintaxis

```
<uses-feature android:name="cadena de caracteres"
  android:required="booleano"
  ... />
```

Ejemplo

```
<uses-feature android:name="android.hardware.location" />
<uses-feature android:name="android.hardware.wifi"
  android:required="false" />
```

En el caso en que la característica no se requiera obligatoriamente, la aplicación debe probar dinámicamente si el dispositivo la posee o no, para adaptar su comportamiento. Para ello, es preciso usar el método `hasSystemFeature` de la clase `PackageManager` cuya instancia devuelve el método `getPackageManager`. Este método recibe como parámetro la característica que se quiere probar como cadena de caracteres y devuelve un valor booleano.

Sintaxis

```
public abstract boolean hasSystemFeature (String name)
```

Ejemplo

```
boolean gpsPresent = getPackageManager()
.hasSystemFeature(PackageManager.FEATURE_LOCATION_GPS);
```

b. uses-configuration

La etiqueta `uses-configuration` permite, a su vez, indicar las combinaciones de hardware requeridas por la aplicación. Cada uso de esta etiqueta corresponde con una combinación de hardware y software compatible.

Esta etiqueta incluye varios atributos, dos de los cuales describimos a continuación. Por ejemplo, el atributo `android:reqHardKeyboard` permite especificar si la aplicación requiere un teclado físico o no. El atributo `android:reqTouchScreen` permite indicar si la aplicación requiere una pantalla táctil.

Por defecto, en ausencia de esta etiqueta, se considera que la aplicación no requiere ni teclado físico, ni pantalla táctil.

Sintaxis

```
<uses-configuration android:reqHardKeyboard="booleano"
  android:reqTouchScreen="[finger|notouch|stylus|undefined]"
/>
```

Ejemplo

```
<uses-configuration android:reqHardKeyboard="true" />
```

Firma digital de la aplicación

Toda aplicación Android debe estar firmada digitalmente para poder instalarse y ejecutarse sobre un sistema Android, bien sea sobre un emulador o sobre un dispositivo Android. Esto permite, en particular, identificar al desarrollador de la aplicación.

La aplicación se distribuye bajo la forma de un archivo con formato APK (*Android Package*).

El archivo `.apk` es un archivo de formato zip que contiene toda la aplicación: los archivos Java compilados como .class, los archivos de recursos y los archivos XML compilados con formatoWBXML (wap binary xml).

Durante la fase de desarrollo de una aplicación, el desarrollador puede compilarla, instalarla y ejecutarla numerosas veces. Esto supone firmar cada una de estas compilaciones, sin lo cual no es posible instalarlas sobre el sistema Android. No obstante, hasta aquí, no se ha requerido ninguna solicitud de firma digital y todas las compilaciones de las aplicaciones han podido instalarse sin problema sobre el emulador o sobre un dispositivo Android. Lo que contradice los principios expuestos.

La razón es sencilla. Para no entorpecer el trabajo del desarrollador solicitándole firmar cada una de las compilaciones, el proceso de firma difiere según el modo de compilación usado.

1. Compilación en modo debug

Es el modo de compilación usado por defecto en Eclipse con el plug-in ADT durante la fase de desarrollo de la aplicación. Durante la compilación de una aplicación en este modo, el plug-in ADT crea automáticamente un almacén de claves y crea una clave privada específicamente dedicada a la depuración (debug).

Esta clave se utiliza a continuación para firmar automáticamente el archivo apk, optimizarlo e instalarlo sobre el sistema Android. Todas estas fases se realizan de forma totalmente transparente para el desarrollador. Esto le permite, por tanto, no tener que preocuparse de este aspecto y centrarse en el desarrollo de la aplicación.

La aplicación firmada de esta forma no puede distribuirse en Play Store.

2. Compilación en modo release

Una vez terminada la aplicación, ésta debe compilarse en modo release (final). Este modo, a diferencia del modo debug (depuración), no firma automáticamente la aplicación. Esto permite firmar la aplicación con la clave privada de la cuenta de desarrollador. La aplicación firmada con esta clave podrá, a continuación, ser publicada y distribuida a terceros.

La firma de una aplicación final se realiza mediante un certificado digital auto-firmado cuya clave privada pertenece únicamente al desarrollador de la aplicación. Las etapas de creación de esta clave privada y del almacén de claves que las contiene hay que hacerlas sólo la primera vez. Las compilaciones siguientes podrán, en lo sucesivo, utilizar la clave creada anteriormente.

Es posible utilizar una clave nueva para cada aplicación, incluso para cada versión. No obstante, se recomienda utilizar la misma, especialmente en el caso de actualizar una aplicación, con el fin de:

- Permitir una actualización rápida y sencilla de la aplicación a los usuarios existentes. Una actualización de una misma aplicación requiere obligatoriamente el uso de una misma clave, sin lo cual la actualización se considerará como una nueva aplicación completa, sin vínculo alguno con la versión anterior. Y, dado que esta nueva aplicación utiliza el mismo nombre de paquete, el sistema rechazará su instalación.
- Poder ejecutarlas en el mismo proceso de sistema si lo desean y compartir los mismos datos propios de la aplicación como archivos de preferencias. Esta funcionalidad permite, por

ejemplo, instalar por separado la aplicación principal, gratuita, y los módulos suplementarios independientes, de pago.

La clave utilizada deberá, por tanto, tener una fecha de validez lo suficientemente extensa como para permitir cubrir todas las aplicaciones y todas las duraciones de sus actualizaciones. La validez de esta fecha se verificará únicamente en la fase de instalación de la aplicación sobre el sistema Android.

El desarrollador debe proteger y conservar el almacén y la clave secreta, así como las contraseñas asociadas, bajo el riesgo de que le roben su identidad. Sin ello, la seguridad de las aplicaciones y de los datos de usuario asociados no estará garantizada.

a. Protección del código

Proguard es una herramienta que permite, entre otros, reducir, optimizar y ofuscar el código Java compilado, sin modificar su funcionamiento. Esto permite, a la vez, optimizar el tamaño del archivo binario y hacer que la aplicación sea más complicada de estudiar.

Para ello, Proguard toma en cuenta únicamente el código Java utilizado y deja de lado el código no utilizado. Renombra las clases, los métodos y los campos utilizando nombres cortos sin significado alguno. Por último, optimiza el archivo binario generado.

De este modo, resulta muy difícil comprender el código Java de un programa binario producido mediante la herramienta Proguard que se haya podido descompilar.

El uso de la herramienta Proguard es opcional. Está indicada para aquellos desarrolladores que quieran proteger el código de sus aplicaciones y se recomienda encarecidamente cuando la aplicación utiliza la librería de verificación de licencias para reforzar la seguridad de la clave del dispositivo.

El sistema de compilación de Android gestiona el uso de la herramienta de forma automática y transparente al desarrollador. La herramienta se utiliza solamente para las compilaciones en modo release.

Por defecto, un proyecto Android no utiliza la herramienta Proguard. Para activar el uso de esta herramienta, hay que agregar la propiedad `proguard.config` en el archivo`default.properties` que se encuentra en la raíz del proyecto. Esta propiedad debe especificar la ruta y el nombre del archivo de configuración destinado a la herramienta Proguard. Existe un archivo de configuración por defecto, `proguard.cfg`, que se provee en la raíz del proyecto y es apropiado para la mayoría de los proyectos.

Sintaxis

```
proguard.config=/carpetas/proguard.cfg
```

Ejemplo

```
proguard.config=proguard.cfg
```

El ejemplo utiliza el archivo `proguard.cfg` que se provee por defecto.

Una vez ejecutada, la herramienta crea varios archivos en la carpeta `proguard` del proyecto. Esos archivos son `dump.txt`, `mapping.txt`, `seeds.txt` y `usage.txt`. Permiten, en particular, convertir las trazas de excepción ofuscadas con los nombres de las clases, de los métodos y de los miembros originales.

Las compilaciones siguientes borrarán estos archivos. Se recomienda, por tanto, salvaguardarlos, así como el proyecto completo, para cada aplicación publicada.

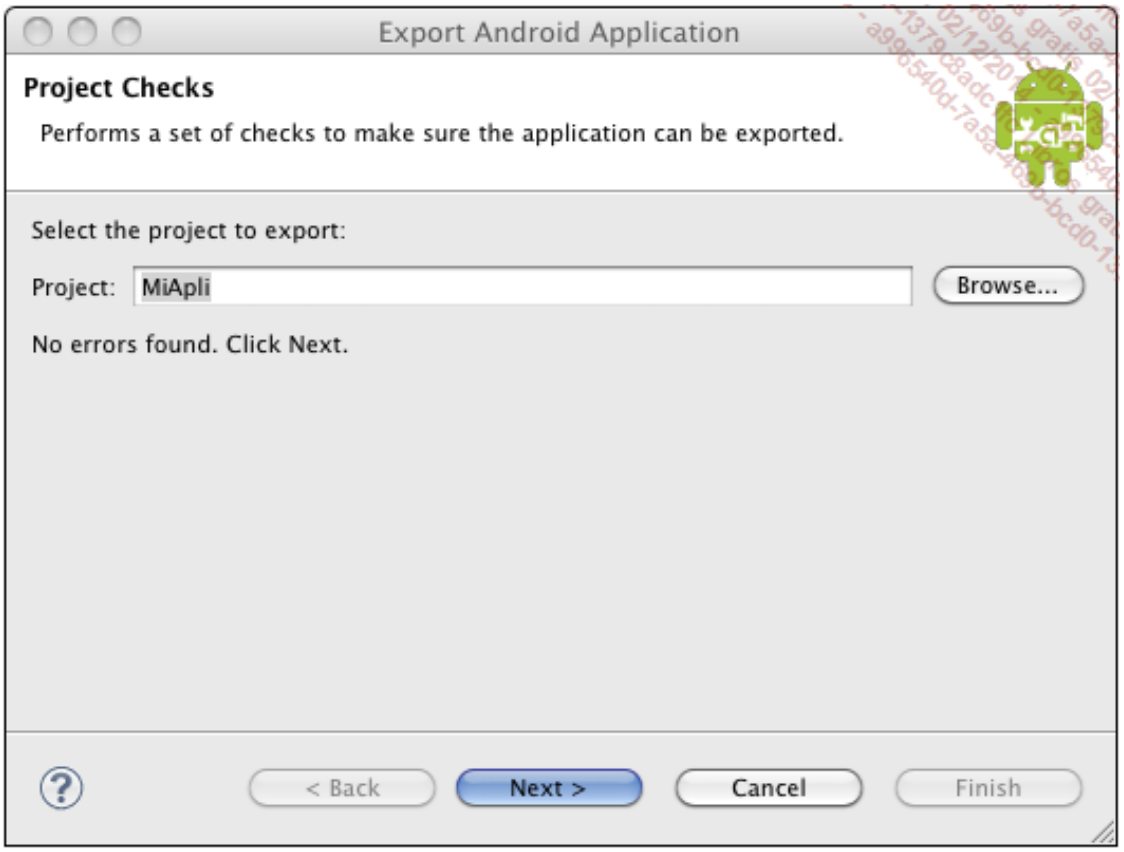
Es posible que sea necesario parametrizar Proguard para que no modifique los nombres de los métodos especificados en los atributos `android:onClick` de los widgets con el objetivo de mantener funcionales estas relaciones.

b. Firmar la aplicación

He aquí cómo compilar y firmar una aplicación en modo release en Eclipse:

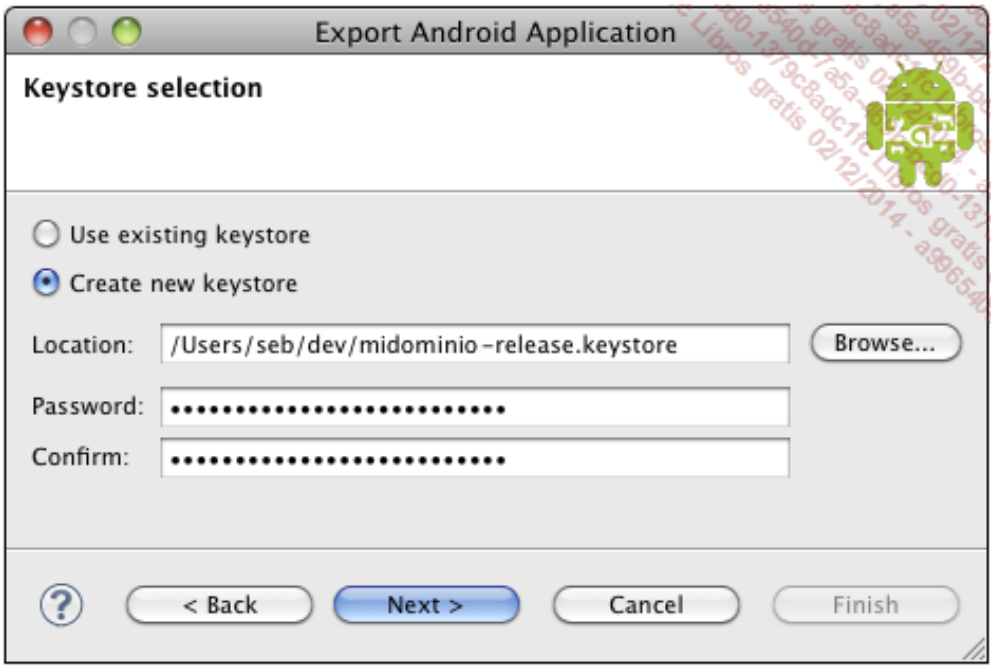
- En la vista **Package Explorer** de Eclipse, haga clic con el botón derecho sobre el proyecto y, a continuación, haga clic en **Android Tools - Export Signed Application Package...**.

Aparece la ventana **Export Android Application**.



- Verifique que el proyecto está presente en el campo **Project**. Si no fuera el caso, haga clic en el botón **Browse...** para seleccionar el proyecto.
- Haga clic en el botón **Next >** y seleccione **Create new keystore** para solicitar la creación de un nuevo almacén de claves. Haga clic sobre el botón **Browse...** para especificar la ubicación y el nombre del archivo de almacén que se creará. Introduzca una contraseña con seis caracteres como mínimo en el campo **Password** y confírmela introduciéndola de nuevo en el campo**Confirm**.

Esta contraseña debe memorizarse bien pues se le solicitará con cada compilación en modo release. Se recomienda encarecidamente escoger una contraseña segura para proteger el almacén.



- Haga clic en el botón **Next**.

La etapa siguiente consiste en crear la clave privada.

- Informe la clave privada en el campo **Alias**. Indique una contraseña de seis caracteres como mínimo en el campo **Password** y confírmela introduciéndola de nuevo en el campo **Confirm**.

Esta contraseña debe memorizarse bien pues se le solicitará con cada compilación en modo release. Por motivos de seguridad evidentes, se recomienda utilizar una contraseña distinta a la del almacén de claves, y que a su vez sea segura.

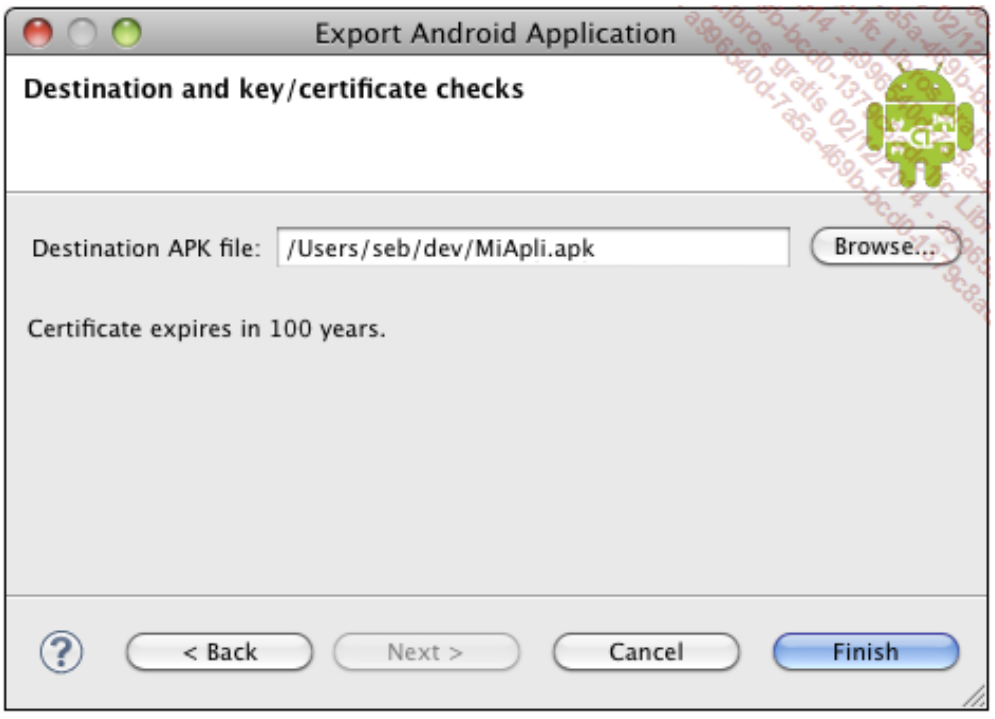
- Indique a continuación el número de años durante los cuales será válida esta clave, entre uno y mil años.

La publicación en Play Store requiere obligatoriamente una fecha de fin de validez superior al 22 de octubre de 2033. Puesto que no cuesta nada, no dude en especificar una fecha lo suficientemente lejana.

- Finalice la etapa especificando la identidad del creador de la clave.

Esta información tiene un carácter meramente informativo limitado a la clave. No se utiliza posteriormente, por ejemplo en Play Store.

- Haga clic sobre el botón **Next**.
- Introduzca la ubicación y el nombre del archivo `.apk` que quiere crear y firmar con esta clave privada.



- Haga clic en el botón **Finish**.

La clave se genera y almacena en el nuevo almacén. El proyecto se compila en modo release. El archivo `.apk` se crea, a continuación, en la ubicación indicada anteriormente, optimizado y firmado con la clave privada.

c. Instalar la aplicación

Antes de distribuir el archivo `.apk` del modo release, se recomienda encarecidamente instalarlo y ejecutarlo sobre el emulador o sobre un dispositivo Android para verificar que el archivo no se ha corrompido y que la aplicación funciona correctamente.

Para ello, es necesario suprimir previamente la aplicación compilada en modo debug del sistema Android puesto que las claves utilizadas por los archivos `.apk`, debug y release no son idénticas. El sistema Android lo detectará y denegará la instalación del nuevo archivo apk.

- Desinstale la aplicación compilada en modo debug del sistema Android. Bien utilizando el menú del sistema Android, o bien utilizando la herramienta `adb` por línea de comandos especificándole el paquete que desea borrar.

Sintaxis

```
adb uninstall [opciones] paquete
```

La opción `-k` indica que no se borren los datos de la aplicación y la caché de la aplicación.

Ejemplo

```
$ adb uninstall es.midominio.android.miaplicacion
Success
```

El sistema Android no contiene ninguna traza de la aplicación, y podemos instalar la aplicación final.

Para ello, utilizaremos de nuevo la herramienta `adb` por línea de comandos especificándole el archivo `apk` a instalar.

Sintaxis

```
adb install [opciones] archivo.apk
```

La opción `-r` es útil para realizar una actualización. Permite conservar los datos ya instalados por la versión anterior.

La opción `-s` permite instalar la aplicación sobre el almacenamiento externo del dispositivo en lugar de sobre su almacenamiento interno.

Ejemplo

```
$ adb install /Users/seb/dev/MiAplicacion.apk
475 KB/s (13404 bytes in 0.027s)
    pkg: /data/local/tmp/MiAplicacion.apk
Success
```

Publicación de la aplicación en Play Store

Una vez compilada la aplicación final, firmada con la clave privada de la cuenta del desarrollador, y probada en un entorno real, puede ser publicada.

Existen numerosos lugares de venta donde puede publicarse la aplicación. La que contiene más aplicaciones es Play Store, gestionada por la compañía Google. Describiremos a continuación las distintas etapas para publicar una aplicación en esta app'Store (tienda de aplicaciones).

1. Inscripción

Play Store es el lugar de venta de aplicaciones Android que provee la empresa Google. Este lugar de venta permite, por un lado, a los desarrolladores publicar sus aplicaciones y, por otro lado, a los usuarios descargarlas e instalarlas directamente sobre sus dispositivos Android. El usuario puede instalar estas aplicaciones bien utilizando la aplicación dedicada **Play Store** disponible en su dispositivo, o bien surfeando en la web de Play Store disponible en la dirección https://play.google.com/store

La apertura de una cuenta de desarrollador en Play Store es de pago. Aquel desarrollador que quiera publicar aplicaciones gratuitas o de pago debe abonar la tasa de $25 US una única vez durante su inscripción. Dicha inscripción es válida de por vida, y para todas las aplicaciones que el desarrollador quiera publicar.

El desarrollador que quiera publicar una aplicación en Play Store debe, en primer lugar, tener una cuenta de Google.

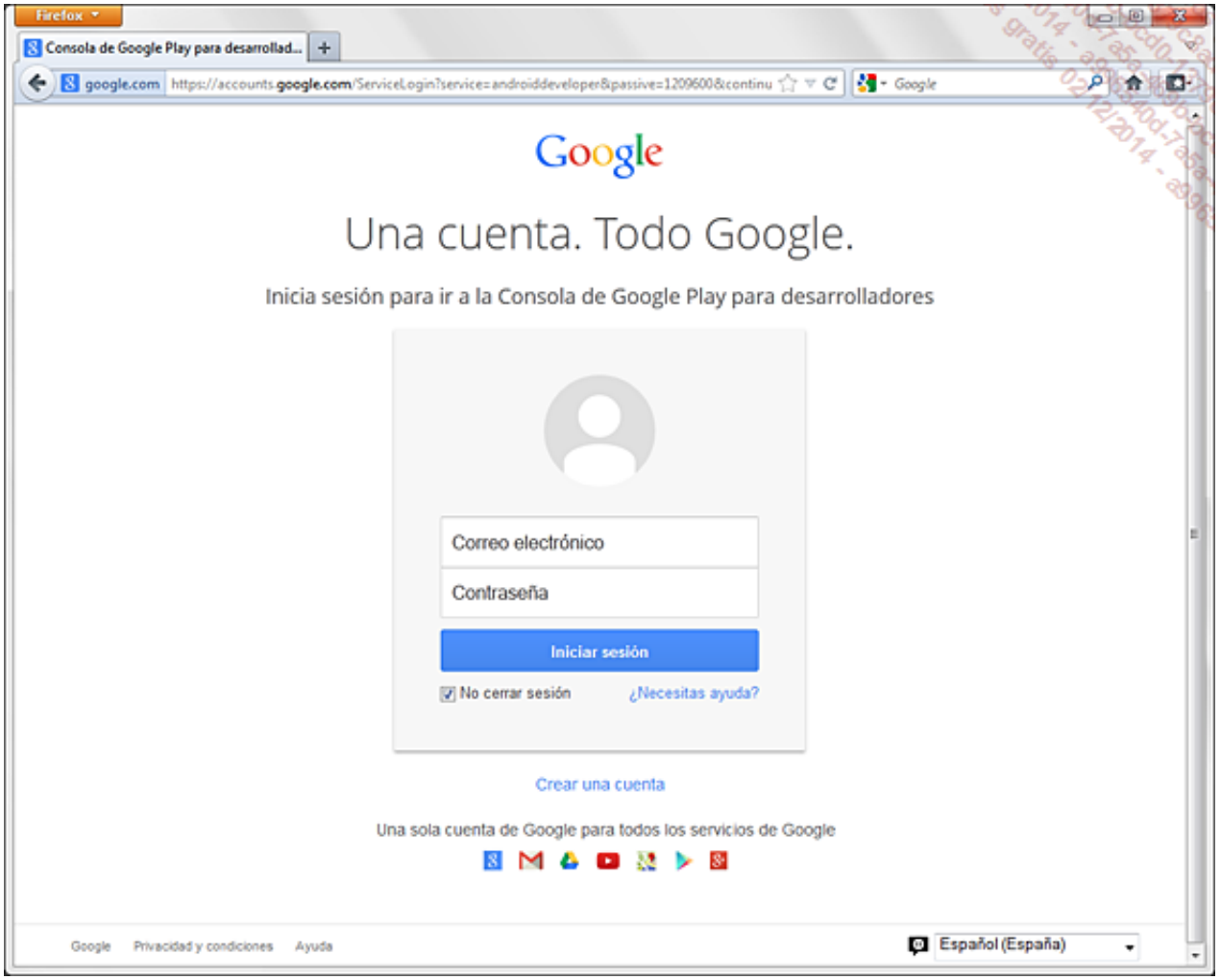
La cuenta de Google puede ser una cuenta de Gmail o una cuenta de Google Apps.

Esta cuenta permite crear una cuenta de desarrollador utilizada para gestionar la publicación de las aplicaciones. La dirección de correo electrónico así como la demás información de la cuenta de Google no se divulgarán a los usuarios de las aplicaciones. Tras cada publicación de la aplicación, es posible especificar una dirección de correo electrónico específica que se comunicará a los usuarios para que puedan ponerse en contacto con el desarrollador.

- Abra un navegador de Internet e introduzca la siguiente dirección: http://play.google.com/apps/publish

Se producirá una redirección sobre la página de autenticación de Google.

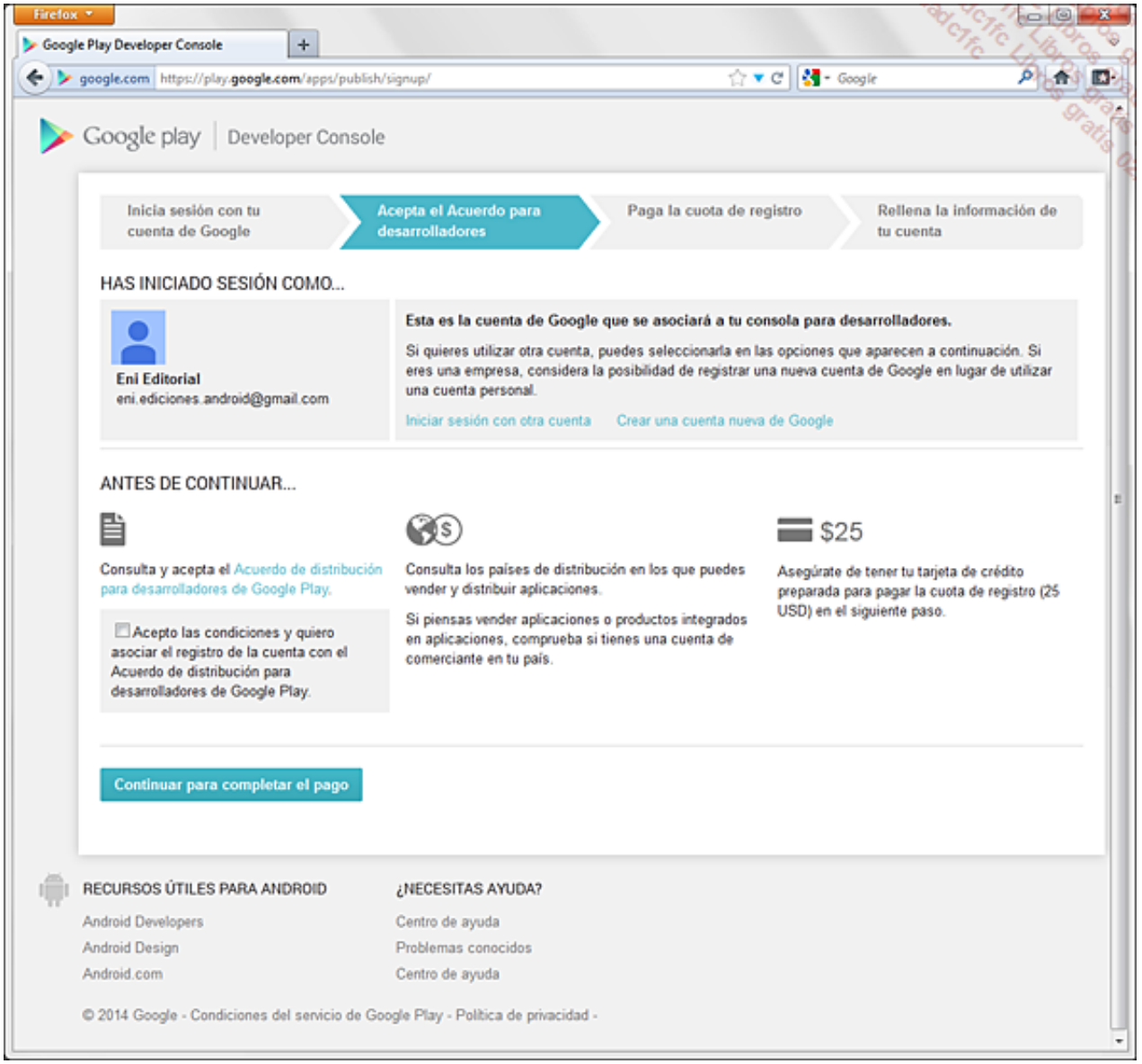
- Introduzca la dirección de correo electrónico y la contraseña de la cuenta Google que se utilizará para crear y administrar el acceso a Play Store. Si fuera necesario, puede crear una cuenta de Google haciendo clic en **Crear una cuenta**.



