

PROGRAMACIÓN PARA ANDROID

1

Introducción a Android



ÍNDICE

MOTIVACIÓN.....	3
PROPÓSITOS	4
PREPARACIÓN PARA LA UNIDAD	5
1. ¿QUÉ ES ANDROID?.....	7
2. UN POCO DE HISTORIA	9
3. ARQUITECTURA	11
4. VERSIONES DE ANDROID	14
5. EL PROBLEMA DE LA FRAGMENTACIÓN	16
CONCLUSIONES	19
RECAPITULACIÓN	20
AUTOCOMPROBACIÓN	21
SOLUCIONARIO.....	27
PROPUESTAS DE AMPLIACIÓN	28
BIBLIOGRAFÍA.....	30



MOTIVACIÓN



Dirígete hacia tu meta sin vacilar; nosotros te ayudamos a llegar hasta allí.

Si estás siguiendo este curso, es porque estás interesado en desarrollar aplicaciones para dispositivos móviles basados en Android.

Pero, ¿qué es Android?, ¿cómo es su arquitectura?, ¿qué versiones hay?

En este capítulo, y antes de meternos en la materia de programación propiamente dicha, daremos unas pinceladas que nos permitan tener una visión global de la plataforma.

También hablaremos de la “fragmentación”, un problema que deberemos “sufferir” como programadores Android.

PROPOSITOS

Esta unidad es una introducción enfocada a conocer un poco más a fondo la plataforma Android y al final conseguirás:

- Conocer su historia.
- Conocer su arquitectura.
- Conocer las distintas versiones.
- Hablaremos del problema de la fragmentación y de la forma en la que nos afecta a los programadores.



PREPARACIÓN PARA LA UNIDAD

No es necesaria ninguna preparación previa para seguir esta unidad.



1. ¿QUÉ ES ANDROID?

Android es un conjunto de herramientas y aplicaciones destinadas a dispositivos móviles. Está desarrollado por la Open Handset Alliance (capitaneada por Google) y sigue la filosofía de código abierto.

Incluye un sistema operativo, librerías de abstracción y aplicaciones finales.

Sus principales características son:

- Kernel basado en Linux (2.6).
- Framework de aplicaciones que permite reutilizar y reemplazar sus componentes.
- Navegador web integrado basado en Webkit.
- Gráficos optimizados 2D (librería propia) y 3D (basados en OpenGL ES).
- SQLite para almacenamiento de datos.
- Soporte multimedia para los formatos más utilizados de sonido, vídeo e imagen (MPG4, H.264, MP3, AAC, AMR, JPG, PNG, GIF).
- Soporte para telefonía GSM*.
- Soporte Bluetooth*.
- Soporte EDGE*.
- Soporte 3G*.
- Soporte Wifi*.
- Soporte para cámara*.
- Soporte GPS*.

- Soporte compás*.
- Soporte acelerómetro*.
- Gran entorno de desarrollo que incluye: documentación, emulador de dispositivos, herramientas de *debug* y análisis de uso de memoria/CPU, *plugin* para el entorno de desarrollo Eclipse y varias utilidades complementarias.

(*) Si el *hardware* del terminal lo soporta.



Figura 1. Logotipo oficial de Android

Así pues las ventajas de utilizar Android son claras:

A los fabricantes/operadora les ofrece un sistema operativo muy potente sin coste de licencias con los que sacar el máximo partido a sus terminales. Al ser código abierto y modular, permite personalizar el sistema sustituyendo los componentes del sistema que crean necesario.

A los desarrolladores una plataforma de desarrollo muy completa con la que es muy sencillo y rápido desarrollar nuevas aplicaciones. Además gracias al Android Market pueden distribuir sus aplicaciones de una forma sencilla y barata entre millones de clientes potenciales de todo el mundo.

Los usuarios finales pueden optar a teléfonos de última generación mucho más económicos que sacan el máximo partido a la conectividad a Internet, con miles de aplicaciones disponibles que se instalan de forma muy sencilla. Además pueden actualizar sus terminales con nuevas versiones (siempre que su *hardware* lo soporte) o sustituirlos por ROMs modificadas por otros usuarios.

Todo el código de Android es abierto y está liberado bajo licencia Apache v2.0 (a excepción de las modificaciones del kernel que han sido liberadas bajo la GNU GPLv2)



2. UN POCO DE HISTORIA

En julio de 2005 Google compró una pequeña empresa con sede en Palo Alto (California) llamada Android INC. Esto disparó los rumores. Se especulaba con la posibilidad de que la Google estuviera desarrollando su propio teléfono libre, independiente de las operadoras, que obtendría beneficios de la publicidad en las búsquedas de personas.

Todos estos rumores quedaron desmentidos finalmente cuando, en noviembre de 2007, se anunció la creación de *Open Handset Alliance* (OHA), un consorcio de 34 empresas del mundo de las telecomunicaciones creado para “cambiar la experiencia del uso de los teléfonos móviles”.

Entre sus filas había empresas operadoras (T-Mobile, Telefónica...), compañías de software (Google, eBay...), empresas de comercialización (Wind River Systems, Aplix...), de microchips (Intel, Nvidia, Texas Instruments...) y fabricantes de móviles (HTC, LG...).

Al mismo tiempo que se anunciaba su creación, la OHA lanzaba el “Android Software Development kit” y varios meses después (agosto de 2008) el “Android SDK 0.9 beta”.

Eran los primeros pasos de Android, un sistema operativo para móviles de código abierto diseñado para competir con: Apple (iPhone), Microsoft (Windows Mobile), Nokia (Symbian), Palm (WebOs), Research In Motion (Blackberry) y Samsung (Bada). Sin lugar a dudas un objetivo mucho más ambicioso que crear un simple móvil.

El primer terminal en salir al mercado con Android fue el HTC G1 (Dream en España) lanzado por T-Mobile en septiembre de 2008. También se lanzó una versión modificada con acceso Root para desarrolladores denominada Dev Phone-1.

A España (con unos meses de retraso) Android llegó de la mano de Telefónica con su HTC-Dream al que le siguió, pocos meses después, el HTC-Magic de Vodafone.

Desde su lanzamiento, el éxito de Android ha sido innegable, cada vez son más los terminales lanzados al mercado que engrosan sus filas, compañías como HTC o Samsung han visto como sus beneficios han aumentado considerablemente gracias al éxito de sus terminales Android.

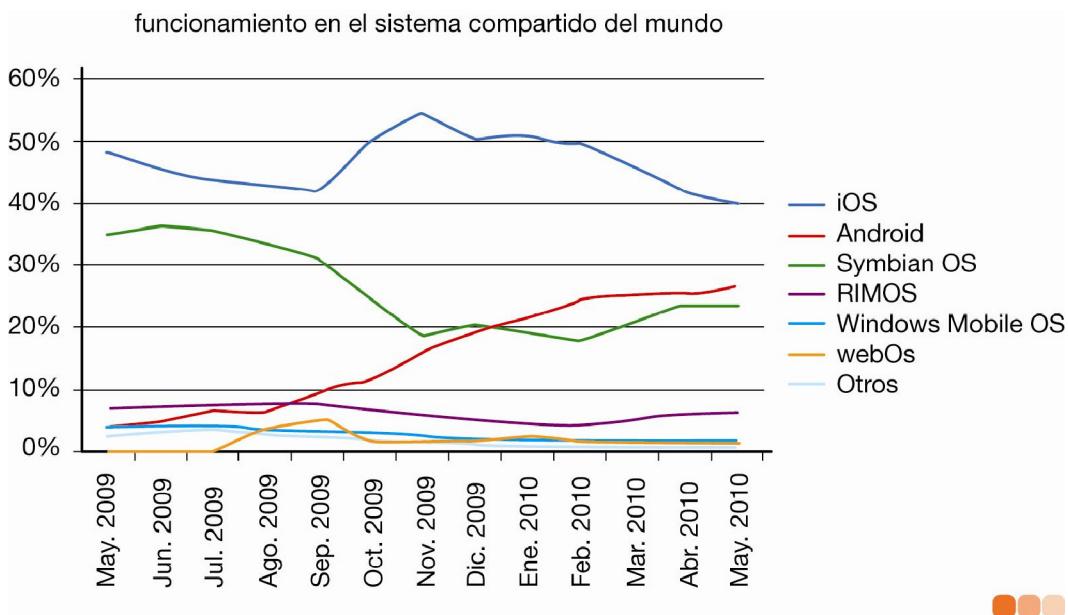


Figura 2. Uso de S.O. móviles según admob.com

El 5 de enero de 2010 Google empezó a comercializar su propio terminal libre bajo el nombre de Nexus One. Incorporaba la versión 2.1 de Android y se vendía a través de su propia web. La experiencia no fue todo lo satisfactoria que se esperaba ya que no se alcanzó el nivel de ventas deseado, y en junio de 2010 se anunció que dicho terminal se dejaría de distribuir de esta forma.

En julio de 2010 se lanzó el Dev Phone-2 un Nexus con privilegios *root* destinado a los desarrolladores de ROMs.

Android también ha hecho incursiones en *tablets* de la mano de compañías como HP, Dell, Notion INC o MSI (por citar algunos), eBooks (como el Alex de Grammata o el Nook de Barnes&Noble) e incluso la anunciada Google TV parece ser que estará basada en Android.

3. ARQUITECTURA

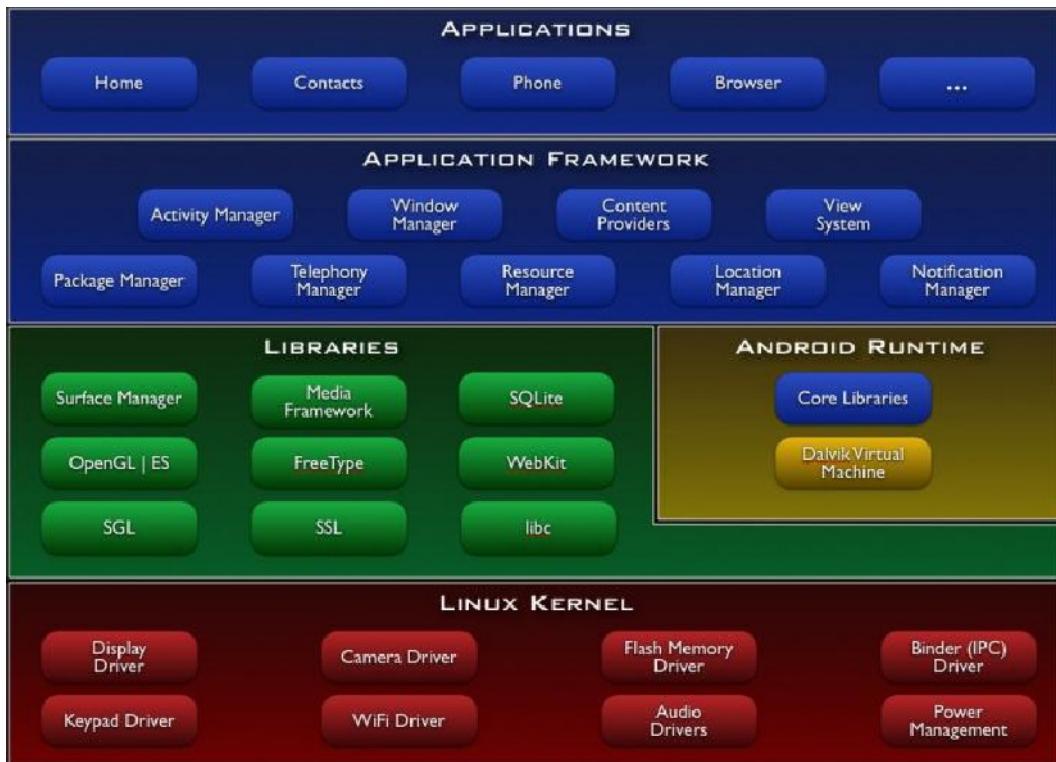


Figura 3. Esquema de la arquitectura Android. (Fuente: dev.android.com)

El kernel de Linux

Es la capa más baja del sistema. Es el encargado de interactuar con el *hardware*. Contiene todos los *drivers* necesarios para ello. Está basado en el kernel GNU Linux 2.6 al que el equipo de Google ha añadido ciertas modificaciones.

Librerías

Por encima del kernel se encuentran las librerías que incorpora el sistema. Suelen estar programadas en C/C++ y normalmente no accederemos a estas librerías directamente ya que usaremos las de la capa superior (el *framework*).

Máquina virtual (Dalvik)

Bien por evitar temas de licencias, por tratar de optimizarla o por intentar luchar con la fragmentación existente en Java ME, Google decidió su propia máquina virtual con su propio *bytecode*.

Basada en registros y no en la pila, está optimizada para el uso en dispositivos móviles que, normalmente, tienen menos *hardware* y una necesidad de ahorro de energía mayor que el resto de ordenadores.

Como las aplicaciones están escritas en Java, la compilación se realiza en dos fases: primero se llama a un compilador “normal de Java” y luego se realiza una última pasada para convertir el *bytecode* de Java a *bytecode* de Dalvik generando el ejecutable final (con extensión .dex).

Su diseño permite que se puedan ejecutar varias máquinas virtuales de forma simultánea. Así pues, cada aplicación se ejecuta en su propia máquina virtual (lo que añade un nivel más de seguridad).

Framework de aplicaciones

Facilita un marco de desarrollo que estandariza y facilita la programación de aplicaciones Android.

Estas librerías están escritas en Java y ampliamente documentadas mediante JavaDoc.

Como programador tendrá acceso completo a todos los APIs usados en las aplicaciones base que incorpora Android.

El diseño del *framework* está enfocado en la reutilización de componentes. Siempre que se sigan las reglas de seguridad que incorpora, cualquier aplicación puede publicar sus capacidades, para que luego cualquier otra aplicación pueda usarla.

Por ejemplo si necesitamos seleccionar un contacto, podremos llamar a la agenda y usar su buscador con una sola línea de código (en vez de tener que programar nuestro propio buscador).

A lo largo de los siguientes capítulos iremos profundizando en muchas de las librerías que incorpora este *framework*.



Aplicaciones

Android, por defecto, viene con una serie de aplicaciones preinstaladas: agenda de contactos, calendario, gestor SMS, navegador web, etc.

El usuario puede instalar nuevas aplicaciones (tus aplicaciones entre otras) gracias al Android Market (al que dedicaremos un capítulo más adelante) u otros programas similares.

Debido al diseño abierto, todas las “aplicaciones base” pueden ser sustituidas sin problema. Es decir, si nos gusta más otro lector de correo, solo tenemos que instalarlo.

Existe una excepción a esto, es la impuesta a partir de la versión 3.0 (octubre de 2010) que impide sustituir el *launcher* (lanzador de aplicaciones) para evitar la fragmentación.

4. VERSIONES DE ANDROID

Cada nueva versión de Android incorpora nuevas funcionalidades y soluciona problemas de las anteriores. Cada versión lleva asociado un “API Level” numérico que permite identificarla. De esta forma el sistema puede conocer si es capaz de correr o no una determinada aplicación.

Como programador deberás tener muy en cuenta esto ya que dependiendo el nivel de API sobre el que desarrolles, podrás utilizar o no ciertas funcionalidades, y limitarás la visibilidad de su aplicación (solo los terminales que corran su versión o una posterior podrán instalar y ejecutarla).

Este es un resumen de los principales cambios que se han introducido en las diferentes versiones. Para un listado completo se recomienda consultar la documentación oficial¹

Android 1.1

- Fecha: febrero de 2009.
- API Level: 2.

Estabilizaba la versión anterior y la hacía funcional. Era la versión que traía el primer terminal que salió al mercado (HTC Dream).

Android 1.5 (Cupcake)

- Fecha: mayo de 2009.
- API Level: 3.

Rediseño y mejora del interfaz de usuario, mejoras de rendimiento, nuevas funcionalidades (teclado virtual, *widgets*, *live folders*, grabación/reproducción de vídeo, Google Talk, upload a Youtube/Picasa...).

¹ <http://developer.android.com/sdk/>.



Android 1.6 (Donut)

- Fecha: octubre de 2009.
- API Level: 4.

Nuevas características de usuario (nueva caja de búsqueda, mejorada la cámara y la galería, soporte VPN, nuevos indicadores de batería, mejoras de accesibilidad), mejoras en el Android Market, nuevas tecnologías (*framework* de búsquedas mejorado, reconocimiento de voz, soporte para gestos, nuevas resoluciones de pantalla...).

Android 2.0/2.1 (Eclair)

- Fecha: enero de 2010.
- API Level: 7.

Nuevas características de usuario (mejoras en los contactos, email, mensajería, cámara, teclado virtual, calendario), nuevas tecnologías (bluetooth 2.1, nuevas APIs en el *framework*).

Android 2.2 (Froyo)

- Fecha: junio de 2010.
- API Level: 8.

Nuevas características de usuario (nueva home, soporte Exchange mejorado, cámara mejorada, *hotspot*, teclado virtual multi-idioma), mejoras de rendimiento, nuevas tecnologías de plataforma (media *framework*, *bluetooth*), nuevos servicios para desarrolladores (Cloud to Device Messaging, nuevos informes de error), nuevas APIs para programadores (almacenamiento externo de apps, media *framework*, cámara, gráficos, backup de datos, política de seguridad de dispositivo, ui *framework*).

Android 2.3 (Gingerbread)

- Fecha: noviembre de 2010.
- API Level: 9.

Mejoras en interfaz gráfica. Nuevas características del usuario.

5. EL PROBLEMA DE LA FRAGMENTACIÓN

Android es joven y está en plena fase de desarrollo, esto ha provocado que desde su lanzamiento en 2008 hasta 2010 hayan aparecido 6 versiones (Windows en sus 25 años de vida “solo” ha tenido 20). Lo que ha provocado que convivan al mismo tiempo versiones muy diferentes.

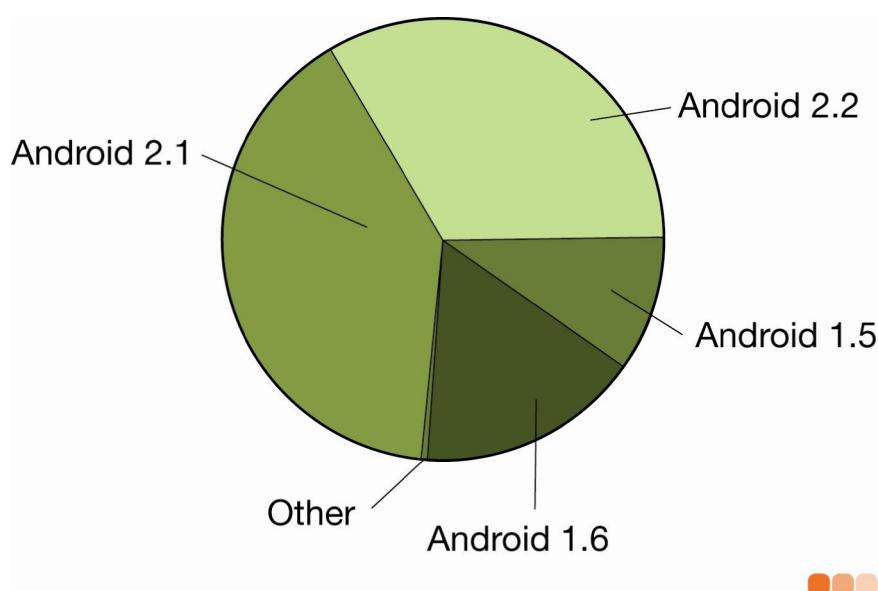


Figura 4. Uso de versiones Android a 2 de agosto de 2010. (Fuente: Google Dev)

Las operadoras han modificado el sistema añadiendo nuevas capas al interfaz de usuario (Sense, Motorblur), lo que ralentiza las actualizaciones en sus terminales (tienen que adaptar estas modificaciones antes de lanzar las actualizaciones).



Esto supone un problema para los desarrolladores, ya que no pueden sacar todo el jugo a las nuevas novedades sin renunciar a una cuota de mercado considerable.

Por otro lado, los usuarios se ven frustrados porque ven que sus terminales no se actualizan a la velocidad que les gustaría.

Para intentar atajar el problema, Google ha tomado varias medidas que empezarán a aplicarse a partir de la versión 2.3 (Gingerbread):

- Las actualizaciones de sistema se espaciarán intentando no sacar más de una al año. Para ello ciertas aplicaciones del sistema se actualizarán a través del Android Market.
- No se permitirá modificar el *launcher*, así que los interfaces de usuario personalizados como Motoblur o Sense desaparecerán y ya no se dependerá tanto de las operadoras.



CONCLUSIONES



Dirige tu futuro. En **Master.D**, te acompañamos incondicionalmente.

Android es una solución completa que permite a los fabricantes dotar a los dispositivos móviles de un sistema operativo muy potente.

Su filosofía Open Source, su completo SDK, su potencia, su facilidad de programación, su bajo coste y su facilidad para publicar aplicaciones lo convierten en una opción muy interesante para los desarrolladores.

Los usuarios finales de Android se encuentran con un sistema fácil de usar, completamente enfocado a la interconectividad, con un gran número de aplicaciones (gratuitas y de pago).

RECAPITULACIÓN

Como programador hay que tener en cuenta que Android es un sistema joven que está alcanzando su madurez. Esto nos obliga a los desarrolladores, estar al tanto de los cambios que se van produciendo en cada versión sin perder de vista el público objetivo para el que estamos creando.

Su filosofía basada en código abierto nos permite consultar las fuentes y aprender viendo como los ingenieros de Google han desarrollado cada una de las librerías y aplicaciones.



AUTOCOMPROBACIÓN

- 1. ¿Bajo qué licencia están liberadas las modificaciones del kernel realizadas por el equipo de Android?**
 - a) Apache 2.0.
 - b) GPLv2.
 - c) GPLv3.
 - d) Creative Commons.

- 2. ¿Bajo que licencia están liberadas las aplicaciones y la plataforma (no el kernel) de Android?**
 - a) Apache 2.0.
 - b) GPLv2.
 - c) GPLv3.
 - d) Creative Commons.

- 3. ¿Cómo se llamó el primer terminal lanzado al mercado con Android?**
 - a) HTC Legend.
 - b) HTC Dream.
 - c) HTC Tattoo.
 - d) HTC Desire.

4. **¿Como se llamó el primer terminal destinado al público general que Google lanzó al mercado?**
 - a) Droid.
 - b) Lexus One.
 - c) Nexus One.
 - d) Google One.
5. **¿En qué año apareció el primer SDK beta de Android?**
 - a) 2007.
 - b) 2008.
 - c) 2009.
 - d) 2010.
6. **¿Como se llama la máquina virtual que ejecuta las aplicaciones Android?**
 - a) JavaME.
 - b) Java.
 - c) Java Hotspot.
 - d) Dalvik.
7. **¿Quién es el responsable del desarrollo de Android?**
 - a) Open Handset Alliance.
 - b) Google.
 - c) T-Mobile.
 - d) Sun.
8. **¿Qué extensión tienen los archivos que ejecuta la máquina virtual de Android?**
 - a) .exe.
 - b) .dex.
 - c) .jar.
 - d) .bin.

**9. ¿A qué se refiere el término fragmentación de Android?**

- a) A dividir el código en módulos pequeños para reutilizarlo.
- b) Al problema derivado de tener varias versiones de Android en el mercado.
- c) A la posibilidad de sustituir módulos de sistema.
- d) A la cantidad de fabricantes que incorporan Android en sus terminales.

10. ¿Para qué entorno de desarrollo es el *plugin* que incorpora el SDK?

- a) Netbeans.
- b) JBuilder.
- c) BlueJ.
- d) Eclipse.

11. ¿Qué nombre recibe la versión 1.5?

- a) Froyo.
- b) Cupcake.
- c) Eclair.
- d) Donut.

12. ¿Qué nombre recibe la versión 1.6?

- a) Froyo.
- b) Cupcake.
- c) Eclair.
- d) Donut.

13. ¿Qué nombre recibe la versión 2.1?

- a) Froyo.
- b) Cupcake.
- c) Eclair.
- d) Donut.

14. ¿Qué nombre recibe la versión 2.2?

- a) Froyo.
- b) Cupcake.
- c) Eclair.
- d) Donut.

15. ¿Qué motor de base de datos incorpora Android?

- a) MySQL.
- b) Postgres.
- c) SQLite.
- d) Access.

16. ¿En Android se puede sustituir la agenda de contactos por otra aplicación?

- a) Sí.
- b) No.
- c) Depende de la versión.
- d) Depende del fabricante.

17. ¿Qué motor utiliza el navegador de Android?

- a) KHTML.
- b) Webkit.
- c) HTML.
- d) Gecko.

18. ¿En qué se basa el soporte 3D de Android?

- a) Mesa 3D.
- b) OpenGL.
- c) LWJGL (*Lightweight Java Game Library*).
- d) Librería propia.



19. ¿En que se basa el soporte para gráficos 2D?

- a) Swing.
- b) AWT.
- c) Open2D.
- d) Librería propia.

20. ¿Cómo se llama la aplicación que permite descargar e instalar nuevas aplicaciones en nuestro terminal?

- a) Android Market.
- b) Android Installer.
- c) Android Downloader.
- d) Android Publisher.



SOLUCIONARIO

1.	b	2.	a	3.	b	4.	c	5.	b
6.	d	7.	a	8.	b	9.	b	10.	d
11.	b	12.	d	13.	c	14.	a	15.	c
16.	a	17.	b	18.	b	19.	d	20.	a



Descúbrete a ti mismo, conviértete en un P8.10.

PROPUESTAS DE AMPLIACIÓN

Es recomendable leer los detalles completos de los cambios que se han introducido en cada una de las diferentes versiones.

Pueden ser consultados en la notas oficiales (<http://developer.android.com/sdk/>).

BIBLIOGRAFÍA

- <http://developer.android.com/guide/basics/what-is-android.html>.
- <http://www.openhandsetalliance.com/>.
- <http://source.android.com/> (código fuente de Android).
- <http://www.androphones.com/> (base de datos de terminales que usan Android).
- <http://metrics.admob.com/> (estudio de telefonía móvil).

PROGRAMACIÓN PARA ANDROID

2

Entorno de trabajo



ÍNDICE

MOTIVACIÓN.....	3
PROPÓSITOS	4
PREPARACIÓN PARA LA UNIDAD	5
1. ARRANCANDO ECLIPSE	7
2. LA PERSPECTIVA DE EDICIÓN (JAVA).....	9
3. EL EMULADOR Y LA PERSPECTIVA DDMS	11
CONCLUSIONES	15
RECAPITULACIÓN	16
AUTOCOMPROBACIÓN	17
SOLUCIONARIO.....	23
PROPUESTAS DE AMPLIACIÓN	24
BIBLIOGRAFÍA.....	25



MOTIVACIÓN



Dirígete hacia nosotros, y nosotros te guiaremos.

Eclipse es IDE¹ de código abierto multiplataforma. Inicialmente fue desarrollado por IBM que liberó el código que es gestionado actualmente por la fundación sin ánimo de lucro Eclipse.org.

El núcleo de este entorno se denomina RCP², su diseño está pensado para poder ser utilizado como base en desarrollos de nuevos IDEs específicos para cada lenguaje de programación (Eclipse para Java, Eclipse para C/C++, Eclipse para PHP...).

Eclipse está muy presente en los entornos de desarrollo de las empresas y es la opción recomendada con Google para desarrollar aplicaciones Android.

En nuestro caso vamos a usar un Eclipse para Java al que le hemos añadido un *plugin* para integrar las características específicas de Android y que integra el emulador con el IDE.

En los siguientes capítulos vamos a aprender las características de este IDE que nos van a resultar muy útiles para crear aplicaciones Android.

1 Entorno de Desarrollo Integrado.

2 *Rich Client Platform* o plataforma de cliente enriquecido.

PROPOSITOS

- Instalar y configurar Eclipse.
- Instalar el Android SDK.
- Conocer los componentes principales del entorno de trabajo.



PREPARACIÓN PARA LA UNIDAD

Aunque se pueden desarrollar aplicaciones usando otros editores (como Netbeans, etc.), vamos a utilizar Eclipse³ por ser el entorno soportado por la plataforma.

Así pues, antes de continuar es necesario que en el ordenador de trabajo tengas instalado:

- Eclipse⁴ (Eclipse IDE para Java, Eclipse IDE para java EE, Eclipse para RPC/plug-in o Eclipse clasic).
- Plugin eclipse JDT (incluido normalmente con Eclipse).
- Java development kit⁵: JDK5 o JDK 6 (no JRE).
- Android SDK⁶.
- ADT plugin para Eclipse⁷.

Todas estas herramientas están disponibles para Windows, Mac y Linux (consulte la documentación⁸ de Android para ver los requisitos de sistema).

3 Para este curso se ha utilizado Eclipse para Java + Java SE en un entorno Mac.

4 <http://eclipse.org/>.

5 <http://oracle.com/technetwork/java/javase/downloads/>.

6 <http://developer.android.com/sdk/>.

7 <http://developer.android.com/sdk/eclipse-adt.html>.

8 <http://developer.android.com/sdk/requirements.html>.



1. ARRANCANDO ECLIPSE

Lo primero que nos pregunta Eclipse al arrancar es el *workspace* a usar. Un *workspace*⁹ es una carpeta en la que se van a guardar los proyectos y las preferencias del entorno de trabajo.

Gestor de dispositivos virtuales Android (AVD)

Una vez elegida nuestra carpeta de trabajo debemos configurar los dispositivos virtuales Android donde probaremos nuestro código.

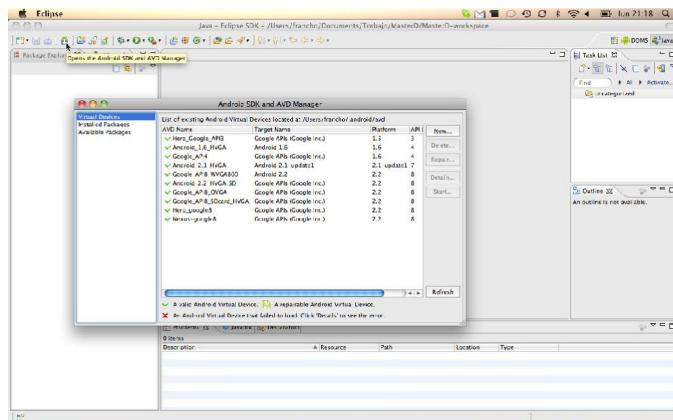


Figura 1. Eclipse, gestor de dispositivos virtuales de Android

Para ello abriremos el Gestor AVD pulsando en el icono de Android que tenemos en la barra de herramientas¹⁰.

⁹ Nosotros hemos elegido una carpeta llamada “MasterD-workspace” dentro de nuestra carpeta “Documentos”.

Tras actualizar las los paquetes disponibles añadiremos las configuraciones que necesitemos.

Es una buena idea añadir varias diferentes para probar cómo funcionan nuestros programas en los diferentes dispositivos.

En nuestro caso, a pesar de desarrollar para Android 2.2, probaremos cómo se comportan en Android 1.5. También se pueden crear diferentes plataformas para probar las diferentes resoluciones de pantalla.

Perspectivas y vistas

Eclipse trabaja con perspectivas. Por defecto, al abrirlo entraremos en la edición en la que se nos muestran las vistas del navegador de archivos, el editor, la lista de tareas...

Podemos añadir o quitar vistas a una perspectiva desde el menú de “View”.

Tenemos una serie de perspectivas predefinidas que ordenan las vistas dependiendo de la actividad que estemos realizando. Así pues, tendremos la vista de edición (Java) que usaremos cuando piquemos código, la de *debug* facilita la depuración, etc.

Nosotros vamos a utilizar principalmente dos: la perspectiva de edición Java y la perspectiva DDMS (propia de Android).

10 Si no lo hemos hecho antes, nos pedirá que configuremos la ruta en la que se encuentra instalado el Android SDK dentro del menú de preferencias.



2. LA PERSPECTIVA DE EDICIÓN (JAVA)

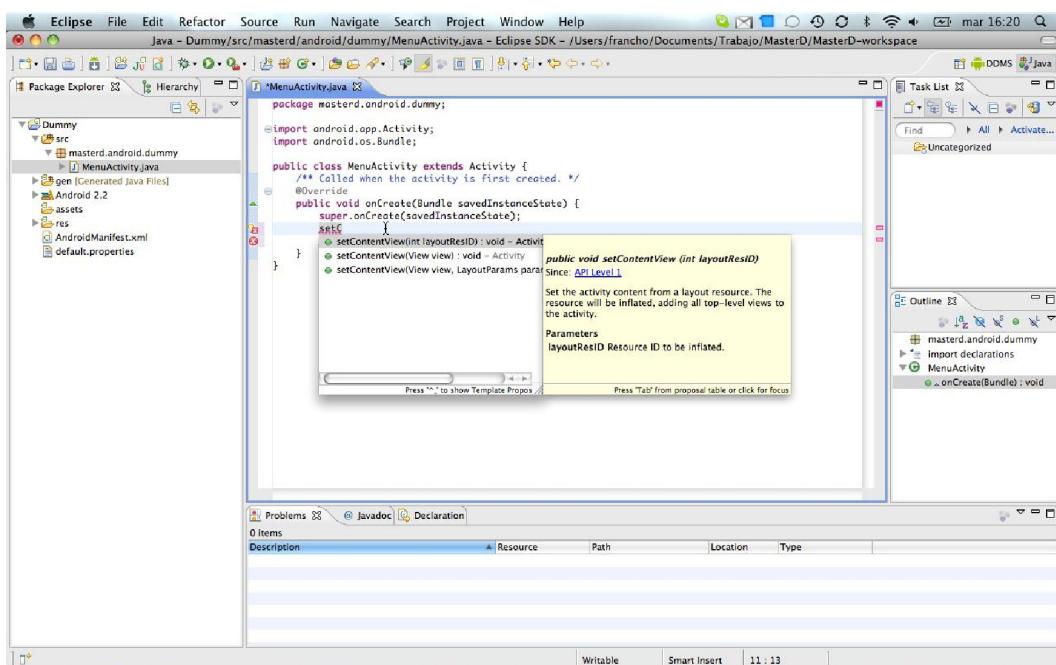


Figura 2. Editando código con Android

La parte central de esta perspectiva está ocupada por la vista del editor donde se abrirán los ficheros de clases y *xml* de configuración (entre otros). Incorpora funciones que nos facilitan la vida como autocompletado de código, generación de importes automáticos, ayuda contextual (*javadoc*).

Aparte del editor, también usaremos las siguientes vistas:

■ *Package explorer*

Esta pestaña permite navegar por los proyectos abiertos para acceder directamente a las clases y demás ficheros que lo componen.

■ *Problems*

Muestra los errores y *warnings* del proyecto. Pulsando sobre ellos podremos acceder directamente a la línea de código que los ha generado. También tiene opciones para ayudar a corregir dichos errores.

■ *Javadoc*

Muestra la documentación disponible para la clase o método sobre la que estamos.

■ *Outline*

Nos ofrece un resumen esquemático del contenido del fichero que estamos trabajando. Permite navegar por él de forma más rápida (por ejemplo al pulsar sobre el nombre de un método accederemos a él directamente).

3. EL EMULADOR Y LA PERSPECTIVA DDMS



Figura 3. El emulador de Android.

Para probar y depurar nuestras aplicaciones tenemos dos opciones: o instalarlas en un terminal o en el emulador.

El emulador nos permite probarlas “sin miedo a romper nada” además de ver como funcionan en distintas versiones de sistema y configuraciones de hardware (distintas configuraciones de pantalla, simular posiciones GPS, etc.).

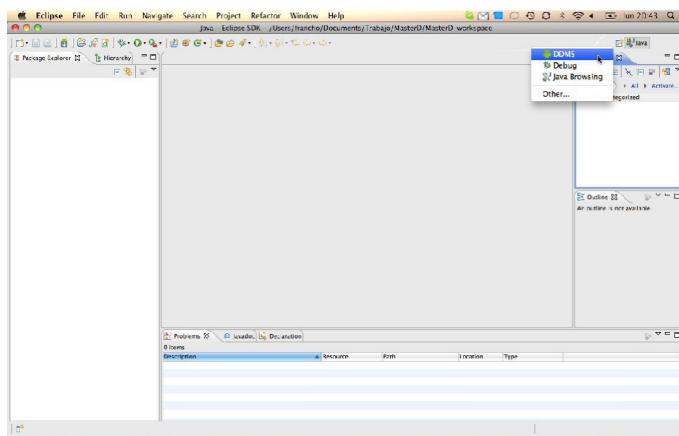


Figura 4. Eclipse, añadiendo la pestaña de depuración

El *plugin* de Android para Eclipse nos provee de una perspectiva especial en la que se agrupan todas las vistas necesarias para interactuar con a máquina virtual y observar como se comportan nuestros programas. Esta perspectiva se llama *DDMS (Dalvik Debug Monitor Service)*.

Como la vamos a usar mucho la vamos a añadir como pestaña para poder cambiar rápidamente de modo edición a modo depuración con un solo clic (este paso es opcional, siempre podremos acceder a ella a través de los menús).

Veamos las principales vistas que componen esta perspectiva de trabajo:

■ Devices

La primera pestaña que nos encontramos en esta perspectiva es la de “Dispositivos”. En ella se listan todos los emuladores y/o terminales conectados.

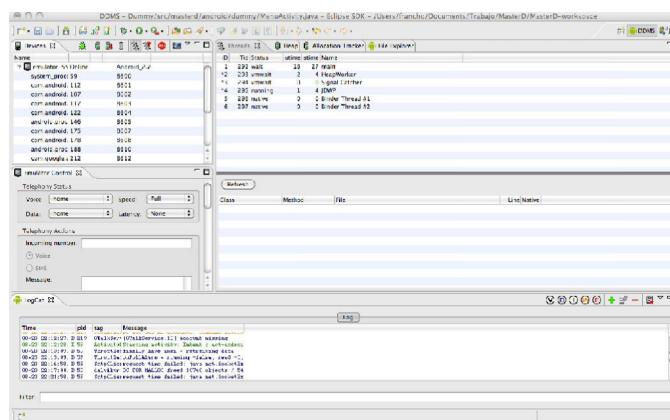


Figura 5. Eclipse, pestaña DDMS (*Dalvik Debug Monitor Service*)

Con su barra de herramientas nos permite comprobar el estado del debugger, activar la vista de threads, parar procesos o tomar capturas de pantalla.



- *Emulator control*

Permite simular llamadas, entrada de SMS, cambiar los parámetros de red, enviar una posición GPS, etc.

- *LogCat y Output*

La pestaña de “LogCat”, es una de las más importantes ya que nos permitirá consultar los mensajes de *debug* que incluyamos en nuestros programas como veremos más adelante.

“Output”, por otro lado, muestra mensajes de funcionamiento, como por ejemplo cuando se lanza el emulador, etc.

- *File Explorer, Heap and Threads*

La pestaña del explorador de archivos nos permite examinar las carpetas y ficheros del dispositivo/emulador, permitiendo también modificarlos

En la de “heap” podemos observar el estado de la pila, y en la de “threads” los hilos de ejecución.



CONCLUSIONES



Diferencia es igual a triunfo profesional.

Eclipse es un IDE de desarrollo ampliamente adoptado por las empresas, su arquitectura permite ampliarlo mediante la instalación de plugins.

Con el *plugin* de Android instalado, obtendremos un entorno de trabajo muy potente en el que podremos, además de teclear el código de forma asistida, probar el resultado del mismo gracias al emulador y las herramientas de *debug*.

Conviene perder un poco de tiempo aprendiendo a manejar Eclipse ya que esto nos facilitará mucho las cosas cuando nos metamos a programar.

RECAPITULACIÓN

Tras este capítulo deberías tener tu ordenador con todas las herramientas instaladas y estar listo para empezar a programar Android.



AUTOCOMPROBACIÓN

1. ¿Qué es Eclipse?

- a) Un entorno de desarrollo integrado.
- b) Un entorno de diseño referencial.
- c) Un entorno de composición lineal.
- d) Un entorno de dibujo fractal.

2. ¿Con qué trabajaremos en Eclipse?

- a) Perspectivas y vistas.
- b) Vistas y sentencias.
- c) Perspectivas visuales.
- d) Ninguna de las anteriores.

3. ¿Qué significa SDK?

- a) Software Development Kit.
- b) Source Development Kit.
- c) SDK Development Kit.
- d) Soft Development Kit.

4. ¿Qué es el AVD?

- a) Es un gestor de dispositivos virtuales.
- b) Es el encargado de gestionar los proyectos en Eclipse.
- c) Es un gestor interno de Android.
- d) Es el proceso que gestiona la memoria intermedia.

5. ¿Cómo podríamos gestionar un emulador?

- a) Con DDMS.
- b) Con AVD.
- c) Con Eclipse.
- d) Con Linux.

6. ¿Qué es una perspectiva en Eclipse?

- a) La forma de indentar el código.
- b) Un conjunto de estilos y colores que definen el aspecto del IDE.
- c) Es una agrupación de editores y/o vistas de manera que faciliten una determinada tarea.
- d) Eclipse no usa perspectivas.

7. ¿A qué hace referencia el acrónimo DDMS?

- a) *Debug Display Manager System*.
- b) *Dod Discovery Metadata Specification*.
- c) *Dalvik Data Manager Specification*.
- d) *Dalvik Debug Monitor Service*.

8. ¿Qué ventana o vista usaremos para ver la salida de *debug* de nuestros terminales o emuladores?

- a) *Devices*.
- b) *Logcat*.
- c) *Output*.
- d) *Debug*.



9. ¿Qué ventana o vista usaremos para ver la mensajes derivados del funcionamiento del emulador (no de las aplicaciones)?
- a) *Devices*.
 - b) *Logcat*.
 - c) *Output*.
 - d) *Debug*.
10. ¿Dónde podemos probar nuestras aplicaciones?
- a) Solo en el emulador.
 - b) Solo en un terminal conectado al ordenador.
 - c) En el emulador o en un terminal conectado al ordenador.
 - d) En una terminal.
11. ¿Dónde podemos ver los hilos de ejecución de nuestra aplicación?
- a) *Heap*.
 - b) *Treads*.
 - c) *Logcat*.
 - d) *Devices*.
12. ¿Cómo se llama el núcleo del editor Eclipse?
- a) RCP.
 - b) CRT.
 - c) CPR.
 - d) CPT.
13. ¿En qué vista tenemos que fijarnos para ver qué errores tenemos que corregir en la aplicación?
- a) *Logcat*.
 - b) *Problems*.
 - c) *Output*.
 - d) *Editor*.

14. ¿Qué es el *workspace*?

- a) La carpeta de trabajo en la que se van a almacenar los proyectos y las preferencias de los mismos.
- b) La zona de edición.
- c) El espacio de memoria reservado para el emulador.
- d) La carpeta en la que se almacenan los ficheros temporales de compilación.

15. ¿Qué muestra la ventana *outline*?

- a) Una navegación rápida sobre el fichero que editamos.
- b) Una ventana de edición de pantallas.
- c) Las tareas pendientes.
- d) Las variables desechadas.

16. ¿Dónde podemos consultar la documentación Javadoc?

- a) Solo en la ayuda contextual.
- b) Solo en la vista Javadoc.
- c) En la ayuda contextual y en la vista Javadoc.
- d) Solo en un navegador externo.

17. ¿Con qué vista podemos abrir rápidamente todos los ficheros que componen nuestra aplicación?

- a) *Outline*.
- b) *Package Explorer*.
- c) *Files*.
- d) *Project*.

18. ¿Qué requisitos son necesarios para poder desarrollar aplicaciones con el IDE de Eclipse.org?

- a) Eclipse + Plugin JDT + Java JDK + Android SDK + plugin ADT.
- b) Eclipse + Plugin JDT + Java JRE + Android SDK + plugin ADT.
- c) Eclipse + Java JDK + Android SDK + plugin ADT.
- d) Eclipse + Java JRE + Android SDK + plugin ADT.



19. ¿Desde qué vista podremos tomar una captura de pantalla de nuestra aplicación?

- a) No se puede, usaremos la tecla “impr pant”.
- b) *Logcat*.
- c) *Output*.
- d) *Devices*.

20. ¿Desde qué vista podemos simular posiciones GPS en el emulador?

- a) *Devices*.
- b) *Logcat*.
- c) *Console*.
- d) *Emulator Control*.



SOLUCIONARIO

1.	a	2.	a	3.	a	4.	a	5.	a
6.	c	7.	d	8.	b	9.	c	10.	c
11.	b	12.	a	13.	b	14.	a	15.	a
16.	c	17.	b	18.	a	19.	d	20.	d



Diferénciate del resto; ser un P8.10 tiene su recompensa.

PROPUESTAS DE AMPLIACIÓN

Visualiza en Youtube alguno de los numerosos vídeos de instalación del SDK de Android y Eclipse.



BIBLIOGRAFÍA

- Eclipse.org:
 - <http://eclipse.org/>.
- Android SDK:
 - <http://developer.android.com/sdk/index.html>.
- Máquina virtual Android:
 - <http://es.wikipedia.org/wiki/Dalvik>.

PROGRAMACIÓN PARA ANDROID

3

“Hola Mundo”



Escuela Universitaria
de Formación Abierta

master.D
GRUPO



ÍNDICE

MOTIVACIÓN.....	3
PROPÓSITOS	4
PREPARACIÓN PARA LA UNIDAD	5
1. CREANDO EL PROYECTO	7
2. COMPONENTES DEL PROYECTO	9
3. "HOLA MUNDO" AL DETALLE.....	12
4. PROBANDO NUESTRA APLICACIÓN	15
CONCLUSIONES	17
RECAPITULACIÓN	18
AUTOCOMPROBACIÓN	19
SOLUCIONARIO.....	25
PROPUESTAS DE AMPLIACIÓN	26
BIBLIOGRAFÍA.....	27



MOTIVACIÓN



Distinción, calidad, implicación, inmediatez, cercanía, personalización... Cuenta con nosotros.

Los programas “Hola Mundo” o “Hello World” suelen ser la primera toma de contacto con un lenguaje de programación. Son programas sencillos, que simplemente muestran dicha frase en la pantalla.

En este capítulo crearemos nuestro propio “Hola Mundo” en versión Android.

PROPOSITOS

Conocer la estructura básica de un proyecto Android y entender su organización.



PREPARACIÓN PARA LA UNIDAD

Para el seguimiento de esta unidad es necesario tener instalado y configurado correctamente el entorno de trabajo descrito en el capítulo anterior.



1. CREANDO EL PROYECTO

Tras arrancar el Eclipse al seleccionar nuestro directorio de trabajo, elegiremos el menú “archivo”/“nuevo”/“proyecto android”.

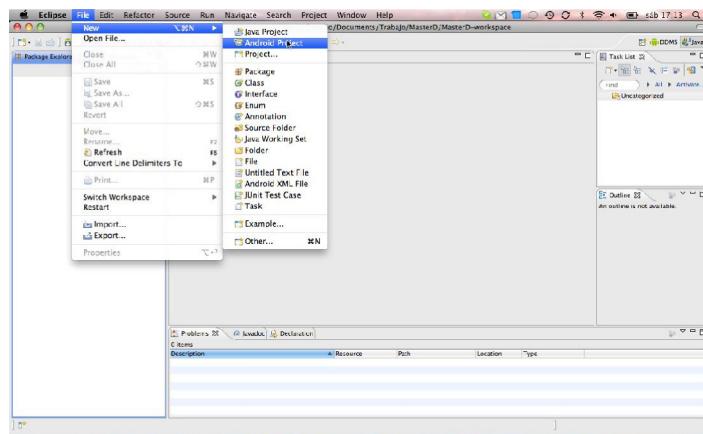


Figura 1. IDE Eclipse.

Se mostrará un diálogo emergente en el que configuraremos los datos básicos del proyecto:

- Primero deberemos poner el nombre que hemos elegido para el proyecto. En este caso “HolaMundo”.
- En el apartado de contenido, dejaremos las opciones por defecto (crear un proyecto nuevo en la ubicación predeterminada).
- Lo siguiente es escoger la versión de Android a la que está enfocado, por ejemplo Android 2.2.

- En las propiedades deberemos teclear el nombre de aplicación que queremos que se muestre en el menú del terminal. Para este proyecto usaremos “Hola Mundo”.
- El nombre de paquete se utiliza como espacio de nombres y estructura de organización de código (como los *packages* de Java). En este caso “es.masterd.holamundo”.
- Finalmente, seleccionaremos la opción de “Create Activity” para que Eclipse nos cree la clase que se lanzará por defecto al arrancar la aplicación¹. La llamaremos MainActivity.
- Una vez completado el formulario pulsaremos el botón de finalizar para completar este paso.

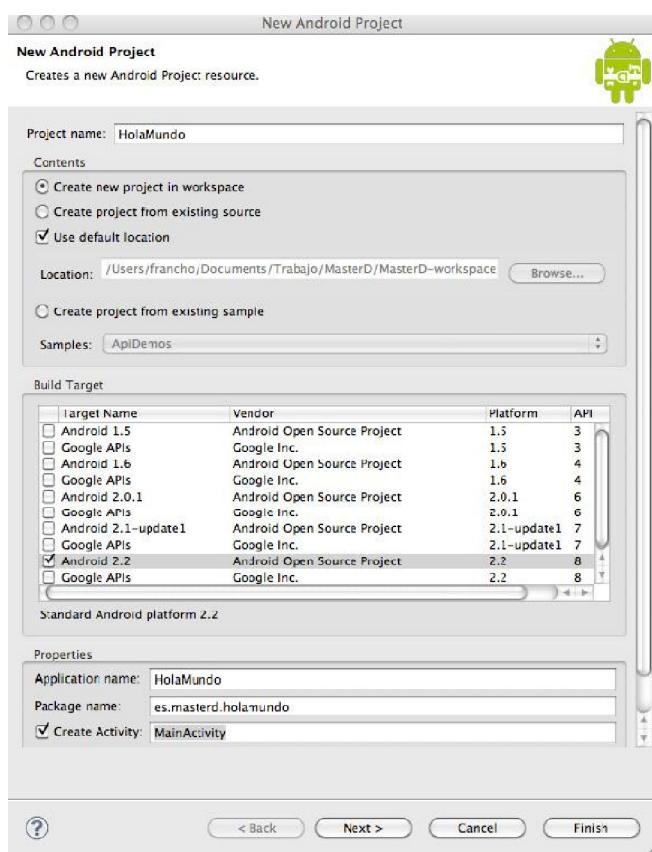


Figura 2. Nuevo proyecto Android.

¡¡Felicidades, acabas de crear tu primer proyecto Android!!

1 Se recomienda utilizar las reglas de estilo de Java y nombrar las clases empezando por mayúsculas. También es una buena idea poner un sufijo para agrupar clases del mismo tipo. En este caso se ha usado el sufijo Activity porque nuestra clase heredará de esta.



2. COMPONENTES DEL PROYECTO

Los proyectos de Android siguen una estructura fija de carpetas que debemos mantener.

Podemos observarla fácilmente usando la vista de “Package Explorer”.

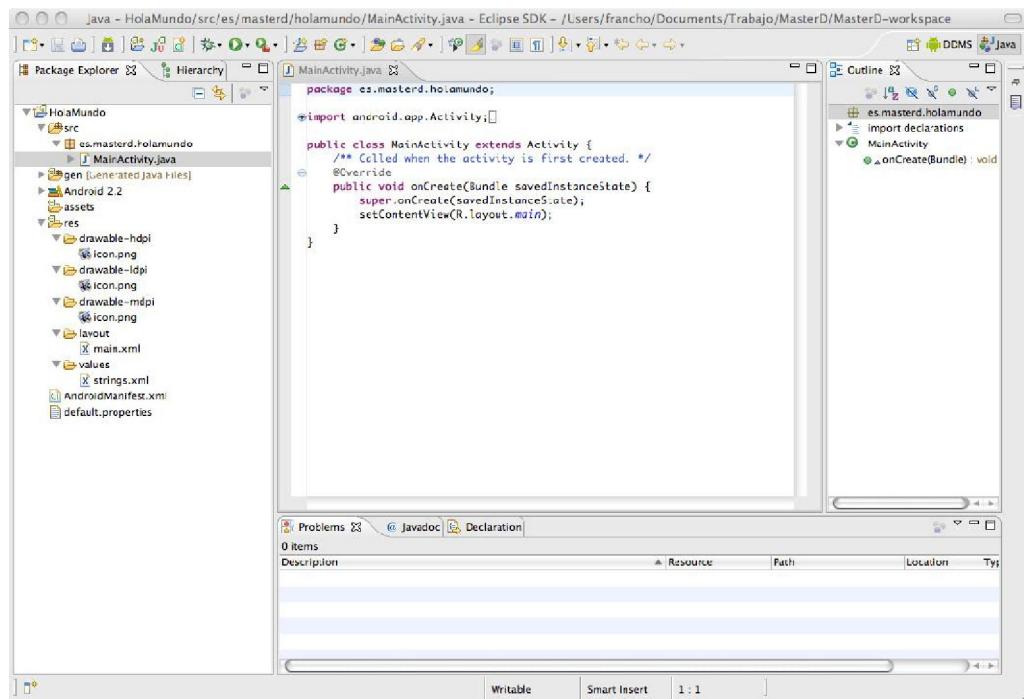


Figura 3. Proyecto “HolaMundo”

La carpeta de fuentes (src)

La primera carpeta que nos encontramos es “src”. Contiene el código fuente organizado en paquetes.

En esta carpeta será donde colocaremos las clases java que programemos.

Como hemos marcado la casilla de crear Activity, Eclipse ya nos ha creado el paquete principal y dentro nos ha colocado la clase MainActivity.java

La carpeta de archivos generados (gen)

En ella se colocaran los archivos que el compilador genera en sus repasos. Como el archivo de recursos R (del que hablaremos más adelante).

Es una carpeta que normalmente no tendremos que tocar.

La carpeta de recursos varios (assets)

Sirve para almacenar recursos que pueda necesitar nuestra aplicación como por ejemplo ficheros de música, zip, etc.

Se podrá acceder a ellos como si se tratara de un sistema de ficheros gracias a la clase del sistema AssetManager.

La clase de recursos (res)

Esta carpeta, junto con la carpeta de fuentes es la que más vamos a usar.

Contiene todos los recursos que necesita la aplicación como definición de pantallas, iconos, gráficos, cadenas de texto localizadas...

Todos los archivos que se coloquen aquí serán indexados por el compilador y se genera el fichero de recursos “R” que nos permitirá acceder a ellos de una forma rápida.

Debido a la gran variedad de archivos que puede almacenar, está dividida en subcarpetas:

- anim: ficheros xml de definición de animaciones.
- color: ficheros xml de definición de colores.
- drawable: ficheros bitmap (.png, .9.png, jpg, .gif) o xml con contenidos que se “dibujarán”, es decir, iconos, imágenes, fondos, definición de botones...
- layout: ficheros xml que definen la capa de interfaz de usuario.



- menu: ficheros xml con la definición de los menús de la aplicación.
- raw: ficheros binarios que no tienen cabida en el resto de carpetas².
- values: ficheros xml de definición de valores simples como estilos, cadenas de texto para localización.
- xml: archivos xml varios que pueden ser accedidos en tiempo de ejecución con el método Resources.getXml().

Como veremos más adelante, ciertas carpetas de *resources* pueden tener varias versiones para adaptarse mejor a los diferentes tamaños y formatos de pantalla, idiomas, etc.

El manifiesto Android (AndroidManifest.xml)

Todos los proyectos deben tener un archivo en la carpeta principal, de nombre *AndroidManifest.xml* en el que se detallan las características principales de la aplicación (módulos principales, permisos necesarios, nombre, icono...).

²Si los archivos binarios no van a ser accedidos con la clase R, es mejor colocarlos en la carpeta “asserts”.

3. “HOLA MUNDO” AL DETALLE

Como hemos visto, al crear el proyecto Eclipse nos ha generado automáticamente la estructura de carpetas de nuestra aplicación y ha creado los ficheros básicos para que esta funcione:

Lo primero que ha creado es la clase principal `MainActivity.java` dentro de la carpeta “src” y en el paquete “`es.masterd.holamundo`”.

```
package es.masterd.holamundo;

import android.app.Activity;
import android.os.Bundle;

public class MainActivity extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }
}
```

Código 1: ./src/es/masterd/holamundo/MainActivity.class

Las actividades son las encargadas de mostrar las interfaces gráficas de usuario. Deben extender la clase `Activity`.

Android al crear una `Activity` llama a su método `onCreate()` que es el encargado de hacer todo lo necesario para que la pantalla pueda ser mostrada al usuario.

En nuestro caso, hace una llamada a `setContentView()` para cargar una vista ya definida. Como parámetro le pasa su identificador a través de la clase de recursos generada.

`R.layout.main` hace referencia a un fichero xml de definición de pantalla (es decir situado en la carpeta `./res/layout`) llamado `main.xml`



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

Código 2: ./res/layout/main.xml

En él se define una pantalla en la que los componentes se van a agrupar de forma lineal uno detrás de otro (*LinearLayout*) y que solo tiene un componente de texto (*TextView*).

