bemoob
make your phone smarter

CURSO DE PROGRAMACIÓN EN ANDROID PARA PRINCIPIANTES

FAQSAndroid.com

INTRODUCCIÓN

Android es hoy por hoy uno de los sistemas operativos con más oportunidades, donde un desarrollador puede crear aplicaciones que sean conocidas a nivel mundial

Durante 10 semanas hemos estado publicando en **FAQsAndroid** un curso en el que aprender a hacer una aplicación desde cero. Ha sido realizado por **Robert P.**, uno de los editores del blog que tiene a sus espaldas ya algunas aplicaciones, como HD Contact Photos.

El curso ha sido recopilado en este PDF completamente gratuito para tenerlo a mano en un solo lugar.

También hay a disposición de quien quiera usarlo un **HILO OFICIAL** del curso en Foromóviles.com:

http://foromoviles.com/threads/479-Curso-de-programación-android

ÍNDICE

CAPÍTULO 1

Instalación del Android SDK y Eclipse

Cada vez que Google publica una nueva versión de su sistema operativo, y mucho antes de que ésta llegue a cualquiera de sus teléfonos de forma oficial, los de Mountain View publican el **SDK** (Software Development Kit) del sistema, que contiene una imagen de éste y todos sus programas, así como del framework de Android, y que sirve a los desarrolladores para adaptar sus aplicaciones a la nueva versión antes de que ésta llegue al gran público.

Para poder gestionar todo este sistema de versiones y subversiones en un sistema operativo *vivo*, como es el caso de Android, resulta necesario un software que se encargue tanto de la sincronización de los paquetes e imágenes (denominamos imagen a una copia completa de una partición) ya instaladas, como de la instalación de los nuevos de forma lo más transparente posible.

En este capítulo aprenderemos a instalar el gestor de paquetes SDK de Android, fundamental para poder desarrollar aplicaciones para este sistema, así como el entorno de desarrollo de aplicaciones Eclipse, que si bien no es el más sencillo de utilizar, sí que es el más completo y utilizado por desarrolladores de múltiples plataformas.

Gestor de paquetes SDK

Para ello Google proporciona el **SDK Manager**, un programa que se sincroniza con los servidores de Google y que nos informa tanto de la disponibilidad de nuevos paquetes como del estado de los que ya tenemos.

La aplicación está disponible tanto para entornos Windows como MAC OS X y Linux, si bien *en este tutorial nos centraremos en la versión para sistemas Microsoft Windows*.

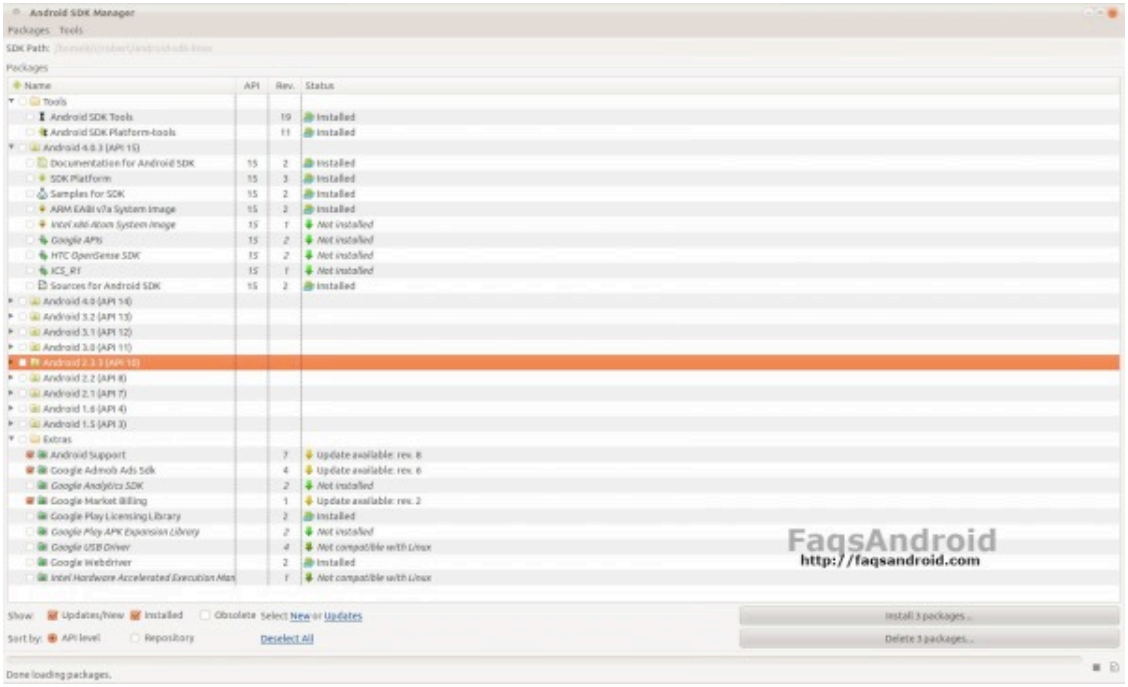
En el caso de Windows la instalación de la aplicación puede realizarse mediante **un archivo comprimido, en formato zip**, que deberemos descomprimir en el directorio raíz del sistema (habitualmente *c:*\) o bien mediante un instalador, lo que minimiza no sólo el tiempo ocupado en el proceso, sino también las probabilidades de error (en caso que optemos por el instalador, es conveniente indicar el directorio *c:\android-sdk-windows* como directorio de instalación cuando el sistema nos lo pregunte).

Una vez realizada la instalación deberemos ejecutar el programa **SDK Manager.exe** que, al abrirlo por primera vez, procederá a descargar e instalar el resto de archivos necesarios para la aplicación y que no se incluyen en la descarga inicial, para lo cual necesitará acceder a Internet.

Una vez que el gestor de SDK esté completamente instalado, procederemos a instalar las versiones del SDK que necesitemos, dependiendo de las versiones del sistema para las cuales vayamos a desarrollar (en nuestro caso se recomienda la instalación de la versión 15, correspondiente a Android 4.0.3).

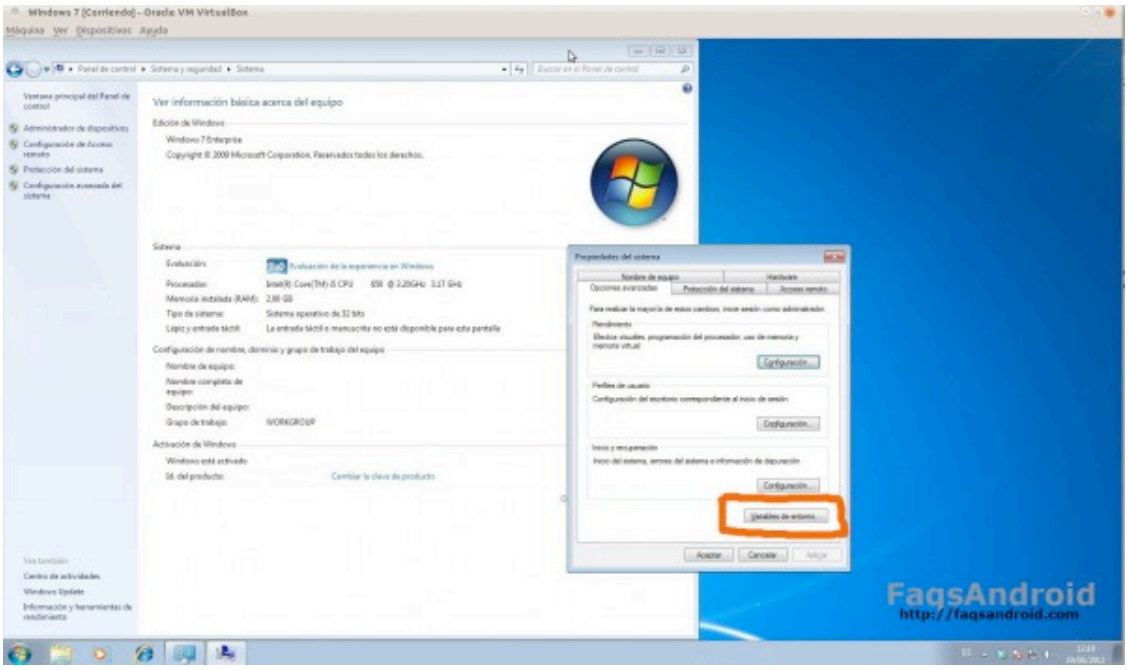
Es importante que iniciemos el gestor de paquetes varias veces, instalando las actualizaciones que hubiera, hasta que el sistema gestor indique que no hay actualizaciones disponibles.

Asimismo, en determinados ordenadores, dependiendo de la configuración de red, firewalls, proxys, etc puede ocurrir que no se sea posible establecer una conexión segura (https), en cuyo caso es posible forzar la descarga a través de una conexión insegura (http) accediendo a la configuración del gestor (*Tools → Options*) y marcando la casilla "*forze https sources to be fetched using http*".



Ajustes manuales

En algunos sistemas Windows es necesario definir una variable de entorno denominada *ANDROID_SDK_HOME*, cuyo contenido ha de ser la ruta completa al directorio de instalación del *SDK*, para que todo funcione correctamente.



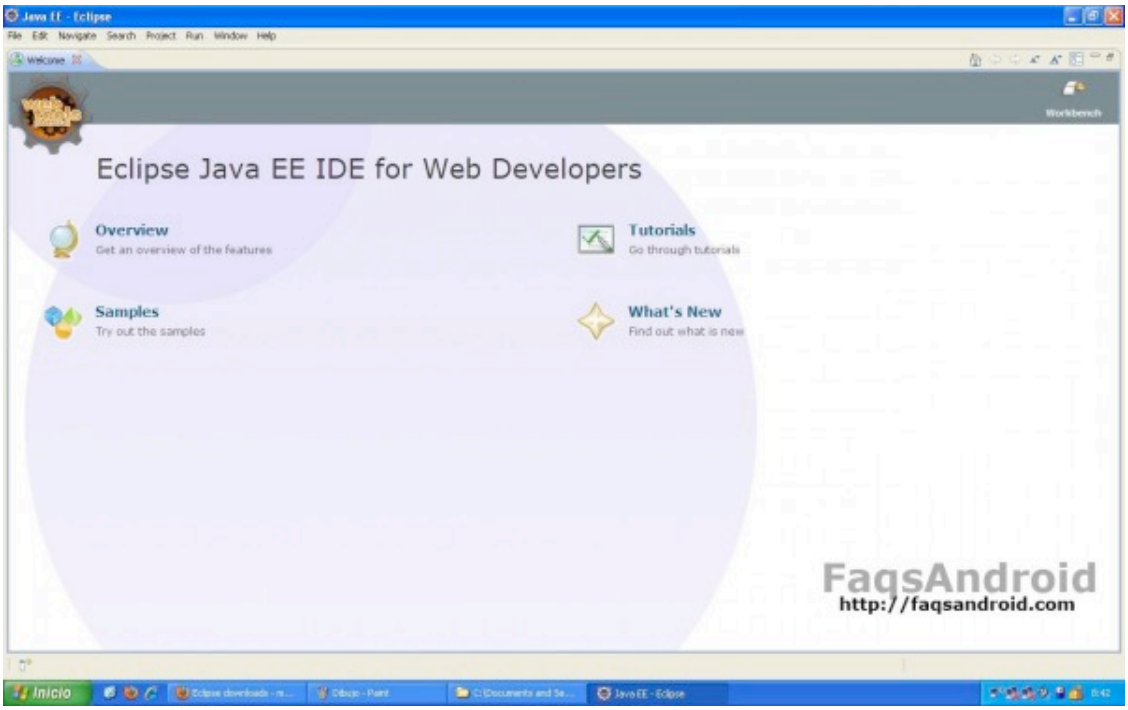
De igual manera, en los sistemas UNIX, deben añadirse los directorios [directorio-instalación]/platform-tools y [directorio-instalación]/tools al path de sistema (variable de entorno *PATH*).