- Haga clic en el botón **Iniciar sesión**.

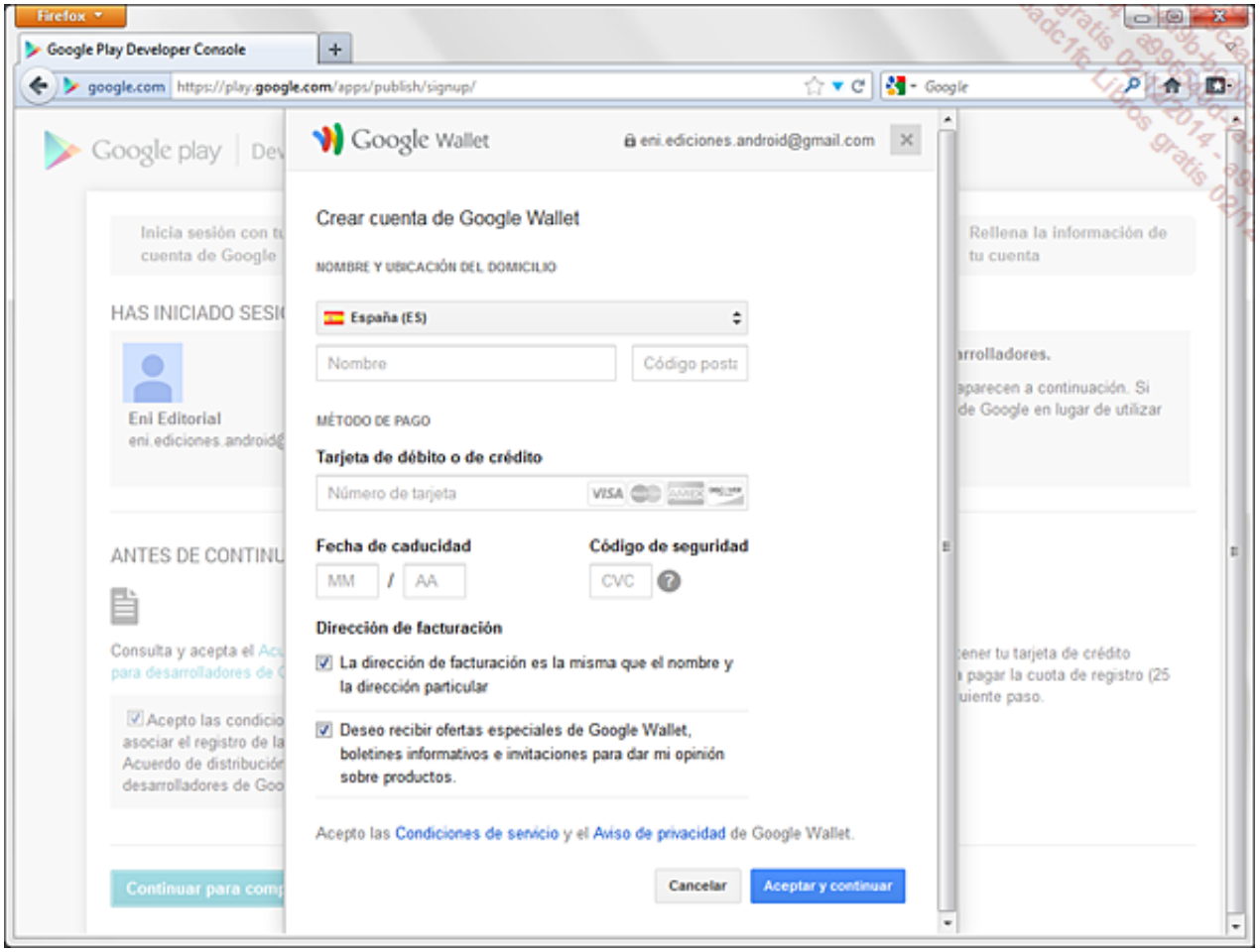
Primera etapa de creación de la cuenta de desarrollador en Play Store: leer el Contrato relativo a la distribución en Google Play y aceptarlo.

- Marque la opción **Acepto las condiciones y quiero...**.
- Haga clic en el botón **Continuar para completar el pago**.



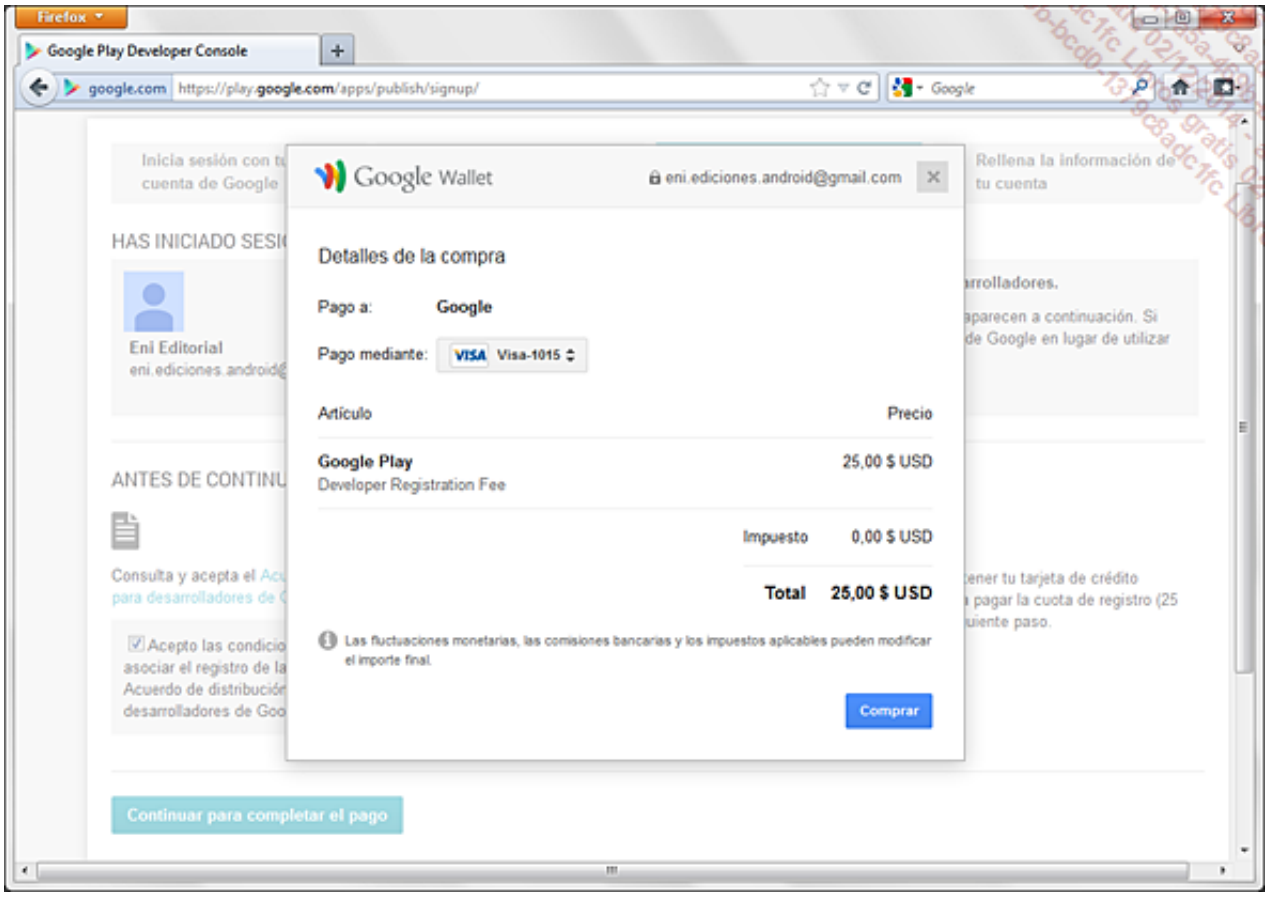
La siguiente etapa consiste en crear una cuenta en Google Wallet.

- Complete los datos del formulario: nombre y dirección, información acerca del método de pago, etc.
- Haga clic en el botón **Aceptar y continuar**.



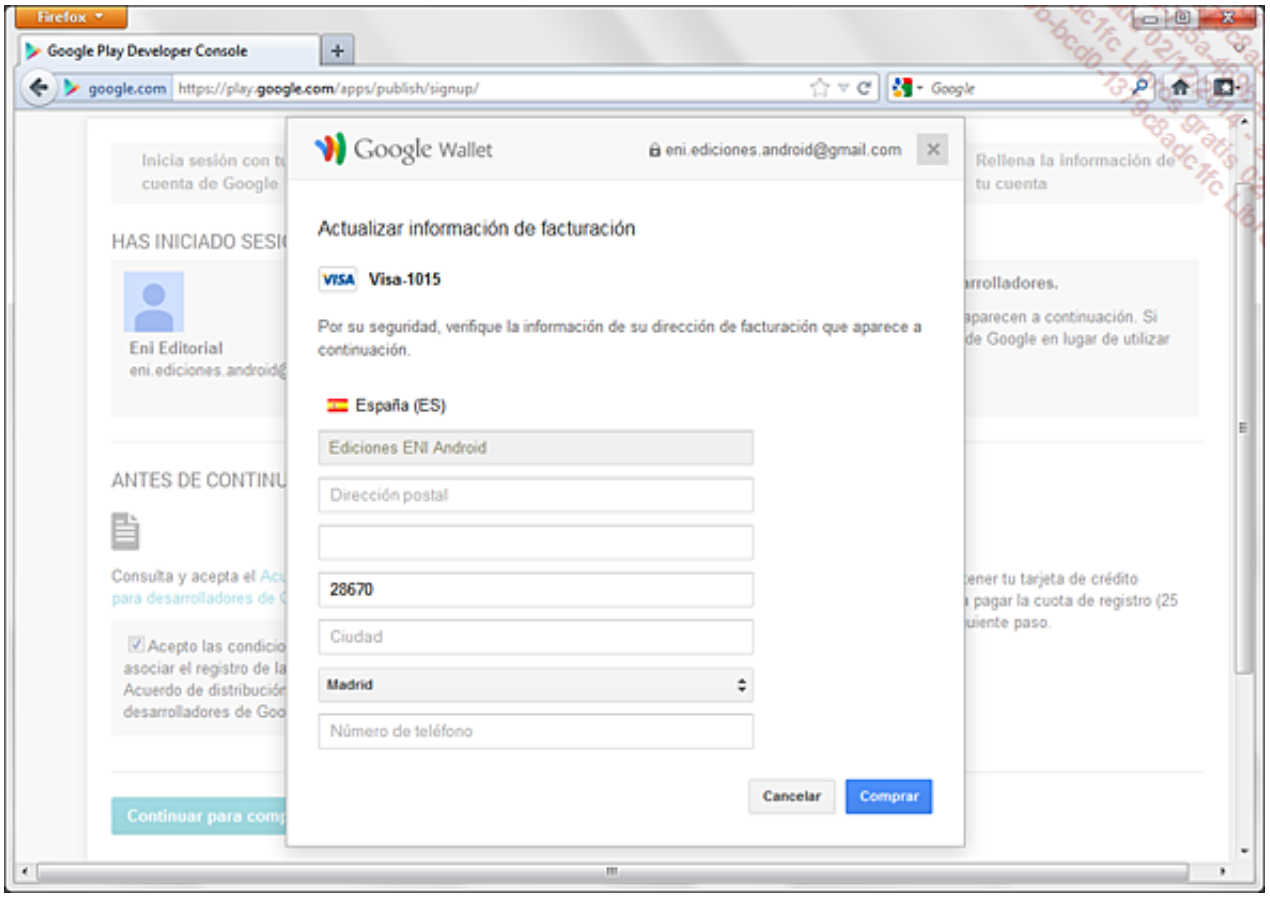
A continuación, aparece una ventana que le muestra la factura de la suscripción. El coste de la suscripción, recordamos, es de 25 $ USD y se paga una única vez.

- Compruebe la información y haga clic en **Comprar**.



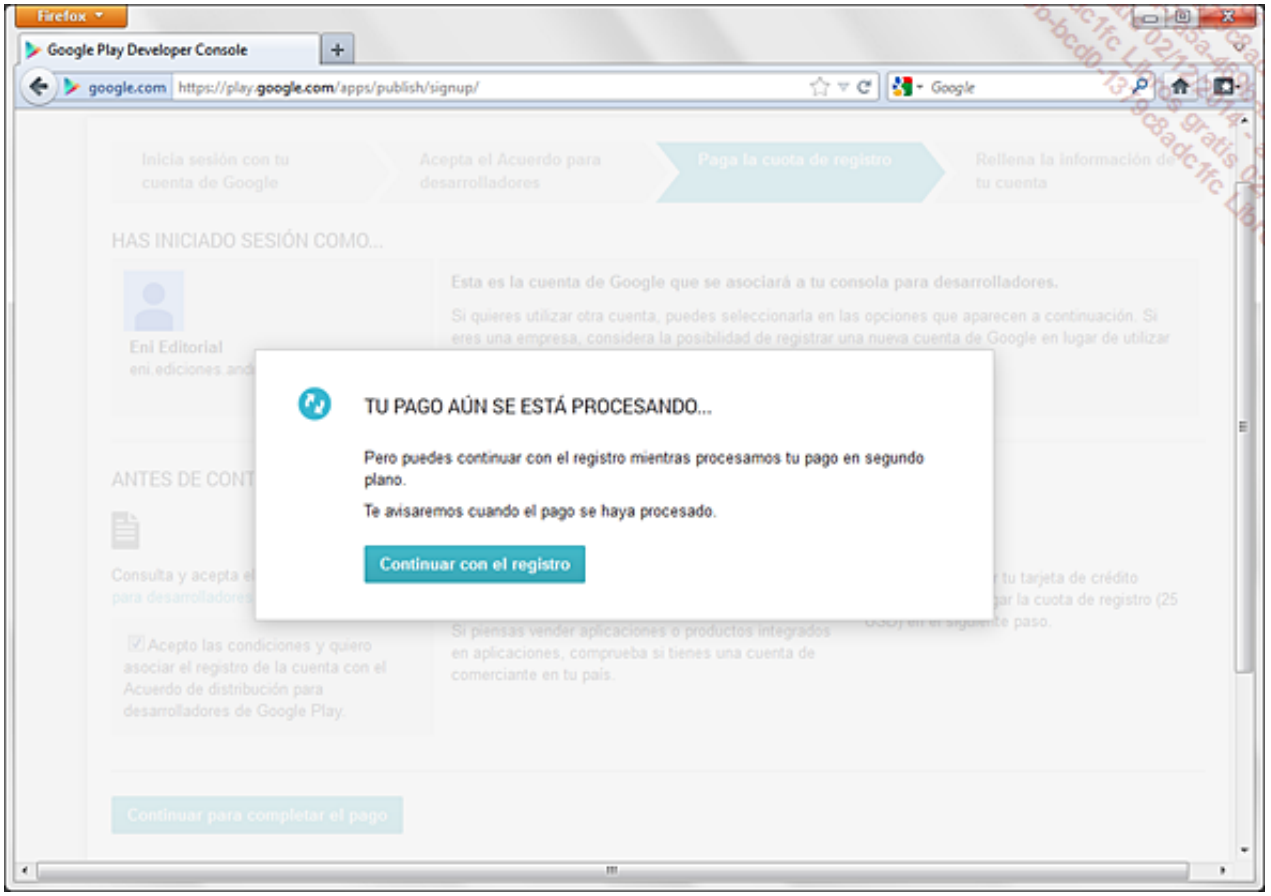
Google le pide, a continuación, información de facturación: dirección postal y número de teléfono.

- Rellene el formulario y haga clic en **Comprar**.



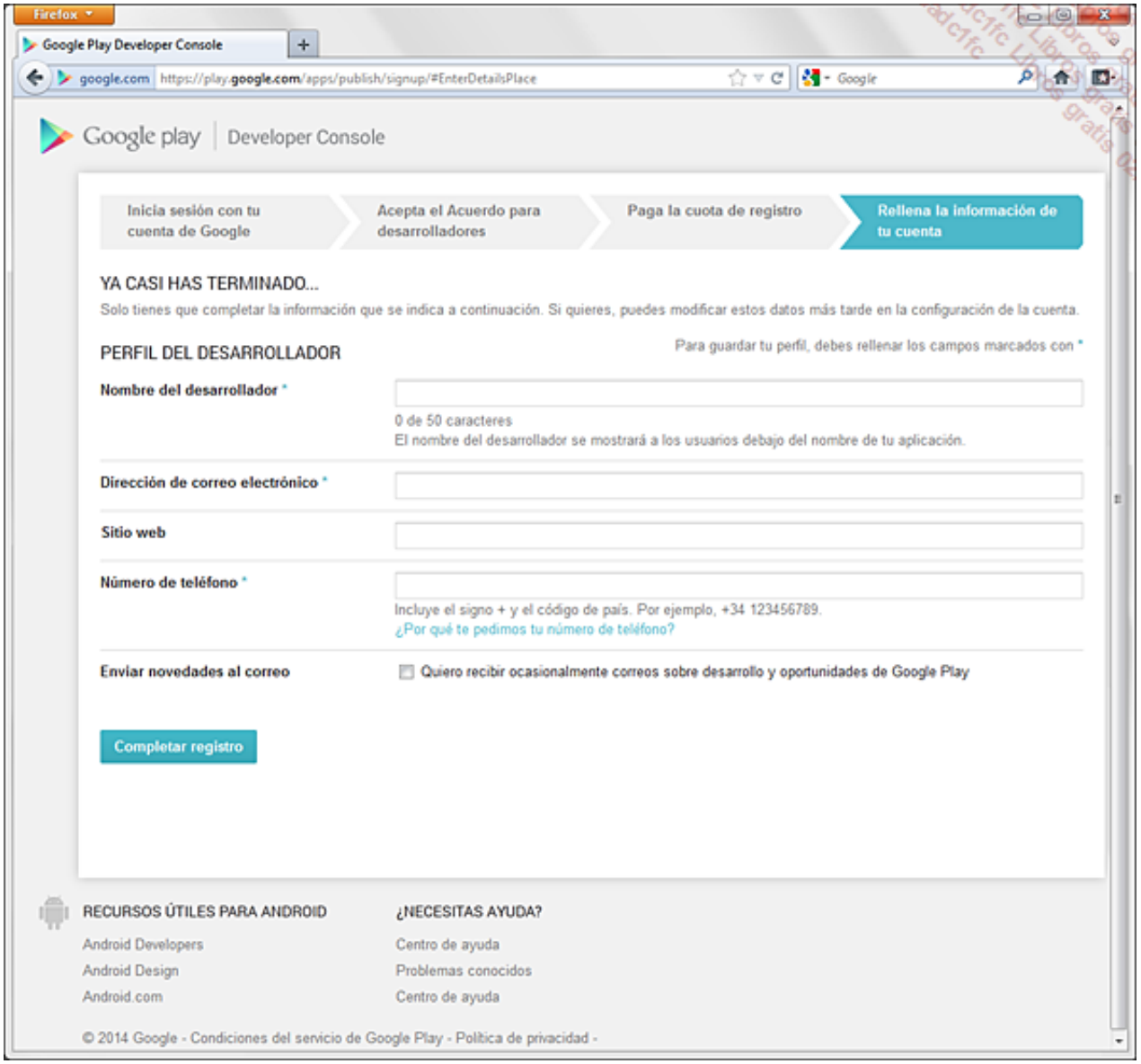
A continuación, Google le informa de que el pago está en curso de verificación, y que el proceso puede continuar en paralelo.

- Haga clic en **Continuar con el registro**.

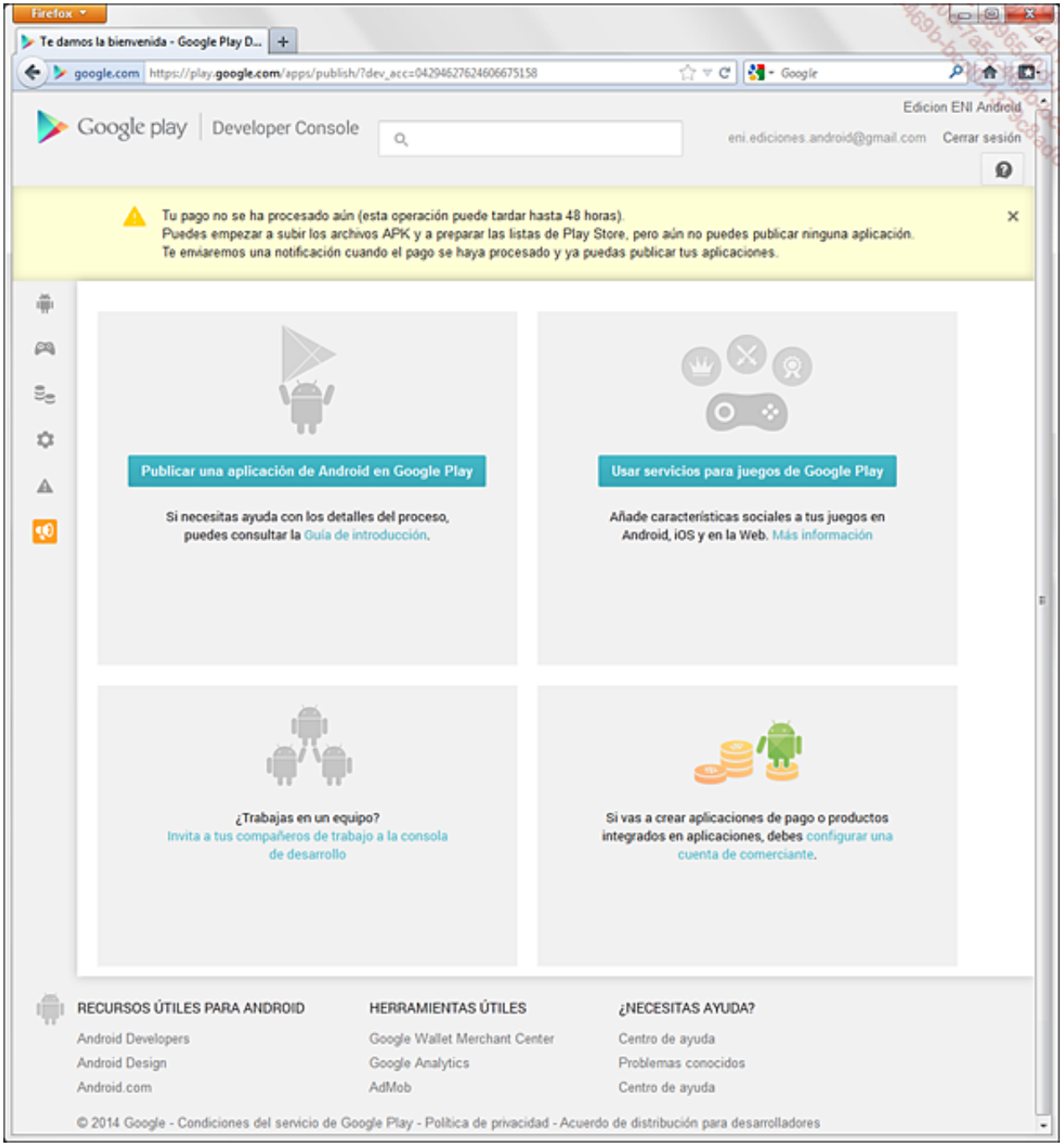


La última etapa consiste en introducir la información relativa a la cuenta de desarrollador: nombre (el que se mostrará como desarrollador en Play Store para las aplicaciones que publicará), número de teléfono y dirección de correo electrónico, como mínimo.

- Haga clic en **Completar registro**.



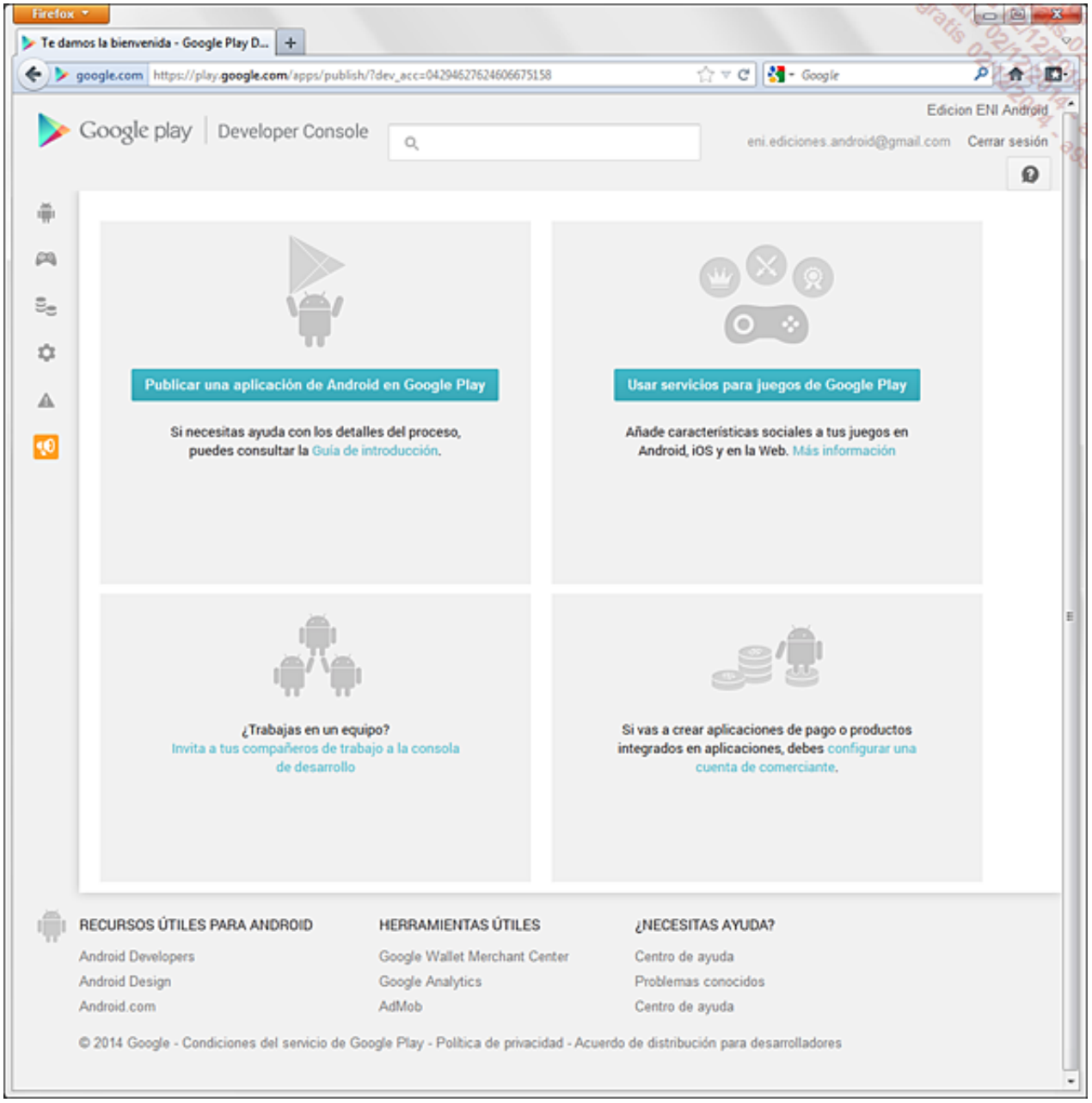
La inscripción ha terminado: el pago puede que no se haya hecho efectivo, necesariamente, en el momento, tal y como indica el mensaje que muestra la página de inicio de la cuenta de desarrollador.