El componente de texto mostrará la cadena de texto llamada referenciada por @string/hello, es decir, dentro del fichero res/values/strings.xml el valor del ítem llamado “hello”.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="hello">Hello World, MainActivity!
    </string>
    <string name="app_name">HolaMundo</string>
</resources>
```

Código 3: ./res/values/strings.xml

Como hemos visto antes, para que la aplicación funcione es necesario definir el AndroidManifest.xml.

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

Código 4: ./AndroidManifest.xml

Como se puede apreciar, aparte de definir el paquete por defecto y los datos de versión, también se define el ícono, de nuevo con una referencia (se buscará en res/drawable*/icon.* y se cogerá el que más se adapte a la pantalla que estemos usando). Al crear el proyecto se copian los iconos por defecto para que esta referencia funcione.

El nombre de la aplicación es otra referencia al fichero de strings antes citado.

Seguidamente, se definen los componentes de la aplicación (en este caso una sola Activity). Además se le añaden dos filtros para que dicha actividad sea la usada como principal (android.intent.action.MAIN) y también sea incluida en el menú de aplicaciones (android.intent.category.LAUNCHER).



4. PROBANDO NUESTRA APLICACIÓN

Bien, pues ya está todo. Solo falta probar cómo funciona nuestra aplicación. Para ello, basta con pulsar la opción correspondiente del menú "run". Se lanzará el emulador, se instalará y se ejecutará.

En el menú de aplicaciones aparecerá también el ícono que hemos definido.



Figura 4. "Hola Mundo" corriendo en el emulador



CONCLUSIONES



Destrezas, conocimiento y actitud son los elementos que te ayudarán a conseguir el **triunfo profesional**.

Como se ha podido comprobar en este capítulo, crear una proyecto Android es muy sencillo: con unos pocos clics de ratón y rellenando un simple formulario hemos conseguido una aplicación totalmente funcional.

RECAPITULACIÓN

- Los proyectos Android están estructurados en carpetas que debemos respetar.
- Todos los proyectos deben tener un AndroidManifest.xml que define los aspectos básicos de la aplicación.
- Las fuentes deben ir dentro de la carpeta “src”.
- Los recursos que vayan a ser referenciados mediante la clase de recursos R irán en la carpeta “res”.
- En “assets” van recursos binarios que se accederán con la clase AssetManager.



AUTOCOMPROBACIÓN

- 1. ¿Cómo se llama el fichero de configuración de las aplicaciones Android?**
 - a) Android.conf.
 - b) AndroidConf.xml.
 - c) AndroidManifest.xml.
 - d) AndroidManifest.conf.
- 2. Si en el código java se encuentra una referencia a R.strings.app_name ¿a qué hace referencia?**
 - a) A la cadena llamada app_name dentro de ./res/values/strings.xml.
 - b) Al fichero ./res/values/app_name.
 - c) Al fichero ./res/values/app_name.
 - d) Al fichero ./assets/app_name.
- 3. ¿En qué carpeta colocarías la definición de pantalla de su aplicación?**
 - a) ./res/.
 - b) ./res/screen/.
 - c) ./res/layout/.
 - d) ./assets/.

4. ¿En qué carpeta se coloca el código fuente de nuestras clases?

- a) En cualquiera que no sea la raíz.
- b) ./res/.
- c) ./source/.
- d) ./src/.

5. ¿Dónde se colocan las animaciones?

- a) ./res/xml/.
- b) ./res/anim/.
- c) ./res/layout/.
- d) ./res/values/.

6. ¿Dónde se colocan las imágenes que se usan en el diseño de las aplicaciones?

- a) ./res/values/.
- b) ./res/raw/.
- c) ./res/drawable/.
- d) ./res/images/.

7. ¿Qué extensión tienen que tener los ficheros de código fuente Android?

- a) .and.
- b) .xml.
- c) .java.
- d) .src.

8. ¿Qué es el fichero de recursos R?

- a) Una clase que facilita el acceso a todos los ficheros colocados dentro de la carpeta ./res/.
- b) Un fichero con los nombres de ficheros usados en todo el proyecto.
- c) Un fichero que genera el compilador con el listado de las fuentes.
- d) En Android no existe ningún fichero de recursos R.



- 9. Uno de los siguientes ficheros no se puede meter en el directorio ./res/drawable, ¿cuál?**
 - a) photo.png.
 - b) photo.gif.
 - c) photo.xml.
 - d) photo.bmp.

- 10. ¿De qué clase tienen que heredar las clases encargadas de mostrar las pantallas que interactúan con el usuario?**
 - a) MainActivity.
 - b) Activity.
 - c) Screen.
 - d) Gui.

- 11. ¿En qué fichero se define el icono de la aplicación?**
 - a) MainActivity.java.
 - b) AndroidManifest.xml.
 - c) strings.xml.
 - d) res/drawable/.

- 12. ¿Qué filtro hay que poner a una Activity para que aparezca en el menú de aplicaciones?**
 - a) android.intent.action.MAIN.
 - b) android.intent.category.LAUNCHER.
 - c) android.intent.category.MAINMENU.
 - d) android.intent.category.MAINLAUNCHER.

- 13. ¿Qué filtro hay que poner a una Activity para indicar que es la actividad de inicio?**
 - a) android.intent.action.MAIN.
 - b) android.intent.category.LAUNCHER.
 - c) android.intent.category.MAINMENU.
 - d) android.intent.category.MAINLAUNCHER.

14. ¿Qué método de Activity hay que llamar para establecer una vista?

- a) setMainView().
- b) setContentView().
- c) setMainLayout().
- d) setContentLayout().

15. Desde un fichero xml de *layout*, ¿cómo harías referencia a una cadena guardada en ./res/values/strings.xml llamada app_name?

- a) ref=app_name.
- b) R.string.app_name.
- c) @string/app_name.
- d) \$app_name.

16. ¿Cuál de las siguientes no es una subcarpeta válida del directorio res?

- a) color.
- b) menu.
- c) music.
- d) anim.

17. ¿Cuál de estas carpetas no deberíamos tocar normalmente?

- a) src.
- b) gen.
- c) res.
- d) assets.

18. ¿Con qué clase podremos acceder en tiempo de ejecución al contenido de la carpeta assets?

- a) Activity.
- b) Asset.
- c) AssetDirectory.
- d) AssetManager.



19. ¿Qué componente usamos para mostrar un texto en un *layout*?

- a) TextView.
- b) ViewText.
- c) TextLabel.
- d) LabelText.

20. ¿Qué *layout* usaremos si queremos que las vistas que contiene se apilen una detrás de otra?

- a) RelativeLayout.
- b) AbsoluteLayout.
- c) LinearLayout.
- d) PushLayout.



"HOLA MUNDO"

SOLUCIONARIO

1.	c	2.	a	3.	c	4.	d	5.	b
6.	c	7.	c	8.	a	9.	d	10.	b
11.	b	12.	b	13.	a	14.	b	15.	c
16.	c	17.	b	18.	d	19.	a	20.	c



Dirígete hacia el triunfo profesional. Un P8.10 siempre consigue lo que se propone.

PROPUESTAS DE AMPLIACIÓN

Partiendo del código fuente de este capítulo:

- Cambia el texto que aparece por pantalla para que muestre otra frase.
- Cambia el nombre de la aplicación.
- Cambia el icono de la aplicación.
- Intenta añadir una segunda línea de texto.



BIBLIOGRAFÍA

- Guía Android: Descubriendo con el HolaMundo
 - [http://www.maestrosdelweb.com/editorial/descubriendo-android-con-el-hello-world/.](http://www.maestrosdelweb.com/editorial/descubriendo-android-con-el-hello-world/)

PROGRAMACIÓN PARA ANDROID

4

Fundamentos de las
aplicaciones Android



ÍNDICE

MOTIVACIÓN.....	3
PROPÓSITOS	4
PREPARACIÓN PARA LA UNIDAD	5
1. CONCEPTOS BÁSICOS	7
2. COMPONENTES DE LAS APLICACIONES.....	8
3. <i>INTENTS</i>	11
4. ANDROIDMANIFEST	12
5. ACTIVIDADES Y TAREAS.....	13
6. PROCESOS E HILOS	15
7. CICLO DE VIDA DE LOS COMPONENTES.....	16
8. LIMPIEZA DE PROCESOS.....	20
9. NUESTRO SEGUNDO PROGRAMA.....	21
CONCLUSIONES.....	27
RECAPITULACIÓN	28
AUTOCOMPROBACIÓN	29
SOLUCIONARIO.....	35
PROPUESTAS DE AMPLIACIÓN	36
BIBLIOGRAFÍA.....	37



MOTIVACIÓN



Dirígete hacia tu meta sin vacilar; nosotros te ayudamos a llegar hasta allí.

En el capítulo anterior empezaron a aparecer conceptos propios de la programación Android como las Actividades (Activity). En este capítulo ampliaremos este y otros conceptos que son la base de las aplicaciones Android.

PROPOSITOS

En este capítulo los propósitos son los siguientes:

- Estudiar los conceptos básicos de un proyecto Android.
- Definir los componentes de las aplicaciones Android y su ciclo de vida.
- Estudiar el concepto de Actividades y tareas.
- Definir la limpieza de procesos por parte del sistema.
- Realizar una aplicación que abrirá varias pantallas que pasarán datos entre ellas.



PREPARACIÓN PARA LA UNIDAD

Es necesario tener terminado y haber comprendido el ejercicio “hola mundo” del capítulo anterior, ya que en este capítulo ampliaremos su funcionalidad.



1. CONCEPTOS BÁSICOS

Ya hemos visto que un proyecto Android se compone de varias carpetas claramente estructuradas, pero en los terminales lo que se instala son ficheros con extensión .apk.

Los ficheros de paquetes de aplicaciones o apk, son generados por la herramienta apk¹ una vez terminada la compilación (Eclipse se encarga de llamarlo) y contienen todo lo necesario para que la aplicación funcione.

A nivel de ejecución nuestras aplicaciones tendrán su propio entorno seguro de ejecución:

- Por defecto, cada aplicación se ejecutará en su propio proceso Linux. El sistema lo creará cuando llamemos a la aplicación y lo destruirá cuando la aplicación no haya sido usada durante un rato y se necesiten recursos para otra aplicación.
- Cada proceso correrá su propia máquina virtual. Así pues cada aplicación correrá aislada del resto. Esto permite que, ante un hipotético fallo de la aplicación solo se vea afectada su máquina virtual, no el resto, con lo que el sistema seguiría corriendo sin problema.
- Por defecto², a cada aplicación se le asigna un identificador de usuario (*uid*) distinto. Los permisos de los archivos referentes a la aplicación (datos, caché, etc.) son puestos de tal forma que solo este usuario pueda acceder a ellos quedando vetados al resto de aplicaciones.

¹ Suele encontrarse en el directorio “tools” de la carpeta de instalación de Android.

² Es posible asignar el mismo “uid” a dos aplicaciones distintas para que compartan la máquina virtual y los recursos.

2. COMPONENTES DE LAS APLICACIONES

Como ya hemos mencionado antes, la característica principal de Android es la reutilización de componentes de una aplicación por otra.

Veamos un ejemplo: imaginemos que estamos programando una aplicación para gestionar nuestra biblioteca de libros y queremos incluir la foto en la ficha. Pues bien, en cuanto al código en vez de tener que programar todo el sistema de captura o selección de imagen podemos llamar a los que ya traen otras aplicaciones como Gallery o la cámara fotográfica. Les pasaremos el control y una vez el usuario haya seleccionado la foto correspondiente, el programa nos devolverá el control junto con la foto para que podamos usarla.

Esta filosofía obliga a dividir las aplicaciones en módulos independientes que solo hagan una cosa, es decir, son como “miniaPLICACIONES que solo realizan una función”.

Veamos el caso contrario: imagina que estás programando una aplicación para la red social Twitter. Un módulo claro sería el de enviar un mensaje o “Twitt”. Siguiendo esta filosofía esto sería una Activity independiente que recibiría como parámetro el mensaje a enviar (si no recibe ninguno mostraría el formulario para capturarlos), los enviaría a Twitter usando su API y finalmente se cerraría la Activity devolviendo el control a la que le había llamado. Si se programa de esta manera (y con los filtros adecuados en el `AndroidManifest.xml`) cada vez que otra aplicación quiera compartir algo por Twitter podrá llamar a nuestra actividad pasándole los parámetros y esperar a que ella sea la que haga el trabajo.

Es decir, las aplicaciones Android no tienen por qué tener un único punto de entrada, podemos definir todos los que necesitemos y, además, pueden comportarse de formas distintas.

Para conseguir esto, Android nos ofrece cuatro tipos de componentes básicos: actividades, servicios, receptores de mensajes de difusión y proveedores de contenidos.



Actividades (Activity)

Las actividades son las encargadas de mostrar las interfaces de usuario e interactuar con él. Suelen tener asociada una definición de pantalla y son las encargadas de responder a los eventos de usuario (pulsaciones de botones, selección de listas, etc.).

Deben heredar de la clase *Activity*.

El aspecto de la vista se aplica pasando un objeto tipo *View*³ al método *Activity.setContentView()*. Este método es el encargado de dibujar la pantalla. Normalmente la vista que pasemos ocupará toda la pantalla (aunque hay formas de hacer que sea mostrada como ventana flotante). Las *Activity* también pueden llamar a componentes modales que se mostrarán sobre su *View* como por ejemplo ventanas de dialogo o menús.

Por cada pantalla distinta habrá una *Activity* distinta. Pasar de una pantalla a otra se consigue lanzando nuevas *Activity* y volver a la anterior se consigue cerrando la *Activity* actual.

Normalmente las aplicaciones tienen una *Activity* marcada como punto de entrada.

Veamos un ejemplo: imagine que programa un lector de correo.

Esta aplicación tendrá las siguientes actividades:

- *ListadoActivity*: mostrará el listado de mensajes del *inbox*.
- *VerMensaje*: mostrará el contenido de un mensaje.
- *NuevoMensaje*: recibirá como parámetros los datos necesarios, si no recibe nada mostrará el formulario y una vez se pulse enviar mandará el mensaje.

En el *AndroidManifest.xml* definiremos como puntos de entrada “*ListadoActivity*” y “*NuevoMensaje*”, de esta forma el resto de aplicaciones podrán reutilizar nuestro código (por ejemplo, cuando quieran mandar un e-mail).

Servicios

Los servicios no tienen en interfaz visual, corren en segundo plano y se suelen encargar de realizar tareas que deben continuar ejecutándose cuando nuestra aplicación no está en primer plano.

Los servicios extienden de la clase base *Service*.

³ Los objetos *View* son los encargados de dibujar una parte rectangular de la pantalla. Los objetos *View* a su vez pueden contener más objetos *View*. Todos los componentes de la interfaz de usuario (botones, campos de texto, imágenes, etc.) son subclases de *View*.

Siguiendo nuestro ejemplo anterior, la aplicación de correo tendrá un servicio que se encargará de chequear y descargar los correos cada cierto tiempo.

Es posible conectar con un servicio en ejecución (o lanzarlo) usando la interfaz que nos aporta la clase *Service*.

En la aplicación de correo podríamos, por ejemplo, forzar la recarga de correo o modificar la frecuencia de comprobación de nuevos mensajes.

Los servicios nos ofrecen un mecanismo para ejecutar tareas pesadas sin bloquear el resto de la aplicación ya que se ejecutan en un hilo distinto.

Receptores de mensajes de distribución (*BroadcastReceiver*)

Los receptores de mensajes no hacen nada excepto recibir y reaccionar ante mensajes de difusión.

Extienden la clase *BroadcastReceiver*.

No tienen interfaz de usuario, pero son capaces de lanzar Activity en respuesta a un evento o usar el “NotificationManager” para alertar al usuario.

El sistema lanza muchas notificaciones de sistema (llamada entrante, llegada de SMS, cambio de posición GPS...) a las que podemos añadir las nuestras.

Siguiendo con la aplicación de correo, tendría un “BroadcastReceiver” que escucharía a los mensajes de tipo NUEVO_CORREO que lanzaría nuestro servicio cada vez que detectara uno. El “BroadcastReceiver”, mandaría un aviso a la barra de sistema para notificar al usuario.

Proveedores de contenido (*ContentProvider*)

Los proveedores de contenido hacen que un determinado grupo de datos estén disponibles para distintas aplicaciones.

Extienden la clase *ContentProvider* para implementar los métodos de la interfaz, las aplicaciones no acceden a este interfaz directamente, sino que lo hacen a través de una clase *ContentResolver*.

Este sistema permite acceder a datos almacenados en el sistema de ficheros, bases de datos SQLite o cualquier otra fuente de datos a través de un sistema unificado.

El lector de correo podría disponer de un “ContentProvider” que permitiera acceder al *inbox* y los datos de mensajes de una forma sencilla.



3. INTENTS

Ya hemos visto que los “ContentProviders” se activan cada vez que una aplicación hace uso de un “ContentResolver”.

El resto de componentes (*Activity*, *Service* y *BroadcastReceiver*) se activan a través de unos mensajes asíncronos llamados “Intent”.



Un *Intent* es un objeto que contiene todos los datos del mensaje.

Hay tres métodos distintos para activar cada uno de los componentes:

- Las Activity se lanzan (o se traen a primer plano) pasando un Intent al método `Context.startActivity()` o `Activity.startActivityForResult()`. Dentro de la Activity llamada podemos abrir el objeto Intent recibido para obtener los parámetros usando el método `getIntent()`.
- Para interactuar o lanzar servicios pasaremos el Intent al método `Context.startService()`. El Intent podrá ser analizado en su método `onBind()`.
- Para producir un mensaje de difusión deberemos pasar el Intent al método de difusión como `Context.sendBroadcast()`, `Context.sendOrderedBroadcast()` o `Context.sendStickyBroadcast()` de esta forma el Intent será entregado a todas las clases “BroadcastReceiver” que estén escuchando que podrán analizarlo en su método `onReceive()`.

4. ANDROIDMANIFEST

Para poder usar un componente, no solo es necesario crearlo extendiendo sus clases correspondientes, sino que también tenemos que definirlo en el `AndroidManifest.xml`

Este fichero de configuración está basado en XML que podemos editar directamente. El *plugin* de Android para Eclipse también nos permite editarlo a través de un formulario.

Cada uno de los componentes tiene su propio *tag* xml asignado:

- `<activity>` para actividades.
- `<service>` para servicios.
- `<receiver>` para receptores de mensajes de difusión.
- `<provider>` para proveedores de contenidos.

Otro *tag* que aparece en `AndroidManifest.xml` es `<intent-filter>`, este *tag* permite categorizar los componentes de forma que cuando tengamos que llamarlos (nosotros u otra aplicación) no tengamos que conocer el nombre concreto del Intent, será Android quién basándose en su categoría y parámetros elegirá el componente que mejor se adapte.



5. ACTIVIDADES Y TAREAS

Así pues, conforme abrimos actividades (Activity) se van apilando en una pila de tareas. Cada vez que una actividad termina (por ejemplo llamando al método `finish()`, o pulsando el botón “back”) se saca de la pila, pasando el control a la que se había abierto anteriormente (dejándola en la cima de la cola).

Una misma Activity puede estar varias veces en la pila si ha sido llamada en diferentes ocasiones (por ejemplo, en la aplicación de correo, la Activity de “VerCorreo” podría estar duplicada si hubiéramos abierto diferentes e-mails y no los hubiéramos cerrado).

Esta pila de actividades es enviada a un segundo plano cuando la aplicación pierde el foco (por ejemplo, cuando se lanza otra o se pulsa el botón “home”).

Cuando la aplicación vuelve a tomar el control, la pila es traída de nuevo al primer plano.

Este comportamiento por defecto puede ser modificado mediante *flags* que se pasan al objeto Intent a partir de las propiedades de la activity descritas en el AndroidManifest⁴.

⁴ Se recomienda leer la documentación oficial de Android para ver todas las opciones disponibles en el AndroidManifest. <http://developer.android.com/guide/topics/manifest/manifest-intro.html>.

Casos de uso	Modo de lanzamiento	Instancias múltiples	Comentarios
Normal (para casi todos los casos)	"standard"	Sí	Comportamiento por defecto. Se crea una nueva actividad en la pila de tareas y se apunta el "Intent" a él.
	"singleTop"	Condicional	Si hay una instancia de la actividad en la cima de la pila de tareas, se apunta el "Intent" a ella llamando a su método <code>onNewIntent()</code> en vez de crear una nueva.
Especializado (para casos especiales)	"singleTask"	No	El sistema crea la activity y la coloca al principio de una pila de tareas. Si la activity ya estaba creada llama a su método <code>onNewIntent()</code> .
	"singleInstance"	No	Igual que "singleTask" con la diferencia que en su pila dedicada no se meten más actividades.

Si una pila de tareas es abandonada por el usuario durante un período de tiempo (por ejemplo, al lanzar otra aplicación) y el sistema necesita más recursos, la pila de actividades se limpia (a excepción de la actividad inicial). Como siempre este comportamiento por defecto puede ser modificado mediante atributos del Manifiesto:

Atributo	Comportamiento
<code>alwaysRetainTaskState</code>	Si es <i>true</i> en la actividad de inicio, la tarea mantendrá todas las actividades en la pila, incluso aunque se abandone durante largos períodos de tiempo,
<code>clearTaskOnLaunch</code>	Si este atributo es <i>true</i> en la actividad de inicio de la tarea, la pila será limpia (a excepción de la tarea inicial) cada vez que la pila sea llevada a segundo plano.
<code>finishOnTaskLaunch</code>	Similar a <code>clearTaskOnLaunch</code> pero en vez de afectar a toda la pila, solo afecta a la actividad que lo tiene a <i>true</i> , es decir, la actividad se lanzará y desaparecerá de la pila dejando el resto como estaba.



6. PROCESOS E HILOS

Como ya hemos mencionado, cada aplicación se lanza en un proceso Linux distinto. Por defecto, todos los componentes y procesos de la aplicación correrán en este hilo.

Este comportamiento se puede modificar con el atributo *process* de cada uno de los componentes (*activity*, *service*, *provider* y *receiver*). El tag *application* también puede llevar este atributo para que sea aplicado a todos sus elementos.

Dentro de nuestros programas para gestionar las tareas pesadas podemos utilizar hilos para ejecutarlas en segundo plano. Para ello se usa el objeto Thread de Java, aunque Android nos provee de otros objetos para facilitarnos el trabajo, como Handler, AsyncTask o Looper (por citar algunos).

7. CICLO DE VIDA DE LOS COMPONENTES

Cada tipo de componente tiene un ciclo de vida distinto. Las clases que extendemos para crearlos contienen métodos de “callback” que permiten responder ante cada cambio de estado.

Una cosa importante a recordar es que estos métodos deben llamar obligatoriamente al mismo método de su superclase.

Ciclo de vida de los ContentProvider

Los ContentProvider solo están levantados mientras son referenciados mediante un ContentResolver.

Ciclo de vida de los Receiver

Los objetos de tipo Receiver también tienen un ciclo de vida muy corto ya que solo se activan cuando se produce un mensaje de difusión que capturan mediante su método “callback2”:

```
void onReceive(Context curContext, Intent broadcastMsg)
```

Mientras se esté ejecutando este método se considera que el Receiver está activo.

Esto supone un problema cuando la tarea a ejecutar es pesada y debemos lanzarla en un hilo aparte. Al terminar el método el sistema considerará que el Receiver queda inactivo (aunque el hilo que hemos lanzado siga ejecutándose) y puede ser que sea eliminado de la pila interrumpiendo dicha tarea.

La solución es sencilla: lanzar un servicio desde este método que se encargue de hacer las tareas pesadas, así aunque el receiver sea considerado inactivo, el servicio, que tiene su propio ciclo de vida seguirá ejecutándose.

Ciclo de vida de las Activity

Las actividades tienen básicamente tres estados:

- **Activo o en ejecución:** la actividad está en primer plano, es decir, arriba de la pila de la tarea y está respondiendo a las acciones del usuario.
- **Pausado:** sigue siendo visible para el usuario, pero ha perdido el foco. Por ejemplo porque se ha mostrado una nueva actividad en modo diálogo flotando sobre ella. Antes de entrar en este estado es recomendable salvar los datos y estado de la interfaz ya que nadie nos asegura que no sea destruida porque el sistema necesite memoria.
- **Parado:** la aplicación ya no es visible para el usuario. Queda disponible para que el sistema, si necesita más memoria la quite de la pila sin ningún miramiento. Siendo necesaria crearla desde 0 la próxima vez que se necesite.

La clase Activity tiene una serie de métodos que son llamados cada vez que se cambia de estado y que permiten, por ejemplo, guardar el estado para que la próxima vez que se necesite la actividad se cargue más rápido.

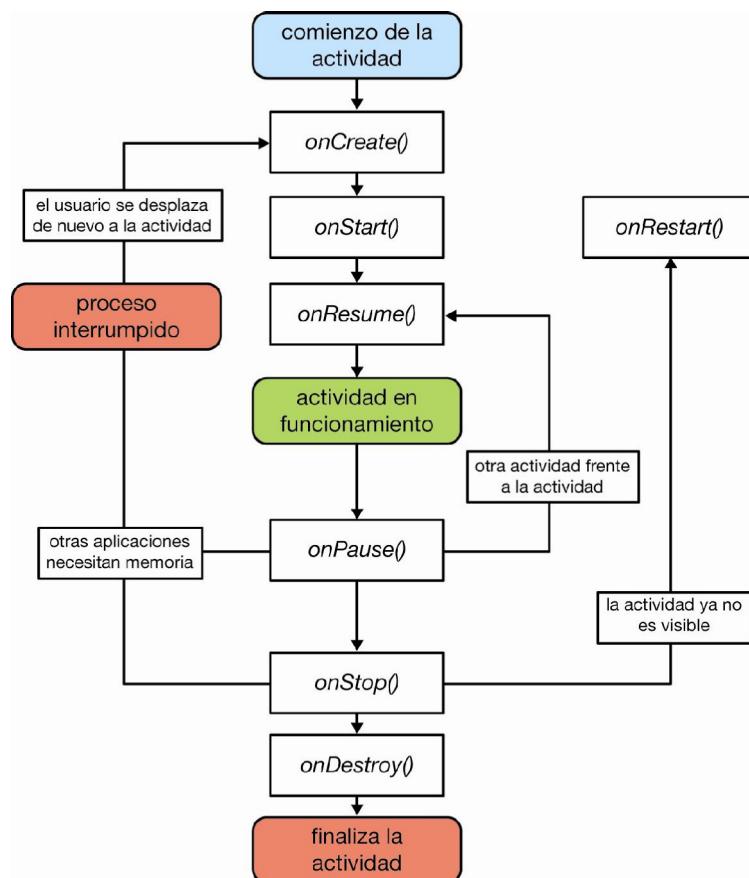


Figura 1. Cambios de estado de una actividad según la documentación de Android

Veamos los principales métodos:

- `onCreate(Bundle savedInstanceState)` se llama al crear la Actividad. Normalmente siempre se sobrescribe usándolo para configurar la vista, enlazar datos con listas, crear adaptadores, etc. Puede recibir como parámetro el estado anterior de la actividad para que pueda ser restaurada completamente si es necesario.
- `onPause()` se llama justo antes de que el sistema traiga a primer plano otra actividad. Es la última oportunidad de guardar datos, ya que nadie nos asegura el tiempo que la actividad va a permanecer en la pila. En este método también se suelen parar las tareas pesadas, animaciones y cualquier otra cosa que consuma CPU.
- `onStop()` se llama cuando la actividad va a ser ocultada y no se va a usar durante un tiempo. Si hay una necesidad de recursos, este método puede no ser llamado, por eso es mejor guardar los datos en el estado anterior (`onPause`).
- `onDestroy()` es el último método llamado antes de destruir la actividad bien porque se ha llamado al método `finish()` o bien porque el sistema está destruyendo actividades para conseguir más memoria. Se puede usar el método `isFinishing()` para comprobar si es un caso u otro.

Imaginemos una actividad que muestra el tiempo de una determinada ciudad, los datos los obtiene de una conexión a Internet. Las conexiones son un recurso “caro” que puede ralentizar las aplicaciones y consume conexión de datos.

Dicha conexión la haríamos al crear la actividad (posiblemente en un segundo plano), lo que está claro es que, una vez obtenidos los datos, sería una buena idea guardar el estado de la actividad para evitar que si es destruida, cuando volvamos a entrar en ella haya que descargar los datos de nuevo.

Para esto implementaríamos el método `onSaveInstanceState()` que guardaría un Boundle que es el mismo que luego será pasado al método `onCreate` si hay que recrear la aplicación desde cero. De esta forma y con el código adecuado en `onCreate()`, la segunda vez que se creara la actividad, no sería necesario descargar de nuevo los datos de Internet.

El ciclo de vida de Service

Un servicio puede ser utilizado de dos formas. Dependiendo de como sea lanzado su ciclo de vida será uno u otro.

- Si el servicio se lanza con `startService()` correrá hasta que sea terminado. Igual que una actividad, los servicios se configuran en el método `onCreate()` y se libera en el método `onDestroy()`. Un servicio puede ser terminado externamente con una sola llamada `Context.stopService()` o desde el mismo servicio con el método `Service.stopSelf()` o `Service.stopSelfResult()`.

- Si por el contrario hacemos una llamada a `Context.bindService()` el servicio se levantará (sino está levantado ya) y podremos interactuar con el mediante una interfaz que dicho servicio deberá exportar. Una vez terminemos cortaremos la conexión con `Context.unbindService()`.

Estos métodos pueden convivir, por ejemplo podemos lanzar un servicio con `startService()` y luego conectar a él con una llamada a `bindService()`. Para que esto pueda ser posible es necesario implementar los siguientes métodos:

- `IBinder onBind(Intent intent)`.
- `boolean onUnbind(Intent intent)`.
- `void onRebind(Intent intent)`.

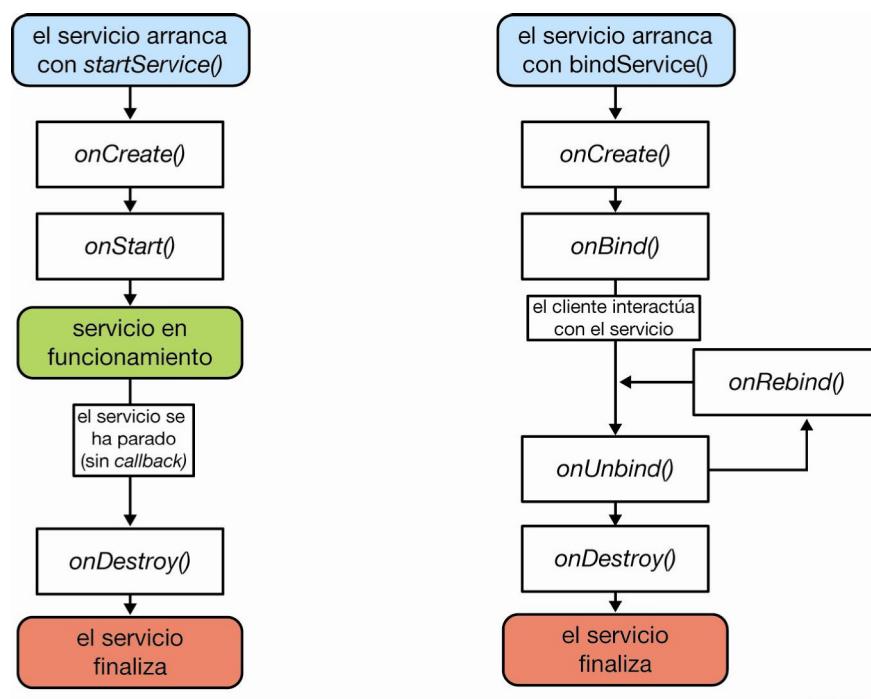


Figura 2. Ciclo de vida de un servicio según la documentación de Android



8. LIMPIEZA DE PROCESOS

Como ya hemos mencionado antes, el sistema libera recursos destruyendo componentes inactivos cuando necesita más memoria. Para ello se eliminará el de menos importancia:

1. Primero se eliminarán los procesos vacíos. Estos procesos están relacionados con aplicaciones ya cerradas que se mantienen en memoria para agilizar la siguiente vez que se cargue dicha aplicación.
2. Procesos en segundo plano. El método `onStop()` ha sido llamado y ya no son visibles. El sistema mantiene una lista de procesos en `background` y elimina primero el más antiguo
3. Si sigue necesitando más recursos, elimina los procesos de servicio.
4. Si todavía necesita más, elimina los procesos pausados, a los que ha invocado el método `onPause()`.
5. Finalmente, si la situación es crítica y sigue necesitando más memoria elimina el proceso en primer plano.

Como se puede ver es importante implementar correctamente los métodos de estado de todos nuestros componentes para evitar perder información en los diferentes cambios de estado.



9. NUESTRO SEGUNDO PROGRAMA

Partiendo del código fuente de nuestra primera aplicación “hola mundo” vamos a ampliar su funcionalidad para comprobar cómo llamar a una segunda actividad y cómo pasar parámetros entre ellas.

Lo primero es modificar el *layout* main.xml para añadir dos botones que lanzarán las nuevas actividades.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="vertical"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
    >
    <TextView
        android:id="@+id/texto"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/hello"
    />
    <Button
        android:id="@+id/boton1"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/otra"
    />
    <Button
        android:id="@+id/boton2"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/otraMas"
    />
</LinearLayout>
```

También deberemos crear un nuevo layout que usaremos para mostrar los parámetros pasados. Lo llamaremos ./res/layout/otra.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
    <TextView
        android:id="@+id/texto"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/otra"
        />
    <TextView
        android:id="@+id/parametros"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/hello"
        />
    <Button
        android:id="@+id/boton"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/otra"
        />
</LinearLayout>
```

En strings.xml meteremos las nuevas cadenas que hemos usado en los layouts:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="hello">Hello World, MainActivity!
    </string>
    <string name="app_name">HolaMundo</string>
    <string name="salir">Salir</string>
    <string name="otra">Otra Actividad</string>
    <string name="otraMas">Otra Actividad (devuelve
    parámetros)</string>
</resources>
```

Luego modificaremos la actividad principal (MainActivity.java) añadiendo el comportamiento a los botones.

```
package es.masterd.holamundo;
import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.Toast;
public class MainActivity extends Activity {
protected static final int CODIGO_PETICION = 100;
/** Called when the activity is first created. */
@Override
```



```
public void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.main);  
    // Buscamos los componentes de botón  
    Button boton1 = (Button)  
        findViewById(R.id.boton1);  
    Button boton2 = (Button)  
        findViewById(R.id.boton2);  
    // El botón1 al pulsarse abrirá una actividad  
    boton1.setOnClickListener(new OnClickListener()  
    {  
        public void onClick(View v) {  
            Intent intent = new Intent();  
            intent.setClass(getApplicationContext(),  
                OtraActivity.class);  
            startActivity(intent);  
        }  
    });  
    // El botón2 al pulsarse pasará parámetros a otra actividad  
    // para que nos los vuelva modificados  
    boton2.setOnClickListener(new OnClickListener()  
    {  
        public void onClick(View v) {  
            Intent intent = new Intent();  
            intent.setClass(getApplicationContext(),  
                OtraParametrosActivity.class);  
            // Pasamos un parámetro  
            intent.putExtra("parametro1", "esto es un  
                parámetro");  
            // Llamamos a la nueva actividad indicando que esperamos  
            // que nos devuelva un valor  
            // asociado a la constante CODIGO_RETORNO  
            startActivityForResult(intent, CODIGO_PETICION);  
        }  
    }  
    /**  
     * Este método será llamado cuando recibamos la respuesta de la actividad llamada  
     */  
    @Override  
    protected void onActivityResult(int requestCode,  
        int resultCode, Intent data) {  
        super.onActivityResult(requestCode, resultCode,  
            data);  
        if(requestCode == CODIGO_PETICION) {  
            // Extraemos el resultado  
            String resultado = data.getStringExtra  
                ("resultado");  
            // Mostramos un mensaje de usuario  
            Toast toast =  
                Toast.makeText(getApplicationContext(),"Valor devuelto: "+resultado,Toast.LENGTH_LONG);  
            toast.show();  
        }  
    }  
}
```

Ahora crearemos dos nuevas actividades.

Una que no haga nada a excepción de mostrar su vista y se lanzará cuando se pulse el botón 1 llamada OtraActivity.java:

```
package es.masterd.holamundo;
import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.TextView;
public class OtraActivity extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        // cargamos la vista
        setContentView(R.layout.otra);
        // Localizamos los componentes de la vista
        TextView texto = (TextView) findViewById(R.id.texto);
        Button boton = (Button) findViewById(R.id.boton);
        // Cambiamos el texto del campo de texto
        texto.setText(R.string.otra);
        // Cambiamos el texto del botón
        boton.setText(R.string.salir);
        // Asignamos el comportamiento al botón:
        // cuando se pulse se cerrará para volver a la anterior
        boton.setOnClickListener(new OnClickListener() {
            public void onClick(View v) {finish();}
        });
    }
}
```

La otra actividad capturará un parámetro y devolverá otro. La llamaremos OtraParametrosActivity.java:

```
package es.masterd.holamundo;
import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.TextView;
public class OtraParametrosActivity extends Activity {
    private static final int CODIGO_RETORNO_OK = 1;
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        // cargamos la vista
        setContentView(R.layout.otra);
        // Localizamos los componentes de la vista
        TextView texto = (TextView) findViewById(R.id.texto);
        Button boton = (Button)
```



```
        findViewById(R.id.boton);
        TextView parametros = (TextView)
            findViewById(R.id.parametros);
        // Cambiamos el texto del campo de texto
        texto.setText(R.string.otra);
        // Cambiamos el texto del botón
        boton.setText(R.string.salir);
        // Asignamos el comportamiento al botón:
        // cuando se pulse se cerrará para volver a la anterior
        // actividad
        boton.setOnClickListener(new OnClickListener() {
            public void onClick(View v) {
                // cerrará la actividad sacándola de la pila y
                // devolviendo el control a la Actividad anterior
                devuelveParametros();
            }
        });
        // Mostramos los parámetros recibidos
        Intent intent = getIntent();
        Bundle todosParametros = intent.getExtras();
        parametros.setText(todosParametros.getString("parametro1"));
    }
    protected void devuelveParametros() {
        Intent data = new Intent();
        data.putExtra("resultado", "texto devuelto");
        setResult(CODIGO_RETORNO_OK, data);
        finish();
    }
}
```

Finalmente tenemos que actualizar el `AndroidManifest.xml` para que la aplicación reconozca las nuevas actividades:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
    xmlns:android="http://schemas.android.com/apk/res/android"
    package="es.masterd.holamundo"
    android:versionCode="1"
    android:versionName="1.0">
    <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
                    :name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity android:name=".OtraActivity" />
        <activity android:name=".OtraParametrosActivity" />
    </application>
</manifest>
```

Como se puede apreciar, hemos introducido varios conceptos nuevos en nuestra aplicación:

1. El uso del método `findViewById()` que nos permite buscar un componente de la vista.
2. La creación de Listeners que responden a eventos de View, en este caso hemos asociado una acción al evento clic de un botón.
3. También hemos visto como crear un nuevo Intent y lanzarlo (con y sin parámetros).

Finalmente hemos visto como capturar los datos devueltos por la actividad llamada gracias al método `onActivityResult`.



CONCLUSIONES



Destrezas, conocimiento y actitud son los elementos que te ayudarán a conseguir el **triunfo profesional**.

Android es un sistema robusto y estructurado. Es necesario conocer su jerarquía de componentes para elegir el más adecuado en cada momento.

Así mismo es necesario conocer los ciclos de vida de cada una de ellas para poder adaptarlas a nuestras necesidades.

Debemos tener en cuenta en todo momento la posibilidad de que nuestros componentes sean destruidos por el sistema, por ello deberemos sobrescribir los métodos adecuados para que esto sea transparente al usuario.

RECAPITULACIÓN

- Las aplicaciones Android se ejecutan cada una en su propio proceso Linux en el que se ejecuta su propia máquina virtual. A cada aplicación se le asigna su propio *uid*.
- Los componentes básicos de una aplicación Android son: actividades, servicios, receptores de mensajes de difusión y los proveedores de contenido
- Si queremos interactuar con el usuario usaremos una Activity.
 - Si tenemos que realizar una tarea en un segundo plano usaremos un *Service*.
 - Para responder a mensajes de difusión programaremos un *BroadcastReceiver*.
 - Para facilitar el acceso a datos programaremos un *ContentProvider*
- Todos los componentes que utilicemos deben estar definidos en el *AndroidManifest.xml*, en el que podemos modificar su comportamiento por defecto.
- Es importante conocer el ciclo de vida de cada componente ya que, si no sobrescribimos correctamente sus métodos para que nuestra aplicación funcione correctamente.
- Android cuando necesita más recursos, libera memoria destruyendo procesos según su política de limpieza. Nuestros programas deben estar preparados para ello.



AUTOCOMPROBACIÓN

1. ¿Qué afirmación es la correcta?

- a) Cada aplicación Android corre en su propio proceso Linux.
- b) Una aplicación Android puede tener varios procesos Linux.
- c) Una aplicación Android tiene tantos procesos Linux como hilos.
- d) Android es un sistema monoproceso.

2. ¿Cuántas máquinas virtuales puede haber corriendo en un dispositivo Android?

- a) Una por aplicación instalada.
- b) Tantas como permita la memoria (independientemente del número de aplicaciones ejecutándose).
- c) Solo una.
- d) Una por aplicación que se está ejecutando

3. ¿Qué herramienta se encarga de empaquetar las aplicaciones en un fichero apk?

- a) Aapt.
- b) android.
- c) compilador.
- d) zip.

4. **¿Cuál no es un componente Android?**
 - a) Activity.
 - b) Service.
 - c) BroadcastReceiver.
 - d) Process.
5. **¿Dónde pondremos la programación que tiene que interactuar con un usuario?**
 - a) Activity.
 - b) Service.
 - c) BroadcastReceiver.
 - d) ContentProvider.
6. **Imagina una aplicación que tiene que realizar una carga inicial muy pesada de datos en una base de datos. La primera vez que se arranque, ¿dónde iría este código?**
 - a) Activity.
 - b) Service.
 - c) BroadcastReceiver.
 - d) ContentProvider.
7. **Queremos hacer una aplicación que responda ante un aviso de nuevo e-mail, ¿qué programaríamos?**
 - a) Activity.
 - b) Service.
 - c) BroadcastReceiver.
 - d) ContentProvider.
8. **Tenemos una base de datos SQLite a la que queremos acceder de forma transparente, ¿qué programaremos?**
 - a) Activity.
 - b) Service.
 - c) BroadcastReceiver.
 - d) ContentProvider.



9. ¿Qué método se llama al crear una Actividad?

- a) `onCreate()`.
- b) `onStart()`.
- c) `onResume()`.
- d) `onPreStart()`.

10. Dentro del ciclo de vida de una actividad, ¿cuál es el último método que se llama?

- a) `onStop()`.
- b) `onPause()`.
- c) `onDestroy()`.
- d) `onFinish()`.

11. ¿Cuál es el mejor método para guardar datos y evitar que se pierdan cuando una aplicación pasa a segundo plano?

- a) `onResume()`.
- b) `onPause()`.
- c) `onStop()`.
- d) `onDestroy()`.

12. ¿Qué método se llama en una Activity después de `onCreate()`?

- a) `onStart()`.
- b) `onResume()`.
- c) `onPause()`.
- d) `onStop()`.

13. ¿Cómo terminarías la Activity actual?

- a) `kill()`.
- b) `stop()`.
- c) `end()`.
- d) `finish()`.

14. ¿Qué tipo de objeto debes instanciar para poder lanzar nuevas Activity?

- a) *Activity()*.
- b) *Process()*.
- c) *Intent()*.
- d) *Service()*.

15. ¿Cuál es el ciclo de vida de un *ContentProvider*?

- a) Solo está activo mientras se usa un *ContentResolver*.
- b) El del Intent que lo llama.
- c) Todo el ciclo de la aplicación.
- d) Debo crearlo/destruirlo específicamente.

16. ¿Cómo puedo iniciar un servicio?

- a) Sólo con *Context.startService()*.
- b) Sólo con *Context.bindService()*.
- c) con *Context.startService()* y/o *Context.bindService()*.
- d) Ninguna de las anteriores es válida

17. ¿Qué método termina un servicio?

- a) *Context.stopService()*.
- b) *Service.stopSelf()*.
- c) *Service.stopSelfResult()*.
- d) Cualquiera de las anteriores.

18. ¿Qué método se usa para buscar un componente de una vista?

- a) *findViewById()*.
- b) *findView()*.
- c) *viewId()*.
- d) *searchView()*.



19. ¿Qué método debemos usar para asignar un comportamiento a un evento de clic?

- a) `setOnClickListener()`.
- b) `setOnClick()`.
- c) `addOnClickListener()`.
- d) `addOnClick()`.

20. Si el sistema necesita más memoria, ¿qué proceso eliminará primero?

- a) Servicio.
- b) Procesos pausados.
- c) Procesos en segundo plano.
- d) Procesos en primer plano.



SOLUCIONARIO

1.	a	2.	d	3.	a	4.	d	5.	a
6.	b	7.	c	8.	d	9.	a	10.	c
11.	b	12.	a	13.	d	14.	c	15.	a
16.	c	17.	d	18.	a	19.	a	20.	c



Diferénciate del resto; ser un P8.10 tiene su recompensa.

PROPUESTAS DE AMPLIACIÓN

Prueba a modificar la versión de nuestro programa “Hola Mundo” añadiendo un tercer botón que lance una nueva actividad que recibirá dos parámetros enteros y devolverá como resultado la suma de los dos.



BIBLIOGRAFÍA

- <http://developer.android.com/guide/topics/manifest/manifest-intro.html>.
- <http://developer.android.com/guide/topics/intents/intents-filters.html>.

PROGRAMACIÓN PARA ANDROID

5

Interfaz de usuario



ÍNDICE

MOTIVACIÓN.....	3
PROPÓSITOS	4
PREPARACIÓN PARA LA UNIDAD	5
1. ENTENDIENDO INTERFAZ DE USUARIO DE ANDROID	7
2. LAYOUTS TÍPICAS.....	10
3. VIEWS Y EVENTOS DE USUARIO	18
4. COLECCIONES DE DATOS Y VIEWS.....	24
5. MENÚS.....	33
6. DIÁLOGOS Y NOTIFICACIONES.....	36
7. ESTILOS Y TEMAS.....	43
CONCLUSIONES	49
RECAPITULACIÓN	50
AUTOCOMPROBACIÓN	51
SOLUCIONARIO	57
PROPUESTAS DE AMPLIACIÓN	58
BIBLIOGRAFÍA.....	59



MOTIVACIÓN



Dirige tu futuro. En Master.D, te acompañamos incondicionalmente.

Por definición¹, la interfaz de usuario es *el medio con que el usuario puede comunicarse con una máquina, un equipo o una computadora, y comprende todos los puntos de contacto entre el usuario y el equipo, normalmente suelen ser fáciles de entender y fáciles de accionar*.

De nada sirve tener un programa muy potente programado a la perfección. Si la experiencia de usuario es mala o es difícil de usar, lo más seguro es que el usuario deje de utilizarlo.

Así pues no debemos descuidar esta faceta de nuestras aplicaciones.

En este capítulo aprenderemos las técnicas básicas que podemos usar en Android para desarrollar interfaces de usuario.

¹ http://es.wikipedia.org/wiki/Interfaz_de_usuario.

PROPOSITOS

El objetivo principal de esta unidad es dar a conocer las nociones básicas de la creación de interfaces de usuario en Android:

- Tener una visión global de las herramientas y clases que el SDK nos ofrece para estos menesteres.
- Aprender a definir desde cero una pantalla.
- Aprender a interactuar con el usuario capturando los eventos que dispara.
- Aprender a aplicar estilos a nuestra aplicación.



PREPARACIÓN PARA LA UNIDAD

Para esta unidad necesitaremos el entorno de desarrollo Eclipse listo para ser usado con Android.

1. ENTENDIENDO INTERFAZ DE USUARIO DE ANDROID

Como ya hemos adelantado en el capítulo anterior, los componentes de la interfaz de usuario de Android son objetos que descienden de la clase View.

Estos objetos View están organizados en forma de árbol y pueden contener a su vez nuevos objetos View permitiéndonos, así, crear interfaces muy completas.

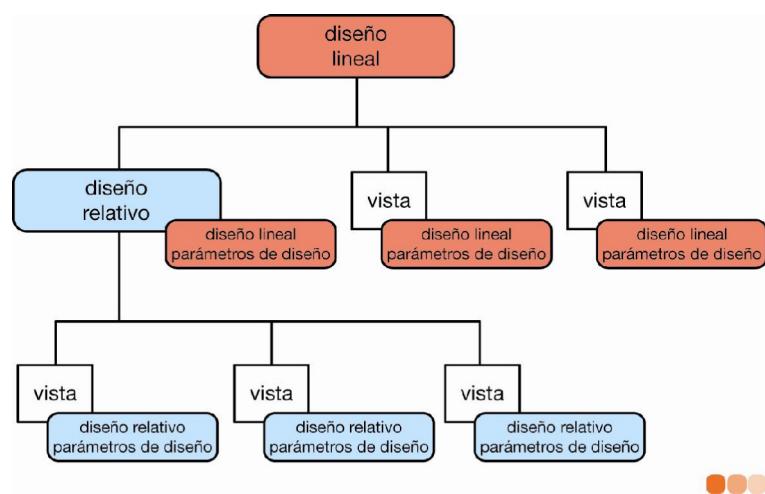


Figura 1. Ejemplo de árbol de objetos View. (Fuente: Android.com)

Este árbol de objetos View se puede definir de dos formas:

- Con un fichero XML que colocamos dentro del directorio res/layout (la mayoría de las veces usaremos este método).
- Mediante código (útil si queremos crear nuestros propios componentes de tipo View).

Para dibujar la interfaz, el sistema necesita que le pasemos el objeto View raíz para ir descendiendo por todos sus nodos y presentar así toda la interfaz. Esto se consigue con el método `Activity.setContentView()` de nuestras actividades.

Internamente Android se encarga de gestionar el dibujo a través de una llamada al método `draw()` de cada vista. Es decir que cada View se encarga de dibujarse a sí misma.

Este proceso de dibujo se realiza en dos pasadas. Primero, se llama al método `measure(int, int)` para que cada objeto View defina el tamaño que necesita y luego se llama al método `layout(int,int,int,int)` para que el objeto sea posicionado dentro de la vista actual.

Así pues, para que Android sepa dibujar un objeto View debemos proveerle de unos datos mínimos que son la anchura y la altura. `LayoutParams` es una clase base que sirve para definir la dimensión y puede tomar los siguientes valores:

- Un número exacto.
- La constante `FILL_PARENT` que indica que la vista intentará ser tan grande como su padre (menos el *padding*).
- La constante `WRAP_CONTENT` que indica que la vista intentará ser lo suficientemente grande para mostrar su contenido (más el *padding*).

La clase `View.MeasureSpec` especifica el tamaño y cómo deben ser posicionadas:

- `UNSPECIFIED`, el padre determina el tamaño deseado del hijo.
- `EXACTLY`, el padre impone un tamaño exacto al hijo.
- `AT_MOST`, el padre impone un tamaño mínimo al hijo. El hijo debe asegurarse de que él (y sus descendientes) ocuparán por lo menos ese tamaño.

Otro atributo básico que solemos usar cuando es el atributo *id*. Se trata de un número entero que sirve para identificar cada objeto view de forma única. Cuando lo declaramos a través de un xml de *resource* podemos hacer referencia a la clase de recursos R usando una @:

- `android:id="@+id/boton"` → hace referencia a un *id* ya existente asociado a la etiqueta “boton”.
- `android:id="@+id/boton2"` → crea una nueva etiqueta en la clase R llamada “boton2”.
- `android:id="@+id/list"` → hace referencia a una etiqueta definida en la clase R del sistema denominada “list”.

Además de estas propiedades básicas, y dependiendo del tipo de objeto, los componentes de tipo View podrán tener otros atributos, como márgenes, colores, fondos, etc. Para saber qué atributos podemos usar en cada caso debaremos consultar su documentación.



Context

Otra cosa a tener en cuenta es que muchos métodos referidos a la vista nos pedirán como parámetro el contexto (*Context*).

El *Context* es un interfaz a la información global de la aplicación. A través de él se puede acceder a los recursos y clases y operaciones, como por ejemplo el lanzamiento de actividades, manejo de Intents, etc.

Dependiendo de dónde estemos, podremos acceder a este contexto de diferentes maneras, como por ejemplo:

- Muchos objetos implementan el método `getContext()`.
- Las actividades implementan este interfaz, así que podemos referenciarlo desde ellas pasándose a sí mismo (`this`) o `NombreActividad.this`.
- También podemos usar `getApplicationContext()` o `getApplication()`.

2. LAYOUTS TÍPICAS

Los *layout* son objetos de tipo ViewGroup (que a su vez desciende de la clase View) cuya finalidad es posicionar otros objetos en pantalla, es decir, son objetos que nos permiten maquetar el contenido.

Vamos a ver ejemplos de las más típicas. Para ello vamos a usar varios objetos TextView (que mostrarán textos) que dispondremos de diferentes maneras usando distintos layouts. La definición de los layouts la vamos a hacer usando ficheros XML que colocaremos en res/layout.

En el objeto de tipo TextView usaremos su propiedad android:text para definir el texto que aparecerá por pantalla.

FrameLayout

Es la más sencilla de todas, no permite ningún tipo de ajuste, y todos sus objetos hijos se empiezan a dibujar en su esquina superior izquierda (si hay varios se apilan uno encima de otro)

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent">
    <TextView android:text="texto 1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
    <TextView android:text="texto 2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
    <TextView android:text="texto 3"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
</FrameLayout>
```



Figura 2. FrameLayout

LinearLayout

Apila a sus hijos de forma consecutiva (uno detrás de otro) empezando a dibujar el primero en su esquina superior izquierda.

Mediante su propiedad “android:orientation” podemos definir la forma en la que se apilan:

- Horizontal: de arriba abajo.
- Vertical: de izquierda a derecha.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="texto 1" />
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="texto 2" />
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="texto 3" />
</LinearLayout>
```



Figura 3. LinearLayout (orientación="horizontal")



Figura 4. LinearLayout (orientación="vertical")



Si queremos que las Views hijas se redimensionen de forma proporcional para ocupar toda la pantalla, deberemos poner el valor FILL_PARENT en su campo “layout_width”, y luego añadir un nuevo campo “width” con valor 0 y otro “weight” con un peso relativo.

De esta forma el *LinearLayout* calcula el espacio que tiene disponible y redimensiona sus hijos para que ocupen todo.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="horizontal"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent">
    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:width="0dip"
        android:layout_weight="1"
        android:gravity="center"
        android:text="texto 1" />
    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:width="0dip"
        android:layout_weight="1"
        android:gravity="center"
        android:text="texto 2" />
    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:width="0dip"
        android:layout_weight="1"
        android:gravity="center"
        android:text="texto 3" />
</LinearLayout>
```

Nótese que en este caso hemos añadido la propiedad *gravity* para centrar los textos dentro del TextView.



Figura 5. LinearLayout *proporcional*

RelativeLayout

Permite especificar la posición de cada una de las vistas hijas respecto al resto de los componentes (o respecto a la vista contenedora).

Es decir, podemos hacer cosas como “centra este componente en el layout” o “el componente A va a la izquierda de B y debajo de C”. Para ello, *RelativeLayout* define atributos como: *layout_toLeft*, *layout_below*, *layout_centerInParent*, etc.

Para referenciar a otra View usaremos su *id*.

En este caso “Texto1” lo hemos centrado dentro del parent (que está ocupando toda la pantalla).

“Texto2” va encima de “Texto1”, como no le hemos indicado otra cosa, se alinea a la izquierda.

“Texto3” se alinea a la derecha de “Texto2” y debajo de “Texto1”.

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



```
        android:layout_height="fill_parent">
<TextView android:text="texto 1"
        android:id="@+id/Texto1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerInParent="true"
        android:background="#F99"
        android:textColor="#000"
    />
<TextView android:text="texto 2"
        android:id="@+id/Texto2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_above="@id/Texto1"
        android:background="#9F9"
        android:textColor="#000"
    />
<TextView android:text="texto 3"
        android:id="@+id/Texto3"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_toRightOf="@id/Texto2"
        android:layout_below="@id/Texto1"
        android:background="#99F"
        android:textColor="#000"
    />
</RelativeLayout>
```

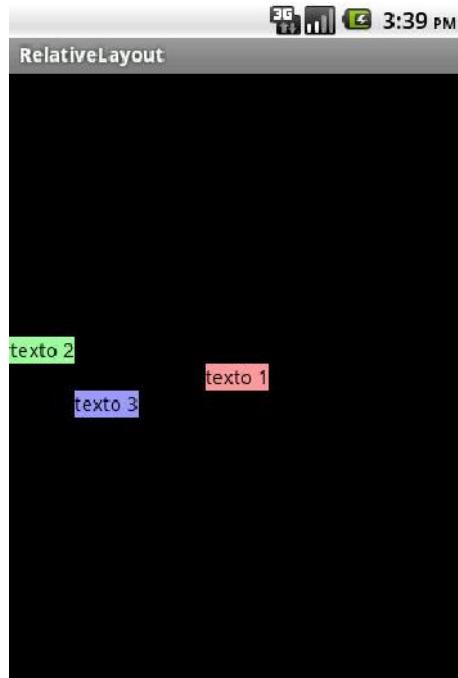


Figura 6. RelativeLayout

TableLayout

Posiciona sus hijos en formato de filas y columnas. Cada fila viene marcada por un objeto “TableRow” que a su vez contiene 0 o más objetos View que serán las columnas.

Los objetos View de las columnas pueden ser de diferente tipo. Así pues, podemos tener una tabla en la que una celda sea una imagen, otra un texto y otra un botón.

TableLayout también define atributos que nos permite decidir qué columnas deben estirarse (*stretchColumns*) o encogerse (*shrinkColumns*), también podemos ocultar columnas (*collapseColumns*).

```
<?xml version="1.0" encoding="utf-8"?>
    <TableLayout
        xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:stretchColumns="0" >
    <TableRow>
        <TextView
            android:text="texto 1"
            android:background="#55F"
            android:textColor="#000"/>
        <TextView android:text="texto 2"
            android:background="#99F"
            android:textColor="#000"/>
    </TableRow>
    <TableRow>
        <TextView android:text="texto 3"
            android:background="#F55"
            android:textColor="#000"/>
        <TextView android:text="texto 4"
            android:background="#F99"
            android:textColor="#000"/>
    </TableRow>
</TableLayout>
```

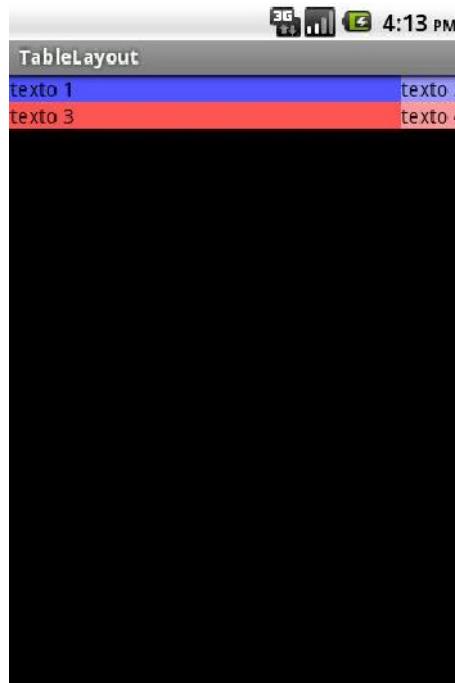


Figura 7. TableLayout

En este caso hemos definido una tabla de dos filas y dos columnas, marcando la primera columna (la 0) como “estirable”.

Resumen rápido de *layouts*

<i>FrameLayout</i>	Contenedor sencillo. No permite control sobre la posición de los hijos. Todo va arriba a la izquierda.
<i>Gallery</i>	Scroll horizontal que muestra imágenes a partir de una lista.
<i>GridView</i>	Muestra una rejilla de x columnas e y filas.
<i>LinearLayout</i>	Organiza sus hijos apilándolos en una fila o columna sencilla. Muestra un scroll si el contenido sobrepasa la pantalla.
<i>ListView</i>	Muestra un listado de una única columna a partir de un origen de datos.
<i>RelativeLayout</i>	Permite posicionar sus hijos posicionándolos respecto al resto.
<i>ScrollView</i>	Muestra un scroll sencillo
<i>Spinner</i>	Muestra un elemento sencillo que permite elegir uno entre una colección de datos.
<i>SurfaceView</i>	Provee acceso directo a la superficie de dibujo. Pensado para aplicaciones que necesitan dibujar a nivel de pixel, como los juegos
<i>TabHost</i>	Permite colocar pestañas que activan una Activiy u otra dependiendo de la que está seleccionada.
<i>TableLayout</i>	Muestra las vistas en un formato de filas y columnas, permitiendo controlar su aspecto.
<i>ViewFlipper</i>	Muestra un ítem cada vez, dentro de un campo de texto de una fila. Se puede configurar intervalos de tiempos a los que se intercambiará la vista.
<i>ViewSwitcher</i>	Similar a <i>ViewFlipper</i> .

3. VIEWS Y EVENTOS DE USUARIO

Como ya hemos mencionado todos los componentes de la vista descienden del objeto View, es decir, todos tienen un *interface* común con el que nuestro código puede interactuar.

Para capturar los eventos de usuario podemos definir clases centinela o “listeners” que implementan interfaces que son llamados cada vez que se produce un determinado evento.

Así pues con el método `setOnClickListener()` podremos definir qué clase responde cuando un usuario pulse en nuestra vista, con `setOnKeyListener()` la clase que responderá a eventos de teclado, etc.

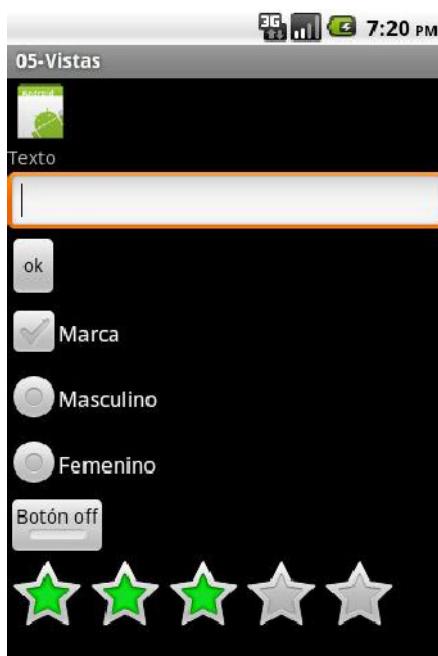


Figura 8. Objetos pulsables.



Ya conocemos el objeto `TextView` que nos permite mostrar textos.

Veamos otros objetos básicos de tipo `View` que nos permiten añadir más funcionalidades a nuestros interfaces y su forma básica de interactuar con el usuario.

ImageView

Permite mostrar imágenes. El origen las imágenes puede ser variado: desde un gráfico colocado en el directorio de recursos a proveedores de contenidos.