Instalación de Eclipse

Eclipse es un entorno de desarrollo de aplicaciones muy potente y que dispone de plugins para la mayoría de los lenguajes de programación, entre ellos Android; que de hecho es un subconjunto de un lenguaje mayor denominado Java, que fue desarrollado originariamente por Sun y adaptado posteriormente por Google.

La instalación de Eclipse es muy sencilla **y consiste en descomprimir el contenido de un archivo zip**, que contiene la aplicación y las librerías, en el directorio de nuestra elección (recomendamos *c:\eclipse*).

Una vez instalado, procederemos a acceder al mismo, haciendo doble clic en el icono de la aplicación, que se encuentra en el directorio que hayamos creado para la misma (si lo deseamos podemos crear un acceso directo al *escritorio* de la forma habitual).



La primera vez que accedamos nos preguntará por el directorio de trabajo, que será aquel en el que se ubicarán todos los programas que desarrollemos. Por comodidad, resulta conveniente marcar la casilla que indica al sistema que nuestra elección es definitiva, lo cual impedirá que, en próximas ejecuciones, se nos vuelva a preguntar por dicho directorio.

Plugin ADT para Eclipse

Como hemos indicado, Eclipse es un entorno de desarrollo muy potente disponible para gran cantidad de lenguajes, a través de los plugins correspondientes.

El *ADT* (Android Development Tools) es el plugin desarrollado por Google para permitir la integración del SDK de Android en Eclipse, permitiendo el desarrollo de programas Android de forma nativa desde Eclipse.

La instalación se realiza manualmente desde el propio Eclipse siguiendo los siguientes pasos:

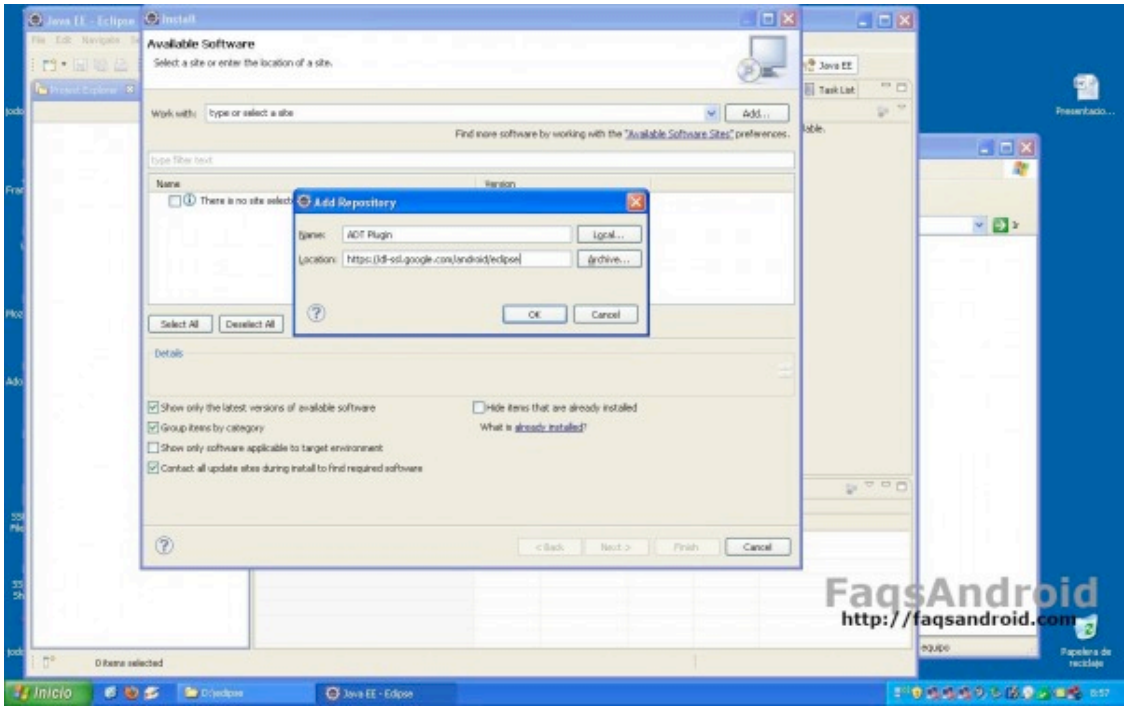
1. Seleccionar *Help → Install new software*,
2. Hacer clic en *add*, en la parte superior derecha de la ventana,
3. En el diálogo indicar "ADT Plugin" como *nombre* y https://dl-ssl.google.com/android/eclipse como *localización*,
4. Hacer clic en *ok*,
5. En la ventana que aparezca, marcar la casilla junto a *Developer Tools* y hacer clic en *next*,
6. Hacer clic en el botón *next* (cuando aparezca),
7. Leer y aceptar el acuerdo de licencia y hacer clic en *finish*,

8. Al acabar la instalación, y antes de realizar los pasos siguientes, reiniciar eclipse.

Al reiniciar Eclipse, aparecerá una ventana en la que nos indicará que, para desarrollar en Android es necesario instalar el SDK. Haremos clic en la opción "*Use existing SDKs*" e introduciremos la ruta del directorio de instalación que corresponda (comúnmente *c:\android-sdk-windows*).

Sólo en caso que no apareciera el diálogo indicado en el apartado anterior, deberemos enlazar el ADT con el gestor de paquetes SDK de Android manualmente, para lo cual realizaremos los siguientes pasos (en Eclipse):

1. Seleccionar *Window → Preferences*,
2. Seleccionar *Android* en el panel de la izquierda,
3. Junto al cuadro *SDK Location*, hacer clic en *Browse* y navegar hasta seleccionar el directorio en el que se ha instalado el gestor de SDK (comúnmente *c:\android-sdk-windows*),
4. Hacer clic en *apply* y después en *ok*.

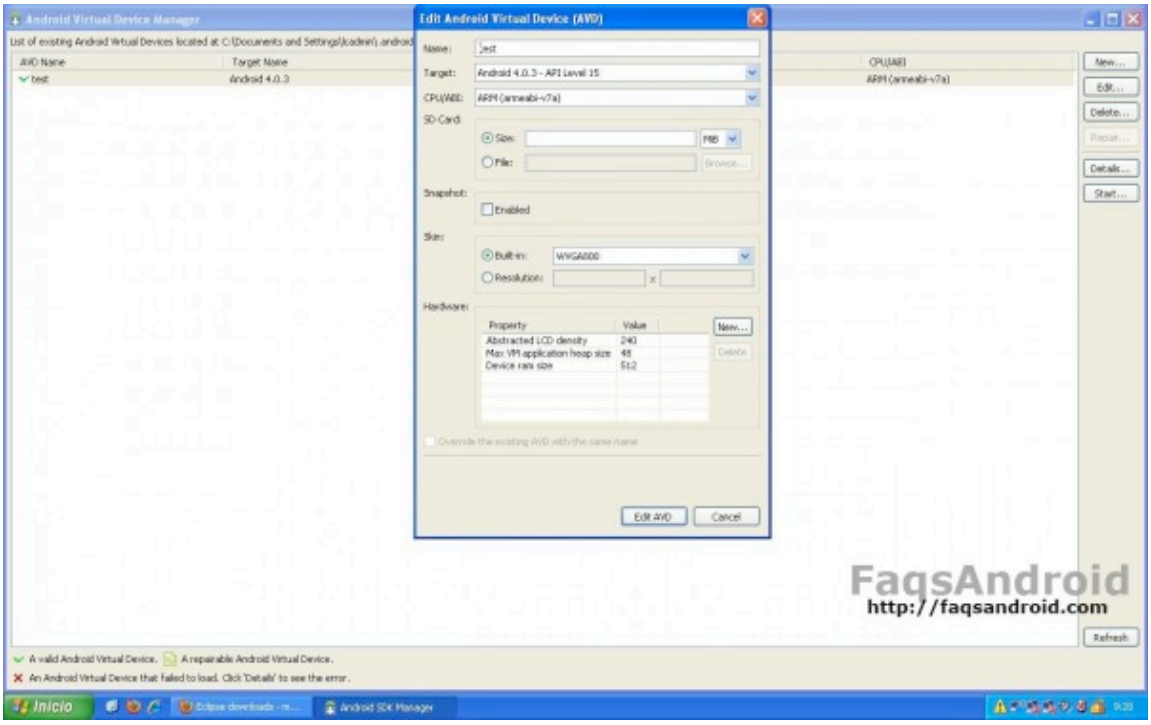


Creación de una máquina virtual

El gestor de paquetes SDK incluye un **emulador Android** que podremos ejecutar para probar nuestras aplicaciones.

De hecho, **el gestor se integra tan bien con Eclipse que podremos incluso debugar** las aplicaciones, generar puntos de parada, comprobar los valores de las diferentes variables, etc.

Para crear la máquina virtual correspondiente a una versión concreta de Android deberemos abrir el gestor de paquetes SDK y acceder a *Manage AVDs*, dentro del menú *tools*.