En este momento puede importar archivos APK y preparar la publicación de su primera aplicación, aunque no podrá publicarla hasta que no se haya hecho efectivo el pago.

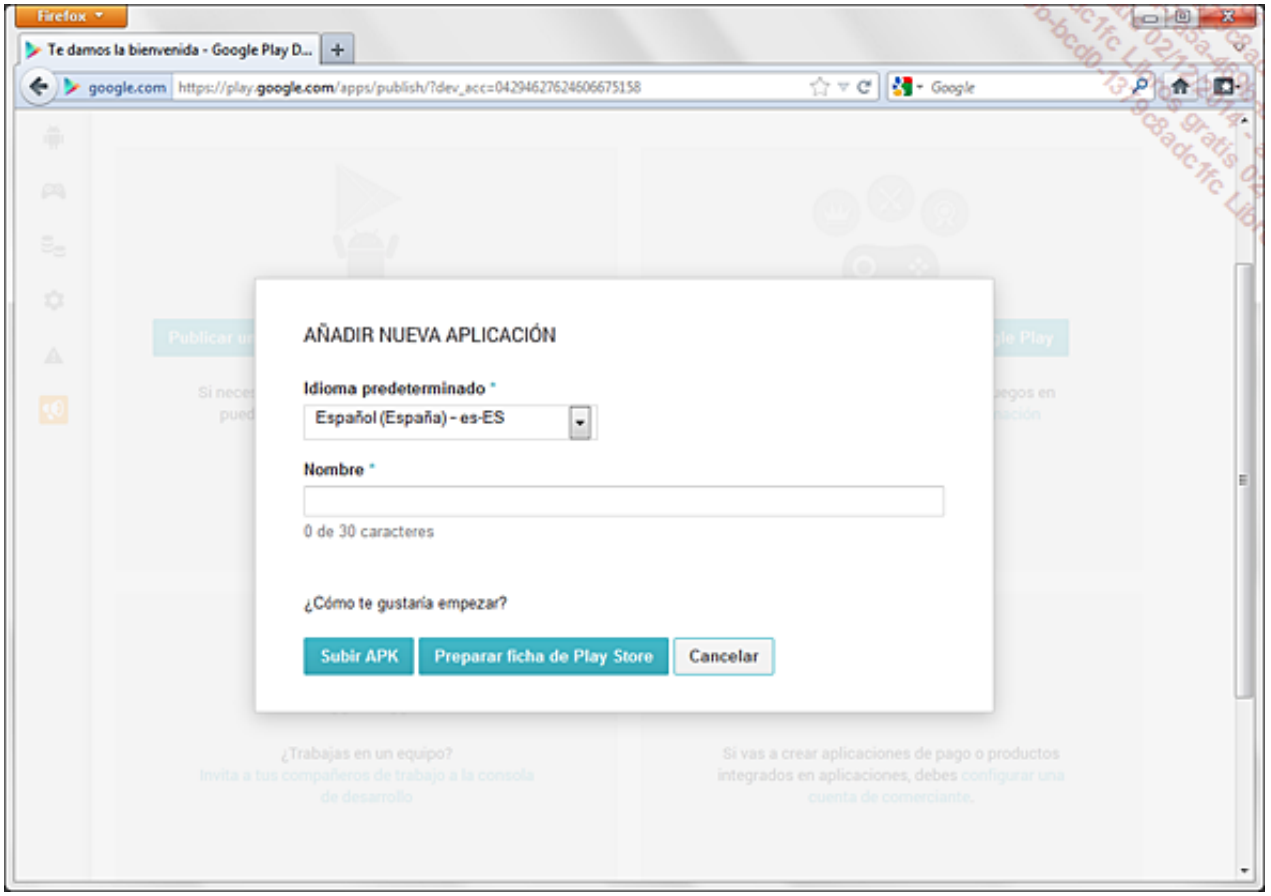
2. Publicación

- Si todavía no lo ha hecho, conéctese a su cuenta de desarrollador de Play Store.



- Haga clic en el botón **Publicar una aplicación**.
- Se abre un pop-up que le pide la primera información: idioma por defecto, título de la aplicación. A continuación, puede comenzar a completar la ficha de Play Store de su aplicación, y subir el archivo APK de la aplicación.

Es necesario proveer numerosa información relativa a la aplicación antes de poder publicarla. Es posible completar únicamente parte de la información, sin publicar la aplicación: por defecto, la aplicación se queda en modo borrador, y está visible únicamente para el desarrollador.



Una vez validado el formulario y publicada la aplicación, es posible modificar la mayoría de los elementos del formulario. Es posible que exista cierto retardo a la hora de propagar las modificaciones en Play Store.

Veamos, a continuación, la información solicitada.

Para cada uno de los archivos que se proveen en el formulario es preciso, una vez seleccionado el archivo, hacer clic en el botón **Importar** correspondiente para cargarlo en el servidor.

a. Archivo .apk

Archivo .apk de la versión preliminar de la aplicación: archivo binario con formato apk de la aplicación compilada en modo release.

Recuerde, el nombre del paquete utilizado por la aplicación es el identificador único de la aplicación en Play Store. Una vez cargado, este nombre no puede modificarse ni reutilizarse para otra aplicación. Play Store verifica, tras la primera importación del APK, que el nombre del paquete no esté ya referenciado.

b. Ficha en Google Play Store

Esta sección permite describir la aplicación en distintos idiomas, proveer elementos gráficos para la ficha de Play Store, clasificarla e indicar, eventualmente, un precio. Todos estos elementos pueden modificarse posteriormente.

Idioma: por defecto, las reseñas de la aplicación deben proveerse, como mínimo, en el idioma

indicado como idioma por defecto durante la creación. Haciendo clic sobre **agregar idioma** es posible agregar otras lenguas complementarias.

De forma similar a la traducción de la aplicación, se recomienda encarecidamente traducir la descripción de la aplicación en la mayoría de idiomas posibles; como mínimo en las mismas lenguas en las que esté soportada la aplicación.

Título: nombre de la aplicación utilizado en Play Store. Dos aplicaciones distintas no pueden tener el mismo nombre. Es la ley del primero en llegar la que se lleva el nombre. Por el contrario, es posible especificar un nombre distinto en cada idioma.

Descripción: texto que describe la aplicación. El usuario podrá leerlo antes de descargar la aplicación. Es preciso, por tanto, tratar de hacerla atractiva aquí. Es indispensable proveer una descripción por cada idioma.

Cambios recientes: texto que describe los cambios que aporta la última versión de la aplicación tales como nuevas funcionalidades, correcciones de bugs…

Texto promocional: texto que figura junto a los elementos gráficos promocionales **Gráfico promocional** vistos anteriormente.

Cuando el usuario realiza una búsqueda de texto en Play Store, éste utiliza los datos de los campos título y descripción para componer la lista de aplicaciones correspondientes. Es preciso, por tanto, pensar bien los términos empleados.

Elementos gráficos: los elementos gráficos forman parte de los factores determinantes cuando un usuario decide instalar una aplicación. Es, por tanto, primordial seleccionarlos cuidadosamente para que reflejen, lo mejor posible, el trabajo que ha realizado. Se le solicitarán capturas de pantalla, para diferentes tamaños de pantalla. Deben incluirse, como mínimo, dos capturas de pantalla. No es necesario proveer imágenes para cada tamaño de pantalla, si bien es recomendable. También debe prever un icono en alta resolución, que se utilizará como elemento visual en la ficha de Play Store. Esta imagen debe tener obligatoriamente formato PNG (32 bits, con canal alfa) y un tamaño de 512 x 512 píxeles.

Tipo de aplicación y **Categoría**: toda aplicación debe clasificarse en una categoría. Esto permite al usuario poder consultar las aplicaciones disponibles en cada una de las categorías.

Clasificación del contenido: la aplicación debe clasificarse según su contenido y su uso. Existen cuatro niveles. Se muestran a continuación, en orden del más restrictivo al más permisivo:

- `Nivel 1 - Estricto` (Nivel de madurez alto).
- `Nivel 2 - Moderado` (Nivel de madurez medio).
- `Nivel 3 - Amplio` (Nivel de madurez bajo).
- `Todos` (Para todos los públicos).

El nivel debe escogerse teniendo en cuenta las referencias al alcohol, al tabaco, a las drogas, a los juegos de azar y apuestas, a la incitación al odio...

Es responsabilidad del desarrollador escoger la categoría correspondiente. Por ejemplo, una aplicación que utilice los datos de localización del usuario requerirá un nivel superior o igual al nivel 3. Para más información acerca de los criterios y de las clasificaciones correspondientes, diríjase a la siguiente dirección https://support.google.com/googleplay/android-developer/answer/188189

c. Tarifas y disponibilidad

País/Tarifas: la aplicación puede publicarse de forma gratuita o de pago. El desarrollador puede seleccionar los países en los que se distribuirá la aplicación. En el caso de una aplicación de pago, y en la medida de lo posible, el precio de la aplicación estará indicado en la divisa local del usuario. El desarrollador puede bien fijar libremente la tarifa local de la aplicación, o bien utilizar el botón**Conversión automática** para dejar que Play Store realice las conversiones automáticamente según el cambio de divisa actual. Esta conversión automática se produce con un simple clic en este botón. Una vez definida, cada tarifa se fija hasta la próxima modificación por parte del desarrollador.

Si la aplicación se publica de forma gratuita, jamás podrá ser de pago en un futuro. Hacer de pago una aplicación gratuita supone volver a publicarla con un nombre diferente, con un nombre de paquete también diferente. Conviene, por tanto, definir previamente la estrategia comercial que se quiere utilizar.

En el momento de fijar la tarifa, el desarrollador debe tener en cuenta que Play Store deduce un 30% del coste sobre cada una de las transacciones. De este modo, para una aplicación con una tarifa de 10 euros, el desarrollador recibirá 7 euros brutos de los que tendrá que descontar a continuación eventuales impuestos y tasas hasta obtener la ganancia neta.

Dispositivos compatibles: muestra las funcionalidades requeridas por la aplicación y especificadas en su manifiesto, como se ha visto al principio de este capítulo. Se muestra el número de dispositivos que disponen de estas funcionalidades. Haciendo clic sobre el enlace **Mostrar los dispositivos** permite mostrar la lista de dispositivos compatibles. Si fuera necesario, es posible excluir alguno de forma manual. La aplicación se publicará únicamente para aquellos dispositivos que formen parte de la lista final.

d. Coordenadas

Esta parte está relacionada con la información de contacto que podrán utilizar los usuarios de Play Store para ponerse en contacto con el programador. Es información pública. Es preciso proveer al menos un tipo de contacto de entre los siguientes:

Sitio Web: dirección del sitio de Internet del desarrollador o, mejor todavía, de la aplicación, si existe.

Dirección e-mail: dirección de correo electrónico en la que los usuarios pueden contactar con el desarrollador.

Teléfono: número de teléfono en el que los usuarios pueden contactar con el desarrollador.

e. Aceptar

Los términos del *Reglamento del programa Play Store* deben aceptarse. Este reglamento define, entre otros, la política de contenidos que debe respetar la aplicación. Este reglamento está disponible en castellano en la dirección https://play.google.com/intl/ALL_es/about/developer-content-policy.html.

Por otro lado, es necesario confirmar que la aplicación respeta las leyes de exportación americanas, en especial si la aplicación hace uso directamente o no de la encriptación.

Una vez completado el formulario, haga clic en el botón **Publicar** para publicar la aplicación en Play Store o en el botón **Guardar** para guardar el formulario en el estado de borrador y retomarlo posteriormente. La publicación toma, por regla general, entre 2 y 4 horas.

La publicación de la aplicación en Play Store es automática. No existe un control previo de la aplicación. No obstante, es posible que exista un control a posteriori en cualquier momento, llevado a cabo por la empresa Google, que le obligue a retirar la aplicación de Play Store, lo cual

ocurre rara vez y sólo afecta a aplicaciones conflictivas...

3. ¿Y después?

Tras la publicación, la aplicación se agrega a la lista de aplicaciones de la cuenta del desarrollador. En lo sucesivo, puede gestionar cada una de sus aplicaciones. Para ello, basta con hacer clic en el nombre de la aplicación para mostrar su página de publicación. Es posible modificar todos los campos, excepto el archivo apk.

Para actualizar el binario de la aplicación, es decir el archivo `.apk`, hay que hacer clic en el enlace**Importar la actualización** que figura en la página de publicación de la aplicación y cargar el nuevo archivo binario.

Recuerde, el atributo `android:versionCode` del manifiesto debe tener un valor superior al del binario publicado. Y, evidentemente, el nombre del paquete de la aplicación no deben haber cambiado.

El botón **Anular la publicación** permite retirar la aplicación de la publicación y, por tanto, de Play Store. La aplicación y su ficha de publicación se borrarán de la cuenta del desarrollador de forma que pueda publicarlas de nuevo.

Para cada aplicación de la lista figura su nombre, su versión, su categoría, su nota media reseñada por los usuarios, el número total de descargas y el número de instalaciones todavía efectivas, el precio y el estado de la publicación.

Es posible consultar más información haciendo clic en el enlace correspondiente:

- **Comentarios**: enlace que permite consultar los detalles de las notas asignadas por los usuarios y sus comentarios.
- **Estadísticas**: enlace que permite consultar diversas estadísticas correspondientes a las instalaciones de la aplicación. El primer gráfico representa el histórico del número de instalaciones acumuladas desde la fecha de publicación de la aplicación. Los siguientes gráficos indican los repartos de las instalaciones según:
  - La versión de los sistemas Android.
  - El modelo del dispositivo.
  - El país del usuario.
  - El idioma del sistema Android y, por tanto, del usuario.

    De forma complementaria figuran, a su vez, los datos correspondientes al conjunto de las aplicaciones de Play Store repartidas según los mismos criterios, salvo por el modelo del dispositivo dado que se trata de un dato sensible.
- **Productos integrados con la aplicación**: enlace que permite gestionar las ventas integradas con la aplicación.
- **Errores**: enlace que permite consultar los informes de bloqueos o de crash de la aplicación reenviados por los usuarios. Este enlace se muestra solamente si existe al menos un informe. Cada uno de estos informes proporciona el nombre de la excepción elevada, el método en el que se ha producido la excepción, el modelo del dispositivo y la pila de llamadas a los métodos que han provocado las excepciones. El usuario puede, también, incluir un mensaje en el informe. Los informes idénticos se agrupan en uno solo.

Introducción

Los sensores son, en efecto, una de las funcionalidades más novedosas de dispositivos portátiles tales como smartphones y tabletas. La mayoría de dispositivos Android integran varios sensores, ofreciendo una mejora significativa en la experiencia de usuario: geolocalización, reacción a la inclinación, detección de la cantidad de luz en el entorno, medida de la temperatura ambiente o incluso medir el número de pasos que ha realizado el usuario.

La dificultad para el desarrollador, si es que la hay, consiste en ubicarse entre la multitud de configuraciones de hardware distintas: no todos los dispositivos Android disponen de los mismos sensores.

En este capítulo veremos cómo utilizar estos sensores y, a continuación, realizaremos un estudio detallado de los sensores de posición y de la geolocalización.

Fundamentos

El uso de sensores utiliza, principalmente, las siguientes clases del paquete android.hardware :

- La clase `SensorManager`: permite recuperar la lista de sensores disponibles en el dispositivo, y se encarga también de instanciar un objeto de tipo `Sensor` y vincularlo con un gestor de eventos.
- La clase `Sensor`, que representa a un sensor. Permite obtener los valores medidos por el sensor, así como sus características (nombre, precisión, frecuencia de muestreo, consumo eléctrico, etc.).
- Las clases `SensorEventListener` y `SensorEvent`, que permiten gestionar los eventos producidos por cada sensor.

Detectar un sensor

No todos los sensores soportados por la plataforma Android están, necesariamente, integrados en los dispositivos Android. Es preciso, por tanto, antes de utilizar un sensor, asegurarse de que está presente en el dispositivo donde se ejecuta la aplicación.

Existen dos estrategias diferentes para gestionar la disponibilidad de un sensor:

- Si el sensor es indispensable para el correcto funcionamiento de la aplicación, es preferible indicar en Google Play Store que la aplicación debe presentarse únicamente en aquellos dispositivos que dispongan de dicho sensor.
- Si el sensor es un extra, no esencial, es preciso detectar en tiempo de ejecución de la aplicación la disponibilidad del mismo.

El primer caso está resuelto, como con las demás restricciones de hardware, directamente en el archivo de manifiesto de la aplicación, utilizando la opción `<uses-feature>` (consulte el capítulo Publicar una aplicación, sección Preliminares - Filtros para el mercado, para una descripción completa).

Sintaxis

```
<uses-feature android:name="nombre_del_sensor" android:required="true" />
```

Ejemplo

```
<uses-feature android:name="android.hardware.sensor.light"
android:required="true"/>
```

En el segundo caso, la clase `SensorManager` permite verificar la presencia de un sensor en función de su tipo, bien enumerando los sensores disponibles en el dispositivo o bien instanciando un sensor de un tipo concreto y verificando el resultado de la instanciación.

Para recuperar un objeto de tipo `SensorManager` hay que invocar al método `getSystemService`de la clase `Context`, pasándole como parámetro la constante `SENSOR_SERVICE`.

Ejemplo

```
public class MainActivity extends Activity{

   @Override
   public void onCreate(Bundle savedState) {
      super.onCreate(savedState);

      SensorManager  sensorManager = (SensorManager)
                  this.getSystemService(SENSOR_SERVICE);
   }
}
```

La lista de sensores presentes en un dispositivo Android puede obtenerse invocando al método`getSensorList` del objeto `SensorManager`. Este método devuelve una lista de objetos de tipo`Sensor`.

Sintaxis

```
public List<Sensor> getSensorList (int type)
```

Ejemplo

```
List<Sensor> sensors =sensorManager.getSensorList
(Sensor.TYPE_ALL);
```

El parámetro de tipo entero `type` se corresponde con el tipo del sensor del que se desea recuperar una instancia. Los posibles tipos los devuelve como datos estáticos la clase `Sensor`.

Los tipos de sensores disponibles se muestran en la siguiente tabla:

TYPE_ACCELEROMETER	Acelerómetro
TYPE_ALL	Todos los tipos
TYPE_AMBIENT_TEMPERATURE	Temperatura ambiente
TYPE_GAME_ROTATION_VECTOR	Rotación (no calibrado)
TYPE_GEOMAGNETIC_ROTATION_VECTOR	Rotación geomagnética
TYPE_GRAVITY	Gravedad
TYPE_GYROSCOPE	Giroscopio
TYPE_GYROSCOPE_UNCALIBRATED	Giroscopio (no calibrado)
TYPE_LIGHT	Luz
TYPE_LINEAR_ACCELERATION	Aceleración lineal
TYPE_MAGNETIC_FIELD	Campo magnético
TYPE_MAGNETIC_FIELD_UNCALIBRATED	Campo magnético (no calibrado)
TYPE_ORIENTATION	Orientación (deprecado)
TYPE_PRESSURE	Presión
TYPE_PROXIMITY	Proximidad
TYPE_RELATIVE_HUMIDITY	Humedad relativa
TYPE_ROTATION_VECTOR	Rotación
TYPE_SIGNIFICANT_MOTION	Movimiento significativo
TYPE_STEP_COUNTER	Podómetro
TYPE_STEP_DETECTOR	Detector de gas
TYPE_TEMPERATURE	Temperatura (deprecado)

Como información, si bien la mayoría de los sensores que se enumeran en esta lista están directamente vinculados a sensores físicos (sensor de campo magnético, sensor de luz, etc.) otros son sensores llamados 'sensores lógicos' (por ejemplo, el sensor de podómetro): los datos se extrapolan a partir de otros sensores (o grupo de sensores). Esta diferencia no modifica en nada el uso de los sensores.

El método `getDefaultSensor` de la clase `SensorManager` permite, para cada tipo de sensor, obtener el sensor por defecto para un tipo de medida.

Sintaxis

```
public Sensor getDefaultSensor (int type)
```

Ejemplo

```
Sensor sensorLuz =
   sensorManager.getDefaultSensor(Sensor.TYPE_LIGHT);
```

Si el sensor que se pasa como parámetro no está presente en el dispositivo del usuario, el método`getDefaultSensor` devuelve un valor nulo: esto permite comprobar la presencia de un sensor antes de su uso.

Obtener los valores

La obtención de los valores medidos por los sensores se obtiene mediante un gestor de eventos específico, de tipo `SensorEventListener`. Esta interfaz presenta dos métodos,`onAccuracyChanged` y `onSensorChanged`.

Sintaxis

```
void onAccuracyChanged(Sensor sensor, int accuracy)
void onSensorChanged(SensorEvent event)
```

El método `onAccuracyChanged` se invoca cuando se modifica la precisión del sensor. El parámetro`accuracy` puede tomar los valores "alta" (`SensorManager.SENSOR_STATUS_ACCURACY_HIGH`), "media" (`SensorManager.SENSOR_STATUS_ACCURACY_MEDIUM`) o "baja" (`SensorManager.SENSOR_STATUS_ACCURACY_LOW`).

El método `onSensorChanged` se invoca con cada cambio del valor del sensor.

Cuando el sistema invoca al método `onSensorChanged`, se transmite un objeto del tipo`SensorEvent` que permite, entre otros, obtener los valores medidos por el sensor, almacenados en la propiedad `values` de tipo `float[]`.

Cada sensor posee medidas y unidades específicas, de modo que es necesario procesar dichas medidas en función del tipo de sensor. El sentido y la interpretación que debemos realizar de dichas medidas en función del tipo de sensor se explican en la siguiente dirección:http://developer.android.com/reference/android/hardware/SensorEvent.html#values

Por lo general, cuando el sensor devuelve un valor único, éste se almacena en el primer elemento de la tabla. Cuando las medidas se expresan según los ejes de coordenadas (aceleración, por ejemplo), los valores para los ejes x, y y z se almacenan respectivamente en los tres primeros elementos de la tabla.

La definición de los ejes x, y y z se realiza de manera relativa respecto al sentido natural de uso del terminal. El eje x se corresponde con el eje horizontal cuando se sostiene el dispositivo en su uso por defecto (que es, en teoría, el modo vertical para los smartphones y el modo apaisado para las tabletas). El eje y se corresponde con el eje vertical, mientras que el eje z es el eje de la profundidad.

El objeto `SensorEvent` posee, a su vez, la propiedad `timestamp`, que indica en qué instante se ha tomado la medida, y `accuracy`, que indica la precisión de la misma.

Es posible asociar un objeto de tipo `SensorEventListener` a un sensor mediante uno de los métodos `registerListener` de la clase `SensorManager`.

Sintaxis

```
boolean registerListener (SensorEventListener listener,
Sensor sensor, int rateUs, int maxBatchReportLatencyUs)
boolean registerListener (SensorEventListener listener,
Sensor sensor, int rateUs, Handler handler)
boolean registerListener (SensorEventListener listener,
Sensor sensor, int rateUs, int maxBatchReportLatencyUs,
Handler handler)
boolean registerListener (SensorEventListener listener,
Sensor sensor, int rateUs)
```

La versión más sencilla de estos métodos es la siguiente:

```
boolean registerListener (SensorEventListener listener,
Sensor sensor, int rateUs)
```

Recibe como parámetros un objeto de tipo `SensorEventListener`, una instancia de `Sensor`, así

como un parámetro entero que indica qué frecuencia de medida debe utilizar el sistema. Los posibles valores para este parámetro son SENSOR_DELAY_NORMAL, SENSOR_DELAY_UI,SENSOR_DELAY_GAME, y SENSOR_DELAY_FASTEST. Desde la versión 9 de la API (Android 2.3) también es posible indicar un delay en microsegundos.

Observe que este parámetro no tiene más que un valor indicativo, que cada terminal puede respetar o no.

Las variantes de registerListener reciben, como parámetro suplementario, un objeto de tipoHandler que permite delegar la producción de los eventos en un thread separado (por defecto, es el thread principal el que se utiliza).

Por último, la forma más completa del método, disponible desde la API 19, incluye una noción de procesamiento por lotes: las medidas se toman siempre según el intervalo indicado, aunque los resultados se devuelven por lotes, con un retardo que será como máximo el indicado por dicho parámetro (expresado en microsegundos). Esto permite limitar el consumo de energía del dispositivo tomando medidas precisas.

Un sensor se vuelve activo cuando, gracias al SensorManager, se invoca a su métodoregisterListener. A partir de ese momento, el sensor está habilitado para consumir energía hasta que se invoque al método unregisterListener.

Sinstaxis

```
public void unregisterListener (SensorEventListener listener)
public void unregisterListener (SensorEventListener listener, Sensor
sensor)
```

La primera versión del método deshabilita todos los sensores que están supervisados por elSensorEventListener que se pasa como parámetro. La segunda versión deshabilita únicamente el sensor que se pasa como parámetro.

Resulta fundamental invocar a dicho método lo antes posible tras el uso del sensor para limitar el consumo de energía por parte de la aplicación, o cuando una actividad se pone en pausa.

Del mismo modo, según el ciclo de vida de una actividad, el vínculo de un SensorEventListenercon un sensor debe realizarse en el método onResume de la actividad: de este modo el sensor no se utilizará más que durante el tiempo en que la actividad esté, realmente, en primer plano.

Localización geográfica

La localización geográfica, o geolocalización, permite localizar al dispositivo Android en un instante determinado de forma más o menos precisa proporcionando múltiple información y, en particular, sus coordenadas geográficas: la latitud y la longitud.

Cualquier aplicación puede, en lo sucesivo, utilizar la geolocalización para localizar al dispositivo y a su usuario.

Para respetar la privacidad del usuario, es conveniente prevenir al usuario de que la aplicación utilizará sus coordenadas geográficas. Dejarle la posibilidad de activar y desactivar el sistema de geolocalización en cualquier momento, por ejemplo, mediante una casilla de opción a marcar, es todavía mejor. Por defecto, esta casilla tendrá que estar desactivada para que el propio usuario la active dando su acuerdo de forma explícita para ser localizado.

Según el material que equipe al dispositivo Android, el sistema puede apoyarse sobre uno o varios dispositivos que permiten proveer las coordenadas de localización. Este material puede ser un receptor GPS, la red de telefonía móvil asociada, o las redes Wi-Fi circundantes.

Cada uno de estos dispositivos posee sus ventajas y sus desventajas. Por un lado, el sistema GPS permite obtener coordenadas precisas a cambio de un consumo de batería importante, de un entorno exterior y de un retardo en la localización que puede ser superior al de otros dispositivos.

Por otro lado, el sistema Android utiliza menos batería para captar la red de telefonía móvil y las redes Wi-Fi. Lo hace más rápidamente, incluso en el interior, aunque la localización es menos precisa.

El sistema de geolocalización de Android permite adaptarse automáticamente al material provisto por el dispositivo y activado por el usuario. Por ello, el desarrollador no debe suponer ningún dispositivo a la hora de proveer el acceso a la aplicación salvo en casos particulares. La aplicación debe adaptarse automáticamente a todos los entornos de hardware.

Por ello, no debe precisarse qué sistema se quiere utilizar sino de qué nivel de precisión desea disponer. El sistema determinará a continuación de forma automática sobre qué componentes de hardware apoyarse para responder a esta demanda.

1. Permisos

Para poder utilizar el sistema de geolocalización, la primera etapa consiste en dotar a la aplicación del derecho de localizar el dispositivo. Para ello, es preciso agregar el o los permisos correspondientes en el manifiesto: existen dos permisos, que dependen del nivel de precisión deseado.

El permiso `android.permission.ACCESS_COARSE_LOCATION` solicita permiso para utilizar los componentes de localización que tengan una precisión menor, como por ejemplo la combinación de redes de telefonía móvil y Wi-Fi.