Tiene métodos y propiedades para controlar la manera en la que se mostrarán dichas imágenes (recortadas, estiradas, etc.).

```
<ImageView android:id="@+id/idImageView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:src="@drawable/icon" />
```

Definición de un objeto `View` que muestra el icono de la aplicación

A nivel de código podemos cambiar la imagen en tiempo de ejecución con los métodos `setImage`.

```
ImageView image = (ImageView) findViewById(R.id.idImageView);
image.setImageResource(R.drawable.icon);
```

Colocando una imagen del directorio de Resources en el `ImageView`

EditText

Los campos “`EditText`” son como los “`TextView`” pero con la propiedad “`editable`” a `true`, con lo que el usuario puede modificar su texto.

```
<EditText android:id="@+id/idEditText"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content" />
```

Definiendo un `EditText`

Podemos capturar sus cambios a través del evento “`onKey`”

Button

Permiten definir botones típicos que reaccionan cuando el usuario los pulsa.

```
//Manipulando el EditText
final EditText editText = (EditText)
findViewById(R.id.idEditText);
editText.setOnKeyListener(new OnKeyListener() {
    public boolean onKey(View v, int keyCode, KeyEvent event) {
// Si el evento es que la tecla enter se ha pulsado
        if ((event.getAction() == KeyEvent.ACTION_DOWN)
            && (keyCode == KeyEvent.KEYCODE_ENTER)) {
// Mostramos un mensaje de usuario con el texto tecleado
```

```

        Toast toast = Toast.makeText(editText.getContext(),
            editText.getText(), Toast.LENGTH_SHORT);
        toast.show();
        return true;
        return false;
    }
);

```

Cuando el usuario edite el texto y pulse “Enter” se mostrará un mensaje

Su definición XML:

```

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

```

Definiendo el botón en la definición del layout

Definiendo el comportamiento al pulsar:

```

// Manipulando el Boton
Button boton = (Button) findViewById(R.id.idBoton);
boton.setOnClickListener(new OnClickListener() {
    public void onClick(View v) {
        Toast.makeText(MainActivity.this, "boton pulsado",
        Toast.LENGTH_LONG).show();
    }
});

```

Definiendo el comportamiento de un botón

CheckBox

Define un botón de dos estados, pulsado y no pulsado, que cambia de una a otra posición conforme se pulsa o no.

```

<CheckBox android:id="@+id/idCheckbox"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Marca" />

```

Definiendo un checkbox

Aparte de poder capturar eventos como si fuera un botón, por ejemplo definiendo la clase OnClickListener, también podemos comprobar su estado en cada momento, consultando su método *isChecked()*.

```

// Manipulando el CheckBox
final CheckBox checkBox = (CheckBox)
findViewById(R.id.idCheckbox);
// Consultado si está pulsado
if (checkBox.isChecked()) {
// Cambiamos su estado
checkBox.setChecked(false);
}

```

Comprobando el estado de un CheckBox



RadioButton

Son botones de dos estados (pulsado y no pulsado). Al contrario que el *CheckBox*, los *RadioButton* una vez que pasan a estado pulsado, el usuario no puede hacerlos cambiar de estado.

Suelen usarse varios al mismo tiempo agrupados por un *RadioGroup*. De esta forma, al pulsar uno de ellos el resto pasa a modo “no pulsado”.

```
<RadioGroup  
    android:layout_width="fill_parent"  
    android:layout_height="wrap_content"  
    android:orientation="vertical">  
    <RadioButton android:id="@+id/idGenreMasculino"  
        android:layout_width="wrap_content"  
        android:layout_height="wrap_content"  
        android:text="Masculino" />  
    <RadioButton android:id="@+id/idGenreFemenino"  
        android:layout_width="wrap_content"  
        android:layout_height="wrap_content"  
        android:text="Femenino" />  
</RadioGroup>
```

Definiendo un conjunto de RadioButton

Podemos definir un “listener” común para todos los *RadioButton*.

```
// Manipulando los RadioButton  
OnClickListener radioListener = new OnClickListener() {  
    public void onClick(View v) {  
        RadioButton radio = (RadioButton) v;  
        Toast.makeText(MainActivity.this, radio.getText(),  
        Toast.LENGTH_SHORT).show();  
    }  
};  
RadioButton masculino = (RadioButton)  
findViewById(R.id.idGenreMasculino);  
masculino.setOnClickListener(radioListener);  
RadioButton femenino = (RadioButton)  
findViewById(R.id.idGenreFemenino);  
femenino.setOnClickListener(radioListener);
```

Definiendo el comportamiento del radiobutton

ToggleButton

Muestra un botón de dos estados con una luz que se ilumina cuando el estado es pulsado.

El texto también cambia dependiendo del estado.

```
final ToggleButton togglebutton = (ToggleButton)
findViewById(R.id.idToggleButton);
togglebutton.setOnClickListener(new OnClickListener() {
    public void onClick(View v) {
        ToggleButton tb = (ToggleButton) v;
        if (tb.isChecked()) {
            Toast.makeText(MainActivity.this, "Pulsado",
Toast.LENGTH_SHORT).show();
        } else {
            Toast.makeText(MainActivity.this, "No pulsado",
Toast.LENGTH_SHORT).show();
        }
    }
});
```

Manipulando un ToggleButton

```
<ToggleButton android:id="@+id/idToggleButton"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textOn="Botón on"
    android:textOff="Botón off"/>
```

Definición de un ToggleButton

RatingBar

Es una extensión de “ProgressBar” (barra de progreso) que permite mostrar una barra de evaluación con la que puntuar cosas.

El usuario puede elegir el número de estrellas que da y esto se convierte en un valor numérico que nuestro programa puede capturar.

Este es un buen ejemplo de componente que extiende la funcionalidad de otro, añadiendo nuevas propiedades y métodos.

Por ejemplo, en su definición XML podemos indicar el número de estrellas que se mostrarán y el valor de cada una de ellas.

```
<RatingBar android:id="@+id/idRatingbar"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:numStars="5"
    android:stepSize="1.0"/>
```



También define nuevos métodos para responder a eventos de usuario, en este caso para responder cuándo el usuario cambia la puntuación.

```
// Manejando un RatingBar
final RatingBar ratingbar = (RatingBar)
findViewById(R.id.idRatingbar);
ratingbar.setOnRatingBarChangeListener
(new OnRatingBarChangeListener() {
    public void onRatingChanged(RatingBar ratingBar, float rating,
        boolean fromUser) {
        Toast.makeText(MainActivity.this, "Puntuación: " + rating,
        Toast.LENGTH_SHORT).show();
    }
});
```

Otros componentes View

Aparte de los aquí mencionados, existen otros muchos componentes de tipo View que podemos usar en nuestros interfaces como WebView, VideoView, MediaController, ProgressBar y muchos otros.

Con cada nueva versión del sistema operativo, se incorporan nuevos. Se recomienda leer la documentación² para conocer la lista completa y forma de uso de todos los componentes.

2 <http://developer.android.com/reference/android/view/View.html>.

4. COLECCIONES DE DATOS Y VIEWS

Una de las tareas más típicas con las que un programador se enfrenta es la de mostrar colecciones de datos (listados, galerías fotográficas, etc.), presentándolas de forma amigable al usuario.

Para facilitarnos el trabajo, Android nos provee una serie de clases que derivan de AdapterView.

AdapterView es una subclase de ViewGroup en la que sus Views hijas vienen definidas por un adaptador de datos (Adapter).

Adapter es un interfaz que define una forma común de acceso a las colecciones.

La forma más básica de usar estos Adapter es utilizar alguno de los Adapter implementados en Android como por ejemplo ArrayAdapter que nos permite enlazar colecciones de datos de tipo Array.

Ejemplo

Primero definimos un *Layout* que contenga la vista de tipo ListView (donde se mostrará el listado).

Para identificarlo vamos a usar un *id* ya definido en android: @android:id/list.

También hemos añadido una View de texto también con un *id* predefinido @android/empty que usaremos más adelante.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="vertical"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent">
    <ListView android:id="@+id/list"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent" />
    <TextView android:id="@+id/empty"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
</LinearLayout>
```



```
        android:layout_width="fill_parent"
        android:layout_height="fill_parent" />
<TextView android:id="@+id/empty"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent" />
</LinearLayout>
res/layout/simplelist.xml
```

Veamos la Activity que enlaza nuestros datos con esta vista.

```
package es.masterd.cap05;
import android.app.Activity;
import android.os.Bundle;
import android.widget.ArrayAdapter;
import android.widget.ListAdapter;
import android.widget.ListView;
public class ArrayAdapterActivity extends Activity {
@Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.simplelist);
        // Nuestra colección de datos
        String[] listado = new String[] { "item 1", "item 2",
        "item 3", "item 4" };
        // Creamos un Adapter para acceder a los datos de nuestro listado
        // Cada item se mostrará en un view definido por Android
        // (simple_list_item_1)
        ListAdapter adapter = new ArrayAdapter<String>(this,
            android.R.layout.simple_list_item_1, listado);
        // Enlazamos nuestro adapter con nuestra vista
        ListView listView = (ListView)
            findViewById(android.R.id.list);
        listView.setAdapter(adapter);
    }
}
```

Partiendo de un “array de String” con nuestros datos, hemos creado un ArrayAdapter al que le hemos tenido que pasar el contexto, la vista que se usará para dibujar cada ítem (en este caso reutilizamos una definida por Android) y la colección de datos a mostrar.

Finalmente hemos localizado la vista de tipo ListView que hemos definido en el Layout y le hemos asociado el Adapter.

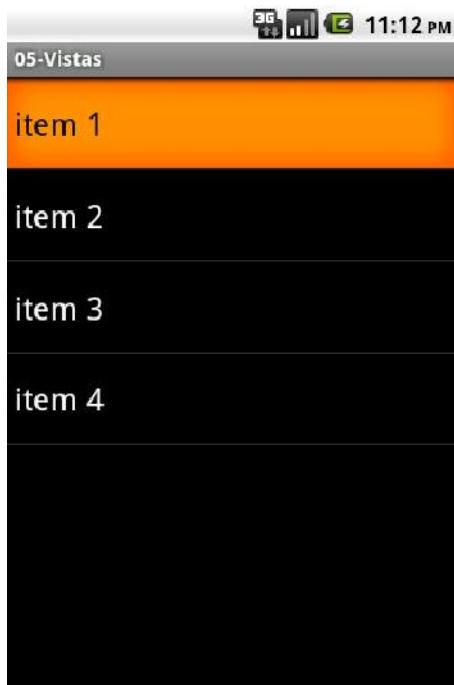


Figura 9. ArrayAdapter en funcionamiento

Cuando el usuario pulsa sobre un ítem se dispara el evento “onItemClick” que podemos capturar con el método `ListView.setOnItemClickListener()`.

```
// Mostramos un mensaje cuando el usuario pulse un ítem
listView.setOnItemClickListener(new OnItemClickListener() {
    public void onItemClick(AdapterView<?> parent, View view,
    int position, long id) {
        ListAdapter la = (ListAdapter) parent.getAdapter();
        String texto = (String) la.getItem(position);
        Toast.makeText(view.getContext(), texto,
        Toast.LENGTH_SHORT).show();
    }
});
```

Gestionando el evento OnItemClick

Para facilitarnos aún más la vida, Android nos provee de una clase Activity especiales que integra la funcionalidad de las listas llamada `ListActivity`.

Esta clase espera que la vista asignada tenga una view con identificador `android:id/list` y otra con el identificador `android:id/empty`.

Cuando el adaptador no tiene datos mostrará la vista “empty”.

Así pues, el código anterior usando una `ListActivity` quedaría de la siguiente forma:

```
package es.masterd.cap05;
import android.app.ListActivity;
import android.os.Bundle;
import android.view.View;
```



```
import android.widget.ArrayAdapter;
import android.widget.ListAdapter;
import android.widget.ListView;
import android.widget.Toast;
public class MiListActivity extends ListActivity {
@Override
protected void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.simplelist);
// Nuestra colección de datos
String[] listado = new String[] { "item 1", "item 2",
"item 3", "item 4" };
// Creamos un Adapter para acceder a los datos de nuestro listado
// Cada item se mostrará en un view definido por android
// (simple_list_item_1)
ListAdapter adapter = new ArrayAdapter<String>(this,
android.R.layout.simple_list_item_1, listado);
// Enlazamos nuestro adapter con nuestra vista
setListAdapter(adapter);
}
@Override
protected void onListItemClick(ListView l, View v,
int position, long id) {
super.onListItemClick(l, v, position, id);
String texto = (String) getListAdapter().getItem(position);
Toast.makeText(MiListActivity.this, texto,
Toast.LENGTH_SHORT).show();
}
}
```

Pero... ¿qué pasa si queremos mostrar listados más elaborados en los que hay que mostrar más información por ítem?

En ese caso, no nos queda otro remedio que escribirnos nuestro propio adaptador.

Imaginemos que queremos mostrar un listado con las versiones de Android. Por cada versión se mostrará su logotipo, el número de versión y el nombre de la misma.

Suponiendo que las imágenes que vamos a usar son recursos (estarán en la carpeta “res/drawable”) la siguiente clase nos valdría para contener la información de una versión:

```
package es.masterd.cap05.miadapter;
public class AndroidVersion {
String nombre;
String version;
int logo;
public AndroidVersion(String nombre, String version,
int logo) {
super();
this.nombre = nombre;
this.version = version;
this.logo = logo;
}
public String getVersion() {
return version;
```

```

    }
    public void setVersion(String version) {
        this.version = version;
    }
    public String getNombre() {
        return nombre;
    }
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
    public int getLogo() {
        return logo;
    }
    public void setLogo(int logo) {
        this.logo = logo;
    }
}

```

La colección de versiones se podría definir como un ArrayList:

```

ArrayList<AndroidVersion> versiones = new ArrayList
<AndroidVersion>();
versiones.add(new AndroidVersion("Android", "", 
R.drawable.logo));
versiones.add(new AndroidVersion("CupCake", "1.5",
R.drawable.cupcake));
versiones.add(new AndroidVersion("Donut", "1.6",
R.drawable.donut));
versiones.add(new AndroidVersion("Eclair",
"2.0/2.1", R.drawable.eclair));

```

Aunque bastaría con extender la clase ArrayList y sobrescribir el método *getView()*, en este caso vamos a escribir un adaptador completo. Para ello, extendemos la clase BaseAdapter.

```

public class VersionesAdapter extends BaseAdapter {
    private ArrayList<AndroidVersion> versiones;
    private LayoutInflator mInflater;
    public VersionesAdapter(Context context,
    ArrayList<AndroidVersion> vers) {
        this.mInflater = LayoutInflator.from(context);
        this.versiones = vers;
    }
    public int getCount() { return versiones.size(); }
    public AndroidVersion getItem(int position) {
        return versiones.get(position);
    }
    public long getItemId(int position) { return position; }
    public View getView(int position, View convertView,
    ViewGroup parent) {
        ViewHolder holder = null;
        if (convertView == null) {
            convertView = mIn-
flater.inflate(R.layout.lista_versionitem, null);
            holder = new ViewHolder();
            holder.hNombre = (TextView)
            convertView.findViewById(R.id.idNombre);
            holder.hVersion = (TextView) convertView

```



```
        convertView.findViewById(R.id.idVersion);
        holder.hImage = (ImageView)
        convertView.findViewById(R.id.idLogo);
        convertView.setTag(holder);
    } else {
        holder = (ViewHolder) convertView.getTag();
    }
    AndroidVersion version = getItem(position);
    holder.hNombre.setText(version.getNombre());
    holder.hVersion.setText(version.getVersion());
    holder.hImage.setImageResource(version.getLogo());
    return convertView;
}
class ViewHolder {
    TextView hNombre;
    TextView hVersion;
    ImageView hImage;
}
}
miadapter.VersionesAdapter
```

El constructor de la clase carga un objeto “LayoutInflater” y para optimizar el código lo cachea. Será el encargado de “inflar” la definición XML de *Layout* y convertirlo en un objeto java que podamos manejar.

También se cargan los datos en un campo interno para poder acceder luego a ellos.

Los métodos *getItem()*, *getItemId()* y *getCount()* son métodos sencillos que devuelven el ítem, el identificador de ítem y el número de ítem de la colección respectivamente.

El método *getView()* se llama cada vez que hay que dibujar la lista (por ejemplo, al hacer *scroll*), es el encargado de dibujar una fila de la lista. Debido a que se llama tantas veces, es necesario que sea muy rápido para dar sensación de fluidez en nuestro interfaz.

En este caso, hemos utilizado dos mecanismos: el primero es cachear el *LayoutInflater* al construir el Adapter de esta forma no hay que crear un objeto cada vez que se redibuja.

Además utilizamos una clase contendor (*ViewHolder*) que asociamos a la vista (en su propiedad *tag*), de esta forma, si la vista ya ha sido creada no hace falta volver a crearla y volver a localizar los campos dentro de ella.

Esta clase infla el layout que hemos definido con un XML para presentar los datos.

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

```

<ImageView android:id="@+id/idLogo"
    android:layout_width="72dip"
    android:layout_height="72dip"
    android:src="@drawable/donut"
    android:padding="5dip"
/>
<TextView android:id="@+id/idVersion"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="texto 2"
    android:paddingTop="5dip"
    android:layout_toRightOf="@+id/idLogo"
/>
<TextView android:id="@+id/idNombre"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="texto 3"
    android:layout_toRightOf="@+id/idLogo"
    android:layout_below="@+id/idVersion"
/>
</RelativeLayout>

```

listaversion_item.xml

Solo queda usar nuestro Adapter en un ListActivity tal y como hemos hecho antes:

```

package es.masterd.cap05.miadapter;
import java.util.ArrayList;
import android.app.ListActivity;
import android.os.Bundle;
import es.masterd.cap05.R;
public class VersionesActivity extends ListActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.simplelist);
        ArrayList<AndroidVersion> versiones =
            new ArrayList<AndroidVersion>();
        versiones.add(new AndroidVersion(
            "Android", "", R.drawable.logo));
        versiones.add(new AndroidVersion
            ("CupCake", "1.5", R.drawable.cupcake));
        versiones.add(new AndroidVersion
            ("Donut", "1.6", R.drawable.donut));
        versiones.add(new AndroidVersion
            ("Eclair", "2.0/2.1", R.drawable.eclair));
        VersionesAdapter adapter = new VersionesAdapter(
            VersionesActivity.this, versiones);
        setListAdapter(adapter);
    }
}

```

miadapter.VersionesActivity

Los adapter son usados por muchos controles por lo que, si se programan bien, pueden ser reutilizados.

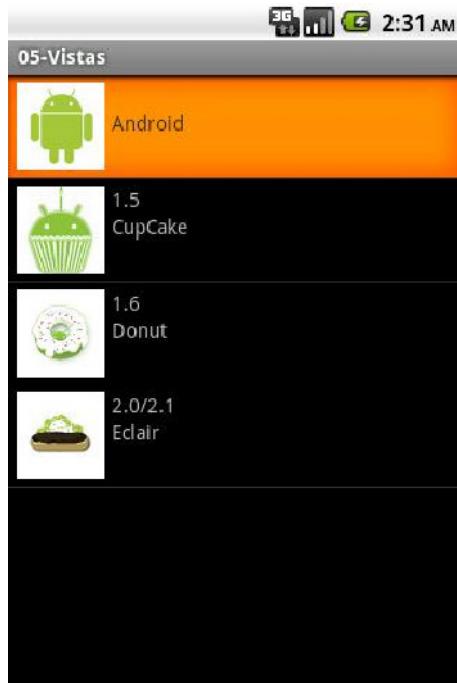


Figura 10. VersionesAdapter en acción

Imaginemos que queremos mostrar la información de versiones usando un control *Gallery*, que muestra una tira de *Views* con *scroll* horizontal y coloca la seleccionada en el centro.

Pues bien, bastaría con editar el *layout* y sustituir el *ListView* por una vista tipo *Gallery* y crear una *Activity* que las enlazara:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:
    android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
<Gallery android:id="@+id/list"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:spacing="2dip" />
</LinearLayout>
```

res/layout/gallery.xml

```
public class VersionesGalleryActivity extends Activity {
    @Override
        protected void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
            // setContentView(R.layout.simplelist);
            setContentView(R.layout.gallery);
            ArrayList<AndroidVersion> versiones =
                new ArrayList<AndroidVersion>();
            versiones.add(new AndroidVersion
```

```

        ("Android", "", R.drawable.logo));
versiones.add(new AndroidVersion(
"CupCake", "1.5", R.drawable.cupcake));
versiones.add(new AndroidVersion(
"Donut", "1.6", R.drawable.donut));
versiones.add(new AndroidVersion
("Eclair", "2.0/2.1", R.drawable.eclair));
Gallery gallery = (Gallery)
findViewById(android.R.id.list);
VersionesAdapter adapter =
new VersionesAdapter
(VersionesGalleryActivity.this, versiones);
gallery.setAdapter(adapter);
}
}

miadapter.VersionesGalleryActivity

```

Como se puede apreciar, con solo cambiar unas pocas líneas de código el aspecto de nuestra aplicación ha cambiado radicalmente.

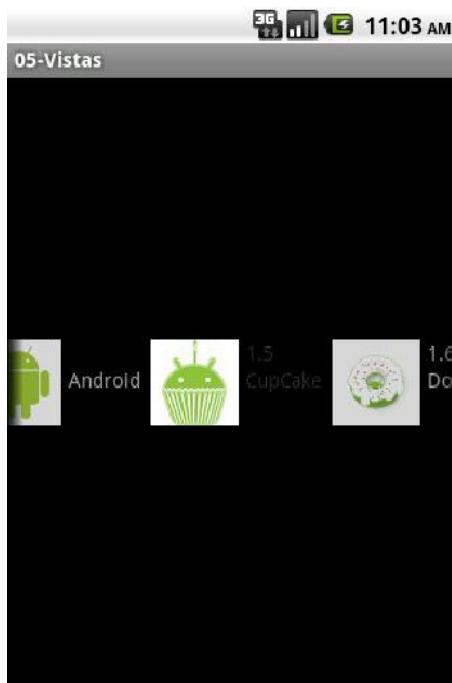


Figura 11. Componente Gallery, utilizando nuestro Adapter

Otros componentes que usan adapters son: ExpandableListView, GridView o Spinner.



5. MENÚS

Los menús son un componente típico al que los usuarios están acostumbrados, tanto que la mayoría de los terminales Android incorporan un botón para desplegarlos.

Android nos provee de un interfaz de programación para gestionar los diferentes menús:

- *Options Menu*: el menú principal de la Activity. Se despliega cuando el usuario pulsa la tecla menú. Se pueden dividir en dos grupos:
 - *Icon Menu*: muestra un menú con ícono. Soporta un máximo de seis ítems y no puede contener checkboxes o radiobutton.
 - *Expanded Menu*: → se muestra cuando se superan el número de ítems máximo y se muestra el botón de “más”.
- *Context Menu*: → son los menús contextuales que se pueden desplegar cuando el usuario realiza una pulsación prolongada sobre un objeto View.
- Submenus: → se trata de menús que se despliegan al pulsar un ítem de un menú.

Definiendo menús

La forma más rápida de definir un menú es a través de un fichero xml de recursos. Se coloca en res/menu/.

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:
       android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/menuAcercaDe"
          android:title="Acerca de"
          android:icon="@drawable/ic_menu_about" />
    <item android:id="@+id/menuQuit"
          android:title="Salir"
          android:icon="@drawable/ic_menu_quit" />
</menu>
```

Definición de un menú como recurso (res/menu/menuprincipal.xml)

Por cada ítem, definimos su identificador (que luego usaremos a través de la clase R), su ícono y el texto de su etiqueta.

Para usar este menú deberemos “inflarlo” (convertir el xml en un objeto) con una clase “Inflater”, en este caso con “MenuInflater”.

Como queremos que el menú sea un menú de actividad, sobrescribiremos el método *onCreateOptionsMenu()* de nuestra Activity.

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

"Inflando" el menú de nuestra actividad

El siguiente paso es capturar la respuesta del usuario. Para ello, como es un menú de Activity, sobrescribimos el método *onOptionsItemSelected()*.

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case R.id.menuAcercaDe:
            String aboutTxt = "(c) 2010 Master-D";
            Toast.makeText(VersionesGalleryActivity.this,
                           aboutTxt, Toast.LENGTH_LONG).show();
        return true;
        case R.id.menuQuit:
            finish();
        return true;
        default:
        return false;
    }
}
```

Gestionando la respuesta del usuario

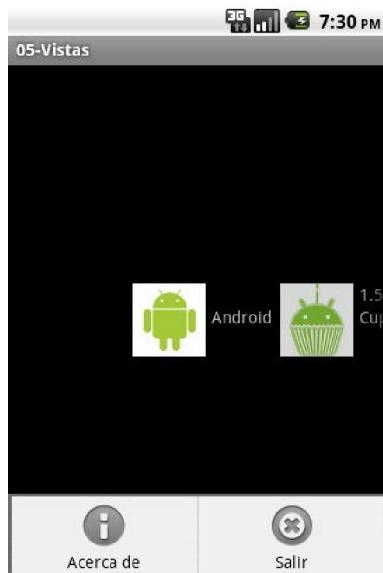


Figura 12. Aspecto del menú de actividad

Los menús contextuales se gestionan de forma similar, pero sobrescribiendo los métodos correspondientes de la vista que disparará el menú contextual:

- `onCreateContextMenu()`: → para inflar el menú.
- `onContextItemSelected()`: para gestionar las selecciones.

Para crear submenús basta con colocar un *tag* de menú dentro de uno de ítem:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/
       apk/res/android">
    <item android:id="@+id/menuAcercaDe"
          android:title="Acerca de"
          android:icon="@drawable/ic_menu_about" />
    <item android:id="@+id_submenu"
          android:title="Archivo">
        <menu>
            <item android:id="@+id/menuArchivoNuevo"
                  android:title="Nuevo" />
            <item android:id="@+id/menuArchivoAbrir"
                  :title="Abrir" />
        </menu>
    </item>
</menu>
```

Ejemplo de definición de un submenú

Los menús también permiten usar el *tag* `<group>` para agrupar opciones, activar/desactivar elementos, generar menús automáticos basándose en filtros de Intents, etc.

Se recomienda consultar la documentación oficial para ver todas las opciones.

6. DIÁLOGOS Y NOTIFICACIONES

Ciertas situaciones nos obligan a mostrar mensajes a los usuarios informándoles de diferentes tipos de avisos.

Android nos ofrece tres formas de mostrar notificaciones al usuario:

- Notificaciones sobreimpresas.
- Notificaciones en la barra de estado.
- Ventanas de diálogo.

Notificaciones sobreimpresas (*Toast Notifications*)

Son avisos que se muestran sobreimpresos en la pantalla. Sólo ocupan el espacio que necesitan para mostrar el aviso.

Automáticamente desaparecen pasado un lapso de tiempo determinado.

Ya hemos usado este tipo de avisos en ejemplos anteriores. Para crearlo solo necesitamos pasar el contexto de la aplicación, el texto y la duración.

```
Context context = getApplicationContext();
Toast aviso = Toast.makeText(context, "Pulsado",
    Toast.LENGTH_SHORT);
aviso.show();
```

Mostrando un aviso Toast

Para indicar la duración del mensaje en pantalla usamos una constante predefinida (en este caso corta duración)



Las ventanas de *Toast* se pueden personalizar. Para ello debemos definir su aspecto con un layout xml que luego inflaremos y asignaremos a la vista raíz del *Toast*:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/toastRaiz"
    android:orientation="horizontal"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:background="#EFAA"
    android:padding="5dip"
>
<ImageView android:id="@+id/imagen"
    android:layout_width="72dip"
    android:layout_height="72dip"
    android:src="@drawable/donut"
    android:padding="5dip"
/>
<TextView android:id="@+id/texto"
    android:layout_width="wrap_content"
    android:layout_height="fill_parent"
    android:text="texto"
    android:textColor="#f00"
    android:gravity="center_vertical"
/>
</LinearLayout>
```

res/layout/mi_toast.xml

Obsérvese que al *LinearLayout* contenedor le hemos asignado un *id* único que luego usaremos al inflarlo:

Nuestro *Toast* tendrá un aspecto similar a este:

```
// Cargamos la vista personalizada
ViewGroup vistaRaiz = (ViewGroup)
findViewById(R.id.toastRaiz);
LayoutInflater inflater = getLayoutInflater();
View layout = inflater.inflate(R.layout.mi_toast, vistaRaiz);
// Cambiamos el texto (buscamos el label y lo modificamos)
TextView texto = (TextView)
layout.findViewById(R.id.texto);
texto.setText("Esto es un aviso personalizado");
// Creamos el toast y le asignamos la vista que hemos personalizado
Toast toast = new Toast(getApplicationContext());
toast.setView(layout);
// Lo centramos en pantalla
toast.setGravity(Gravity.CENTER_HORIZONTAL
| Gravity.CENTER_VERTICAL, 0, 0);
// Permanecerá más rato en pantalla
toast.setDuration(Toast.LENGTH_LONG);
// Lo mostramos
toast.show();
```

Mostrando un *Toast* con un aspecto personalizado

Notificaciones en la barra de estado (*Status Bar*)

Este tipo de avisos muestran un ícono en la barra de estado acompañado de un texto. Cuando el usuario pulsa sobre ellos el sistema dispara un Intent (normalmente una Activity).

Un ejemplo claro de notificación en la barra de estado son los avisos que se muestran cuando llega un e-mail nuevo.

Para mostrar un aviso en la barra de estado debemos usar el gestor de notificaciones del sistema (NotificationManager).

Luego debemos crear un objeto “Notification” en el que definiremos el ícono, el mensaje corto que se mostrará y el momento en el que debe mostrarse.

Esta información básica puede ser complementada con un título y un texto que se mostrará al desplegar la barra de estado y con un Intent que se disparará cuando el usuario pulse sobre dichos mensajes.

```
// Conseguimos una referencia al servicio encargado
// de mostrarlos
String ns = Context.NOTIFICATION_SERVICE;
NotificationManager notificationManager =
(NotificationManager) getSystemService(ns);
// Definimos el aviso
int icono = R.drawable.notification_icon;
CharSequence aviso = "Gooooooool !!!";
long ahora = System.currentTimeMillis();
Notification notification = new Notification(
(icono, aviso, ahora));
// Definimos los detalles del aviso
Context context = getApplicationContext();
CharSequence titulo = "Gol en la Romareda";
CharSequence texto = "Zaragoza 1 - Racing 0";
Intent notificationIntent = new Intent(this,
MainActivity.class);
PendingIntent contentIntent = PendingIntent.getActivity(this, 0,
notificationIntent, 0);
notification.setLatestEventInfo(context, titulo, texto,
contentIntent);
// Pasamos el aviso al manager
final int GOLNOTIFICACION_ID = 1;
notificationManager.notify(GOLNOTIFICACION_ID, notification);
Lanzando un aviso a través de la barra de estado
```

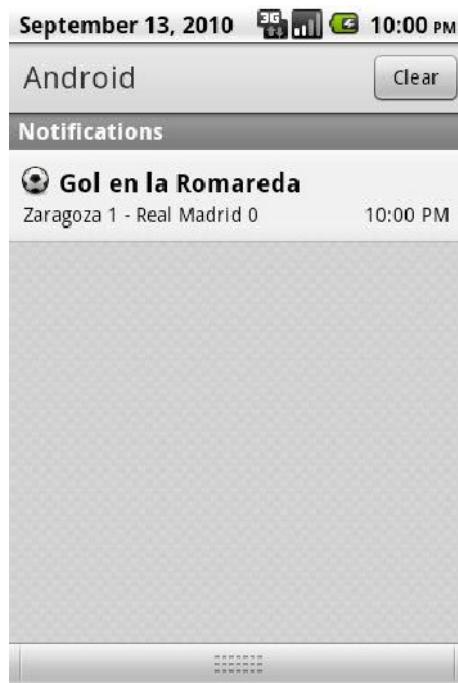


Figura 13. Los detalles se ven al desplegar la barra

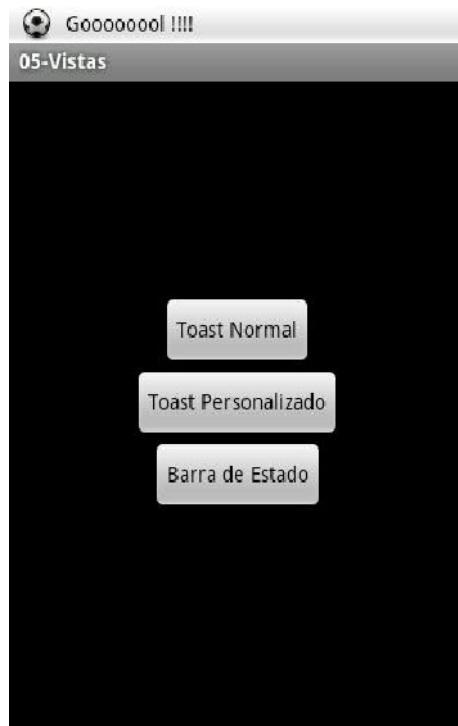


Figura 14. Notificación mostrada en la barra de estado

Las notificaciones también se pueden personalizar con objetos *layouts* propios, modificar los mensajes, asignar acciones como vibración, sonidos, parpadeo de *leds*, etc.

Diálogos (*Dialog*)

Los diálogos son pequeñas ventanas que se muestran delante de nuestras Actividades recibiendo el foco. Pueden recibir interacción por parte del usuario.

Vienen a ser como pequeñas actividades que realizan tareas secundarias de la actividad actual.

La clase encargada de gestionarlas es *Dialog* de la que heredan varias clases especializadas que facilitan su uso:

- *AlertDialog*
- *ProgressDialog*
- *DatePickerDialog*
- *TimePickerDialog*

Por supuesto, si estas clases no cubren nuestras necesidades, podemos extender la clase *Dialog* y personalizarla a nuestro gusto.

Como los diálogos se muestran desde una actividad, la clase *Activity* incorpora una serie de métodos para gestionarlos:

- *onCreateDialog(int)*: se llama la primera vez que se invoca el diálogo, es el encargado de crearlo.
- *onPrepareDialog(int)*: se llama justo antes de mostrarlo.
- *showDialog(int)*: muestra el diálogo.
- *dismissDialog(int)*: cierra el diálogo, la actividad guarda su estado.
- *removeDialog(int)*: cierra el diálogo y lo elimina completamente.

Todos los métodos reciben un número entero. La mejor forma de gestionar los diálogos es usar un switch dentro de estos métodos para gestionar cada tipo de diálogo.

El siguiente ejemplo sobrescribe el método *onCreateDialog()* que gestiona dos diálogos:

- Un *AlertDialog* que muestra un mensaje y dos botones, cuando se pulsa el botón de “sí” la actividad se cierra, con el “no” cierra el diálogo.
- El otro es un diálogo de progreso indeterminado (rueda de progreso).

```
/*
 * Crea los diálogos
 */
@Override
protected Dialog onCreateDialog(int id) {
    Dialog newDialog = null;
    switch (id) {
        case DIALOG_ALERT:
            AlertDialog.Builder builder =
                new AlertDialog.Builder(this);
            builder.setMessage("¿Quieres salir?")
                .setCancelable(false)
                ..setPositiveButton("Si",
                    new DialogInterface.OnClickListener() {
                        public void onClick(DialogInterface dialog, int id) {
                            DiálogosActivity.this.finish();
                        }
                    })
                .setNegativeButton("No",
                    new DialogInterface.OnClickListener() {
                        public void onClick(DialogInterface dialog, int id) {
                            dialog.cancel();
                        }
                    });
            newDialog = builder.create();
            break;
        case DIALOG_PROGRESS:
            newDialog = ProgressDialog.show(DiálogosActivity.this, "",
                "Cargando, espere...", true);
            break;
    }
    return newDialog;
}
```



Figura 15. showDialog(DIALOG_PROGRESS)

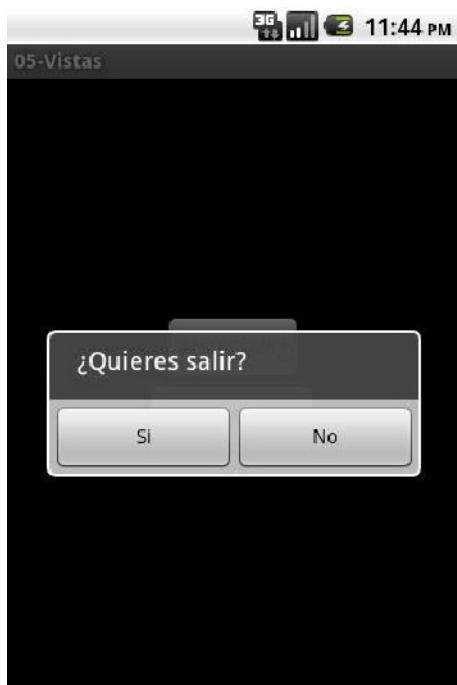


Figura 16. showDialog(DIALOG_ALERT)

Como se puede apreciar, para la creación de diálogos Android implementa el patrón de diseño Builder. A través de él podemos crear el diálogo completo con todas sus características.

La potencia de los diálogos no queda aquí: podemos hacer diálogos de progreso de tipo barra que actualizan el progreso desde la Actividad, podemos añadir colecciones de datos a nuestros diálogos, podemos cambiarles completamente el aspecto, los detalles en la documentación oficial...



7. ESTILOS Y TEMAS

En HTML, con las hojas de estilo CSS podemos cambiar por completo el aspecto de una página web, separando de esta forma la estructura del diseño.

Android nos ofrece un mecanismo similar para poder personalizar el aspecto de nuestra aplicación: los temas y estilos.

Los estilos pueden definir propiedades como tamaños de letra, colores, fondos, sombras y muchas otras cosas.

Se definen en un fichero de recursos distinto al *layout*. Como se trata de definiciones sencillas este fichero irá en el directorio res/values/.

Los ficheros de estilos no tienen por qué tener un nombre en concreto, aunque por claridad suelen llamarse styles.xml y/o themes.xml.

Son ficheros XML en los que el nodo raíz debe ser <resources> y que contienen elementos <style> e <item>.

Estos ficheros en la compilación se convierten en un objeto de recursos que podremos llamar a través de su id definido en la propiedad “name” del nodo “style”.

Los estilos pueden aplicarse cada objeto View individualmente o a Actividades enteras, dependiendo si se hace en el fichero de *layout* o en el manifiesto.

Los estilos se aplican en la etiqueta “style”:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    <TextView android:text="Título 1"
        android:layout_width="fill_parent"
```

```

        android:layout_height="wrap_content"
        style="@style/Titulo" />
<TextView android:text="Título 2"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        style="@style/Titulo.Verde"/>
</LinearLayout>

```

Layout que usa estilos

En este ejemplo, al primer TextView le aplicamos el estilo llamado “Titulo” y al segundo el llamado “Titulo.Verde”.

Estos estilos deberán estar definidos en res/values:

```

<?xml version="1.0" encoding="utf-8"?>
<resources>
<style name="Titulo"
    parent="@android:style/TextAppearance.Large">
    <item name="android:textStyle">bold</item>
    <item name="android:textColor">#FE5F35</item>
    <item name="android:padding">5dip</item>
    <item name="android:gravity">center_horizontal</item>
    <item name="android:shadowColor">#99aaaaaa</item>
    <item name="android:shadowDx">1.0</item>
    <item name="android:shadowDy">1.0</item>
    <item name="android:shadowRadius">0.5</item>
</style>
<style name="Titulo.Verde">
    <item name="android:textColor">#009900</item>
</style>
</resources>

```

res/values/styles.xml.



Figura 17. La vista con estilos



Cada estilo debe heredar de otro, es decir, debe tener un parent. Esto se puede definir de dos formas:

- Con la etiqueta “parent”, como el caso del estilo Titulo que hereda del estilo definido por Android TextAppearance.Large (texto grande).
- Otra forma es la implícita en el nombre como el caso de “Titulo.Verde” que indica que el parent es el estilo llamado “Titulo”, es decir hereda todas las propiedades de “Titulo” y sobrescribe la del color de texto.

Los estilos, tal y como los hemos definido, están bien pero nos obligan a aplicarlos uno a uno a nuestros objetos View.

Hemos mencionado que los estilos también se pueden aplicar a una Activity por lo que deberíamos poder transformar el aspecto sin tener que editar el *layout*. Esto se consigue con los temas.

Para definir un tema sobrescribimos las propiedades definidas en los atributos de Android y les asignamos nuevos aspectos:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
<style name="MiTema" parent="@android:style/Theme.Light">
<item name="android:windowBackground">@drawable/fondo</item>
<item name="android:windowTitleStyle">@style/Titulo</item>
<item name="android:buttonStyle">@style/MiBoton</item>
</style>
</resources>
```

res/values/themes.xml

En este caso nuestro tema cambia el fondo de la ventana poniendo una imagen y aplica un estilo propio al texto de la barra de título y a los botones.

Estos estilos propios deberán estar definidos también:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
    xmlns:android="http://schemas.android.com/apk/res/android"
        package="es.masterd.cap05"
        android:versionCode="1"
        android:versionName="1.0">
    <application android:icon="@drawable/icon"
        android:label="@string/app_name">
        <activity
            android:name=".MainActivity"
            android:theme="@style/MiTema"
            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>
```

AndroidManifest.xml

```

<?xml version="1.0" encoding="utf-8"?>
<resources>
<style name="Titulo"
parent="@android:style/TextAppearance.Small">
    <item name="android:textStyle">bold</item>
    <item name="android:textColor">#FE5F35</item>
    <item name="android:gravity">center_horizontal</item>
    <item name="android:shadowColor">#99000000</item>
    <item name="android:shadowDx">1.0</item>
    <item name="android:shadowDy">1.0</item>
    <item name="android:shadowRadius">0.5</item>
</style>
<style name="MiBoton"
parent="@android:style/Widget.Button">
    <item name="android:background">
        @drawable/bg_verde</item>
    <item name="android:textAppearance">
        ?android:attr/textAppearanceSmall</item>
        <item name="android:textColor">#afa</item>
    </style>
</resources>

```

res/values/styles.xml

Para aplicarlo bastará con referenciarlo en la propiedad “`android:theme`” de una actividad definida en el Manifiesto:

Con solo cambiar la propiedad “`theme`”, el *layout* que hemos usado como ejemplo al principio del capítulo tendrá este aspecto:



MASTER.D

Figura 18. Layout inicial con un tema aplicado



Como se puede ver los atributos que hemos definido en nuestro tema se han aplicado a los objetos View correspondientes el resto han heredado el aspecto del tema padre (*Theme.Light*).

Pero ¿qué atributos podemos modificar? y ¿cómo hacerlo?

La mejor forma de responder a estas preguntas es ver el código fuente de los temas que Android nos proporciona.

De hecho es una buena idea que nuestros temas sean hijos de alguno de los predefinidos, de esta forma si nos olvidamos de definir algún atributo heredaremos las propiedades por defecto.

Los estilos y temas definidos por Android podemos consultarlos el repositorio de código fuente Android (ventajas de usar una plataforma OpenSource):

- <http://android.git.kernel.org/>.

En concreto los estilos están en:

- <http://android.git.kernel.org/?p=platform/frameworks/base.git;a=blob;f=core/res/res/values/styles.xml>.

Y los temas en:

- <http://android.git.kernel.org/?p=platform/frameworks/base.git;a=blob;f=core/res/res/values/themes.xml>.



CONCLUSIONES



Dispones del equipo de tutores de **Master.D** para ayudarte en todo lo que necesites. Ponte en contacto con nosotros.

Como se ha podido comprobar a lo largo de este capítulo Android nos ofrece una serie de herramientas muy potentes con las que podemos crear interfaces de usuario muy potentes y amigables de forma muy sencilla.

El límite está en nuestra imaginación (y en nuestras dotes de diseño).

Cuando diseñas interfaces con Android ten presente siempre que estás desarrollando para dispositivos móviles y táctiles, así que intenta facilitar la vida al usuario: usa botones grandes, textos con colores adecuados, etc.

RECAPITULACIÓN

En este capítulo hemos hablado principalmente del objeto básico `View` que es a partir del que se crean el resto de componentes de la interfaz de usuario Android. Hemos visto `View` especializados como `Button`, `TextView`, `EditText`...

Hemos aprendido que gracias a los *Layouts* podemos maquear estos componentes. Nos hemos detenido en alguno de ellos (`LinearLayout`, `FrameLayout`, `RelativeLayout`...).

Hemos aprendido que los objetos `AdapterView` nos permiten presentar colecciones de datos fácilmente usando `Adapters`. Como ejemplo, hemos usado `ListView` y hemos conocido un tipo de actividad especializada en su uso: `ListView`.

También hemos aprendido a usar menús y diálogos de usuario en todas sus variantes.

Finalmente, hemos descubierto que gracias a los estilos y temas podemos cambiar totalmente el aspecto estético de nuestras aplicaciones.



AUTOCOMPROBACIÓN

- 1. ¿De qué forma se organizan los objetos View en Android?**
 - a) En forma lineal.
 - b) En forma secuencial.
 - c) En forma de árbol.
 - d) No tienen un orden fijo.

- 2. ¿Cómo podemos definir nuestro interface de usuario?**
 - a) Mediante un fichero XML.
 - b) Con código Android.
 - c) Mediante un fichero XML y/o código Android
 - d) Mediante un fichero de tipo png.

- 3. ¿A qué método de nuestra actividad debemos llamar para establecer nuestro objeto View principal?**
 - a) `setMainView`.
 - b) `setContentView`.
 - c) `setRootView`.
 - d) `setView`.

4. ¿En cuántas pasadas se dibuja un objeto tipo View?

- a) En una.
- b) En dos.
- c) En tres.
- d) En cuatro.

5. ¿Qué función se llama en la primera pasada del dibujo?

- a) *getPosition*.
- b) *draw*.
- c) *measure*.
- d) *layout*.

6. ¿Qué función se llama en la segunda pasada del dibujo?

- a) *getPosition*.
- b) *draw*.
- c) *measure*.
- d) *layout*.

7. Para que un objeto se pueda dibujar ¿qué no le debemos pasar?

- a) *AUTO*.
- b) *FILL_PARENT*.
- c) *WRAP_CONTENT*.
- d) Una dimensión exacta.

8. ¿Para identificar cada uno de los componentes de la vista que atributo utilizamos?

- a) *id*.
- b) *name*.
- c) *key*.
- d) *view*.



9. Los identificadores definidos en tiempo de compilación se guardan en una clase generada ¿en cuál?
 - a) En una clase de recursos llamada *Interface*.
 - b) En una clase de recursos llamada *R*.
 - c) En una clase de recursos llamada *Resources*.
 - d) En una clase de recursos llamada *Android*.
10. ¿Cuál de estas formas de acceder al contexto es incorrecta?
 - a) *getMyContext()*.
 - b) *getContext()*.
 - c) *This*.
 - d) *getApplicationContext()*.
11. ¿Qué tipo de objeto es un *layout*?
 - a) *Screen*.
 - b) *Window*.
 - c) *ViewGroup*.
 - d) *Resource*.
12. ¿Qué *layout* sería el óptimo para mostrar un objeto detrás de otro?
 - a) *LinearLayout*.
 - b) *RelativeLayout*.
 - c) *TableLayout*.
 - d) *FrameLayout*.
13. ¿Qué *layout* sería el óptimo para mostrar una cuadrícula de datos?
 - a) *LinearLayout*.
 - b) *RelativeLayout*.
 - c) *TableLayout*.
 - d) *FrameLayout*.

14. ¿Cuál es el mejor componente para mostrar una foto?

- a) *TextView*.
- b) *ImageView*.
- c) *CheckBox*.
- d) *ToggleButton*.

15. ¿Dónde definiremos la funcionalidad de un botón?

- a) En su fichero XML.
- b) En un objeto *OnClickListener* asociado.
- c) En un objeto *ButtonListener* asociado.
- d) Ninguna de las anteriores.

16. ¿Qué método usamos en nuestras *Activity* para acceder a cada uno de los objetos *View* que componen nuestro interfaz de usuario?

- a) *findViewById()*.
- b) *findId()*.
- c) *findViewById()*.
- d) *findLayout()*.

17. ¿Con qué componente podemos mostrar listados de datos?

- a) *TextView*.
- b) *ListView*.
- c) *EditText*.
- d) *View*.

18. ¿Qué objeto es el encargado de asociar una colección de datos con dicho objeto de listado?

- a) *BaseAdapter*.
- b) *ListViewAdapter*.
- c) *DataAdapter*.
- d) Ninguno de los anteriores.



- 19. ¿Qué método del “adapter” es el encargado de dibujar cada una de los ítems de información?**
- a) *getView()*.
 - b) *view()*.
 - c) *adapterView()*.
 - d) *setView()*.
- 20. ¿Qué método debemos sobrescribir para gestionar la opción de menú seleccionada?**
- a) *onClickMenu*.
 - b) *onMenuItemSelected*.
 - c) *onOptionsItemSelected*.
 - d) *onMenu*.



SOLUCIONARIO

1.	c	2.	c	3.	b	4.	b	5.	c
6.	d	7.	d	8.	a	9.	b	10.	a
11.	c	12.	a	13.	c	14.	b	15.	b
16.	c	17.	b	18.	a	19.	a	20.	c



Domina la materia y entréname con nosotros; con tu esfuerzo y el sistema personalizado de **Master.D** pronto serás un **P8.10**.

PROPUESTAS DE AMPLIACIÓN

Se recomienda leer los siguientes capítulos de la documentación oficial:

- Creación de nuestros propios componentes:
<http://developer.android.com/guide/topics/ui/custom-components.html>.
- HelloViews (ejemplos de uso de diferentes objetos View):
<http://developer.android.com/resources/tutorials/views/index.html>.
- Trucos para *layouts*: reusando:
<http://developer.android.com/resources/articles/layout-tricks-reuse.html>.
- Trucos para *layouts*: creando *layouts* eficientes:
<http://developer.android.com/resources/articles/layout-tricks-efficiency.html>.
- Trucos para *layouts*: mezclando *layouts*:
<http://developer.android.com/resources/articles/layout-tricks-merge.html>.



BIBLIOGRAFÍA

- Ejemplos de uso de diferentes Views:
<http://developer.android.com/resources/tutorials/views/index.html>.
- Más información sobre la creación de menús:
<http://developer.android.com/guide/topics/ui/menus.html>.
- Creando diálogos:
<http://developer.android.com/guide/topics/ui/dialogs.html>.
- Aplicando estilos y temas:
<http://developer.android.com/guide/topics/ui/themes.html>.
- Problemas de memoria derivados del mal uso de Context:
<http://developer.android.com/resources/articles/avoiding-memory-leaks.html>.

PROGRAMACIÓN PARA ANDROID

6

Recursos de aplicación



ÍNDICE

MOTIVACIÓN.....	3
PROPÓSITOS	4
PREPARACIÓN PARA LA UNIDAD	5
1. DEFINIENDO RECURSOS	7
2. USANDO RECURSOS.....	11
3. LOCALIZACIÓN.....	14
4. TIPOS DE RECURSOS.....	16
4.1. ANIMACIONES.....	16
4.2. LISTAS DE ESTADO DE COLORES.....	18
4.3. OBJETOS “DIBUJABLES” (<i>DRAWABLE</i>)	19
CONCLUSIONES	27
RECAPITULACIÓN	28
AUTOCOMPROBACIÓN	29
SOLUCIONARIO.....	35
PROPUESTAS DE AMPLIACIÓN	36
BIBLIOGRAFÍA.....	37



MOTIVACIÓN



Dirígete hacia tu meta sin vacilar; nosotros te ayudamos a llegar hasta allí.

El código de nuestras aplicaciones se apoya en una serie de recursos como imágenes, sonidos, etc.

En este capítulo vamos a conocer el API de Android para la gestión de recursos, veremos los tipos de recursos que podemos usar y aprenderemos a crear recursos alternativos para que, por ejemplo, nuestra aplicación sea multi-idioma.

PROPOSITOS

- Conocer la estructura de la carpeta de recursos.
- Conocer los principales tipos de recurso. Nos detendremos en los tipos que todavía no hemos visto: animaciones, colores de estado y objetos Drawabl.
- Aprender a crear recursos alternativos.
- Aprender a crear una aplicación multi-idioma.



PREPARACIÓN PARA LA UNIDAD

Para esta unidad es necesario tener el entorno de desarrollo instalado y configurado.



1. DEFINIENDO RECURSOS

En los capítulos anteriores hemos visto que Android separa los recursos (imágenes, sonidos, etc.) del código colocándolos organizados dentro del directorio `res`. Esto nos facilita su mantenimiento, además de permitirnos usar diferentes recursos dependiendo de la configuración del terminal.

Por organización, los archivos deben ir agrupados en carpetas según su tipo.

Recordemos su estructura:

Carpeta	Tipo s de archivos	Contenido
<code>res/anim/</code>	<code>.xml</code>	Definición de animaciones.
<code>res/color/</code>	<code>.xml</code>	Definición de estados de colores.
<code>res/drawable/</code>	<code>.xml .png, 9.png, .jpg, .gif</code>	Ficheros de mapas de bit (.png, 9.png, .jpg, .gif) o ficheros de definición XML que se compilan en objetos de tipo “dibujable” definidos en la biblioteca de gráficos 2D de Android.
<code>res/layout/</code>	<code>.xml</code>	Definición de diseños de pantalla.
<code>res/menu/</code>	<code>.xml</code>	Definición de menús.
<code>res/raw/</code>	<code>*</code>	Ficheros que son guardados por el sistema “tal cual” sin comprimir, y no tienen cabida en el resto de carpetas.
<code>res/values/</code>	<code>.xml</code>	Definición de valores simples.
<code>res/xml/</code>	<code>.xml</code>	Ficheros xml que pueden ser leídos llamando a <code>Resources.getXML()</code> .

Estas carpetas no pueden contener subcarpetas. Los nombres de los ficheros de recursos deben estar en minúsculas y no contener espacios. Si no se cumplen estas reglas, nuestra aplicación mostrará errores de compilación.

Recursos alternativos

Como hemos mencionado antes, Android nos permite colocar recursos alternativos para que sean usados dependiendo de la configuración del terminal. Por ejemplo: podemos crear iconos más pequeños para que sean mostrados en los terminales con pantallas reducidas o diseñar una disposición de pantalla diferente para cuando nuestra aplicación se esté ejecutando en modo apaisado. Pero ¿cómo se consigue esto?

La solución es muy sencilla, simplemente crearemos una nueva carpeta de recursos añadiendo un sufijo que indica la situación en la que debe usarse.

Por ejemplo si queremos definir unos textos específicos que deben usarse cuando el idioma sea inglés crearemos una nueva carpeta con el sufijo “-en” y dentro de ella colocaremos sólo los recursos que queremos sobrescribir para este idioma:

./res/values/strings.xml → cadenas para el idioma por defecto

./res/values-en/strings.xml → cadenas para idioma inglés

Lo mismo sucedería para las imágenes. Si queremos crear una imagen específica para las pantallas grandes deberíamos usar:

./res/drawable/fondo.png → fondo por defecto

./res/drawable-hdpi/fondo.png → fondo para pantallas grandes

./res/drawable-ldpi/fondo.png → fondo para pantallas pequeñas

La siguiente tabla muestra los sufijos que podemos utilizar:

Modificador	Valores	Descripción
mcc y mnc	Ejemplo: ./res/drawable-mmc214 ./res/drawable-mmc214-mnc03	Recursos específicos por códigos móviles de país y/o códigos de operadora. Los ejemplos contendrían recursos específicos para terminales de España (mmc = 214). El otro además filtra por la operadora (mnc = 03 indica Orange).
Lenguaje y región	Ejemplo: ./res/values-es ./res/values-fr-rFR	Idioma definido por dos letras según las reglas ISO (639-1), también puede incluir la región (precedida por la letra 'r' minúscula). En el ejemplo definiríamos recursos para el idioma español. El segundo los definiría para el idioma francés, región francesa.
Tamaño de pantalla	<i>small</i> <i>normal</i> <i>large</i> Ejemplo: ./res/layout-small	<i>Small</i> : pantallas pequeñas. <i>Normal</i> : pantallas medianas. <i>Large</i> : pantallas grandes.

Modificador	Valores	Descripción
Aspecto de pantalla	<i>long</i> <i>notlong</i> Ejemplo: <i>./res/drawable-long</i>	Este valor tiene en cuenta el ratio de aspecto de la pantalla: <i>Long</i> : para pantallas “alargadas”. <i>Notlong</i> : pantallas no alargadas.
Orientación de la pantalla	<i>port</i> <i>land</i> Ejemplo: <i>./res/layout-land</i>	Depende de si la pantalla está en modo apaisado (<i>land</i>) o normal (<i>port</i>).
Modo dock	<i>car</i> <i>desk</i> Ejemplo: <i>./res/drawable-car</i>	¿Está el terminal en modo coche (<i>car</i>) o conectado al ordenador (<i>desk</i>)?
Modo diurno	<i>night</i> <i>notnight</i> Ejemplo: <i>./res/layout-night</i>	Se basa en el reloj para determinar si es de día (<i>notnight</i>) o de noche (<i>night</i>).
Densidad de la pantalla en pixeles	<i>ldpi</i> <i>mdpi</i> <i>hdpi</i> <i>nodpi</i> Ejemplo: <i>./res/drawable-nodpi</i>	<i>ldpi</i> : pantallas de baja densidad (120 dpi). <i>mdpi</i> : pantallas de densidad media (160 dpi). <i>hdpi</i> : pantallas de densidad alta (240 dpi). <i>nodpi</i> : recursos que no deben ser escalados para ajustarse a la densidad de la pantalla.
Tipo de pantalla táctil	<i>notouch</i> <i>stylus</i> <i>finger</i> Ejemplo: <i>./res/layout-notouch</i>	<i>notouch</i> : no soporta el modo táctil. <i>stylus</i> : pantallas resistivas que son usadas con un lápiz. <i>finger</i> : pantallas táctiles en general.
Disponibilidad de teclado	<i>keysexposed</i> <i>keyssoft</i> <i>keyshidden</i> Ejemplo: <i>./res/drawable-keysoft</i>	<i>keysexposed</i> : el dispositivo tiene un teclado disponible. <i>keyshidden</i> : el dispositivo tiene un teclado de hardware, pero está escondido y no tiene habilitado el teclado por software <i>keyssoft</i> : el dispositivo tiene un teclado por software oculto o no
Método primario de entrada	<i>nokeys</i> <i>qwerty</i> <i>12key</i> Ejemplo: <i>./res/layout-qwerty</i>	<i>nokeys</i> : el dispositivo no tiene un teclado físico. <i>qwerty</i> : el dispositivo tiene un teclado <i>qwerty</i> (visible o no). <i>12key</i> : el dispositivo tiene un teclado de 12 teclas.

Modificador	Valores	Descripción
Tecla de navegación disponible	<p><i>navexposed</i> <i>navhidden</i> Ejemplo: <i>./res/layout-navhidden</i></p>	<p><i>navexposed</i>: las teclas de navegación están disponibles para el usuario.</p> <p><i>navhidden</i>: las teclas de navegación no están disponibles.</p>
Método primario de navegación (no táctil)	<p><i>nonav</i> <i>dpad</i> <i>trackball</i> <i>wheel</i> Ejemplo: <i>./res/layout-nonav</i></p>	<p><i>nonav</i>: el dispositivo no tiene otro método de navegación que no sea la pantalla táctil.</p> <p><i>dpad</i>: el dispositivo tienen un <i>pad</i> direccional.</p> <p><i>trackball</i>: el dispositivo dispone de un <i>trackball</i>.</p> <p><i>wheel</i>: el dispositivo dispone de una rueda de navegación.</p>
Versión del sistema	<p>Ejemplos: <i>v4</i> <i>v7</i> <i>./res/layout-v4</i></p>	El nivel de API soportado por el dispositivo. Muy útil para utilizar componentes nuevos y mantener la compatibilidad con los dispositivos antiguos.

Para afinar más nuestras aplicaciones podemos utilizar varios modificadores al mismo tiempo.

Por ejemplo: imagina que quieras tener tu aplicación traducida al alemán. Aparte de crear un *./res/values-de/strings.xml* con los textos traducidos, tendríamos que tener en cuenta que las palabras en este idioma son más largas y posiblemente no quepan en la maquetación original, así que crearíamos una nueva carpeta para los diseños de pantalla de terminales con pantalla pequeña e idioma alemán: *./res/layout-ldpi-de/*.

En estas carpetas alternativas no es necesario duplicar todos los recursos, simplemente pondremos aquellos que modifiquemos (el resto se cogerán de la carpeta por defecto).

Un recurso que suele ser obligado desdobljar es el icono de la aplicación¹ que deberemos adaptar en su tamaño para los diferentes tamaños de pantalla para evitar que quede mal en el *launcher*.

¹ http://developer.android.com/guide/practices/ui_guidelines/icon_design.html.



2. USANDO RECURSOS

A todos los recursos que colocamos en las subcarpetas de ./res/ se puede acceder a través de la clase R de nuestro proyecto.

Esta clase R la genera el comando aapt en una pasada anterior a la compilación (Eclipse, por defecto, la va generando continuamente conforme cambiamos los recursos). Contiene todos los identificadores de recursos para poder referenciarlos.

Al igual que la carpeta “res”, la clase R se organiza en subclases, así por ejemplo el ícono que colocamos en ./res/drawable/icon tiene su correspondencia en R.drawable.icon (que es un identificador estático de tipo *int* y sirve para acceder al recurso).

Así pues los ID de recurso están compuestos de:

Clase R que contienen todos los recursos.

Subclase de recurso, cada grupo tiene la suya (*drawable, string, style, layout...*).

Nombre del recurso que, según el tipo, será: el nombre del fichero sin la extensión² o el atributo xml “android:name” si es un valor sencillo (cadena, estilo, etc.).

Tenemos dos formas de acceder a los recursos definidos en la clase R:

En el código, accediendo a las propiedades de la clase R directamente (R.string.nombre).

En los ficheros XML, usando una notación especial: @grupo_recursos/nombre_recurso, es decir, el recurso anterior se accedería con @string/nombre.

² Por eso no se pueden utilizar subcarpetas, ni espacios ni caracteres especiales en los nombres de archivo de recursos.

Si lo que queremos es acceder a un recurso definido por el sistema anteponemos el prefijo android-:

- Desde el código: android.R.layout.simple_list_item_1.
- En los ficheros XML: @android:layout/simple_list_item_1.

Referenciando atributos de estilo

Cuando aplicamos estilos a nuestros *layout* puede interesarnos acceder a un atributo concreto de un estilo, para eso tenemos una sintaxis específica que podemos usar en nuestros XML:

```
?[<nombre_paquete>]:[<tipo_recurso>]<nombre_recurso>
```

Así por ejemplo si queremos colocar un texto pequeño usaremos:

```
?android:attr.textAppearanceSmall
```

Si queremos, también podemos utilizar nuestros propios atributos.

Primero lo definimos con un tag “attr” dentro de ./res/values/attr.xml.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <attr name="cabecera" format="reference" />
</resources>
./res/values/attr.xml
```

Ahora ya podemos usar esa propiedad en nuestros estilos:

Primero definimos un estilo de texto llamado “TituloRojo”, y luego lo aplicamos al atributo que hemos creado llamado “Cabecera”. Obsérvese que como es un atributo propio, no usamos el espacio de nombres “android:”.