Para crear una instancia del emulador haremos clic en el botón *new* en la ventana de gestión de máquinas virtuales, procediendo a rellenar los campos correspondientes del formulario con los valores que sigue:

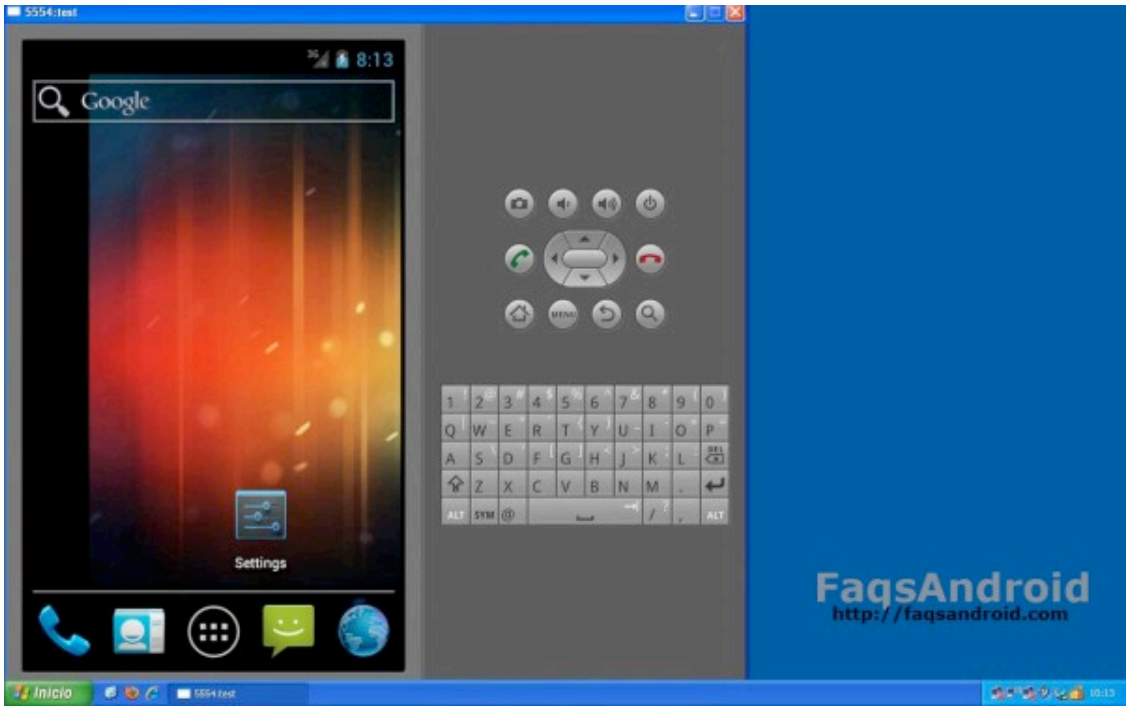
- Name: ICS
- Target: Android 4.0.3 – API Level 15
- CPU/ABI: ARM (armabi-v7a)
- SD Card: Size 32 mb

También será necesario añadir soporte para la tarjeta de memoria, para lo cual haremos clic en el botón *new* en el apartado correspondiente a hardware del formulario, procediendo a seleccionar la propiedad *SD Card Support*, tras lo cual haremos clic en *ok* y posteriormente en *create AVD*.

Android en tu PC

Una vez creado el emulador podemos ejecutarlo, para lo cual lo haremos clic en su nombre (en la pantalla de gestión de máquinas virtuales, y posteriormente seleccionaremos *start*.

Es normal que el emulador tarde unos minutos en arrancar, dependiendo de la potencia del ordenador en el que lo ejecutemos.



En el próximo capítulo hablaremos de las diferentes partes de una aplicación Android y crearemos nuestra primera aplicación de prueba, que ejecutaremos en el emulador.

CAPÍTULO 2

Esqueleto de una aplicación Android

La estructura de las aplicaciones en Android se orienta claramente al usuario, o quizás sería más exacto decir a su dispositivo.

En cualquier caso, usuario y dispositivo se expresan en una lengua, y este último, además, tiene unas características físicas determinadas y ejecuta una u otra versión del sistema.

Así, como veremos a continuación, **cuando creemos una aplicación deberemos tener en cuenta las características de los dispositivos a los que se dirige, tanto en lo relacionado con el hardware como con la versión de Android que lo controla.**

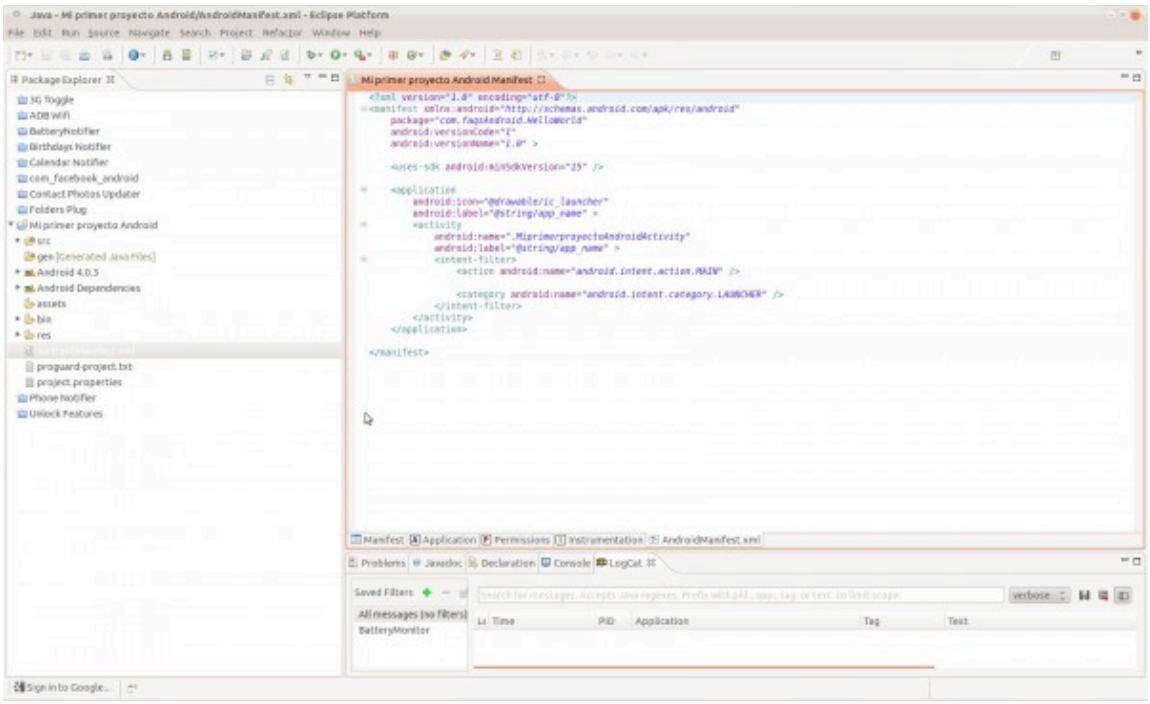
El manifiesto

El manifiesto (archivo *manifest.xml* en cualquier proyecto Android) **es un archivo en formato XML** (*eXtensible Markup Language*) **en el que se definen las características generales de una aplicación**, a saber:

- El paquete: Es una cadena que describe de forma única a una aplicación, no siendo posible añadir una aplicación en la Play Store si ya existe otra con el mismo nombre de paquete. De igual manera, si instalamos en nuestro dispositivo una app con el mismo nombre (de paquete) que otra ya existente, la nueva sustituirá a la anterior.
- El nombre: Es el nombre que aparecerá en la tienda, o el nombre que nos enseña el instalador cuando nos disponemos a instalar la aplicación.
- La versión: El sistema define dos números diferentes para la versión de una aplicación: por un lado el número de compilación, que debe incrementarse en sucesivas actualizaciones (no es posible actualizar en la tienda una app con un número de compilación inferior al de la versión existente). Y por otro el número de la versión, que es el número que se mostrará al usuario y sobre el que no hay restricciones.
- La versión de Android a la que se dirige: Se indica en este punto la versión (mínima y máxima) necesaria para que la aplicación se ejecute correctamente,

no siendo posible instalar una aplicación en un dispositivo que tenga una versión de Android que no se encuentre en el rango especificado.

- Los permisos: Lista de permisos que necesita la aplicación para ejecutarse correctamente, y que se le presentarán al usuario cuando instale la aplicación. Tal y como hemos indicado en alguna ocasión, **Android no soporta aún que el usuario seleccione qué permisos desea otorgar, de entre la lista de permisos solicitados**, así que éste deberá aceptarlos todos para poder instalar la aplicación.
- Lista de actividades, servicios y receptores de mensajes: Enumeraremos en este apartado las actividades (ventanas) de nuestra aplicación, así como los servicios que ésta ofrecerá y los procesos de recepción de mensajes, así como los parámetros que activarán cada uno de ellos, si los hubiera.



Ejecutar un programa

E**n Android los programas no sólo se ejecutan cuando el usuario los invoca**, haciendo clic en el icono correspondiente en el Drawer, por ejemplo; **sino que éstos también se activan automáticamente dependiendo del estado del sistema.**

Como veremos en las próximas sesiones, es posible indicar que una aplicación, o un servicio ofrecido por ésta, debe ejecutarse automáticamente cuando el dispositivo se inicie, o cuando se reciba una llamada o un mensaje, por ejemplo.

Idiomas

Al principio de este artículo nos referíamos al idioma del dispositivo; pues bien, **en Android podemos personalizar la lengua en la que se muestra nuestra aplicación de forma transparente tanto al usuario como a la propia aplicación.**

Para ello, lógicamente, deberemos incluir en nuestra aplicación las traducciones de las diferentes cadenas que visualizamos en ésta.

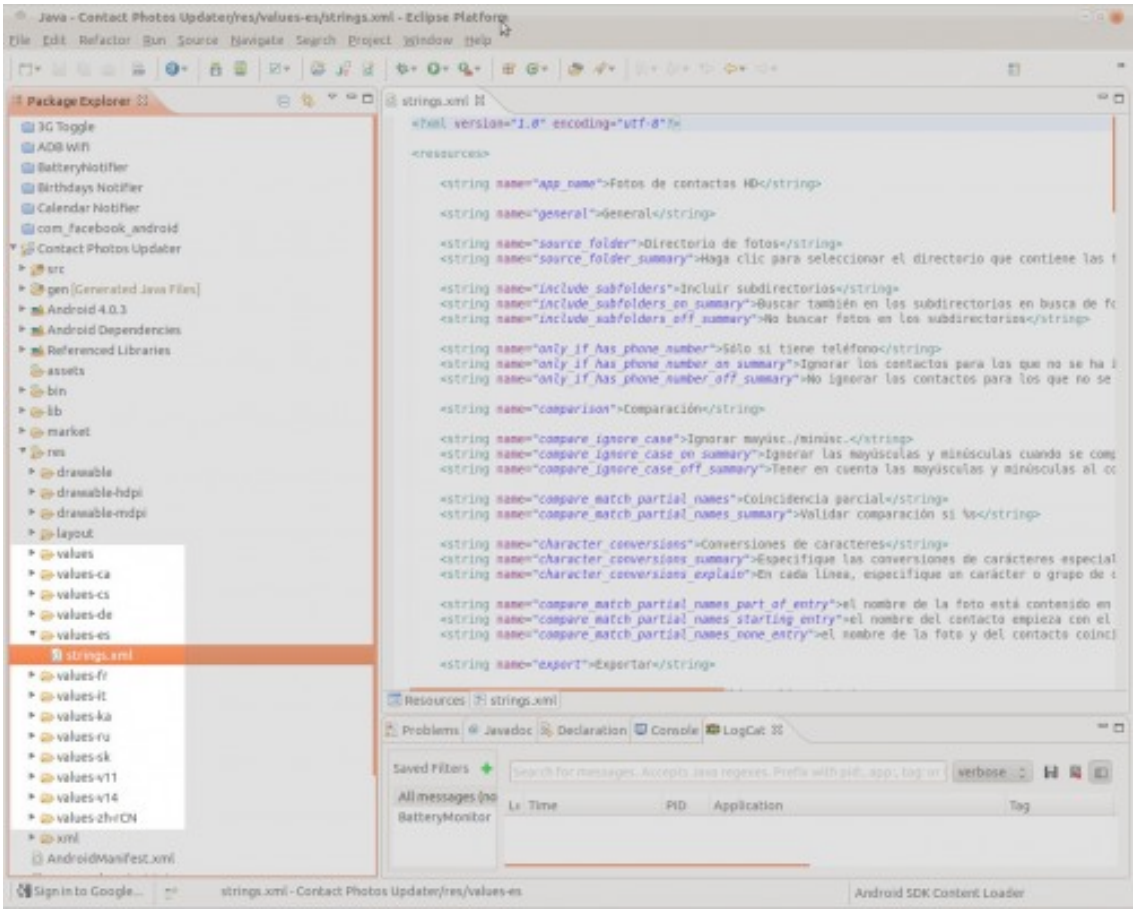
El directorio *res*, en un proyecto Android, contiene todos los datos relacionados con los recursos de una aplicación, tales como imágenes, a las que nos referiremos a continuación, y cadenas de caracteres.