El permiso `android.permission.ACCESS_FINE_LOCATION` solicita permiso para utilizar los componentes de localización más precisos, como el GPS, además de aquellos de menor precisión. Este permiso incluye de forma implícita al permiso`android.permission.ACCESS_COARSE_LOCATION`.

Ejemplo

```
<?xml version="1.0" encoding="utf-8"?>
<manifest>
  <uses-permission
    android:name="android.permission.ACCESS_FINE_LOCATION" />
</manifest>
```

2. Gestor de localización

Android proporciona la clase `LocationManager` que permite gestionar el servicio del sistema de geolocalización. La instancia de este servicio, el gestor, se recupera utilizando el método`getSystemService` y pasándole como parámetro el nombre del servicio deseado. El nombre del servicio de geolocalización está contenido en una constante: `Context.LOCATION_SERVICE`.

Sintaxis

```
public abstract Object getSystemService (String name)
```

Ejemplo

```
LocationManager locManager =
  (LocationManager) getSystemService(Context.LOCATION_SERVICE);
```

En función de los permisos de localización solicitados en el manifiesto, este gestor de accesos tiene más o menos dispositivos de localización.

Algunos de estos métodos solicitan que se les provea un conjunto de criterios para determinar qué dispositivo de localización entre los disponibles responde mejor a estos criterios. Estos criterios se deben proporcionar en un objeto de tipo `Criteria` creado utilizando su constructor por defecto.

Sintaxis

```
public Criteria ()
```

Ejemplo

```
Criteria criterios = new Criteria();
criterios.setAccuracy(Criteria.ACCURACY_HIGH);
```

Otros métodos permiten indicar explícitamente el nombre del dispositivo que se quiere utilizar, en forma de cadena de caracteres. Existen varias constantes que sirven para designar a estos dispositivos:

- `GPS_PROVIDER`: constante que contiene el nombre del sistema de GPS. Este dispositivo requiere el permiso `android.permission.ACCESS_FINE_LOCATION`.
- `NETWORK_PROVIDER`: constante que contiene el nombre del sistema de redes de telefonía móvil y Wi-Fi. Este dispositivo requiere el permiso`android.permission.ACCESS_COARSE_LOCATION`.
- `PASSIVE_PROVIDER`: constante que designa el sistema que permite recuperar las actualizaciones de la localización solicitada por las demás aplicaciones. Esto permite evitar solicitudes de localización superfluas y mutualizar aquellas recibidas por las aplicaciones. Este dispositivo, si bien requiere el permiso`android.permission.ACCESS_FINE_LOCATION`, no certifica que los valores de localización devueltos tengan una precisión fina.

Observe que una aplicación puede utilizar uno o varios dispositivos de localización, de forma simultánea o no.

3. Recuperar los datos de localización

Es posible adoptar varios escenarios para recuperar la localización del usuario. Es posible que ciertas aplicaciones requieran localizar al usuario una única vez; otras a intervalos regulares. Ciertas aplicaciones no requieren una gran precisión, otras, por el contrario, requieren una precisión muy fina.

El desarrollador deberá reflexionar sobre la mejor estrategia que se quiere implementar teniendo en cuenta no sólo las necesidades de la aplicación sino también las restricciones de hardware: conviene reducir al máximo el consumo energético.

Un objeto de tipo `Location` representa una localización en un momento dado. Este objeto contiene mucha información como, por ejemplo, la latitud, la longitud, la altitud, la precisión de la posición, la velocidad instantánea, el dispositivo que la ha enviado…

La clase `Location` proporciona a su vez métodos utilitarios, en concreto el método`distanceBetween` que permite calcular fácilmente la distancia entre dos puntos GPS o incluso el método `distanceTo` para calcular la distancia entre el objeto de tipo `Location` y otro objeto del mismo tipo.

Preste atención, una localización más actual no tiene por qué ser más precisa que otra más antigua. Hay que comparar, por tanto, los distintos objetos de tipo `Location` recibidos para guardar únicamente la localización más precisa aunque sea más antigua...

Es posible recuperar la localización del usuario desde la caché o buscándola una única vez de forma regular.

Para estas dos técnicas, es preciso indicar la acción a realizar cuando la localización se haya encontrado, lo que puede llevar cierto tiempo, o tras eventos que afecten al dispositivo, como por ejemplo su activación o desactivación. Esto puede realizarse de dos formas.

La primera forma consiste en utilizar un objeto de tipo `PendingIntent`.

Los datos de localización con forma de un objeto de tipo `Location` se agregan a los datos `Extra`del intent cuando se conocen. La clave del valor extra correspondiente es`KEY_LOCATION_CHANGED`. Las claves `KEY_PROVIDER_ENABLED` y `KEY_STATUS_CHANGED` seenvían cuando ocurren respectivamente los eventos de activación, de desactivación y de cambio de estado en el dispositivo correspondiente.

Ejemplo

```
Location loc = (Location) intent.getParcelableExtra(
  android.location.LocationManager.KEY_LOCATION_CHANGED);
```

La segunda forma consiste en implementar la interfaz `LocationListener` y proporcionar o no un objeto de tipo `Looper` que permita invocar a los métodos de la interfaz sobre un thread distinto al thread por defecto, es decir el thread principal. La interfaz `LocationListener` incluye cuatro métodos que se invocan cuando se encuentra una nueva localización, cuando el usuario ha activado el dispositivo, cuando lo ha desactivado o cundo ha cambiado el estado del dispositivo. Estos métodos son, respectivamente: `onLocationChanged`, `onProviderDisabled`,`onProviderEnabled` y `onStatusChanged`.

Sintaxis

```
public abstract void onLocationChanged (Location location)
public abstract void onProviderDisabled (String provider)
public abstract void onProviderEnabled (String provider)
public abstract void onStatusChanged (String provider, int status,
  Bundle extras)
```

Ejemplo

```
LocationListener locListener = new LocationListener() {
  public void onLocationChanged(Location location) {
    Log.d(TAG, "Localización recibida: lat="+
      location.getLatitude()+" / lon="+location.getLongitude());
  }

  public void onProviderDisabled(String provider) {
    Log.d(TAG, "Dispositivo desactivado: "+provider);
  }
  public void onProviderEnabled(String provider) {
    Log.d(TAG, "Dispositivo activado: "+provider);
  }

  public void onStatusChanged(String provider, int status,
    Bundle extras) {
    StringBuffer buf = new StringBuffer(
      "Estado modificado: dispositivo="+provider+" / estado=");
    switch (status) {
    case LocationProvider.AVAILABLE:
      buf.append("En servicio");
      Integer t = (Integer)extras.get("satellites");
      if (t != null)
        buf.append("(satellites="+t+")");
        break;
      case LocationProvider.OUT_OF_SERVICE:
        buf.append("Fuera de servicio");
        break;
      case LocationProvider.TEMPORARILY_UNAVAILABLE:
        buf.append("Temporalmente no disponible");
        break;
      default:
        break;
    }
    Log.d(TAG, buf.toString());
  }
};
```

a. En caché

El gestor de localización provee el método `getLastKnownLocation` que permite recuperar inmediatamente, desde una caché, la última localización enviada por el dispositivo especificado como parámetro. Este método devuelve un objeto de tipo `Location` o `null` si no hay disponible ninguna localización para este dispositivo. Presenta la ventaja de ser inmediato dado que no activa el dispositivo. Es un método simple y poco costoso de obtener una localización temporal mientras se espera a recibir otra más precisa. En contrapartida, la localización devuelta puede ser muy antigua, o poco precisa…

Sintaxis

```
public Location getLastKnownLocation (String provider)
```

Ejemplo

```
Location loc =
  locManager.getLastKnownLocation(LocationManager.NETWORK_PROVIDER);
```

b. Una sola vez

Ciertas aplicaciones sólo necesitan obtener la localización del usuario una única vez, por ejemplo para geolocalizar un comentario del usuario. Estas aplicaciones pueden utilizar el método`getLastKnowLocation` que hemos visto anteriormente, pero esta última posición puede ser demasiado antigua, o no muy precisa.

En este caso, desde Android 2.3 (API 9), es posible utilizar el método `requestSingleUpdate`. El retardo en la obtención del dato de localización depende de numerosos factores y puede llevar cierto tiempo.

Para versiones anteriores a Android 2.3 (API 9), el desarrollador puede utilizar actualizaciones regulares, descritas más adelante, y detener este procedimiento una vez haya ocurrido la primera recepción de los datos de localización.

Según la firma usada, el método `requestSingleUpdate` recibe como parámetro o bien directamente el nombre del dispositivo de localización que se quiere utilizar o bien un objeto de tipo `Criteria` que permite identificar a este dispositivo. Recibe también como parámetros un objeto que implementa la interfaz `PendingIntent` o un objeto de tipo `LocationListener` y el thread asociado según la solución escogida para recuperar los datos de localización.

Sintaxis

```
public void requestSingleUpdate (String provider,
  PendingIntent intent)
public void requestSingleUpdate (String provider,
  LocationListener listener, Looper looper)
public void requestSingleUpdate (Criteria criteria,
  PendingIntent intent)
public void requestSingleUpdate (Criteria criteria,
  LocationListener listener, Looper looper)
```

Ejemplo

```
locManager.requestSingleUpdate(criterios, locListener, null);
locManager.requestSingleUpdate(LocationManager.GPS_PROVIDER,
  pendingIntent);
```

c. Periódicamente

Ciertas aplicaciones requieren obtener periódicamente la localización del usuario (por ejemplo, una aplicación que registre el recorrido efectuado por el usuario). O incluso, requieren obtener varios datos de localización para guardar, únicamente, el más preciso.

En este caso, el desarrollador puede utilizar uno de los métodos `requestLocationUpdates`. Estos métodos permiten obtener varios datos de localización. Reciben como parámetros un entero`minTime` que permite especificar el retardo mínimo, en milisegundos, entre dos notificaciones de localización así como un número de coma flotante `minDistance` que permite especificar la distancia mínima, en metros, que debe haberse recorrido entre dos notificaciones. La combinación de estos dos valores permite definir el periodo de actualización de los datos de localización.

Si se informan los parámetros `minTime` y `minDistance` a `0`, es posible obtener actualizaciones de datos de localización lo más a menudo posible. En contrapartida, la batería del dispositivo puede consumirse muy rápido. Por ello, para ahorrar batería, se recomienda

especificar un valor de `minTime` lo más grande posible, no inferior a los 60 segundos, incluso aunque de forma interna, el refresco de los datos de localización pueda tener lugar con mayor o menor frecuencia que el valor especificado.

De forma similar a los métodos `requestSingleUpdate`, según la firma utilizada, el método`requestLocationUpdates` recibe como parámetro o bien directamente el nombre del dispositivo de localización que se quiere utilizar o bien un objeto de tipo `Criteria` que permite identificar a este dispositivo. Recibe, también, como parámetros un objeto de tipo`PendingIntent` o un objeto que implementa la interfaz `LocationListener` y el thread asociado según la solución escogida para recuperar los datos de localización.

Sintaxis

```
public void requestLocationUpdates (String provider, long minTime,
  float minDistance, PendingIntent intent)
public void requestLocationUpdates (String provider, long minTime,
  float minDistance, LocationListener listener, Looper looper)
public void requestLocationUpdates (String provider, long minTime,
  float minDistance, LocationListener listener)
public void requestLocationUpdates (long minTime, float minDistance,
  Criteria criteria, PendingIntent intent)
public void requestLocationUpdates (long minTime, float minDistance,
  Criteria criteria, LocationListener listener, Looper looper)
```

Ejemplo

```
locManager.requestLocationUpdates(0, 0, criterios, locListener,
  null);
locManager.requestLocationUpdates(LocationManager.GPS_PROVIDER, 0,
  0, pendingIntent);
```

d. Detener las actualizaciones

Para detener la recuperación de los datos de localización realizada mediante los métodos`requestSingleUpdate` y `requestLocationUpdates`, es preciso utilizar uno de los dosmétodos `removeUpdates` pasando como parámetro la acción concreta, es decir, o bien el objeto de tipo `PendingIntent`, o bien el objeto que implementa la interfaz `LocationListener`.

Sintaxis

```
public void removeUpdates (PendingIntent intent)
public void removeUpdates (LocationListener listener)
```

Ejemplo

```
locManager.removeUpdates(locListener);
```

Google Maps

Otro uso corriente de las aplicaciones es mostrar mapas geográficos que permiten al usuario situar rápidamente lugares, eventos, tiendas…

Por defecto, el sistema Android no incluye tales posibilidades. La empresa Google proporciona su biblioteca externa de cartografía `com.google.android.maps` para agregar tales funcionalidades al sistema básico.

La mayor parte de los dispositivos Android disponen de sistemas que incluyen esta biblioteca. Por ello, permiten a las aplicaciones de terceros mostrar e incorporar tales mapas geográficos, desde Android 1.5 (API 3).

Si bien esta biblioteca no forma parte de la plataforma Android básica, sí es, a día de hoy, una funcionalidad que se utiliza de forma tan común en tantas aplicaciones que resulta evidente abordar este tema.

1. Implementación

Además de la instalación de la biblioteca Maps, la aplicación debe declarar el uso de esta biblioteca y el desarrollador debe obtener autorización de uso de esta librería de la empresa Google. Estos temas se abordan en esta sección.

a. Instalación del SDK

Como ya se ha indicado más arriba, la plataforma Android no incluye por defecto la biblioteca Maps.

Una aplicación que quiera tener acceso a esta biblioteca debe estar compilada con el SDK que contenga esta biblioteca. Para ello, es preciso utilizar los módulos complementarios provistos por Google que contienen las API de Google correspondientes.

- Ejecute la herramienta **Android SDK Manager**.
- En la lista de la izquierda, seleccione **Available Packages** y, a continuación, en la lista de la derecha seleccione uno de los módulos **Google APIs by Google Inc.** correspondientes a la versión del SDK Android deseado.
- Haga clic en el botón **Install Selected**.
- Finalice la instalación como hemos visto (véase el capítulo El universo Android - Entorno de desarrollo).

Si se quieren realizar las pruebas en un emulador, hay que tener en cuenta que la imagen de sistema utilizada por AVD esté acompañada por los módulos complementarios de Google.

b. Configuración de la aplicación

Para poder utilizar la biblioteca externa Maps, la aplicación debe declarar este uso con la etiqueta`uses-library` que debe agregar en la etiqueta `application` del manifiesto.

Sintaxis

```
<uses-library android:name="com.google.android.maps" />
```

La etiqueta `uses-library` permite compilar la aplicación utilizando la biblioteca especificada. Además, permite asegurar que sólo se autoriza la instalación de esta aplicación

en sistemas Android que dispongan de esta librería.

Dado que la biblioteca se comunica con un servidor para descargar los mapas geográficos, la aplicación debe incluir la autorización para conectarse a Internet. Para ello, hay que agregar el permiso `android.permission.Internet` en el manifiesto.

Ejemplo

```
<?xml version="1.0" encoding="utf-8"?>
<manifest ...>
  <application ...>
    <uses-library android:name="com.google.android.maps" />
  </application>
  <uses-permission android:name="android.permission.INTERNET" />
</manifest>
```

c. Obtener una clave de uso

Todo usuario del servicio Google Maps debe proveer una clave de uso que permita al servicio identificarlo.

En concreto, cada una de las vistas `MapView` de una aplicación deben proveer esta clave de uso. Las vistas `MapView` se describen más adelante.

Sin esta clave, la aplicación funcionará correctamente salvo por el detalle de que los mapas geográficos no se visualizarán. Las vistas correspondientes estarán vacías o, para ser más exactos, contendrán una malla vacía.

Esta clave es gratuita y dependiente del certificado utilizado para firmar la aplicación. Esto significa que una única clave sirve para todas las vistas `MapView` de una misma aplicación y, por extensión, para varias aplicaciones firmadas con el mismo certificado.

Por el contrario, la clave será diferente para cada uno de los certificados utilizados para firmar la aplicación. En concreto, habrá una clave para la compilación en modo debug y otra para la compilación en modo release.

Habrá que tener presente reemplazar la clave de debug por la clave del modo release en el código fuente del proyecto antes de realizar la compilación en modo release que producirá la aplicación que se publicará.

La generación de la clave utiliza la firma MD5 del certificado de firma.

Para obtener una clave para el modo debug, hay que recuperar el valor de la firma MD5 del certificado correspondiente.

El archivo correspondiente al certificado de debug se llama `debug.keystore`. Está almacenado en la carpeta `.android` que figura en la raíz de la carpeta del usuario.

Es posible encontrar la ubicación del archivo buscándola en **Preferences** de Eclipse, y a continuación **Android** y **Build**.

La generación de la firma MD5 del certificado se realiza mediante la herramienta `keytool`.

Sintaxis

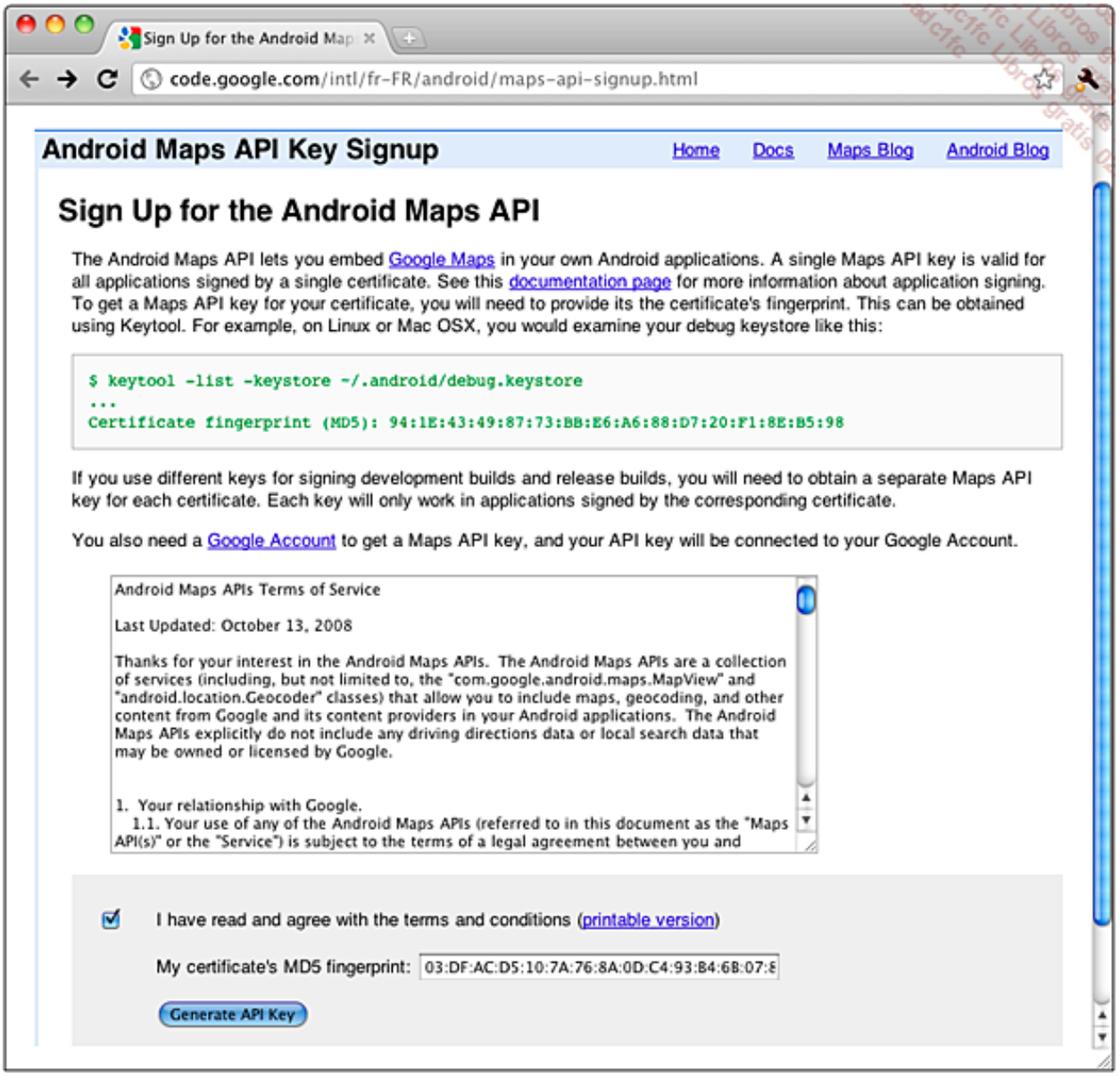
```
keytool -list -alias nombre_clave -keystore nombre_tienda.keystore
```

Las contraseñas de la tienda y de la clave para el modo debug son: `android`.

Ejemplo

```
$ keytool -list -alias androiddebugkey -keystore
/Users/seb/.android/debug.keystore
Introduzca la contraseña del Keystore:
androiddebugkey, 31 mayo 2010, PrivateKeyEntry,
Firma del certificado (MD5):
03:DF:AC:D5:10:7A:76:8A:0D:C4:93:B4:6B:07:86:10
```

- Diríjase a continuación a la siguiente dirección: http://code.google.com/android/maps-api-signup.html
- Lea los términos del Servicio Android Maps API que describen, en especial, las condiciones y restricciones de uso del servicio. Valide la opción **I have read and agree with the terms and conditions**.
- Copie y pegue la firma MD5 del certificado en el campo **My certificate's MD5 fingerprint**.



- Haga clic en el botón **Generate API Key**.

Se le pide conectarse a la cuenta de Google si no lo hubiera hecho ya.

Aparece la siguiente pantalla, que muestra la clave única generada, asociada a la firma MD5 proporcionada.

- Guarde la página web o, como mínimo, la clave, anotando a qué certificado está asociada. Si fuera necesario, es posible generar la misma clave varias veces.

El proceso de generación de la clave para el modo release es completamente idéntico al que se ha descrito aquí para el modo debug. Hay que pasar por las mismas etapas utilizando el archivo del certificado en modo release en lugar del que se ha utilizado aquí en modo debug.

2. Uso

La visualización de un mapa geográfico se realiza desde una vista dedicada representada por la clase `MapView`.

a. Declaración de la vista

La declaración de esta vista puede hacerse en un archivo layout. La etiqueta que debe utilizar es el nombre de la clase `MapView`: `com.google.android.maps.MapView`. Deben informarse varios atributos:

- El atributo `android:clickable` permite indicar si el usuario puede interactuar con la vista. Para esta vista, esto equivale a indicar si el usuario está autorizado o no a navegar por el mapa.
- El atributo `android:apiKey`, específico de esta vista, permite indicar la clave de uso que se proveerá al servidor para autorizar la descarga de mapas.

Sintaxis

```
<com.google.android.maps.MapView
  android:clickable="booleano"
```

```
  android:apiKey="cadena de caracteres"
  ... />
```

Ejemplo

```
<?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" >
  <com.google.android.maps.MapView
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:clickable="true"
    android:apiKey="0p6cjHRJBqcHwYoP14iUGkGox2vE8hgC5pf4v3g" />
</LinearLayout>
```

b. MapActivity

La vista `MapView` la gestiona una actividad que hereda de la clase `MapActivity`.

La clase `MapActivity` tiene en cuenta el funcionamiento asociado al uso de una vista de tipo`MapView`, como la visualización del mapa, los threads asociados… Es preciso, por tanto, crear una actividad que herede de la clase `MapActivity` y pasarle la `MapView` asociada.

El método `isRouteDisplayed` debe implementarse. Este método devuelve un valor booleano que indica si la vista se utiliza para proveer indicaciones de navegación.

Sintaxis

```
public class NombreDeClase extends MapActivity
```

Ejemplo

```
public class MiActividadMapa extends MapActivity {

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

  @Override
  protected boolean isRouteDisplayed() {
    return false;
  }

}
```

Es posible agregar botones en el mapa que permitan hacer zoom para acercar o alejar la vista. Para ello, hay que recuperar la instancia de la vista `MapView` e invocar a su método`setBuiltInZoomControls` pasando como parámetro un valor booleano que indique la activación de los botones.

Sintaxis

```
public void setBuiltInZoomControls(boolean on)
```

Ejemplo

```
MapView mapa = (MapView) findViewById(R.id.mapa);
```

```
mapa.setBuiltInZoomControls(true);
```

Hay dos modos disponibles para la representación del mapa: el modo satélite y el modo plano. Elmétodo `setSatellite` permite utilizar uno u otro de los modos pasándole como parámetro un valor especificando si se quiere utilizar el modo satélite o no, en cuyo caso se mostrará el modo plano.

Sintaxis

```
public void setSatellite(boolean on)
```

Ejemplo

```
mapa.setSatellite(true);
```

c. Geolocalización

La clase `MyLocationOverlay` provee la posibilidad de detectar la posición actual del dispositivo y, por tanto, del usuario, y mostrarla en tiempo real sobre el mapa.

Gráficamente, aparece un círculo azul parpadeando que indica la posición actual, envuelto de un círculo azul semi transparente más o menos amplio representando el nivel de precisión de la posición. Cuando más amplio sea el círculo, menor será la precisión, y a la inversa.

Para utilizarla, es preciso crear una instancia de la clase `MyLocationOverlay` utilizando el constructor. Éste recibe como parámetros el contexto y la vista `MapView` sobre la que debe mostrarse la posición del usuario.

Sintaxis

```
public MyLocationOverlay(android.content.Context context,
  MapView mapView)
```

Ejemplo

```
MyLocationOverlay locUsuario =
  new MyLocationOverlay(this, mapa);
```

A continuación, hay que activar la búsqueda de localización y su representación utilizando elmétodo `enableMyLocation`. Este método devuelve un valor booleano que vale verdadero si al menos un dispositivo de localización ha podido activarse, o falso en caso contrario.

Sintaxis

```
public boolean enableMyLocation()
```

Ejemplo

```
locUsuario.enableMyLocation();
```

Para detener la actualización de la localización y no mostrar más la posición del usuario, hay que invocar al método `disableMyLocation`.

La actualización de la localización en curso supone un consumo de recursos elevado, por ello hay que tener en cuenta no utilizarla más que cuando se está mostrando el mapa. Para ello, se recomienda invocar al método `enableMyLocation` en el método `onResume` de la actividad y detener la actualización invocando al método `disableMyLocation` en el método `onPause`.

Sintaxis

```
public void disableMyLocation()
```

Ejemplo

```
locUsuario.disableMyLocation();
```

La clase `MyLocationOverlay` hereda de la clase `Overlay`. Todo objeto de tipo `Overlay`debe agregarse a una lista de objetos de este tipo gestionado por el objeto de tipo `MapView`. Éste utilizará esta lista para mostrar, en orden, los elementos de la lista sobre el mapa. La lista se recupera utilizando el método `getOverlays`.

Sintaxis

```
public final java.util.List<Overlay> getOverlays()
```

Ejemplo

```
mapa.getOverlays().add(mLocUsuario);
```

También es posible mostrar una brújula sobre el mapa representando la orientación actual del dispositivo. Para ello, hay que utilizar los métodos `enableCompass` y `disableCompass` de la misma forma que para los métodos `enableMyLocation` y `disableMyLocation`.

Sintaxis

```
public boolean enableCompass()
public void disableCompass()
```

Ejemplo

```
locUsuario.enableCompass();
procesamiento();
locUsuario.disableCompass();
```

La búsqueda de la localización del usuario puede tomar cierto tiempo. Una vez realizada la primera localización, y mostrada en el mapa, puede ser interesante centrarla sobre el plano y hacer zoom sobre ella.

Estas acciones pueden realizarse utilizando el objeto de tipo `MapController` asociado a la vista`MapView`. La clase `MapController` proporciona varios métodos útiles para navegar sobre el mapa.

El método `getController` de la clase `MapView` permite recuperar este objeto.

Sintaxis

```
public MapController getController()
```

Ejemplo

```
MapController controlador = mapa.getController();
```

El método `animateTo` permite centrar el mapa sobre un punto determinado desplazando el mapa de forma animada. Este método recibe como parámetro un objeto de tipo `GeoPoint` con las coordenadas, latitud y longitud, del punto. El punto de tipo `GeoPoint` correspondiente a la posición actual del usuario lo devuelve el método `getMyLocation`.