Si luego quisieramos acceder a este atributo al definir un *layout* podríamos usar la sintaxis mencionada:

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent">
<TextView
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:text="No hay datos disponibles"
    style="?attr/cabecera" />
</FrameLayout>
./res/layout/milayout.xml
```



```
<?xml version="1.0" encoding="utf-8"?>
<resources>
<style name="MiTema" parent=
    "@android:style/Theme.Light">
    <item name="android:windowBackground">
        @drawable/fondo</item>
    <item name="cabecera">
        @style/TituloRojo</item>
</style>
<style name="TituloRojo"
    parent="@android:style/TextAppearance.Large">
    <item name="android:textColor">#FF0000</item>
    <item name="android:textStyle">bold</item>
</style>
</resources>
```

./res/values/style.xml

3. LOCALIZACIÓN

Android Market nos ofrece un canal de distribución a nivel mundial para nuestras aplicaciones. Si queremos acceder a un mayor número de usuarios no debemos descuidar el tema de los idiomas.

Ya hemos visto que podemos crear recursos alternativos basados en el idioma (creando la carpeta alternativa con el sufijo correspondiente), pero para ello debemos acostumbrarnos a usar dichos recursos desde el principio. De esta forma adaptar nuestra aplicación será más sencillo.

Las reglas básicas que debemos seguir son:

- Escribir todos los mensajes y textos que se van a usar en un fichero de recursos (`res/values/strings.xml`).
- Referenciar a estos recursos desde nuestro código Java mediante el uso de la clase `R.string`.
- Referenciar estos recursos en nuestros ficheros xml de *layout* a través de `@string/`.

Una vez que hayamos desarrollado la aplicación y hayamos comprobado que todo funciona correctamente, llega el momento de añadir nuevos locales:

- Crearemos la carpeta de recursos alternativos para el idioma (por ejemplo `./res/values-en/strings.xml`).
- Crearemos una entrada por cada cadena que sea necesario traducir (las que sean iguales en los dos idiomas no hace falta reescribirlas).
- Si tenemos gráficos que contengan textos, también deberemos crear su correspondiente versión en la carpeta alternativa (ej.: `./res/drawable-en/`).
- Lo mismo con el resto de recursos (por ejemplo si tenemos alguna locución en audio, la grabaremos en inglés en `./res/raw-en/`).



Una vez que tengamos todo traducido, probaremos la aplicación en el emulador, configurándolo en el idioma que queramos probar (Menú principal/ajustes/idioma y país).

Deberemos comprobar que todos los textos salen traducidos y que la maquetación no se rompe. Hay que tener en cuenta que algunos idiomas como el alemán pueden generar problemas, ya que sus palabras suelen ser más largas. Si es necesario corregir este tipo de problemas crearemos el *layout* alternativo correspondiente (en nuestro ejemplo dentro de ./res/layout-en/).

Una vez que la adaptación esté concluida, si es posible la daremos a probar a alguien nativo para que nos detecte posibles fallos.

Una buena idea es contar con la ayuda de los usuarios para este tipo de tareas. En la red existen multitud de herramientas que permiten realizar estas labores de traducción de forma colaborativa, como por ejemplo:

- <http://translations.launchpad.net>
- <http://crowdin.net/project/ankidroid>
- <http://translate.sourceforge.net/wiki/pootle/index>

4. TIPOS DE RECURSOS

Como hemos visto, nuestras aplicaciones pueden usar recursos de lo más dispar: desde diseños de pantalla a imágenes, pasando por ficheros binarios o valores...

Ya hemos visto algunos como las cadenas, menús, *layouts* o estilos, en este capítulo vamos a ver el resto:

4.1. ANIMACIONES

Además de usar código crear animaciones, Android nos permite definir dos tipos de efectos animados usando XML de recursos:

- Animaciones dinámicas: permiten realizar transiciones como rotar, fundir, mover o redimensionar.
- Animaciones fotograma a fotograma: mediante una sucesión de fotogramas conseguimos realizar el efecto animado.



Ejemplos

Veamos unos ejemplos de animaciones:

Para definir una animación que mueva una vista de la derecha de la pantalla (`fromXDelta = "100%p"`) hasta su posición dentro del *layout* (`toXDelta="0"`) y que dure un segundo (`duration ="1000"`) este sería el código:

```
<?xml version="1.0" encoding="utf-8"?>
<set
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:interpolator="@android:anim/accelerate_interpolator">
    <translate
        android:fromXDelta="100%p"
        android:toXDelta="0"
        android:duration="1000" />
</set>
```

res/anim/desplaza.xml

También podemos “jugar” con el canal alfa (transparencia). Para hacer un fundido de transparente a sólido:

```
<?xml version="1.0" encoding="utf-8"?>
<alpha
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:interpolator="@android:anim/
    accelerate_interpolator"
    android:fromAlpha="0.0"
    android:toAlpha="1.0"
    android:duration="1000" />
```

res/anim/fadein.xml

Como se puede ver, a las animaciones se les puede aplicar un objeto *Interpolator* que modifica el comportamiento de la animación (por ejemplo, la acelera, decelera, repite, etc.). Si no queremos crear los nuestros propios, Android nos provee de unos predefinidos³ válidos para la mayoría de las situaciones.

Para luego aplicar estas animaciones en nuestro código podemos usar el tag “`animation`” en nuestros xml o invocarlas con código.

Imaginemos que queremos aplicar la animación anterior (*fadein*) a un objeto *View*:

```
Imagen = (ImageView) findViewById(R.id.Imagen);
Animation animacion = AnimationUtils.loadAnimation
( this, R.anim.fadein );
Imagen.startAnimation(animacion);
```

Aplicando una animación a una vista

³ Ver <http://developer.android.com/reference/android/view/animation/Interpolator.html>.

Para definir una animación de tipo fotograma a fotograma una vez que tengamos los componentes (por ejemplo unas imágenes), usariamos un xml similar a este:

```
<?xml version="1.0" encoding="utf-8"?>
<animation-list
    xmlns:android="http://schemas.android.com/apk/res/android"
        android:oneshot="false">
    <item android:drawable="@drawable/cara1"
        android:duration="200" />
    <item android:drawable="@drawable/cara2"
        android:duration="200" />
    <item android:drawable="@drawable/cara3"
        android:duration="200" />
    <item android:drawable="@drawable/cara4"
        android:duration="200" />
    <item android:drawable="@drawable/cara5"
        android:duration="200" />
</animation-list>
```

res/anim/fotograma.xml

En este caso usamos cinco estados de una cara que irán cambiando cada 200 milisegundos en un bucle sin fin (`oneshot="false"`).

Esta animación puede ser usada como un `drawable` por ejemplo para un fondo de una vista:

```
Imagen.setBackgroundResource(resource);
AnimationDrawable animacion = (AnimationDrawable)
    Imagen.getBackground();
animacion.start();
```

Usando una animación como fondo de una vista

Las animaciones se pueden aplicar a View o GroupView. También se pueden definir con código java y se les pueden asignar *Listeners* que realicen acciones por ejemplo al terminar la animación.

4.2. LISTAS DE ESTADO DE COLORES

Las listas de estado de colores permiten definir un XML que puede ser aplicado como un color, con la diferencia de que cambian de color según el estado de la vista:

```
<?xml version="1.0" encoding="utf-8"?>
<selector
    xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:state_pressed="true"
        android:color="#ffff0000"/> <!-- presionado -->
    <item android:state_focused="true"
        android:color="#f00f"/> <!-- tiene el foco -->
    <item android:color="#000"/> <!-- por defecto -->
</selector>
```

res/color/texto_botones.xml



Los colores se definen usando cuatro propiedades canal alfa (A), canal rojo (R), canal verde (G) y canal azul (B) de cualquiera de las siguientes maneras:

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

4.3. OBJETOS “DIBUJABLES” (*DRAWABLE*)

Los objetos que colocamos en la carpeta `./res/drawable/` se convierten en objetos que pueden ser recuperados a través del Api de Android. Estos objetos pueden a su vez ser utilizados como parte de otro recurso (por ejemplo, como fondo de una vista).

Hay diferentes **tipos** de *drawables*:

Tipo	Clase creada	Descripción
Ficheros bitmap	<i>BitmapDrawable</i>	Archivos gráficos .jpg, .png o .gif.
Gráficos expandibles	<i>NinePatchDrawable</i>	Archivos .9.png capaces de expandirse sin perder su aspecto.
Listas de capas	<i>LayerDrawable</i>	XML que contiene un listado de otros objetos <i>drawable</i> que son dibujados en orden.
Lista de estados	<i>StateListDrawable</i>	XML que contiene un listado de otros objetos que son dibujados según el estado del objeto. Se utiliza por ejemplo para definir botones en todos sus estados.
Lista de niveles	<i>LevelListDrawable</i>	XML con un listado de objetos <i>drawable</i> asignados a un valor numérico.
Transiciones	<i>TransitionDrawable</i>	XML que define la transición entre otros dos objetos <i>drawable</i> .
Clips	<i>ClipDrawable</i>	XML que define otro <i>drawable</i> que es recortado en base a un valor numérico.
Escalables	<i>ScaleDrawable</i>	XML que define un <i>drawable</i> que se escala dependiendo de un valor.
Formas	<i>ShapeDrawable</i>	XML que define figuras geométricas.

Detengamos en alguno de ellos:

Bitmaps

Para los ficheros de tipo bitmap es preferible usar archivos .PNG, los JPG si no queda más remedio y se desaconseja el uso de .GIF.

Hay que tener en cuenta que los gráficos colocados en la carpeta “drawable” pueden ser optimizados automáticamente por la utilidad “aapt” incluida en el SDK que optimiza la paleta de colores para que los gráficos usen menos memoria sin perder calidad.

Si no queremos que nuestros gráficos sean modificados, deberemos colocarlos en la carpeta ./res/raw/ y leerlos luego como un stream.

Android intenta estirar o encoger el archivo si es necesario para adaptarlo a la vista donde va a ser usado. Si vemos que esto provoca pérdidas al usarlo en pantallas de diferente tamaño, deberemos crear un recurso alternativo.

.9.png

Basados en la técnica usada en el diseño web estos archivos dividen una imagen en 9 trozos (4 esquinas, 4 laterales, y 1 fondo) de esta forma la imagen puede estirarse o encogerse sin perder su aspecto.

Para crear estos archivos partiremos de un diseño gráfico y usaremos el comando draw9patch incluido en la carpeta “tolos” del SDK.

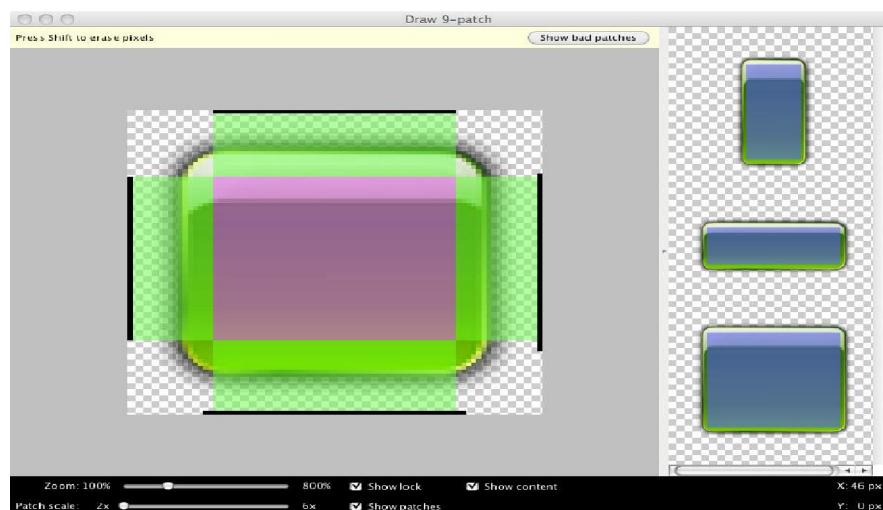


Figura 1. draw9patch en acción



Una vez cargado el gráfico original, se trata de dibujar a la izquierda y arriba unas guías que servirán para que Android “corte” el gráfico en los 9 trozos. A la derecha se nos muestra como quedará el gráfico estirado, tanto vertical, como horizontalmente.

Opcionalmente podemos definir otras dos guías (abajo y a la derecha) que servirán como margen de escritura (representado como un recuadro azul sobre la previsualización).

Como se puede apreciar, con este sistema la imagen se estira y encoge sin perder calidad en sus esquinas.

Los gráficos 9.png se generan a partir de un archivo png ya existente.

Lista de estados

Muy utilizados para hacer botones. Mediante un XML definimos los objetos *drawable* que se activarán en cada uno de los estados (la combinación de seleccionado y/o pulsado).

Los *drawables* usados en cada uno de los estados suelen ser archivos 9.png.

```
<?xml version="1.0" encoding="utf-8"?>
<selector
    xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:state_focused="true"
          android:state_pressed="false"
          android:drawable="@drawable/btn_bg_verde_on_off" />
    <item android:state_focused="true"
          android:state_pressed="true"
          android:drawable="@drawable/btn_bg_verde_on_on" />
    <item android:state_focused="false"
          android:state_pressed="true"
          android:drawable="@drawable/btn_bg_verde_off_on" />
    <item android:drawable="@drawable/btn_bg_verde_off_off" />
</selector>
```

res/drawable/boton_verde.xml

Este *drawable* puede ser referenciado como otro recurso:

```
<Button android:id="@+id	btn"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:background="@drawable/boton_verde" />
```

Usando nuestro drawable de estado en un botón

Drawable de niveles

Son ficheros XML en los que definimos una colección de objetos *drawable* asociados con un valor numérico entero.

El sistema mostrará el que se corresponda con el nivel actual definido por el método *Drawable.getLevel()*.

Para cambiar el nivel podemos usar el método *Drawable.setLevel()*.

Veamos un **ejemplo** en el que una vista de tipo *SeekBar* cambia el aspecto de otra vista cambiando el nivel de su fondo:

Primero definimos el objeto *LevelListDrawable* que a su vez hace referencia a otros *Drawable*.

```
<?xml version="1.0" encoding="utf-8"?>
<level-list
    xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:drawable="@drawable/panel_gris"
          android:maxLevel="0" />
    <item android:drawable="@drawable/panel_rojo"
          android:maxLevel="1" />
    <item android:drawable="@drawable/panel_amarillo"
          android:maxLevel="2" />
    <item android:drawable="@drawable/panel_verde"
          android:maxLevel="3" />
</level-list>
```

res/drawable/panel.xml

Luego definimos nuestros *layout* que hace uso de este objeto *Drawable*.

En nuestra activity modificamos el método *onCreate()* para que cuando la barra modifique su valor, cambie el fondo del panel. Para ello creamos un *listener* que traslada el valor de progreso al nivel del *Drawable* usado de fondo.

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
        android:id="@+id/mainlayout"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent">
<LinearLayout android:id="@+id/panel"
    android:background="@drawable/panel"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_margin="20dip"
    android:layout_alignParentBottom="true">
<TextView android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Este texto tiene como fondo un objeto
    Drawable definido con un XML y cambia de color respecto a
    un valor numérico"
    android:textColor="#000"
```



```
        android:shadowColor="#FFF"
        android:shadowDx="1"
        android:shadowDy="1"
        android:shadowRadius=".5" />
    </LinearLayout>
    <TextView android:id="@+id/nivel"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_above="@id/panel"
        android:layout_alignParentRight="true"
        android:text="0"
        android:padding="5dip" />
    <SeekBar android:id="@+id/barra"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_above="@id/panel"
        android:layout_toLeftOf="@+id/nivel"
        android:layout_alignParentLeft="true"
        android:max="3"
        android:progress="1" />
</RelativeLayout>
```

res/layout/resources.xml


```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:
        android="http://schemas.android.com/apk/res/android">
<gradient android:startColor="#3000"
        android:centerColor="#f000"
        android:endColor="#ff4e2d5c"
        android:centerY="0.9"
        android:angle="270" />
<corners android:radius="4dp" />
<padding android:left="7dp"
        android:top="7dp"
        android:right="7dp"
        android:bottom="7dp" />
</shape>
```

res/drawable/panel_gris.xml

Los *Drawable* tipo Clip y Expandable se utilizan de forma similar a estos objetos, con la diferencia de que ellos estiran o recortan el *Drawable* en base al nivel.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.resources);
// Configura el cambio de color del panel en base a la barra
final TextView nivel = (TextView) findViewById(R.id.nivel);
final View panel = findViewById(R.id.panel);
SeekBar barra = (SeekBar) findViewById(R.id.barra);
barra.setOnSeekBarChangeListener
(new OnSeekBarChangeListener() {
@Override
public void onProgressChanged
(SeekBar arg0, int nLevel, boolean arg2) {
```

```

        nivel.setText(" " +newLevel);
        Drawable fondo = panel.getBackground();
        fondo.setLevel(newLevel);
    }
    @Override
        public void onStartTrackingTouch(SeekBar arg0) {}
    @Override
        public void onStopTrackingTouch(SeekBar arg0) {}
});
}

```

Método onCreate de ResourcesActivity

Formas

Finalmente vamos a definir un fondo para un panel usando un xml drawable.

```

<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:
    android="http://schemas.android.com/apk/res/android">
<gradient android:startColor="#3000"
    android:centerColor="#f000"
    android:endColor="#ff4e2d5c"
    android:centerY="0.9"
    android:angle="270" />
<corners android:radius="4dp" />
<padding android:left="7dp"
    android:top="7dp"
    android:right="7dp"
    android:bottom="7dp" />
</shape>

```

res/drawable/panel_gris.xml

Este fondo tendrá los bordes redondeados, un degradado semitransparente de fondo y un margen de escritura.

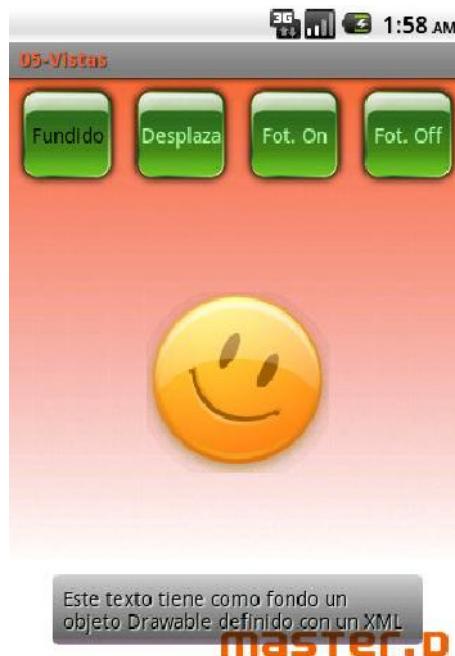


Figura 2. Ejemplo de panel definido con XML

Estos archivos de formas son muy útiles ya que se consiguen resultados espectaculares y están muy optimizados en cuanto a uso de memoria.

Siempre que sea posible usaremos este tipo de objeto *Drawable* en vez de *Bitmap*. De esta forma conseguiremos que nuestras aplicaciones ocupen menos tamaño y necesiten menos recursos de memoria.

Una buena idea es combinar este tipo de *Drawable* con otros. Por ejemplo, podemos crear un fondo basado en un *LayerDrawable* (capas) que contenga un *ShapeDrawable* con un degradado y un *BitmapDrawable* con un escudo semi-transparente.



CONCLUSIONES



Distínguete del resto. Para **Master.D** eres único; personalizamos tu camino hacia el éxito.

Como se ha podido apreciar, el API de Android para gestión de recursos es muy potente y nos ofrece un gran abanico de posibilidades que podemos usar en nuestro código.

Es una buena idea dedicar un poco de tiempo a probar cada una de ellas para conocerlas a fondo y saber usar en cada momento la más adecuada.

También es aconsejable acostumbrarse a separar los textos de idioma en un fichero de recursos para que sea sencillo su traducción.

RECAPITULACIÓN

- Los recursos se definen dentro de la carpeta ./res/.
- Deben estar organizados en subcarpetas que vienen definidas por el API de Android.
- Se pueden formar recursos en base a otros recursos, por ejemplo un fondo que hemos definido en la carpeta *drawable* puede ser usado para crear otro *drawable*.
- Se pueden crear recursos alternativos creando carpetas con sufijos. El sistema elegirá automáticamente el recurso que más se adecue al entorno de ejecución en base a la configuración del terminal (idioma, tamaño de pantalla, etc.).
- Si no encuentra un recurso específico cargará el “por defecto”.
- Podemos definir recursos “inteligentes” que reaccionan ante situaciones, por ejemplo, colores que se aplican automáticamente según el estado de la vista. Esto nos ahorra mucho trabajo de programación.
- Todos los recursos se pueden acceder desde nuestro código gracias a la clase generada R .tiporecurso.nombre.
- Desde un recurso definido con XML podemos acceder a otro referenciándolo con @tiporecurso/nombre.



AUTOCOMPROBACIÓN

- 1. Los recursos se guardan en la carpeta “res”, pero ¿dónde se coloca esta carpeta dentro de nuestro proyecto?**
 - a) Donde queramos.
 - b) Donde queramos, pero por convenio se usa ./res/.
 - c) En /res.
 - d) En el paquete, junto a las clases que las van a usar.

- 2. ¿En qué subcarpeta se guardan los diseños de vista?**
 - a) res/layout.
 - b) res/view.
 - c) res/drawable.
 - d) res/xml.

- 3. ¿Qué tipo de datos podemos guardar en la carpeta res/anim?**
 - a) .xml.
 - b) .anim.
 - c) .res.
 - d) .java.

4. **¿Qué tipo de archivo no puede ir en res/drawable?**
 - a) .png.
 - b) .9.png.
 - c) .jpg.
 - d) .tiff.
5. **Si quiero traducir mi aplicación a inglés, ¿en qué carpeta colocaré sus textos?**
 - a) res/values/strings-en.xml.
 - b) res/values-en/strings.xml.
 - c) res-en/values/strings.xml.
 - d) Da igual siempre que lo defina en el manifiesto.
6. **Si quiero hacer un diseño especial para pantallas de alta resolución ¿dónde colocaré sus imágenes?**
 - a) res/drawable/.
 - b) res/drawable-hdpi/.
 - c) res/drawable-mdpi/
 - d) res/drawable-ldpi/.
7. **¿Dónde colocaré las imágenes para pantallas de baja resolución?**
 - a) res/drawable/.
 - b) res/drawable-hdpi/.
 - c) res/drawable-mdpi/.
 - d) res/drawable-ldpi/.
8. **¿Dónde colocaré los recursos *drawable* que se van a utilizar en todos los tamaños?**
 - a) res/drawable/.
 - b) res/drawable-hdpi/.
 - c) res/drawable-mdpi/.
 - d) res/drawable-ldpi/.



9. ¿Qué sufijo usaremos en nuestra carpeta de recursos si queremos que sea usado solo cuando el dispositivo tiene rueda de navegación?
- a) -nonav.
 - b) -dpad.
 - c) -trackball.
 - d) -wheel.
10. Al archivo “res/drawable/ejemplo.png” se podrá acceder desde nuestro código java con la clase R ¿Cuál será su ruta?
- a) R.drawable.ejemplo.
 - b) R.drawable.ejemplo_png.
 - c) R.resources.drawable.ejemplo.
 - d) R.getDrawable(“ejemplo.png”).
11. ¿Qué tag xml usaremos para definir una animación fotograma a fotograma?
- a) <animation />.
 - b) <animation-list />.
 - c) <animation-frame />.
 - d) <animation-picture />.
12. ¿Cuál de las siguientes formas de definir colores es incorrecta?
- a) #FFF.
 - b) #FFFF.
 - c) #FFFFFF.
 - d) #FFFFFFF.
13. ¿Un fichero “.9.png” en qué clase se convierte al cargarse?
- a) BitmapDrawable.
 - b) NinePatchDrawable.
 - c) PngDrawable.
 - d) ShapeDrawable.

14. ¿Cómo referenciaremos desde un *layout xml* una cadena llamada **título**?

- a) @string/titulo.
- b) @res/string_titulo.
- c) ?string/titulo.
- d) ?res/string_titulo.

15. ¿Cómo referenciamos el estilo definido por Android llamado **textAppearance Small**?

- a) ?attr.textAppearanceSmall.
- b) ?android:attr.textAppearanceSmall.
- c) @attr.textAppearanceSmall.
- d) @android:attr.textAppearanceSmall

16. ¿Que *tag* usaremos para definir un estilo?

- a) <style>.
- b) <resources>.
- c) <theme>.
- d) <css>.

17. ¿Qué *tag* usaremos para definir colores que cambien según el estado?

- a) <selector>.
- b) <color>.
- c) <color-list>.
- d) <color-state>.

18. ¿Cómo puedo recuperar un archivo .xml guardado en la carpeta “res/xml/” llamado “ejemplo.xml”?

- a) R.xml.ejemplo.
- b) R.xml.ejemplo_xml.
- c) Resources.getXML().
- d) No se puede recuperar directamente.



19. ¿Cuál de los siguientes nombres de carpeta de recursos es incorrecto?

- a) res/drawable-es/.
- b) res/drawable-es-small/.
- c) res/drawable-layout/.
- d) res/drawable-car/.

20. ¿Qué comando usaremos para convertir un png a .9.png?

- a) draw9patch.
- b) draw9.
- c) png9patch.
- d) png9.



SOLUCIONARIO

1.	c	2.	a	3.	a	4.	d	5.	b
6.	b	7.	d	8.	a	9.	d	10.	a
11.	b	12.	c	13.	b	14.	a	15.	b
16.	a	17.	a	18.	c	19.	c	20.	a



Decídete a ser el mejor, sé un P8.10.

PROPUESTAS DE AMPLIACIÓN

Se recomienda leer con detenimiento el capítulo “Resources Types” de manual de Android y probar cada uno de los recursos para luego saber cuál usar en cada situación.

- <http://developer.android.com/guide/topics/resources/available-resources.html>.

También es recomendable leer la guía de estilo de iconos:

- http://developer.android.com/guide/practices/ui_guidelines/icon_design.html.



BIBLIOGRAFÍA

- Recursos para compatibilidad:
 - <http://developer.android.com/guide/topics/resources/providing-resources.html#Compatibility>.
- Más sobre animaciones
 - <http://developer.android.com/guide/topics/resources/animation-resource.html>.
 - <http://developer.android.com/reference/android/view/animation/Animation.AnimationListener.html>.

PROGRAMACIÓN PARA ANDROID

7

Cargando/Recuperando datos



ÍNDICE

MOTIVACIÓN.....	3
PROPÓSITOS	4
PREPARACIÓN PARA LA UNIDAD	5
1. ALMACENAMIENTO DE DATOS EN ANDROID	7
2. PREFERENCIAS COMPARTIDAS	8
3. ALMACENAMIENTO EN LA MEMORIA INTERNA	13
4. ALMACENAMIENTO EN LA MEMORIA EXTERNA	17
5. BASES DE DATOS	19
6. DATOS EN RED.....	27
7. DATOS DE SESIÓN	28
CONCLUSIONES	31
RECAPITULACIÓN	32
AUTOCOMPROBACIÓN	33
SOLUCIONARIO	39
PROPUESTAS DE AMPLIACIÓN	40
BIBLIOGRAFÍA.....	41



MOTIVACIÓN



¿Decidido a mejorar? **Master.D** es la herramienta que necesitas para cambiar.

Una de las tareas típicas que van a tener que realizar la mayoría de nuestras aplicaciones es guardar y recuperar datos.

Si tenemos en cuenta que la mayoría de dispositivos de Android, además de la memoria interna, suelen tener dispositivos de memoria externa como tarjetas SD, que las aplicaciones se ejecutan en su propio entorno en el que no se comparten los datos, que las actividades funcionan como miniaplicaciones y pueden necesitar compartir datos con otra... (en definitiva, multitud de situaciones), necesitaremos contar con herramientas que nos faciliten el trabajo.

En este capítulo conoceremos las que nos ofrece el SDK.

PROPOSITOS

En esta unidad aprenderemos a:

- Usar la clase *SharedPreferences*.
- Almacenar y recuperar datos de la memoria interna del terminal.
- Almacenar y recuperar datos de la memoria externa.
- Acceder a bases de datos SQLite.
- Crear y usar proveedores de contenidos.
- Usar la clase *Application* para almacenar datos de sesión.



PREPARACIÓN PARA LA UNIDAD

Como siempre, necesitaremos tener el entorno de trabajo configurado para trabajar con Android.



1. ALMACENAMIENTO DE DATOS EN ANDROID

La mayoría de los programas informáticos necesitan manejar datos persistentes. Para ello solemos utilizar bases de datos, ficheros, etc.

El SDK de Android nos ofrece una serie de herramientas dedicadas a facilitar estas tareas que nos permiten:

- El intercambio de preferencias.
- Almacenar/recuperar datos en la memoria interna del terminal.
- Almacenar/recuperar datos en la memoria externa (tarjetas SD, etc.).
- Acceso a bases de datos SQLite.
- Compartir datos usando la red
- Compartir datos de sesión entre las diferentes Activity de una aplicación.

2. PREFERENCIAS COMPARTIDAS

La clase *SharedPreferences* nos permite guardar datos privados de tipo primitivo (*boolean*, *float*, *int*, *long* y *string*) asociados a una clave.

Los datos que guardemos con este sistema serán persistentes, es decir, quedarán almacenados a pesar de que nuestra aplicación termine.

La clase Activity incorpora dos métodos que nos permiten acceder a las preferencias:

- *getPreferences()*:→si solo vamos a usar un grupo de preferencias.
- *getSharedPreferences*: → si vamos a usar varios grupos (deberemos pasar como parámetro el grupo a usar).

Por ejemplo, imaginemos que queremos mantener un contador que nos indique las veces que se ha lanzado una Activity.

```
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    SharedPreferences settings = getPreferences(0);
    int veces = settings.getInt("numVeces", 0);
    Toast.makeText(this, "Cargado " + veces + " veces",
        Toast.LENGTH_LONG)
    show();
}
```

Accediendo a las preferencias compartidas

Para mostrarlo al arrancar, usaríamos el siguiente código en nuestra Activity:

Vemos que una vez cargadas las preferencias accedemos a ellas a través de sus métodos *get*. Debemos indicar la clave a recuperar y el valor por defecto a usar en caso de que dicha clave no exista.



Para guardar este contador cada vez que la Activity termine deberemos usar un editor de *SharedPreferences*:

```
@Override  
protected void onStop() {  
    // Necesitamos un editor de preferencias para editarlas  
    SharedPreferences settings = getPreferences(0);  
    SharedPreferences.Editor editor = settings.edit();  
    // Cambiamos el valor  
    int nActual = settings.getInt("numVeces", 0);  
    editor.putInt("numVeces", nActual + 1);  
    // Validamos los cambios  
    editor.commit();  
    // Pasamos el control al padre para que termine la Actividad  
    super.onStop();  
}
```

Guardando una preferencia compartida

El uso más típico, aunque no el único, para esta forma de acceso a datos son las preferencias de usuario. De hecho es tan típico que Android incluye un tipo de Activity dedicada a la edición de preferencias *PreferenceActivity*.

PreferenceActivity

Es una Activity especializada que, partiendo de un fichero de definición XML, muestra una pantalla en la que el usuario puede elegir sus preferencias. Cuando esta actividad termina las preferencias son guardadas mediante *SharedPreferences* por lo que el resto de actividades de la aplicación pueden recuperarlas como hemos visto antes.

Para crear una pantalla de edición de preferencias solo tenemos que extender la clase *PreferenceActivity* y en su método *onCreate()* asignar el fichero XML de configuración de preferencias:

```
public class PreferenciasActivity extends PreferenceActivity {  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        addPreferencesFromResource(R.xml.preferences);  
    }  
}
```

Preferencias Activity.java

La definición de preferencias está referenciada por R.xml.preferences, así que estará dentro de la carpeta de recursos: ./res/xml/preferences.xml.

```
<?xml version="1.0"  
    encoding="utf-8"?>  
<PreferenceScreen xmlns:  
    android="http://schemas.android.com/apk/res/android">  
    <PreferenceCategory  
        android:title="@string/pref_skyn_title">
```

```

<ListPreference
    android:key="pref_color"
    android:title="@string/pref_color_title"
    android:summary="@string/pref_color_summary"
    android:entries="@array/pref_color_entries"
    android:entryValues="@array/pref_color_entryvalues"
    android:dialogTitle="@string/pref_color_dialogtitle"
/>
</PreferenceCategory>
</PreferenceScreen>

```

res/xml/preferences.xml

En este caso, hemos definido una pantalla de preferencias que solo pedirá un dato (color del título) y lo asociará a la clave “pref_color”.

Para obtener el valor lanzará una ventana de diálogo en la que se mostrará la lista de colores disponibles (*entries*). Cuando el usuario seleccione uno, se guardará su correspondiente *entryValue*.

Tanto *entries*, como *entryValues* son un array de *strings* que hemos guardado en ./res/values/:

```

<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string-array name="pref_color_entryvalues">
        <item>#FF6666</item>
        <item>#6666FF</item>
        <item>#66FF66</item>
        <item>#000000</item>
    </string-array>
    <string-array name="pref_color_entries">
        <item>Rojo</item>
        <item>Azul</item>
        <item>Verde</item>
        <item>Negro</item>
    </string-array>
</resources>

```

res/values/pref_arrays.xml

Como vemos, con muy pocas líneas de código hemos programado una pantalla totalmente funcional de configuración de preferencias de usuario.

Tiene además una ventaja añadida: dicha pantalla seguirá la estética y las normas de funcionamiento del resto de aplicaciones Android.

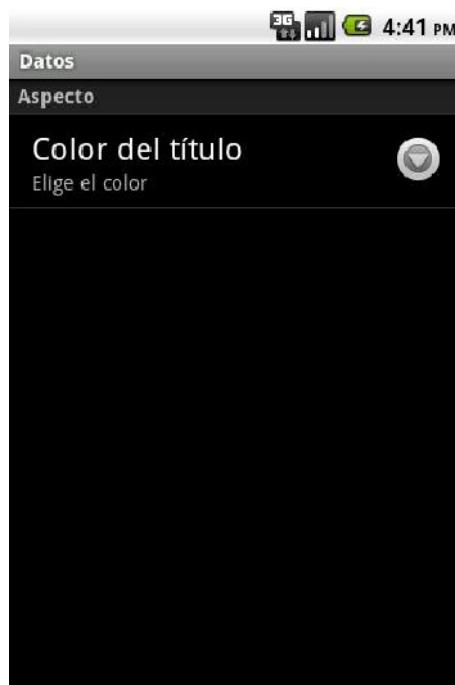


Figura 1. Pantalla de preferencias



Figura 2. Eligiendo un color

Para luego utilizar estas preferencias, por ejemplo para poner el color del título de nuestra actividad, recuperaremos el objeto *SharedPreferences*.

En este caso hemos sobreescrito el método *OnResume* para volver a la actividad, y, si ha cambiado el color, se apliquen los cambios.

```
@Override
protected void onResume() {
    super.onResume();
    try {
        // Aplicamos las preferencias del usuario al título
        PreferenceManager.setDefaultSharedPreferences(
            this, R.xml.preferences, false);
        SharedPreferences p = PreferenceManager
            .getDefaultSharedPreferences(this);
        String colorUsuario = p.getString("pref_color", "#000000");
        // Lo aplicamos al título
        TextView titulo = (TextView) findViewById(R.id.titulo);
        int color = Color.parseColor(colorUsuario);
        titulo.setTextColor(color);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Aplicando las preferencias de usuario

Android nos ofrece una gran cantidad de componentes (listas desplegables, campos de texto, *checkbox*, *radiobuttons*, etc.).

Un ejemplo de pantalla de preferencias más compleja se puede ver en los códigos de ejemplo que incluye el SDK:

- <http://developer.android.com/resources/samples/ApiDemos/res/xml/preferences.html>.



3. ALMACENAMIENTO EN LA MEMORIA INTERNA

Hay que tener en cuenta que, por defecto, estos datos serán privados y el resto de aplicaciones no podrán acceder a ellos (aunque podemos modificar este comportamiento).

Cuando se desinstala la aplicación su directorio de datos se borra junto con todo su contenido así que todos los datos que hayamos guardado en la memoria interna serán borrados.

Existen dos métodos específicos que nos permiten acceder a estos datos:

- `openFileOutput()` →devuelve un *OutputStream* sobre el que podemos escribir con su método `write()`.
- `openFileInput()` →devuelve un *InputStream* que podemos usar para leer el archivo a través de `read()`.

En ambos casos podemos utilizar las técnicas de Java para estos casos (*buffers, readers, writers, etc.*).

Como siempre en estos casos, hay que acordarse de cerrar el archivo al final de las operaciones.

Veamos un ejemplo práctico: imaginemos que queremos crear una Activity para editar un texto. Al entrar a ella cargaremos el texto previamente guardado y, cuando salgamos, guardaremos de nuevo.

En el *layout* vamos a usar un *EditText* que será el encargado de mostrar/recoger el texto:

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

```

        android:layout_height="fill_parent">
<EditText android:hint="Escribe un texto..." 
        android:id="@+id/EditText"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:padding="5dip"
        android:gravity="top|left"/>
</LinearLayout>

```

res/layout/editor.xml

En nuestra activity sobrescribiremos el método `onCreate()` para que lea el archivo guardado previamente y lo cargue en el `EditText`:

```

public class MeminternaActivity extends Activity {
    private String fichero = "misdatos";
    private EditText editor;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.editor);
        editor = (EditText) findViewById(R.id.EditText);
        String texto = loadData();
        editor.setText(texto);
    }
    protected String loadData() {
        FileInputStream fis = null;
        String texto = "";
        try {
            fis = openFileInput(fichero);
            InputStreamReader reader = new InputStreamReader(fis);
            BufferedReader buffreader = new BufferedReader(reader);
            String linea = "";
            while ((linea = buffreader.readLine()) != null) {
                texto += linea + "\n";
            }
        } catch (Exception e) {
            Toast.makeText(this, "Error al leer el fichero",
            Toast.LENGTH_LONG)
            .show();
        } finally {
            try {
                fis.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        return texto;
    }
}

```

Leyendo datos en nuestra Activity



Cuando salgamos de la Activity haremos el paso contrario: guardaremos en el fichero los datos del TextEdit.

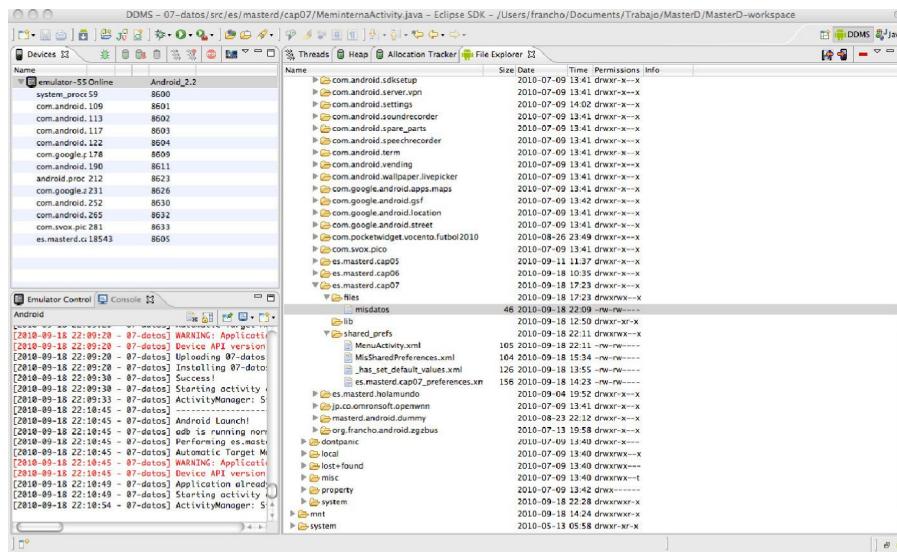


Figura 3. Accediendo al sistema de ficheros del dispositivo con Eclipse

```
/*
 * onStop()
 */
@Override
protected void onStop() {
    String texto = editor.getText().toString();
    saveData(texto);
    super.onStop();
}
/*
 * Guarda los datos en la memoria interna
 */
protected void saveData(String texto) {
    FileOutputStream fos = null;
    try {
        fos = openFileOutput(fichero, Context.MODE_PRIVATE);
        fos.write(texto.getBytes());
    } catch (Exception e) {
        Toast.makeText(this, "Error al guardar el fichero",
        Toast.LENGTH_LONG).show();
    } finally {
        try {
            fos.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Guardando los datos en un fichero

Hay que tener en cuenta que, aunque el resto de programas no tengan acceso a estos ficheros (a no ser que al abrirlos se pase un parámetro menos restrictivo), el usuario (y nosotros) puede tener acceso a los mismos usando por ejemplo herramientas de *debug* (como Eclipse):

Ficheros de caché

Si nuestra aplicación consume muchos recursos de red puede ser una buena idea implementar un sistema de caché para no consumir conexión de datos de manera innecesaria.

Para los casos en los que queremos guardar un caché en la memoria interna en vez de almacenar un dato de forma persistente, usaremos el método `getCacheDir()` para obtener el directorio de caché asignado a nuestra aplicación, y luego mediante el uso de la clase `File` grabar o recuperar el dato.

Hay que tener en cuenta que los ficheros de caché pueden ser borrados en cualquier momento por el usuario o el sistema si necesita más espacio. Aunque no tenemos que fiarnos de esto y deberemos hacer nosotros el control del espacio que ocupa nuestra caché, ya que un terminal con la memoria interna llena se ralentiza muchísimo.



4. ALMACENAMIENTO EN LA MEMORIA EXTERNA

La memoria interna de los terminales suele ser limitada, por eso en algunos casos puede interesar almacenar la información en la memoria externa del teléfono (normalmente tarjetas de memoria).

Aunque su uso es sencillo, antes de intentar guardar o leer los datos de una memoria externa deberemos realizar unas comprobaciones rutinarias para comprobar que:

- La memoria está insertada.
- Tenemos permiso de escritura en la tarjeta (no está protegida contra escritura).

También hay que tener en cuenta que en cualquier momento el usuario puede decidir sacar la tarjeta por lo que nuestro código deberá estar preparado para tales situaciones.

Para comprobar el estado de la memoria externa usamos la clase *Environment*. Por ejemplo, para comprobar si la tarjeta está montada y tenemos permiso de escritura podríamos usar un código similar a este al principio de nuestro método *guardarDatos()*.

```
// Comprobamos el estado de la memoria externa
String state = Environment.getExternalStorageState();
if (! state.equals(Environment.MEDIA_MOUNTED)) {
    Toast.makeText(this, "Tarjeta SD no montada o
    protegida contra escritura", Toast.LENGTH_LONG).show();
    return;
}
```

Comprobando podemos grabar en la memoria externa

Una vez comprobado que podemos acceder, podemos usar métodos similares a usando `getExternalFileDir()`¹ que nos devuelve un objeto “File” con el cual podemos acceder al sistema de ficheros. Este método recibe un parámetro con el tipo de directorio al que queremos acceder.

Por ejemplo, si quisiéramos guardar algo en el directorio de descargas de la tarjeta de memoria usaríamos: `Environment.DIRECTORY_DOWNLOADS`.

El usar estos tipos predefinidos ayuda a que nuestros archivos sean indexados correctamente, además el usarlos nos asegura que cuando nuestra aplicación se borre, dichos archivos sean eliminados también.

Si pasamos `null` como parámetro podremos acceder a la raíz de la tarjeta y deberemos ser nosotros quienes indiquemos la ruta completa.

Si queremos que nuestros datos no se borren al desinstalar la aplicación podemos guardarlos en cualquiera de las carpetas públicas de la tarjeta, normalmente suelen estar en el raíz de la carpeta y tener nombres genéricos como `./Pictures`, `./Music`, etc.

Caché en la memoria externa

También podemos guardar nuestros ficheros de caché en la memoria externa. Para obtener el directorio de caché a usar llamaremos a `getExternalCacheDir()`².

Guardando los ficheros de caché en la memoria externa podemos conseguir liberar espacio de la memoria del terminal, lo que repercute en un mejor funcionamiento del mismo.

¹ Esta función apareció en el API 8, las versiones anteriores usan `getExternalStorageDirectory()`. Consulta el manual para los detalles.

² En versiones anteriores a API 8 se usa `getExternalStorageDirectory()` y se guardan los datos en `/Android/data/nombre_paquete/caché/`.



5. BASES DE DATOS

SQLite es un motor de bases de datos relacionales ligero muy potente. Almacena toda la información del SGBD en un único fichero y no necesita tener ningún demonio o servicio corriendo para poder ser usado.

Android ofrece soporte total para este motor de base de datos a través del paquete android.database.sqlite.

Aunque no es estrictamente necesario, se recomienda utilizar clases hijas de SQLiteOpenHelper sobrescribiendo su método `onCreate()`. De esta forma, la primera vez que se intenta acceder a la base de datos, y esta no existe, podemos crear su estructura y cargar los datos iniciales necesarios. También podemos sobrescribir el método `onUpdate()` si es necesario actualizar la estructura o datos en un actualización de la aplicación:

```
public class JuegoSQLHelper extends SQLiteOpenHelper {
    public JuegoSQLHelper(Context context) {
        super(context, JuegoDB.DB_NAME, null,
              JuegoDB.DB_VERSION );
    }
    @Override
    public void onCreate(SQLiteDatabase db) {
        if(db.isReadOnly()) { db=getWritableDatabase(); }
        db.execSQL("CREATE TABLE jugadores
                   (_ID INTEGER PRIMARY KEY AUTOINCREMENT, nombre TEXT,
                    puntos INTEGER);");
    }
    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion,
                         int newVersion) {
        // Cuando haya cambios en la estructura deberemos
        // incluir el código
        // SQL necesario para actualizar la base de datos
    }
}
```

JuegoSQLHelper.java

Usar esta clase es muy sencillo. Por ejemplo, para insertar un registro, deberíamos crear un objeto “ContentValues” en el que asociamos los valores con los nombres de los campos, y luego llamaríamos al método *insert* pasando el nombre de la tabla y estos valores.

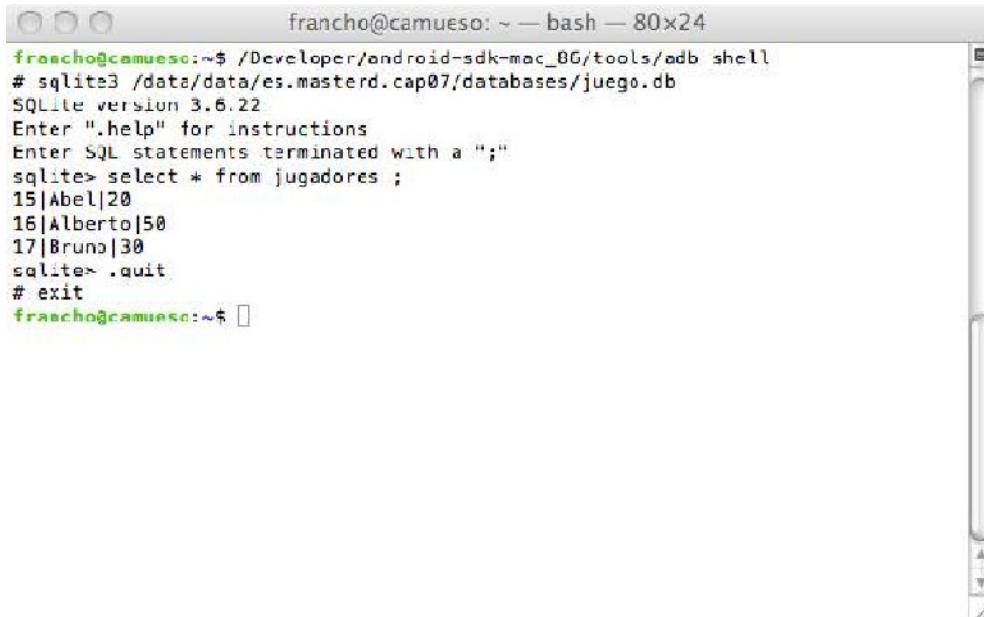
```
JuegoSQLHelper dbHelper = new JuegoSQLHelper(context);
SQLiteDatabase jugadoresDB =
dbHelper.getWritableDatabase();
ContentValues values = new ContentValues();
values.put("nombre", "Pepe");
values.put("puntos", "100");
jugadoresDB.insert("jugadores", "", values);
Insertando un registro en nuestra base de datos
```

Como hemos mencionado, nuestra base de datos SQLite se almacena en un archivo dentro del directorio de datos de nuestra aplicación en la ruta:

`/data/data/<nombre_aplicacion>/databases/<nombre_bd>`

Si necesitamos acceder a ella podemos usar el comando “adb shell” (incluido en la carpeta tools del SDK) para conectarnos al terminal o al emulador.

Una vez dentro, usaremos el comando “sqlite3 <path_db>” para conectarnos al SGBD.



The screenshot shows a terminal window with the following session:

```
francho@camueso: ~ — bash — 80x24
francho@camueso:~$ /Developer/android-sdk-mac_0G/tools/adb shell
# sqlite3 /data/data/es.masterd.cap07/databases/juego.db
SQLite version 3.8.22
Enter ".help" for instructions
Enter SQL statements terminated with a ";" 
sqlite> select * from jugadores ;
15|Abel|20
16|Alberto|50
17|Bruno|30
sqlite> .quit
# exit
francho@camueso:~$
```

Figura 4. Ejemplo de debug de una base de datos usando adb.



Proveedores de contenidos

Las bases de datos programadas con el método anterior, solo son visibles para nuestra aplicación. La única forma de compartir sus datos con el resto es usar un proveedor de contenido (ContentProvider).

Los objetos “ContentProvider” almacenan y recuperan datos usando una interfaz unificada. De esta forma es muy sencillo enlazarlos a través de un objeto “ContentResolver”.

Ante una petición de información, el objeto “ContentResolver” es capaz de averiguar cuál es el objeto “ContentProvider” que proporciona esos datos, consultarlo y devolver una respuesta estándar.

Los datos son devueltos como si fueran una tabla sencilla de base de datos en la que hay registros que contienen campos. Cada registro contiene un campo obligatorio llamado *_ID* que se usa como identificador único del mismo.

En el caso de las bases de datos, este campo *_ID* suele estar también incluido en el diseño de nuestras tablas como un campo numérico autoincremental.

La respuesta devuelve un objeto tipo Cursor que nos permite recorrer todos los resultados de forma sencilla.

Cada proveedor de contenido expone una URI pública (objeto Uri) similar a una url web, en la que se usa el prefijo “content://” para indicar que se trata de un acceso a un “ContentProvider”.

Android nos ofrece una serie de proveedores de contenido³, ya definidos, que podemos usar, por ejemplo, para acceder a los contactos de la agenda, las imágenes de la galería, los mensajes SMS recibidos, etc.

El programar un proveedor de contenido puede parecer una tarea complicada, pero merece la pena perder un poco de tiempo con ello, ya que de esta forma conseguiremos que nuestro código sea más modular y fácil de reutilizar. Además si lo queremos, podremos compartir los datos de nuestra aplicación para que el resto del ecosistema de nuestro terminal pueda usarlas.

Imaginemos que queremos hacer un juego que mantenga una tabla de mejores puntuaciones. La información se guardaría en una única tabla y tendría tres campos (*_ID*, nombre y puntos).

Ya hemos visto antes el Helper que nos ayudaría a acceder a esta base de datos. Veamos ahora los pasos necesarios para crear nuestro propio proveedor de contenido.

Lo primero de todo es crear una clase que extienda a “ContentProvider”.

³Se puede consultar el listado completo de proveedores de contenido del sistema en <http://developer.android.com/reference/android/provider/package-summary.html>.

```

public class JuegoProvider extends ContentProvider {
    @Override
        public int delete(Uri uri, String selection,
                         String[] selectionArgs) {
    // TODO Auto-generated method stub
        return 0;
    }
    @Override
        public String getType(Uri uri) {
    // TODO Auto-generated method stub
        return null;
    }
    @Override
        public Uri insert(Uri uri, ContentValues values) {
    // TODO Auto-generated method stub
        return null;
    }
    @Override
        public boolean onCreate() {
    // TODO Auto-generated method stub
        return false;
    }
    @Override
        public Cursor query(Uri uri, String[] projection,
                           String selection, String[] selectionArgs,
                           String sortOrder) {
    // TODO Auto-generated method stub
        return null;
    }
    @Override
        public int update(Uri uri, ContentValues values,
                         String selection, String[] selectionArgs) {
    // TODO Auto-generated method stub
        return 0;
    }
}

```

extendiendo la clase ContentProvider

Como vemos al extender la clase ContentProvider nos obligan a implementar los métodos típicos de cualquier aplicación de datos (altas, bajas, consultas y modificaciones), pero antes de dotarlos de funcionalidad definiremos una serie de campos que usaremos en toda la clase:

```

public class JuegoProvider extends ContentProvider {
    public static final String PROVIDER_NAME =
        "es.masterd.juego";
    public static final Uri CONTENT_URI =
        Uri.parse("content://"
        + PROVIDER_NAME + "/jugadores");
    public static final String _ID = "_id";
    public static final String NOMBRE = "nombre";
    public static final String PUNTOS = "puntos";
    private static final String DATABASE_TABLE = "jugadores";
    private static final int JUGADORES = 1;
    private static final int JUGADORES_ID = 2;

```



```
private static final UriMatcher uriMatcher; static {
    Matcher = new UriMatcher(UriMatcher.NO_MATCH);
    uriMatcher.addURI(PROVIDER_NAME, "jugadores", JUGADORES);
    uriMatcher.addURI(PROVIDER_NAME, "jugadores/#",
                      JUGADORES_ID);
}
private SQLiteDatabase jugadoresDB;
Definiendo las propiedades del ContentProvider
```

Lo primero que definimos es el nombre del proveedor de contenido. Será a través del cual accederán a nuestros datos. También definimos la uri base completa, en este caso será “content://es.masted.juego/jugadores”.

Las siguientes constantes sirven para abstraer el nombre “real” de las columnas de nuestra base de datos. Al referenciarlas de esta forma, si por lo que sea les cambiamos el nombre, no tendremos que tocar más que esta clase.

JUGADORES y JUGADORES_ID son dos constantes que nos servirán para ver qué tipo de dato nos están pidiendo (un conjunto de registros o uno solo).

El objeto “UriMatcher”, que además inicializamos de forma estática, contiene el listado de direcciones válidas que entiende nuestro proveedor. Si al llamar a nuestra clase, la Uri que nos pasan no está en este listado, lanzaremos una excepción.

Finalmente, la variable jugadoresDB contendrá el objeto a través del cual accederemos a la base de datos SQLite

El siguiente paso es sobrescribir el método *onCreate()*.

```
@Override
public boolean onCreate() {
    Context context = getContext();
    JuegoSQLHelper dbHelper = new JuegoSQLHelper(context);
    jugadoresDB = dbHelper.getWritableDatabase();
    return (jugadoresDB == null) ? false : true;
}
método onCreate
```

Este método se encarga de inicializar la conexión con la base de datos. Como usamos nuestro Helper, si la base de datos no había sido creada, se creará en este momento (y si tiene que ser actualizada también lo hará entonces).

```
@Override
public String getType(Uri uri) {
    switch (uriMatcher.match(uri)) {
        // para conjunto de jugadores
        case JUGADORES:
            return "vnd.android.cursor.dir/vnd.masterd.jugadores";
        // para un solo jugador
        case JUGADORES_ID:
            return "vnd.android.cursor.item/vnd.masterd.jugadores";
    }
}
```

```

    default:throw new IllegalArgumentException("Unsupported
URI: " + uri);
}
}

método getType()

```

El método `getType()` devuelve el `MimeType` que se aplicará a los objetos de respuesta. En nuestro caso dos, uno para los listados de registros y otro para un solo registro.

```

@Override
public Uri insert(Uri uri, ContentValues values) {
// añade un nuevo jugador
long rowID = jugadoresDB.insert(DATABASE_TABLE, "", values);
// si todo ha ido ok devolvemos su Uri
if (rowID > 0) {
Uri _uri = ContentUris.withAppendedId(CONTENT_URI, rowID);
getContext().getContentResolver().notifyChange(_uri, null);
return _uri;
}
throw new SQLException("Failed to insert row into "
+ uri);
}

método insert()

```

Finalmente, deberíamos sobrescribir todos los métodos referentes a datos, veamos un par de ejemplos:

En este caso simplemente tratamos de insertar los valores que recibimos, si todo ha ido bien, devolvemos la URI del objeto insertado.