Así, deberemos crear un directorio *values-XX* (donde *XX* es un **código de idioma**) para cada una de las lenguas a las que traduzcamos nuestra aplicación, incluyendo en dicho directorio un archivo *strings.xml* con la descripción de todas las cadenas en ese idioma concreto.

El propio sistema se encargará de acceder al archivo correcto dependiendo del idioma que se encuentre activo en el dispositivo, si es que hemos traducido nuestra aplicación a ese idioma, o al idioma por defecto (incluido en el directorio *values*) en caso contrario.

Es una norma no escrita que en el lenguaje por defecto de una aplicación Android es el inglés, por lo que las cadenas en inglés se incluirán en el archivo *strings.xml* del directorio *res/values*.

Adicionalmente, y como veremos, también es posible crear cadenas que varíen según la versión de Android del dispositivo en el que se ejecute la aplicación.



Imágenes

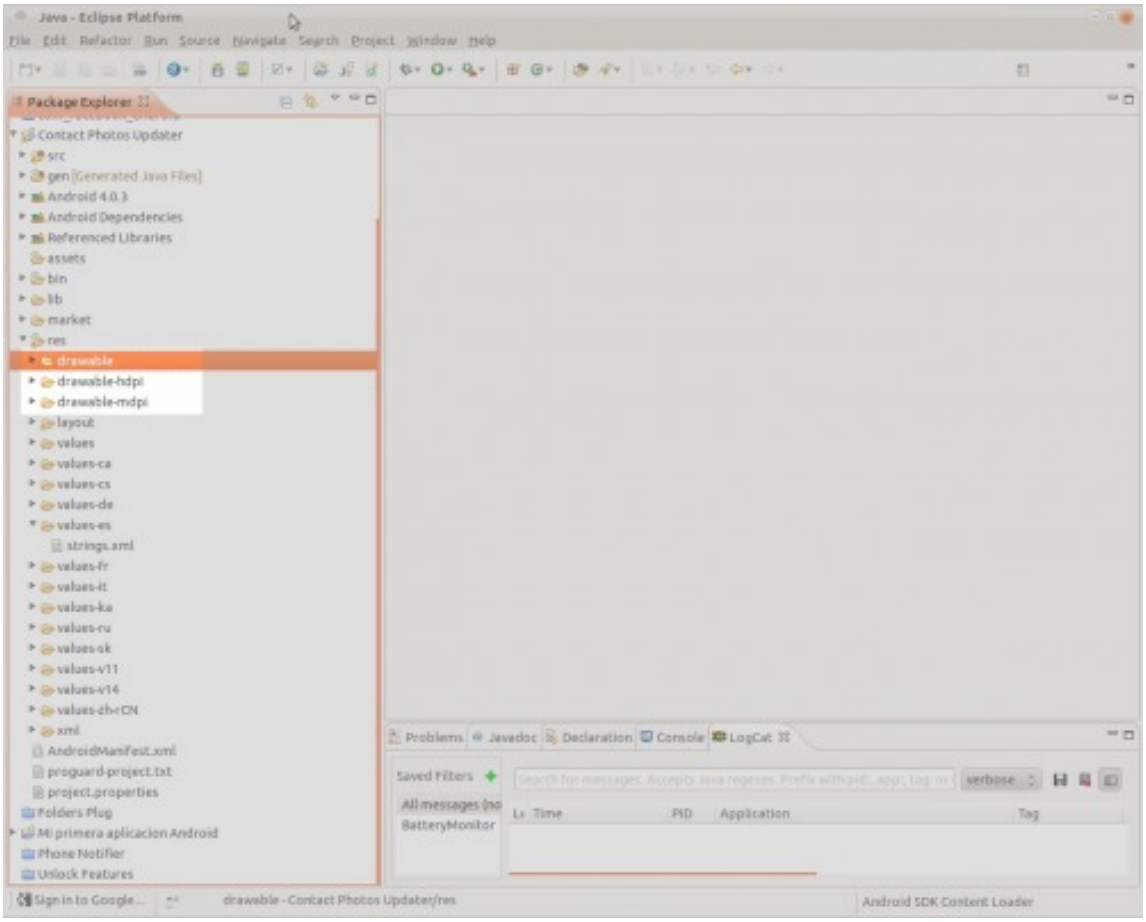
Tal y como ocurre con los idiomas, **es posible y en muchos casos necesario, especificar imágenes e iconos diferentes para cada una de las resoluciones de pantalla de Android.**

No hacerlo supondrá que nuestra aplicación se verá correctamente en los dispositivos con una resolución de pantalla similar a la usada durante el desarrollo pero no en dispositivos con pantalla más pequeña, o más grande.

La resolución de una pantalla la define la relación existente entre el tamaño de la misma (que se suele describir según el tamaño en pulgadas de ésta) y la densidad de píxeles (que es el número de éstos que caben en una pulgada), lo que nos permite hablar de pantallas pequeñas (*ldpi*), medianas (*mdpi*), grandes (*hdpi*) y extra grandes (*xhdpi*).

Incluiremos las imágenes relativas a cada una de las resoluciones en el directorio *res/drawable-XX* (donde *XX* es la resolución de pantalla) que corresponda.

De igual manera, el directorio *res/drawable* incluirá las imágenes que no cambian al cambiar la resolución de la pantalla, si las hubiera.



“Hola mundo”

Crearemos a continuación una pequeña aplicación (de prueba), que no hará nada salvo escribir un mensaje en la pantalla de nuestro emulador, o de nuestro teléfono si la instalamos en éste, pero que nos permitirá comprobar los conceptos explicados en este capítulo.

Para ello realizaremos los siguientes pasos:

1. Abrir Eclipse,
2. Seleccionar *File* → *New* → *Android project* en el menú de Eclipse,
3. Escribir “*Mi primer proyecto Android*” en el cuadro “*Project Name*” y hacer clic en *Next*,
4. Seleccionar *Android 4.0.3* en la lista y hacer click en *Next*,
5. Indicar “*com.faqsAndroid.HelloWorld*” en el cuadro “*Package Name*” y hacer click en *Finish*.

Hecho esto, comprobaremos que en la vista “*Package Explorer*” de Eclipse ha aparecido un elemento denominado “*Mi primer proyecto Android*”, sobre el que podremos hacer doble clic para ver los diferentes elementos que hemos comentado en este tutorial.

En cuanto a los directorios *gen* y *bin*, son directorios internos de Eclipse y no conviene eliminarlos ni modificarlos, mientras que el directorio *src* contiene el código de nuestro programa, que en este caso sólo crea una ventana que muestra un mensaje de texto.

Si ejecutamos la aplicación, comprobaremos que aparece en pantalla el ansiado “Hello World”. Ello es debido a que el esqueleto de una aplicación Android incluye una única ventana que muestra el mencionado mensaje en pantalla.

Si lo deseáis, podéis cambiar el valor de la cadena “*hello*” en el archivo *res/values/strings.xml*.

En próximos capítulos aprenderemos a modificar este esqueleto para adaptarlo a nuestras necesidades concretas, así como a crear otras ventanas según nuestras necesidades.

Ejecución en el emulador

Para ejecutar este programa en el emulador tan sólo es necesario hacer clic con el botón derecho del ratón en el nombre de nuestra aplicación y seleccionar *Run As* → *Android Application* en el menú emergente, o hacer clic directamente en el botón ejecutar de la barra de botones de Eclipse.

Ejecución en un dispositivo

Si por el contrario, preferimos ejecutar la aplicación en nuestro dispositivo, deberemos exportarla, para lo cual realizaremos los siguientes pasos:

1. Seleccionar el nombre de la aplicación en la vista *Package Explorer*,
2. Hacer clic con el botón derecho del ratón en el nombre de la aplicación y seleccionar *Export* en el menú emergente.
3. Seleccionar *Android → Export Android Application* en la pantalla que aparezca y pulsar *Next.*
4. Pulsar *Next*,

La primera vez tendremos que crear nuestro *almacen de claves*, para lo cual seleccionaremos "*Create new keystore*", seleccionaremos una ruta para el almacén de claves (del que **convendría tener copia de seguridad**) e introduciremos la contraseña del mismo, que guardaremos en lugar seguro, tras lo cual haremos clic en *Next*.

En caso que ya hayamos firmado alguna aplicación (o esta misma en alguna otra ocasión), seleccionaremos "*Use existing keystore*", procediendo a introducir la ruta del almacén en el cuadro correspondiente, así como su contraseña, tras lo cual haremos clic en *Next*.

En la siguiente pantalla, en caso de que acabemos de crear el almacén de certificados, tendremos que crear un certificado nuevo, para lo cual deberemos rellenar un formulario con nuestros datos. Es importante que, por seguridad, la contraseña no sea la misma que hemos puesto en el paso anterior, mientras que el campo *alias* nos servirá para identificar el certificado en caso de que tuviéramos varios.

En caso que el almacén de claves ya existiera, simplemente seleccionaremos "*Use existing key*", y procederemos a seleccionar el alias correspondiente, introducir su contraseña y hacer clic en *Next*,

Finalmente, seleccionaremos la ruta en la que debe almacenarse la aplicación en el cuadro "*Destination APK File*" y hacer clic en *Finish*.