Sintaxis

```
public GeoPoint getMyLocation()
public void animateTo(GeoPoint point)
```

Ejemplo

```
controlador.animateTo(locUsuario.getMyLocation());
```

El método `setZoom` en sí mismo permite hacer zoom a un nivel dado.

Sintaxis

```
public int setZoom(int zoomLevel)
```

Ejemplo

```
controlador.setZoom(15);
```

La clase `MyLocationOverlay` provee el método `runOnFirstFix` que recibe como parámetro un objeto de tipo `Runnable`, que contiene el código que se ejecutará cuando se encuentre la primera localización. Puede, por tanto, utilizarse para centrar el plano en la localización encontrada y hacer zoom sobre ella.

Sintaxis

```
public boolean runOnFirstFix(java.lang.Runnable runnable)
```

Ejemplo

```
locUsuario.runOnFirstFix(new Runnable() {
  public void run() {
    MapController controlador = mapa.getController();
    controlador.animateTo(locUsuario.getMyLocation());
    controlador.setZoom(15);
  }
});
```

Otras clases y métodos permiten agregar otros elementos en el mapa y ofrecen al usuario la posibilidad de interactuar con ellos. Se trata, entre otros, de las clases `OverlayItem` e `ItemizedOverlay`.

Introducción

Además de las conexiones Wi-Fi y Bluetooth, cada vez con mayor frecuencia los dispositivos integran la tecnología NFC. Esta tecnología se encuentra en primera línea para los servicios de pago sin contacto y jugará, sin duda, un papel importante en la Internet de los objetos.

Este capítulo presenta los aspectos claves de la tecnología NFC y, en particular, el mecanismo que tiene en cuenta los tags NFC en los sistemas Android. Por último, aborda la lectura y escritura de tags NFC.

La tecnología NFC

Las siglas NFC significan *Near Field Communication* (comunicación por campo cercano). Una comunicación NFC es inalámbrica, de poca intensidad (del orden de unos pocos centímetros). Se caracteriza por la ausencia de cualquier dispositivo adicional entre el emisor y el receptor (a diferencia del Bluetooth, por ejemplo), así como por la rapidez para establecer la comunicación.

1. Los escenarios de uso de NFC

La señal, muy débil, asegura que la comunicación la inicia el usuario del dispositivo, que debe acercar el terminal sobre el propio punto de acceso. Esto permite dotar un primer nivel de seguridad.

La tecnología NFC se ha pensado para tres tipos de uso:

- La lectura, por parte del dispositivo, de puntos NFC que contienen información específica (la dirección de un sitio web, un número de teléfono o información de contacto, etc.). Los tags (puntos de acceso) de tipo "smart-poster" son un ejemplo de uso: una ficha, o cualquier otro elemento, que representan un evento incluyen un tag NFC que contiene la dirección del evento, o que dará acceso a su sitio de Internet.
- La comunicación entre dispositivos NFC. En este caso, la comunicación NFC se entiende como una iniciativa de comunicación. La tecnología Android Beam es una aplicación directa: la comunicación entre dos smartphones compatibles se inicia automáticamente mediante una comunicación de tipo NFC, y pasa a continuación a utilizar la tecnología Bluetooth para el resto del intercambio.
- El uso del dispositivo como simulador de tag NFC. En este caso, el dispositivo y su tag NFC simulado representan un sistema de autenticación segura. Un uso típico, que se expande con rapidez, es el de los medios de transporte públicos en los que el dispositivo NFC tiende a remplazar a los abonos de transporte.

2. Los tags NFC

Actualmente cohabitan varias tecnologías de tags NFC, cada una con sus características específicas en términos de tamaño de memoria disponible, de velocidad de transmisión así como de coste. El espacio de almacenamiento, por ejemplo, varía de 48 bytes a varios kilobytes.

Los dispositivos NFC en Android se consideran como compatibles con las siguientes tecnologías: NfcA, NfcB, NfcF, NfcV, IsoDep y NDEF.

El framework Android es compatible con las siguientes tecnologías, aunque no son, necesariamente, compatibles con todos los dispositivos Android: MifareClassic, MifareUltralight, NfcBarecode y NdefFormatable.

Cada tecnología - NfcA, NfcB, etc. - se corresponde con una norma ISO específica (ISO 14443-3A, ISO 14443-3B, etc.).

El caso de la tecnología NDEF resulta particular: no se habla de una tecnología de tag sino de una definición de estructura de datos (*NFC Data Exchange Format*), compatible con todos los tags de tipo NFC.

El framework Android provee un conjunto de clases que permiten gestionar cada tecnología de tag de la lista anterior. El paquete básico para todas estas clases es `android.nfc.tech`, cada tecnología implementa la interfaz `android.nfc.tech.TagTechnology`.

Los métodos expuestos por la interfaz `TagTechnology` son los siguientes:

```
void close();
void connect();
Tag getTag();
```

```
boolean isConnected();
```

Para ilustrar la diferencia entre la tecnología NDEF y las demás, podemos comparar los métodos expuestos por las clases de la API Android correspondientes (aquí tomaremos el ejemplo de la tecnología NfcA):

Métodos expuestos por la clase `NfcA`:

```
void          close();
void          connect();
static NfcA   get(Tag tag);
boolean       isConnected();

byte[]        getAtqa();
int           getMaxTransceiveLength();
short         getSak();
Tag           getTag();
int           getTimeout();
void          setTimeout(int timeout);
byte[]        transceive(byte[] data);
```

Métodos expuestos por la clase `Ndef`:

```
Void          close();
Void          connect();
static Ndef   get(Tag tag);
Boolean       isConnected();

Boolean       canMakeReadOnly();
NdefMessage   getCachedNdefMessage();
Int          getMaxSize();
NdefMessage  getNdefMessage();
Tag          getTag();
String       getType();
Boolean      isWritable();
Boolean      makeReadOnly();
Void         writeNdefMessage(NdefMessage msg);
```

La clase `NfcA` utiliza principalmente tipos primitivos, mientras que la clase `Ndef` utiliza tipos de objeto complejos.

Compatibilidad con NFC

No todos los dispositivos que funcionan con Android son compatibles con la tecnología NFC. Antes de iniciar una comunicación NFC en una aplicación, es preciso verificar si la tecnología está disponible en el dispositivo del usuario. A continuación, es necesario "informar" al dispositivo que la aplicación es capaz de iniciar una comunicación NFC.

1. Usar con un emulador

Incluso aunque resulta indispensable realizar pruebas en condiciones reales, resulta práctico poder utilizar, en las distintas fases del desarrollo, un emulador de Android.

Para desarrollar funcionalidades NFC, además del emulador Android clásico, el desarrollador debe disponer de un mecanismo que le permita emular tarjetas/tags NFC. El proyecto Open NFC, financiado por la empresa Inside Secure (www.insidesecure.com), provee, de manera gratuita, dicho emulador.

Para hacer funcionar el emulador es preciso instalar tres elementos:

- Una imagen de terminal Android.
- La aplicación ConnectionCenter, que gestiona la conexión entre el emulador de tag y el emulador Android.
- La aplicación NFC Controller Simulation, que simula los tags NFC.

El proceso de instalación, así como el origen de los elementos que se deben descargar, están disponibles en la siguiente dirección: http://open-nfc.org/wp/editions/android/

2. Detectar si el dispositivo es compatible con NFC

En primer lugar, la aplicación debe indicar que requiere la autorización NFC.

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

Este permiso está disponible desde la versión 9 (2.3, Gingerbread) de la API Android, las versiones anteriores de Android no son compatibles con la tecnología NFC.

En función de las necesidades, se plantean dos escenarios: bien la aplicación se muestra como disponible únicamente para aquellos dispositivos compatibles con NFC (técnica de filtrado por dispositivo), o bien la compatibilidad con NFC se comprueba en el momento de uso de NFC (técnica de comprobación en tiempo de ejecución).

a. Filtrado por dispositivo

Como hemos visto (capítulo Publicar una aplicación), Play Store de Google permite filtrar los dispositivos en función de los recursos necesarios para la ejecución de una aplicación. Este filtrado, bien sea para la tecnología NFC o para cualquier otra tecnología, se realiza a nivel del manifiesto de la aplicación.

Para prohibir la instalación de una aplicación en dispositivos que no dispongan de la tecnología NFC, basta con agregar la etiqueta `<uses-feature>` correspondiente:

```
<uses-feature android:name="android.hardware.nfc"
android:required="true" />
```

b. Comprobación en tiempo de ejecución

Si el uso de la tecnología NFC no es esencial para el funcionamiento de la aplicación, sería una lástima privar a una parte de parque de dispositivos de instalar la aplicación. En este caso, en lugar de prohibir la instalación de la aplicación, el desarrollador puede gestionar el acceso a la función NFC en la propia aplicación. Para ello, basta con comprobar si el dispositivo dispone de un adaptador NFC, mediante la función `getDefaultAdapter()`.

```
if(NfcAdapter. getDefaultAdapter(getApplicationContext())==null)
   Log.d(TAG,"Su dispositivo no dispone de la tecnología NFC");
else
   Log.d(TAG,"Enhorabuena, puede utilizar los tags NFC");
```

c. Activación por el usuario

Aunque el dispositivo del usuario sea compatible con NFC, es posible que esté deshabilitado en el dispositivo. El objeto `NfcAdapter` permite detectar el estado del adaptador NFC, mediante el método `isEnabled()`. Este método devuelve `true` si el adaptador está habilitado, o `false` en caso contrario. A continuación resulta sencillo solicitar al usuario que active, él mismo, la función, iniciando una actividad con la acción `Settings.ACTION_NFC_SETTINGS`.

```
NfcAdapter nfcAdapter= NfcAdapter.getDefaultAdapter(getApplicationContext());

if(nfcAdapter ==null)
   return;

if(!nfcAdapter.isEnabled())
   startActivity(new Intent(Settings.ACTION_NFC_SETTINGS));
```

Descubrir un tag NFC

En función del contenido de un tag NFC el sistema Android encapsula la información en `intents` de acciones diferentes. A cada acción de intención le corresponde, en la aplicación, un filtro de intención específico.

Las intenciones creadas por el sistema siguen una jerarquía, que va de más a menos especializada.

- Nivel 1: si el tag contiene un mensaje con formato NDEF, el sistema lee el primer registro del mensaje para tratar de extraer un tipo MIME o una URI. Si dicha extracción es correcta, es la aplicación capaz de trabajar con el tipo MIME, o la aplicación indicada por la URI, la que se inicia. La intención correspondiente tiene como acción `ACTION_NDEF_DISCOVERED`.
- Nivel 2: si el tag no contiene un mensaje en formato NDEF, o si no se encuentra ninguna aplicación en la etapa 1, el sistema selecciona la aplicación teniendo en cuenta las tecnologías de tag que cada aplicación ha declarado como capaz de procesar, a nivel de los filtros de intención. El sistema crea, en este caso, una intención de acción`ACTION_TECH_DISCOVERED`.
- Nivel 3: por último, si no se ha podido seleccionar ninguna aplicación en las dos etapas anteriores, se crea una intención de tipo `ACTION_TAG_DISCOVERED`, que se provee a la aplicación que tenga en cuenta esta acción de intención.

Es tarea del desarrollador declarar los filtros de intención que mejor se correspondan con los mensajes utilizados en su aplicación. En efecto, en el caso de que existan varias aplicaciones disponibles, el sistema proporciona la lista de aplicaciones candidatas y deja que el usuario seleccione la aplicación que se iniciará.

Es importante que el desarrollador declare la compatibilidad de los tags NFC en su aplicación. Resulta, en efecto, fundamental que la aplicación se seleccione lo antes posible en el proceso de selección por el sistema, en caso contrario corre el riesgo de que alguna otra aplicación se le adelante y se inicie antes. Por otro lado, si el desarrollador declara la compatibilidad con una tecnología que no gestiona correctamente puede contrariar al usuario, ¡que desinstalará la aplicación tras el primer error!

1. Compatibilidad con una intención ACTION_NDEF_DISCOVERED

En el caso de un mensaje con formato NDEF, es el propio mensaje el que se encapsula en la intención `ACTION_NDEF_DISCOVERED`.

El sistema lee el primer registro del mensaje NDEF, para determinar la estructura del conjunto del mensaje.

A continuación se enumeran los formatos compatibles con la intención y los filtros de intención correspondientes:

- `TNF_ABSOLUTE_URI`

  El filtro de intención correspondiente a este formato es el siguiente:

```
<intent-filter>
    <action android:name="android.nfc.action.NDEF_DISCOVERED"/>
    <category android:name="android.intent.category.DEFAULT"/>
   <data android:scheme="http"
              android:host="www.misitioweb.es"
              android:pathPrefix="/inicio.html" />
</intent-filter>
```

- `TNF_MIME_MEDIA`

Es compatible con todos los tipos MIME: la etiqueta `<data android:mimetype>` permite filtrar el o los tipos compatibles.

```
<intent-filter>
    <action android:name="android.nfc.action.NDEF_DISCOVERED"/>
    <category android:name="android.intent.category.DEFAULT"/>
    <data android:mimeType="text/plain" />
</intent-filter>
```

- `TNF_EXTERNAL_TYPE`

  Este formato permite definir un tipo de mensaje propietario. El filtro de intención menciona el dominio, que se corresponde con la organización que declara este tipo propietario, y el tipo deseado. Esta información se indica en la propiedad `pathPrefix` de la etiqueta `data`. La etiqueta `data` contiene también el esquema utilizado, así como la mención al tipo externo.

```
<intent-filter>
    <action android:name="android.nfc.action.NDEF_DISCOVERED" />
    <category android:name="android.intent.category.DEFAULT" />
    <data android:scheme="vnd.android.nfc"
        android:host="ext"
        android:pathPrefix="/com.midominio:miTipo"/>
</intent-filter>
```

- `TNF_WELL_KNOWN`

  Este formato debe verse como un formato que encapsula varios subtipos, y que provee convenciones de formato de escritura de datos, de cara a hacer el conjunto más compacto y eficaz. Los subtipos, prefijados por `RTD` (*Record Type Definition*), se corresponden con los que pueden codificarse en un tag NFC: principalmente, texto (`RTD_TEXT`), una URI (`RTD_URI`) o un smart_poster (`RTD_SMART_POSTER`). Por ejemplo, el subtipo `RTD_URI` permite almacenar el protocolo en un byte, en lugar de escribirlo completamente, como sería necesario en un registro de formato `TNF_ABSOLUTE_URI`.

  El filtro de intención compatible con un registro de tipo WELL_KNOWN es el correspondiente a su subtipo.

Se recomienda almacenar los mensajes en formato TNF_WELL_KNOWN, este formato es más eficaz que los formatos TNF que encapsula. ¡No hay que olvidar que los tags NFC baratos no pueden almacenar más de 48 bytes!

- `TNF_EMPTY`, `TNF_UNCHANGED` y `TNF_UNKNOWN`

  Estos tipos de registro no son compatibles con un mensaje de tipo NDEF. En su lugar, se inicia una intención de tipo `ACTION_TECH_DISCOVERED`.

2. Compatibilidad con una intención ACTION_TECH_DISCOVERED

Si se selecciona la aplicación en la segunda etapa, la intención provista tendrá una acción`ACTION_TECH_DISCOVERED`. La información encapsulada en la intención será la relativa a las tecnologías del tag y al contenido en formato binario. El paquete `android.nfc.tech` provee las clases de encapsulación para cada tecnología, facilitando la explotación del contenido del mensaje.

Para filtrar correctamente la intención creada por el sistema, el desarrollador debe especificar en el`manifest` la lista de tecnologías con las que es compatible su aplicación.

Esta lista debe declararse en un archivo XML, en forma de enumeración. La raíz de la enumeración es el tag `<tech-list>`.

El formato del archivo XML es el siguiente:

```
<resources xmlns:xliff="urn:oasis:names:tc:xliff:document:1.2">
    <tech-list>
        <tech>...</tech>
        <tech>...</tech>
   </tech-list>
</resources>
```

El archivo puede contener varias etiquetas `<tech>`, cada una de las etiquetas especifica una tecnología. El sistema interpreta esta información considerando la lista como una lista "`Y`": la aplicación se inicia si el tag leído es compatible con todas las tecnologías enumeradas.

Es posible especificar varias listas de tecnologías, informando varios tags `<tech-list>` en el archivo XML. Cada lista se comporta respecto a las demás como una enumeración "`O`". Este mecanismo permite especificar varios subconjuntos de tecnologías.

El siguiente ejemplo presenta un archivo XML que indica que la aplicación es compatible con los tags de tecnologías NfcA y NDEF o MifareClassic y MifareUltraLight.

```
<resources xmlns:xliff="urn:oasis:names:tc:xliff:document:1.2">
    <tech-list>
        <tech>android.nfc.tech.NfcA</tech>
        <tech>android.nfc.tech.Ndef</tech>
    </tech-list>

    <tech-list>
        <tech>android.nfc.tech.MifareClassic</tech>
        <tech>android.nfc.tech.MifareUltralight</tech>
    </tech-list>
</resources>
```

El archivo XML de las tecnologías compatibles puede llamarse de cualquier forma. Se ubica en la carpeta `/res/xml`, y se incluye una etiqueta XML en el `manifest` indicando el nombre de archivo de las `tech-lists`.

```
<activity>
...
<intent-filter>
    <action android:name="android.nfc.action.TECH_DISCOVERED"/>
</intent-filter>

<meta-data android:name="android.nfc.action.TECH_DISCOVERED"
    android:resource="@xml/mi_lista_tech " />
...
</activity>
```

Si la aplicación tiene como objetivo escribir tags NFC sobre un soporte, es preciso filtrar este tipo de intención, teniendo en cuenta que los tags están vacíos en el momento de la selección de la aplicación por parte del sistema. La elección se realiza, por tanto, de manera natural sobre las tecnologías soportadas en el tag.

3. Compatibilidad con una intención ACTION_TAG_DISCOVERED

Una intención `ACTION_TAG_DISCOVERED` será más compleja de gestionar, pues la intención no encapsula más que el contenido en formato binario. Es tarea del desarrollador parsear correctamente la tabla binaria para extraer la información.

El filtro de intención que se corresponde con intenciones de tipo `ACTION_TAG_DISCOVERED`refleja bien el aspecto genérico de este tipo de compatibilidad, no indica ningún contenido para el mensaje,

ni tampoco ninguna tecnología.

```
<intent-filter>
    <action android:name="android.nfc.action.TAG_DISCOVERED"/>
</intent-filter>
```

4. Android Application Records

Android 4.0 incluye un nuevo tipo de registro en los tags NFC: los *Android Application Records* (AAR). Este tipo de registro permite especificar explícitamente el nombre de la aplicación compatible con un tag NFC.

Para ello, tras descubrir un tag que contiene un mensaje de tipo NDEF, el sistema Android analiza todos los registros del mensaje con la intención de encontrar un registro AAR. Si este registro existe, se inicia la aplicación mencionada en el registro. Si la aplicación no estuviera instalada en el dispositivo del usuario, el sistema propondrá descargarla directamente de Play Store.

A diferencia del mecanismo de descubrimiento por filtro de intención, el inicio de la aplicación mediante un registro AAR se realiza a nivel de la aplicación, no a nivel de la actividad.

Los sistemas Android de versión anterior a la versión 4.0 no son compatibles con los registros AAR; el desarrollador que desee incluir una compatibilidad con las versiones 2.x y 3.x deberá implementar también los filtros de intención.

5. Foreground dispatch

Cuando su aplicación de gestión de tags NFC se ejecuta, puede resultar molesto para el usuario que presenta un tag que se abra una ventana de notificación que pide seleccionar qué aplicación debe utilizar el tag: lo deseable, en este caso, sería forzar al sistema a utilizar la aplicación en curso.

Para ello, conviene implementar en la actividad en curso, la misma que debe utilizar el tag, un mecanismo de sobrecarga, llamado foreground dispatch.

Este mecanismo se declara en el código de la actividad, habitualmente en los métodos`onCreate()` y `onResume()`, con ayuda del método `enableForegroundDispatch(...)` del objeto `NfcAdapter`.

Los parámetros esperados por el método `enableForegroundDispatch` son los siguientes:

- La actividad que se ejecutará cuando se descubra el tag.
- Un objeto de tipo `PendingIntent`, que se rellenará por el sistema y se proveerá a la actividad. Para no volver a abrir la actividad en curso, el `Pending Intent` se declarará con el flag `Intent.FLAG_ACTIVITY_SINGLE_TOP`.
- Una tabla de `IntentFilter`, que indicará al sistema cuáles son las intenciones compatibles con la actividad.
- Por último, una tabla de dos dimensiones indicando qué tecnologías NFC son compatibles con la actividad.

Por otro lado, es indispensable que la actividad que invoca al método`enableForegroundDispatch(...)` se habilite. Por ello, la llamada se hará en el método`onResume()` de la actividad.

Del mismo modo, conviene deshabilitar el mecanismo de foreground dispatch cuando se pone la actividad en pausa, invocando al método `disableForegroundDispatch()` desde el método `onPause()` de la actividad.

Es posible recuperar la intención que se envía a la actividad cuando se descubre el tag mediante el método `onNewIntent(Intent)` de la actividad. Para ello, es preciso sobrecargar este método para procesar la intención y su contenido.

```
...
NfcAdapter nfcAdapter;

@Override
public void onCreate(Bundle savedState) {
   super.onCreate(savedState);
// ...

   nfcAdapter = NfcAdapter.getDefaultAdapter(this);

}

@Override
public void onResume() {
   super.onResume();
   if(nfcAdapter==null)
      return;
   nfcAdapter.enableForegroundDispatch(this, getPendingIntent(),
getIntentFilters(), getTechLists());
}


@Override
public void onPause() {
   super.onPause();
   if(nfcAdapter==null)
      return;
   nfcAdapter.disableForegroundDispatch(this);
}

@Override
public void onNewIntent(Intent intent) {
    Tag tagFromIntent = intent.getParcelableExtra(NfcAdapter.EXTRA_TAG);
    Toast.makeText(this, "Tag descubierto", Toast.LENGTH_LONG).show();
}
...
```

Tradicionalmente, el mecanismo de foregroundDispatch se utiliza para escribir tags NFC, la compatibilidad de los tags de lectura se ha declarado en el manifiesto. En este escenario, es posible imaginar cómo se invoca al método `enableForegroundDispatch` en el método`onClick` de un botón de la interfaz, por ejemplo.

El mecanismo de foregroundDispatch es prioritario al mecanismo de Android Application Record.

Leer un tag NFC

La lectura de un tag NFC se realiza una vez se ha decidido qué actividad se hace cargo del mismo. El tag y su contenido se proveen a la actividad en la intención, que contiene también la acción que permite determinar qué tipo de mensaje contiene el tag; basta con comprobar el valor de la acción de la intención para saber qué método utilizar para descifrar el o los mensajes.

```
if(intent.getAction().equals(NfcAdapter.ACTION_NDEF_DISCOVERED))
   Log.d(TAG,"Lectura de un tag NDEF");
else
if(intent.getAction().equals(NfcAdapter.ACTION_TECH_DISCOVERED))
   Log.d(TAG,"Lectura de un tag TECH");
```

Veremos con detalle la lectura de los distintos tipos de mensajes NDEF.

1. Determinar el contenido de un tag NDEF

En el caso de un tag que contenga mensajes NDEF, la lectura se lleva a cabo recuperando en forma de `ParcelableArrayExtra` el valor parseable con nombre`NfcAdapter.EXTRA_NDEF_MESSAGES` contenido en la intención y, a continuación, convirtiendo cada elemento de la tabla en un objeto de tipo `NdefMessage` (no hay que olvidar que un tag NDEF puede contener varios mensajes con formato NDEF).

```
Parcelable[] parcelableNedfMessages =
intent.getParcelableArrayExtra(NfcAdapter.EXTRA_NDEF_MESSAGES);

if(parcelableNedfMessages==null)
   return;

NdefMessage[] messages =
   new NdefMessage[parcelableNedfMessages.length];

for(int i =0;i<parcelableNedfMessages.length;i++)
   messages[i] = (NdefMessage)parcelableNedfMessages[i];
```

Cada `NdefMessage` que se extrae contiene uno o varios `NdefRecord`, que son las entidades que contienen los mensajes propiamente dichos.

En cada objeto `NdefRecord`, el mensaje se almacena como una tabla de bytes, llamada `Payload`, a la que se accede mediante el método `getPayload()` del objeto `NdefRecord`. Es preciso convertir estos datos en función del formato del registro `NdefRecord`. Para ello, el objeto`NdefRecord` expone los métodos: `getTnf()`, que devuelve el formato del `NdefRecord`, y`getType()`, que devuelve el subtipo en el caso de que el tipo de `NdefRecord` sea`TNF_WELL_KNOWN`.

La API 16 de Android presenta varios métodos para la clase `NdefRecord`, que facilitan la lectura e interpretación de los registros. Veremos cómo leer el contenido de registros `NdefRecord` mediante estos métodos, y cómo reemplazar estos métodos cuando la aplicación está preparada para una API anterior.

2. Leer una URI

La API 16 de Android provee el método `toUri()` que permite extraer una URI de un `NdefRecord`.

```
Uri tagUri= ndefRecord.toUri();
```

En las versiones anteriores de Android es preciso hacer referencia a las especificaciones del formato de los registros `NdefRecord`, disponibles en la siguiente dirección: http://www.nfc-forum.org/specs/

Estas especificaciones indican que la codificación sigue las siguientes reglas:

- El payload es una tabla de bytes.
- El primer byte (`payload[0]`) contiene un indicador que define el protocolo de la URI. Este formato se utiliza para aumentar la memoria disponible para el detalle de la URI, ahorrando así el espacio necesario para escribir el protocolo.
- El resto del payload constituye la URI propiamente dicha, codificada en UTF-8.

El protocolo puede decodificarse según las correspondencias dadas en la tabla siguiente (extracto):

Decimal	Hex	Protocolo
0	0x00	N/A. No prepending is done, and the URI field contains the unabridged URI.
1	0x01	http://www.
2	0x02	https://www.
3	0x03	http://
4	0x04	https://
5	0x05	tel:
6	0x06	mailto:
7	0x07	ftp://anonymous:anonymous@
8	0x08	ftp://ftp.
9	0x09	ftps://
…	…	…

Codificación del prefijo de la URI (extraído de NFC Forum)

La extracción de una URI se realiza de la siguiente manera:

```
private Uri readUri(byte[] payload) {
   String[] protocolo = new String[] {
         "http://www.",
         "https://www.",
         "http://",
         "https://",
         "tel:",
         "mailto:",
         "ftp://anonymous:anonymous@",
         "ftp://ftp.",
         "ftps://"
         [...] // Todos los protocolos.
      };

   if(payload.length<2)
      return null;

   int prefijoIndice = (payload[0] & (byte)0xFF);

   if (prefijoIndice < 0 || prefijoIndice >= protocolo.length)
      return null;

   String prefijo = protocolo[prefijoIndice];
   String sufijo = null;
```

```
   try {
      sufijo = new String(Arrays.copyOfRange(payload, 1,
payload.length),"UTF-8");
   } catch (UnsupportedEncodingException e) {
      e.printStackTrace();
   }
   return Uri.parse(prefijo + sufijo);
}
```

El método se invoca de la siguiente manera:

```
Uri miURI = readUri(ndefRecord.getPayload());
```

3. Leer una cadena de caracteres

En el caso de un `NdefRecord` cuyo subtipo sea `RTD_TEXT`, los primeros bytes del payload contienen información acerca de la codificación del contenido.

- Un primer elemento define la codificación, UTF-8 o UTF-16, así como la longitud del código que le sigue.
- El siguiente elemento define el código del lenguaje utilizado.