```

@Override
public Cursor query(Uri uri, String[] projection,
String selection,
String[] selectionArgs, String sortOrder) {
SQLiteQueryBuilder sqlBuilder = new SQLiteQueryBuilder();
sqlBuilder.setTables(DATABASE_TABLE);
if (uriMatcher.match(uri) == JUGADORES_ID) {
sqlBuilder.appendWhere(
_ID + " = " + uri.getPathSegments().get(1)
);
}
if (sortOrder == null || sortOrder == "") {
sortOrder = NOMBRE;
}
Cursor c = sqlBuilder.query(
jugadoresDB, projection, selection,
selectionArgs, null, null, sortOrder
);
c.setNotificationUri(getContext().getContentResolver(), uri);
return c;
}

Método query()

```



Igual que en el *insert*, se trata de generar una consulta (usamos la clase SQLiteQueryBuilder para ayudarnos) a partir de los parámetros recibidos. Una vez ejecutada creamos el cursor que debemos devolver.

De igual manera, implementaríamos los métodos *update* y *delete* si fuera necesario.

El siguiente paso es publicar nuestro proveedor en el AndroidManifest. Bastará un ítem similar a este:

```
<provider
    android:name=".db.JuegoProvider"
    android:authorities="es.masterd.juego" />
```

En el atributo *name* pondremos el nombre de nuestra clase, y en *authorities* debemos usar le mismo nombre que hemos definido en la clase (PROVIDER_NAME)

Ya está todo listo. A partir de ahora, si por ejemplo quisiéramos insertar un registro en nuestra tabla, lo haríamos a través del proveedor de contenido:

```
Uri uri =
Uri.parse("content://es.masterd.juego/jugadores");
ContentValues values = new ContentValues();
values.put("nombre", "Federico");
values.put("puntos", "100");
getContentResolver().insert(uri, values);
```

Insertando un registro usando nuestro ContentProvider

Para mostrar todos los registros en un listado podemos asociar el cursor con un Adapter, por ejemplo vamos a usar los predefinidos por Android dentro de un ActivityList.

Para ello, debemos crear una proyección de campos que se usarán en la consulta.

Así mismo para que el Adapter sepa enlazar las vistas con los campos del resultado preparamos dos arrays que los asocian (camposDb y camposView).

Como vista para cada item, usaremos una de Android que muestra el resultado en dos líneas (android.R.layout.two_line_list_item).

Finalmente, como Adapter, también usaremos uno predefinido: SimpleCursorAdapter.

Con todo configurado solo nos queda asociar el adapter a la ListActivity y listo, ya tenemos nuestro listado:

```
public class DbActivity extends ListActivity {
    private Cursor cursor;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.jugadores);
        final String[] columnas = new String[] {
```

```

JuegoProvider._ID, // 0
JuegoProvider.NOMBRE, // 1
JuegoProvider.PUNTOS, // 2
};
Uri uri =
Uri.parse("content://es.masterd.juego/jugadores/*");
cursor = managedQuery(uri, columnas, null, null, null);
String[] camposDb = new String[] {
JuegoProvider.NOMBRE, JuegoProvider.PUNTOS
};
int[] camposView = new int[] {
    android.R.id.text1, android.R.id.text2
};
SimpleCursorAdapter adapter = new SimpleCursorAdapter(
this,
    android.R.layout.two_line_list_item,
    cursor,
    camposDb,
    camposView
);
setListAdapter(adapter);
}
}

```

DbActivity: Listando los registros de nuestra tabla



master.D

Figura 5. Listado de datos basado en ContentProvider



6. DATOS EN RED

Para acceder a datos que están en la red además de las archiconocidas clases del paquete `java.net.*` Android incorpora unas propias que las complementan bajo `android.net.*`.

Si diseña una aplicación que hace uso de la red, debe tener en cuenta la optimización para que el caudal de datos sea el menor posible y mejorar la experiencia del usuario.

Por ejemplo, si tienes que implementar un servicio web y puedes elegir, escoge JSON antes que SOAP, ya que ante los mismos datos, la respuesta JSON será considerablemente más ligera. Si a esto añades que las peticiones sean de tipo REST, facilitará el trabajo para que sus respuestas puedan ser cacheadas.

7. DATOS DE SESIÓN

Hay datos que son importantes para la sesión actual, pero no lo suficiente como para ser guardados de forma persistente. Estos datos de sesión suelen tener que estar disponibles para todas las Activity de nuestra aplicación.

Una forma de solucionar esto podría ser “arrastrar” estos datos pasándolos como parámetros al lanzar un Intent, otra forma sería guardar dichos datos en un fichero temporal que fuera borrado cuando la ejecución terminara...

Existe una forma más sencilla de solucionar este problema: usar un objeto *Application*.

Los objetos Application definidos en el AndroidManifest son creados cuando arranca la aplicación y tienen su misma vida. Es decir, mientras haya una Activity en la pila o un servicio relacionado funcionando, el objeto mantendrá sus propiedades.

Podemos aprovecharnos de esta característica para crear nuestra propia clase Application que nos sirva, entre otras cosas, como contenedor de los datos de sesión que necesitemos.

Imaginemos que estamos trabajando en un juego que tiene varias Activity (pantalla de juego, de configuración, pausa, records, etc.) y necesitamos que todas ellas sepan el nivel de juego en el que se encuentra el jugador.

Así pues, tendríamos que crear una clase Application que contuviera un campo nivel:

```
public class MiApplication extends Application {  
    int nivel = 0;  
    public int getNivel() {  
        return nivel;  
    }  
    public void setNivel(int nivel) {  
        this.nivel = nivel;  
    }  
}
```

MiActivity.java



El siguiente paso es definir esta clase como objeto Application en el AndroidManifest. Para ello pondremos su nombre en la propiedad android:name del item application.

De esta forma, nuestros componentes podrán recuperar el objeto “MiApplication” a través del método `getApplication()` de Activity:

```
MiApplication app = (MiApplication) getApplication();  
int nivel = app.getNivel();  
Recuperando un dato de sesión
```




CONCLUSIONES



Destrezas, conocimiento y actitud son los elementos que te ayudarán a conseguir el **triunfo profesional**.

El SDK nos facilita una serie de clases y mecanismos que permiten acceder y manejar datos de forma sencilla.

Los proveedores de contenidos son de los más interesantes y potentes que nos permiten tener una forma sencilla y unificada de acceder a cualquier tipo de datos. Además, nos permiten exportar nuestros datos para que otras aplicaciones puedan usarlos.

Del mismo modo, nuestras aplicaciones pueden aprovechar los proveedores de contenido que nos facilita Android o cualquier otra aplicación.

RECAPITULACIÓN

En este capítulo hemos aprendido a:

- Compartir valores sencillos usando la clase “SharedPreferences”.
- Grabar y recuperar datos en la memoria interna usando `openFileOutput()` y `openFileInput()`.
- Usar la memoria externa con `getExternalStorageDirectory()`.
- Acceder a bases de datos SQLite usando SQLiteOpenHelper.
- Manejar proveedores de contenido (ContentProvider).
- Implementar nuestro propio proveedor de contenido.
- Guardar datos de sesión usando una clase Application.



AUTOCOMPROBACIÓN

- 1. ¿Qué clase nos permite guardar datos asociados a una clave de forma persistente?**
 - a) SharedPreferences.
 - b) SharedData.
 - c) KeyPreferences.
 - d) KeyData.

- 2. ¿Qué tipos de datos puede guardar la clase *SharedPreferences*?**
 - a) Cualquier tipo de datos.
 - b) Sólo los datos que implementen su interface.
 - c) Datos de tipo primitivo (*boolean, float, int, long, string*).
 - d) Ninguna de las anteriores.

- 3. ¿Puedo editar directamente una preferencia compartida?**
 - a) Sí.
 - b) No, no se puede.
 - c) No, tengo que usar una clase Editor [se obtiene llamando a *edit()*].
 - d) No, tengo que usar una clase Editor [se obtiene llamando a *edit()* y luego validar los cambios con *commit()*].

4. ¿Qué tipo de Activity podemos usar para implementar un editor de preferencias?
 - a) Activity.
 - b) ListActivity.
 - c) MapActivity.
 - d) PreferenceActivity.
5. ¿Qué necesita la actividad *PreferenceActivity*?
 - a) Un *layout* con la configuración de las preferencias.
 - b) Un fichero xml con la configuración de las preferencias.
 - c) Un *layout* que muestre el formulario de preferencias.
 - d) Nada en especial.
6. ¿Cómo puedo recuperar una preferencia guardada mediante la clase *PreferenceActivity*?
 - a) Con el método getSharedPreferences de Context.
 - b) Con el método getSharedPreferences de la Activity.
 - c) Con el método getSharedPreferences de la clase Application.
 - d) Creando un objeto SharedPreferences.
7. Para obtener un *FileOutputStream* que nos permita guardar datos en la memoria interna ¿qué método usaremos?
 - a) *getOutputStream*.
 - b) *openFileOutput*.
 - c) *openOutputStream*.
 - d) *getFileOutput*.
8. Por defecto, los datos que guardemos en memoria usando un *File OutputStream* serán:
 - a) Públicos, cualquiera podrá leerlos.
 - b) Públicos (solo podrán leerlos aplicaciones que conozcan su clave).
 - c) Privados, solo puede leerlos mi aplicación.
 - d) Depende el parámetro que use al crearlo.



9. ¿Qué método debo usar para conocer cuál es mi directorio de caché?

- a) `getCache()`.
- b) `getCacheDir()`.
- c) `getMyCacheDir()`.
- d) Tengo que definirlo yo.

10. Para almacenar datos en la memoria externa, ¿qué deberemos comprobar?

- a) Que la memoria está insertada.
- b) Que tenemos permisos de escritura.
- c) Que la memoria está insertada y que tenemos permisos de escritura.
- d) Nada, se encarga el sistema.

11. ¿Podemos almacenar los datos de caché en memoria externa?

- a) Sí, gracias a `getExternalCacheDir()`.
- b) Sí, gracias a `getExternalCache()`.
- c) Sí, si creamos nosotros la ruta.
- d) No, el cache siempre va en la memoria interna.

12. ¿Qué clase usaremos para crear una base de datos SQLite?

- a) `SQLite`.
- b) `SQLiteCreate`.
- c) `SQLiteOpenHelper`.
- d) `ContentValues()`.

13. ¿Qué clase utilizamos para almacenar y recuperar datos de una forma unificada?

- a) `Content`.
- b) `ContentProvider`.
- c) `ContentValues`.
- d) `ContentData`.

14. ¿En las bases de datos qué campo usamos por convenio como clave primaria?
- a) Id.
 - b) key.
 - c) _ID.
 - d) _key.
15. ¿Qué tipo de datos devuelve una *query* invocada a través de un proveedor de contenido?
- a) ContentData.
 - b) Cursor.
 - c) HashMap.
 - d) Depende de la consulta.
16. ¿Qué *tag* usamos en el *AndroidManifest* para declarar un proveedor de contenido?
- a) <provider>.
 - b) <content-provider>.
 - c) <content>.
 - d) No es necesario declararlo.
17. ¿Qué *adapter* incluido en Android podemos usar para mostrar directamente los datos de una consulta realizada a través de un proveedor de contenido?
- a) ArrayAdapter.
 - b) BaseAdapter.
 - c) SimpleCursorAdapter.
 - d) SQLiteAdapter.
18. ¿Qué clase podemos usar para guardar datos de sesión?
- a) Session.
 - b) ContentProvider.
 - c) UserData.
 - d) Application.



19. Para realizar las consultas SQLite dentro de un proveedor de contenido ¿Qué clase utilizamos?

- a) SQLiteQueryBuilder.
- b) SQLite.
- c) SQLite3.
- d) SQLite3Query.

20. Para pasar los valores de los campos al método *update* de nuestro proveedor de contenido, ¿qué clase utilizaremos?

- a) ContentValues.
- b) HashMap.
- c) Un objeto que los mapee.
- d) Cualquiera de las anteriores.



SOLUCIONARIO

1.	a	2.	c	3.	d	4.	d	5.	b
6.	a	7.	b	8.	c	9.	b	10.	c
11.	a	12.	c	13.	b	14.	c	15.	b
16.	a	17.	c	18.	d	19.	a	20.	a



Descubre hasta dónde te puede llevar tu esfuerzo. Convírtete en un P8.10.

PROPUESTAS DE AMPLIACIÓN

Ejercicios propuestos:

- Termina el adaptador comentado en este capítulo y añade los métodos *update* y *delete*.
- Realiza una aplicación que muestre un listado de los contactos almacenados en la agenda (usa el proveedor de contenido que nos facilita Android para tales casos).



BIBLIOGRAFÍA

- <http://developer.android.com/resources/samples/ApiDemos/src/com/example/android/apis/content/index.html>.
- <http://developer.android.com/guide/topics/providers/content-providers.html>.

PROGRAMACIÓN PARA ANDROID

8

Mapas y GPS



Escuela Universitaria
de Formación Abierta

master.D
GRUPO



ÍNDICE

MOTIVACIÓN.....	3
PROPÓSITOS	4
PREPARACIÓN PARA LA UNIDAD	5
1. MAPAS	7
1.1. CREAR UN PROYECTO CON SOPORTE PARA GOOGLE MAPS	8
1.2. CREAR UN DISPOSITIVO VIRTUAL CON SOPORTE DE API.....	9
1.3. CONFIGURAR EL ANDROIDMANIFEST	9
1.4. OBTENER UNA CLAVE DE USO DEL API DE GOOGLE MAPS	10
1.5. UTILIZAR LA LIBRERÍA DE MAPAS.....	12
2. USO DE LOS SERVICIOS GPS.....	15
2.1. <i>DEBUG</i> DE APLICACIONES GPS.....	16
CONCLUSIONES	19
RECAPITULACIÓN	20
AUTOCOMPROBACIÓN	21
SOLUCIONARIO.....	27
PROPUESTAS DE AMPLIACIÓN	28
BIBLIOGRAFÍA.....	29



MOTIVACIÓN



Dirígete hacia tu meta sin vacilar; nosotros te ayudamos a llegar hasta allí.

Hasta la popularización del uso de *smartphones*, el uso principal de los sistemas GPS (*Global Positioning System*) estaba limitado a la guía en trayectos.

Con la aparición de los terminales móviles que incorporaban GPS en sus características, su uso se ha extendido. Si a esto le incrementamos la potencia añadida de disponer mapas gratuitos detallados de cualquier parte del mundo se entiende la gran aceptación que están adquiriendo últimamente las aplicaciones con características de geoposicionamiento: juegos, fotografías geolocalizadas, redes sociales, etc.

En este capítulo aprenderemos a usar alguna de las herramientas que nos ofrece Android para incorporar esta funcionalidad a nuestras aplicaciones.

PROPOSITOS

Con esta unidad conseguiremos:

- Aprender a integrar los servicios de Google Maps en nuestras aplicaciones.
- Aprender a interactuar con los sensores del dispositivo, en concreto con el GPS.



PREPARACIÓN PARA LA UNIDAD

Necesitaremos tener el entorno de desarrollo Android instalado y configurado. Aunque se puede usar el emulador para probar los ejemplos, si es posible, usaremos un terminal Android con GPS para realizar las pruebas.



1. MAPAS

Google Maps es un servicio gratuito que ofrece mapas desplazables, fotos satelitales e incluso fotos a pie de calle.

Fue anunciado por Google en 2005. Inicialmente, era una aplicación web, a la que poco a poco fueron añadiéndose funcionalidades: nuevas capas, la aplicación de escritorio Google Earth e incluso una API para poder usarla dentro de aplicaciones de terceros.

El soporte para acceder a Google Maps nos viene dado por una librería externa (incluida en el SDK): com.google.android.maps.

Su clase principal es MapView, que como su nombre indica es un objeto View (ViewGroup) especializado en mostrar los mapas y todos sus controles necesarios.

Cuando los objetos MapView tienen el foco, interpretan las pulsaciones de teclas, gestos y pulsaciones para desplazar, seleccionar o hacer zoom sobre los mapas.

Un atributo imprescindible cuando definimos un MapView es su APIkey, ya que sin una válida, los mapas no se mostrarán.

Pero vayamos por partes, para realizar una aplicación que use mapas debemos seguir los siguientes pasos:

- Instalar el añadido de Google Maps (si usamos el SDK de Android ya nos viene instalado por defecto).
- Crear un nuevo proyecto (o reconfigurar uno) para que tenga soporte Google Maps.
- Crear un dispositivo virtual con soporte para Google Maps (si no se hace, no podremos probar nuestra aplicación en el emulador).
- Añadir la librería com.google.android.maps al manifiesto.

- Si se van a obtener los datos de la red habrá que activar permisos de uso de Internet en el AndroidManifest.
- Usar las clases que necesitemos en nuestra programación.
- Obtener una clave de Google Maps.
- Firmar nuestra aplicación correctamente (la firma debe corresponder con la que usamos al crear la clave de Google Maps).

Como ya hemos dicho, con la instalación que hicimos al principio, usamos el SDK completo con Eclipse así que podemos saltarnos el primer paso.

1.1. CREAR UN PROYECTO CON SOPORTE PARA GOOGLE MAPS

Si nuestra aplicación va a hacer uso de las librerías externas de servicios de Google, cuando creemos el proyecto debemos elegir como “target” uno con la etiqueta “Google APIs”.

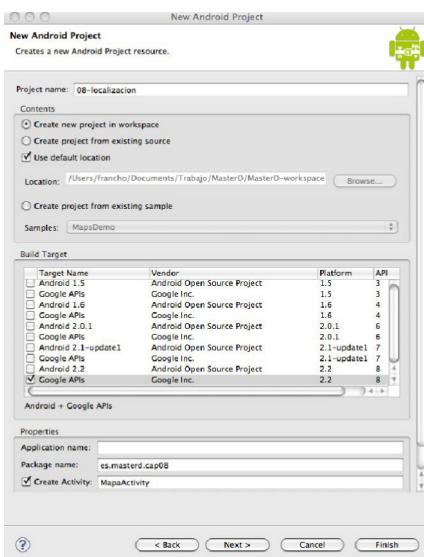


Figura 1. Creando un nuevo proyecto que incluirá mapas

De esta forma el compilador incluirá en nuestro paquete apk las librerías necesarias para su funcionamiento.

Si el proyecto ya está creado, podemos modificar el *target* desde el menú contextual del proyecto.

Existen varias versiones de librería emparejadas con cada una de las versiones de Android. Por temas de compatibilidad intentaremos usar siempre la menor.



1.2. CREAR UN DISPOSITIVO VIRTUAL CON SOPORTE DE API

Así mismo, si todavía no lo hemos hecho, deberemos crear un dispositivo virtual (AVD) que soporte Google API y, ya que estamos, le añadimos soporte GPS para poder probar la última parte del capítulo.

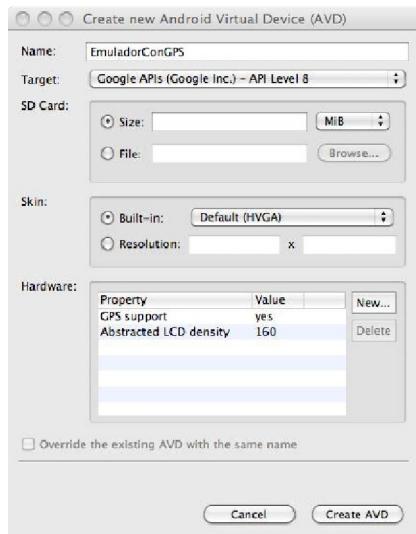


Figura 2. Creando un emulador con soporte GPS y Google Maps

Si no tenemos ningún emulador correctamente configurado, no podremos probar nuestras aplicaciones si no es en un terminal “real”.

1.3. CONFIGURAR EL ANDROIDMANIFEST

Para poder usar las clases de la librería externa debemos referenciarla en el manifiesto usando el tag `<uses-library>`.

Así mismo, si los datos de los mapas los vamos a obtener de la red, deberemos solicitar permiso para usar Internet.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
    xmlns:android="http://schemas.android.com/apk/res/android"
        package="es.masterd.cap08"
        android:versionCode="1"
        android:versionName="1.0">
    <application android:icon="@drawable/icon"
        android:label="@string/app_name">
        <activity android:name=".MapaActivity"
            android:label="@string/app_name">
```

```

<intent-filter>
<action android:name="android.intent.action.MAIN" />
<category
    android:name="android.intent.category.LAUNCHER" />
</intent-filter>
</activity>
<uses-library android:name="com.google.android.maps" />
</application>
<uses-permission android:name="android.permission.INTERNET" />
</manifest>

```

Ejemplo de AndroidManifest para uso de Maps

1.4. OBTENER UNA CLAVE DE USO DEL API DE GOOGLE MAPS

El objeto `MapView` muestra mapas obteniendo las imágenes de un servicio *on-line*. Para empezar a usarlo debemos aceptar y cumplir sus condiciones. Por ello, Google nos obliga crear una clave asociada a la firma de nuestras aplicaciones.

Por defecto, Eclipse nos crea una firma de desarrollo que se usa para firmar las aplicaciones que instalamos en el emulador.

Para obtener nuestra clave, primero tenemos que obtener nuestra firma. Esto se hace desde una consola o ventana de sistema, ejecutando el comando `keytool` al que pasaremos nuestra biblioteca de firmas. Consulte el manual¹ para las instrucciones concretas de su sistema operativo.

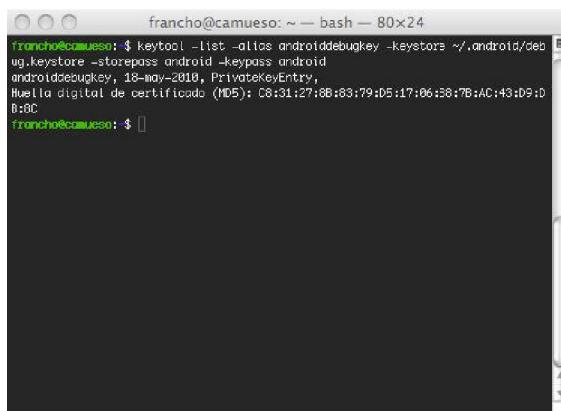


Figura 3. Obteniendo nuestra firma de desarrollo

¹ <http://code.google.com/intl/es-ES/android/add-ons/google-apis/mapkey.html>.



Con esta firma copiada en el portapapeles, entraremos en la web de alta:

- <http://code.google.com/intl/es-ES/android/maps-api-signup.html>.

Tras llenar el formulario y aceptar las condiciones, se nos facilitará la clave a usar en nuestros objetos MapView.

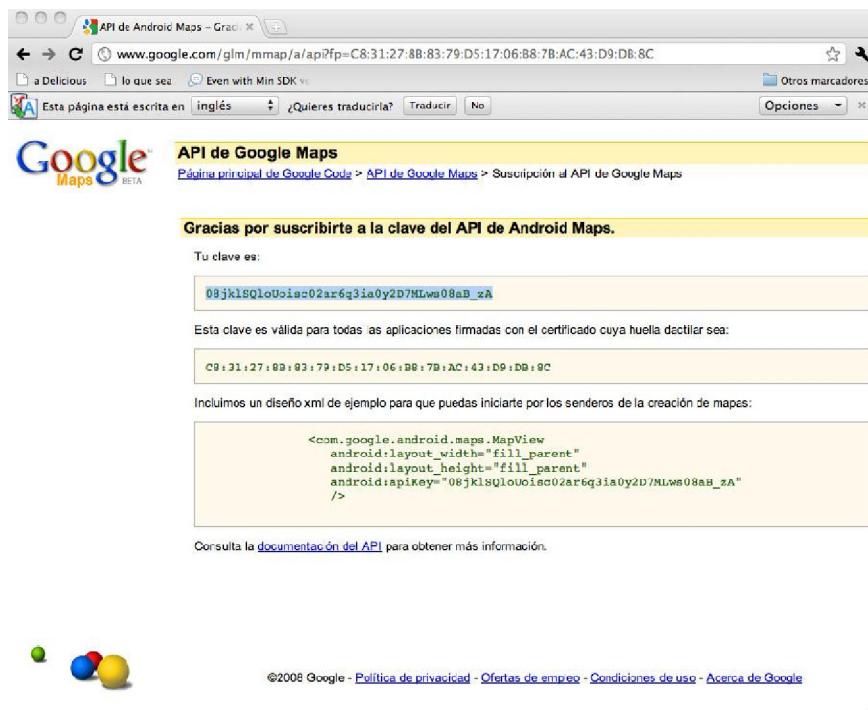


Figura 4. Clave de Google Maps

Una cosa importante que hay que tener en cuenta es que, cuando queramos publicar una aplicación, por ejemplo subiéndola al Android Market, tendremos nuestra firma pública (distinta a la de desarrollo). Esto nos obliga a tener una clave de API por cada firma.

Así que es una buena idea no usar esa clave directamente y colocarla, por ejemplo, en un fichero de recursos para poder cambiarla a la hora de publicar de una forma rápida (por ejemplo con un script que sobreesciba el fichero de clave).

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="googlemaps_apikey">
        08jklSQLoUoisc02ar6q3ia0y2D7MLws08aB_zA
    </string>
</resources>
```

Para facilitar la claridad, la clave la guardamos en un fichero de recursos: ./res/values/string_apimaps.xml

1.5. UTILIZAR LA LIBRERÍA DE MAPAS

Bien ya tenemos todos los requisitos cumplidos, empiezemos a teclear código.

Lo primero que vamos a necesitar es una vista con un objeto MapView. Como se trata de objeto View de una librería externa debemos referenciarlo con su espacio de nombres completo.

En la propiedad `android:apiKey` debemos poner la clave obtenida en el paso anterior. En nuestro caso, en vez de ponerla directamente, la hemos guardado en un fichero de recursos:

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="vertical"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent">
    <com.google.android.maps.MapView
        android:id="@+id/mapa"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:enabled="true"
        android:clickable="true"
        android:apiKey="@string/goologlemaps_apikey" />
</FrameLayout>
./res/layout/mapa.xml
```

Los objetos MapView tienen que ser usados en un tipo de Activity especializado llamado MapActivity.

Con solo cargar nuestro *layout* veremos cargado el mapa del mundo.

```
public class MapaActivity extends MapActivity {
    private MapView mapa;
    private MapController mapController;
    private List<Overlay> mapOverlays;
    private MiItemizedOverlay itemizedoverlay;
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.mapa);
    }
}
```

Uso básico de una MapActivity

Pero un mapa estático no es suficiente, vamos a añadirle funcionalidad. Lo primero será añadir los botones de control de *zoom*. Vamos a usar los propios del objeto View.



Además, vamos a acercar el *zoom* inicial para que se vea más el detalle.

```
// Configure the map
mapa = (MapView) findViewById(R.id.mapa);
mapa.displayZoomControls(true);
mapa.setBuiltInZoomControls(true);
mapController = mapa.getController();
mapController.setZoom(14); // Zoom x14
mapa.setSatellite(true); // Activamos la vista satelite
```

Manipulando el MapView

Ahora vamos a añadir una serie de hitos en el mapa. Para ello vamos a usar la clase *ItemizedOverlay* que a su vez contendrá una colección de *OverlayItem*. Ambas clases pertenecen al paquete *com.google.android.maps*.

OverlayItem define un hito en el mapa junto sus textos asociados e *ItemizedOverlay* es una clase abstracta que pasaremos a nuestro mapa. Nos obliga a sobrescribir dos métodos: *createItem(int pos)* y *size()*.

En nuestro caso, vamos a añadir un método auxiliar que nos permita ampliar puntos junto con un texto a nuestra colección. También sobrescribiremos el método *onTap()* que será invocado cuando alguien pulse sobre uno de nuestros hitos.

```
public class MiItemizedOverlay extends ItemizedOverlay<OverlayItem> {
    private ArrayList<OverlayItem> mOverlays = new ArrayList<OverlayItem>();
    private Context context;
    public MiItemizedOverlay(Context context,
        Drawable defaultMarker) {
        super(boundCenterBottom(defaultMarker));
        this.context = context;
    }
    public void addLocalizacion(double lat,
        double lon, String etiqueta) {
        int lt = (int) (lat * 1E6);
        int ln = (int) (lon * 1E6);
        GeoPoint punto = new GeoPoint(lt, ln);
        OverlayItem item = new OverlayItem(punto, etiqueta, null);
        mOverlays.add(item);
        populate();
    }
    public void clear() {
        mOverlays.clear();
        populate();
    }
    @Override
    protected OverlayItem createItem(int i) {
        return mOverlays.get(i);
    }
    @Override
    public int size() {
        return mOverlays.size();
    }
    @Override
    protected boolean onTap(int index) {
```

```

String etiqueta = mOverlays.get(index).getTitle();
Toast.makeText(context, etiqueta,
Toast.LENGTH_LONG).show();
return false;
}
}

MyItemizedOverlay

```

Obsérvese que para crear un GeoPoint necesitamos pasar números enteros, por eso si tenemos las coordenadas en un “double” debemos multiplicarlas por 1 000 000 y quedarnos con su parte entera.

```

// Marcamos unos puntos en el mapa
mapOverlays = mapa.getOverlays();
Drawable drawable =
this.getResources().getDrawable(R.drawable.marker);
itemizedoverlay = new MiItemizedOverlay(this, drawable);
itemizedoverlay.addLocalizacion(41.669770, -0.812602,
"Punto 1");
itemizedoverlay.addLocalizacion(41.666597, -0.907145,
"Punto 2");
itemizedoverlay.addLocalizacion(41.658657, -0.886501,
"Punto 3");
mapOverlays.clear();
mapOverlays.add(itemizedoverlay);

```

Gestionando hitos

Hemos usado un gráfico propio para poner las marcas en nuestro mapa. Debe ser una imagen plana y Android le añadirá una sombra proyectada. El resultado es este:



Figura 5. Aspecto de nuestro MapView



2. USO DE LOS SERVICIOS GPS

La mayoría de los dispositivos Android traen incorporado un dispositivo GPS con el que podemos obtener las coordenadas donde se encuentra.

Para acceder a estos datos debemos usar un servicio de sistema que obtendremos haciendo una llamada a la función `getSystemService()`. En concreto tendremos que usar un objeto `LocationManager`.

```
LocationManager lm= (LocationManager) getSystemService  
(Context.LOCATION_SERVICE);  
Obteniendo un objeto LocationManager
```

Una vez obtenido el manejador, podemos definir un `Listener` que escuche los cambios de posición y realice las operaciones correspondientes.

`LocationListener` es una clase abstracta que deberemos extender para estos casos. Nos obliga a sobrescribir cuatro métodos:

- `onLocationChanged()`, → cuando la posición GPS cambia, nos pasará el nuevo punto.
- `onProviderDisabled()`, → cuando el usuario desactive el GPS
- `onProviderEnabled()`, → el usuario ha activado el GPS
- `onStatusChanged()`, → el estado del manejador ha cambiado

Por ejemplo, vamos a crear un objeto `Listener` que mostrará por pantalla las coordenadas actuales en un `Toast` al mismo tiempo que centra el mapa en dichas coordenadas.

```
public class MiLocationListener implements LocationListener  
{  
    private Context context;  
    private MapController mapController;
```

```

MiLocationListener(Context context, MapController
mapController) {
    this.context = context;
    this.mapController = mapController;
}
@Override
public void onLocationChanged(Location location) {
    int lat = (int) (location.getLatitude() * 1E6);
    int lon = (int) (location.getLongitude() * 1E6);
    Toast.makeText(context, lat + " " + lon,
        Toast.LENGTH_LONG).show();
    GeoPoint miPunto = new GeoPoint(lat, lon);
    mapController.animateTo(miPunto);
}
@Override
public void onProviderDisabled(String provider) {
    Toast.makeText(context, "GPS desactivado",
        Toast.LENGTH_LONG).show();
}
@Override
public void onProviderEnabled(String provider) {
    Toast.makeText(context, "GPS activado",
        Toast.LENGTH_LONG).show();
}
@Override
public void onStatusChanged(String provider,
    int status, Bundle extras) {
// TODO Auto-generated method stub
}
}

LocationListener

```

Para que nuestro *listener* funcione se lo debemos asignar al manejador que hemos obtenido antes.

Tal y como lo hemos definido, a nuestro *MiLocationListener* tenemos que pasárselo el contexto y el controlador del mapa (para poder posicionar el centro en los nuevos puntos).

```

LocationManager lm = (LocationManager) getSystemService
(Context.LOCATION_SERVICE);
MiLocationListener mlistener = new MiLocationListener
(this, mapController);
lm.requestLocationUpdates(LocationManager.GPS_PROVIDER,
0, 0, mlistener);

```

Asignando el *listener* al servicio de GPS

2.1. DEBUG DE APLICACIONES GPS

Llega el momento de probar nuestra aplicación en el emulador, pero ¿cómo hacemos para simular el movimiento GPS?



Muy sencillo, en la pestaña DDMS de nuestro Eclipse tenemos una vista llamada “Emulator Control”. Desde ella tenemos la opción de enviar coordenadas GPS ficticias.

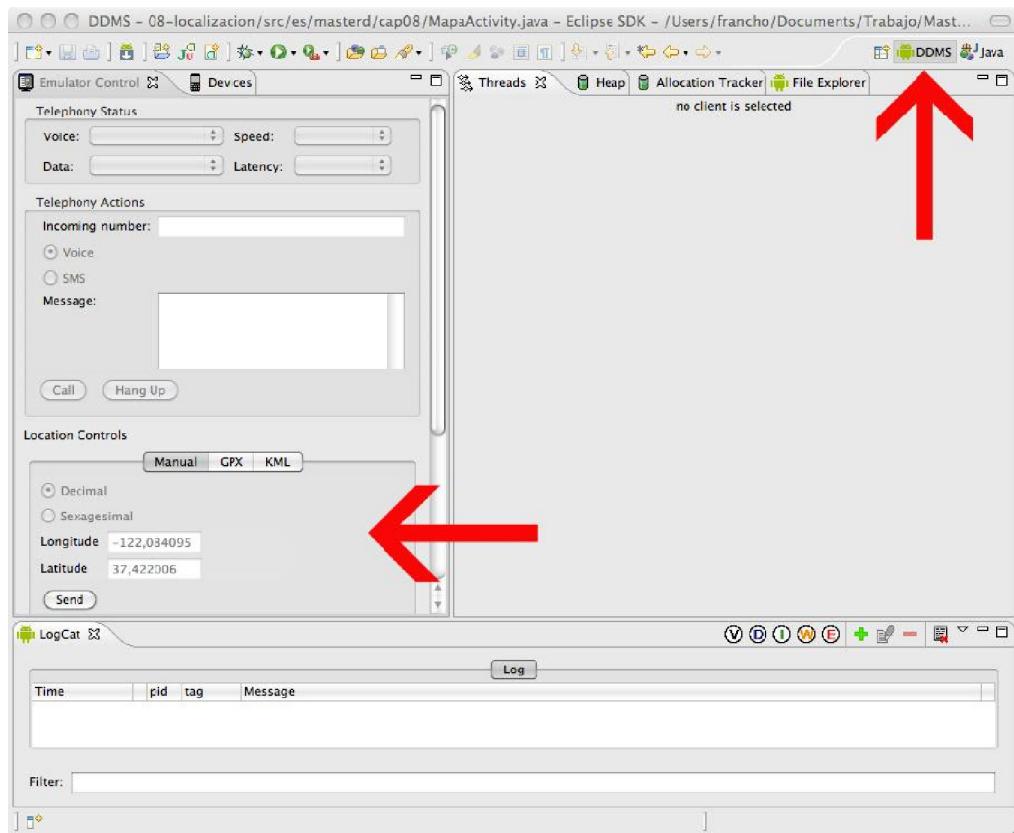


Figura 6. Vista de control de emulador en Eclipse

Podemos introducir las coordenadas de tres formas distintas:

- Definiendo un punto de forma manual.
- Cargando un fichero de datos GPX, el formato estándar de intercambio de datos GPS entre aplicaciones (la mayoría de los dispositivos GPS de mano y aplicaciones de geoposicionamiento soportan este fichero).
- Cargando un fichero de datos KML, →el formato usado Google Maps y Google Earth para intercambio de datos GPS.

Si cargamos un fichero de datos GPS con varias posiciones podremos simular una ruta y ver cómo se comporta nuestra aplicación en esos casos.

Veamos un ejemplo de fichero KML. Aunque estos ficheros pueden ser más complejos y soportan muchos más tags y estructuras de información. Para ser usados por el DDMS se recomienda que sean el formato más sencillo posible:

```
?xml version="1.0" encoding="UTF-8"?>
<kml xmlns="http://earth.google.com/kml/2.2">
<Placemark>
    <Point>
        <coordinates>-0.902675,41.672398,0</coordinates>
    </Point>
</Placemark>
<Placemark>
<name>hito 1</name>
    <Point>
        <coordinates>-0.902643,41.672428,0</coordinates>
    </Point>
</Placemark>
<Placemark>
<name>hito 2</name>
    <Point>
        <coordinates>-0.9026920000000002,41.672401,0</coordinates>
    </Point>
</Placemark>
<Placemark>
    <Point>
        <coordinates>-0.9026110000000002,41.672421,0</coordinates>
    </Point>
</Placemark>
</kml>
```

Ejemplo de fichero KML para simular puntos GPS en el emulador



CONCLUSIONES



Demuestra lo que vales; el triunfo profesional está más cerca de lo que piensas.

Vemos que una vez más Android (y Google) nos proveen de todas las herramientas necesarias para implementar el uso de mapas y GPS en nuestras aplicaciones. Solo queda aplicar nuestra imaginación.

RECAPITULACIÓN

- Para utilizar mapas debemos usar la librería externa `com.google.android.maps` que nos provee un objeto `MapView` que, usado en actividades de tipo `MapActivity`, nos permite mostrar mapas.
- Para interactuar con este View usamos `MapController`, y para definir puntos `ItemizedOverlay`.
- En cambio, el GPS es un servicio del sistema, al que podemos acceder a través de `getSystemService()`, una vez conseguido un objeto de tipo `LocationManager`, podemos añadirle `Listeners` que reaccionen a sus cambios.
- Para probar nuestras aplicaciones que usan el GPS, podemos usar la vista “Emulator Controller” de la pestaña DDMS de Eclipse.



AUTOCOMPROBACIÓN

- 1. ¿Qué librería nos ofrece soporte para mostrar mapas en nuestras aplicaciones?**
 - a) *com.google.android.maps.*
 - b) *com.android.maps.*
 - c) *com.android.google.maps.*
 - d) *android.maps.*

- 2. ¿Cuál es la clase principal de la librería de mapas?**
 - a) *MapLayout.*
 - b) *MapView.*
 - c) *MapGroupView.*
 - d) *MapControllerView.*

- 3. ¿Qué es necesario para poder definir un objeto mapa?**
 - a) *CoordinateKey.*
 - b) *MapaKey.*
 - c) *Mkey.*
 - d) *API key.*

4. **¿Qué permisos necesitaremos declarar en el AndroidManifest para poder mostrar un mapa?**
 - a) ACCESS_FINE_LOCATION.
 - b) INTERNET.
 - c) MAP_ACCESS.
 - d) ACCESS_COARSE_LOCATION.
5. **¿Qué deberemos declarar en el Manifiesto para mostrar mapas, aparte de permisos?**
 - a) <uses-library android:name="com.google.android.maps" />.
 - b) <uses-library android:name="com.android.maps" />.
 - c) <uses-external-library android:name="com.google.android.maps" />.
 - d) <uses-external-library android:name="com.android.maps" />.
6. **¿Dónde colocaremos el API Key del mapa?**
 - a) En el AndroidManifest.xml.
 - b) En la MapActivity.
 - c) En el MapView.
 - d) En un fichero de recursos.
7. **Un objeto MapView ¿dónde puede ser usado?**
 - a) En un ListActivity.
 - b) En un MapActivity.
 - c) En cualquier Activity.
 - d) En un ExternalActivity.
8. **¿Con qué método podemos controlar el nivel de zoom que tiene que mostrar nuestro mapa?**
 - a) `MapView.setZoom(7)`.
 - b) `mapController.setZoom(7)`.
 - c) `MapActivity.setZoom(7)`.
 - d) `setZoomLevel(7)`.

**9. ¿Para activar la capa de vista por satélite qué usaremos?**

- a) El método `setSatellite(true)` de un `MapView`.
- b) La propiedad `setType= "satellite"` en la definición de un `layout`.
- c) El método `setSatellite(true)` de una `MapActivity`.
- d) El método `showSatellite(true)` de un `MapActivity`.

10. ¿Qué objeto usamos para definir puntos geográficos?

- a) `Point`.
- b) `MapPoint`.
- c) `MapViewPoint`.
- d) `GeoPoint`.

11. ¿Qué tipo de parámetros debemos pasar al constructor de dicho objeto?

- a) `Double`.
- b) `Long`.
- c) `Coordinate`.
- d) `Integer`.

12. ¿Qué definición de variable para almacenar una latitud es correcta?

- a) `Double lat = -0.3322234.`
- b) `Long lat = 0.`
- c) `Integer lat = (Integer)(-0.332 * 1E6).`
- d) `Integer lat = Map.getLatitude(0.322344, 12333).`

13. ¿Qué objeto debemos definir para colocar un hito en un mapa?

- a) `OverlayItem`.
- b) `ItemOverlay`.
- c) `GeoPoint`.
- d) `PointGeo`.

14. ¿Cómo obtendremos un controlador de mapa?

- a) Llamando al método `getMapController()` de un `MapActivity`.
- b) Llamando a `getSystemService(Context.MAP_SERVICE)`.
- c) Llamando al método `getController()` de un `MapView`.
- d) Llamando al método `MapView.controller()`.

15. ¿Cómo obtendremos un manejador GPS?

- a) Llamando a `getSystemService(Context.LOCATION_SERVICE)`.
- b) Llamando al método `getGpsController` de un `MapView`.
- c) Llamando al método `getGpsService` de un `MapActivity`.
- d) A través de la superclase `R.services.gps`.

16. ¿Qué tipo de objeto definiremos para observar el GPS?

- a) Una clase que extienda `LocationListener`.
- b) `LocationListener`.
- c) `GpsListener`.
- d) Un clase que extienda `GpsListener`.

17. ¿Qué método deberemos sobrescribir para gestionar los cambios de posición GPS?

- a) `onProviderEnabled`.
- b) `onProviderDisabled`.
- c) `onLocationChanged`.
- d) `onStatusChanged`.

18. Para probar nuestras aplicaciones que usen mapas...

- a) ...sirve cualquier emulador Android.
- b) ...sólo sirve un terminal real.
- c) ...debo crear un emulador con soporte para APIs de Google.
- d) ...sirve cualquier emulador Android siempre que tenga configurado el acceso a Internet.



19. Para publicar una aplicación que use Google Maps en el Android Market...

- a) ...tendré que obtener una nueva API Key con la firma de producción que use para generar la aplicación final.
- b) ...me sirve cualquiera de las firmas que tenga registradas en Google.
- c) ...no necesito ninguna firma ya que se la asigna el Market automáticamente.
- d) ...se solicitarán los datos al usuario para obtener la firma de su copia.

20. Para simular cambios GPS en el emulador, a través del DDMS podré cargar ficheros de coordenadas del tipo:

- a) .GeoPoint.
- b) .kml.
- c) .geo.
- d) .point.



SOLUCIONARIO

1.	a	2.	b	3.	d	4.	b	5.	a
6.	c	7.	b	8.	b	9.	a	10.	d
11.	d	12.	c	13.	a	14.	c	15.	a
16.	a	17.	c	18.	c	19.	a	20.	b



Dedición, esfuerzo, actitud... Un P8.10 siempre es capaz de alcanzar el triunfo profesional.

PROPUESTAS DE AMPLIACIÓN

Pruebe a realizar una aplicación que coloque en un mapa un hito gráfico en cada posición GPS por la que se va pasando.



BIBLIOGRAFÍA

- Manual Android: Location and Maps
 - <http://developer.android.com/guide/topics/location/index.html>.
- Documentación sobre el API de mapas de Google:
 - <http://code.google.com/intl/es-ES/android/add-ons/google-apis/>.

PROGRAMACIÓN PARA ANDROID

9

Servicios de telefonía



ÍNDICE

MOTIVACIÓN.....	3
PROPÓSITOS	4
PREPARACIÓN PARA LA UNIDAD	5
1. MENSAJES DE TEXTO	7
1.1. ENVIAR SMS.....	7
1.2. RECIBIR SMS.....	10
1.3. ACCEDER A MENSAJES GUARDADOS	11
2. LLAMADAS DE TELÉFONO	13
2.1. REALIZAR LLAMADAS TELEFÓNICAS	15
3. ACCEDER A LA AGENDA.....	16
CONCLUSIONES.....	19
RECAPITULACIÓN	20
AUTOCOMPROBACIÓN	21
SOLUCIONARIO	27
PROPUESTAS DE AMPLIACIÓN	28
BIBLIOGRAFÍA.....	29



MOTIVACIÓN



Descubre como con **Master.D** llegarás muy lejos.

Normalmente nuestras aplicaciones Android van a ejecutarse en teléfonos móviles por lo que estamos expuestos a recibir llamadas, SMS, etc. También puede sernos de utilidad para nuestros programas utilizar sus capacidades para enviar SMS o realizar llamadas telefónicas.

En este capítulo conoceremos las clases del SDK que nos permiten hacer estas acciones.

PROPOSITOS

Conocer y manejar las principales clases del paquete android.telephony. En concreto aprenderemos a:

- Enviar SMS.
- Recibir SMS.
- Leer SMS guardados.
- Atender a cambios de estado del teléfono provocados por llamadas.
- Llamar a un número de teléfono.



PREPARACIÓN PARA LA UNIDAD

Para esta unidad es necesario tener el entorno de trabajo configurado y haber comprendido las lecciones anteriores.



1. MENSAJES DE TEXTO

Los SMS (también conocidos como mensajes de texto o mensajes cortos) son uno de los servicios que más se usa en los móviles. Nos permiten enviar y recibir mensajes de texto usando las redes de telefonía (GSM, 3G, etc.).

Aunque Android ya trae una aplicación para gestionar los mensajes de texto, también nos provee de un API de programación para poder manejarlos en nuestras aplicaciones, o incluso crear una aplicación de gestión SMS que sustituya a la que viene por defecto.

1.1. ENVIAR SMS

Lo primero que deberemos hacer para que nuestras aplicaciones puedan enviar SMS será solicitar permiso al usuario. Para ello deberemos declarar en el `AndroidManifest` el permiso `android.permission.SEND_SMS`.

Una vez declarado podremos usar sin problema el método `sendTextMessage()` de la clase `SmsManager`.

Veamos un ejemplo práctico sencillo: vamos a hacer una aplicación que, basándose en los datos de un formulario, envíe un SMS.

Lo primero de todo es solicitar el permiso de envío en nuestro Manifiesto:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
    xmlns:android="http://schemas.android.com/apk/res/android"
        package="es.masterd.u90telefonia"
        android:versionCode="1"
        android:versionName="1.0">
    <application android:icon="@drawable/icon"
        android:label="@string/app_name"
        android:theme="@style/MiTema">
        <activity android:name=".SendSMSActivity"
            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>
<uses-permission
    android:name="android.permission.SEND_SMS"/>
</manifest>

```

Declarando permiso para enviar SMS en el AndroidManifest

Luego crearemos un *layout* que nos sirva para capturar los datos del envío. Necesitaremos un número de teléfono y un texto:

```

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="vertical"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:layout_gravity="center_horizontal">
<TableLayout android:id="@+id/TableLayout01"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content">
<TableRow android:layout_width="fill_parent"
    android:layout_height="wrap_content">
<TextView android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Número Teléfono"
    android:gravity="top|right" />
<EditText android:id="@+id/NumeroTelefono"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="" android:layout_weight="4" />
</TableRow>
<TableRow android:layout_width="fill_parent"
    android:layout_height="wrap_content">
<TextView android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Texto"
    android:gravity="top|right" />
<EditText android:id="@+id/TextoSms"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="" android:lines="3"
    android:layout_weight="4" />
</TableRow>
</TableLayout>
<Button android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/BotonEnviar"
    android:text="Enviar SMS"
    android:layout_gravity="center_horizontal"></Button>
</LinearLayout>

```

Layout de envío



Finalmente crearemos una Activity que se encargue de realizar el envío en cuanto se pulse el botón:

```
public class SendSMSActivity extends Activity {  
    /** Called when the activity is first created. */  
    @Override  
    public void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.sms);  
        final EditText tlf = (EditText) findViewById(R.id.NumeroTelefono);  
        final EditText sms = (EditText) findViewById(R.id.TextoSms);  
        Button boton = (Button) findViewById(R.id.BotonEnviar);  
        boton.setOnClickListener(new OnClickListener() {  
            @Override  
            public void onClick(View v) {  
                String destino = tlf.getText().toString();  
                String texto = sms.getText().toString();  
                enviaSMS(destino, texto);  
                tlf.setText("");  
                sms.setText("");  
            }  
        });  
    }  
    private void enviaSMS(String destino, String texto) {  
        try {  
            // El envío propiamente dicho  
            SmsManager smsMgr = SmsManager.getDefault();  
            smsMgr.sendTextMessage(destino, null, texto, null, null);  
            // Avisamos al usuario  
            Toast.makeText(SendSMSActivity.this, "SMS enviado",  
                Toast.LENGTH_LONG).show();  
        } catch (Exception e) {  
            Toast.makeText(SendSMSActivity.this, "Error en el  
                envío: " + e.getLocalizedMessage(),  
                Toast.LENGTH_SHORT).show();  
        }  
    }  
}
```

Activity de envío de SMS

En este caso, hemos usado dos *toast* para notificar el resultado del envío al usuario.

También existe la posibilidad de definir un Intent que será llamado cuando el envío sea correcto y otro que se llamará si falla y pasarlos al método *sendTextMessage*. Consulta la documentación para los detalles concretos.

1.2. RECIBIR SMS

Antes de nada, para poder manipular los mensajes de texto entrantes, debemos pedir permiso al usuario definiendo en el AndroidManifest el permiso android.permission.RECEIVE_SMS.

El siguiente paso será crear una clase que extienda de BroadcastReceiver. Esta es una clase abstracta que nos obliga a implementar el método `onReceive()`, que será donde colocaremos nuestro código. Recibiremos como parámetros el contexto de la llamada y un Intent con los datos del SMS recibido dentro de la etiqueta “pdus”

PDU son las siglas de *Protocol Data Units*, el protocolo usado en el intercambio de mensajes SMS.

Lo más normal es convertir estos PDUs en una clase SmsMessage para facilitar su manejo.

Veamos un ejemplo práctico. Vamos programar un BroadcastReceiver que cada vez que llegue un mensaje de texto lo muestre en una ventana flotante.

```
public class SmsBroadcastReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        try {
            Object[] pdus = (Object[]) intent.getExtras().get("pdus");
            int numPdus = pdus.length;
            SmsMessage[] mensajes = new SmsMessage[numPdus];
            for(int x=0;x<numPdus; x++) {
                mensajes[x] = SmsMessage.createFromPdu((byte[]) pdus[x]);
                String from = mensajes[x].getOriginatingAddress();
                String text = mensajes[x].getDisplayMessageBody();
                Toast.makeText(context, "De: "+from+"\n\n"+text,
                Toast.LENGTH_LONG).show();
            }
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

Ejemplo de BroadcastReceiver

Pero con esto no basta, debemos registrar nuestro “receptor” para que el sistema sepa que debe notificarlo cuando se produzca el evento.

Para ello, debemos definirlo en correctamente en el AndroidManifest:

```
<receiver android:name=".SmsBroadcastReceiver">
<intent-filter>
<action
    android:name="android.provider.Telephony.SMS_RECEIVED" />
</intent-filter>
</receiver>
```

Declarando el BroadcastReceiver en el AndroidManifest



Si nos fijamos, al declararlo le asignamos un filtro para que el sistema sepa que debe ser avisado cuando se produce un evento de telefonía, en concreto, cuando se reciba un SMS.

1.3. ACCEDER A MENSAJES GUARDADOS

Normalmente las aplicaciones de mensajería almacenan los mensajes enviados y recibidos en distintas carpetas para que el usuario pueda consultarlos a posteriori.

Para poder acceder a dichos mensajes debemos declarar el permiso android.permission.READ_SMS.

A pesar de que, de momento, no es un componente “oficial” y no viene documentada, existe un proveedor de contenido que nos permite acceder a dichas carpetas a través de la Uri content://sms/.

Así pues, para sacar un listado de los mensajes que tenemos en la carpeta de enviados, deberíamos crear una Activity similar a esta:

```
public class SendSMSActivity extends ListActivity {  
    /** Called when the activity is first created. */  
    @Override  
    public void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.sms);  
        // Cargamos la lista de mensajes enviados  
        Uri uri = Uri.parse("content://sms/sent");  
        Cursor cursor = getContentResolver().query  
            (uri, null, null, null, null);  
        startManagingCursor(cursor);  
        // Relacionamos las columnas con la vista  
        String[] columns = new String[] { "address", "body" };  
        int[] names = new int[] { android.R.id.text1,  
            android.R.id.text2 };  
        // Usamos un adapter del sistema  
        SimpleCursorAdapter adapter = new SimpleCursorAdapter  
            (this, android.R.layout.simple_list_item_2, cursor, columns,  
            names);  
        // Lo asociamos con la Activity  
        setListAdapter(adapter);  
    }  
}
```

Listado de mensajes enviados

Igualmente si nosotros quisiéramos guardar en dicha carpeta los mensajes enviados por nuestra aplicación, podríamos usar este proveedor de contenido. Eso si declarando antes el en AndroidManifest el permiso android.permission.WRITE_SMS:

```

try {
    SmsManager smsMgr = SmsManager.getDefault();
    smsMgr.sendTextMessage(destino, null, texto, null, null);
    // Si tenemos acceso al proveedor de contenidos SMS
    // Guardamos el mensaje en la carpeta de enviados
    if (getPackageManager().resolveContentProvider
        ("sms", 0) != null) {
        ContentValues values = new ContentValues();
        values.put("address", destino);
        values.put("body", texto);
        Uri uri = Uri.parse("content://sms/sent");
        getContentResolver().insert(uri, values);
    }

    Toast.makeText(SendSMSActivity.this,
        "SMS enviado",
        Toast.LENGTH_LONG).show();
} catch (Exception e) {
    Toast.makeText(SendSMSActivity.this,
        "Error en el envío: " + e.getLocalizedMessage(),
        Toast.LENGTH_SHORT).show();
}

```

Nuestro código de envío SMS modificado, para que se guarde en el buzón de enviados

Como ya se ha comentado antes, el proveedor de contenidos “sms” no está documentado en la documentación oficial, lo que significa que puede ser modificado sin previo aviso. Por eso, es una buena idea que si lo usamos, lo encerramos en un bloque *try* y realicemos las comprobaciones pertinentes para evitar que nuestra aplicación falle el día de mañana.



2. LLAMADAS DE TELÉFONO

El SDK también nos provee de una serie de herramientas para interactuar con los cambios de estado de terminal provocados por las llamadas telefónicas.

Para poder acceder a ellos, deberemos obtener el permiso android.permission.READ_PHONE_STATE que deberemos declarar en el AndroidManifest:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
    xmlns:android="http://schemas.android.com/apk/res/android"
        package="es.masterd.u90telefonia"
        android:versionCode="1"
        android:versionName="1.0">
    <application android:icon="@drawable/icon"
        android:label="@string/app_name"
        android:theme="@style/MiTema">
        <activity android:name=".LlamadasActivity"
            android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <uses-permission
            android:name="android.permission.READ_PHONE_STATE" />
    </manifest>
```

A partir de este momento, y gracias a la clase TelephonyManager, podremos responder a los cambios de estado del teléfono.

Esto es muy útil (aparte del uso común del *dialer*) para realizar acciones automáticas que mejoren la experiencia de usuario.

Imaginemos por ejemplo que programamos un juego. Sería una buena idea que si el terminal recibe una llamada, el juego quede en pausa, y una vez colgemos el teléfono se reanude la partida.

Veamos un ejemplo sencillo de uso de esta clase (en lugar de sacar los *toast*, pondríamos el código necesario para interactuar con el juego):

```
public class LlamadasActivity extends Activity {
    @Override
        protected void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
            setContentView(R.layout.agenda);
            TelephonyManager tManager = (TelephonyManager)
                getSystemService(Context.TELEPHONY_SERVICE);
            tManager.listen(new TelefonoListener(),
                PhoneStateListener.LISTEN_CALL_STATE);
        }
    class TelefonoListener extends PhoneStateListener {
        @Override
        public void onCallStateChanged(int state, String number) {
            switch (state) {
                case TelephonyManager.CALL_STATE_RINGING:
                    Toast.makeText(LlamadasActivity.this,
                        "Telefono sonado...." + number,
                        Toast.LENGTH_LONG).show();
                    break;
                case TelephonyManager.CALL_STATE_OFFHOOK:
                    Toast.makeText(LlamadasActivity.this, "descolgado..",
                        Toast.LENGTH_LONG).show();
                    break;
                case TelephonyManager.CALL_STATE_IDLE:
                    Toast.makeText(LlamadasActivity.this, "En espera...",
                        Toast.LENGTH_LONG).show();
                    break;
                default:
                    super.onCallStateChanged(state, incomingNumber);
                    break;
            }
        }
    }
}
```

Usando un TelephonyManager

Como se puede observar, el *TelephonyManager* lo obtenemos como servicio de sistema.

Una vez obtenido, le cargamos un *listener* personalizado que “escuchará” los cambios de estado provocados por llamadas telefónicas (*PhoneStateListener*.LISTEN_CALL_STATE).

Para ello hemos sobrescrito el método *onCallStateChanged* que recibe dos parámetros: el nuevo estado y el número de teléfono que llama.

Luego solo es cuestión de filtrar cada uno de los estados y reaccionar de la forma que necesitemos.



En este caso, hemos programado una respuesta a las llamadas, pero de igual forma podemos actuar ante, por ejemplo, cambios en la intensidad de la señal, cambios en la conexión de datos, etc.

2.1. REALIZAR LLAMADAS TELEFÓNICAS

Bajo el paquete android.internal.telephony tenemos disponibles todas las clases de bajo nivel para gestionar los servicios de telefonía.

Sin embargo, no es necesario meterse a tan bajo nivel ya que el sistema trae instalado por defecto en todos los terminales una aplicación (llamada PhoneApp) que se encarga de todas realizar las llamadas por nosotros.

Si nos fijamos, en los móviles Android hay diferentes formas de iniciar una llamada telefónica: desde la agenda, pulsando sobre un enlace directo, desde el registro de llamadas, etc. pero todas ellas en última instancia muestran la misma aplicación.

Ya hemos visto que Android nos permite reutilizar módulos del resto de aplicaciones gracias a los filtros de *itents*, y este es el mejor ejemplo de uso.

Así pues, para realizar una llamada deberemos crear un *itent* al que pasaremos el número de teléfono y aplicaremos el filtro necesario para que la aplicación PhoneApp responda a nuestra solicitud y gestione la llamada.

Veamos cómo se haría de forma práctica. Vamos a dotar a un botón de nuestro *layout* de la funcionalidad de realizar una llamada telefónica:

```
Button llamar = (Button) findViewById(R.id.llamar);
llamar.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
        try {
            Intent llamada = new Intent(Intent.ACTION_CALL);
            Uri uriTlf = Uri.parse("tel:976445545");
            llamada.setData(uriTlf);
            startActivity(llamada);
        } catch (Exception e) {
            Toast.makeText(v.getContext(),
                    "Error al realizar la llamada", Toast.LENGTH_LONG)
                    .show();
            e.printStackTrace();
        }
    }
});
```

Boton de llamada

Para que el código anterior funcione no debemos olvidarnos de declarar en el AndroidManifest el permiso de android.permission.CALL_PHONE.

3. ACceder a la agenda

Desde el Api 5 (Android 2.0)¹ se incorporó al SDK la superclase `ContactsContract` que tal y como dice la documentación “Es el contrato entre el proveedor de contenidos de contactos y las aplicaciones”.

Se trata de un proveedor de contenidos flexible y abierto que facilita que las distintas aplicaciones añadan su propia información a nuestra agenda de contactos.

La información se guarda bajo tres enfoques:

- Una fila en la tabla `ContactsContract.Data` puede almacenar cualquier tipo de información personal (teléfono, e-mail, etc.). Los tipos de información están abiertos. Es decir, hay una serie de tipos predefinidos, pero cada aplicación puede añadir los suyos propios (por ejemplo: cuenta de Twitter, Skype, etc.).
- Una fila en `ContactsContract.RawContracts` representa un conjunto de datos que describen una sola cuenta (por ejemplo, una de las cuentas de Gmail).
- Finalmente, una fila en la tabla `ContactsContract.Contacts` representa una asociación de `RawContracts` que presumiblemente describen a la misma persona (cuando uno de estos `RawContracts` cambia el resto de asociados y se actualiza si es necesario).

Veamos un ejemplo de uso de este proveedor de contenido. Vamos programar una pequeña Activity que muestre un listado de todos los contactos de nuestra agenda, y al pulsar sobre uno de ellos se desplegará el listado de teléfonos que tenemos para esa persona.

¹ Para versiones anteriores se debe usar el antiguo API `android.provider.Contacts.People`.



Como siempre, lo primero de todo es pedir permiso al usuario. En este caso como sólo vamos a leer los contactos sólo declararemos en el AndroidManifest el permiso de android.permission.READ_CONTACTS.

Para mostrar el listado vamos a usar SimpleCursorAdapter, un adaptador que viene con el SDK al que debemos pasar el cursor resultado de la consulta, la vista a usar en los ítems y dos arrays que sirven para enlazar los componentes de la vista y los campos del cursor.

```
public class AgendaActivity extends ListActivity {  
    private Cursor cursor;  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.agenda);  
        String[] campos = new String[]  
        { ContactsContract.Contacts.DISPLAY_NAME };  
        int[] views = new int[] { android.R.id.text1 };  
        cursor = getContentResolver().query(  
            ContactsContract.Contacts.CONTENT_URI,  
            null, null, null, null);  
        SimpleCursorAdapter adapter = new SimpleCursorAdapter(  
            AgendaActivity.this, // Contexto  
            android.R.layout.simple_list_item_1, // Vista item  
            cursor, // Cursor con datos  
            campos, // Proyección de campos  
            views // Proyección de vistas  
        );  
        setListAdapter(adapter);  
    }  
}
```

Listado de contactos de la agenda

Veamos otra forma de recorrer cursos. En el evento lanzado al pulsar sobre un ítem, haremos una nueva consulta para obtener los teléfonos que tenemos asociados a ese usuario.

El resultado debemos convertirlo a un array de *String* que usaremos para generar una alerta que permita seleccionarlos.