Después tendremos que copiar el archivo generado en el dispositivo e instalarlo, para lo cual deberemos habilitar la instalación de aplicaciones de fuentes externas (habilitar *configuración → seguridad → orígenes desconocidos* en el dispositivo).

Un apunte

Por eficiencia, Eclipse utiliza una caché interna que le permite acceder a los archivos muy rápidamente.

Cuando realizamos modificaciones a través del propio Eclipse, tanto la caché como los propios archivos se actualizan correctamente, tal y como debe ser.

Sin embargo, cuando actualizamos directamente los archivos en el disco, mediante las utilidades de copiar-pegar, mover, cambiar de nombre, etc del sistema operativo, Eclipse no reacciona tan bien como debiera y, en ocasiones, no detecta estos cambios.

Para evitar esto es conveniente forzar un refresco de la cache en Eclipse cada vez que hacemos algún cambio desde fuera del entorno, para lo cual haremos clic en el botón derecho del ratón encima del nombre del proyecto en la vista "*Package Explorer*" y seleccionaremos la opción *refresh*.

En el próximo capítulo empezaremos el desarrollo de la aplicación que servirá para presentar los diversos componentes de Android, un gestor de archivos, centrándonos en esa primera sesión en el diseño de la pantalla.

CAPÍTULO 3

Actividades y Layouts en Android

Una actividad (del inglés *activity*) **es una tarea realizada por una aplicación y que se centra en la interacción con el usuario**, bien sea porque un proceso necesita unos datos de éste o por cualquier otro motivo.

Así, todas las actividades se basan en la creación de una ventana (a pantalla completa o no), en la que se posicionan los diversos componentes (botones, cuadros de texto o edición, imágenes, etc) necesarios para la comunicación con el usuario.

Los layouts, que se asocian a las diferentes ventanas o vistas, **definen el mapa que permite a Android posicionar cada uno de los elementos en el lugar correspondiente de la pantalla**.

Tal como indicamos en el capítulo anterior, **los layouts deben definirse teniendo en cuenta** tanto el tamaño de los elementos que contiene como el tamaño del elemento, habitualmente **la pantalla** u otro layout, que hace de contenedor del mismo, siendo posible definir layouts diferentes dependiendo del tamaño de dicho contenedor.

Ciclo de vida de las actividades

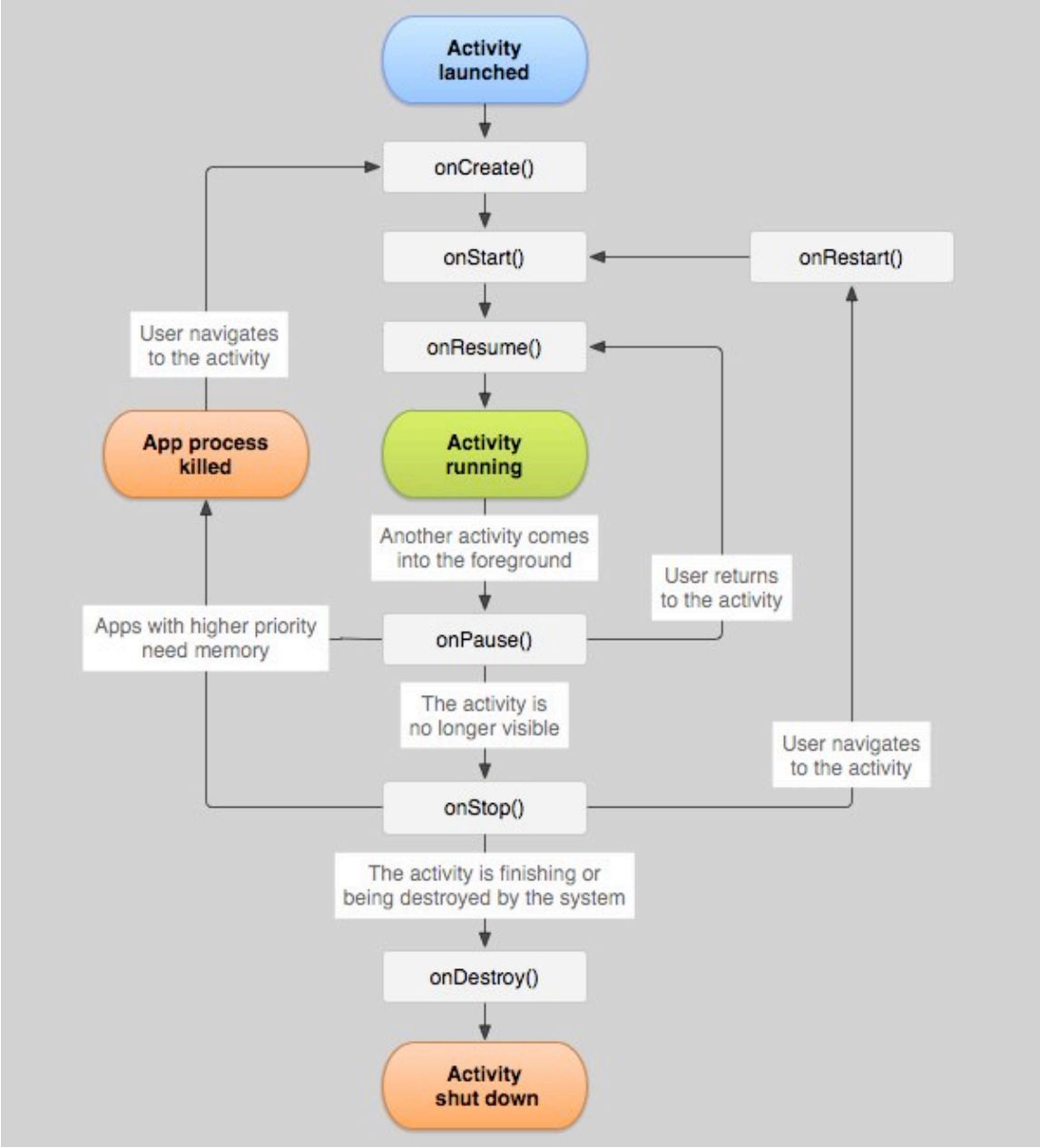
En informática existen dos conceptos básicos, que se explican entre los tres primeros días de carrera, y que nos acompañan durante toda nuestra vida laboral, y que son los conceptos de *pila* y *cola*.

Una *pila* es una estructura en la que el último elemento que entra es el primero que sale, el penúltimo en entrar es el segundo en salir y sucesivamente.

Por contra, una *cola* es una estructura en la que el primer elemento en entrar es también el primero en salir, mientras que el último en entrar será, lógicamente, el último en salir.

En Android el sistema gestiona las diferentes actividades usando el concepto de pila, de forma que, en un momento determinado, **sólo la actividad que se encuentra en la cima de esta pila se está ejecutando**, mientras que el resto están en *standby* y la memoria que ocupan puede ser asignada a otros procesos.

Así, cuando una actividad se encuentra en primer plano y otra ocupa la cima de dicha pila, aquella pasa a segundo plano y es susceptible de ser eliminada del sistema, en lo que se denomina "*ciclo de vida de una actividad*" y que se expresa gráficamente en la siguiente imagen.



Crear una actividad

Desgraciadamente el ADT no se integra con Eclipse tan bien como debería, así que para crear una actividad tendremos que realizar una serie de pasos (de forma manual):

Crear la clase asociada a la actividad, y que implementará todos sus métodos

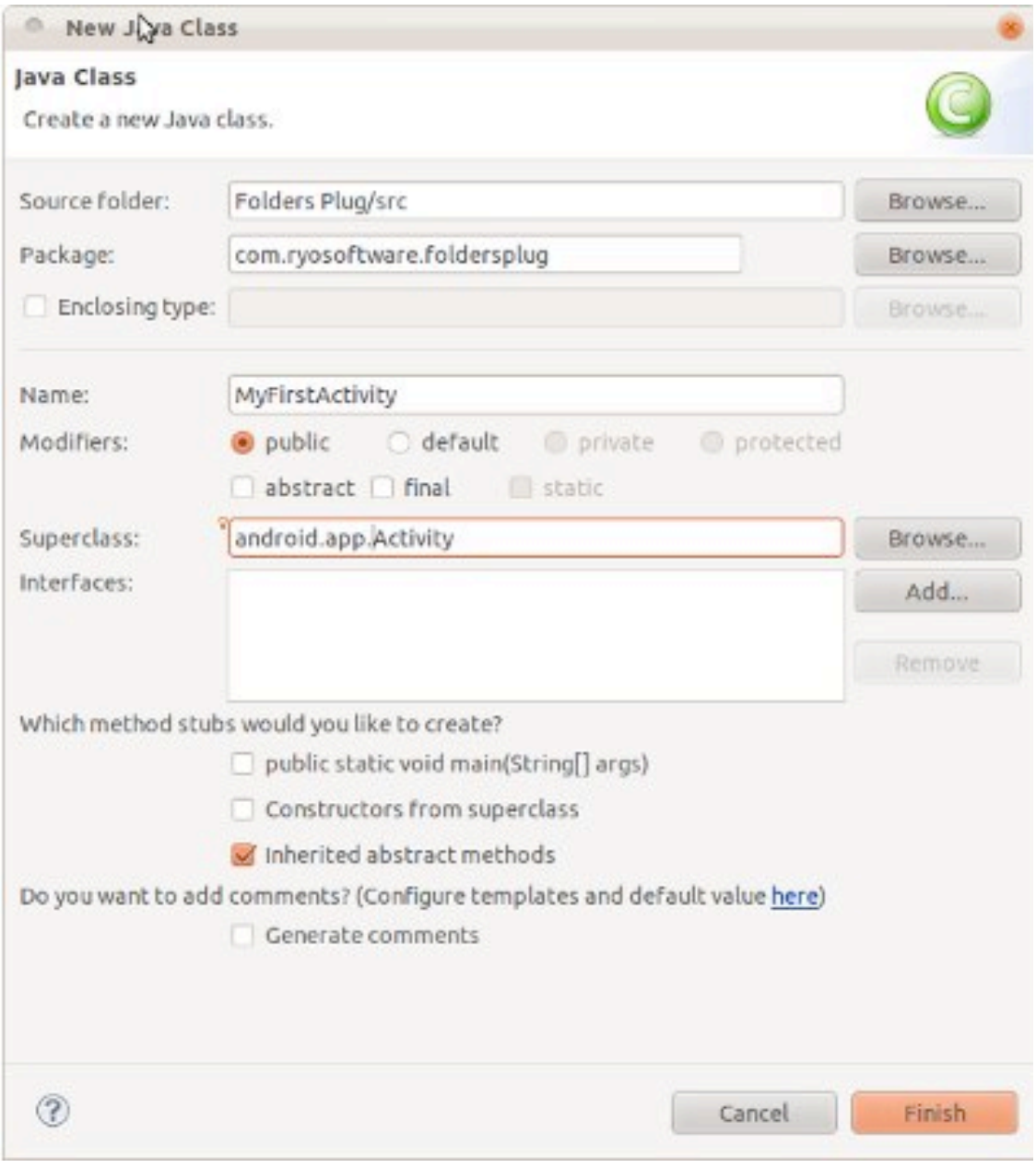
Para ello haremos clic en *File* → *New* → *Class* e introduciremos los siguientes valores en la pantalla de creación de la clase:

- **Package**: Seleccionaremos el paquete al que pertenece la clase de entre los definidos por nuestra aplicación,
- **Name**: Introduciremos el nombre de la actividad, teniendo en cuenta que no se aceptan espacios ni caracteres especiales. De hecho, una buena práctica es

poner la primera letra de cada palabra que forme el nombre en mayúsculas y el resto en minúsculas (por ejemplo *MiPrimeraActividad*),

- **Superclass**: Indicaremos que la clase que estamos creando es una actividad introduciendo el tipo *android.app.Activity*.