La extracción del texto del registro consiste, por tanto, en determinar la codificación de la cadena de caracteres "útiles" de payload y, a continuación, transcribir la tabla de bytes como una cadena de caracteres en función de su codificación.

```
private String readText(byte[] payload) throws
UnsupportedEncodingException {

   String formatoCodificacion =
      ((payload[0] & 0200) == 0) ? "UTF-8" : "UTF-16";

   int longitudCodigoLenguaje = payload[0] & 0077;

   String codigoLenguaje =
      new String(payload, 1, longitudCodigoLenguaje, "US-ASCII");

   return new String(payload,
      longitudCodigoLenguaje + 1,
      payload.length - longitudCodigoLenguaje - 1,
      formatoCodificacion);
}
```

4. Leer un tipo MIME

La lectura de un registro de tipo `TNF_MIME_MEDIA` es muy sencilla, pues existen métodos fuertemente tipados disponibles desde la versión 9 de Android.

Un tag de tipo `TNF_MIME_MEDIA` incluye dos campos: el tipo MIME, que debe leerse mediante el método `getType()` del objeto `NdefRecord`, y un contenido (value), al que se accede mediante el método `getValue()`.

```
   private void readMimeTypeTag(NdefRecord record) {
      String mimeType=new String(record.getType());
      String value =new String(record.getPayload());
   }
```

5. Leer un tag de tipo TNF_WELL_KNOWN

Android trata los tags de tipo `TNF_WELL_KNOWN` del mismo modo que los tags de tipo equivalente: si el subtipo del tag `TNF_WELL_KNOWN` es `RTD_URI`, el tag se leerá como un tag de

tipo`TNF_ABSOLUTE_URI`.

El método `getType()` del objeto NdefRecord devuelve el subtipo del tag, tal y como se indica en la siguiente tabla:

Subtipo (RTD)	Descripción
RTD_SMART_POSTER	Registro de tipo URI, basado en el análisis del payload.
RTD_TEXT	Registro de tipo MIME de valor text/plain.
RTD_URI	URI.

Basta, entonces, con determinar el subtipo del tag `TNF_WELL_KNOWN`, y procesar el payload del tag en función del subtipo.

```
   if(record.getType()==NdefRecord.RTD_TEXT) {
      //... tratar el payload como para leer una cadena de
caracteres
   }
   else if (record.getType()==NdefRecord.RTD_URI) {
      //... tratar el payload como para leer una URI
   }
```

Escribir un tag NFC

Como con la lectura, la escritura de un tag se realiza, habitualmente, tras comprobar la compatibilidad del tag detectada por la intención.

Se recomienda encarecidamente crear únicamente mensajes NFC con formato `NdefMessage`, que es, tal y como hemos visto antes, independiente de la tecnología de tags subyacente.

La escritura de un tag sigue la estructura de un objeto de tipo `NdefMessage`: el mensaje incluye uno o varios registros (objetos de tipo `NdefRecord`), que pueden ser de distintos tipos.

La escritura del tag se realiza mediante un objeto de tipo `android.nfc.tech.Ndef`, que provee métodos de alto nivel para la escritura.

Es posible obtener una instancia de `Ndef` utilizando el método estático `get(Tag)` de la clase `Ndef`. El tag que se pasa como parámetro es el objeto que la intención provee una vez descubierto el tag.

```
Tag tag = intent.getParcelableExtra(NfcAdapter.EXTRA_TAG);
Ndef ndef = Ndef.get(tag);
// ... escritura del tag
```

El método `get(tag)` puede devolver null, en caso de que el tag no esté soportado por la tecnología NDEF.

La escritura es muy sencilla: basta con inicializar la operación de escritura mediante el método`connect()`, escribir el tag con el método `writeNdefMessage(NdefMessage)` y cerrar la conexión invocando al método `close()`.

```
Try {
   ndef.connect();
   ndef.writeNdefMessage(mensaje);
   ndef.close();
} catch (IOException e) {
      e.printStackTrace();
}
```

Estos métodos no deben invocarse en el proceso principal.

Veremos, a continuación, con detalle la construcción del `NdefMessage`, objeto que se pasa como parámetro al método `writeNdefMessage()`.

Según la API Android utilizada, resultará más o menos complejo construir registros integrados en el mensaje. La API 14 incluye un `helper` estático que permite crear un tag de tipo URI, y la API 16 incluye, también, otros `helpers` para crear registros de tipo `TNF_MIME_MEDIA` y`TNF_EXTERNAL_TYPE`.

1. Definir un registro NdefRecord con la API 9

El desarrollador que quiera incluir la compatibilidad con la API 9 de Android debe utilizar el constructor por defecto del objeto `NdefRecord`, cuya firma es la siguiente:

```
NdefRecord record = new NdefRecord (short tnf, byte[] type, byte[] id, byte[] payload);
```

- `tnf` representa el formato del registro creado.
- `type` indica el tipo de registro.
- `id` permite asignar un identificador al registro (resulta interesante si el mensaje incluye

varios registros).

- `payload` es una tabla de bytes que contiene el mensaje propiamente dicho.

Evidentemente, la construcción de la tabla de bytes que constituye el `payload` será el foco de nuestra atención, mientras que los demás parámetros están claramente definidos por los valores provistos por la API de Android. Veremos la construcción de los principales tipos de `payload`, para los diferentes niveles de API.

a. Contruir un payload de tipo texto

Como hemos visto en la lectura, el payload de un registro de tipo texto debe comenzar con información relativa a la codificación del texto y, a continuación, el propio texto codificado.

- El primer bit define el formato del texto: UTF-8 (bit a 0) o UTF-16 (bit a 1).
- El segundo bit vale 0 (inutilizado en la norma actual).
- Los 6 siguientes bits indican la longitud de la codificación del idioma.

A continuación, se incluye el valor de la codificación del idioma (codificado en "US-ASCII", por ejemplo), y a continuación el propio texto.

La longitud total de la tabla de bytes que constituye el payload será, por tanto, la suma del byte de información acerca de la codificación, el tamaño del código de idioma y el tamaño del mensaje.

Es posible, entonces, construir el payload de la siguiente manera:

```
byte[] buildTextPayload(String mensaje, String codificacion,
Locale idioma) {
   Charset charsetCodificacion = Charset.forName(codificacion);
   byte[] payload;
   byte[] bIdioma =
         idioma.getIdioma().getBytes(Charset.forName("US-ASCII"));
   byte[] bMensaje= mensaje.getBytes(charsetCodificacion);
   int codigoCodificacion = codificacion.equals("UTF-8") ? 0 : 128;

   payload = new byte[1 + bIdioma.length + bMensaje.length];
   payload[0] = (byte)(codigoCodificacion + bIdioma.length);
   System.arraycopy(bIdioma, 0, payload, 1, bIdioma.length);
   System.arraycopy(bMensaje, 0, payload, 1 + bIdioma.length,
messageByte.length);
   return payload;
}
```

b. Construir un payload de tipo URI

Por motivos de eficacia, vistos antes, se recomienda escribir los mensajes de tipo URI utilizando el formato TNF_WELL_KNOWN. La tabla de bytes que componen el payload contiene, en este caso, la siguiente información:

- El primer byte contiene el código correspondiente al prefijo de la URI, tal y como se ha definido en la tabla de la sección Leer una URI.
- Los siguientes bytes contienen el sufijo de la URI, codificado en UTF-8.

La tabla de bytes tendrá, entonces, un tamaño igual a 1 + el tamaño del sufijo.

La construcción del payload se realiza de la siguiente manera:

```
byte[] buildUriPayload(String uri) {
   String[] protocolo = new String[] {"",
            "http://www.",
            "https://www.",
```

```
            "http://",
            "https://",
            "tel:",
            "mailto:",
            "ftp://anonymous:anonymous@",
            "ftp://ftp.",
            "ftps://"
            //[...] // Todos los protocolos.
         };

   byte[] payload;
   int indicePrefijo = -1;

   for(int i = 1;i<protocolo.length;i++)
      if(uri.startsWith(protocolo[i])) {
         indicePrefijo = i;
         break;
      }


   if(indicePrefijo==-1)
      return null;

   uri = uri.substring(protocolo[indicePrefijo].length());
   byte[] bUri = uri.getBytes(Charset.forName("UTF-8"));

   payload = new byte[1 + bUri.length];
   payload[0] = (byte)indicePrefijo;
   System.arraycopy(bUri, 0, payload, 1, bUri.length);

   return payload;
   }
```

No hay que olvidar sustraer el protocolo de la URI antes de codificarlo en UTF-8 y volver a copiar la tabla de bytes en el payload.

Preste atención, la tabla que enumera los protocolos tiene, como primer elemento, una cadena vacía. El bucle for que determina el prefijo debe tenerlo en cuenta y comenzar en el segundo elemento de la tabla.

Viendo esta forma de construir el payload, podemos deducir fácilmente el método de construcción de un payload para un registro de tipo Absolute_URI: basta con considerar que el prefijo de la URI es la cadena vacía - que se corresponde con tener un 0 en el primer byte del payload, y codificar toda la URI (prefijo incluido) en UTF-8 en el resto de la tabla de bytes.

2. Definir un registro NdefRecord con las API 14 y 16

La API 14 de Android provee un método que simplifica al máximo la creación de un registro ndefRecord de tipo URI:

```
NdefRecord ndefRecord
=NdefRecord.createUri("http://www.miurl.es");
```

Del mismo modo, la API 16 incluye algunos métodos estáticos que permiten crear directamente registros `NdefRecord` sin tener que codificar específicamente el payload. Entre estos métodos, destacaremos los métodos `createMime()` y `createExternal()`.

- El método `createMime(String mimeType, byte[]mimeData)` permite crear simplemente un registro `ndefRecord` de tipo `TNF_MIME_MEDIA`, y recibe como parámetros el tipo MIME (en forma de cadena de caracteres) y el contenido a codificar (en

forma de tabla de bytes).

```
byte[] data = getData();
NdefRecord ndefRecord = NdefRecord.createMime("image/jpeg",data);
```

- Del mismo modo, el método `createExternal(String domain, String type, byte[]data)` permite crear un registro de tipo `TNF_EXTERNAL_TYPE`, y recibe como parámetros el dominio, el tipo y los datos.

El único punto destacable de este método es la clase del dominio y del tipo: según la recomendación del foro NFC, esta información no debería ser sensible a mayúsculas y minúsculas, lo cual no es compatible con los principios de Android (los filtros de intención sí son sensibles). El sistema gestiona esta diferencia aplicando el método `toLowerCase()` sistemáticamente a estos dos parámetros en el método `createExternal()`.

Introducción

Dado que resulta imposible abordar todos los temas referentes a Android, parece necesario profundizar en tres materias más antes de dar por concluido este libro.

Aparecidos con la versión de Android 1.5 (API 3), los App Widgets, primer tema que abordamos en este capítulo, han conocido, desde su aparición, un alcance cada vez mayor: los usuarios se han acostumbrado a personalizar sus dispositivos Android.

El segundo tema, la protección de las aplicaciones de pago, todavía no ha encontrado, al parecer, la misma popularidad. Sin duda porque tiene que ver únicamente con las aplicaciones de pago, o bien porque los desarrolladores no le dedican la atención o el tiempo de desarrollo que merece. Y, por tanto, su uso resulta primordial para cualquier desarrollador que quiera ver retribuido su trabajo.

Terminaremos con la implementación de lo que es la última tendencia en términos de monetización: las compras integradas.

App Widget

A día de hoy, la mayoría de sistemas operativos de los ordenadores personales permiten agregar pequeñas aplicaciones directamente sobre el escritorio del usuario.

Estas aplicaciones, llamadas comúnmente widgets, gadgets o con menor frecuencia viñetas activas, permiten al usuario visualizar rápidamente cierta información y le proporcionan cierta funcionalidad asociada a un dominio de uso específico. Por ejemplo, existen widgets que muestran los datos meteorológicos actuales, el progreso de la bolsa, las últimas novedades de la actualidad, fotografías y otros que proporcionan funciones de calculadora, de reloj…

Android permite agregar tales widgets sobre el escritorio del usuario, es decir, la pantalla de bienvenida. De hecho, el escritorio es, en sí mismo, una aplicación. Se ejecuta automáticamente tras el arranque del sistema. Por ello, en concreto, Android propone integrar estos widgets en otra aplicación, por ejemplo, en este caso, la aplicación de pantalla de bienvenida.

La interfaz de usuario española del sistema Android utiliza el término widget tal cual, sin traducción.

Presionando durante cierto tiempo sobre la zona vacía del escritorio es posible mostrar un menú que nos propone, en concreto, una lista de widgets.

Para el desarrollador, este término puede tener otro sentido. En efecto, puede designar a los componentes gráficos estudiados anteriormente (véase el capítulo Descubrir la interfaz de usuario - Widgets) tales como los textos, los botones…

Para diferenciar ambos sentidos y así evitar confusiones, los widgets destinados a ser agregados en el escritorio se denominan en Android: App Widget.

1. Creación

El App Widget es un receptor de eventos especializado de tipo `AppWidgetProvider`. Esta clase hereda de la clase `BroadcastReceiver` y está especializado en el funcionamiento de los App Widgets. Es, por defecto, la clase madre de todos los App Widgets.

La creación de un App Widget comienza escribiendo su clase principal, que hereda, por tanto, de la clase `AppWidgetProvider`.

Sintaxis

```
public class NombreDeClaseDelAppWidget extends AppWidgetProvider {
    ...
}
```

Ejemplo

```
public class MiAppWidget extends AppWidgetProvider {
}
```

2. Declaración

Como todo componente de tipo `BroadcastReceiver`, un App Widget debe declararse en el manifiesto para poder ser creado por el sistema.

La sintaxis es, por tanto, la misma que se utilizó en los receptores de eventos. El atributo`android:name` debe contener el nombre de la clase del componente.

Hay que precisar en el filtro de acciones que el componente desea recibir la acción`android.appwidget.action.APPWIDGET_UPDATE`. También hay que agregar una etiqueta meta-data que permita indicar el archivo XML que incluye los elementos de configuración del App Widget descritos más adelante. Para ello, el atributo `android:name` debe informar el valor`android.appwidget.provider` y el atributo `android:resource` el nombre del archivo XML.

Sintaxis

```
<receiver android:name="cadena de caracteres" >
  <intent-filter>
    <action
      android:name="android.appwidget.action.APPWIDGET_UPDATE" />
  </intent-filter>
  <meta-data android:name="android.appwidget.provider"
```

```
    android:resource="recurso xml" />
</receiver>
```

Ejemplo

```
<receiver android:name=".MiAppWidget" >
  <intent-filter>
    <action
      android:name="android.appwidget.action.APPWIDGET_UPDATE" />
  </intent-filter>
  <meta-data android:name="android.appwidget.provider"
    android:resource="@xml/appwidget-config" />

</receiver>
```

3. Configuración

Todo App Widget requiere sus elementos de configuración.

Para ello, es preciso crear un nuevo archivo XML en la carpeta `res/xml`. El nombre del archivo importa poco. Es el archivo que se informa en la etiqueta `meta-data` vista anteriormente. Este archivo debe incluir la etiqueta principal `appwidget-provider`.

Sus atributos permiten proveer información asociada al App Widget.

Sintaxis

```
<appwidget-provider
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:minWidth="dimensión mínima en ancho (dp)"
    android:minHeight="dimensión máxima en ancho (dp)"
    android:updatePeriodMillis="entero"
    android:initialLayout="recurso layout"
    android:configure="cadena de caracteres"
    ...
/>
```

Ejemplo

```
<appwidget-provider
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:minWidth="146dp"
    android:minHeight="72dp"
    android:updatePeriodMillis="3600000"
    android:initialLayout="@layout/appwidget_init"
    android:configure="es.midominio.android.miapli.Conf
iguracionAppWidgetActividad"
/>
```

Los atributos `android:minWidth` y `android:minHeight` permiten indicar el tamaño mínimo del App Widget en dp, exclusivamente.

La aplicación Home posiciona los App Widgets sobre una malla. Para determinar el tamaño a informar en estos atributos, se aconseja utilizar la siguiente fórmula:

```
tamaño_en_dp = (número_de_casillas * 74) - 2
```

El atributo `android:resizeMode`, aparecido con la versión 3.0, permite al usuario redimensionar el App Widget horizontalmente (`resizeMode= "horizontal"`), verticalmente (`resizeMode= "vertical"`), en ambos sentidos (`resizeMode= "horizontal | vertical"`), o sin posibilidad de redimensionar (`resizeMode= "none"`).

El atributo `android:widgetCategory`, aparecido con la versión 4.2, permite indicar en qué ubicación puede posicionarse el App Widget: sobre la pantalla "home" o sobre la pantalla de bloqueo. En el primer caso, se indicará el va-lor `home_screen`, en el segundo `keyguard`. Es posible combinar ambos valores, de modo que el App Widget esté disponible tanto en el escritorio como en la pantalla de bloqueo.

El atributo `android:updatePeriodMillis` permite al desarrollador especificar la frecuencia con la que se quiere actualizar el App Widget mediante la llamada a su método `onUpdate`. El valor debe especificarse en milisegundos.

Si se encuentra en reposo, el dispositivo se activará para realizar esta actualización. Esto

puede causar problemas de consumo de recursos si la actualización se realiza con demasiada frecuencia. Para evitar esto, el sistema fuerza un valor mínimo de 30 minutos en el caso en el que el desarrollador haya especificado un valor inferior.

Para evitar la salida del modo reposo del dispositivo o para especificar un valor de actualización inferior a 30 minutos, es posible utilizar en su lugar una alarma de tipo `AlarmManager` configurada de tal forma que sólo se produzca cuando el dispositivo esté activo. En este caso, el atributo`android:updatePeriodMillis` debe valer 0 para desactivarlo. Esta solución permite también escoger al usuario la frecuencia con la que se quiere actualizar la información, por ejemplo en la actividad de configuración.

En este caso, el atributo `android:updatePeriodMillis` debe valer 0 para desactivarlo. Esta solución permite también escoger al usuario la frecuencia con la que se quiere actualizar la información, por ejemplo en la actividad de configuración.

El atributo `android:initialLayout` permite indicar el layout que se utiliza en la creación del App Widget hasta que se actualice con otro layout.

Del mismo modo, en el caso de un App Widget que pueda instalarse sobre la pantalla de bloqueo, el atributo `android:initialKeyguardLayout` permite indicar el layout utilizado en la creación del App Widget en este caso.

El atributo `android:configure` es opcional. Permite especificar el nombre completo de la clase que representa la actividad de configuración del App Widget; si existe, y que se ejecutará tras la creación del App Widget. Esta actividad de configuración se describe más adelante.

A modo de recordatorio, es posible especificar el icono y el texto que se quiere mostrar en la lista de widgets presentada al usuario. Estos atributos son, respectivamente, `android:icon`y `android:label`.

A partir de Android 3.0 (API 11), el nuevo atributo `android:previewImage` permite especificar una imagen que representa una captura de pantalla del App Widget en estado de funcionamiento. Esta imagen permite al usuario previsualizar el App Widget antes de instalarlo. Si no se especifica este atributo, el App Widget se representará mediante su icono, en lugar de con su imagen de previsualización, en la lista de App Widgets presentada al usuario. La aplicación Widget Preview, preinstalada en el sistema Android, del emulador permite crear fácilmente capturas de pantalla de un App Widget.

4. Ciclo de vida

La clase `AppWidgetProvider` define métodos correspondientes a los eventos que debe tratar un App Widget. Filtra las intenciones recibidas e invoca a sus métodos internos correspondientes. Estos métodos son los métodos `onUpdate`, `onDeleted`, `onEnabled` y `onDisabled`.

a. onEnabled

El método `onEnabled` permite inicializar los datos comunes a todas los mismos App Widget. Este método se invoca únicamente tras la primera creación de un App Widget de un tipo dado. Es decir, si, por ejemplo, el usuario agrega varios App Widgets del mismo tipo en el escritorio, sólo el primero recibirá esta llamada. Este método recibe como parámetro el contexto de la aplicación.

Sintaxis

```
public void onEnabled (Context context)
```

Ejemplo

```
@Override
public void onEnabled(Context context) {
  super.onEnabled(context);
}
```

b. onDisabled

El método `onDisabled` es el simétrico al método `onEnabled`. Es el último momento para poder limpiar lo que es

necesario, borrar datos temporales, cerrar conexiones abiertas... Este método se invoca únicamente cuando el último App Widget se elimina: si el usuario agrega varios App Widgets del mismo tipo sobre el escritorio, sólo el último recibirá esta llamada cuando se suprima. Este método recibe como parámetro el contexto de la aplicación.

Sintaxis

```
public void onDisabled (Context context)
```

Ejemplo

```
@Override
public void onDisabled(Context context) {
  super.onDisabled(context);
}
```

c. onUpdate

El método `onUpdate` se invoca una primera vez cuando el App Widget se crea. A continuación, se invoca a intervalos regulares definidos por el desarrollador. Si existen varios App Widgets del mismo tipo, este método sólo se invoca una vez para procesar todas estas instancias. Todos estos App Widgets se actualizarán casi al mismo tiempo. La cuenta del intervalo regular comienza con la creación del primero de los App Widgets. Este método recibe como parámetros el contexto de la aplicación, una instancia de tipo `AppWidgetManager` así como una tabla de identificadores únicos de los Widgets existentes.

Sintaxis

```
public void onUpdate (Context context,
  AppWidgetManager appWidgetManager, int[] appWidgetIds)
```

Ejemplo

```
@Override
public void onUpdate(Context context,
    AppWidgetManager appWidgetManager, int[] appWidgetIds) {
  super.onUpdate(context, appWidgetManager, appWidgetIds);
  for (int appWidgetId : appWidgetIds) {
      procesamiento(appWidegtId);
  }
}
```

d. onDeleted

El método `onDeleted` se invoca cuando se borra el App Widget, por ejemplo cuando el usuario lo lleva a la papelera. Este método recibe como parámetro el contexto de la aplicación y la tabla de los identificadores de los App Widgets suprimidos.

Este método no está nombrado correctamente en Android 1.5 (API 3). Para más información y una corrección: http://groups.google.com/group/android-developers/msg/e405ca19df2170e2

Sintaxis

```
public void onDeleted (Context context, int[] appWidgetIds)
```

Ejemplo

```
@Override
public void onDeleted(Context context, int[] appWidgetIds) {
  super.onDeleted(context, appWidgetIds);
}
```

5. RemoteViews

De forma similar a una actividad, la apariencia del App Widget se define en un archivo Layout. Éste se provee a la aplicación, que lo integra en su propio layout, como por ejemplo la aplicación de escritorio.

Android utiliza un objeto de tipo `RemoteViews` que permite contener el layout del App Widget y proveerlo a la aplicación host.

Sólo algunos componentes gráficos pueden utilizarse en un layout de App Widget. Son los

componentes `AnalogClock`, `Button`, `Chronometer`, `ImageButton`, `ImageView`,`ProgressBar` y `TextView`. A partir de Android 3.0 (API 11), también es posible utilizar nuevos componentes más interactivos como `GridView`, `ListView`, `StackView`, `ViewFlipper` y`AdapterViewFlipper`. Estos últimos pueden actualizar sus vistas automáticamente utilizando las nuevas clases `RemoteViewsService` y `RemoteViewsFactory`.

En el curso de su ciclo de vida y de su interacción con el usuario, el App Widget tendrá, sin duda, que cambiar su apariencia; modificar una imagen, cambiar un texto…

La clase `RemoteViews` provee todo un conjunto de métodos para modificar los distintos objetos, las distintas vistas, que representan la interfaz gráfica del App Widget.

En primer lugar, hay que crear la instancia de tipo `RemoteViews` utilizando uno de sus constructores. Éste recibe como parámetros el nombre del paquete que contiene el layout y el identificador único del layout del App Widget.

Sintaxis

```
public RemoteViews (String packageName, int layoutId)
```

Ejemplo

```
RemoteViews vistas = new RemoteViews(context.getPackageName(),
  R.layout.appwidget);
```

Este objeto permite, a continuación, modificar los atributos de las vistas que contiene utilizando un conjunto de modificadores, cada uno dedicado a un tipo de componente gráfico particular. Estos métodos reciben como parámetros el identificador único de la vista y los datos a proveer.

Sintaxis de algunos de estos métodos

```
public void setImageViewResource (int viewId, int srcId)
public void setProgressBar (int viewId, int max, int progress,
  boolean indeterminate)
public void setTextColor (int viewId, int color)
public void setTextViewText (int viewId, CharSequence text)
public void setViewVisibility (int viewId, int visibility)
```

Ejemplo

```
vistas.setTextViewText(R.id.appwidget_titulo, "Nuevo título");
```

Es posible especificar las intenciones que se ejecutarán tras hacer clic sobre una de las vistas del App Widget. Para ello, hay que utilizar el método `setOnClickPendingIntent` que recibe como parámetros el identificador único de la vista y el objeto de tipo `PendingIntent` que se quiere ejecutar como respuesta al clic.

Sintaxis

```
public void setOnClickPendingIntent (int viewId,
  PendingIntent pendingIntent)
```

Ejemplo

```
Intent intent = new Intent(context, MiActividadPrincipal.class);
PendingIntent pIntent = PendingIntent.getActivity(context, 0,
  intent, 0);
vistas.setOnClickPendingIntent(R.id.appwidget_icono, pIntent);
```

Una vez modificado el App Widget, hay que utilizar el objeto de tipo `AppWidgetManager` recibido como parámetro en el método `onUpdate` para solicitar la actualización del App Widget. La clase`AppWidgetManager` provee varios métodos `updateAppWidget` que reciben como parámetros uno o varios identificadores únicos designando el o los App Widgets cuyas vistas se modificarán.

La jerarquía de estas vistas de tipo `RemoteViews` se pasa como segundo parámetro.

A modo de recordatorio, la lista de los identificadores de los App Widgets se recibe a su vez como parámetro del método `onUpdate` y puede utilizarse aquí de manera sencilla.

Sintaxis

```
public void updateAppWidget (int appWidgetId, RemoteViews views)
public void updateAppWidget (int[] appWidgetIds, RemoteViews views)
```

Ejemplo

```
@Override
public void onUpdate(Context context,
    AppWidgetManager appWidgetManager, int[] appWidgetIds) {
  super.onUpdate(context, appWidgetManager, appWidgetIds);
  for (int appWidgetId : appWidgetIds) {
    RemoteViews vistas =
      new RemoteViews(context.getPackageName(),
        R.layout.appwidget);
    procesamiento(appWidgetId);
    appWidgetManager.updateAppWidget(appWidgetId, vistas);
  }
}
```

Preste atención, cuando se esté utilizando un AVD recién creado, es posible que las actualizaciones de los App Widgets no se tengan en cuenta. Para más información:http://code.google.com/p/android/issues/detail?id=8889

6. Actividad de configuración

Ciertos App Widgets puede que no sean completamente usables hasta que se hayan configurado. Por ejemplo, un App Widget que muestre el desfase horario con una zona geográfica solicitará al usuario que elija la zona geográfica deseada.

Una solución consiste en crear el App Widget con una zona por defecto y agregarle un`pendingIntent` que ejecute una actividad que permita especificar la zona cuando el usuario haga clic sobre el App Widget. O, directamente, ejecutar la actividad desde la creación del App Widget detectando si se trata de una creación o de una actualización...

Para facilitar esta operación, la aplicación host se encarga de ejecutar la acción`android.appwidget.action.APPWIDGET_CONFIGURE` antes de crear el App Widget. Si existe una actividad que responda a esta acción, entonces ésta se ejecutará, y el App Widget podrá, a continuación, crearse según los parámetros de configuración especificados mediante esta actividad.

En la práctica, el App Widget se crea ANTES que la ejecución de la actividad de configuración porque, recordemos, en este caso se ejecuta su método `onUpdate` mientras que, en teoría, esto no debería pasar. La ejecución del método `onUpdate` se realiza ANTES de ejecutar la actividad de configuración.

a. Declaración

La actividad de configuración debe indicar que puede responder a la solicitud`android.appwidget.action.APPWIDGET_CONFIGURE` agregando esta acción en su filtro de intención que figura en el manifiesto.

Ejemplo

```
<activity android:name=".ConfigurationAppWidgetActividad">
  <intent-filter>
    <action
    android:name="android.appwidget.action.APPWIDGET_CONFIGURE" />
  </intent-filter>
</activity>
```

Como complemento, la actividad deberá declararse en el archivo de configuración del App Widget mediante el atributo `android:configure` descrito anteriormente.

b. Creación

La actividad de configuración es una actividad como cualquier otra. No obstante, la ejecuta la aplicación host que espera, obligatoriamente, un valor resultado que indique si la etapa de configuración se ha desarrollado

correctamente o no. De este valor depende el resto de eventos.