```
@Override  
protected void onListItemClick(ListView l, View v,  
    int position, long id) {  
    Cursor tlfCur = getContentResolver().query(  
        ContactsContract.CommonDataKinds.Phone.CONTENT_URI,  
        // Url de búsqueda  
        null, // proyección de campos (sacamos todos)  
        ContactsContract.CommonDataKinds.Phone.CONTACT_ID  
        + " = ?", // Condición  
        new String[] { "" + id }, // Campos para la condición  
        null); // Orden  
    // Generar la lista de teléfonos para crear un diálogo  
    int nTelefonos = tlfCur.getCount();  
    final String[] telefonos = new String[nTelefonos];
```

```
int x = 0;
while (tlfCur.moveToNext()) {
    int col = tlfCur
        .getColumnIndex(ContactsContract.CommonDataKinds.
            Phone.NUMBER);
    telefonos[x++] = tlfCur.getString(col);
}
tlfCur.close();
// Creamos el dialogo
AlertDialog.Builder builder = new AlertDialog-
    .Builder(this);
builder.setTitle("Selecciona teléfono");
builder.setItems(telefonos, new
    DialogInterface.OnClickListener() {
        public void onClick(DialogInterface dialog, int item) {
            telefonos[item],
            Toast.LENGTH_SHORT).show();
        }
    });
AlertDialog alert = builder.create();
alert.show();
}
```

Mostrando teléfonos disponibles



CONCLUSIONES



Domina la materia y entrénate para alcanzar todos tus objetivos.

El API de telefonía de Android nos permite acceder a las funciones básicas de cualquier teléfono móvil. Gracias a ella podemos realizar o recibir llamadas, enviar SMS o acceder a la agenda por poner algún ejemplo.

Así que no es muy difícil integrar estas funcionalidades en nuestros programas o sustituir las aplicaciones de marcado o mensajería por las nuestras.

Las clases de telefonía también nos permiten mejorar la experiencia de usuario realizando acciones automáticas. Recordemos el ejemplo de la pausa automática en un juego cuando se recibe una llamada.

RECAPITULACIÓN

- SmsManager es la clase encargada de las funciones relacionadas con la mensajería SMS.
- Para reaccionar ante la llegada de un SMS podemos implementar un BroadcastReceiver que definiremos en el AndroidManifest como filtro del *intent* android.provider.Telephony.SMS_RECEIVED.
- La clase TelephonyManager permite implementar *Listeners* (PhoneStateListener) que escuchan a los cambios de estado del teléfono.
- Para realizar llamadas, la forma más sencilla es usar una llamada a un *intent* filtrando por Intent.ACTION_CALL.
- En cuanto a la agenda, podemos acceder y modificar sus datos gracias a su estructura de datos abierta y a la clase ContactsContract.
- En cualquier caso deberemos accordarnos de declarar los permisos necesarios en el AndroidManifest o nuestra aplicación fallará en tiempo de ejecución.



AUTOCOMPROBACIÓN

- 1. ¿Qué clase deberemos usar para mandar un SMS?**
 - a) SmsSender.
 - b) SmsManager.
 - c) Sms.
 - d) SmsProvider.

- 2. ¿Qué permiso deberemos declarar para enviar un SMS?**
 - a) android.permission.sms.SEND.
 - b) android.permission.SEND_MESSAGE.
 - c) android.permission.SEND_SMS.
 - d) android.permission.sms.SEND_MESSAGE.

- 3. ¿Cómo se llama el método que invocaremos para enviar un SMS de texto?**
 - a) sendTextMessage.
 - b) sendSms.
 - c) sendMessage.
 - d) sendTextSms.

4. **¿Cómo debemos instanciar la clase que usaremos para enviar un SMS?**
 - a) new SmsSender(context).
 - b) new SmsManager(context).
 - c) getSystemService(Context.SMS_SERVICE).
 - d) SmsManager.getDefault().
5. **¿Qué clase extenderemos si queremos atender a los mensajes SMS entrantes?**
 - a) BroadcastReceiver.
 - b) SmsReceiver.
 - c) BroadcastListener.
 - d) SmsListener.
6. **¿Cómo deberemos definir en el AndroidManifest una clase para atender mensajes SMS entrantes?**
 - a) Como una activity con el filtro de *intent* android.provider.Telephony.SMS_RECEIVED.
 - b) Como un receiver con un filtro de *intent* android.provider.Telephony.SMS_RECEIVED.
 - c) Como un service con un filtro de *intent* android.provider.Telephony.SMS_RECEIVED.
 - d) No es necesario definirlo en el AndroidManifest.
7. **¿Qué permiso deberemos definir en el AndroidManifest para poder gestionar la recepción de SMS?**
 - a) android.permission.RECEIVE_SMS.
 - b) android.permission.sms.RECEIVE.
 - c) android.permission.message.RECEIVE_TEXT.
 - d) android.permission.RECEIVE_TEXT.



- 8. ¿Qué método será llamado en un BroadcastReceiver cuando se reciba un mensaje?**
 - a) onNewSms.
 - b) onReceive.
 - c) onReceiveSms.
 - d) onMessage.

- 9. ¿Qué permiso hay que declarar para poder leer los mensajes guardados?**
 - a) android.permission.READ_SMS.
 - b) android.permission.sms.READ.
 - c) android.permission.messages.READ.
 - d) android.permission.read.SMS.

- 10. ¿Qué proveedor de contenidos podemos usar para acceder a los SMS recibidos?**
 - a) content://sms/in.
 - b) content://sms/inbox.
 - c) content://telephony/sms/in.
 - d) content://telephony/sms/inbox.

- 11. ¿Qué proveedor de contenidos podemos usar para guardar un mensaje en la carpeta de SMS enviados?**
 - a) content://sms/sent.
 - b) content://sms/sentbox.
 - c) content://telephony/sms/sent.
 - d) content://telephony/sms/sentbox.

- 12. ¿Son oficiales los proveedores de contenido para manipular los SMS guardados?**
 - a) Son una clase externa.
 - b) Si, son componentes oficiales del SDK.
 - c) Están en el SDK, pero no documentados, por lo que pueden sufrir modificaciones.
 - d) No existen como parte del SDK.

13. ¿Qué permiso debemos declarar para poder guardar un SMS?

- a) android.permission.WRITE_SMS.
- b) android.permission.sms.WRITE.
- c) android.permission.telephony.WRITE_SMS.
- d) android.permission.telephony.SAVE_SMS.

14. ¿Qué permiso debemos declarar para poder responder ante una llamada entrante?

- a) android.permission.TELEPHONY.
- b) android.permission.INCOMING_CALL.
- c) android.permission.phone.STATE.
- d) android.permission.READ_PHONE_STATE.

15. ¿Qué clase usaremos para obtener información sobre servicios de telefonía?

- a) TelephonyManager.
- b) PhoneManager.
- c) TelephonySystemService.
- d) PhoneSystemService.

16. ¿Cómo instanciamos la clase TelephonyManager?

- a) new TelephonyManager().
- b) getSystemService(Context.TELEPHONY_SERVICE).
- c) TelephonyManager.getInstance().
- d) Es una clase con métodos que podemos usar de manera estática.

17. Si queremos que nuestra aplicación realice una acción cuando se reciba una llamada entrante ¿qué tipo de Listener deberemos definir?

- a) TelephonyListener.
- b) TelephonyStateListener.
- c) PhoneListener.
- d) PhoneStateListener.



18. ¿Qué estado deberemos observar para detectar que está entrando una llamada?

- a) PhoneManager.INCOMING_CALL.
- b) PhoneManager.CALL_STATE_RINGING.
- c) TelephonyManager.INCOMING_CALL.
- d) TelephonyManager.CALL_STATE_RINGING.

19. Para realizar una llamada, ¿con qué filtro deberemos crear un *intent*?

- a) Intent.ACTION_CALL.
- b) Intent.CALL.
- c) Intent.PHONE_CALL.
- d) Intent.PHONE.

20. Para acceder a los datos de la agenda ¿qué clase usaremos?

- a) ContactsContract.
- b) Contacts.
- c) Person.
- d) PhoneContacts.



SOLUCIONARIO

1.	b	2.	c	3.	a	4.	d	5.	a
6.	b	7.	a	8.	b	9.	a	10.	b
11.	a	12.	c	13.	a	14.	d	15.	a
16.	b	17.	d	18.	d	19.	a	20.	a



¿Decidido a ser el mejor? Pues entonces, serás un **P8.10**.

PROPUESTAS DE AMPLIACIÓN

- Se recomienda leer la documentación y profundizar en la clase ContactsContract ya que nos puede ser muy útil para nuestras aplicaciones.
 - <http://developer.android.com/resources/articles/contacts.html>.
- Un ejemplo de uso sobre la misma:
 - <http://developer.android.com/resources/samples/BusinessCard/index.html>.



BIBLIOGRAFÍA

- <http://developer.android.com/reference/android/telephony/package-summary.html>
- <http://developer.android.com/reference/android/provider/ContactsContract.html>.

PROGRAMACIÓN PARA ANDROID

10

Sensores



Escuela Universitaria
de Formación Abierta

master.D
GRUPO



ÍNDICE

MOTIVACIÓN.....	3
PROPÓSITOS	4
PREPARACIÓN PARA LA UNIDAD	5
1. SENSORMANAGER	7
2. GESTOS.....	13
CONCLUSIONES	19
RECAPITULACIÓN	20
AUTOCOMPROBACIÓN	21
SOLUCIONARIO.....	27
PROPUESTAS DE AMPLIACIÓN	28
BIBLIOGRAFÍA.....	29



MOTIVACIÓN



Dispondrás de las mejores herramientas para tu formación.

Los terminales y *tablets* en los que ejecutar nuestras aplicaciones suelen traer una serie de sensores que permiten mejorar la experiencia de usuario.

- Dependiendo del modelo y de parte de los sensores táctiles de la pantalla, podemos encontrarnos con sensores de proximidad, acelerómetros, brújulas, sensores de luz, sensores meteorológicos, etc.
- Como no podía ser de otra forma, el SDK de Android nos ofrece un API de acceso a sus funcionalidades, con lo que es muy sencillo hacer que nuestras aplicaciones interactúen con ellos.
- En este capítulo veremos la forma de hacerlo.

PROPOSITOS

- Conocer la clase SensorManager y las posibilidades que nos ofrece.
- Aprender a manejar la librería Gesture para que nuestra aplicación reconozca signos táctiles.



PREPARACIÓN PARA LA UNIDAD

- Como siempre necesitaremos tener el entorno de desarrollo configurado.
- Además, para probar los ejercicios es recomendable tener un terminal físico ya que, aunque se puede, probar aplicaciones de sensores en el emulador es más complicado (además de no transmitir la sensación real de uso).



1. SENORMANAGER

- El paquete android.hardware contiene las clases necesarias para acceder la información que nos facilitan los sensores.
- La clase que nos permite acceder a los sensores es SensorManager. Podremos obtener una instancia de ella llamando a *Context.getSystemService()* pasándole como parámetro la constante SENSOR_SERVICE.
- Cada tipo de sensor tiene asignada una constante en la clase Sensor. Las principales son:
 - TYPE_ACCELEROMETER: mide la aceleración en los tres ejes (x, y, z).
 - TYPE_GYROSCOPE: mide la rotación respecto a los ejes.
 - TYPE_LIGHT: responde a la intensidad de luz de entorno.
 - TYPE_MAGNETIC_FIELD: muestra la atracción magnética que hay en cada eje.
 - TYPE_ORIENTATION: devuelve la orientación actual del terminal respecto a los tres ejes medida en grados.
 - TYPE_PROXIMITY: devuelve la distancia entre el terminal y un objeto.
 - TYPE_TEMPERATURE: capta la temperatura ambiente en grados centígrados.

A pesar de que como se puede apreciar hay muchos sensores y que cada uno de ellos provee de información distinta. La clase SensorManager nos permite controlarlos de una forma homogénea.

Veamos un ejemplo: supongamos que queremos hacer una aplicación que obtenga datos del acelerómetro y los muestre en un TextView.

Lo primero que deberíamos hacer es obtener una instancia del SensorManager. Esto se suele hacer en el método *onCreate*.

```
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    acelerometro = (TextView) findViewById(R.id.acelerometro);
    mSensorManager = (SensorManager)
        getSystemService(SENSOR_SERVICE);
}
```

Obteniendo un SensorManager

- El siguiente paso sería crear un *Listener* que escuche los eventos generados por el sensor que nos interesa. Para ello debemos implementar el interfaz SensorEventListener.
- Una buena opción sería que nuestra Activity lo implementara:

```
public class MainActivity extends Activity implements
    SensorEventListener {
```

Activity que implementa un *listener* para sensores

- Esto nos obligará a desarrollar un método *onSensorChanged* que será el que recoja el evento y el método *onAccuracyChanged* que se invocará cuando cambie la precisión.

```
@Override
public void onSensorChanged(SensorEvent event) {
    synchronized (this) {
        String txt = "\n\nSensor: ";
        Log.d("sensor", event.sensor.getName());
        switch (event.sensor.getType()) {
            case Sensor.TYPE_ACCELEROMETER:
                txt += "acelerometro\n";
                txt += "\n x: " + event.values[0];
                txt += "\n y: " + event.values[1];
                txt += "\n z: " + event.values[2];
                acelerometro.setText(txt);
                break;
        }
    }
}
@Override
public void onAccuracyChanged(Sensor sensor, int accuracy) {
```

Método que será llamado cada vez que un sensor cambie

- Cada sensor devuelve su propia información. En este caso según hemos leído en la documentación nos vendrán tres valores que se corresponderán con la aceleración en cada uno de los ejes (x, y, z).



- Además como se puede apreciar hay un par de consideraciones que debemos tener en cuenta:
 - Nuestro método *onSensorChanged* puede ser llamado a causa de un evento generado por otro sensor, por lo que debemos asegurarnos de que ha sido el acelerómetro el que lo ha disparado.
 - Como los dispositivos tienen varios sensores e incluso pueden tener varios sensores del mismo tipo, debemos asegurarnos de que la parte crítica del código no se ejecute de forma simultánea, por eso es una buena idea encerrarla en un bloque “*synchronized*”.
- El siguiente paso sería enlazar nuestro *listener* (en este caso nuestra Activity que implementa la interfaz) con los eventos del sensor que queramos escuchar. Para ello deberemos registrarlo:

```
@Override  
protected void onResume() {  
    super.onResume();  
    mSensorManager.registerListener(  
        this,  
        mSensorManager.getDefaultSensor  
            (Sensor.TYPE_ACCELEROMETER),  
        SensorManager.SENSOR_DELAY_NORMAL  
    );  
}
```

Registrando el Listener

- En este caso lo vamos a hacer en el método *onResume*, para que cada vez que la Activity vuelva a primer plano volvamos a escuchar los sensores.
- Como primer parámetro hemos pasado el objeto *Listener* (en este caso la propia Activity), como segundo el sensor a escuchar (obtenido con la clase *getDefaultSensor* de *SensorManager*) y como tercer parámetro la velocidad de refresco a utilizar, tenemos tres posibilidades:
 - **SENSOR_DELAY_NORMAL**: tasa de refresco normal (válida para la mayoría de los casos).
 - **SENSOR_DELAY_GAME**: tasa de refresco alta (valida para los casos en los que necesitamos un refresco más frecuente, como por ejemplo los juegos).
 - **SENSOR_DELAY_FASTEST**: la mayor tasa de refresco (para aquellas situaciones en las que necesitemos una precisión extra).
- Finalmente, con el fin de no sobrecargar el terminal y ahorrar batería, sobrescribiremos el método *onStop* para que cuando la Activity pase a segundo plano, liberaremos el *SensorManager* quitando nuestro *listener*:

```

@Override
protected void onStop() {
    mSensorManager.unregisterListener(
        this,
        mSensorManager.
        getDefaultSensor(Sensor.TYPE_ACCELEROMETER)
    );
    super.onStop();
}

```

Liberando el *listener*antes de parar la Activity

Como hemos visto acceder a la información de los sensores es muy sencillo y no requiere que solicitemos ningún permiso especial en el AndroidManifest.

También hemos comentado que la interface de acceso común nos permite consultarlos de la misma forma. Por ejemplo si quisieramos acceder a la información de Orientación para hacer, por ejemplo, una brújula el código sería muy similar:

```

public class MainActivity extends Activity implements
SensorEventListener {
    private SensorManager mSensorManager;
    private TextView orientacion;
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        orientacion = (TextView)
            findViewById(R.id.orientacion);
        mSensorManager = (SensorManager)
            getSystemService(SENSOR_SERVICE);
    }
    @Override
    protected void onResume() {
        super.onResume();
        mSensorManager.registerListener
            (this, mSensorManager.getDefaultSensor
            (Sensor.TYPE_ORIENTATION),
            SensorManager.SENSOR_DELAY_NORMAL);
    }
    @Override
    protected void onStop() {
        mSensorManager.unregisterListener
            (this, mSensorManager.getDefaultSensor
            (Sensor.TYPE_ORIENTATION));
        super.onStop();
    }
    @Override
    public void onAccuracyChanged(Sensor sensor,
        int accuracy) { }
    @Override
    public void onSensorChanged(SensorEvent event) {
        String txt = "\n\nSensor: ";
        synchronized (this) {

```



```
        switch (event.sensor.getType()) {
            case Sensor.TYPE_ORIENTATION:
                txt += "orientation\n";
                txt += "\n azimut: " + getDireccion
                (event.values[0]);
                txt += "\n y: " + event.values[1] + "°";
                txt += "\n z: " + event.values[2] + "°";
                orientacion.setText(txt);
            break;
        }
    }

    private String getDireccion(float values) {
        String txtDirection = "";
        if (values < 22) txtDirection = "N";
        else if (values >= 22 && values < 67) txtDirection = "NE";
        else if (values >= 67 && values < 112) txtDirection = "E";
        else if (values >= 112 && values < 157) txtDirection = "";
        else if (values >= 157 && values < 202) txtDirection = "S";
        else if (values >= 202 && values < 247) txtDirection = "SO";
        else if (values >= 247 && values < 292) txtDirection = "O";
        else if (values >= 292 && values < 337) txtDirection = "NO";
        else if (values >= 337) txtDirection = "N";
        return txtDirection;
    }
}
```

Obviamente deberemos consultar la documentación para saber a qué se corresponderán los valores devueltos. En este caso será la posición en cada uno de los ejes medida en grados, por lo que deberemos hacer una pequeña transformación antes de mostrarla.

Pero, ¿cómo podemos saber de qué sensores dispone el terminal? Pues muy sencillo, basta con llamar al método *getSensorList* de la clase *SensorManager* pasando un parámetro por el que filtraremos el tipo de sensor a buscar. Este método nos devolverá un *array* de objetos tipo *Sensor* con una entrada por cada sensor encontrado:

```
public class ListaSensoresActivity extends ListActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.list);
        setTitle("Listado de sensores");
        SensorManager sManager = (SensorManager)
            getSystemService(SENSOR_SERVICE);
        List<Sensor> sensores =
            Manager.getSensorList(Sensor.TYPE_ALL);
        SensorAdapter adapter = new SensorAdapter(this,
            android.R.layout.simple_list_item_1, sensores);
        setListAdapter(adapter);
    }
    class SensorAdapter extends ArrayAdapter<Sensor> {
        private int tResId;
        public SensorAdapter(Context context,
            int textViewResourceId,
```

```
        List<Sensor> objects) {
    super(context, textViewResourceId, objects);
    this.tResId = textViewResourceId;
}

@Override
public View getView(int position, View view,
ViewGroup parent) {
    if (view == null) {
        view = getLayoutInflater().inflate
(tResId, null);
    }
    Sensor s = getItem(position);
    TextView text = (TextView)
view.findViewById(android.R.id.text1);
    text.setText(s.getName());
    return convertView;
}
}
}
```

Como se puede comprobar el código anterior mostrará un listado con todos los sensores disponibles.

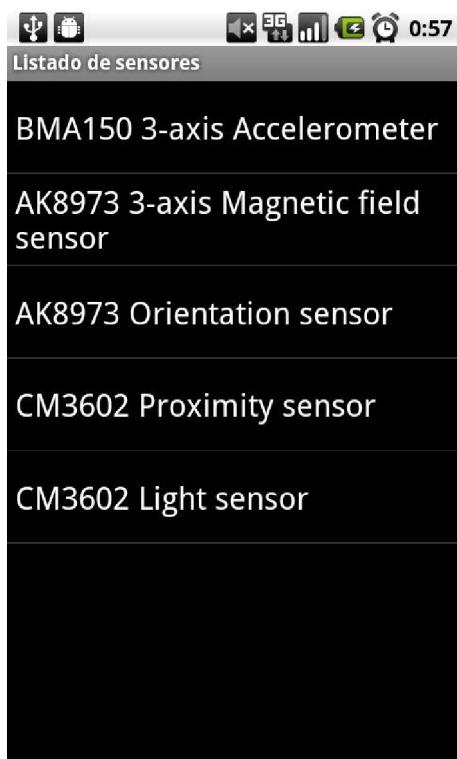


Figura 1. Listado de sensores.



2. GESTOS

- La mayoría (por no decir todos) terminales Android tienen pantallas táctiles que responden a las pulsaciones sobre ellas.
- Hasta ahora hemos utilizado esta funcionalidad de manera indirecta, programando *listeners* que se disparan al “clickear” sobre un objeto View, o colocando *scrolls*, por poner unos ejemplos.
- En este capítulo vamos a aprender a sacar más partido a las capacidades táctiles del terminal enseñándole a reconocer gestos, lo que puede mejorar sensiblemente la usabilidad de nuestras aplicaciones.
- Aunque el soporte básico para pantallas táctiles está incluido en el SDK desde las primeras versiones, no es hasta la versión 1.6 cuando se incorpora la clase android.gesture, una clase de utilidad que nos facilita mucho la vida.
- Hay que tener en cuenta que lo descrito en este apartado solo va a ser compatible con el API 4 o posteriores.
- La clase Gesture se basa en reconocer una serie de gestos predefinidos guardados en un archivo biblioteca. Así pues, lo primero que debemos hacer es crearnos nuestra propia biblioteca de gestos. Para ello la mejor opción es usar el programa “Gestures Builder” que nos viene preinstalado en los emuladores. Necesitaremos, eso sí, usar un emulador con soporte de tarjeta SD, ya que el fichero de gestos se intentará guardar en ella.
- El código fuente de esta aplicación viene en los ejemplos que acompañan al SDK, así que otra opción es usarlo para crear un proyecto nuevo e instalarlo en un terminal físico. Esto lo podemos hacer desde el diálogo de nuevo proyecto: Seleccionaremos primero la plataforma Android (superior a 1.6) y luego la opción “Crear proyecto a partir de un ejemplo”.

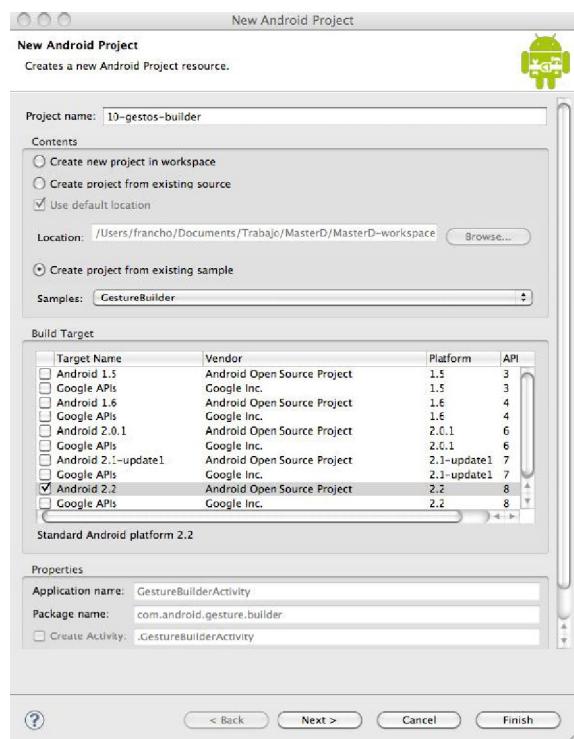


Figura 2. Creando un proyecto a partir de un ejemplo

Una vez lanzada la aplicación se trata de dibujar los gestos predefinidos en pantalla y etiquetarlos con un nombre (que será el que usaremos luego para identificarlos).



Figura 3. Nuestra biblioteca de gestos



Podemos crear varios gestos con la misma etiqueta y de esta forma tendremos cubierta distintas formas de dibujar el mismo gesto (por ejemplo, si el usuario es zurdo es posible que lo empiece por el lado contrario).

Una vez terminados de editar todos los gestos, a través de la pestaña DDMS de eclipse (o de un cable USB si estamos usando un terminal) podemos sacar el fichero generado que se encuentra en /mnt/sdcard/gestures que colocaremos en el directorio ./res/raw de nuestro proyecto.

El siguiente paso es colocar nuestro “capturador de gestos”. Para ello, debemos poner un objeto android.gesture.GestureOverlayView. Hay que tener presente que, como este no es un objeto de espacio de nombres por defecto, debemos nombrarlo con toda su ruta.

Definiendo atributos XML en el GestureOverlayView podemos modificar su comportamiento (por ejemplo mostrar/ocultar la línea dibujada por el usuario, colores, duración, etc.

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="vertical"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        >
    <android.gesture.GestureOverlayView
        android:id="@+id/gestures"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:layout_weight="1.0" />
    <TextView
        android:id="@+id/NombreGesto"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:text="¿?"
        android:gravity="center_vertical|center_horizontal"
        android:textAppearance="?android:attr/textAppearanceLarge"
        />
</RelativeLayout>
```

Nuestro layout con soporte para gestos

Ahora vamos a darle la funcionalidad. Queremos que cuando el usuario dibuje una figura que esté en nuestra biblioteca de gestos, su nombre aparezca en pantalla (en el TextView).

Para asignar comporta a cada gesto debemos implementar un SensorEventListener y asignarlo al objeto que hemos colocado en nuestro *layout*.

Nosotros vamos a implementar el listener en la propia Activity así que la definiremos de la siguiente forma:

```
public class GestosActivity extends Activity implements
OnGesturePerformedListener {
```

```

@Override
public void onGesturePerformed
    (GestureOverlayView gView, Gesture gesture) {
// TODO
}
}

Activity que funciona como listener

```

En el método *onCreate*, aparte de las acciones típicas (cargar el *layout*, etc.) cargaremos la librería de gestos para su posterior uso:

```

@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    gestureView = (GestureOverlayView)
        findViewById(R.id.gestures);
    nombreGesto = (TextView) findViewById-
       ById(R.id.nombreGesto);
    gestLib = GestureLibraries.fromRawResource
        (this, R.raw.gestures);
    if (!gestLib.load()) {
        Toast.makeText(this,
            "Error al cargar gestos predefinidos",
            Toast.LENGTH_LONG
        ).show();
        finish();
    }
}

```

Ejemplo de carga de librería de gestos

El siguiente paso es asignar el *listener* al objeto *GestureOverlayView*. Como este tipo de *listeners* consumen bastante batería es buena idea asignarlos cuando la actividad viene a primer plano y liberarlos cuando pasa a un segundo:

```

@Override
protected void onResume() {
    super.onResume();
    gestureView.addOnGesturePerformedListener(this);
}
@Override
protected void onStop() {
    gestureView.removeOnGesturePerformedListener(this);
    super.onStop();
}

```

Asignando y liberando el listener

Finalmente, solo nos queda escribir el código del método *onGesturePerformed* que será llamado cuando el gesto se haya dibujado. En nuestro caso vamos a mostrar el nombre de la etiqueta en un campo *TextView*.

```

@Override
public void onGesturePerformed(GestureOverlayView

```



```
gOverlayView, Gesture gesture) {  
    ArrayList<Prediction> predicciones =  
        gestLib.recognize(gesture);  
    if (predicciones.size() > 0) {  
        Prediction prediccion = predicciones.get(0);  
        if (prediccion.score > 1.0) {  
            nombreGesto.setText(prediccion.name);  
        }  
        return;  
    } else {  
        nombreGesto.setText("ξ" + prediccion.name + "?");  
    }  
} else {  
    nombreGesto.setText("gesto desconocido");  
}  
}
```

Dando funcionalidad a nuestros gestos

Listo, ya tenemos nuestra aplicación preparada para funcionar con gestos. Es una buena idea para estos casos dar a probar a diferentes personas para observar cómo escriben los gestos (por ejemplo, no todas las personas escriben las letras empezando por el mismo punto). De esta forma podremos añadir los alias que sean necesarios a nuestra biblioteca de gestos antes de publicarla.

CONCLUSIONES



Dominio de la materia y entrenamiento de las habilidades:
Master.D es la herramienta de tu **triunfo profesional**.

- Los dispositivos Android suelen traer incorporados una serie de sensores que nos permiten interactuar con el usuario de formas diferentes a como estamos acostumbrados.
- Podemos, por ejemplo, descolgar automáticamente el teléfono al acercárnoslo a la cara, apagar una alarma agitando el terminal, cambiar el fondo dependiendo de la orientación geográfica, controlar juegos moviendo el terminal, escribir textos con el dedo, etc.
- En definitiva, un montón de formas diferentes de mejorar la experiencia de usuario, para las que Android nos facilita una serie de APIs y herramientas potentes y fáciles de usar.

RECAPITULACIÓN

- A pesar de que los sensores que incorporan nuestros dispositivos son de lo más variados, Android nos ofrece un API de acceso único que nos permite acceder a toda su información de forma uniforme gracias a la clase SensorManager.
- SensorEventListener nos permite implementar *Listeners* especializados que serán invocados cuando se produzca un evento del sensor o sensores para el que lo registramos.
- Hay que recordar que es una buena idea liberar los *listeners* de sensores cuando nuestra Activity vaya a pasar a un segundo plano para evitar un consumo excesivo de batería.
- Desde la versión 1.6 de Android disponemos en el SDK de una clase (Gesture) que nos permite utilizar librerías de gestos en nuestras aplicaciones.
- Para ello, primero deberemos guardar en un archivo binario los gestos que reconocerá nuestra aplicación. Para esto, podemos usar el ejemplo que trae el SDK llamado “Gesture Builder”. Luego deberemos utilizar un objeto GestureOverlayView al que asociaremos un *listener* de tipo SensorEventListener que será el encargado de acceder a la librería e identificar el gesto cuando este sea dibujado en la pantalla.



AUTOCOMPROBACIÓN

1. ¿Que clase usamos para acceder a los sensores?

- a) Sensors.
- b) SensorManager.
- c) PhoneSensors.
- d) PhoneManager.

2. ¿Cómo consigo una instancia de SensorManager?

- a) new X.
- b) Es una clase estática que no necesita instancia.
- c) Context.getSystemService(SENSOR_SERVICE).
- d) Cualquiera de las anteriores.

3. Si quiero acceder a los datos de rotación ¿qué constante usaré?

- a) TYPE_ACCELEROMETER.
- b) TYPE_GYROSCOPE.
- c) TYPE_ORIENTATION.
- d) TYPE_MAGNETIC_FIELD.

4. **¿Qué constante debe usarse para obtener un sensor que detecte movimientos del móvil?**
 - a) TYPE_ACCELEROMETER.
 - b) TYPE_GYROSCOPE.
 - c) TYPE_ORIENTATION.
 - d) TYPE_MAGNETIC_FIELD.
5. **¿Qué constante debe usarse para obtener un sensor que sirva de brújula?**
 - a) TYPE_ACCELEROMETER.
 - b) TYPE_GYROSCOPE.
 - c) TYPE_ORIENTATION.
 - d) TYPE_MAGNETIC_FIELD.
6. **Si quisiéramos hacer que la pantalla se apagara a acercarnos el móvil a la cara ¿qué sensor usaríamos?**
 - a) TYPE_MAGNETIC_FIELD.
 - b) TYPE_PROXIMITY.
 - c) TYPE_LIGHT.
 - d) TYPE_TEMPERATURE.
7. **¿Qué tipo de listener deberemos definir para interactuar con sensores?**
 - a) SensorListener.
 - b) SensorsListener.
 - c) SensorManagerListener.
 - d) SensorEventListener.
8. **¿Qué método/s nos obliga a implementar la clase SensorEventListener?**
 - a) onSensorChanged.
 - b) onAccuracyChanged.
 - c) onSensorChanged y onAccuracyChanged.
 - d) onSensorChanged u onAccuracyChanged.



9. ¿Con qué método asociamos nuestro SensorEventListener con el SensorManager?
 - a) registerListener.
 - b) addListener.
 - c) setListener.
 - d) putListener.

10. ¿Qué método usaremos cuando queramos dejar de escuchar los sensores?
 - a) unregisterListener.
 - b) removeListener.
 - c) unsetListener.
 - d) deleteListener.

11. ¿Qué constante define la velocidad por defecto de refresco de sensores?
 - a) SENSOR_DELAY.
 - b) SENSOR_DELAY_NORMAL.
 - c) SENSOR_DELAY_DEFAULT.
 - d) Ninguna de las anteriores.

12. Si estoy programando los controles de un juego basados en sensores, ¿que velocidad de refresco usaré?
 - a) SENSOR_DELAY_FASTER.
 - b) SENSOR_DELAY_GAME.
 - c) SENSOR_DELAY_NORMAL.
 - d) SENSOR_DELAY_FAST.

13. Si necesito saber con la máxima precisión posible los datos de sensores, ¿qué velocidad de refresco usaré?
 - a) SENSOR_DELAY_FASTEST.
 - b) SENSOR_DELAY_FASTER.
 - c) SENSOR_DELAY_SUPERFAST
 - d) SENSOR_DELAY_MINIMUM

14. ¿Qué tipo de objeto recibiremos como parámetro en el método *onSensorChanged?*
- a) Event.
 - b) SensorEvent.
 - c) Sensor.
 - d) Float[].
15. ¿Qué clase nos da soporte para identificar gestos en nuestro terminal?
- a) Gestures.
 - b) SensorGestures.
 - c) Gesture.
 - d) SensorGesture.
16. ¿A partir de qué versión está disponible la clase que proporciona reconocimiento de gestos?
- a) Android 1.5.
 - b) Android 1.6.
 - c) Android 2.1.
 - d) Android 2.2.
17. ¿Cómo se llama el objeto View que debemos colocar en nuestros layout para detectar gestos?
- a) GestureView.
 - b) GesturesView.
 - c) GestView.
 - d) GestureOverlayView.
18. La biblioteca de gestos que generemos/usemos, ¿dónde deberemos colocarla?
- a) Es un binario, por lo que deberá ir en la carpeta ./res/raw.
 - b) Es un xml que debe ir en ./res/values.
 - c) Es una definición de layout que pondremos en ./res/layout.
 - d) Son plantillas de tipo drawable, por lo que irán en ./res/drawable.



19. ¿Qué método invocaremos para cargar la biblioteca de gestos?

- a) GestureLibraries.fromRawResource.
- b) GestureLibraries.loadResource.
- c) new GestureLibraries(nombreRecurso).
- d) new GestureLibraries(idRecurso).

20. ¿Qué listener definiremos para responder a eventos relacionados con gestos?

- a) GestureListener.
- b) OnGestureListener.
- c) OnGestureLibrariesListener.
- d) OnGesturePerformedListener.



SOLUCIONARIO

1.	b	2.	c	3.	b	4.	a	5.	c
6.	b	7.	d	8.	c	9.	a	10.	a
11.	b	12.	b	13.	a	14.	b	15.	c
16.	b	17.	d	18.	a	19.	a	20.	d



Destaca entre la multitud; ser un **P8.10** te puede ayudar a llegar muy lejos.

PROPUESTAS DE AMPLIACIÓN

- Una aplicación de código abierto que hace uso de los sensores para simular un Tricorder¹:
- [http://code.google.com/p/moonblink/source/browse/trunk/Tricorder/.](http://code.google.com/p/moonblink/source/browse/trunk/Tricorder/)

¹ Dispositivo ficticio de bolsillo usado en el universo Star Trek para obtener lecturas de diverso tipo.



BIBLIOGRAFÍA

- [http://developer.android.com/resources/articles/gestures.html.](http://developer.android.com/resources/articles/gestures.html)

PROGRAMACIÓN PARA ANDROID

11

Multimedia



Escuela Universitaria
de Formación Abierta

master.D
GRUPO



ÍNDICE

MOTIVACIÓN.....	3
PROPÓSITOS	4
PREPARACIÓN PARA LA UNIDAD	5
1. MULTIMEDIA EN ANDROID	7
2. REPRODUCIR AUDIO.....	9
3. REPRODUCIR VÍDEO	13
4. GRABAR SONIDO.....	15
5. GRABAR VÍDEO	18
6. APROVECHAR LAS APLICACIONES DEL SISTEMA.....	23
CONCLUSIONES	25
RECAPITULACIÓN	26
AUTOCOMPROBACIÓN	27
SOLUCIONARIO.....	33
PROPUESTAS DE AMPLIACIÓN	34
BIBLIOGRAFÍA.....	35



MOTIVACIÓN



Dirígete hacia tu meta sin vacilar; nosotros te ayudamos a llegar hasta allí.

Hoy en día cualquier aplicación que se precie hace uso de recursos multimedia, ya sea para colocar efectos de sonido, incluir imágenes o vídeos, etc.

Nuestros dispositivos Android, además, suelen incorporar por lo menos una cámara y un micrófono, por lo que es interesante aprender a sacarles partido.

Para ello, en este capítulo vamos a perfilar las clases y herramientas que el SDK de Android nos facilita a los desarrolladores para estos menesteres.

PROPOSITOS

En esta unidad aprenderemos a manejar las clases del paquete android.media:

- Reproducción de sonidos con la clase MediaPlayer.
- Reproducción de vídeos con VideoView y MediaController.
- Grabación de sonidos con MediaRecorder.
- Grabación de vídeos con MediaRecorder.
- Uso de las aplicación de cámara del sistema.



PREPARACIÓN PARA LA UNIDAD

Para esta unidad, además de un entorno de trabajo (Eclipse con el SDK de Android) sería interesante disponer de un terminal físico donde poder probar las aplicaciones, ya que algunas de las funcionalidades necesarias no están soportadas por los emuladores.



1. MULTIMEDIA EN ANDROID

Aparte de los mecanismos que ya hemos visto en las lecciones anteriores que nos permiten reutilizar *intents* de otras aplicaciones (en este caso para por ejemplo reproducir un sonido), el SDK de Android nos facilita una serie de clases para el manejo de archivos multimedia agrupadas en el paquete android.media.

Las clases principales de este paquete son:

- MediaPlayer que es la responsable de reproducir archivos de audio y vídeo.
- MediaRecorder, que nos permite guardar archivos sonoros o vídeos.
- JetPlayer, que permite acceder a música interactiva que usa el formato de SONiVOX.

Estas clases pueden acceder a multimedia situados en:

- Recursos incorporados a nuestro proyecto (normalmente colocados en la carpeta “res/raw”).
- Archivos locales almacenados en la tarjeta SD del terminal.
- Archivos remotos a los que accedemos a través de una URL.

A la hora de incorporar archivos sonoros a nuestros proyectos debemos ser muy cautos, ya que estos engordarán considerablemente el tamaño final de nuestro fichero .apk.

En cuanto a los formatos, el SDK, por defecto, incorpora soporte para los siguientes formatos¹:

Tipo	Formato	Encoder	Decoder	Detalles	Tipos de ficheros soportados
Audio	AAC LC/LTP		X	Contenido Mono/Stereo cualquier combinación de velocidades de bits estándar de hasta 160 kbps y velocidades de muestreo de 8 a 48 kHz	3GPP (.3gp) and MPEG-4 (.mp4, .m4a). Sin soporte para raw AAC (.aac)
	HE-AACv1 (AAC+)		X		
	HE-AACv2 (mejorado AAC+)		X		
	AMR-NB	X	X	4,75 to 12,2 kbps sampled @ 8kHz	3GPP (.3gp)
	AMR-WB		X	9 tasas de 6,60 kbit/s a 2,85 kbit/s sampled @ 16kHz	3GPP (.3gp)
	MP3		X	Mono/Stereo 8-320Kbps contantes (CBR) o variables (VBR)	MP3 (.mp3)
	MIDI		X	MIDI Tipos 0 y 1. DLS Versión 1 y 2. XMF y móvil XMF. Soporte para formatos de tonos telefónicos RTTTL/RTX, OTA, e iMelody	Tipo 0 y 1 (.mid, .xmf, .mxmf). también RTTTL/RTX (.rtttl, .rtx), OTA (.ota), y iMelody (.imy)
	Ogg Vorbis		X		Ogg (.ogg)
Imagen	PCM/WAVE		X	8- and 16-bit lineales PCM (tasas limitadas por hardware)	WAVE (.wav)
	JPEG	X	X	Base + progresivo	JPEG (.jpg)
	GIF		X		GIF (.gif)
	PNG		X		PNG (.png)
Vídeo	BMP		X		BMP (.bmp)
	H.263	X	X		3GPP (.3gp) y MPEG-4 (.mp4)
	H.264 AVC		X		3GPP (.3gp) y MPEG-4 (.mp4)
	MPEG-4 SP		X		3GPP (.3gp)

¹Fuente: <http://developer.android.com/guide/appendix/media-formats.html>.



2. REPRODUCIR AUDIO

Como ya hemos mencionado antes, para reproducir sonidos disponemos de dos clases: MediaPlayer y JetPlayer.

MediaPlayer es la más usada y la veremos enseguida con detalle, pero antes daremos unas pinceladas de uso de JetPlayer.

Jet es un motor de sonido que permite reproducir sonidos de forma interactiva (activando/desactivando canales, saltando a un determinado punto, etc.), por poner un ejemplo, sería la librería que utilizaríamos si quisieramos programar un “Guitar Hero”.

El SDK incorpora un creador de archivos .jet que nos permite componer música (o efectos especiales) en este formato.

Si se quiere profundizar en este tema se recomienda estudiar el ejemplo incluido con el SDK llamado Jetboy², ya que hace uso de este sistema para añadir efectos sonoros al juego.

Fuera del ámbito de los juegos, lo más normal es reproducir sonidos usando la clase MediaPlayer.

MediaPlayer usa un sistema asíncrono basado en eventos para controlar la reproducción. Mediante llamadas a los métodos de MediaPlayer podemos cambiar de un estado a otro (por ejemplo de pausa a reproducción). El siguiente diagrama muestra con detalle todos los estados posibles:

Como se puede apreciar, lo primero es cargar y preparar el MediaPlayer, una vez está listo podemos invocar al método *start* para comenzar a reproducir el sonido.

² <http://developer.android.com/resources/samples/JetBoy/index.html>.

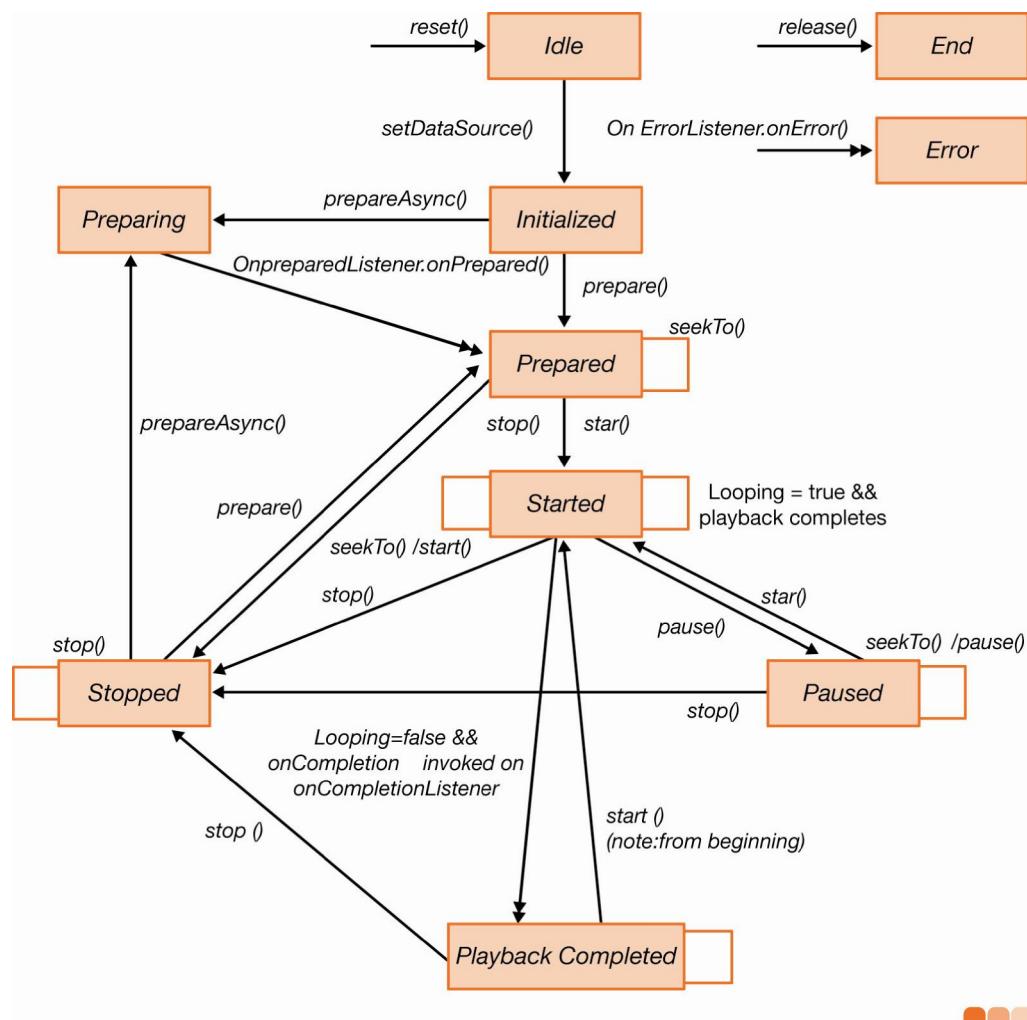


Figura 1. Estados de MediaPlayer según la documentación de Android.com



De ahí podemos pasar a estado pausado (método `pause`) o parado (método `stop`) y volver a la reproducción de nuevo usando `seekTo` para posicionarnos en un milisecondo determinado.

Finalmente, cuando hemos terminado de usar el MediaPlayer debemos liberarlo usando su método `release`.

Veamos un ejemplo práctico de cómo reproducir un archivo de audio colocado en una web. En el *layout* habremos definido tres botones que nos servirán para manejar el sonido: *play*, *pause* y *stop*.



En nuestra Activity usaremos un MediaPlayer como propiedad a la que mandaremos los cambios de estado cuando se pulsen los botones.

```
public class PlaySoundActiviy extends Activity {
    private String URL SONIDO =
        "http://www.soundjay.com/human/sounds/applause-3.mp3";
    private MediaPlayer mediaPlayer;
    private int posicion;
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.player);
        Button play = (Button) findViewById(R.id.play);
        play.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                mediaPlay();
            }
        });
        Button pause = (Button) findViewById(R.id.pause);
        pause.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                mediaPausa();
            }
        });
        Button stop = (Button) findViewById(R.id.stop);
        stop.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                mediaStop();
            }
        });
    }
}
```

Veamos cada uno de los métodos referentes al manejo de sonido. MediaPlay deberá parar la reproducción anterior (si la hay), crear una nueva instancia de MediaPlayer, cargar el sonido desde la URL y cuando esté todo listo comenzar a reproducirlo:

```
public void mediaPlay() {
    mediaStop();
    mediaPlayer = new MediaPlayer();
    (mediaPlayer == null) {
        Toast.makeText(PlaySoundActiviy.this,
        "Error al cargar el MediaPlayer",
        Toast.LENGTH_LONG).show();
        finish();
    }
    try {
        mediaPlayer.setDataSource(URL SONIDO);
        mediaPlayer.prepare();
        mediaPlayer.start();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Si el terminal no tiene soporte multimedia o todos los recursos están ocupados, al intentar crear el objeto MediaPlayer puede ser *null*. Por eso, es una buena idea comprobar siempre antes de empezara a usarlo para evitar errores no deseados.

También es una buena idea tratar de capturar las posibles Excepciones a la hora de cargar y preparar el archivo para evitar que el programa termine la ejecución por un fallo de sonido.

```
public void mediaStop() {
    if(mediaPlayer != null) {
        try {
            mediaPlayer.release();
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

Al parar la ejecución liberaremos el MediaPlayer para no ocupar recursos innecesarios.

```
public void mediaPausa() {
    if(mediaPlayer == null) {
        return;
    }
    if(mediaPlayer.isPlaying()) {
        posicion = mediaPlayer.getCurrentPosition();
        mediaPlayer.pause();
    } else {
        mediaPlayer.seekTo(posicion);
        mediaPlayer.start();
    }
}
```

En cuanto a la pausa, si el archivo se está reproduciendo, guardaremos su posición actual en una variable antes de pararlo, de esta forma al pulsar de nuevo el botón, nos colocaremos de nuevo en ella y reanudaremos la reproducción:

Dependiendo de la aplicación será interesante también pausar o parar el sonido dependiendo del estado de la Activity. Por ejemplo, si lo que estamos reproduciendo es una música de fondo, lo normal sería que si la Activity va a un segundo plano, la música parara, y cuando la volviera a primer plano, se reanudara.

En cualquier caso, lo que sí que suele ser una buena idea siempre, liberar el MediaPlayer cuando la Activity se destruye (*onDestroy*).



3. REPRODUCIR VÍDEO

De igual forma que hemos reproducido un sonido, podríamos reproducir un vídeo usando la clase MediaPlayer. Sin embargo, este sistema tiene el inconveniente de que nos obliga a programar los botones típicos de control de cualquier reproductor (*play*, *pause*, *stop*, avanzar, retroceder...).

Como esta es una tarea típica y rutinaria, para facilitarnos las cosas el SDK de Android incorpora una clase que se encarga de esto: MediaController.

Se trata de un objeto View que debemos instanciar mediante programación y que muestra una ventana flotante con estos controles.

La forma de utilizarla sería la siguiente:

Lo primero que debemos hacer es definir un *Layout* con un objeto de tipo VideoView que será donde se muestre la película.

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

Luego mediante programación creamos el objeto MediaController y lo asociamos a la vista donde debemos mostrar la película:

```
videoView.setMediaController(controlVideo);
VideoView videoView = (VideoView) findViewById(R.id.zonaVideo);
MediaController controlVideo =
new MediaController(PlayVideoActivity.this);
controlVideo.setAnchorView(videoView);
```

De igual forma debemos asignar el control a la vista de vídeo.

Ya solo nos queda cargar el vídeo y empezar la reproducción.

```
videoView.setVideoURI(Uri.parse(URL_VIDEO));  
videoView.start();
```

Como se puede apreciar, cuando empieza a ver el vídeo, los controles son visibles y pasados unos segundos se ocultan automáticamente, si pulsamos sobre el vídeo los controles vuelven a aparecer.



4. GRABAR SONIDO

Para la captura de audio y vídeo disponemos de la clase MediaRecorder (también incluida) en el paquete android.media.

El control de esta clase, al igual que MediaPlayer, está basado en una máquina de estados que podremos ir cambiando llamando a sus correspondientes métodos.

Los pasos a seguir para grabar un archivo de audio o vídeo son:

- Crear una nueva instancia de MediaRecorder.
- Definir la fuente de sonido.
- Definir el fichero de salida.
- Definir el formato de salida.
- Definir el *codec* de salida.
- Invocar al método *prepare*.
- Llamar a *start* para empezar a grabar.
- Llamar a *stop* para terminar de grabar
- Una vez terminado, si vamos a grabar un nuevo archivo utilizaremos *reset*.
- Finalmente, cuando no vayamos a usar más “la grabadora”, invocaremos al método *release* para liberar recursos.

El siguiente diagrama muestra todos los estados posibles y sus transiciones:

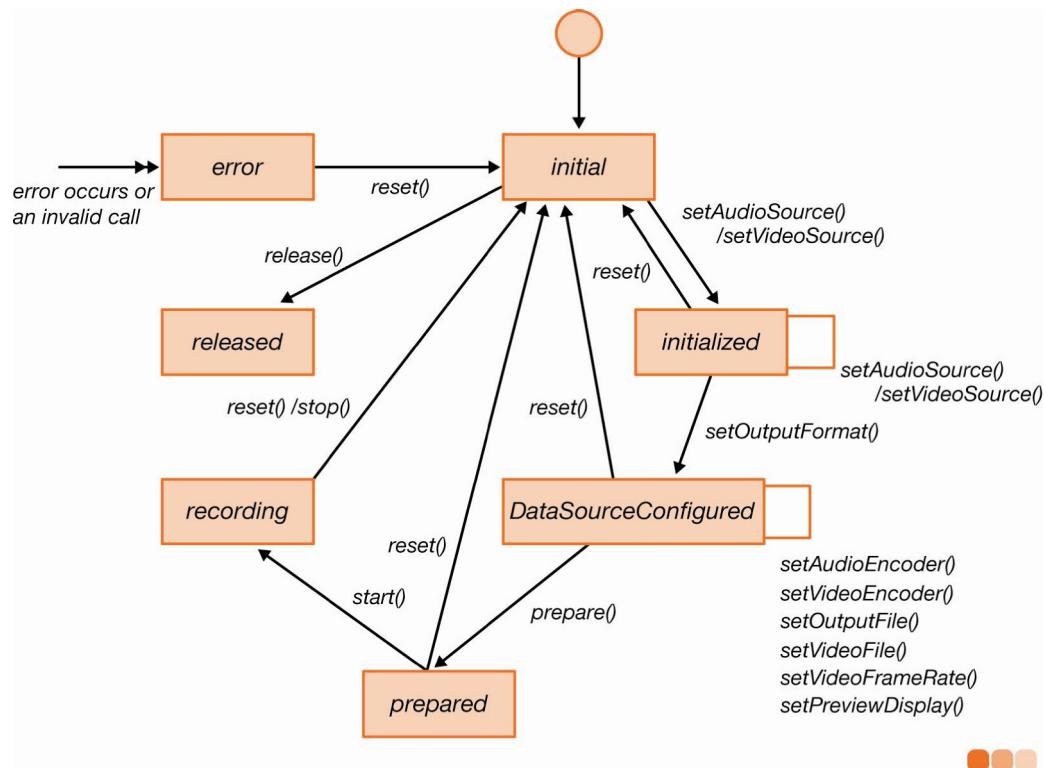


Figura 2. Estados de MediaRecorder según la documentación de android.com

Veamos cómo sería esto en código:

```
MediaRecorder grabadora = new MediaRecorder();
```

Lo primero sería obtener un objeto MediaRecorder, para ello crearemos uno nuevo usando `new`.

El siguiente paso es definir la ruta en la que queremos guardar el archivo de sonido. Podemos hacerlo directamente (como en este ejemplo) o usar clases como ContentResolver³ para integrarlo mejor con el sistema.

```
grabadora.setOutputFile("/sdcard/test.3gp");
```

Luego definimos la fuente de la que vamos a capturar el sonido, en este caso el micrófono.

```
grabadora.set AudioSource(MediaRecorder.AudioSource.MIC);
```

³ ContentResolver es una clase que permite a las aplicaciones acceder al modelo de contenido, es decir, con ella podríamos hacer que nuestro archivo de sonido grabado quedaría integrado en la biblioteca de medios que usa el reproductor multimedia.



Luego definiremos el formato de salida y el *codec* a usar:

```
grabadora.setOutputFormat(MediaRecorder.OutputFormat.THREE_GPP);  
grabadora.setAudioEncoder(MediaRecorder.AudioEncoder.AMR_NB);
```

Una vez hemos configurado todos los parámetros, deberemos cambiar a estado “preparado” y empezaremos a grabar llamando a *start*.

```
grabadora.prepare();  
grabadora.start();
```

Finalmente, cuando queramos parar de grabar, invocaremos a *stop* y, como no vamos a grabar nada más, liberaremos los recursos usados por la grabadora.

```
grabadora.stop();  
grabadora.release();
```

Para que esta aplicación funcione, deberemos declarar en el AndroidManifest el uso del permiso android.permission.RECORD_AUDIO.

5. GRABAR VÍDEO

Ya sabemos que para grabar vídeo podemos usar MediaRecorder, pero está claro que deberemos hacer alguna modificación para poder previsualizar en la pantalla lo que estamos grabando.

Esto se consigue colocando un objeto SurfaceView sobre el que mostraremos lo que está captando la cámara.

SurfaceView es un objeto que nos provee de una superficie sobre la que “dibujar” contiene un surfaceHolder que es el contenedor que da acceso a la superficie y con el que podemos interactuar a través de una clase *callback* (SurfaceHolder.Callback).

Así que lo primero que tendremos que hacer es definir un objeto SurfaceView en nuestro *layout*:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="horizontal"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:gravity="center_vertical">
    <ToggleButton android:id="@+id/rec"
        android:layout_width="180dp"
        android:layout_height="wrap_content"
        android:textOn="Grabando"
        android:textOff="Grabar" />
    <SurfaceView android:id="@+id/zonaVideo"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
</LinearLayout>
```

Hemos colocado también un botón de tipo “ToggleButton” para empezar/parar la grabación.

El siguiente paso será enlazar la cámara con nuestra superficie de dibujo. Para ello, hemos mencionado que debemos implementar SurfaceHolder.Callback. En



nuestro caso, vamos a integrar esta funcionalidad en nuestra Activity de grabación y además definiremos tres propiedades (que usaremos luego) para almacenar la cámara, la superficie y el MediaRecorder:

```
public class RecordVideoActivity extends Activity implements
    SurfaceHolder.Callback {
    private Camera cam;
    private SurfaceView camView;
    private SurfaceHolder camHolder;
    private MediaRecorder recorder;
    @Override
    public void surfaceChanged(SurfaceHolder arg0,
        int arg1, int arg2, int arg3) {
        // TODO Auto-generated method stub
    }
    @Override
    public void surfaceCreated(SurfaceHolder arg0) {
        // TODO Auto-generated method stub
    }
    @Override
    public void surfaceDestroyed(SurfaceHolder arg0) {
        // TODO Auto-generated method stub
    }
}
```

Como vemos, este interface nos obliga a desarrollar tres métodos que serán llamados al crear la superficie, al destruirla y cuando la superficie cambie (cambio de orientación, nuevas dimensiones, etc.).

Al crear la superficie cargaremos la cámara en un objeto, en el caso de que algo falle, está claro que no podemos continuar (no podemos grabar vídeo sin cámara).

```
public void surfaceCreated(SurfaceHolder holder) {
    try {
        cam = Camera.open();
    } catch (Exception e) {
        e.printStackTrace();
        Toast.makeText(getApplicationContext(), "Camara no
disponible!",
        Toast.LENGTH_LONG).show();
        finish();
    }
}
```

De igual forma cuando la superficie se destruya, liberaremos los recursos relacionados (cámara y MediaRecorder).

```
@Override
public void surfaceDestroyed(SurfaceHolder holder) {
    cam.stopPreview();
    cam.release();
    cam = null;
    if (recorder != null) {
        recorder.release();
        recorder = null;
    }
}
```

Cuando la superficie cambie de tamaño deberemos parar la previsualización y volverla a arrancar para que esta se adapte al nuevo tamaño.

```
public void surfaceChanged(SurfaceHolder holder,
    int format, int width,
    int height) {
    (previewRunning) {
        cam.stopPreview();
    }
    Camera.Parameters camParams = cam.getParameters();
    camParams.setPreviewFormat(PixelFormat.JPEG);
    cam.setParameters(camParams);
    try {
        cam.setPreviewDisplay(holder);
        cam.startPreview();
        previewRunning = true;
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Ya tenemos nuestro interface implementado. Es el momento de asignarlo al objeto que hemos creado en el layout.