Seguidamente haremos clic en *finish* para crear la clase, lo cual creará el archivo asociado a la clase y lo mostrará en el editor de Eclipse, para que podamos implementar sus métodos.



Publicitar la actividad

No obstante, crear una actividad no es sinónimo de poder utilizarla, para lo cual es necesario añadirla en el manifiesto de nuestra aplicación (archivo *manifest.xml*).

Activar una actividad

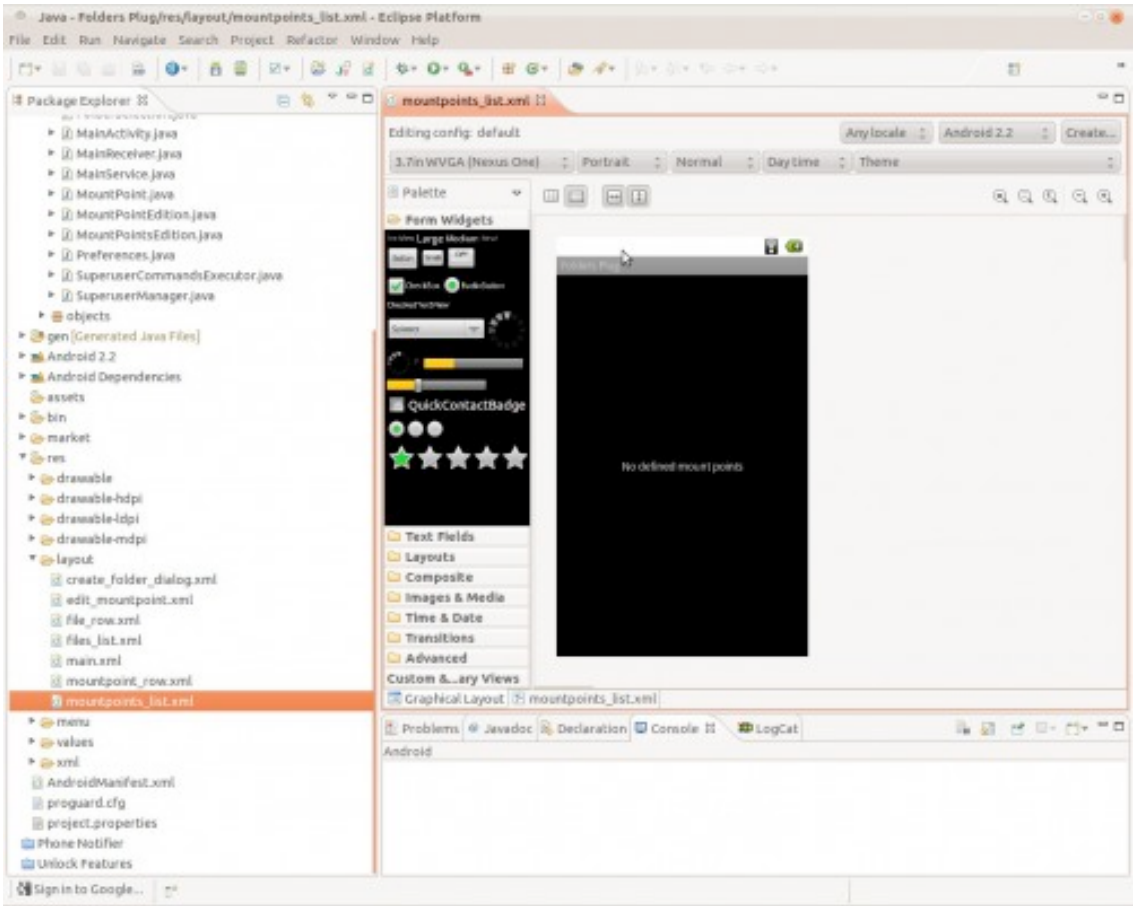
Una función o procedimiento se ejecuta de forma *síncrona* cuando el flujo de ejecución del programa en cuestión no vuelve al procedimiento principal (el que realiza la llamada) hasta que la función que ha sido llamada finaliza; mientras que una función se ejecuta de forma *asíncrona* cuando el proceso que realiza la llamada sigue ejecutándose aunque la función llamada no haya finalizado.

La creación y activación de una actividad es un proceso asíncrono que se realiza generalmente desde otra actividad, usando para ello las funciones *startActivity* o *startActivityForResult*, heredadas de la superclase, y que se diferencian en que la última permite obtener un resultado, también de forma asíncrona.

Los layouts

Si un actividad define el contenedor en el que se colocan los componentes de una ventana, un layout describe cómo se colocan dichos componentes en la misma.

Los layouts se estructuran en forma de árbol, con un elemento raíz que se posiciona sobre la ventana, y sucesivos compartimentos (nodos) en los que se colocan los elementos finales u hojas (imágenes, cuadros de edición o de texto, botones, etc).



Los contenedores de elementos más utilizados son los de tipo *LinearLayout* y *RelativeLayout*.

LinearLayout

Es un contenedor en el que los diferentes elementos se posicionan uno debajo del otro, o uno a la derecha del otro, dependiendo de si se define de tipo vertical u horizontal.

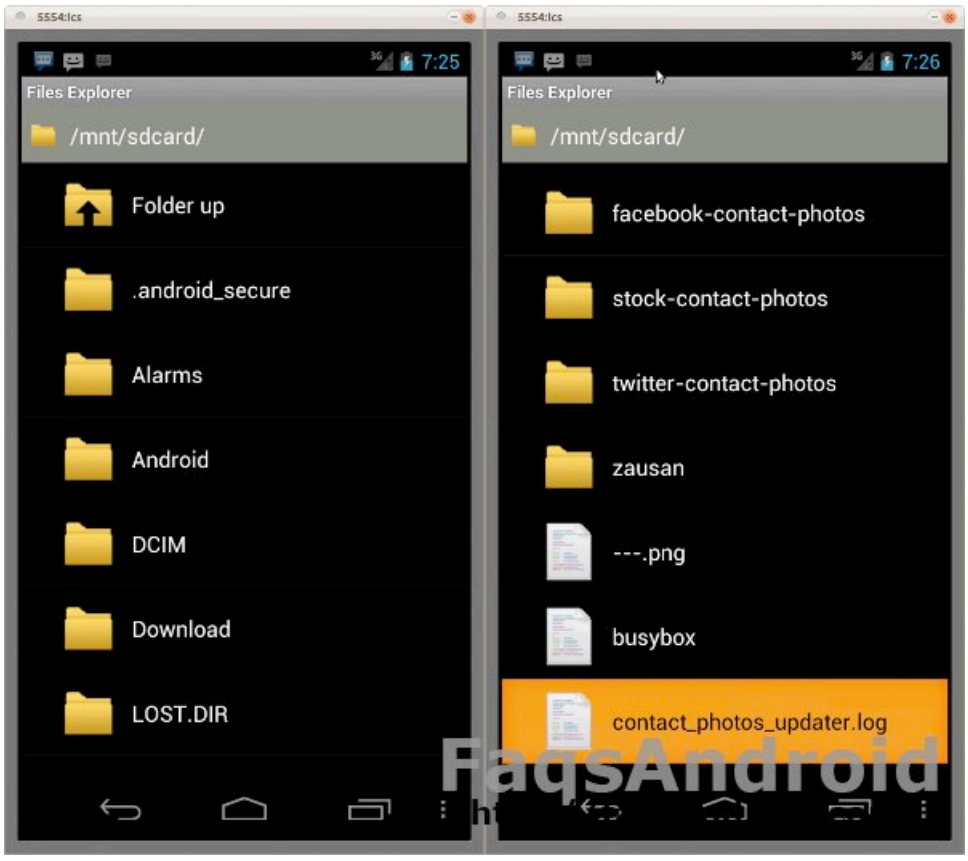
RelativeLayout

Se trata de un contenedor más complejo en el que es posible indicar la posición relativa de un elemento respecto de otro, permitiéndose incluso el solapamiento de éstos.

De hecho, por defecto todos los componentes se alinean en la parte superior izquierda del contenedor, siendo responsabilidad del diseñador posicionarlos convenientemente mediante el uso de atributos y la definición de dependencias, según sea el caso.

Para cada uno de los contenedores, así como para los elementos finales (botones, imágenes, etc) es posible definir multitud de atributos, como por ejemplo el tamaño horizontal o vertical, la visibilidad, el color o imagen de fondo, etc.

Descarga de la aplicación del curso



Os animamos a descargaros la primera parte de la aplicación que estamos desarrollando en este curso Android, que amplía los conceptos introducidos en estas primeras sesiones y que contiene un código totalmente funcional que podréis probar en el emulador o en vuestros dispositivos.

Una vez descarguéis la aplicación, deberéis almacenarla en el directorio de trabajo de Eclipse (el que seleccionasteis durante el proceso de instalación).

Seguidamente será necesario importarla desde Eclipse, para que éste la reconozca y podáis acceder a la misma, bien sea para editarla o para ejecutarla en el emulador; para lo cual deberéis seguir los siguientes pasos:

1. Seleccionar la opción *import* en el menú *File*,
2. En la siguiente pantalla, seleccionar "*Existing project into workspace*" y pulsar *next*,
3. Hacer clic en *browse* y navegar hasta el directorio en el que hemos almacenado la aplicación (hay que entrar en el directorio, hasta que veamos el archivo manifest.xml),
4. Hacer clic en *finish*.