Al ejecutar la actividad de configuración, la aplicación host proporciona el identificador del App Widget en los valores `Extras` del intent mediante la clave`AppWidgetManager.EXTRA_APPWIDGET_ID`. Es posible utilizar el valor`AppWidgetManager.INVALID_APPWIDGET_ID` como valor por defecto que permite detectar la omisión del identificador del App Widget.

Ejemplo

```
Intent intent = getIntent();
Bundle extras = intent.getExtras();
if (extras != null) {
  mAppWidgetId =
    extras.getInt(AppWidgetManager.EXTRA_APPWIDGET_ID,
                  AppWidgetManager.INVALID_APPWIDGET_ID);
}
```

Una vez el usuario haya ejecutado la actividad de configuración, si todo funciona correctamente, debe devolver el valor `RESULT_OK`. En este caso, la aplicación host crea el App Widget.

Pero si, por ejemplo, el usuario sale de la actividad sin validar la configuración del App Widget, la actividad puede considerarlo como una anulación de la configuración y, por tanto, devolver el valor`RESULT_CANCELED`. En este caso, la aplicación host anula la creación del App Widget. Si se trata de la aplicación de inicio, el App Widget no se muestra jamás.

Se recomienda especificar el valor resultado de `RESULT_CANCELED` desde la creación de la actividad de configuración de modo que éste sea el resultado que se devuelva, salvo si se provee explícitamente el valor `RESULT_OK`.

En ciertos casos particulares, por ejemplo, cuando el usuario hace clic en el botón **Inicio**cuando se está mostrando la actividad de configuración, el resultado `RESULT_CANCELED` no se toma en cuenta. El método `onDeleted` del App Widget no se invoca. Si bien el App Widget no aparece sobre la pantalla, su creación no se ha anulado. Existe sin mostrarse. Es algo así como un App Widget fantasma. Este bug hace que el desarrollador deba tener en cuenta este caso particular en su código. Para más información:http://code.google.com/p/android/issues/detail?id=9362,http://code.google.com/p/android/issues/detail?id=2539. El comentario número 15 explica, en particular, una solución.

En particular, el resultado se devuelve mediante el método `setResult` ya estudiado (véase el capítulo Los fundamentos - Actividad). El valor del resultado debe estar acompañado del identificador único del App Widget que permite a la aplicación host saber cuál es el App Widget afectado por el resultado. Para ello, hay que crear una intención y agregar en los datos extras el par clave-valor cuya clave es la constante `AppWidgetManager.EXTRA_APPWIDGET_ID` y el valor es el identificador del App Widget recuperado anteriormente.

Recuerde que el método `setResult` no finaliza la actividad. Hay que hacerlo explícitamente utilizando el método `finish`.

Ejemplo

```
Intent resultado = new Intent();
resultado.putExtra(AppWidgetManager.EXTRA_APPWIDGET_ID,
  mAppWidgetId);
setResult(RESULT_CANCELED, resultado);

Intent resultado = new Intent();
resultado.putExtra(AppWidgetManager.EXTRA_APPWIDGET_ID,
  mAppWidgetId);
setResult(RESULT_OK, resultado);
finish();
```

Para ello, la actividad debe utilizar la instancia del objeto de tipo `AppWidgetManager`. Esta instancia se recupera utilizando el método estático `getInstance` de la clase`AppWidgetManager`. Este método recibe como parámetro el contexto aplicativo.

Sintaxis

```
public static AppWidgetManager getInstance (Context context)
```

Ejemplo

```
AppWidgetManager appWidgetManager =
  AppWidgetManager.getInstance(context);
```

Proteger las aplicaciones de pago

Como hemos visto, una aplicación publicada se presenta bajo la forma de un archivo apk. Basta a un tercero con tener este archivo en posesión suya para que pueda compartirlo con otras personas. Esto no supone, por lo general, un problema para las aplicaciones gratuitas puesto que el desarrollador quiere que se distribuyan e instalen lo más posible, y poco importa el medio. Por el contrario, no ocurre lo mismo con las aplicaciones de pago. Estas copias ilegales son una pérdida de ingresos para el desarrollador...

Para protegerse contra el uso de copias ilegales de aplicaciones de pago (y únicamente de aplicaciones de pago) Android provee un sistema de verificación de licencias de las aplicaciones compradas e instaladas obligatoriamente a través de Play Store. Este servicio existe para los sistemas Android 1.5 (API 3) o superiores.

El principio de este sistema es simple y toma el aspecto de una librería llamada LVL (*License Verification Library*) o librería de verificación de licencia. Esta librería permite, mediante la aplicación Play Store instalada en el sistema Android, comunicarse con el servidor de licencias en línea. Éste devuelve, de forma segura, el estado de la licencia propia de la aplicación para el usuario correspondiente.

Es la aplicación Play Store la encargada de gestionar la comunicación en línea con el servidor. El desarrollador sólo tiene que ejecutar los comandos utilizando la librería y procesar las respuestas.

En función del resultado devuelto por la aplicación Play Store, el desarrollador podrá aplicar la estrategia que prefiera como, por ejemplo: autorizar el uso de la aplicación durante un cierto periodo de tiempo, restringir ciertas funcionalidades de la aplicación, o incluso bloquear por completo el acceso a la aplicación.

Este sistema requiere una conexión a la red para comunicarse con el servidor de licencias. Es preciso, por tanto, prever una alternativa cuando no exista conexión a la red como, por ejemplo, una caché de la licencia.

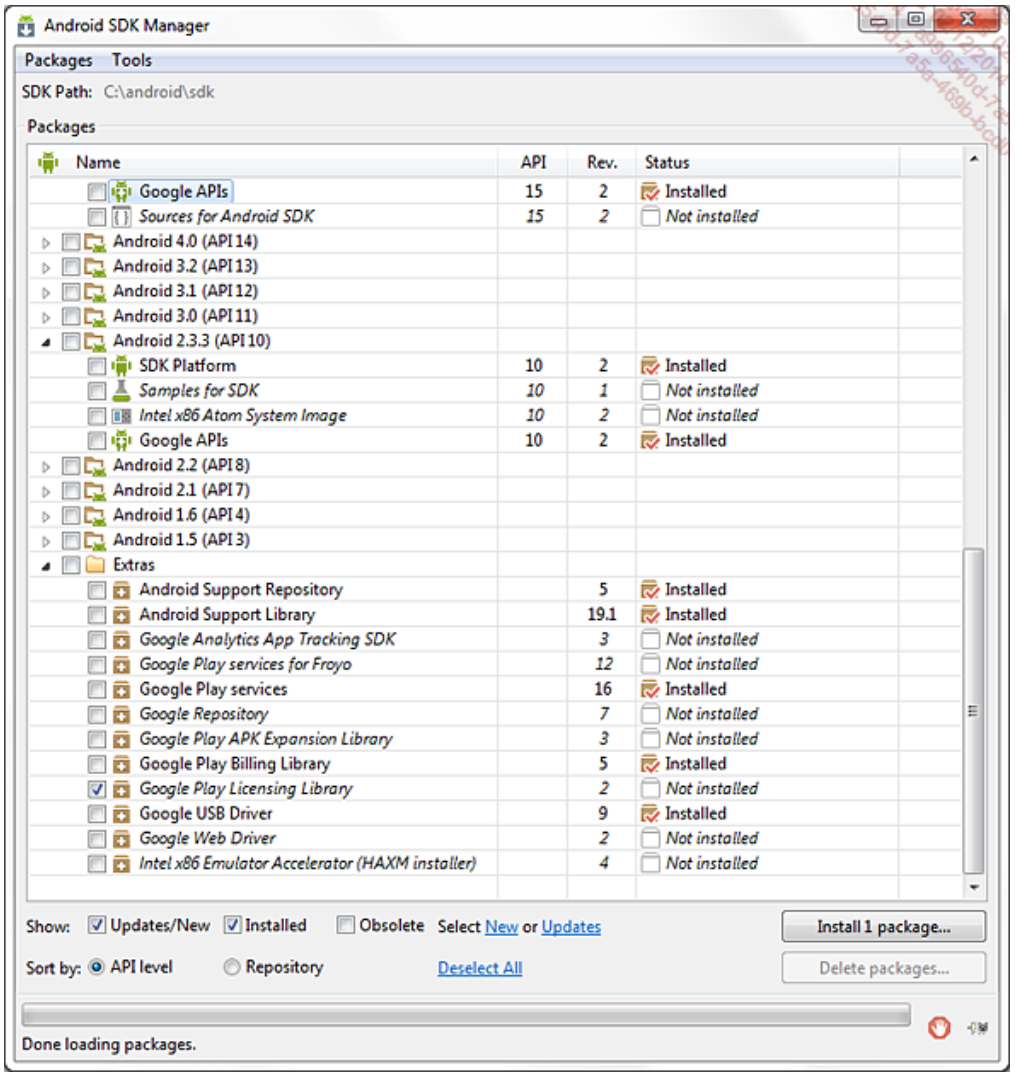
1. Instalación de la LVL

La librería de verificación de licencias requiere que la aplicación esté configurada para compilarse con la plataforma Android 1.5 (API 3) o superior. Esta librería se provee bajo la forma de archivos de código fuente que deben integrarse en el proyecto. Esta integración puede hacerse directamente copiando los archivos en el proyecto, o de manera indirecta, creando una librería de archivos de código fuente vinculada al proyecto. Esta última solución permite simplificar su reutilización. Esta librería no debe integrarse como librería externa, es decir, compilada de forma separada e integrada en los proyectos como archivo jar.

a. Descarga

La librería LVL se puede descargar como módulo utilizando el Android SDK Manager. Este módulo contiene, entre otros, los archivos de código fuente de la LVL, una aplicación que sirve de ejemplo y la documentación javadoc.

- Ejecute la herramienta **Android SDK Manager**.
- En la lista de la izquierda, seleccione **Available packages** y, a continuación, en la lista de la derecha, despliegue la carpeta **Extras** y marque **Google Play Licensing Library**.



Para probar la LVL desde el emulador, el AVD debe estar basado sobre un sistema **Google APIs by Google Inc**. Es preciso, por tanto, llegado el caso, marcar la opción correspondiente en la pantalla anterior, como es el caso aquí para la versión 15.

- Haga clic en el botón **Install xx Packages** y finalice la instalación como ya se ha visto en capítulos anteriores (véase el capítulo El universo Android - Entorno de desarrollo).

Los archivos se descargan e instalan en la carpeta *sdk*`/extras/google/play_licensing`.

b. Integración de la LVL en el código fuente

La primera forma de integrar el código de la LVL en un proyecto consiste en copiar directamente los archivos del código fuente. Éste es el método más sencillo y el más rápido. No obstante, es el menos productivo a largo plazo si lo que queremos es poder utilizar esta librería y mantenerla en varios proyectos distintos.

- Copie la carpeta `library/src/com` de la LVL en la carpeta `src` del proyecto.
- En Eclipse, actualice el proyecto para tener en cuenta los nuevos paquetes`com.google.android.vending.licensing` y`com.google.android.vending.licensing.util` así como las clases que contienen.

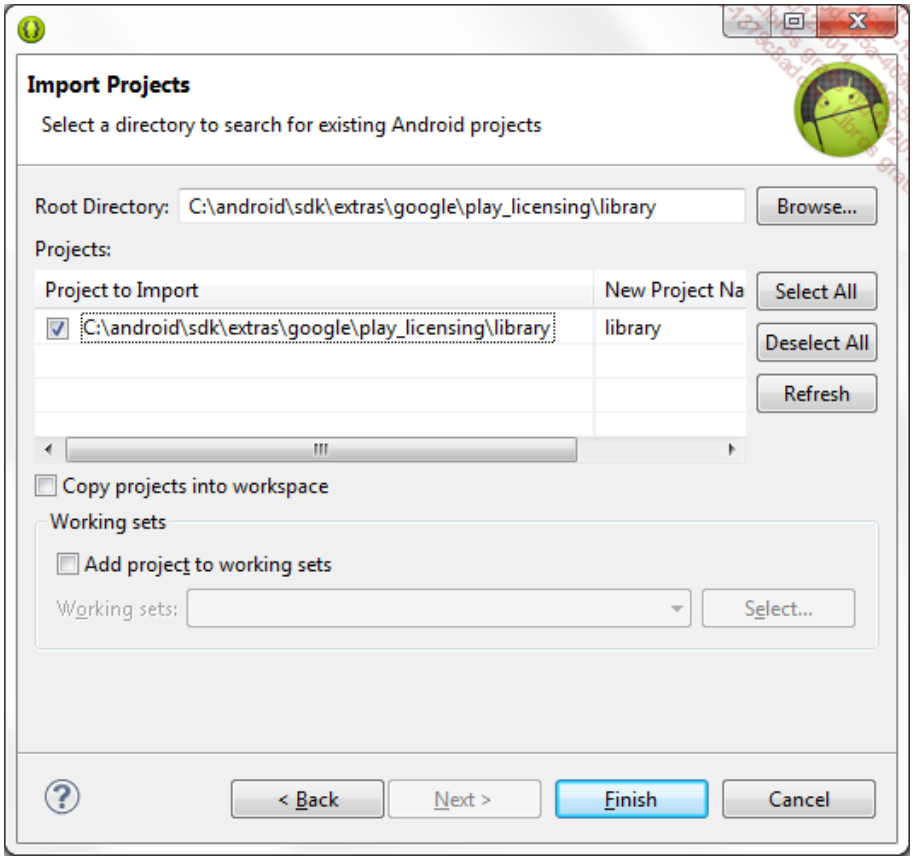
c. Integración de la LVL como librería

La segunda forma de integrar el código fuente de la LVL en un proyecto consiste en importarla como librería. Este método es algo más largo de llevar a cabo la primera vez pero permite mantener un único código fuente que puede integrarse, a continuación, en varios proyectos distintos.

Para poder modificar el módulo LVL descargado sin miedo a que sea sustituido por alguna actualización, copie la carpeta `library` de la LVL en la carpeta correspondiente al espacio de trabajo de Eclipse. Esta copia es la que modificaremos y utilizaremos.

La creación del proyecto de librería sigue el mismo principio de creación que un proyecto Android clásico aunque utiliza los archivos de código fuente provistos.

- En el asistente de creación del proyecto Android, indique un nombre de proyecto, marque la opción **Create project from existing source** e indique la carpeta `library` de la LVL.
- Seleccione una versión adecuada en el campo **Build Target**. Ésta debe ser igual o superior a Android 1.5 (API 3).

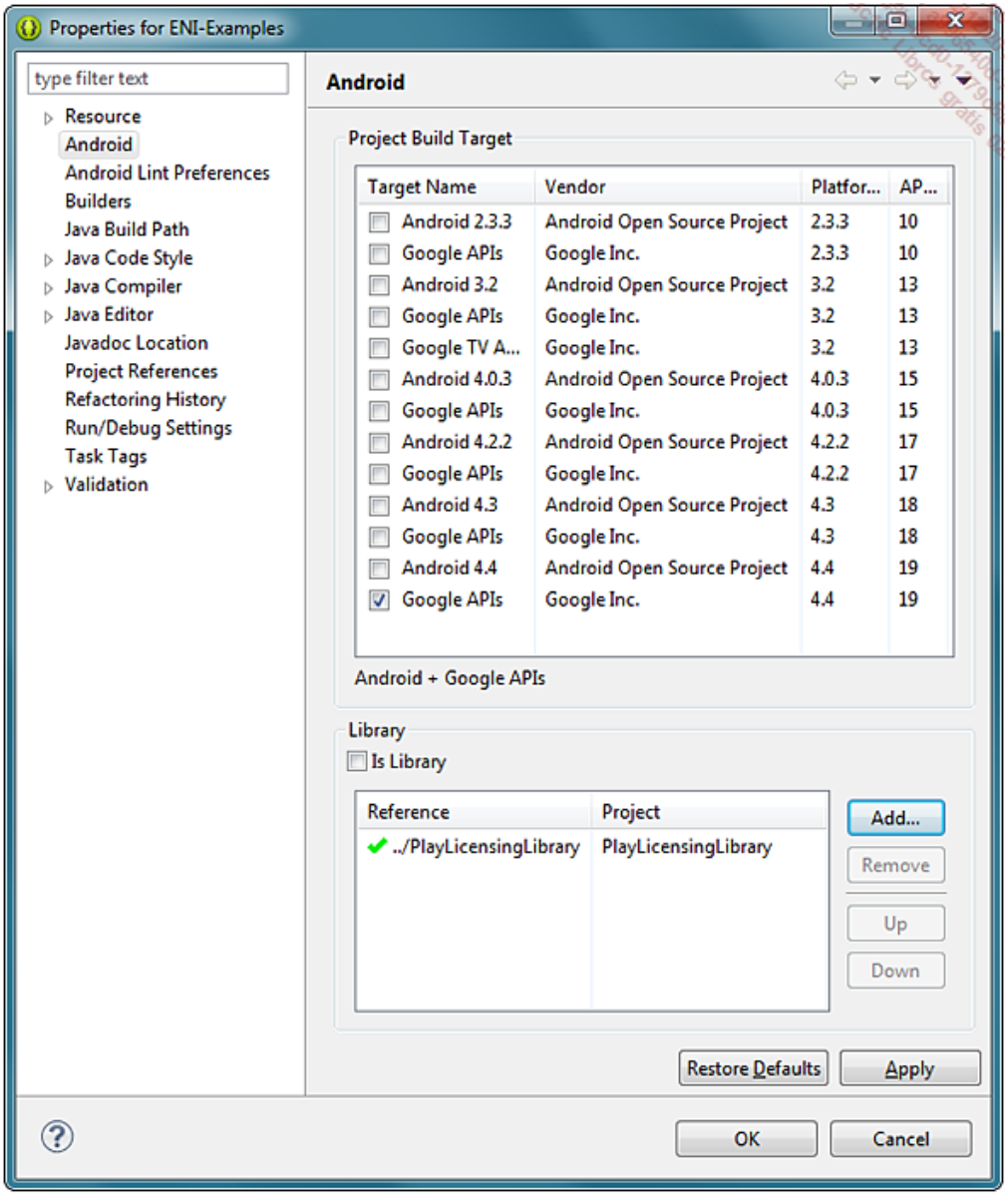


- Haga clic en el botón **Finish**.

Se ha creado el proyecto.

Ahora, hay que indicar al proyecto que contiene la aplicación que agregue una referencia a este proyecto de modo que, cuando se compile la aplicación, los archivos de código fuente de la librería se agregarán a la compilación para producir un único binario.

- En Eclipse, abra la ventana de las propiedades del proyecto de la aplicación Android. Para ello, haga clic con el botón derecho sobre el nombre del proyecto y, a continuación, **Properties**.
- Seleccione Android en la lista de la izquierda y después, en la parte derecha, haga clic en el botón **Add...** y seleccione el proyecto correspondiente a la librería.



Haga clic en el botón **OK**.

La librería queda enlazada al proyecto y se integrará en cada compilación como si los archivos de código fuente formaran parte del proyecto.

2. Uso de la LVL

Una vez descargada la LVL e integrada en su proyecto, hay que seleccionar la política de verificación de licencia para adoptar en la aplicación. Algunas de estas políticas encriptan los datos que registran de manera local. La verificación de la licencia puede realizarse a continuación.

a. Política

Cada aplicación puede escoger la estrategia que desea adoptar cuando recibe el resultado de una verificación de licencia. Esta política a aplicar debe implementarse en la aplicación.

El desarrollador puede crear sus propias políticas implementando una interfaz Java. Para ayudarle en esta tarea, Android provee dos políticas por defecto ya implementadas que basta con instanciar para dar respuesta a los casos más comunes:

- `ServerManagedPolicy`: es la política recomendada y utilizada por defecto. Se trata de una política que utiliza los parámetros de configuración reenviados por el servidor de licencia. Esta política funciona incluso sin conexión gracias a una caché segura, ofuscada. Algunos de estos parámetros de configuración son: fecha y hora de fin de validez de la caché, número máximo de reintentos de verificación de licencia antes de bloquear el acceso a la aplicación… Su constructor recibe como parámetros el contexto de la actividad en curso y un objeto de tipo `Obfuscator` descrito en el párrafo siguiente.

Sintaxis

```
public ServerManagedPolicy(Context context, Obfuscator obfuscator)
```

- `StrictPolicy`: es la política más estricta, que sólo autoriza el acceso a la aplicación si el servidor indica que la licencia es válida. La ventaja es que ofrece una mayor seguridad pues no conserva trazas de la licencia en una caché como hace la política anterior. La otra cara de la moneda es que requiere una conexión de red funcional con cada verificación, lo cual puede resultar molesto para un gran número de usuarios.

Sintaxis

```
public StrictPolicy()
```

b. Ofuscación

Ciertas políticas, como la política `ServerManagedPolicy`, requieren almacenar los datos relativos a la licencia en un almacén persistente local. Estos datos deben ser seguros, puesto que permiten determinar si el usuario tiene acceso o no a la aplicación.

Para ello, es preciso utilizar un ofuscador de datos. Como ocurre con las políticas, el desarrollador puede escoger crear su propio ofuscador o bien puede utilizar un ofuscador ya implementado provisto por Android: el `AESObfuscator`. Como su propio nombre indica, este ofuscador utiliza la encriptación AES (*Advanced Encryption Standard*, o estándar de cifrado avanzado) para proteger los datos.

Para proteger al máximo los datos, el ofuscador `AESObfuscator` utiliza valores específicos del desarrollador, de la aplicación y del dispositivo Android para generar claves de encriptación AES lo más variadas posibles.

Estos datos son:

- una tabla de veinte bytes inicializada con valores tomados al azar. Por ejemplo:

```
private static final byte[] SALT = new byte[] {
    -42, 13, -37, 5, 86, 45, -123, 102, -15, -3,
    123, 5, 42, -115, 2, 110, 25, 53, 5, -128
};
```

- una cadena de caracteres identificando a la aplicación, típicamente el nombre del paquete de la aplicación utilizando el método `getPackageName`.
- una cadena de caracteres identificando el dispositivo Android de la forma más unívoca posible como, por ejemplo, el valor de la constante:`android.Settings.Secure.ANDROID_ID`.

La creación de una instancia `AESObfuscator` se realiza mediante el uso del ofuscador. Éste recibe como parámetros los datos detallados anteriormente.

Sintaxis

```
public AESObfuscator(byte[] salt, String applicationId,
  String deviceId)
```

Ejemplo

```
String deviceId =
  Secure.getString(getContentResolver(), Secure.ANDROID_ID);
AESObfuscator obf =
  new AESObfuscator(SALT, getPackageName(), deviceId);
```

c. Verificación de la licencia

Requisito previo obligatorio para autorizar la aplicación de la que se quiere verificar la licencia, es preciso que el proyecto de la aplicación incluya el permiso de verificación de la licencia.

Para ello, hay que agregar la línea siguiente en el manifiesto:

Sintaxis

```
<uses-permission android:name="com.android.vending.CHECK_LICENSE" />
```

Ejemplo

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
    xmlns:android="http://schemas.android.com/apk/res/android"
    package="es.midominio.android.miapli"
    android:versionCode="1"
    android:versionName="1.0">
```

El uso de la LVL se realiza mediante una instancia

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

    <uses-permission
      android:name="com.android.vending.CHECK_LICENSE" />
</manifest>
```

de tipo `LicenseChecker` que pertenece al paquete `com.google.android.vending.licensing`. El constructor de la clase`LicenseChecker` recibe como parámetros el contexto de la actividad en curso, la instancia de la política escogida así como la clave pública de nuestra cuenta de desarrollador. Esta clave se recupera desde la cuenta de desarrollador de Play Store como se explica más adelante.

Sintaxis

```
public LicenseChecker(Context context, Policy policy,
  String encodedPublicKey)
```

Ejemplo

```
LicenseChecker lc = new LicenseChecker(this, policy, "MIIBI...");
```

Todo objeto de tipo `LicenseChecker` debe obligatoriamente invocar a su método`onDestroy` en el método `onDestroy` de la actividad que la ha creado.

La fase de verificación de la licencia se desarrolla en dos etapas. La primera etapa consiste en iniciar la verificación de la licencia mediante una llamada al método `checkAccess` del objeto de tipo `LicenseChecker`. Se ejecutará, generalmente, esta llamada en el método `onCreate` de la actividad principal de modo que se verifique la licencia desde el inicio de la aplicación. Este método recibe como parámetro un objeto que implementa la interfaz `LicenseCheckerCallback`. Este objeto se utiliza durante la segunda etapa: la recuperación del resultado.

Sintaxis

```
public synchronized void
  checkAccess(LicenseCheckerCallback callback)
```

La interfaz `LicenseCheckerCallback` permite gestionar todo tipo de resultado tras la verificación de la licencia. Está compuesta por los tres métodos `allow`, `dontAllow` y`applicationError` que se invocan respectivamente cuando la licencia se confirma como válida, inválida, o cuando el desarrollador ha cometido algún error en la implementación de la secuencia de verificación que impide su buen funcionamiento. El valor entero que se devuelve como parámetro de los métodos `allow` y `dontAllow` indica el motivo de la aceptación o del rechazo: el valor puede ser `Policy.LICENSED`, `Policy.RETRY` ou `Policy.NOT LICENSED`.

Sintaxis

```
public void allow(int reason)
public void dontAllow(int reason)
public void applicationError(int errorCode)
```

Es responsabilidad del desarrollador realizar, a continuación, las acciones necesarias en función del resultado. Por ejemplo, en el método `dontAllow`, el desarrollador podrá informar al usuario que su licencia no es válida, proponerle adquirir una licencia en Play Store y, si no lo desea, forzar a que la aplicación se cierre.

Los métodos de la interfaz `LicenseCheckerCallback` se invocan, generalmente, desde threads distintos al thread UI principal de la aplicación, de modo que no bloqueen la aplicación durante la fase de verificación. Por ello, el desarrollador deberá tener en cuenta no invocar directamente los métodos para actualizar la interfaz de usuario desde los métodos de la interfaz`LicenseCheckerCallback`. En su lugar, podrá utilizar un objeto de tipo `Handler` o el método`runOnUIThread` para actualizar la interfaz de usuario en el thread principal.

3. Probar

Para poder probar la LVL, debe poseer una cuenta de desarrollador en Play Store (véase el capítulo Publicar una aplicación - Publicación de la aplicación en Play Store).

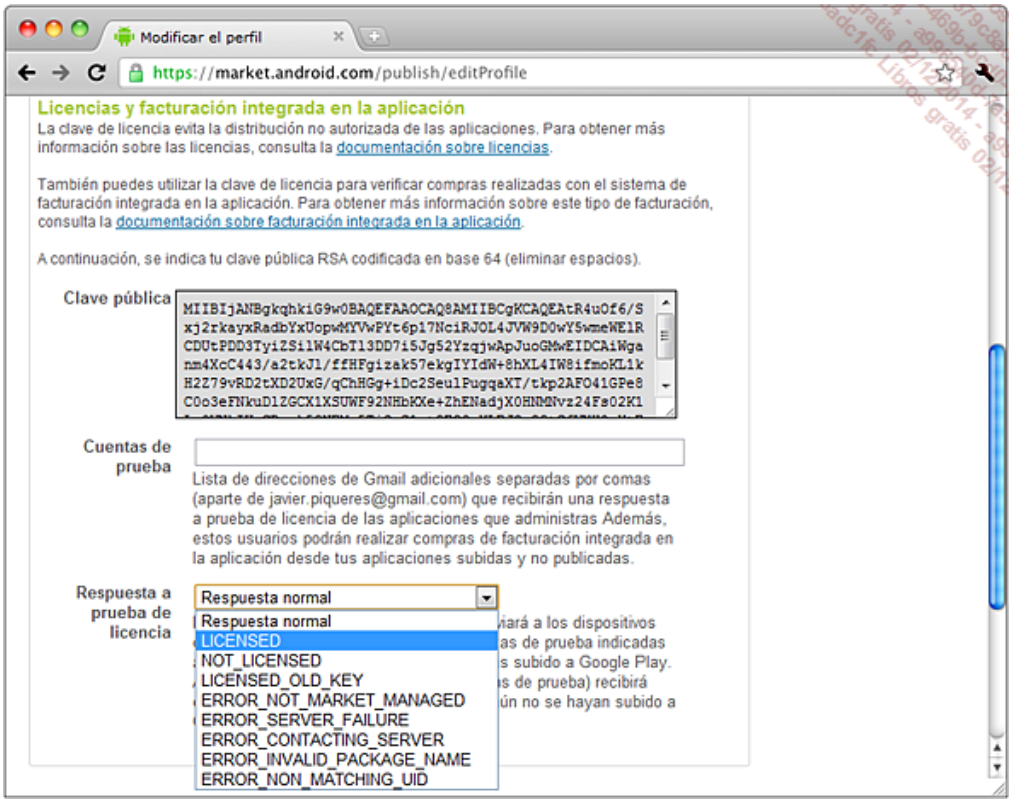
Tras la creación de la cuenta de desarrollador en Play Store, se genera automáticamente un par de claves pública/privada RSA (de los apellidos de sus tres inventores, R. Rivest, A. Shamir y L. Adleman) de 2048 bits que pertenecen al desarrollador y son válidas para todas las aplicaciones en pruebas o publicadas con su cuenta.