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.video_rec);
    camView = (SurfaceView) findViewById(R.id.zonaVideo);
    camHolder = camView.getHolder();
    camHolder.addCallback(this);
    camHolder.setType
        (SurfaceHolder.SURFACE_TYPE_PUSH_BUFFERS);
}
```

Vemos que además de asignar el *callback* al contenedor, también le hemos asignado el tipo a SURFACE_TYPE_PUSH_BUFFERS para que sea la cámara la que gestione los *bufferes* de la superficie.

Bien, llega el momento de dar funcionalidad a nuestro botón. Según su estado empezaremos o pararemos de grabar:

```
ToggleButton btn = (ToggleButton) findViewById(R.id.rec);
btn.setOnCheckedChangeListener(new OnCheckedChangeListener() {
    @Override
    public void onCheckedChanged(CompoundButton buttonView,
        boolean isChecked) {
        try {
            if (isChecked) {
                grabar();
            } else {
                pararGrabar();
            }
        } catch (IllegalStateException e) {
            e.printStackTrace();
        }
    }
});
```



El método `grabar()` es el encargado de crear el objeto `MediaRecorder`, inicializarlo e iniciar la grabación:

```
protected void grabar() throws IllegalStateException {
    cam.unlock();
    recorder = new MediaRecorder();
    recorder.setCamera(cam);
    recorder.setAudioSource
        (MediaRecorder.AudioSource.MIC);
    recorder.setVideoSource
        (MediaRecorder.VideoSource.DEFAULT);
    recorder.setOutputFormat
        (MediaRecorder.OutputFormat.MPEG_4);
    recorder.setAudioEncoder(
        MediaRecorder.AudioEncoder.AMR_NB);
    recorder.setVideoEncoder
        (MediaRecorder.VideoEncoder.MPEG_4_SP);
    String filename = Environ-
ment.getExternalStorageDirectory().getAbsolutePath()
+ "/miVideo.3gp";
    recorder.setOutputFile(filename);
    recorder.setMaxDuration(2000);
    recorder.setMaxFileSize(10240);
    recorder.setPreviewDisplay(camHolder.getSurface());
try {
    recorder.prepare();
    recorder.start();
} catch (IOException e) {
    e.printStackTrace();
}
}
```

Como se puede apreciar el código usado en este método es muy similar al que hemos utilizado para grabar sonido, únicamente hemos tenido que asociar la cámara y definir algún parámetro extra referente al vídeo.

Parar la grabación es similar. Simplemente hay que llamar al método `stop` y liberar el recurso:

```
protected void pararGrabar() {
    recorder.stop();
    recorder.reset();
}
```

Finalmente, para que nuestra aplicación funcione deberemos declarar los siguientes permisos en el `AndroidManifest`:

```
<uses-permission
    android:name="android.permission.RECORD_AUDIO" />
<uses-permission android:name="android.permission.CAMERA" />
<uses-permission
    android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
```

Opcionalmente también podemos definir las *features* que necesita nuestra *app*, así no se podrá instalar en dispositivos que no cumplan los requisitos:

```
<uses-feature android:name="android.hardware.camera"/>
<uses-feature
    android:name="android.hardware.camera.autofocus"/>
```

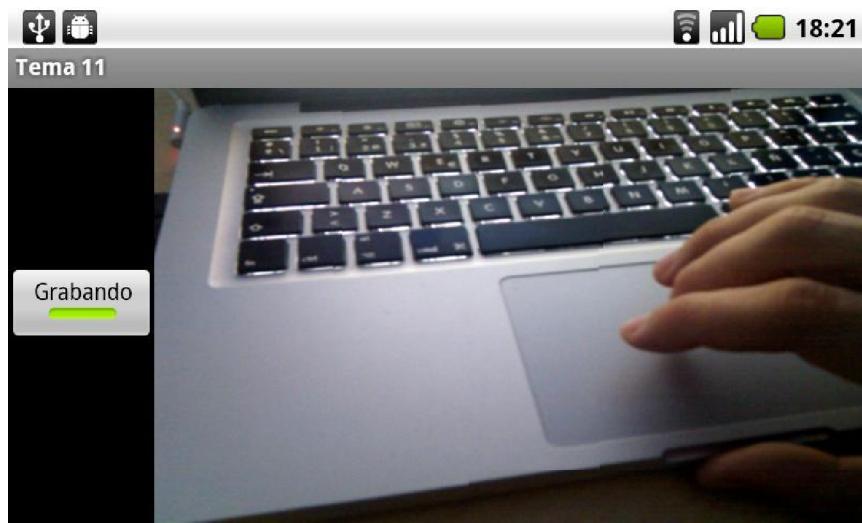


Figura 3. Grabadora en funcionamiento

Listo, ya tenemos nuestra grabadora de vídeo lista para ser usada



6. APROVECHAR LAS APLICACIONES DEL SISTEMA

En el ejemplo anterior hemos visto que a través de una clase *camera* podíamos acceder a la cámara del dispositivo, sin embargo esto nos obligaba a tener que definir una superficie de previsualización y a programar todas las funcionalidades que necesitáramos (botón de captura, etc.).

Existe otra forma de acceder a la cámara sin tener que programar tanto...

...¿Se imagina cuál?...

...Efectivamente, usando *Intents* para llamar a las aplicaciones que ya están programadas y que saben sacar fotos o grabar vídeos.

Veamos un ejemplo sencillo: supongamos que para nuestra aplicación necesitamos tomar una foto que deberemos incluir en un objeto tipo *ImageView*.

Pues bien, para tomar la foto bastaría con lanzar un *Intent* de la siguiente forma:

```
Intent intent = new Intent  
("android.media.action.IMAGE_CAPTURE");  
startActivityForResult(intent, TOMAR_FOTO);
```

Luego deberíamos sobrescribir el método *onActivityResult* de nuestra *Activity* para poder procesar el resultado:

```
@Override  
public void onActivityResult(int requestCode,  
int resultCode, Intent respuesta) {  
    if (resultCode == Activity.RESULT_OK && requestCode ==  
        TOMAR_FOTO) {  
        Bitmap bm = (Bitmap) respuesta.getExtras  
            () .get("data");  
        foto = (ImageView) findViewById(R.id.foto);  
        foto.setImageBitmap(bm);  
    }  
}
```

Al ejecutarse este código, se lanzará la aplicación de cámara fotográfica del sistema (o nos permitirá elegir entre las alternativas que tengamos instalada), una vez hayamos tomado la foto (podremos usar toda la funcionalidad como ajuste de color, zoom, etc.), la foto⁴ será devuelta a nuestra Activity para que podamos procesarla.

De igual forma que hemos tomado una foto, podemos capturar vídeo... Inténtalo, no es tan difícil.

⁴En este ejemplo nos devolverá una foto en tamaño miniatura. Consulte el manual para ver todas las opciones disponibles.

■ <http://developer.android.com/reference/android/provider/MediaStore.html>.

CONCLUSIONES



Destrezas, conocimiento y actitud son los elementos que te ayudarán a conseguir el **triunfo profesional**.

En este capítulo hemos visto que, como siempre, Android nos ofrece todas las herramientas necesarias para poder sacar partido a todas las características de nuestros terminales.

Hemos visto que, además de facilitarnos las clases necesarias para que podamos crear nuestros propios reproductores/grabadores de audio y vídeo, nos permite usar los que vienen preinstalados con el sistema para poder integrarlo con nuestras aplicaciones sin tener que reinventar la rueda.

RECAPITULACIÓN

A lo largo de las páginas de este capítulo hemos conocido las principales clases del paquete android.media.

Hemos aprendido que para reproducir audio o vídeo debemos usar la clase MediaPlayer y para grabar usaremos MediaRecorder.

También hemos visto otras formas de realizar estas tareas como, por ejemplo, reproducir un vídeo usando un objeto VideoView enlazado con un VideoController o capturar una foto lanzando un *Intent* que llama a la aplicación de cámara del sistema.



AUTOCOMPROBACIÓN

1. ¿Qué clase utilizamos para reproducir sonidos?

- a) MediaPlayer.
- b) MultimediaPlayer.
- c) SoundPlayer.
- d) MultimediaController.

2. ¿Qué clase utilizaremos para reproducir música interactiva?

- a) MediaPlayer.
- b) MusicPlayer.
- c) IMusicPlayer.
- d) JetPlayer.

3. ¿Cuál de estas afirmaciones es verdadera?

- a) Solo podemos reproducir archivos almacenados en la tarjeta SD.
- b) Solo podemos reproducir archivos almacenados en la memoria interna.
- c) Solo podemos reproducir archivos incluidos como *resources* en nuestro proyecto.
- d) Podemos reproducir archivos de la tarjeta SD, recursos incluidos en nuestro proyecto o ficheros referenciados por una URL.

4. **¿Qué método deberemos llamar una vez hayamos configurado nuestra clase *player*, justo antes de empezar a reproducir?**
 - a) *setData*.
 - b) *release*.
 - c) *prepare*.
 - d) *seekTo*.
5. **¿Qué método deberemos llamar para parar la reproducción?**
 - a) *seekTo*.
 - b) *pauseSound*.
 - c) *release*.
 - d) *stop*.
6. **Cuando hayamos terminado de reproducir y no queramos volver a usar la clase, ¿qué debemos hacer?**
 - a) Poner a *null* la variable.
 - b) Llamar al método *release* para liberar recursos.
 - c) Llamar al método *finish* para liberar recursos.
 - d) No es necesario hacer nada, ya se encarga el recolector de basura.
7. **Si la reproducción ha sido pausada (con una llamada a *pause*), ¿cómo continúo con la misma?**
 - a) Llamando al método *seekTo* para colocarnos en el milisegundo donde lo dejamos y luego llamando a *play*.
 - b) Volviendo a invocar a *pause*.
 - c) Invocando a *play*.
 - d) Invocando al método *resume*.
8. **¿Qué método utilizaría si quisiera reproducir un archivo de la red, si tengo su URL definida en la constante **URL SONIDO**?**
 - a) *setRemoteSource(URL SONIDO)*.
 - b) *setDataSource(URL SONIDO)*.
 - c) *setSource(URL SONIDO)*.
 - d) No se puede directamente. Primero debo descargar el archivo a un temporal y luego reproducirlo como local.



9. ¿Qué tipo de objeto podemos usar en nuestros *layout* para ver un vídeo?

- a) MultimediaLayout.
- b) MultimediaView.
- c) VideoView.
- d) VideoLayout.

10. ¿Qué clase deberemos asociar a un VideoView si queremos poner controles?

- a) VideoControl.
- b) MediaController.
- c) MediaPlayer.
- d) MediaPlayer.

11. ¿A qué método debemos llamar para cargar un vídeo remoto en el objeto VideoView?

- a) setVideo.
- b) setVideoURI.
- c) setVideoURL.
- d) setVideoRemote.

12. ¿Qué clase usamos para grabar un sonido?

- a) SoundRecorder.
- b) MediaRecorder.
- c) SoundMediaRecorder.
- d) Recorder.

13. ¿Qué permiso debemos declarar para poder grabar sonido?

- a) android.permission.RECORD.
- b) android.permission.RECORD_AUDIO.
- c) android.permission.RECORD_MIC.
- d) android.permission.RECORD_MULTIMEDIA.

- 14. ¿Qué objeto View podemos usar para previsualizar el contenido de la cámara?**
- a) CameraView.
 - b) CamView.
 - c) SurfaceView.
 - d) MediaView.
- 15. Si vamos a guardar un vídeo grabado en la tarjeta SD, ¿qué permiso debemos declarar?**
- a) android.permission.WRITE_EXTERNAL_STORAGE.
 - b) android.permission.WRITE_SD.
 - c) android.permission.WRITE_VIDEO.
 - d) android.permission.WRITE.
- 16. ¿Qué permiso debemos declarar para capturar imágenes de la cámara?**
- a) android.permission.READ_CAMERA.
 - b) android.permission.READ_HARDWARE.
 - c) android.permission.CAMERA.
 - d) No es necesario declarar ningún permiso.
- 17. ¿Qué acción debemos pasar como parámetro a un Intent para utilizar la aplicación de cámara para capturar una imagen para nuestra aplicación?**
- a) android.media.action.CAMERA.
 - b) android.media.action.SAVE_IMAGE.
 - c) android.media.action.GET_PHOTO.
 - d) android.media.action.IMAGE_CAPTURE.
- 18. ¿Si queremos grabar del micrófono qué parámetro pasaremos a set AudioSource?**
- a) MediaRecorder.AudioSource.MIC.
 - b) MediaRecorder.AudioSource.MICROPHONE.
 - c) MediaRecorder.AudioSource.MICRO.
 - d) MediaRecorder.AudioSource.INTERNAL_MIC.



19. ¿En qué carpeta de nuestro proyecto debemos colocar los archivos de sonido o vídeo?

- a) res/sound.
- b) res/video.
- c) res/raw.
- d) res/values.

20. Cuando capturo un vídeo, ¿puedo limitar su tiempo de captura en ms?

- a) Sí, con el método `setMaxDuration`.
- b) Sí, con el método `setMaxSize`.
- c) Sí, con un `Timer` que pare la grabación pasado un tiempo.
- d) No, no se puede.



SOLUCIONARIO

1.	a	2.	d	3.	d	4.	c	5.	d
6.	b	7.	a	8.	b	9.	c	10.	b
11.	b	12.	b	13.	b	14.	c	15.	a
16.	c	17.	d	18.	a	19.	c	20.	a



Decídete a ser el mejor, sé un P8.10.

PROPUESTAS DE AMPLIACIÓN

- JetCreator:
http://developer.android.com/guide/topics/media/jet/jetcreator_manual.html.
- Conocer la clase MediaStore para poder integrar nuestro contenido multimedia en ella:
<http://developer.android.com/reference/android/provider/MediaStore.html>.



BIBLIOGRAFÍA

- Formatos soportados:

<http://developer.android.com/guide/appendix/media-formats.html>.

- Media:

<http://developer.android.com/guide/topics/media/index.html>.

- MediaPlayer:

<http://developer.android.com/reference/android/media/MediaPlayer.html>.

- MediaRecorder:

<http://developer.android.com/reference/android/media/MediaRecorder.html>.

PROGRAMACIÓN PARA ANDROID

12

Widgets



ÍNDICE

MOTIVACIÓN.....	3
PROPÓSITOS	4
PREPARACIÓN PARA LA UNIDAD	5
1. WIDGETS	7
2. CREACIÓN DE UN <i>WIDGET</i>.....	10
2.1. DEFINIR UN <i>LAYOUT</i>	10
2.2. CREAR UN APPWIDGETPROVIDER	11
2.3. CREAR EL FICHERO DE CONFIGURACIÓN DEL <i>WIDGET</i>	12
2.4. DEFINIR EL <i>WIDGET</i> EN EL ANDROIDMANIFEST.XML	14
3. CICLO DE VIDA DE UN <i>WIDGET</i>	16
4. FORMAS DE ACTUALIZAR <i>WIDGETS</i>	18
4.1. USAR UN SERVICIO.....	19
4.2. USAR UN TIMERTASK	20
CONCLUSIONES	23
RECAPITULACIÓN	24
AUTOCOMPROBACIÓN	27
SOLUCIONARIO.....	33
PROPUESTAS DE AMPLIACIÓN	34
BIBLIOGRAFÍA.....	35



MOTIVACIÓN



Disfruta de un futuro mejor; nosotros te guiamos hacia tu triunfo profesional.

Los *widgets* de escritorio son una de las características del sistema Android. Se trata de pequeñas vistas que muestran información relevante de las aplicaciones dentro de otras aplicaciones.

El uso más conocido es el de los *widgets* es el de la aplicación de escritorio. Los usuarios pueden personalizar cada uno de los escritorios colocando información de sus aplicaciones preferidas (el tiempo, reproductor multimedia, *facebook*, relojes analógicos, etc.).

Así pues una de las cosas que podemos hacer para mejorar la experiencia de usuario de nuestra aplicación es crear nuestros propios *widgets*.

En este capítulo aprenderemos todo lo necesario para poder realizar esta tarea.

PROPOSITOS

Los objetivos marcados para esta unidad son:

- Entender la naturaleza y arquitectura de los *widgets*.
- Conocer el ciclo de vida de un *widget*.
- Distinguir las diferentes partes que componen un *widget*.
- Aprender a crear *widgets* de escritorio básicos.
- Aprender a definir servicios que actualicen *widgets*.
- Conocer métodos alternativos de actualización de *widgets*.



PREPARACIÓN PARA LA UNIDAD

Para esta unidad necesitaremos tener el entorno de desarrollo (Eclipse + *Plugin ADT*) configurado.

El código de *widgets* puede ser probado en el emulador.



1. WIDGETS

Tal y como dice el manual: “los widgets son vistas en miniatura que pueden ser embebidas en otras aplicaciones”.

El uso más común, que vamos a tratar en este capítulo, es el de colocar pequeños trozos de información (que se actualizan con más o menos frecuencia) en la pantalla principal.

Fueron introducidos en la versión 1.5 de Android y rápidamente se convirtieron en una de las principales características del sistema.

Son muy utilizados en las personalizaciones que las fábricas hacen de sus terminales (como por ejemplo, HTC Sense, Motorola Motorblur, etc.).

Ejemplos típicos son los widgets de reloj, de información meteorológica, de titulares de noticias, eventos del calendario, etc.

Para colocar un *widget* en el escritorio, el usuario no tiene más que hacer una pulsación larga en cualquier zona libre del mismo para que se despliegue el menú contextual.



Figura 1. Menú contextual de escritorio

Desde este menú, pulsando sobre la opción *Widgets* se mostrarán todos los disponibles.

Una vez seleccionado uno aparecerá en el espacio asignado (podremos recolocarlo arrastrándolo a otro sitio). Quedando reservada esa cuadrícula para que la aplicación muestre la información deseada.

Los usuarios, pueden interactuar con el *widget*, por ejemplo con el *widget* del reproductor de música podemos cambiar de canción, parar la reproducción, etc.

Para quitarlo de la pantalla solo tendremos que arrastrarlo a la papelera que se muestra al seleccionarlo.



Figura 2. Ejemplos de widgets incluidos en el sistema

A nivel técnico:

Los *widgets* se colocan dentro de un *AppWidgetHost*¹.

Son objetos *BroadcastReceiver* especiales asociados a un fichero XML de configuración. Esto les permite funcionar de forma aislada al resto de las aplicaciones.

Así pues para crear un *widget* tendremos que crear:

- Un “proveedor de mensajes de difusión para ‘widget’” o lo que es lo mismo, un objeto *AppWidgetProvider*.
- También deberemos crear un *layout* que defina su aspecto y componentes de vista.
- También es imprescindible definir la frecuencia de actualización

Opcionalmente podremos definir un “Configurador de ‘Widget’” que permitirá al usuario definir los parámetros del mismo (por ejemplo, en el caso de un *widget* de meteorológico permitiría seleccionar la ciudad de la que queremos saber el tiempo que hace).

¹ El escritorio contiene un *AppWidgetHost* donde colocamos los *widgets* a través de su menú contextual, tal y como hemos visto antes.

2. CREACIÓN DE UN *WIDGET*

Para comprender las diferentes partes que componen un *widget* vamos a crear uno muy sencillo que muestre un número aleatorio que irá cambiando en intervalos regulares.

Los pasos a seguir para crear un *widget* son:

2.1. DEFINIR UN *LAYOUT*

Lo primero que tenemos que hacer es definir su aspecto. Para ello crearemos un fichero xml de *layout*.

Sin embargo, debido a que se ejecutan en un entorno especial, los *widgets* utilizan un tipo de vista especial llamado *RemoteViews*.

Solo los objetos que implementan el interface *RemoteView* pueden ser utilizados en estas vistas.

Los controladores que actualizan el contenido de un *RemoteView* se ejecutan en segundo plano. Nosotros como programadores no podremos acceder a ellos directamente, tendremos que llamar a métodos que los actualicen de forma remota.

Es decir, si yo tengo un objeto *TextView* llamado “nombre”, no podré hacer una llamada del estilo: *nombre.setText(“Pepe”)*. Lo que sí podrá hacer es una llamada remota a través de un método de un objeto *RemoteView*: *rView.setTextViewText(R.id.nombre, “Pepe”)*.

Actualmente², los objetos que lo implementan (y por lo tanto, podemos usar en nuestros *widgets*) son:

² Esto puede cambiar en futuras versiones, consulte la documentación.



- *Layouts:*
 - FrameLayout.
 - LinearLayout.
 - RelativeLayout.
- *Views:*
 - AnalogClock.
 - Button.
 - Chronometer.
 - ImageButton.
 - ImageView.
 - ProgressBar.
 - TextView.

En nuestro caso solo necesitamos un campo TextView con algunos parámetros de estilo definidos para hacer su aspecto más agradable:

```
<?xml version="1.0" encoding="utf-8"?>
<TextView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/aleatorio"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:text="@string/cargando"
    android:background="@drawable/fondo"
    android:textSize="35dip"
    android:textStyle="bold"
    android:gravity="center_vertical|center_horizontal"
    android:layout_margin="5dip"
    android:shadowColor="#6000"
    android:shadowDx="2"
    android:shadowDy="2"
    android:shadowRadius=".5" />
```

res/layout/randomwidget.xml

En este caso solo hemos utilizado un objeto, pero nuestra vista podría ser tan compleja como quisieramos siempre y cuando solo usemos los objetos View citados anteriormente.

2.2. CREAR UN APPWIDGETPROVIDER

El siguiente paso es crear la clase que deberá reaccionar ante los mensajes de Broadcast dirigidos a nuestro *widget*.

Para ello extenderemos la clase AppWidgetProvider.

Esta es una clase muy similar a BroadcastReceiver con la diferencia de que pasa los datos relevantes para el *widget* y los pasa como parámetros a métodos que son llamados con cada evento relevante.

Así pues, podemos por ejemplo sobrescribir el método *onUpdate* para que cuando recibamos en mensaje de “actualízate” cambiemos el contenido del *widget* por un nuevo número aleatorio:

```
public class RandomWidget extends AppWidgetProvider {
    @Override
    public void onUpdate(Context context,
        AppWidgetManager appWidgetManager,
        int[] appWidgetIds) {
        for(int x=0 ; x < appWidgetIds.length ; x++) {
            // Calculamos el número aleatorio a mostrar
            int numAleatorio = (new Random()).nextInt(100);
            // Lo colocamos en una vista remota
            RemoteViews rView = new RemoteViews(
                context.getPackageName(),
                R.layout.randomwidget
            );
            rView.setTextViewText(R.id.aleatorio, "" +
                numAleatorio);
            // Mandamos la vista al widget
            int id = appWidgetIds[x];
            appWidgetManager.updateAppWidget(id, rView);
        }
    }
}
```

Una cosa que debemos tener en cuenta es que podemos tener varios *widgets* del mismo tipo colocados en el escritorio al mismo tiempo (por ejemplo, varios relojes, etc.), así que nuestra clase deberá actualizar todos en el método *onUpdate* (por eso ejecutamos el código en un bucle).

2.3. CREAR EL FICHERO DE CONFIGURACIÓN DEL *WIDGET*

Hemos visto que la tercera cosa que necesitamos para definir un *widget* es asignarle un período de refresco. Esto (y otros parámetros) se define en su fichero de configuración.

Los ficheros de configuración de *widget* son archivos xml situados en la carpeta de recursos res/xml/.

Además del tiempo de refresco (en milisegundos), contienen la dimensión mínima del *widget* y, opcionalmente, la Activity de configuración asociada.

La pantalla home de Android por defecto, está diseñada basada en una rejilla de 4 x 4 celdas.



Así que las dimensiones de nuestros *widgets* de escritorio deberán estar basadas en celdas, por lo que nos podremos elegir entre un amplio abanico de tamaños como³: 4 x 1, 3 x 3, 2 x 3, etc.

Estos tamaños deberemos convertirlos a dip. Para ello usaremos la siguiente fórmula:

$$\text{Tamaño mínimo en dip} = (\text{Número de celdas} * 74\text{dip}) - 2\text{dip}$$

En nuestro caso vamos a crear un *widget* de 1 x 1 celda, por lo que su tamaño será de 72 dip x 72 dip.

Para crear el fichero de configuración podemos usar la opción “Android XML file” del menú “File/New”.

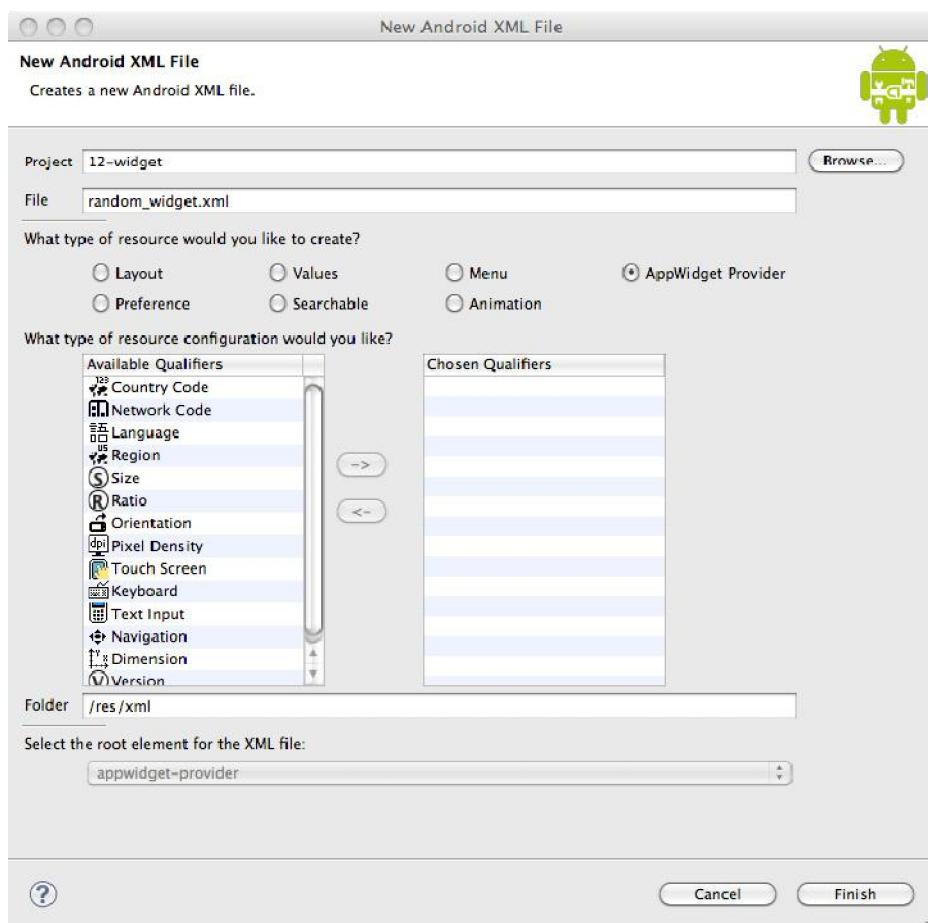


Figura 3. Diálogo de creación de un fichero de configuración de Widget

3 Véase la documentación para ampliar conocimientos sobre el diseño de los *widgets* en :
■ http://developer.android.com/guide/practices/ui_guidelines/widget_design.html.

Una vez editado, bien mediante el asistente bien tecleando el código a mano, obtendremos un fichero xml similar a este:

```
<?xml version="1.0" encoding="utf-8"?>
<appwidget-provider xmlns:
    android="http://schemas.android.com/apk/res/android"
    android:initialLayout="@layout/relojwidget"
    android:updatePeriodMillis="10000"
    android:minHeight="72dip"
    android:minWidth="72dip"
/>
```

res/xml/randomwidget_conf.xml

Hay que decir que, como veremos más adelante, aunque hemos puesto que se refresque cada 10 segundos, esto no nos asegura que nuestro *widget* sea actualizado exactamente en esa frecuencia (ya que funcionamos gracias a un sistema de mensajes de *broadcast*).

Es más, la documentación nos aconseja espaciar estas actualizaciones lo máximo posible (mínimo una hora) para ahorrar batería.

2.4. DEFINIR EL *WIDGET* EN EL ANDROIDMANIFEST.XML

Finalmente, deberemos juntar todas las piezas del puzzle en el manifiesto, asociando nuestra clase *AppWidgetReceiver* con el fichero de configuración de *widget* y definiendo un filtro para que sea avisada cuando se lance un aviso *broadcast* de actualización de *widget*.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:
    android="http://schemas.android.com/apk/res/android"
    package="es.masterd.u12"
    android:versionCode="1"
    android:versionName="1.0">
    <application android:icon="@drawable/icon"
        android:label="@string/app_name">
        <receiver android:name=".RandomWidget"
            android:label="@string/widget_random">
            <intent-filter>
                <action
                    android:name=
                    "android.appwidget.action.APPWIDGET_UPDATE"
                />
            </intent-filter>
            <meta-data
                android:name="android.appwidget.provider"
                android:resource="@xml/randomwidget_conf" />
        </receiver>
        <uses-sdk android:minSdkVersion="3" />
    </application>
</manifest>
```

AndroidManifest



Una vez instalada, cuando el usuario pulse sobre el menú de *widgets*, le aparecerá el nuestro entre las opciones y podrá instalarlo tantas veces como quiera.

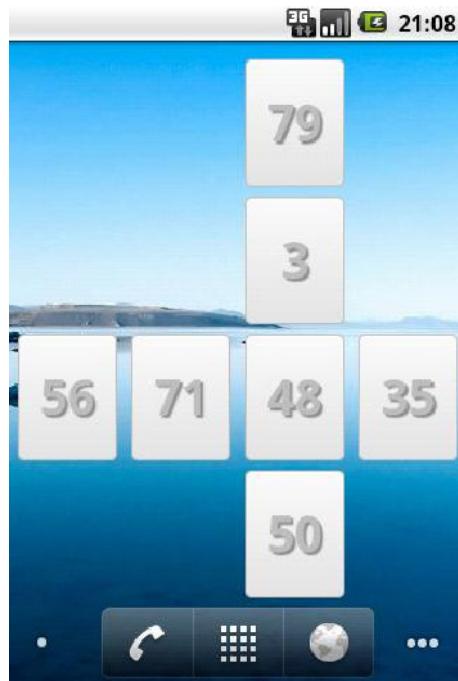


Figura 4. Ocho instancias de nuestro *widget* en funcionamiento

3. CICLO DE VIDA DE UN *WIDGET*

El ciclo de vida de un *widget* se podría resumir en las siguientes fases:

- Definición de *widget*.
- Creación de la instancia.
- Llamadas periódicas al método *onUpdate* (según la frecuencia definida).
- Respuesta a eventos *onClick*.
- Borrado del *widget*.
- Desinstalación.

El ciclo de vida del *widget* empieza con la definición del *widget*. Ya hemos visto que necesitamos un *layout* o una clase *AppWidgetProvider*, un fichero de configuración y una entrada en el Manifiesto.

Con todo esto, cuando instalamos la aplicación el sistema sabe que hay un nuevo *widget* disponible.

La segunda fase se inicia cuando el usuario selecciona el *widget* y lo instala en el escritorio. Android invoca a la Activity de configuración (si ha sido definida). Esta fase invoca al método *onEnabled* de la clase *AppWidgetProvider*. Esta llamada indica que hay por lo menos una instancia del *widget* en funcionamiento.

Cada vez que el tiempo de refresco se cumple, Android, a través del *BroadcastReceiver* que hemos definido en el manifiesto, llama al método *onUpdate*. En otras palabras, crea una instancia del objeto *AppWidgetProvider*, llama a su método *onUpdate* y lo libera dejándolo a disposición del recolector de basura.

Lo ideal es que este proceso dure el menor tiempo posible para que pueda ser liberado lo antes posible (en el siguiente capítulo veremos diferentes enfoques para conseguir esto).



Por otro lado, debido a la naturaleza de los *widgets*, y a que son invocados por llamadas del sistema que pueden ser terminadas en cualquier momento (y sus instancia destruidas), no podemos usar variables estáticas en nuestras clases AppWidgetProvider, tendremos que usar trucos alternativos como SharedPreferences, bases de datos, etc.

Una cosa de la que tenemos que acordarnos en nuestros métodos *onUpdate* es la de definir los eventos OnClick de los componentes que sean necesarios. De esta forma Android podrá gestionar dichos eventos.

Cuando un *widget* se envía a la papelera se genera un evento de *onDelete* al AppWidgetProvider y, si ya no quedan más *widgets* de ese tipo en funcionamiento, se invoca también *onDisable*.

La última fase se produciría cuando desinstalamos la aplicación: se borrarán todos los archivos y nuestro *widget* desaparecerá del listado de *widgets* disponibles.

4. FORMAS DE ACTUALIZAR WIDGETS

Ya hemos visto que la velocidad de ejecución del método *onUpdate* es crítica, así que cuando preveamos que nuestro método puede convertirse en algo pesado debemos buscar alternativas para no ralentizar el sistema.

En este capítulo veremos dos enfoques alternativos al método *onUpdate* que hemos visto en el ejemplo anterior.

Para el ejemplo vamos a utilizar dos *widgets* que muestran un reloj en pantalla. La idea es que se actualice cada minuto.

El primero se utiliza una clase de utilidad de Java (*TimerTask*) para ejecutar periódicamente la actualización.

El segundo se basa en el uso de un objeto Service propio de Android.

La definición del *layout* que usan ambos es la siguiente:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:layout_margin="7dip">
    <TextView android:id="@+id/hora"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/cargando"
        android:background="@drawable/fondo"
        android:textSize="55dip"
        android:gravity="center_vertical|center_horizontal"
        android:shadowColor="#600000"
        android:shadowDx="5"
        android:shadowDy="3"
        android:shadowRadius=".5" />
</LinearLayout>
```

res/layout/relojwidget.xml



4.1. USAR UN SERVICIO

```
public class RelojWidget2 extends AppWidgetProvider {  
    @Override  
    public void onUpdate(Context context,  
        AppWidgetManager appWidgetManager,  
        int[] appWidgetIds) {  
        context.startService( new Intent(context,  
            ActualizarRelojService.class));  
    /**  
     * Servicio de actualización del widget  
     */  
    public static class ActualizarRelojService extends Service {  
        public void onStart(Intent intent, int startId) {  
            // Build the widget update for today  
            RemoteViews updateViews = new RemoteViews(  
                this.getPackageName(),  
                R.layout.relojwidget);  
            final java.text.DateFormat dateformat =  
                DateFormat.getTimeFormat(this);  
            String ahora = dateformat.format  
                ( System.currentTimeMillis() );  
            updateViews.setTextViewText(R.id.hora, ahora);  
            // Push update for this widget to the home screen  
            ComponentName thisWidget = new ComponentName  
                (this, RelojWidget2.class);  
            AppWidgetManager manager =  
                AppWidgetManager.getInstance(this);  
            manager.updateAppWidget(thisWidget, updateViews);  
        }  
        @Override  
        public IBinder onBind(Intent intent) {  
            // No permitimos conectar con este servicio  
            return null;  
        }  
    }  
}
```

RelojWidget2

Nuestro método `onUpdate` solo hará una cosa: lanzar un servicio que se ejecutará en segundo plano y se encargará de realizar la actualización. Una vez lanzado el servicio, el `AppWidgetProvider` quedará liberado.

El fichero de configuración del `widget` será similar a los que ya hemos usado.

```
<?xml version="1.0" encoding="utf-8"?>  
<appwidget-provider  
    xmlns:android="http://schemas.android.com/apk/res/android"  
        android:minWidth="294dp"  
        android:minHeight="72dp"  
        android:updatePeriodMillis="30000"  
        android:initialLayout="@layout/relojwidget"  
>  
</appwidget-provider>
```

Donde sí que tenemos algún cambio es en el AndroidManifest, ya que nos vemos obligados a definir el servicio. Como es una clase interna la referenciamos con el nombre de su clase padre seguido de un signo “\$” y el nombre de la clase del servicio.

```
<receiver
    android:name=".RelojWidget2"
    android:label="@string/widget_reloj2">
<intent-filter>
<action
    android:name="android.appwidget.action.APPWIDGET_UPDATE" />
</intent-filter>
<meta-data android:name="android.appwidget.provider"
    android:resource="@xml/relojwidget2_conf" />
</receiver>
<service
    android:name=".RelojWidget2$ActualizarRelojService" />
```

4.2. USAR UN TIMERTASK

Este método se basa en usar clases puras de java para programar las recargas en vez de delegar esa tarea en el sistema.

Así que lo primero que debemos hacer es definir un *widget* que no se recargue, o lo que es lo mismo, poner el updatePeriodMillis a cero:

```
<?xml version="1.0" encoding="utf-8"?>
<appwidget-provider
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:minWidth="294dp"
    android:minHeight="72dp"
    android:updatePeriodMillis="0"
    android:initialLayout="@layout/relojwidget"
    >
</appwidget-provider>
```

Con esta configuración nuestro *widget* no se actualizará, así que deberá ser él mismo el que programe sus recargas.

Para ello, crearemos una clase de TimerTask que se encargue de crear las RemoteView y pasárselas al *widget*.

La frecuencia de recarga quedará definida cuando creamos la instancia de esta clase.

```
public class RelojWidget3 extends AppWidgetProvider {
    @Override
    public void onUpdate(Context context,
        AppWidgetManager appWidgetManager,
        int[] appWidgetIds) {
        Timer timer = new Timer();
        timer.scheduleAtFixedRate(
            new ActualizarRelojTimerTask
                (context, appWidgetManager), 1, 30 * 1000);
```



```
}

/**
 * Clase timer que se encargará de actualizar el widget cada cierto tiempo
 */
class ActualizarRelojTimerTask extends TimerTask {
    private Context context;
    public ActualizarRelojTimerTask(Context context,
        AppWidgetManager appWidgetManager) {
        this.context = context;
    }
    @Override
    public void run() {
        RemoteViews updateViews = new RemoteViews(
            context.getPackageName(),
            R.layout.relojwidget );
        final java.text.DateFormat dateformat =
        DateFormat.getTimeFormat(context);
        String ahora = dateformat.format(
        System.currentTimeMillis());
        updateViews.setTextViewText(R.id.hora, ahora);
        ComponentName thisWidget = new ComponentName(
            context,
            RelojWidget3.class);
        AppWidgetManager manager =
        AppWidgetManager.getInstance(context);
        manager.updateAppWidget(
            thisWidget, updateViews);
    }
}
```

Con este método conseguiremos una mayor precisión en el momento de la actualización, pero a cambio perderemos mucha eficiencia en el ahorro del consumo de batería, por lo que hay que usarlo con mucha precaución.



CONCLUSIONES



Demuestra lo que vales; el triunfo profesional está más cerca de lo que piensas.

Como hemos podido ver, utilizar *widgets* para mostrar información relevante de nuestras aplicaciones es muy sencillo.

Podemos, por ejemplo, mostrar el estado de un inventario, alertas ante sucesos determinados, las últimas noticias, cuentas atrás, mensajes de usuarios, etc.

Si vamos a colocar el *widget* en el escritorio, deberemos seguir la guía de estilo del manual para integrarlo con el resto de componentes. Tendremos que diseñar su tamaño en función de las celdas que vaya a ocupar (tenemos 4 x 4 disponibles). Si es posible, es una buena idea preparar varias versiones del *widget* de diferentes tamaños para que el usuario pueda elegir el que más se adecúe a sus necesidades.

Con un poco de diseño gráfico en nuestros fondos podemos conseguir resultados espectaculares.

Otra cosa a tener en cuenta es la frecuencia de actualización ya que puede consumir la batería rápidamente. Por eso se recomienda que no baje de los 60 minutos.

RECAPITULACIÓN

Para crear un *widget* debemos seguir los siguientes pasos:

1. Definir su aspecto creando un *layout*.
2. Crear un clase que extienda AppWidgetProvider.
3. Crear un fichero de configuración de *widget*.
4. Definir el *widget* en el AndroidManifest.

Debemos tener en cuenta que los *widget* descienden de BroadcastReceiver, por lo que funcionan mediante mensajes que activan métodos. Así que nuestra clase de tipo AppWidgetProvider podrá recibir llamadas a los siguientes métodos:

- *onEnabled*, al crear una instancia del *widget*.
- *onUpdate*, → se invocará cuando toque refrescar el contenido del *widget*.
- *onDelete*, → será llamado cuando el *widget* es enviado a la papelera.
- *onDisable*, → cuando un *widget* sea borrado y ya no queden más instancias del mismo.

Otra característica de los *widgets* es que, debido a que sus componentes de tipo View son gestionado por un proceso que se ejecuta en segundo plano, no podemos usarlos de la forma convencional y debemos usar los métodos de interface que nos provee la clase RemoteView.

Solo aquellos componentes que implementen estos métodos podrán ser utilizados en los *widgets*.



Finalmente, hemos visto diversos enfoques para actualizar los datos:

- Colocando el código dentro del método *onUpdate*.
- Lanzando un servicio desde el *onUpdate* para que se ejecute en segundo plano. Esta es una buena opción si es una tarea un poco pesada y no queremos ralentizar el terminal.
- Usando nuestro propio objeto Timer para gestionar las recargas, lo que nos da una mejor tasa de refresco a cambio de sacrificar el ahorro de energía.



AUTOCOMPROBACIÓN

1. ¿Qué es un *widget*?

- a) Son vistas en miniatura que pueden ser embebidas en otra aplicación.
- b) Son vistas en miniatura que solo pueden ser embebidas en el escritorio.
- c) Son pequeños programas que pueden ser embebidos en nuestros *layout*.
- d) Son pequeños programas que pueden ser embebidos en el explorador.

2. ¿Desde qué versión Android están disponibles los *widgets*?

- a) 1.0.
- b) 1.5.
- c) 1.6.
- d) 2.1.

3. ¿Cómo se coloca un *widget* en el escritorio?

- a) Desde el menú de la aplicación.
- b) Desde el menú de aplicaciones.
- c) Desde el menú contextual del escritorio.
- d) Instalando la aplicación de *widget* necesaria.

4. ¿Cómo se borra un *widget* del escritorio?

- a) Desde el menú de *widget*.
- b) Desde el menú de la aplicación.
- c) Con la tecla borrar.
- d) Arrastrándolo a la papelera.

5. ¿De qué tipo de objeto heredan las clases de *widget*?

- a) BroadcastReceiver.
- b) BroadcastWidget.
- c) Widget.
- d) WidgetReceiver.

6. ¿Cuál de estas opciones no es imprescindible para crear un *widget*?

- a) Un objeto de tipo *widget*.
- b) Un objeto de tipo AppWidgetProvider.
- c) Un *layout*.
- d) Un fichero de configuración de *widget*.

7. ¿Qué tipo de objetos View podemos usar en nuestros *widgets*?

- a) Layout.
- b) Widgets.
- c) View.
- d) RemoteView.

8. ¿Podemos acceder a las propiedades de los objetos RemoteView?

- a) Sí, directamente. Igual que lo hacemos en una Activity.
- b) Sí, indirectamente, usando métodos de interface.
- c) Tanto directa como indirectamente.
- d) No, no podemos de ninguna manera.



- 9. ¿Qué objeto no puede ser utilizado en un *layout* de *widget*?**
 - a) FrameLayout.
 - b) Button.
 - c) TableLayout.
 - d) AnalogClock.
- 10. ¿Qué clase deberemos extender para gestionar las actualizaciones del contenido del *widget*?**
 - a) BroadcastReceiver.
 - b) AppWidgetHost.
 - c) Service.
 - d) AppWidgetProvider.
- 11. ¿Qué propiedades debemos definir, como mínimo, en el fichero de configuración del *widget*?**
 - a) initialLayout, minHeight, minWidth, config.
 - b) minHeight, minWidth, updatePeriodMillis, config.
 - c) initialLayout, minHeight, minWidth, updatePeriodMillis.
 - d) initialLayout, minHeight, minWidth, updatePeriodMillis, config.
- 12. ¿Qué *tag* deberemos usar en el *AndroidManifest* para definir un *widget*?**
 - a) <activity>.
 - b) <widget>.
 - c) <service>.
 - d) <receiver>.
- 13. ¿Qué filtro de *intent* deberemos definir dentro del *tag* que define un *widget* en el *AndroidManifest*?**
 - a) android.appwidget.action.APPWIDGET_UPDATE.
 - b) android.appwidget.action.APPWIDGET.
 - c) android.appwidget.action.WIDGET_RECEIVER.
 - d) android.appwidget.action.WIDGET_UPDATE.

14. ¿Qué método se llamará al crearse un *widget*?

- a) onEnabled.
- b) onUpdate.
- c) onDelete.
- d) onDisable.

15. ¿Qué método se llamará cuando se deba actualizar un *widget*?

- a) onEnabled.
- b) onUpdate.
- c) onDelete.
- d) onDisable.

16. ¿Qué tipo de objeto deberemos utilizar para actualizar el estado de un *widget*?

- a) AppWidgetManager.
- b) ComponentName.
- c) RemoteViews.
- d) Service.

17. ¿Qué método deberemos sobrescribir obligatoriamente al definir un *service*?

- a) onStart.
- b) onBind.
- c) onStop.
- d) onUpdate.

18. ¿Qué clase podemos utilizar para programar las actualizaciones del *widget*?

- a) Thread.
- b) ThreadTask.
- c) TimerTask.
- d) UpdaterTask.



19. ¿Qué método de la clase TimerTask debemos sobrescribir?

- a) *onUpdate*.
- b) *start*.
- c) *onCreate*.
- d) *run*.

20. Si tenemos un servicio llamado “MiService” dentro de un *widget* llamado “MiWidget”, ¿cómo lo definiremos en el AndroidManifest?

- a) <service android:name=".MiService" />.
- b) <service android:name=".MiWidget" />.
- c) <service android:name=".MiWidget.MiService" />.
- d) <service android:name=".MiWidget\$MiService" />.



SOLUCIONARIO

1.	a	2.	b	3.	c	4.	d	5.	a
6.	a	7.	d	8.	b	9.	c	10.	d
11.	c	12.	d	13.	a	14.	a	15.	b
16.	a	17.	b	18.	c	19.	d	20.	d



Dedición, esfuerzo, actitud... Un P8.10 siempre es capaz de alcanzar el triunfo profesional.

PROPUESTAS DE AMPLIACIÓN

Se recomienda ver la siguiente conferencia del Google IO que habla de cómo mejorar la velocidad de nuestras aplicaciones:

- <http://www.youtube.com/watch?v=c4znvD-7VDA>.



BIBLIOGRAFÍA

- App Widgets:

<http://developer.android.com/guide/topics/appwidgets/index.html>.

- Widget Design Guides:

http://developer.android.com/guide/practices/ui_guidelines/widget_design.html.

PROGRAMACIÓN PARA ANDROID

13

Publicando en el Market



Escuela Universitaria
de Formación Abierta

master.D
GRUPO



ÍNDICE

MOTIVACIÓN.....	3
PROPÓSITOS	4
PREPARACIÓN PARA LA UNIDAD	5
1. EL ANDROID MARKET	7
2. ¿ESTÁ LISTA NUESTRA APLICACIÓN?	9
3. PUBLICAR EN ANDROID MARKET.....	13
4. MEJORAR NUESTRA APLICACIÓN	15
CONCLUSIONES	19
RECAPITULACIÓN	20
AUTOCOMPROBACIÓN	21
SOLUCIONARIO.....	27
PROPUESTAS DE AMPLIACIÓN	28
BIBLIOGRAFÍA.....	29



MOTIVACIÓN



Dispondrás de las mejores herramientas para tu formación.

Está claro que, normalmente, el fin último de nuestras aplicaciones es conseguir que la gente se las instale y las use en sus terminales. Así pues no solo tenemos que programarlas, sino que también tenemos que distribuirlas.

Como no podía ser de otra forma, también en esto nos ayuda Android poniendo a nuestra disposición el Android Market.

Se trata de una tienda mundial con escaparates en la mayoría de los terminales Android, desde los que los usuarios pueden descargar e instalar nuestras aplicaciones (gratuitas o de pago) de una forma sencilla.

Pero no solo es publicar, también deberemos hacer un seguimiento para corregir posibles *bugs* o añadir mejoras.

En este capítulo conocemos los pasos necesarios para publicar una aplicación y repasaremos una serie de consejos que nos pueden ser muy útiles para su mantenimiento posterior.

PROPOSITOS

En esta unidad vamos a:

- Conocer el funcionamiento del Android Market desde el punto de vista de desarrollador.
- Aprender a preparar nuestras aplicaciones para su publicación.
- Repasar consejos útiles para mejorar la distribución.
- Repasar consejos útiles que nos ayudarán a mejorar nuestras aplicaciones.



PREPARACIÓN PARA LA UNIDAD

Esta unidad está enfocada a aquellos programadores que han terminado una aplicación Android y quieren distribuirla a los usuarios finales.



1. EL ANDROID MARKET

El Android Market es el sistema de distribución de aplicaciones desarrollado por Google y que va incluido en la mayoría de los terminales Android.

Permite que los usuarios accedan a un inmenso repositorio de aplicaciones de terceros que se instalan con solo pulsar un botón.

Tiene un interface sencillo e intuitivo que permite navegar por categorías, realizar búsquedas, consultar fichas, comentarios...



Figura 1. Android Market

Cualquier programador de cualquier parte del mundo puede subir sus aplicaciones a este escaparate mundial con millones de clientes potenciales.

Las aplicaciones pueden ser gratuitas o de pago y están organizadas en las siguientes categorías:

- Compras
- Comunicación
- Cómics
- Deportes
- Estilo de vida
- Finanzas
- Herramientas
- Juegos
- Multimedia
- Noticias y tiempo
- Ocio
- Productividad
- Referencia
- Salud
- Sociedad
- Temas
- Viajes
- Bibliotecas de software
- Demostración

Las aplicaciones de pago solo están disponibles para determinados países (el número va aumentando poco a poco) y se cobran a través de Google Checkout y se dispone de 48 horas para probarlas (si antes de este tiempo se desinstala el dinero es reembolsado).

Una vez comprada una aplicación podremos instalarla/desinstalarla (sin coste adicional) tantas veces como queramos ya que quedará registrada en nuestra pestaña “Descargas”.

Cada aplicación tiene una ficha en la que se muestran un par de pantallazos de la aplicación, un texto explicativo (escrito por el programador), los datos de contacto del programador y los comentarios y valoraciones de los usuarios que se la han descargado.

Para publicar una aplicación en el Market solo se necesita tener una cuenta de desarrollador y cinco minutos...

Pero... ¿estamos seguros de que nuestra aplicación está lista para ser distribuida al público general?



2. ¿ESTÁ LISTA NUESTRA APLICACIÓN?

En este capítulo vamos a ver los pasos necesarios para comprobar que nuestra aplicación está lista para ser publicada.

¿Está mi código completo y libre de errores?

Parece obvio, pero conviene recordarlo, solo deberíamos publicar aplicaciones terminadas y que han sido suficientemente probadas.

Las prisas pueden ser malas consejeras, subir una aplicación con errores o incompleta puede volverse contra nosotros en forma de usuarios descontentos que puntúan a la baja y posiblemente no vuelvan a probarla cuando esté terminada.

¿He limpiado el código y los ficheros de datos?

Muchas veces cuando desarrollamos colocamos mostramos mensajes, logueamos resultados, inicializamos variables, etc., en definitiva, código *debug* que debemos comentar/borrar antes de generar el apk.

También debemos comprobar que no dejamos ningún fichero o base de datos relleno con datos ficticios.

¿Está el AndroidManifest correcto?

Como hemos visto a lo largo de las unidades anteriores, en el fichero *AndroidManifest.xml* de nuestro proyecto definimos los componentes principales de nuestra aplicación (actividades, proveedores de contenido, permisos necesarios, etc.).

Este fichero es usado por cada dispositivo para comprobar que la aplicación puede ser instalada en ella.

El Android Market también usa este archivo cuando subimos una aplicación. De él toma, entre otras, cosas tan importantes como el nombre de aplicación, el icono, el nombre de paquete, y los requisitos mínimos para su ejecución.

Esto último es importante, ya que el Market solo muestra las aplicaciones que pueden ser instaladas en el terminal que se está consultando¹.

Es decir si en el `AndroidManifest` hemos declarado que nuestra aplicación necesita Android 2.2 y consultamos el Market desde un terminal Android 1.6, no aparecerá en los resultados de nuestras búsquedas. Lo mismo pasará si declaramos que, por ejemplo, nuestra app necesita gps y el dispositivo no dispone de uno.

Concretemos un poco más, los *tags* y atributos a los que debemos prestar una atención especial antes de generar el fichero .apk con la versión que subiremos al Market:

- En el tag **<manifest>**

- **package**, → se utiliza como clave única en el Android Market (no puede haber dos aplicaciones publicadas con el mismo Market). No está permitido usar espacios de nombres reservados como: com.android, example... Hay que tener en cuenta que en el momento que se publica una aplicación el espacio de nombres queda ocupado para siempre (aunque retiremos la app, seguirá ocupado).
- **android:versionCode**, → es un número entero que debe ser incrementado antes de subir una actualización (o de lo contrario dicha actualización será rechazada). También sirve para realizar comprobaciones de compatibilidad desde el código. Es una buena idea hacer corresponder este número con el número de versión de nuestros sistemas de control de versiones (Subversion, SVN, Git...).
- **android:versionName**, → una cadena de texto identificativa de la versión que será la que se muestre al usuario en el Market.

- En el tag **<application>**

- **android:icon**, → es el ícono de la aplicación, además de integrarse en el *launcher*, también se usa en el Market, por lo que debemos asegurarnos de crear un ícono propio que cumpla las reglas de estilo². Hay que tener en cuenta que va a ser la primera impresión que se van a llevar nuestros usuarios. Un buen ícono puede hacer que los usuarios se decanten por nuestra aplicación en vez de la de la competencia.

¹ Véase: *Strategies for Legacy Applications* en http://developer.android.com/guide/practices/screens_support.html#strategies.

² http://developer.android.com/guide/practices/ui_guidelines/icon_design.html.



- Tags **<uses-configuration>** y **<uses-features>** → permiten definir los requisitos de hardware (dispositivos necesarios, tipos de teclados requeridos, etc.), no es obligatorio definirlos, pero sí que es recomendable, para evitar que un usuario que no puede correr la app se la descargue y no pueda ejecutarla, generando posiblemente un voto/comentario negativo.
- Tag **<uses-sdk>**
 - **android:targetSdkVersion**, → plataforma Android para la que está desarrollada la aplicación (obligatorio definirla, sino no podremos subirla).
 - **android:minSdkVersion**, → utilizando diferentes técnicas³ (como el patrón de diseño Reflection), podemos hacer que nuestra aplicación se ejecute en versiones anteriores a la definida en el tag anterior. Este tag define la versión más baja en la que se puede usar.

Como siempre, recomendamos leer el manual para ver todas las opciones que podemos definir en el `AndroidManifest`.

¿He tenido en cuenta a todos los usuarios?

Cuando desarrollamos, normalmente vamos probando nuestro código siempre en el mismo dispositivo (o emulador) y muchas veces olvidamos que existen infinidad de dispositivos y configuraciones posibles. Es típico oír decir a un programador cosas del estilo: “Pues en mi pantalla se ve bien”. ¡Maaaaal...! No todo el mundo tiene tu Nexus One, es posible que esté usándola en un terminal de gama baja con pantalla pequeña y por eso la maquetación se rompe.

Así pues, una vez que todo funciona correctamente en nuestro entorno de desarrollo, debemos probarlo en el mayor número de escenarios posibles.

Es una buena idea crear varios emuladores (diferentes versiones, diferentes tamaños de pantalla, etc.) y probar que todo está correcto. Comprobaremos que todo funciona bien tanto en posición vertical como horizontal.

También es una buena idea dejar probar “usuarios tipo” ajenos al desarrollo, para ver cómo se desenvuelven (sin nuestra ayuda) por las diferentes pantallas y opciones.

¿He comprobado las traducciones?

Si nuestra aplicación es multi-idioma debemos comprobar que hemos traducido correctamente todos los textos en cada uno de los idiomas.

3 <http://developer.android.com/resources/articles/backward-compatibility.html>.

También deberemos probar en el emulador cada uno de los idiomas para comprobar que no se rompe el diseño de las pantallas.

¿He colocado los textos legales?

Si nuestra aplicación necesita de algún texto legal deberemos incluirlo (asegurándonos de que esté traducido a todos los idiomas).

También es una buena idea crear una pantalla de “Acerca de” que muestre nuestra información de desarrolladores.

¿Y el sistema de licencias?

Si la aplicación va a ser de pago y queremos tener un sistema anticopia, podemos implementarlo antes de generar el apk.

Una buena idea es usar el que nos provee Google a través de su librería *License Verification Library (LVL)*⁴.

¿He generado el .APK correctamente?

Mientras desarrollamos, cada vez que ejecutamos nuestra código, Eclipse se encarga de generar nuestra aplicación, firmarla con una firma de desarrollo, subirla al terminal (o emulador) correspondiente y la ejecuta.

Para publicar una aplicación debemos hacer algo similar:

1. Generar el fichero .apk. Es una buena idea limpiar el proyecto y compilarlo de nuevo.
2. Firmar el .apk con una firma con datos reales. → Esta firma deberemos almacenarla en un sitio seguro ya que deberemos firmar todas las actualizaciones con la misma firma (sino serán rechazadas).
3. Ejecutar la utilidad “zipaling”. → Optimiza el contenido del paquete resultante para que se ejecute más rápido.

Todos estos pasos se pueden realizar desde una consola mediante comandos, pero existe una forma más fácil realizar esta tarea: usando el asistente de exportación que tenemos en la pestaña “Manifest” de nuestro AndroidManifest.xml.

Se desplegarán una serie de ventanas que nos irán guiando por cada uno de los pasos.

Bien, ya tenemos nuestra aplicación lista para ser subida.

⁴ <http://developer.android.com/guide/publishing/licensing.html>.



3. PUBLICAR EN ANDROID MARKET

Ya hemos comentado que el Android Market es una plataforma abierta de distribución, esto quiere decir que cualquiera puede publicar sus aplicaciones en ella siempre que se cumplan unos mínimos requisitos:

Lo primero es crearnos una cuenta en Google (si no la tenemos ya). Luego solo tenemos que entrar en <http://market.android.com/publish/signup>.

The screenshot shows a web browser displaying the 'Developer Signup' page for the Android Market at market.android.com/publish/signup. The page has a header with the Android Market logo. Below the header, there's a 'Getting Started' section with instructions: 'Before you can publish software on the Android Market, you must do three things:' followed by a bulleted list: '• Create a developer profile.', '• Pay a registration fee (\$35.00) with your credit card (using Google Checkout)', and '• Agree to the [Android Market Developer Distribution Agreement](#)'. Below this is a 'Listing Details' section with fields for 'Developer Name' (a text input field), 'Email Address' (an input field containing 'android@francho.org'), 'Website URL' (an input field containing 'http://'), 'Phone Number' (an input field), and 'Email updates' (a checkbox). At the bottom of the form is a blue 'Continue >' button.

Figura 2. Alta de desarrollador en Android Market.

Deberemos llenar un pequeño formulario en el que se nos piden nuestros datos de desarrollador y abonar una tasa “simbólica” de registro (a diferencia de los Market de otras plataformas, este pago es único y no es necesario renovarlo anualmente).

Una vez completado el registro recibiremos por e-mail una confirmación.

Desde la zona de desarrolladores podremos gestionar nuestras aplicaciones a través de diferentes formularios de una forma muy sencilla: podremos publicar, retirar del mercado, consultar los mensajes de usuarios e informes de bugs...

Tras llenar el formulario de nueva aplicación y adjuntar el apk que hemos generado siguiendo los pasos de la unidad anterior (y si queremos un par de capturas de pantalla que hayamos sacado con el DDMS) pulsaremos el botón de “publish” y en pocos minutos nuestra aplicación estará disponible en miles de teléfonos a lo largo de todo el mundo.

Bien, ya está subida nuestra aplicación... ¿esto es todo?

¡Ni mucho menos! Ahora empieza el trabajo duro: la promoción y el seguimiento.

Conseguir que nuestra aplicación se haga un hueco entre los miles de aplicaciones que hay disponibles en el Market requiere “guerra de guerrillas”. Deberemos publicitar y dar a conocer nuestra aplicación con todos los medios disponibles a nuestro alcance: foros, blogs de temática relacionada, redes sociales, notas de prensa, etc.

Es recomendable crear una página web de la aplicación en la que demos información detallada y vayamos informando de las novedades.

Junto con esta promoción deberemos hacer un seguimiento extra de los comentarios e informes de bug ya que es normal que la mayoría de los fallos aparezcan en los primeros días. Es importante reaccionar rápido a estos informes para conseguir que el problema afecte al menor número de personas posible.



4. MEJORAR NUESTRA APLICACIÓN

Está claro que ninguna aplicación es perfecta y siempre se puede mejorar. Vamos a ver unas pequeñas directivas que te pueden ayudar a hacer que la aplicación vaya a más:

Haz caso a tus usuarios

Presta atención a lo que dicen las personas que utilizan tu aplicación. Suelen mandar comentarios con buenas sugerencias. Por ello, es interesante crear y cuidar los canales de comunicación con ellos. Además de los que nos provee el Market (e-mail y comentarios) podemos crear otros alternativos: Una cuenta Twitter, una página Facebook, un foro de soporte, etc.

Solucioná los bugs

Ya lo hemos mencionado antes, pero no está de más recalcarlo, es muy importante reaccionar con rapidez para que el usuario no se sienta abandonado.

Para el seguimiento de errores podemos usar una aplicación de seguimiento de bugs. Hay muchas disponibles en Internet.

Eclipse dispone de un *plugin* muy interesante en este sentido llamado Mylyn⁵ que se integra con muchas de estas herramientas y nos facilita mucho el seguimiento y corrección de errores.

5 <http://www.eclipse.org/mylyn/>.

Mejora la usabilidad

Colocar un botón en una parte de la pantalla o en otra puede cambiar por completo la experiencia de usuario (a mejor o a peor), por eso debemos tener muy presentes estos temas.

Cuando diseñemos interfaces o flujos deberemos pensar en facilitar las cosas al usuario.

No debemos olvidar que se trata de interfaces táctiles, que el usuario acciona con los dedos, así que deberemos utilizar controles grandes para que sean fáciles e pulsar.

Mejora la velocidad

El tiempo de respuesta es muy importante en las aplicaciones en general, pero es crítico en las aplicaciones de móviles con interface táctil.

Por ejemplo, cuando el usuario mueve el *scroll* de una ventana espera que este sea fluido, no que vaya a trompicones.

En otras palabras, debemos conseguir que el hilo principal de la aplicación sea lo más rápido posible. Para ello, deberemos usar hilos secundarios para las tareas pesadas.

Otra causa común de lentitud son las vistas complejas. Como regla general no es bueno tener más de cinco vistas anidadas. Si se da el caso, se debe intentar otro enfoque (por ejemplo, usando *RelativeLayout*).

Mejora la apariencia

El aspecto estético también es muy importante, es increíble lo que puede cambiar una aplicación colocando, por ejemplo, un fondo gráfico acertado.

Recordemos que podemos modificar el aspecto de todos los componentes (botones, textos, etc.).

También debemos asegurarnos de que nuestra aplicación sigue las directivas de estilo detalladas en la documentación.

Es decir, que los iconos son del tamaño correcto, que usamos los colores adecuados, etc.



Organiza la información

Las pantallas de nuestros terminales son mucho más pequeñas que las de los ordenadores de sobremesa, lo que nos obliga a seleccionar mucho más la información que hay que mostrar al usuario.

Piensa que, antes de hacer malabares para que toda la información entre en una pantalla, es preferible eliminar información superflua o dividirla en varias.

Integra con el sistema

Investiga las diferentes posibilidades que ofrece Android (filtros de *intents*, búsquedas globales, Live Folders, agenda abierta, etc.).

Por ejemplo, en nuestra aplicación del lector de noticias podríamos añadir código para que sus artículos aparecieran en los resultados de las búsquedas globales, o podríamos añadir la opción de compartir llamando a un *Intent* que desplegaría todas las aplicaciones “sociales” disponibles para esa acción, etc.

Todas estas directivas se podrían resumir en dos: piensa en el usuario y presta atención a los detalles



CONCLUSIONES



Destrezas, conocimiento y actitud son los elementos que te ayudarán a conseguir el **triunfo profesional**.

Como hemos visto el Market de Android, a diferencia de los de otras plataformas, es un entorno abierto en el que es muy sencillo subir aplicaciones. Depende de nosotros subir aplicaciones de calidad por ello debemos esforzarnos en cuidar los detalles para conseguir que nuestra aplicación destaque sobre el resto.

Tampoco debemos descuidar el “servicio postventa (o postdescarga)”: debemos estar atentos a los comentarios de los usuarios, a la aparición de posibles *bugs* y a las posibles opciones de mejora que podamos incorporar.

RECAPITULACIÓN

Android Market es una plataforma de distribución muy potente que nos permite distribuir nuestro trabajo de forma barata y sencilla.

Antes de subir una aplicación debemos hacernos las siguientes preguntas:

- ¿La aplicación está terminada y libre de errores?
- ¿Hemos limpiado el código y vaciado los ficheros de datos?
- ¿He actualizado el AndroidManifest.xml definiendo al máximo mi aplicación?
- ¿He comprobado las traducciones?
- ¿He probado la aplicación en el mayor número de escenarios posibles?
- ¿He añadido los textos legales?
- ¿He implementado el sistema de licencia (si lo necesito)?
- ¿He generado y firmado correctamente el fichero .APK?

Una vez publicada, para que nuestra aplicación siga mejorando es conveniente prestar atención a:

- *Feedback* de los usuarios.
- Los posibles *bugs* que puedan aparecer (debemos corregirlos en el menor tiempo posible).
- Mejorar la usabilidad.
- Mejorar la velocidad.
- Mejorar la apariencia.
- Organizar la información.
- Si es posible integrarla con el sistema.



AUTOCOMPROBACIÓN

1. ¿Qué es el Android Market?

- a) Una aplicación para realizar compras por Internet.
- b) Una base de datos de terminales con Android.
- c) Una plataforma de distribución de aplicaciones Android.
- d) El conjunto de terminales Android repartidos por el mundo.

2. ¿De quién depende el Android Market?

- a) De Google.
- b) De Microsoft.
- c) De Apple.
- d) Es libre, no depende de nadie.

3. ¿Quién puede subir aplicaciones al Market?

- a) Cualquier programador.
- b) Solo las empresas.
- c) Solo Google.
- d) Google y los fabricantes de móviles Android.

4. ¿Qué categoría no está incluida en el Market?

- a) Compras.
- b) Estilo de vida.
- c) Referencia.
- d) Adultos.

5. ¿Qué tipo de aplicaciones hay en el Market?

- a) Solo gratuitas.
- b) Solo de pago.
- c) Gratuitas y de pago.
- d) Depende del terminal.

6. ¿Qué datos del desarrollador se muestran en una ficha de aplicación del Market?

- a) Solo el nombre.
- b) Nombre y página web.
- c) Nombre, página web y e-mail.
- d) Es configurable por el programador.

7. ¿Qué información no se muestra en una ficha de aplicación del Market?

- a) Descripción.
- b) Valoraciones.
- c) Comentarios.
- d) Vídeo demostrativo.

8. Una vez subida una aplicación, ¿cuándo estará disponible?

- a) Hay que esperar a que Google nos la apruebe.
- b) Si es gratuita inmediatamente (las de pago deben ser aprobadas por Google).
- c) Hay que esperar 24 horas a que se propague.
- d) A los pocos minutos (sea de pago o gratuita).

**9. ¿Cuál es la afirmación correcta?**

- a) En cuanto tenga decidido el nombre debo subir una aplicación (aunque esté sin terminar) para que nadie me lo coja.
- b) Es mejor subir las aplicaciones con el código completado y libre de errores.
- c) Las aplicaciones con errores son automáticamente rechazadas por el Market.
- d) El Market permite subir aplicaciones incompletas, pero las marca con un ícono y no aparecen en las búsquedas.

10. Cuando decimos que debemos limpiar el código antes de subir la aplicación, ¿a qué nos referiremos?

- a) Indentar el código para que sea más fácil de seguir.
- b) Quitar o comentar todo el código de depuración que hemos colocado.
- c) Quitar todos los comentarios y líneas de código innecesarios para que el paquete pese menos.
- d) Organizar correctamente nuestras clases en paquetes.

11. Si declaro `tag <manifest package="es.masterd.app1">` en mi AndroidManifest, ¿qué indico?

- a) Que el identificador único de mi aplicación es es.masterd.app. Solo podrá haber en el Market una aplicación con ese “package”.
- b) Que el identificador único de mi aplicación es es.masterd.app. En el Market podrá haber varias aplicaciones con ese tag siempre que el nombre sea diferente.
- c) Estoy indicando que el identificador único de mi aplicación es es.masterd.app. En el Market podrá haber varias aplicaciones con ese tag siempre que el nombre sea diferente y sean del mismo autor
- d) Indico que todas las clases de mi aplicación deben pertenecer a ese paquete.

12. Dentro del tag `<uses-sdk>` del AndroidManifest, la etiqueta `android:targetSdkVersion` ¿qué indica?

- a) La versión ideal sobre la que puede correr la aplicación.
- b) La versión del API del SDK a la que está enfocada.
- c) La versión mínima necesaria para ejecutarse.
- d) La versión máxima necesaria para ejecutarse.

13. ¿Qué indica android:minSdkVersion?

- a) La versión ideal sobre la que puede correr la aplicación.
- b) La versión del API del SDK a la que está enfocada.
- c) La versión mínima necesaria para ejecutarse.
- d) La versión máxima necesaria para ejecutarse.

14. Si defino en el manifiesto que la versión mínima para ejecutarse es Android 2.1, subo la aplicación al Market y la busco, ¿qué aparecerá?

- a) Solo aparecerá si consulto el Market desde un móvil con 2.1 o superior.
- b) Aparecerá siempre, pero solo la podrán instalar los que tengan 2.1 o superior.
- c) Aparecerá siempre y todo el mundo podrá instalarla.
- d) Depende del terminal.

15. ¿Qué pasos son necesarios para generar correctamente un .apk?

- a) Solo compilar.
- b) Compilar y firmar.
- c) Compilar, firmar y comprimir con apk.
- d) Compilar, firmar y pasar el comando “zipalign”.

16. Para crear una cuenta en el Android Market ¿qué se necesita?

- a) Solo una cuenta de Google.
- b) Una cuenta de Google y llenar un pequeño formulario.
- c) Una cuenta de Google, llenar un pequeño formulario y abonar la cuota de alta.
- d) Hay que llenar un extenso formulario de varias páginas.

17. ¿Qué métodos son admitidos para comprar en Android Market?

- a) GoogleCheckout.
- b) GoogleVisa.
- c) Paypal.
- d) Cualquier pago electrónico.

**18. Una vez me notifiquen un *bug*, ¿qué debo hacer?**

- a) Debo apuntarlo para que no se me olvide solucionarlo.
- b) Debo intentar solucionarlo en el menor tiempo posible. Investigaré todo lo que sea necesario.
- c) Debo ignorarlo, seguro que es culpa del usuario.
- d) Lo comprobaré en mi terminal, si no lo reproduzco me olvidaré de él.

19. Las vistas ¿pueden ser un cuello de botella para nuestra aplicación?

- a) Sí, si son muy complejas y tienen muchos niveles de anidación
- b) Sí, solo cuando se usan en listas.
- c) No si solo uso RelativeLayout.
- d) No, nunca.

20. Con relación a la retroalimentación de los usuarios, marca la respuesta incorrecta:

- a) Es una buena idea montar canales extra de comunicación (foros, redes sociales, blogs...).
- b) No hay que tener muy en cuenta sus comentarios ya que no conocen la aplicación tan bien como nosotros.
- c) Google nos facilita una vía de *feedback* a través de los comentarios y puntuaciones en las fichas de aplicaciones.
- d) Pueden aportarnos ideas muy interesantes que no hayamos tenido en cuenta.



SOLUCIONARIO

1.	c	2.	a	3.	a	4.	d	5.	c
6.	c	7.	d	8.	d	9.	b	10.	b
11.	a	12.	b	13.	c	14.	a	15.	d
16.	c	17.	a	18.	b	19.	a	20.	b



Destaca entre la multitud; ser un **P8.10** te puede ayudar a llegar muy lejos.

PROPUESTAS DE AMPLIACIÓN

Son muy recomendables las charlas de las conferencias GoogleIO disponibles en Youtube (subtituladas en inglés).

Con ellas podremos mejorar nuestras aplicaciones gracias a las enseñanzas impartidas por los mejores especialistas:

- <http://www.youtube.com/user/GoogleDevelopers>.



BIBLIOGRAFÍA

- Manual Android:

- <http://developer.android.com/guide/publishing/app-signing.html>.
 - <http://developer.android.com/guide/publishing/versioning.html>.
 - <http://developer.android.com/guide/publishing/preparing.html>.
 - <http://developer.android.com/guide/publishing/publishing.html>.

PROGRAMACIÓN PARA ANDROID

14

Ejemplo práctico: lector RSS



ÍNDICE

MOTIVACIÓN.....	3
PROPÓSITOS	4
PREPARACIÓN PARA LA UNIDAD	5
1. PLANTEAMIENTO DE LA APLICACIÓN	7
2. LA BASE DE DATOS.....	9
3. EL PARSER	14
4. ACTIVIDADES	18
5. LAYOUTS	26
6. VALUES	29
7. DRAWABLES	31
8. ANDROIDMANIFEST.XML	32
CONCLUSIONES	33
RECAPITULACIÓN	34
AUTOCOMPROBACIÓN	35
SOLUCIONARIO.....	41
PROPUESTAS DE AMPLIACIÓN	42
BIBLIOGRAFÍA.....	43



MOTIVACIÓN



Dirige tu futuro. En **Master.D**, te acompañamos incondicionalmente.

A lo largo de las unidades anteriores hemos aprendido los principales aspectos de la programación Android: hemos aprendido a manejar proyectos con Eclipse, a crear interfaces de usuario, a acceder a bases de datos, etc.

Ha llegado el momento de ponerlos en práctica en conjunto y para ello vamos a desarrollar desde cero una aplicación: un lector de *feeds*.

PROPOSITOS

- Aplicar de forma práctica los conocimientos adquiridos en temas anteriores: Interface de usuario, uso de listas, bases de datos, proveedores de contenido, etc.
- Programar una aplicación que muestre en nuestro terminal las noticias del blog de Master.D.



PREPARACIÓN PARA LA UNIDAD

Para poder seguir y comprender este capítulo es necesario haber seguido y entendido las unidades anteriores.



1. PLANTEAMIENTO DE LA APLICACIÓN

Como ya hemos mencionado, nuestro objetivo es mostrar las noticias del blog de Master.D (<http://www.blogmasterd.es/>) en nuestro terminal. Utilizaremos para ello su *feed*¹ (<http://www.blogmasterd.es/feed/>).

Veamos grosso modo los principales componentes que va a tener nuestra aplicación:

Un primer módulo de la aplicación deberá encargarse de descargar el *feed* e interpretarlo convirtiéndolo a un formato de datos con el que sea fácil trabajar en el resto de la aplicación. A este módulo lo llamaremos “parser”.

Debido a que las operaciones de red son costosas y a que el contenido del *feed* no se actualiza con demasiada frecuencia, utilizaremos una base de datos para almacenar las noticias. De esta forma además de ganar en agilidad conseguiremos reducir el consumo de datos.

Finalmente, montaremos las actividades necesarias para mostrar la información obtenida. Necesitaremos dos: una para el listado de actividades y otra para el detalle.

Antes de continuar vamos a detenernos un poco en el origen de datos. Los datos obtenidos a través de este sistema están organizados en un fichero xml.

Lo primero que nos encontramos son los nodos referentes a la configuración del *feed*, en el que nos encontramos con datos como el título, la URL de la web, la fecha de actualización, etc.

Luego, por cada noticia tendremos un nodo tipo ítem, que contiene a su vez los datos de la noticia (título, fecha, etc.).