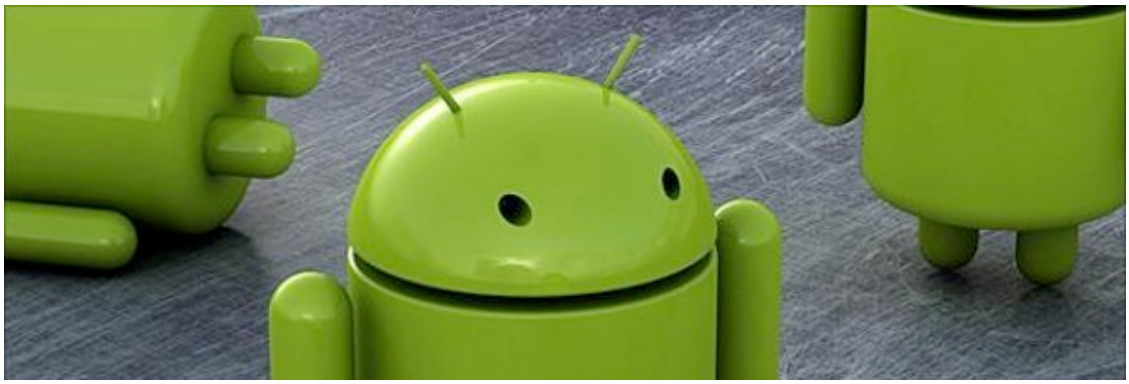
En el próximo bloque os explicaremos cómo crear una pantalla de configuración para vuestras aplicaciones, así como los métodos que usa el sistema para facilitar el acceso a los valores especificados por el usuario.

CAPÍTULO 4

Pantallas de configuración

La práctica totalidad de las aplicaciones disponen de una o más actividades de configuración, y pese a que sería posible implementarlas como si se tratara de una actividad cualquiera, lo cierto es que **es mucho más sencillo usar la API del sistema siempre que ésta exista**, y en el caso de las actividades de configuración el tema está resuelto y muy bien resuelto.

Como veremos en esta parte del curso android, este sistema **incorpora una API específica para tratar estas actividades de configuración** que permite no sólo visualizar la propia ventana, que se muestra de idéntica manera para todas las aplicaciones, proporcionando al usuario una sensación de continuidad que mejora la experiencia de uso; sino que incorpora además funciones específicas para acceder fácilmente a los valores especificados por el usuario, así como otras que permiten ejecutar diferentes acciones cuando éste cambia alguna característica configurable.



Cómo crear una pantalla de configuración

Por comodidad, en este tutorial hablaremos de *pantallas de configuración* para referirnos a las actividades de configuración, y de *características* o *preferencias* de usuario para referirnos a cada uno de los valores que éste puede configurar.

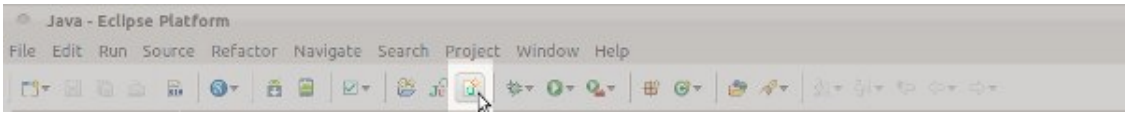
Como hemos indicado antes, **las pantallas de configuración son ventanas**, si bien en este caso la superclase se denomina *PreferenceActivity* y añade ciertas funciones y procedimientos que permiten tratar correctamente las actividades de este tipo aunque, en general, podemos asumir que son simplemente ventanas.

Para crear una nueva pantalla de configuración deberemos seguir los pasos especificados en el tutorial anterior para la creación de una actividad genérica, pero teniendo en cuenta que, en este caso, la superclase se denomina *android.preference.PreferenceActivity*.

No debemos olvidar que es necesario publicar una referencia a la nueva actividad en el manifiesto de nuestra aplicación, ya que en caso que no lo hagamos nuestro programa fallará cuando intentemos acceder a la mencionada pantalla.

Como describir las opciones de configuración

Las diferentes opciones de configuración de nuestra aplicación se describen en un archivo XML que se sitúa en el directorio *res/xml* de ésta, siendo posible editarlo manualmente o mediante la interfaz de Eclipse.

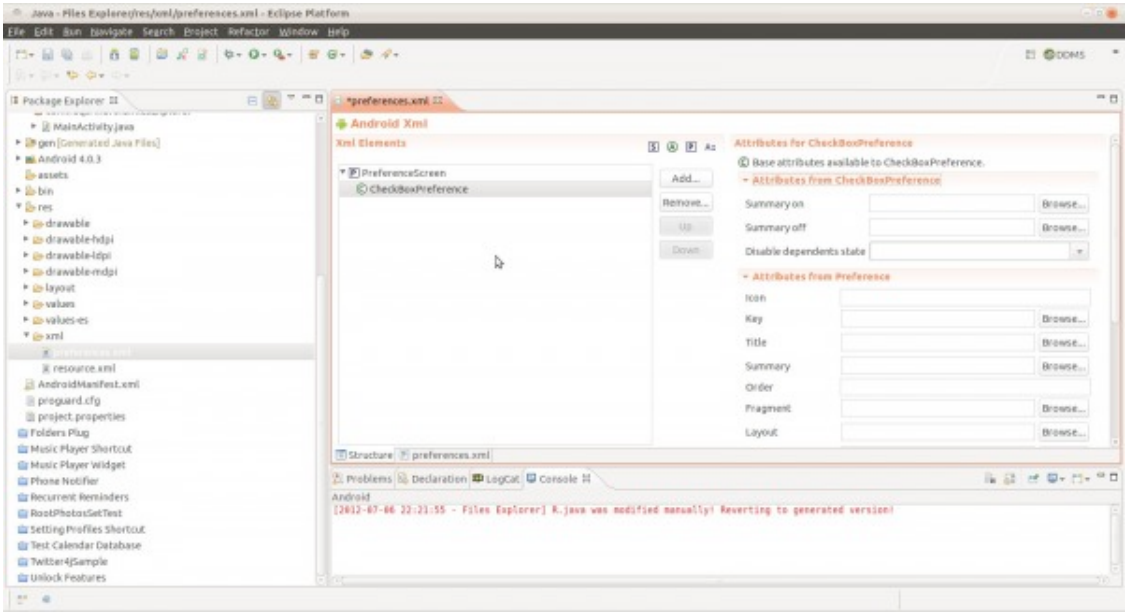


La creación mediante Eclipse se realiza mediante un asistente (ver imagen más arriba para ver cómo acceder a éste) en el que seleccionaremos los siguientes valores:

- **Resource Type**: Preference,
- **File**: El nombre del archivo. Comúnmente preferences.xml,
- **Root Element**: Seleccionar PreferenceScreen.

Una vez creado el archivo, éste se abrirá en el asistente, a través del cual podremos crear los diferentes elementos que lo formen.

En la pantalla de edición deberemos tener en cuenta que en la parte izquierda nos aparecerán los diferentes elementos, estructurados en forma de árbol, mientras que en la parte derecha nos aparecerán las características del elemento concreto seleccionado.



También podremos el botón *add* para añadir un nuevo elemento, y los botones *remove*, *up* y *down* para eliminar, mover hacia arriba o mover hacia abajo el elemento seleccionado, respectivamente.

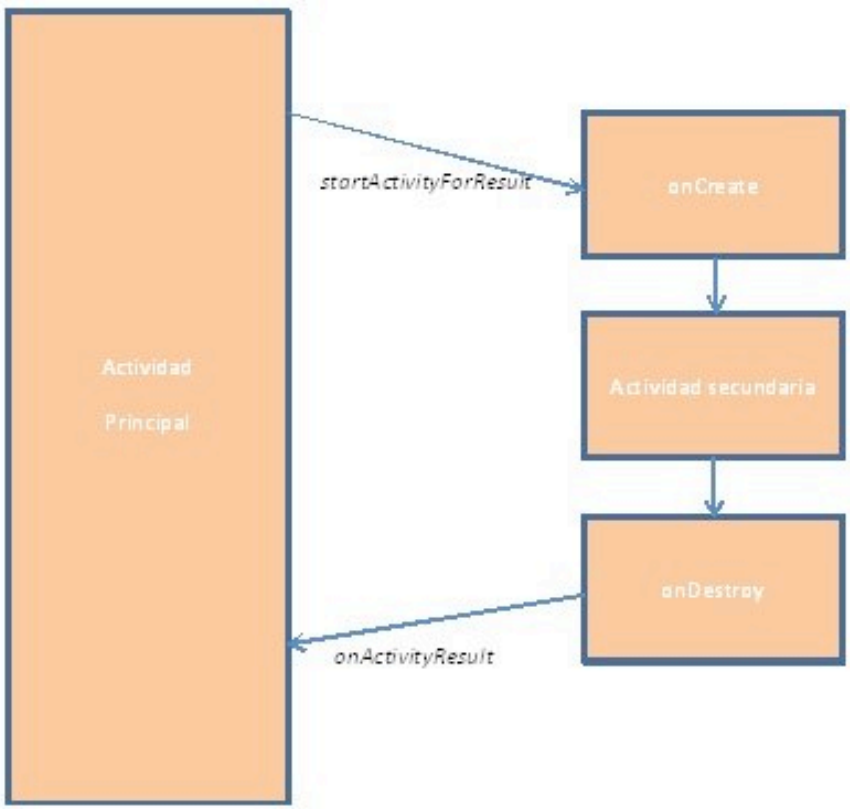
Debe tenerse en cuenta que, tal como se muestra en el código de ejemplo, es conveniente inicializar las características de los diferentes elementos usando referencias en lugar de cadenas de texto concretas, ya que así facilitamos las modificaciones posteriores, así como la futura inclusión de traducciones.

Cómo detectar cuando la pantalla de configuración se cierra

Lo normal es que las actividades de configuración sean lanzadas desde otra actividad, bien por la pulsación de un botón en dicha actividad, bien por la selección de la opción correspondiente en el menú de la misma.

Cuando una actividad desea ser informada de que una actividad que dependa de ella ha finalizado debe iniciarla usando el procedimiento *startActivityForResult*, que ya os comentamos en la sesión anterior.

Cuando una actividad iniciada mediante este procedimiento acabe, el sistema se encargará de activar un evento (*onActivityResult*) en la actividad originadora que le permitirá saber que aquella ha acabado, así como recuperar los eventuales resultados de la ejecución.