La clave privada se guarda, de forma secreta y segura, en Play Store. Permite firmar la respuesta del servidor de licencias y garantizar la seguridad del resultado de la verificación. La clave pública se provee durante la instanciación de objetos de tipo `LicenseChecker` en el código de las aplicaciones.

Para recuperarlo, el desarrollador debe conectarse sobre su cuenta web en Play Store.

- En la página principal de la cuenta, haga clic en **Todas las aplicaciones** en la parte superior izquierda (el icono representa BugDroid).
- Seleccione la aplicación que desea proteger.
- A continuación, seleccione la pestaña **Servicios y API** (en la parte inferior del menú asociado a la aplicación).

Aparece la parte correspondiente a las licencias y a las compras integradas llamada **Licencias y facturación integrada en la aplicación**.



La clave aparece en la zona de texto **Clave pública**. Esta cadena de caracteres es la que hay que utilizar en el código de la aplicación.

Es posible agregar, para las pruebas, cuentas llamadas **Cuentas de prueba**. Estas cuentas se definen en la pestaña **Configuración** de la consola Play Store (pestaña representada por un engranaje), y corresponden con cuentas de usuario Gmail que recibirán las respuestas de las pruebas de verificación de las licencias de las aplicaciones de esta cuenta. Estas respuestas de las pruebas se especifican en el campo **Respuesta a prueba de licencia**.

El campo **Respuesta a prueba de licencia** permite al desarrollador indicar el resultado de la prueba que quiera devolver desde el servidor de licencias para probar sus aplicaciones. Sólo las cuentas Gmail del desarrollador y de los usuarios especificados en el campo anterior obtendrán este código de prueba para las aplicaciones ya transferidas en la cuenta del desarrollador.

Solamente la cuenta del desarrollador podrá recibir este código para aquellas aplicaciones que todavía no hayan sido transferidas a su cuenta y que estén, por tanto, en fase de pruebas.

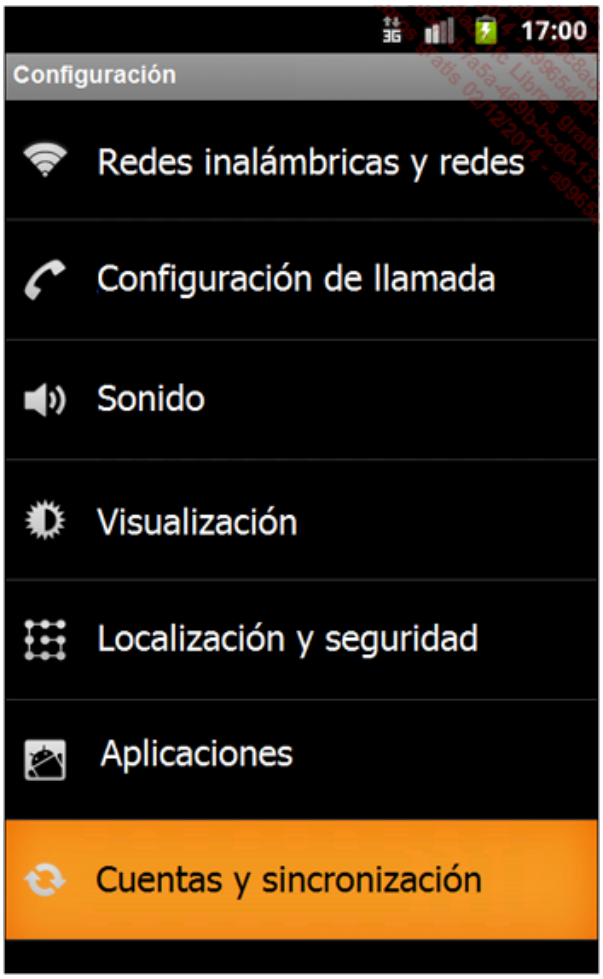
Un clic en el botón **Guardar** permite guardar las modificaciones de la página y tenerlas en cuenta.

Los valores de los campos **Cuentas de prueba** y **Respuesta a prueba de licencia** son válidos para todas las aplicaciones de la cuenta del desarrollador.

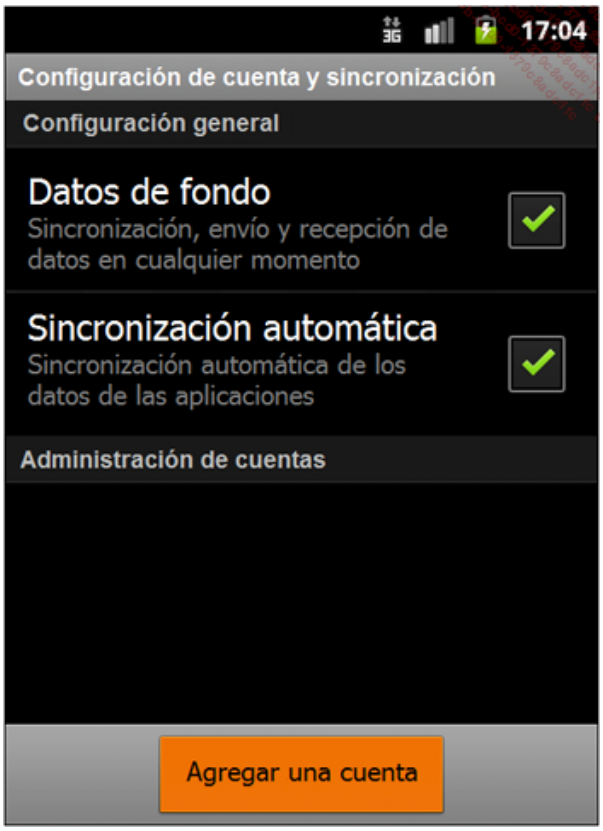
Las pruebas sobre el dispositivo o emulador requieren que se configure una cuenta de Google en el sistema Android para poder identificar al usuario y permitir realizar las verificaciones de las licencias de las aplicaciones.

He aquí las etapas que debe seguir para configurar una cuenta de Google:

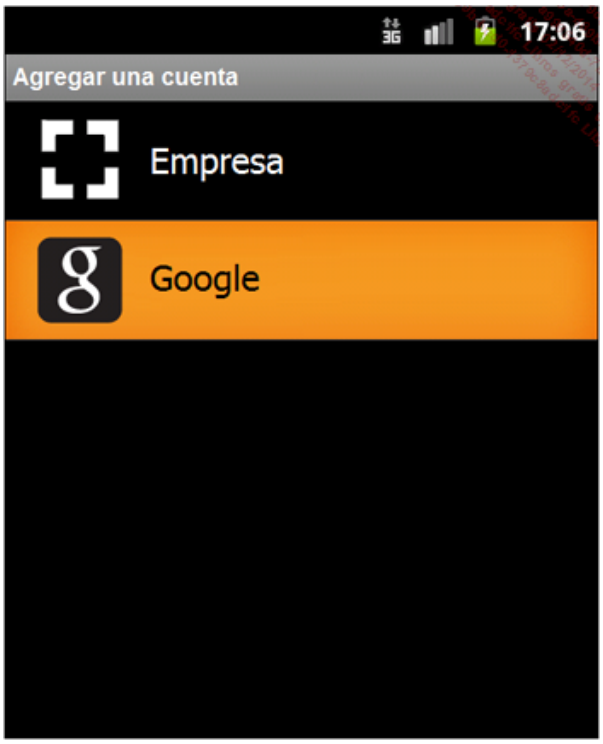
- Desde la pantalla de inicio del sistema Android, presione la tecla **Menú** o abra la lista de aplicaciones, a continuación haga clic en **Configuración**.



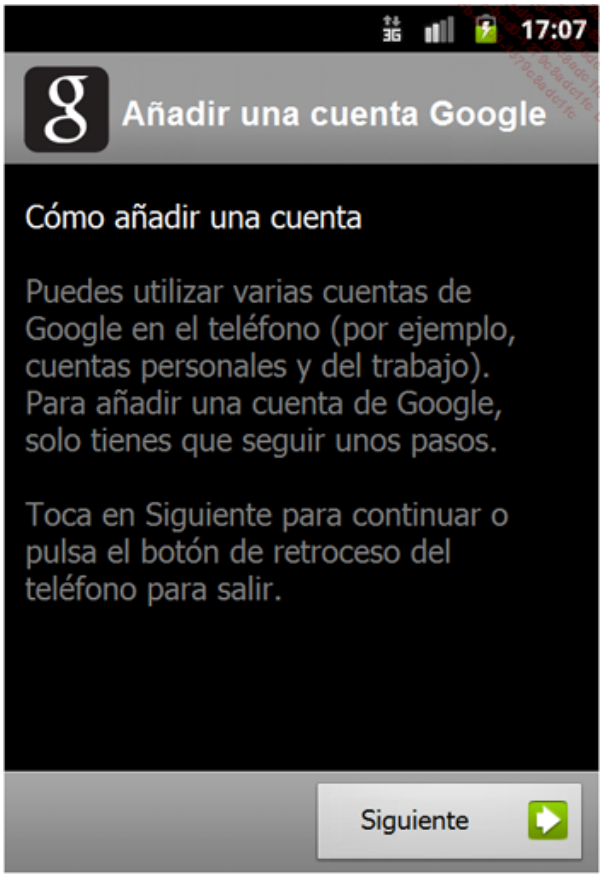
Seleccione **Cuentas y sincronización**.



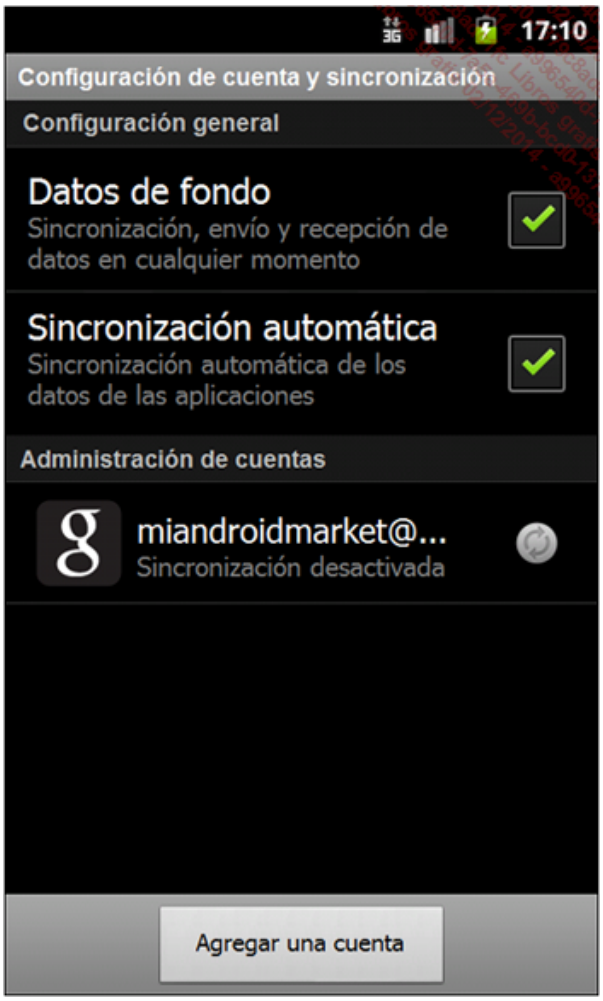
Haga clic en el botón **Agregar una cuenta**.



- Haga clic sobre la opción **Google**.



- A continuación, siga las instrucciones indicando la dirección de correo electrónico Gmail y la contraseña asociadas a la cuenta usada para crear la cuenta de desarrollador, o bien a alguna cuenta de prueba.



a. Probar sobre un dispositivo Android

Para poder probar la LVL sobre un dispositivo Android, es preciso que la versión del sistema sea la 1.5 o superior y que la aplicación Play Store esté presente, pues se comunicará con el servidor de licencias.

b. Probar sobre un emulador

Es posible probar la LVL sobre un emulador Android a condición de que el AVD utilice una imagen que incluya el módulo complementario **Google APIs by Google Inc** versión 8 release 2 o superior. Sólo estas imágenes contienen la parte de la aplicación Play Store que permite conectarse a los servidores de licencias. Y sólo estas imágenes permiten conectarse con una cuenta Google: la del desarrollador o una de las cuentas de prueba.

Recuerde, la descarga de dicha imagen se ha descrito en el párrafo correspondiente a la instalación de la LVL.

Preste atención, la verificación de la licencia no funciona sobre un emulador Google API versión 9 release 2. Esta versión no tiene en cuenta el estado de la licencia seleccionada sobre el servidor. Para más información: http://code.google.com/p/android/issues/detail?id=14252

Proponer compras integradas

Además de las aplicaciones de pago directamente en Play Store, exite otro método para monetizar nuestras aplicaciones: las compras integradas (in-app purchases, en inglés). Una aplicación que propone compras integradas permite al usuario comprar directamente, desde la aplicación, elementos suplementarios: funcionalidades, elementos de juego, etc.

Los estudios demuestran que las compras integradas son, por lo general, más apreciadas por los usuarios, que prefieren este modo de monetización a los métodos tradicionales de las aplicaciones de pago de Play Store. Un extracto de este estudio está disponible en la siguiente dirección:https://www.centrodeinnovacionbbva.com/noticias/tb/32154-lo-freemium-y-la-publicidad-in-app-modelos-lideres-en-monetizacion-de-aplicaciones

Observe que el uso del pago integrado requiere que el desarrollador de la aplicación posea una cuenta en Google Wallet. La creación de esta cuenta puede llevarse a cabo en la siguiente dirección:http://www.google.com/wallet/business/, y es completamente gratuita.

1. Preparación

Como con la protección de las aplicaciones de pago, es la aplicación Play Store instalada en el dispositivo del usuario la encargada de la comunicación entre la aplicación y los servidores de Google. Y, del mismo modo, sólo es posible implementar el pago integrado en aplicaciones publicadas en Play Store.

La comunicación con Play Store se realiza, para el pago integrado, utilizando la librería`InAppBillingService`. Esta librería la provee Google, bajo la forma de un archivo AIDL (*Android Interface Definition Language*), y se distribuye mediante el Android SDK Manager.

Utilizar directamente esta librería resulta algo complicado, y exige la implementación de numerosos elementos. Para facilitar el trabajo, Google provee un ejemplo, bien diseñado, que integra un conjunto de clases que simplifican el uso de la librería `InAppBillingService`. Vamos a utilizar estas clases para reducir la carga de trabajo.

La primera etapa consiste, entonces, en descargar el paquete **Google Play Billing Library**, disponible en la sección **Extras** del SDK Manager.

Una vez descargada y ubicada en la carpeta `<sdk>/extras/google/play_billing/`, se provee la librería `IInAppBillingService.aidl` (en la raíz de la carpeta) así como el ejemplo del que vamos a extraer las clases que nos interesan, en la carpeta `/samples/TrivialDrive/`.

Para utilizar la librería en una aplicación, basta con crear un nuevo paquete en el proyecto. Este paquete se llamará `com.android.vending.billing`. Una vez creado el paquete, es preciso copiar el archivo `IInAppBillingService.aidl`.

A continuación, copie la carpeta`TrivialDrive/src/com/example/android/trivialdrivesample/util` en el proyecto de la aplicación, en la carpeta `/src`. Conviene renombrar el paquete de las clases importadas: en Eclipse, haga clic con el botón derecho sobre la carpeta `util` copiada y seleccione la opción**Refactor** y, a continuación, **Rename**. Indique el nombre del paquete deseado. El proyecto debe compilar sin problemas: se genera un archivo `IInAppBilling-Service.java`automáticamente en la carpeta `/gen` del proyecto.

El pago integrado requiere la autorización `com.android.vending.BILLING`; es preciso, por tanto, declararlo en el manifiesto de la aplicación.

Sintaxis

```
<uses-permission android:name="com.android.vending.BILLING" />
```

Una vez agregado el permiso al manifiesto, hay que importar el APK del proyecto en la consola de desarrollador de Play Store: esta primera importación permite indicar a Play Store que la aplicación

poseerá pagos integrados, de modo que Play Store lo detecta automáticamente en la lectura del manifiesto, gracias al permiso `com.android.vending.BILLING`. Como con cualquier aplicación en Play Store, es necesario que el APK esté firmado (consulte el capítulo Publicar una aplicación).

Una vez importada la aplicación, manifiestamente definida como con pagos integrados, la consola del desarrollador permite agregar productos integrados, que podrán comprarse mediante el pago integrado: no se trata en ningún caso de productos físicos, sino de elementos que desea hacer de pago en su aplicación.

Para agregar un producto, basta con ir a la sección **Productos integrados en la aplicación** en la consola del desarrollador de la aplicación. Una vez en esta sección, haga clic en el botón **Agregar un producto**: se abre una ventana que le invita a seleccionar el tipo de producto que desea agregar y un Id de producto. La librería `IInAnBillingService` sólo es compatible, de momento, con los productos gestionados. Seleccione esta opción e introduzca un identificador para el producto. Haga clic en **Continuar**: el producto se ha creado, y el navegador presenta una ficha de producto, similar a la ficha de una aplicación. Escriba un título, una descripción y un precio. Debe, obligatoriamente, definir un precio para cada país disponible en Play Store. Como con una aplicación, existe un botón**Convertir precios automáticamente** que permite automatizar esta etapa.

Una vez introducida la información obligatoria, basta con hacer clic en Habilitar, en la zona superior de la pantalla: el producto estará activo, aunque puede existir cierto retardo antes de que esté disponible.

La siguiente etapa consiste en recuperar, en la consola de desarrollador de Play Store, la clave pública de la aplicación que utilizará el pago integrado. Es preciso ir a la sección **Servicios y API** de la aplicación, y anotar la clave de licencia: se trata de una clave pública RSA, codificada en Base64. Esta clave es válida para todo el tiempo de vida de la aplicación, y no se modifica cuando se publica una actualización.

Esta clave debe almacenarse en el proyecto que utilice el pago integrado. Se recomienda no almacenar directamente la clave, sino prever una serie de rutinas que permitan recomponerla en tiempo de ejecución de la aplicación. Un primer enfoque sugerido por Google consiste en separar esta clave pública en varias cadenas y reconstruir el conjunto para su uso.

Una vez realizadas estas operaciones, es posible utilizar el pago integrado en la aplicación.

2. Uso del pago integrado

Entre las clases importadas en el proyecto con el ejemplo de Google, la clase principal es`IabHelper`. Esta clase se encarga de realizar las principales operaciones para las compras integradas.

La sintaxis del constructor es la siguiente:

```
public IabHelper(Context context, String base64PublicKey)
```

La cadena `base64PublicKey` que se pasa como parámetro del constructor corresponde a la clave pública recuperada de la consola del desarrollador de la aplicación.

a. Iniciar la comunicación con Play Store

La primera etapa consiste en conectarse al servicio Play Store de compras integradas. Esta operación de conexión la gestiona, completamente, la instancia de `IabHelper`, mediante el método `startSetup`.

Sintaxis

```
public void startSetup(final OnIabSetupFinishedListener listener)
```

El parámetro que se pasa al método es un `Listener` que se invocará cuando se complete la configuración de la conexión. Esta interfaz contiene un método `onIabSetupFinished` que es

preciso sobrecargar.

Ejemplo

```
IabHelper billingHelper = new IabHelper(this, publicKey);
billingHelper.startSetup(new OnIabSetupFinishedListener() {
   @Override
   public void onIabSetupFinished(IabResult result) {
      if(result.isSuccess())
         Toast.makeText(getApplicationContext(), "La conexión al
servicio de billing está activa", Toast.LENGTH_SHORT).show();
      else
         Toast.makeText(getApplicationContext(), "La conexión al
servicio de billing ha fallado" , Toast.LENGTH_SHORT).show();
   }
});
```

Cuando el proceso de solicitud se termina, o cuando la actividad se destruye, es preciso invocar al método `dispose` del objeto `IabHelper`.

Ejemplo

```
@Override
public void onDestroy() {
   super.onDestroy();
   if(iabHelper!=null)
      iabHelper.dispose();
}
```

b. Obtener la lista de productos

El método `queryInventory` del objeto `IabHelper` permite obtener la lista de productos que pueden comprarse en la aplicación.

Sintaxis

```
public Inventory queryInventory(boolean detalleProductos, List<String>
idProductos)
```

El parámetro `detalleProductos` permite especificar si se desea obtener los detalles acerca de los productos (precio, descripción, etc.). El segundo parámetro es una lista de identificadores de los productos de los que se desea obtener los detalles. Cada identificador se corresponde con el que se ha indicado en la consola del desarrollador. Si no se indicara ningún identificador como parámetro, el método devolverá únicamente la lista de productos que ha comprado el usuario.

Este método se ejecuta en el thread principal, lo cual puede resultar problemático, las consultas podrían ser más o menos largas en función de la lista de productos y la calidad de la conexión de red del terminal.

También existe una versión asíncrona del método `queryInventory`, que recibe como parámetro suplementario un objeto de tipo `IabHelper.QueryInventoryFinishedListener`, cuyo método `onQueryInventoryFinished` se invoca cuando termina el procesamiento.

Sintaxis

```
public void queryInventoryAsync(final boolean detalles, final
List<String> idProductos, final QueryInventoryFinishedListener
listener)
public void onQueryInventoryFinished(IabResult resultado, Inventory
inventario)
```

Ejemplo

```
ArrayList<String> productos = new ArrayList<String>();
productos.add("producto_1");
iabHelper.queryInventoryAsync(true, productos, new Query
InventoryFinishedListener() {
   @Override
   public void onQueryInventoryFinished(IabResult result,
Inventory inventory) {
      if(result.isFailure())
         Toast.makeText(getApplicationContext(),
            "Recuperación de los ítems
errónea",Toast.LENGTH_SHORT).show();
      else
         Toast.makeText(getApplicationContext(),
            "Recuperación de los ítems
ok",Toast.LENGTH_SHORT).show();
});
```

El objeto de tipo `Inventory` devuelto por los métodos `queryInventory` contiene información acerca de los productos disponibles y comprados: el método `getSkuDetails` permite obtener información acerca de un producto, y el método `getPurchase` permite obtener información acerca de la solicitud para un producto.

Sintaxis

```
public SkuDetails getSkuDetails(String idProducto)
public Purchase getPurchase(String idProducto)
```

El objeto `SkuDetails` incluye información acerca del título, la descripción, así como el precio del elemento que se ha puesto a la venta. Esta información está accesible utilizando los métodos`get`... correspondientes.

Sintaxis

```
public String getTitle()
public String getDescription()
public String getPrice()
```

El objeto `Purchase` incluye información relativa a la solicitud, en el caso de que el producto sehaya pedido por parte del usuario.

Sintaxis

```
public String getOrderId()
public long getPurchaseTime()
public int getPurchaseState()
public String getDeveloperPayload()
```

c. Comprobar que un producto se ha solicitado

El objeto `Inventory` presenta, también, el método `hasPurchase`, que devuelve `true` si el usuario ha pedido un producto.

Sintaxis

```
public boolean hasPurchase(String idProducto)
```

Ejemplo

```
iabHelper.queryInventoryAsync(true, productos, new Query
InventoryFinishedListener() {
   @Override
```

```
   public void onQueryInventoryFinished(IabResult result,
Inventory inventory) {
      if(inventory.hasPurchase("Producto 1"))
         Toast.makeText(MainActivity.this,
         "El producto 1 se ha solicitado",
Toast.LENGTH_SHORT).show();
        }
});
```

El desarrollador deberá elaborar una estrategia eficaz para la verificación de las compras: no resulta apropiado ejecutar una verificación con cada uso de la aplicación, puesto que la verificación requiere una conexión a Internet. Es conveniente, por tanto, implementar una caché con las compras verificadas para el caso de que no exista ninguna conexión de red disponible, y verificar de manera regular las compras cuando el terminal se encuentre conectado.

d. Solicitar un producto

Play Store gestiona la totalidad del proceso de solicitud de un producto. Para el desarrollador, basta con iniciar el proceso invocando al método `launchPurchaseFlow` del objeto `IabHelper`.

La sintaxis del método es la siguiente:

Sintaxis

```
public void launchPurchaseFlow(Activity activity, String idProducto,
int codigoSolicitud, OnIabPurchaseFinishedListener listener)
```

`activity`: actividad en curso. El proceso debe ejecutarse desde el thread principal de la actividad.

`idProducto`: identificador del producto, tal y como se ha especificado en la consola de desarrollador.

`CodigoSolicitud`: código que ha escogido el desarrollador para identificar la petición. Este código lo devuelve Play Store.

`Listener`: interfaz que expone el método `onIabPurchaseFinished`, que hay que sobrecargar. La invocación de este método es algo particular y se detalla a continuación.

A diferencia del esquema clásico que encontramos en la plataforma Android, el objeto `OnIabPurchaseFinishedListener` que se pasa como parámetro del método `launchPurchaseFlow` no se invoca directamente una vez finaliza el proceso de compra en caso de éxito - ¡sino que se le invoca cuando se produce un error en el proceso!

Al margen de este punto, Play Store, cuando finaliza el proceso, invoca al método `onActivityResult` de la actividad que se ha pasado como parámetro. A continuación, en el método `onActivityResult`, hay que invocar al método `handleActivityResult` del objeto `IabHelper` para que el método `onIabPurchaseFinished` se invoque. El método `handleActivityResult` devuelve `true` si la petición se corresponde efectivamente con una compra integrada, y `false` si la llamada a `onActivityResult` es de otro origen.

Ejemplo

```
private void iniciarCompra()    {
   IabHelper.OnIabPurchaseFinishedListener mPurchaseFinished
Listener =
      new IabHelper.OnIabPurchaseFinishedListener() {
         @Override
         public void onIabPurchaseFinished(IabResult result,
Purchase info){
             if (result.isFailure()) {
                Toast.makeText(MainActivity.this,
"Se ha producido un error durante la compra",
```

```
Toast.LENGTH_LONG).show();
               return;
            }
            else
               Toast.makeText(MainActivity.this, "Compra realizada",
Toast.LENGTH_LONG).show();
         }
      };
   iabHelper.launchPurchaseFlow(this,"Producto_1", 123,
mPurchaseFinishedListener );
 }

@Override
public void onActivityResult(int requestCode, int resultCode,
Intent data) {
   if (!iabHelper.handleActivityResult(requestCode, resultCode,
data)) {
         Toast.makeText(this, "La respuesta no se corresponde con
el proceso de compra", Toast.LENGTH_SHORT).show();
         super.onActivityResult(requestCode, resultCode, data);
   }
}
```

El método `launchPurchaseFlow` posee una variante que permite agregar un tag arbitrario seleccionado por el desarrollador. Este tag se devuelve en el objeto `Purchase`, accesible mediante el método `getDeveloperPayload`.

Google recomienda utilizar este tag para verificar que la solicitud recibida se corresponde con la solicitud enviada.

Sintaxis

```
public void launchPurchaseFlow(Activity act, String sku, String
itemType, int requestCode, OnIabPurchaseFinishedListener listener,
String extraData)
```