```
<?xml version="1.0" encoding="UTF-8"?>
<rss version="2.0"
  xmlns:content="http://purl.org/rss/1.0/modules/content/"
```

¹ Tal y como dice la Wikipedia: *Una fuente web o canal web (en inglés web feed) es un medio de redifusión de contenido web. Se utiliza para suministrar información actualizada frecuentemente a sus suscriptores.*

```

xmlns:wfw="http://wellformedweb.org/CommentAPI/"
xmlns:dc="http://purl.org/dc/elements/1.1/"
xmlns:atom="http://www.w3.org/2005/Atom"
xmlns:sy="http://purl.org/rss/1.0/modules/syndication/"
xmlns:slash="http://purl.org/rss/1.0/modules/slash/"

<channel>
<title>MasterD blog: formación, oposiciones, cursos. Noticias MasterD</title>
<atom:link href="http://www.blogmasterd.es/feed/" rel="self" type="application/rss+xml" />
<link>http://www.blogmasterd.es</link>
<description>Noticias corporativas sobre MasterD, tu mejor formación a distancia: oposiciones, empleo público, técnicos, profesiones...</description>
<lastBuildDate>Thu, 21 Oct 2010 15:28:37 +0000</lastBuildDate>
<language>en</language>
    <sy:updatePeriod>hourly</sy:updatePeriod>
    <sy:updateFrequency>1</sy:updateFrequency>
        <generator>
            http://wordpress.org/?v=3.0</generator>
        <item>
            <title>Opiniones MasterD también en Teachertube</title>
            <link>http://www.blogmasterd.es/opiniones-masterd-2/opiniones-masterd-tambien-en-teachertube/</link>
            <comments>http://www.blogmasterd.es/opiniones-masterd-2/opiniones-masterd-tambien-en-teachertube/#comments</comments>
                <pubDate>Thu, 21 Oct 2010 15:28:37 +0000</pubDate>
                <dc:creator>MasterD</dc:creator>
                <category><![CDATA[Opiniones MasterD]]></category>
                <category><![CDATA[Youtube]]></category>
                <guid isPermaLink="false">
                    http://www.blogmasterd.es/?p=2250</guid>
                <description><![CDATA[Como muchos ya sabréis, los alumnos son la prioridad numero 1 de MasterD y nos gusta ver su evolución cuando ya han salido al mercado laboral y han finalizado sus estudios con nosotros. Para eso se creó la serie de videos &#8230; <a href="http://www.blogmasterd.es/opiniones-masterd-2/opiniones-masterd-tambien-en-teachertube/">Continue reading <span class="meta-nav">&#8594;</span></a>]]></description>
                <content:encoded><![CDATA
                    [...texto completo...]]></content:encoded>
                <wfw:commentRss>http://www.blogmasterd.es/opiniones-masterd-2/opiniones-masterd-tambien-en-teachertube/feed/
                </wfw:commentRss>
                <slash:comments>0</slash:comments>
            </item>
        <!-- ... resto de items ... -->
    </channel>
</rss>

```

Ejemplo de feed del que obtendremos los datos

Empezaremos diseñando la base de datos que nos va a servir para almacenar esta información una vez procesada.



2. LA BASE DE DATOS

Para nuestra aplicación solo nos interesa quedarnos con la información de los artículos, en concreto de cada uno de ellos necesitaremos:

- Título.
- Enlace.
- Fecha de publicación.
- Texto introductorio.

Para prevenir de futuras mejoras, vamos a añadir también alguna información extra:

- Comentarios.
- Autor.

Vemos, pues, que la estructura de la base de datos no es nada complicada: solo una tabla con seis campos (siete si contamos el campo `_ID` que añadiremos como clave primaria).

Como no queremos que haya registros duplicados definiremos el campo URL como único ya que no puede haber dos `<item>` que apunten a la misma URL.

es.masterd.rss.db.FeedDB

Lo primero que vamos a hacer es crear una clase para abstraer los nombres de tabla y campos. De esta forma podremos referenciarlos luego de forma más sencilla.

```
public class FeedDB {  
    /*  
     * Nombre de la base de datos  
     */  
    public static final String DB_NAME = "juego.db";  
    /*  
     * version de la base de datos  
     */
```

```

        public static final int DB_VERSION = 1;
    /**
     * Esta clase no debe ser instanciada
     */
    private FeedsDB () {}
    /*
     * Definición de la tabla posts
     */
    public static final class Posts implements BaseColumns
    {
    /**
     * Esta clase no debe ser instanciada
     */
    private Posts() {}
    /**
     * orden por defecto
     */
    public static final String DEFAULT_SORT_ORDER =
        "_ID DESC";
    /**
     * Abstracción de los nombres de campos y tabla a constantes
     * para facilitar cambios en la estructura interna de la BD
     */
    public static final String NOMBRE_TABLA = "feeds";
    public static final String _ID = "_id";
    public static final String TITLE = "title";
    public static final String LINK = "link";
    public static final String COMMENTS = "comments";
    public static final String PUB_DATE = "pub_date";
    public static final String CREATOR = "creator";
    public static final String DESCRIPTION = "description";
    public static final String _COUNT = "7";
    }
}

```

Constantes de nuestra base de datos

es.masterd.rss.db.FeedsSQLHelper

Una vez definidas las constantes, crearemos la clase Helper que usaremos para crear la base de datos (si es necesario).

```

public class FeedsSQLHelper extends SQLiteOpenHelper {
    /**
     * Constructor
     * @param context
     */
    public FeedsSQLHelper(Context context) {
        super(context, FeedsDB.DB_NAME, null, FeedsDB.DB_VERSION );
    }
    /**
     * Creación de la base de datos
     */
    @Override
    public void onCreate(SQLiteDatabase db) {

```



```
if(db.isReadOnly()) { db=getWritableDatabase(); }
db.execSQL("CREATE TABLE " +
    FeedsDB.Posts.NOMBRE_TABLA + " (" +
    FeedsDB.Posts._ID + " INTEGER PRIMARY KEY AUTOINCREMENT," +
    FeedsDB.Posts.TITLE + " TEXT," +
    FeedsDB.Posts.LINK + " TEXT UNIQUE," +
    FeedsDB.Posts.COMMENTS + " TEXT," +
    FeedsDB.Posts.PUB_DATE + " INTEGER" +
    FeedsDB.Posts.CREATOR + " TEXT," +
    FeedsDB.Posts.DESCRIPTION + " TEXT" + ") "
);
}
@Override
public void onUpgrade(SQLiteDatabase db, int oldVersion,
int newVersion) {
// no tiene que hacer nada
}
}
```

es.masterd.rss.db.FeedProvider

Finalmente crearemos nuestro proveedor de contenido que será el encargado de introducir y sacar datos en la base de datos.

Le vamos a asignar el espacio de nombres es.masterd.blog y responderá a las URI de tipo es.masterd.blog/post/.

Al crearse hará una llamada al Helper que hemos creado en el paso anterior para que cree la base de datos si es necesario.

```
public class FeedProvider extends ContentProvider {
public static final Uri CONTENT_URI =
Uri.parse("content://es.masterd.blog");
private static final int POST = 1;
private static final int POST_ID = 2;
private static final UriMatcher uriMatcher;
static {
    uriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
    uriMatcher.addURI("es.masterd.blog", "post", POST);
    uriMatcher.addURI("es.masterd.blog", "post/#", POST_ID);
}
private SQLiteDatabase feedsDB;
@Override
public boolean onCreate() {
    Context context = getContext();
    FeedsSQLHelper dbHelper = new FeedsSQLHelper(context);
    feedsDB = dbHelper.getWritableDatabase();
    return (feedsDB == null) ? false : true;
}
}
```

Debemos definir nuestro propio content-type. Crearemos uno para el conjunto de noticias y otro para la noticia sencilla.

```
@Override
public String getType(Uri uri) {
    switch (uriMatcher.match(uri)) {
        // para conjunto de posts
        case POST:
            return "vnd.android.cursor.dir/vnd.masterd.post";
        // para un solo post
        case POST_ID:
            return "vnd.android.cursor.item/vnd.masterd.post";
        default:
            throw new IllegalArgumentException(
                    "Unsupported URI: " + uri);
    }
}
```

Luego ya solo nos queda implementar los métodos de acceso a la base de datos:

```
@Override
public Uri insert(Uri uri, ContentValues values) {
    long rowID = feedsDB.replace(Posts.NOMBRE_TABLA, "", values);
    // si todo ha ido ok devolvemos su Uri
    if (rowID > 0) {
        Uri baseUri =
            Uri.parse("content://es.masterd.blog/post");
        Uri _uri = ContentUris.withAppendedId(baseUri, rowID);
        getContext().getContentResolver().notifyChange(_uri, null);
        getContext().getContentResolver().notifyChange(
            baseUri, null);
        return _uri;
    }
    throw new SQLException("Failed to insert row into " + uri);
}

@Override
public int delete(Uri uri, String where, String[] whereargs) {
    int count = 0;
    switch (uriMatcher.match(uri)) {
        case POST:
            count = feedsDB.delete(Posts.NOMBRE_TABLA, where,
                whereargs);
            break;
        case POST_ID:
            String id = uri.getPathSegments().get(1);
            count = feedsDB.delete(Posts.NOMBRE_TABLA,
                Posts._ID + " = " + id
                + (!TextUtils.isEmpty(where) ? " AND (" + where + ')' : ""),
                whereargs);
            break;
        default:
            throw new IllegalArgumentException("Unknown URI " + uri);
    }
    getContext().getContentResolver().notifyChange(uri, null);
    return count;
}
```



```
@Override
public int update(Uri uri, ContentValues values,
    String selection,
    String[] selectionArgs) {
int count = 0;
switch (uriMatcher.match(uri)) {
case POST:
    count = feedsDB.update(Posts.NOMBRE_TABLA, values,
        selection, selectionArgs);
    break;
case POST_ID:
    count = feedsDB.update(Posts.NOMBRE_TABLA, values,
        Posts._ID
        + " = "
        + uri.getPathSegments().get(1)
        + (!TextUtils.isEmpty(selection) ? " AND (" + selection
        + ')' : ""), selectionArgs);
    break;
default:
    throw new IllegalArgumentException("Unknown URI " + uri);
}
getContext().getContentResolver().notifyChange(uri, null);
return count;
}
```

Después de cada operación notificamos a los objetos que nos están escuchando para que se actualicen.

```
@Override
public Cursor query(Uri uri, String[] projection,
String selection, String[] selectionArgs, String sortOrder)
{
    SQLiteQueryBuilder sqlBuilder = new SQLiteQueryBuilder();
    sqlBuilder.setTables(Posts.NOMBRE_TABLA);
    if (uriMatcher.match(uri) == POST_ID) {
        sqlBuilder.appendWhere(Posts._ID + " = "
            + uri.getPathSegments().get(1));
    }
    if (sortOrder == null || sortOrder == "") {
        sortOrder = Posts.DEFAULT_SORT_ORDER;
    }
    Cursor c = sqlBuilder.query(feedsDB, projection,
        selection,
        selectionArgs, null, null, sortOrder);
    // Registraremos los cambios para que se enteren
    // nuestros observers
    c.setNotificationUri(getContext().getContentResolver(),
        uri);
    return c;
}
```

3. EL PARSER

Para procesar los datos XML vamos a utilizar la librería SAX (*Simple Api for Xml*)² ya que se trata de un api muy ligera, que funciona muy bien para este tipo de xml y que además está incluida en el SDK de Android.

La ventaja de SAX sobre otro sistema de “parseo” (como DOM) es que no necesita almacenar grandes cantidades de datos en memoria, ya que recorre el XML de forma secuencial avisando cada vez que se produce un evento xml: inicio de etiqueta, fin de etiqueta, comentario, etc.

Los datos se analizan en una sola dirección, es decir, una vez analizado no se puede volver hacia atrás.

Para funcionar necesitaremos dos objetos:

- Un objeto SAXParser, que se encargará de recorrer el XML y de llamar a los métodos correspondientes del *handler* cada vez que se produzca un nuevo evento.
- Un objeto que extienda la clase abstracta DefaultHandler que será la encargada procesar los datos conforme se vayan produciendo los eventos.

Nuestro *Handler* va a utilizar una serie de propiedades booleanas que nos servirán para identificar en qué *tag* nos encontramos.

Iremos llenando un objeto de tipo ContentValues y cuando tengamos todos los datos necesarios lo insertaremos en la base de datos.

Veamos como implementamos nuestro *handler*:

² <http://www.saxproject.org/>.



es.masterd.rssparser.RssHandler

Este es el interface inicial que vamos a implementar:

```
public class RssHandler extends DefaultHandler implements  
LexicalHandler {  
    @Override  
    public void comment(char[] ch, int start, int length) throws  
        SAXException {  
    }  
    @Override  
    public void endCDATA() throws SAXException {  
    }  
    @Override  
    public void endDTD() throws SAXException {  
    }  
    @Override  
    public void endEntity(String name) throws SAXException {  
    }  
    @Override  
    public void startCDATA() throws SAXException {  
    }  
    @Override  
    public void startDTD(String name, String publicId,  
        String systemId) throws SAXException {  
    }  
    @Override  
    public void startEntity(String name) throws SAXException {  
    }  
}
```

Vamos a definir los campos que nos ayudarán a saber el nodo actual, también definiremos el proveedor de contenido y el contenedor del registro.

```
public class RssHandler extends DefaultHandler  
implements LexicalHandler {  
    // Donde iremos guardando los datos del registro a guardar  
    ContentValues rssItem;  
    // Flags para saber en que nodo estamos  
    private boolean in_item = false;  
    private boolean in_title = false;  
    private boolean in_link = false;  
    private boolean in_comments = false;  
    private boolean in_pubDate = false;  
    private boolean in_dcCreator = false;  
    private boolean in_description = false;  
    private boolean in_CDATA;  
    // Datos de proveedor de contenidos  
    private ContentResolver contentProv;  
    Uri uri = Uri.parse("content://es.masterd.blog/post");
```

Cada vez que el parser detecte un inicio de tag xml disparará el evento y nuestra función startEntity será llamada. Será el momento de cambiar el indicador de tag correspondiente. También, si detectamos que acabamos de entrar en un tag de noticia inicializaremos el registro

```

@Override
public void startElement(String namespaceURI, String localName,
String qName, Attributes atts) throws SAXException {
// Nos vamos a centrar solo en los items
    if(localName.equalsIgnoreCase("item")) {
        in_item = true;
        rssItem = new ContentValues();
    } else if(localName.equalsIgnoreCase("title")) {
        in_title = true;
    } else if(localName.equalsIgnoreCase("link")) {
        in_link = true;
    } else if(localName.equalsIgnoreCase("comments")) {
        in_comments = true;
    } else if(localName.equalsIgnoreCase("pubDate")) {
        in_pubDate = true;
    } else if(localName.equalsIgnoreCase("dc:creator")) {
        in_dcCreator = true;
    } else if(localName.equalsIgnoreCase("description")) {
        in_description = true;
    }
}

```

De igual manera cuando el *tag* termine, la función *endElement* será llamada. Será el momento de cambiar el indicador correspondiente a *false*. Si además detectamos que es el final de una noticia, deberemos guardarla en la base de datos.

```

@Override
public void endElement(String namespaceURI,
String localName, String qName) throws SAXException {
    if(localName.equalsIgnoreCase("item")) {
        contentProv.insert(uri, rssItem);
        rssItem = new ContentValues();
        in_item = false;
    } else if(localName.equalsIgnoreCase("title")) {
        in_title = false;
    } else if(localName.equalsIgnoreCase("link")) {
        in_link = false;
    } else if(localName.equalsIgnoreCase("comments")) {
        in_comments = false;
    } else if(localName.equalsIgnoreCase("pubDate")) {
        in_pubDate = false;
    } else if(localName.equalsIgnoreCase("dc:creator")) {
        in_dcCreator = false;
    } else if(localName.equalsIgnoreCase("description")) {
        in_description = false;
    }
}

```

El método *characters* se llama cuando se tiene el contenido de un *tag* completo. Nosotros lo usaremos para introducir el valor en el registro. Para saber en qué campo estamos, usaremos los indicadores booleanos.

```

@Override
public void characters(char ch[], int start, int length) {
    if (in_item) { // Estamos dentro de un item
        if (in_title) {
            rssItem.put(FeedsDB.Posts.TITLE, new String
(ch, start, length));
        }
    }
}

```



```
        } else if (in_link) {
            rssItem.put(FeedsDB.Posts.LINK, new String
            (ch, start, length));
        } else if (in_description) {
            rssItem.put(FeedsDB.Posts.DESCRIPTION, new
String(ch, start,
            length));
        } else if (in_pubDate) {
            String strDate = new String(ch, start, length);
            try {
                long fecha = Date.parse(strDate);
                rssItem.put(FeedsDB.Posts.PUB_DATE, fecha);
            } catch (Exception e) {
                Log.d("RssHandler", "Error al parsear la fecha");
            }
        }
    }
}
```

es.masterd.rssparser.RssDownloadHelper

Ya tenemos nuestro *handler*. Ahora vamos a crear una clase de ayuda que nos permita descargar y parsear el fichero de noticias.

Se encargará de crear y todos los objetos SAX necesarios, así como de realizar la llamada al *feed*.

```
public class RssDownloadHelper {
    public static void updateRssData(String rssUrl,
        ContentResolver contentResolver) {
        try {
            URL url = new URL(rssUrl);
            // Obtenemos el SAXParser
            SAXParserFactory spf =
                SAXParserFactory.newInstance();
            SAXParser saxParser = spf.newSAXParser();
            // Definimos el manejador lÃ©xico
            saxParser.setProperty
                ("http://xml.org/sax/properties/lexical-handler",
                rssHandler);
            // Creamos el Handler
            RssHandler rssHandler =
                new RssHandler(contentResolver);
            // Obtenemos el Reader
            XMLReader xr = saxParser.getXMLReader();
            xr.setContentHandler(rssHandler);
            // Parseamos el contenido
            InputSource is =
                new InputSource(url.openStream());
            is.setEncoding("utf-8");
            xr.parse(is);
            // El parseo ha concluido
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
}
```

4. ACTIVIDADES

La actividad principal de nuestra aplicación va a ser la que muestre el listado de noticias a la que llamaremos TitularesActivity. Cuando se pulse sobre una noticia se abrirá una nueva Activity (NoticiaActivity) que mostrará el detalle.

[es.masterd.rss.activities.TitularesActivity](#)

Como hemos mencionado la principal función de esta Activity es mostrar el listado de noticias, así que extenderemos un ListActivity.

Además TitularesActivity dispondrá de un menú que se desplegará al pulsar la tecla menú y que permitirá mostrar la ventana de “Acerca de” (un objeto Dialog) y salir de la aplicación.

Otra de las tareas que debe realizar TitularesActivity es actualizar la base de datos de artículos, si es necesario. Como esta es una tarea pesada, usaremos una clase AsyncTask para ejecutar el proceso en un hilo secundario. La velocidad de la aplicación no se verá afectada. Los cambios se verán reflejados inmediatamente en nuestra lista gracias al uso de un CursorAdapter.

Para informar al usuario de que la descarga se está realizando mostraremos una barra de progreso en el título que desaparecerá una vez termine dicha operación.

En cuanto al *layout* asociado a esta Activity, no podría ser más sencillo: un objeto ListView para mostrar el contenido del listado y otro TextView que se mostrará en el caso de que no haya datos.

Veamos todo esto con detalle:

```
public class TitularesActivity extends ListActivity {
    private static final long FRECUENCIA_ACTUALIZACION =
        60*60*1000; // recarga cada hora
    private static final int DIALOG_ABOUT = 0;
    private ActualizarPostAsyncTask tarea;
```



```
@Override  
public void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    requestWindowFeature(Window.FEATURE_PROGRESS);  
    setContentView(R.layout.feeds);  
    setTitle(R.string.titulo_noticias);  
    configurarAdapter();  
}  
@Override  
public void onResume() {  
    super.onResume();  
    cargarNoticias();  
}  
}
```

Lo primero que hemos hecho es definir dos constantes que usaremos luego, la primera indica el tiempo en milisegundos que tiene que pasar entre recarga y recarga. La segunda sirve para identificar el tipo de dialogo “acerca de”

También hemos definido una propiedad en la que almacenaremos la tarea asíncrona de carga de datos como veremos más adelante.

Al crear la actividad, antes de cargar el *layout* solicitamos que nuestra ventana pueda mostrar barras de progreso en el título. Si esta solicitud se hace una vez cargado el *layout* recibiremos una Excepción que, si no tenemos controlada terminará la ejecución.

Después de cargar nuestra interface de usuario llamaremos a dos métodos propios:

- `configurarAdapter()`, → se encargará de cargar la información de la base de datos en nuestro objeto `ListView`.
- `cargarNoticias()`, → creará y lanzará la tarea asíncrona de actualización.

Carga de datos en el ListView

La función `configurarAdapter()` obtendrá los datos de la base de datos a través de una consulta a nuestro proveedor de contenidos `FeedsProvider` que nos devolverá un objeto de tipo `Cursor`

Para enlazar este `Cursor` con la vista usaremos un `SimpleCursorAdapter`. Se trata de un *adapter* incluido en el SDK al que le pasamos los siguientes parámetros:

- En contexto.
- El id del *layout* a usar para “dibujar” los ítems del listado.
- El cursor de datos.
- Un *array* de cadenas con los nombres de los campos de la respuesta SQL que vamos a usar.

- Un array de enteros con los id de los objetos View (deben estar definidos en el *layout* que pasamos como segundo parámetro) donde se debe colocar cada uno de los campos anteriores.

```

public void configurarAdapter() {
    // Obtenemos todos los artículos de la BD
    final String[] columnas = new String[] { Posts._ID, // 0
        Posts.TITLE, // 1
        Posts.PUB_DATE, // 2
    };
    Uri uri = Uri.parse("content://es.masterd.blog/post");
    Cursor cursor = managedQuery(uri, columnas, null, null,
        Posts.PUB_DATE
        + " DESC");
    // Queremos enterarnos si cambian los datos para
    // recargar -el cursor
    cursor.setNotificationUri(getContentResolver(), uri);
    // Para que la actividad se encargue de manejar el cursor
    // según sus ciclos de vida
    startManagingCursor(cursor);
    // Mapeamos las querys SQL a los campos de las vistas
    String[] camposDb = new String[] { Posts.TITLE,
        Posts.PUB_DATE };
    int[] camposView = new int[] { R.id.feedTitulo,
        R.id.feedFecha };
    // Creamos el adapter
    SimpleCursorAdapter adapter =
        new SimpleCursorAdapter(this, R.layout.feeds_item,
            cursor, camposDb, camposView);
    // La fecha se debe formatear
    final java.text.DateFormat dateFormat =
        DateFormat.getLongDateFormat(TitularesActivity.this);
    adapter.setViewBinder(new ViewBinder() {
        @Override
        public boolean setViewValue(View view, Cursor cursor,
            int columnIndex) {
            if (view.getId() == R.id.feedFecha) {
                long timestamp = cursor.getLong(columnIndex);
                ((TextView)
                    view).setText(dateFormat.format(timestamp));
                return true;
            } else {
                return false; // Que se encargue el adapter
            }
        }
    });
    // Todo listo, cargamos el adapter
    setListAdapter(adapter);
}

```

Como se puede ver, también hemos creado un ViewBinder para modificar el mapeo de campos y que la fecha se muestre formateada.



Carga de noticias

Como hemos dicho, la actualización de la base de datos se realizará en un segundo plano, para ello necesitamos una clase AsyncTask que definiremos como clase Interna:

```
class ActualizarPostAsyncTask extends AsyncTask<Void, Void, Void> {  
    @Override  
    protected void onPreExecute() {  
        setBarraProgresoVisible(true);  
        super.onPreExecute();  
    }  
    @Override  
    protected Void doInBackground(Void... params) {  
        MasterdApplication app = (MasterdApplication)  
            getApplication();  
        RssDownloadHelper.updateRssData(app.getRssUrl(),  
            getContentResolver());  
        return null;  
    }  
    @Override  
    protected void onPostExecute(Void result) {  
        SharedPreferences prefs = getPreferences(MODE_PRIVATE);  
        Editor editor = prefs.edit();  
        editor.putLong("ultima_actualizacion",  
            System.currentTimeMillis());  
        editor.commit();  
        setBarraProgresoVisible(false);  
    }  
    @Override  
    protected void onCancelled() {  
        setBarraProgresoVisible(false);  
        // Se ha cancelado, la próxima vez que arranque volverá a  
        // cargarla  
        SharedPreferences prefs = getPreferences(MODE_PRIVATE);  
        Editor editor = prefs.edit();  
        editor.putLong("ultima_actualizacion", 0);  
        editor.commit();  
        super.onCancelled();  
    }  
}
```

La clase AsyncTask es una clase muy útil para ejecutar tareas en segundo plano y mostrar resultados en el hilo principal.

El método doInBackground es el encargado de realizar la “tarea pesada”, el resto de métodos serán llamados antes de ejecutarse, después de ejecutarse o cancelarse.

Para controlar las distancias entre las recargas vamos a almacenar en una preferencia el instante en el que se ha concluido la última recarga.

El método `cargarNoticias` comprobará esta preferencia antes de crear y lanzar una nueva tarea. Hay que tener en cuenta que los objetos de tipo `AsyncTask` no pueden ser reutilizados, así que deberemos crear uno nuevo cada vez.

```
public void cargarNoticias() {
    SharedPreferences prefs = getPreferences(MODE_PRIVATE);
    long ultima = prefs.getLong("ultima_actualizacion", 0);
    if ((System.currentTimeMillis() - ultima) >
        FRECUENCIA_ACTUALIZACION) {
        tarea = new ActualizarPostAsyncTask();
        tarea.execute();
    }
}
```

Las tareas asíncronas consumen muchos recursos, por eso es buena idea parar la tarea si está en ejecución cuando nuestra Activity pase a segundo plano.

```
@Override
protected void onStop() {
    // Si hay una tarea corriendo en segundo plano, la paramos
    if (tarea != null
        && !tarea.getStatus()
        .equals(AsyncTask.Status.FINISHED)) {
        tarea.cancel(true);
    }
    super.onStop();
}
```

También hemos creado un método auxiliar que nos permiten mostrar y ocultar la barra de progreso.

```
public void setBarraProgresoVisible(boolean visible) {
    final Window window = getWindow();
    if (visible) {
        window.setFeatureInt(Window.FEATURE_PROGRESS,
            Window.PROGRESS_VISIBILITY_ON);
        window.setFeatureInt(Window.FEATURE_PROGRESS,
            Window.PROGRESS_INDETERMINATE_ON);
    } else {
        window.setFeatureInt(Window.FEATURE_PROGRESS,
            Window.PROGRESS_VISIBILITY_OFF);
    }
}
```

Con estos métodos tendríamos completa la funcionalidad básica de mostrar el listado de noticias.



Menú

Las opciones de menú las hemos definido en un fichero xml.

```
<?xml version="1.0" encoding="utf-8"?>
<menu
    xmlns:android="http://schemas.android.com/apk/res/android">
    <item
        android:id="@+id/menuAcercaDe"
        android:title="Acerca de"
        android:icon="@drawable/ic_menu_about" />
    <item
        android:id="@+id/menuQuit"
        android:title="Salir"
        android:icon="@drawable/ic_menu_quit" />
</menu>
```

Figura 1. res/menu/menuprincipal.xml

Este menú lo cargaremos sobrescribiendo el siguiente método de nuestra actividad:

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

También deberemos sobrescribir el método que se disparará cuando el usuario seleccione una opción:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case R.id.menuAcercaDe:
            showDialog(DIALOG_ABOUT);
            return true;
        case R.id.menuQuit:
            finish();
            return true;
        default:
            return super.onOptionsItemSelected(item);
    }
}
```

Para mostrar el diálogo de “Acerca de” sobrescribiremos el método *onCreateDialog*:

```
@Override
protected Dialog onCreateDialog(int id) {
    switch (id) {
        case DIALOG_ABOUT:
            AlertDialog dialogAbout = null;
            final AlertDialog.Builder builder;
            LayoutInflater li = LayoutInflater.from(this);
            View view = li.inflate(R.layout.acercade, null);
            builder = new AlertDialog-
                log.Builder(this).setIcon(R.drawable.icon)
```

```

        setTitle(getString(R.string.app_name))
        setPositiveButton("Ok", null)
        setContentView(view);
        dialogAbout = builder.create();
    return dialogAbout;
default:
    return null;
}
}

```

Ya tenemos nuestro menú operativo, lo único que nos queda es gestionar la selección de una noticia. Al pulsar sobre una se deberá abrir la Activity de detalle a la que pasaremos el id de noticia para ver (campo _ID de la consulta):

```

@Override
protected void onListItemClick(ListView l, View v,
int position, long id) {
    Intent i = new Intent();
    i.setClass(TitularesActivity.this,
NoticiaActivity.class);
    i.putExtra("idNoticia", id);
    startActivity(i);
}

```

es.masterd.rss.activities.NoticiaActivity

Esta actividad es muy sencilla, simplemente deberemos realizar una consulta a nuestro RssContentProvider y mostrar los resultados en los campos de la vista.

El método *onCreate* se encargará de cargar el *layout* y asociar sus vistas a propiedades que luego utilizaremos:

```

public class NoticiaActivity extends Activity {
    private TextView titulo;
    private TextView fecha;
    private WebView contenido;
    /**
     * OnCreate
     */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.noticia);
        setTitle(R.string.titulo_noticias);
        titulo = (TextView) findViewById(R.id.feedTitulo);
        // Al pulsar sobre el título se cerrará la ventana
        titulo.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                finish();
            }
        });
        fecha = (TextView) findViewById(R.id.feedFecha);
        contenido = (WebView) findViewById(R.id.feedContenido);
    }
}

```



Finalmente, en el método `onStart` obtendremos el id que nos han pasado, realizaremos la consulta y mostraremos el resultado:

Veamos ahora el resto de los componentes de la aplicación:

```
protected void onStart() {
    super.onStart();
    try {
        Bundle extras = getIntent().getExtras();
        long idNoticia = extras.getLong("idNoticia");
        final String[] columnas = new String[] { Posts._ID, // 0
            Posts.TITLE, // 1
            Posts.PUB_DATE, // 2
            Posts.DESCRIPTION // 3
        };
        Uri uri = Uri.parse("content://es.masterd.blog/post");
        uri = ContentUris.withAppendedId(uri, idNoticia);
        // Query "managed": la actividad se encargará de cerrar y volver a
        // cargar el cursor cuando sea necesario
        Cursor cursor = managedQuery(uri, columnas, null, null,
            Posts.PUB_DATE + " DESC");
        // Queremos enterarnos si cambian los datos para recargar
        // el cursor
        cursor.setNotificationUri(getContentResolver(), uri);
        // Para que la actividad se encargue de manejar el cursor
        // según sus ciclos de vida
        startManagingCursor(cursor);
        // Mostramos los datos del cursor en la vista
        if(cursor.moveToFirst()) {
            titulo.setText(cursor.getString(1));
            java.text.DateFormat dateFormat =
                DateFormat.getLongDateFormat
                    (NoticiaActivity.this);
            fecha.setText(dateFormat.format(cursor.getLong(2)));
            String texto =
                new String(cursor.getString(3).getBytes(), "utf-8");
            contenido.loadDataWithBaseURL(null,
                texto, "text/html", "UTF-8", null);
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

5. LAYOUTS

Para el Dialog de créditos hemos creado un *layout* muy simple, solo un *TextView* en el que cargaremos el texto correspondiente. Como va a contener enlaces a páginas web y teléfono, le activamos la propiedad “autoLink” y de esta forma al pulsar sobre ellos se abrirán en el navegador o el *dialer*.

```
<?xml version="1.0" encoding="utf-8"?>
<TextView xmlns:
    android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:autoLink="all"
    android:text="@string/acerca_de"
    android:background="#eee"
    android:gravity="center"
/>
```

res/layout/acercade.xml

Para la pantalla de listado necesitamos un *layout* que contenga un objeto *ListView*. Como vamos a utilizarlo dentro de un *ListActivity* utilizaremos Ids ya definidos en el espacio de nombres de “android:” (*list* y *empty*).

La propia *ListActivity* se encargará de mostrar u ocultar uno u otro dependiendo de si hay datos o no.

También hemos modificado pequeños detalles estéticos como los márgenes y los separadores entre ítems.

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



```
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:dividerHeight="4dip"
    android:divider="@android:color/transparent"
    android:layout_marginLeft="5dip"
    android:layout_marginRight="5dip" />
<TextView android:id="@+id/empty"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:text="@string/no_hay_feeds"
    android:gravity="center_vertical|center_horizontal" />
</LinearLayout>
```

res/layout/feeds.xml

Para la lista hemos visto que necesitábamos también un *layout* para dibujar cada uno de los titulares.

Por cada noticia mostraremos el título y su fecha.

También hemos hecho ajustes estéticos como poner un fondo o cambiar los tamaños de los textos.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="vertical"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:background="@drawable/fondo_titular"
        android:layout_margin="5dip"
    >
    <TextView android:id="@+id/feedTitulo"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="-"
        android:gravity="left"
        android:textAppearance="?android:attr/textAppearanceMedium"
    />
    <TextView android:id="@+id/feedFecha"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="-"
        android:gravity="right"
        android:textAppearance="?android:attr/textAppearanceSmall"
        android:textColor="#777"
    />
</LinearLayout>
```

res/layout/feeds_item.xml

Finalmente, para mostrar la noticia en detalle crearemos usaremos un *layout* con tres componentes principales: dos *TextView* (para título y fecha) y un *WebView* para la descripción.

Al usar un WebView, el código HTML que contenga la descripción será “renderizado” correctamente.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    android:orientation="vertical"
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_margin="15dip"
    android:background="@drawable/fondo_titular">
    <TextView android:id="@+id/feedTitulo"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="-"
        android:gravity="left|top"
        style="@style/TituloNaranja"
        android:drawableRight="@drawable/close" />
    <TextView android:id="@+id/feedFecha"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="-"
        android:gravity="right"
        android:textAppearance="?android:attr/textAppearanceSmall"
        android:textColor="#777" />
    <WebView android:id="@+id/feedContenido"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="-"
        android:textColor="#777" />
</LinearLayout>
```

res/layout/noticia.xml

Este *layout* va a ser mostrado como ventana flotante que se cerrará cuando el usuario pulse sobre el título. Para indicar esto le hemos añadido un ícono a la derecha del texto de título.

También hemos ajustado diferentes aspectos estéticos.



6. VALUES

Aunque esta aplicación solo está enfocada al idioma español, todas las cadenas de texto las hemos definido en un fichero de *resources*. De esta forma si se quisiera traducir el interface, solo tendríamos que crear un fichero similar con las traducciones.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">Master.D</string>
    <string name="no_hay_feeds">No hay artículos.</string>
    <string name="titulo_noticias">Noticias</string>
    <string name="acerca_de"><b>Curso Android - Tema 14</b>\n
    \n
    (c) 2010 Master.D\n
    Teléfono: 902404140\n\n
    www.master-d.es</string>
</resources>
```

res/values/strings.xml

Un detalle interesante es que para el ítem de “acerca_de” hemos utilizado código HTML básico (**** para mostrar en negrita) y saltos de línea. Además como este texto se va a mostrar en un objeto View con la propiedad autoLink activado, el sistema se encargará de convertir la dirección web y el teléfono en enlaces que activarán el navegador web o el *dialer* respectivamente.

También hemos definido dos temas: uno para las ventanas “normales” (listado) y otro para las ventanas flotantes (noticia).

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <!-- Tema -->
    <style name="MasterD" parent="@android:style/Theme.Light">
        <item name="android:windowTitleStyle">
            @style/Titulo</item>
        <item name="android:windowTitleBackgroundStyle">
            @style/FondoTitulo</item>
```

```

<item name="android:buttonStyle">
    @style/MiBoton</item>
</style>
<!-- Tema ventana flotante -->
<style name="MasterD.Dialog">
    <item name="android:windowIsFloating">true</item>
    <item name="android:windowFrame">@null</item>
    <item name="android:windowContentOverlay">@null</item>
    <item name="android:windowAnimationStyle">
        @android:style/Animation.Dialog</item>
    <item name="android:windowBackground">
        @android:color/transparent</item>
    <item name="android:windowNoTitle">true</item>
</style>
</resources>

```

res/values/themes.xml

Para el diseño nos hemos basado en los colores usados en la web.

```

<?xml version="1.0" encoding="utf-8"?>
<resources>
<style name=
"Titulo" parent="@android:style/TextAppearance.Small">
    <item name="android:textStyle">bold</item>
    <item name="android:textColor">#777777</item>
    <item name="android:gravity">left</item>
    <item name="android:paddingLeft">3dip</item>
</style>
<style name="FondoTitulo">
    <item name="android:background">@drawable/fondo_titulo
    </item>
</style>
<style name="MiBoton" parent="@android:style/Widget.Button">
    <item name="android:textAppearance">
        ?android:attr/textAppearanceSmall
    </item>
    <item name="android:textColor">#afa</item>
</style>
<style name=
"TituloNaranja" parent="@android:style/TextAppearance.Medium">
    <item name="android:textColor">#F56200</item>
    <item name="android:textStyle">bold</item>
</style>
</resources>

```

res/values/styles.xml

Los estilos usados en nuestros temas y *layouts*.



7. DRAWABLES

Para el fondo de los titulares y la noticia hemos creado un objeto *drawable XML* que muestra un degradado con las esquinas redondeadas.

```
<?xml version="1.0" encoding="utf-8"?>
<shape
    xmlns:android="http://schemas.android.com/apk/res/android">
    <gradient
        android:startColor="#FBFBFB"
        android:endColor="#E3E3E3"
        android:centerY="0.9"
        android:angle="270" />
    <corners android:radius="4dp" />
    <padding android:left="7dp"
        android:top="7dp"
        android:right="7dp"
        android:bottom="7dp" />
    <stroke
        android:width="1dp"
        android:color="#BFBFBF"
    />
</shape>
```

El resto son imágenes png que usaremos como iconos de menú o fondo del título.

El ícono de la aplicación para la que, siguiendo la guía de estilo de la documentación Android, es e

Para los iconos y el fondo del título hemos creado tres versiones (una por cada resolución) que hemos colocado en su correspondiente carpeta de *drawable*.

8. ANDROIDMANIFEST.XML

Solo nos queda un último paso: incluir todos los componentes en el manifiesto.

Tenemos que definir las dos actividades y el proveedor de contenidos. También hemos solicitado al usuario permiso para usar Internet.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
    xmlns:android="http://schemas.android.com/apk/res/android"
        package="es.masterd.rss"
        android:versionCode="1"
        android:versionName="1.0">
    <application
        android:icon="@drawable/icon"
        android:label="@string/app_name"
        android:name="MasterdApplication"
        android:theme="@style/MasterD">
        <activity android:name=".activities.TitularesActivity"
            android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity android:name=".activities.NoticiaActivity"
            android:theme="@style/MasterD.Dialog" />
        <provider android:name=".db_FEEDS.FeedProvider"
            android:authorities="es.masterd.blog" />
    </application>
    <uses-permission android:name="android.permission.INTERNET" />
</manifest>
```



CONCLUSIONES



Distinción es igual a triunfo profesional.

Vemos cómo, gracias al SDK de Android, con muy poco código hemos creado una aplicación totalmente funcional y estéticamente atractiva.

El uso de la base de datos la hace rápida y funcional ya que podemos consultar los titulares, incluso sin conexión de red (si estos ya han sido cargados).



Figura 2. Aspecto final de nuestra aplicación

RECAPITULACIÓN

Los pasos que hemos seguido para hacer esta aplicación son:

1. Hemos analizado el problema y los datos iniciales para encontrar el enfoque adecuado.
2. Hemos creado la base de datos necesaria para almacenar los datos que nos interesan.
3. Hemos creado un proveedor de contenido para facilitar el acceso a la base de datos.
4. Hemos creado el paquete necesario para descargar los datos, interpretarlos e insertarlos en la base de datos.
5. Hemos creado un ActivityList para mostrar los datos en un listado.
6. También hemos hecho otra Activity para mostrar el detalle de la noticia.
7. Hemos añadido el menú a la actividad de resultado.
8. Hemos mejorado el aspecto definiendo un estilo y los archivos gráficos necesarios.
9. Hemos definido todos los componentes necesarios en el AndroidManifest.
10. Finalmente, hemos probado que todo funciona correctamente.



AUTOCOMPROBACIÓN

- 1. ¿Qué clase es la encargada de abstraer los nombres de los campos de la tabla de la base de datos?**
 - a) FeedsDB.
 - b) FeedsDB.Posts.
 - c) FeedsSQLHelper.
 - d) FeedsProvider.

- 2. ¿Dónde definimos el nombre de la base de datos?**
 - a) FeedsDB.
 - b) FeedsDB.Posts.
 - c) FeedsSQLHelper.
 - d) FeedsProvider.

- 3. ¿Qué clase es la encargada de crear la base de datos?**
 - a) FeedsDB.
 - b) FeedsDB.Posts.
 - c) FeedsSQLHelper.
 - d) FeedsProvider.

4. ¿Qué clase es la encargada de facilitar el acceso a la base de datos?

- a) FeedsDB.
- b) FeedsDB.Posts.
- c) FeedsSQLHelper.
- d) FeedsProvider.

5. ¿Qué campo usamos como clave primaria en la tabla feeds?

- a) _id.
- b) link.
- c) title.
- d) No hemos definido ninguno.

6. ¿Por qué hemos elegido el nombre_id para el campo primario de la tabla feeds?

- a) Para facilitar su uso en un ContentProvider (Android usa ese nombre por convención).
- b) Para poder pasarlo como parámetro.
- c) Por decisión propia (podríamos haber elegido cualquier otro).
- d) No hemos definido ninguno.

7. ¿Para qué usamos la constante FeedsDB.DB_VERSION?

- a) Para controlar la última vez que hemos actualizado las noticias.
- b) Para saber si se debe actualizar la estructura de la base de datos.
- c) Para saber si la base de datos está creada.
- d) No se usa.

8. ¿Qué URL gestiona nuestro proveedor de contenido FeedsProvider?

- a) content://es.masterd.blog/post/.
- b) content://es.masterd/post/.
- c) content://masterd/post/.
- d) content://data.masterd/post.



- 9. ¿Qué *mime type* está asociado a un conjunto de noticias en nuestro proveedor de contenido?**
 - a) vnd.android.cursor/post.
 - b) vnd.android.cursor/noticias.
 - c) vnd.android.cursor.dir/vnd.masterd.post.
 - d) vnd.android.cursor.dir/vnd.masterd.noticias.
- 10. ¿Qué tipo de datos devolverá el método *query* del proveedor de contenidos *FeedsProvider*?**
 - a) Noticia.
 - b) Noticias.
 - c) Cursor.
 - d) Object.
- 11. ¿Por qué hemos elegido la librería SAX para interpretar el fichero XML?**
 - a) Necesita menos recursos que otras.
 - b) Es la única que podíamos usar.
 - c) Es la única que se integra con la clase ContentProvider.
 - d) No hemos usado SAX.
- 12. ¿Cómo funciona SAX?**
 - a) Recorre el archivo disparando eventos cada vez que detecta un cambio en el xml. Solo recorre el xml una vez.
 - b) Recorre el archivo disparando eventos cada vez que detecta un cambio en el xml. Lo hace recorriendo el xml varias veces.
 - c) Recorre el archivo una vez, generando una estructura en memoria en la que va cargando los datos.
 - d) Recorre el archivo varias veces, generando una estructura en memoria en la que va cargando los datos.
- 13. ¿Qué clase es la encargada de interpretar los eventos de SAX y llamar al RssContentProvider para guardar los datos?**
 - a) DefaultHandler.
 - b) RssHandler.
 - c) RssDownloadHelper.
 - d) SaxParser.

14. ¿Qué método de nuestro *handler SAX* recibe como parámetro los datos que hay entre el inicio y fin de un *tag xml*?
- a) startElement.
 - b) endElement.
 - c) Characters.
 - d) SaxParser.
15. ¿Qué clase extiende TitularesActivity?
- a) Activity.
 - b) ListActivity.
 - c) MapActivity.
 - d) CursorActivity.
16. ¿Qué clase hemos utilizado para descargar los datos en un hilo diferente?
- a) Thread.
 - b) Handler.
 - c) AsyncTask.
 - d) Ninguna de las anteriores.
17. ¿Qué tendremos que editar si queremos cambiar el texto de la ventana “acerca de”?
- a) La clase TitularesActivity.
 - b) La clase AcercaDeActivity.
 - c) El layout acercade.xml.
 - d) El archivo res/values/strings.xml.
18. ¿Qué tipo de Adapter hemos utilizado para mostrar los listados de noticias?
- a) ArrayAdapter.
 - b) SimpleArrayAdapter.
 - c) CursorAdapter.
 - d) SimpleCursorAdapter.



19. ¿Qué utilizamos para evitar que los datos se recarguen cada vez que se inicie la aplicación?

- a) Guardamos la última vez que se actualizó en un Shared Preferences.
- b) Guardamos la última vez que se actualizó en la base de datos.
- c) Guardamos la última vez que se actualizó en un fichero.
- d) Guardamos la última vez que se actualizó en una variable estática.

20. ¿Qué permiso necesita declarar nuestra aplicación para funcionar?

- a) android.permission.DB.
- b) android.permission.INTERNET.
- c) android.permission.WRITE.
- d) No necesita ninguno.



SOLUCIONARIO

1.	b	2.	a	3.	c	4.	d	5.	a
6.	a	7.	b	8.	a	9.	c	10.	c
11.	a	12.	a	13.	b	14.	c	15.	b
16.	c	17.	d	18.	d	19.	a	20.	b



Descubre hasta dónde te puede llevar tu esfuerzo. Convírtete en un P8.10.

PROPUESTAS DE AMPLIACIÓN

Ya tienes los conocimientos básicos necesarios para empezar a realizar sus propias aplicaciones Android. Está claro que la mejor forma de ampliar estos conocimientos es programando.

Y la mejor forma de motivarse para programar es desarrollar una aplicación que te motive. Para empezar elige algo que no sea excesivamente complejo (por lo menos en su primera versión), diseñalo, desarrollalo, pruébalo, súbelo al Market y promocionalo como hemos aprendido en esta unidad.

Otra buena forma de aprender es viendo el código fuente de otros. Existen un montón de proyectos OpenSource basados en Android que nos permiten ver código de otros desarrolladores.



BIBLIOGRAFÍA

- Android Developer:
[http://developer.android.com/.](http://developer.android.com/)
- Librería SAX:
[http://www.saxproject.org/.](http://www.saxproject.org/)