Cómo detectar cuando una característica de configuración cambia

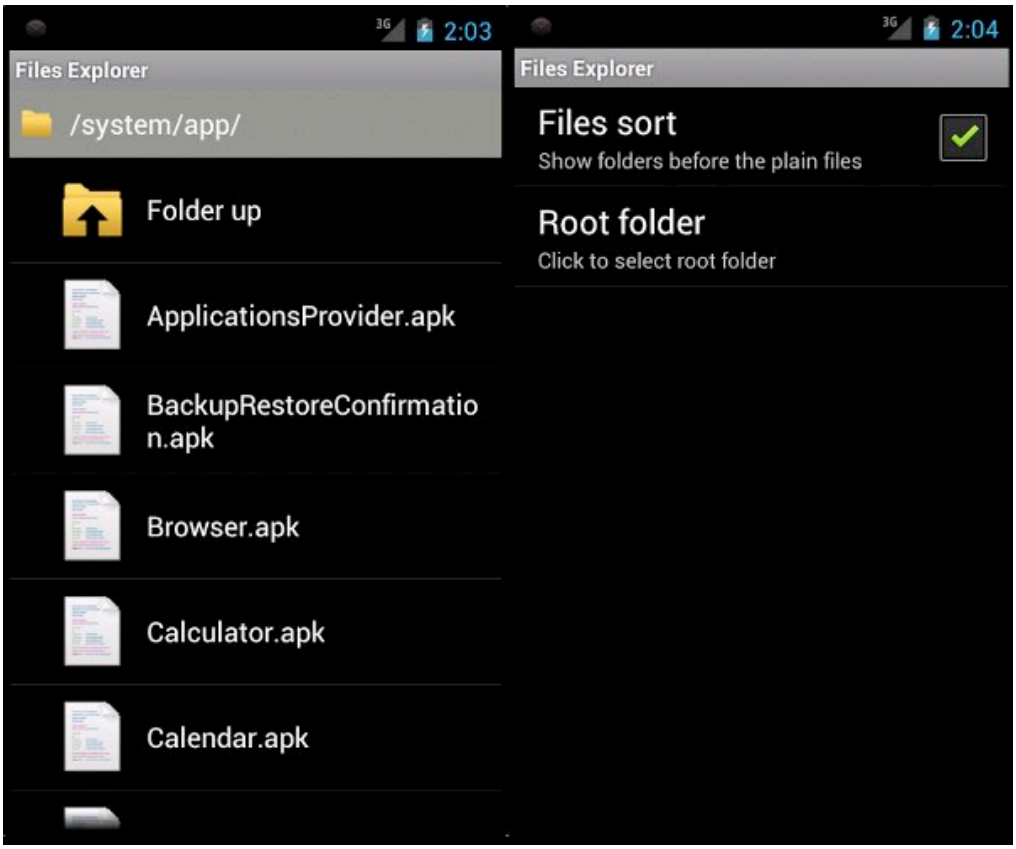
Los servicios, que abordaremos en los siguientes capítulos, se ejecutan sin la intervención del usuario, bien por la activación de un evento en el dispositivo, como por ejemplo la recepción de un mensaje de texto, bien porque implementan acciones que se ejecutan en segundo plano y que no requieren la intervención del usuario.

Es posible que un servicio, o cualquier otra parte de nuestra aplicación, necesite recibir una notificación (evento) cuando el usuario cambie alguna de las preferencias de ésta, para lo cual deberá ejecutar el método *registerOnSharedPreferenceChangeListener* en una instancia de la clase *SharedPreferences*.

El programador debe tener en cuenta que es obligatorio llamar al procedimiento *unregisterOnSharedPreferenceChangeListener* cuando la necesidad de recibir el evento acabe o, en último término, cuando el objeto que recibe el evento se destruya, ya que en caso contrario la aplicación fallará.

Cómo acceder a un valor de una característica concreta

Es posible acceder a las preferencias de usuario en cualquier momento durante la ejecución de nuestras aplicaciones, para lo cual deberemos crear una instancia de un objeto de tipo *SharedPreferences*, sobre la que usaremos los métodos *getBoolean*, *getInt*, *getLong*, *getFloat* o *getString* según corresponda.



Cómo modificar manualmente el valor de una característica concreta

Aunque no es usual, es posible que en una aplicación debáis modificar manualmente una preferencia de usuario, por ejemplo para reflejar la desactivación de una propiedad porque la aplicación no ha sido registrada.

El código que os mostramos a continuación permite realizar dicha modificación.

```
public void setPreference (Context context, String key, String value)
{
SharedPreferences.Editor preferences_editor =
PreferenceManager.getDefaultSharedPreferences(context).edit();
preferences_editor.putString(key, value);
preferences_editor.commit();
}
```

Qué es el contexto de una aplicación

En el código anterior hemos introducido una variable de nombre *context*, de la cual no habíamos hablado con anterioridad, y que es de tipo *Context*.

El tipo *Context* representa una interfase de información global acerca de la aplicación, y es usado siempre que se necesita acceder al sistema o a datos concretos de ésta.

Muy a bajo nivel, las ventanas son de tipo *Context*, por lo que heredan todas sus propiedades.

Así, cuando llamemos a la función anterior desde una actividad, podremos realizar la llamada de la forma

```
setPreference(this, key, value);
```

En el próximo capítulo os hablaremos de los threads, que permiten ejecutar varias partes de código de forma concurrente (simultánea), y os explicaremos las restricciones que les afectan.

Os recomendamos que os descarguéis la nueva versión de la aplicación de prueba que estamos desarrollando.

CAPÍTULO 5

Threads

Algo que no hemos visto hasta el momento en nuestro Curso Android es la capacidad de tener hilos paralelos de procesamiento.

Inicialmente, los sistemas operativos sólo eran capaces de realizar una única tarea de forma simultánea, lo que bloqueaba el proceso, y hasta el ordenador, cuando éste tenía que realizar cálculos importantes o una operación de entrada/salida.

Más adelante, los sistemas fueron capaces de ejecutar varios procesos diferentes a la vez, de forma concurrente, lo que permitía optimizar el uso de la CPU, aprovechando los tiempos en que ésta estaba desocupada para ejecutar otras tareas.

Android es un sistema multi-proceso, lo que quiere decir que es capaz de ejecutar varios procesos (tareas) de forma simultánea, **y multi-flujo**, que significa que puede ejecutar de forma simultánea dos o más porciones de código del mismo proceso.

Recordemos que los procesos se ejecutan en contextos separados, por lo que no comparten datos, mientras que los flujos o hilos de un mismo proceso se ejecutan en el mismo contexto, por lo que comparten el mismo espacio de memoria y pueden interferir entre ellos, lo que debe ser tenido en cuenta por el programador.

Android es un sistema fuertemente orientado al usuario, y a la interacción de éste con el sistema, que se hace efectiva por medio de clics, pulsaciones en la pantalla o en los botones, etc.

Cuando el usuario interacciona con una aplicación y ésta está ocupada en otros quehaceres, el sistema le asigna un *tiempo de gracia* para que finalice lo que esté haciendo y atienda al usuario; asumiendo el sistema que la aplicación ha dejado de funciona en caso que no lo haga, lo que provoca que el sistema la finalice

automáticamente, lo que se conoce con el nombre de *ANR* y que, junto con el de sobras conocido *FC*, es el mensaje de error más habitual en Android.

El programador deberá tener en cuenta este hecho y no realizar en el hilo principal operaciones que puedan bloquear el proceso, tales como bucles con un alto número de iteraciones, operaciones de entrada/salida, accesos a internet, etc.

Convertir una aplicación single-thread en multi-thread

Android distingue entre hilo principal de un proceso y todos sus hilos secundarios, permitiéndose únicamente al hilo principal acceder a la interfase gráfica.

Así, aunque podemos usar un hilo separado para calcular los diferentes elementos de una lista (*ListView*), deberemos usar el hilo principal para añadir los diferentes elementos a la vista y para redibujarla, si es el caso.

En realidad, convertir una aplicación que se ejecuta en un único hilo en multi-hilo es más sencillo de lo que parece.

La clase *Thread*

Android implementa los diferentes threads que se ejecutan en un determinado momento usando la superclase *Thread*, que define un procedimiento denominado *run* que contiene todo el código que ejecutará un hilo concreto.

Así, para iniciar un nuevo hilo, tan sólo es necesario instanciar una variable de tipo *Thread*, en la que habremos implementado convenientemente el procedimiento mencionado en el apartado anterior, y ejecutar la función *start*, tal como se muestra más abajo.

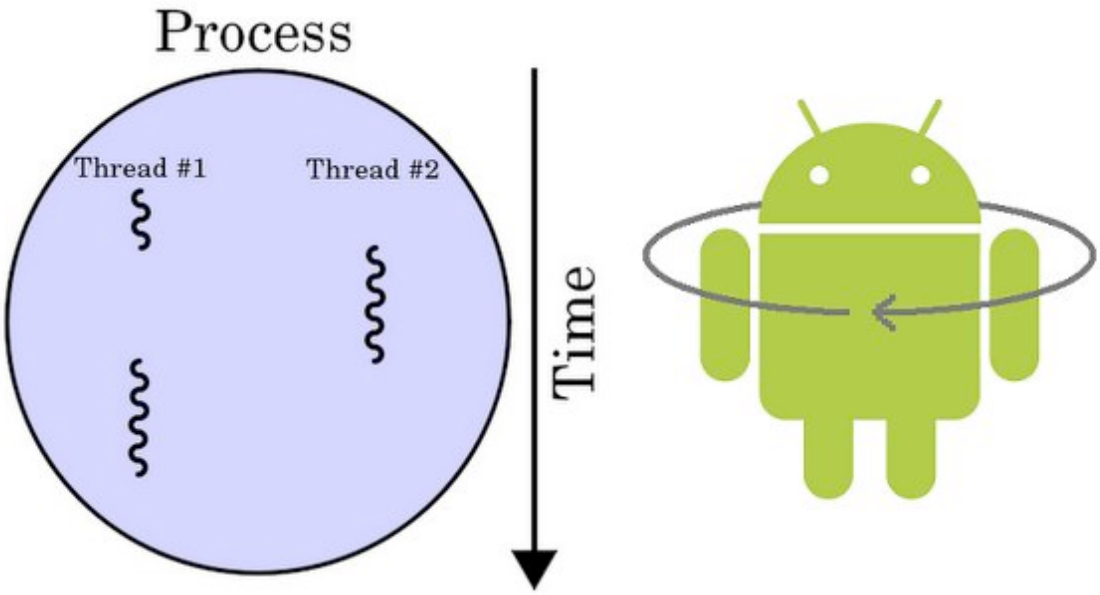
```
class MyThread extends Thread
{
public void run ()
{
...
```

```
}
}

Thread myThread = new MyThread();
myThread.start();
```

El programador debe tener en cuenta que tras la llamada al procedimiento *start*, el proceso principal (el que ha hecho la llamada) continuará su ejecución, que se realizará de forma paralela a la del nuevo hilo.

Asimismo, debe tener en cuenta que **el sistema no le notificará cuando el hilo finalice** (al acabar la función *run*), por lo que deberá ser el propio hilo el que envíe la notificación, si es el caso.



La clase *Handler*

Existen diferentes maneras de enviar mensajes entre los diferentes hilos que implementan un proceso, siendo la más elegante, y comúnmente utilizada, la que se implementa usando instancias de la clase *Handler*.

Un *Handler* es un objeto de sistema capaz de procesar los diferentes mensajes que recibe, según el tipo de éstos y los parámetros adicionales que se reciban.

Es el propio sistema el que se encarga de enviar, de forma asíncrona, los mensajes al *Handler* adecuado, que lo procesa en el contexto del flujo que lo ha creado, lo que permite contextualización.

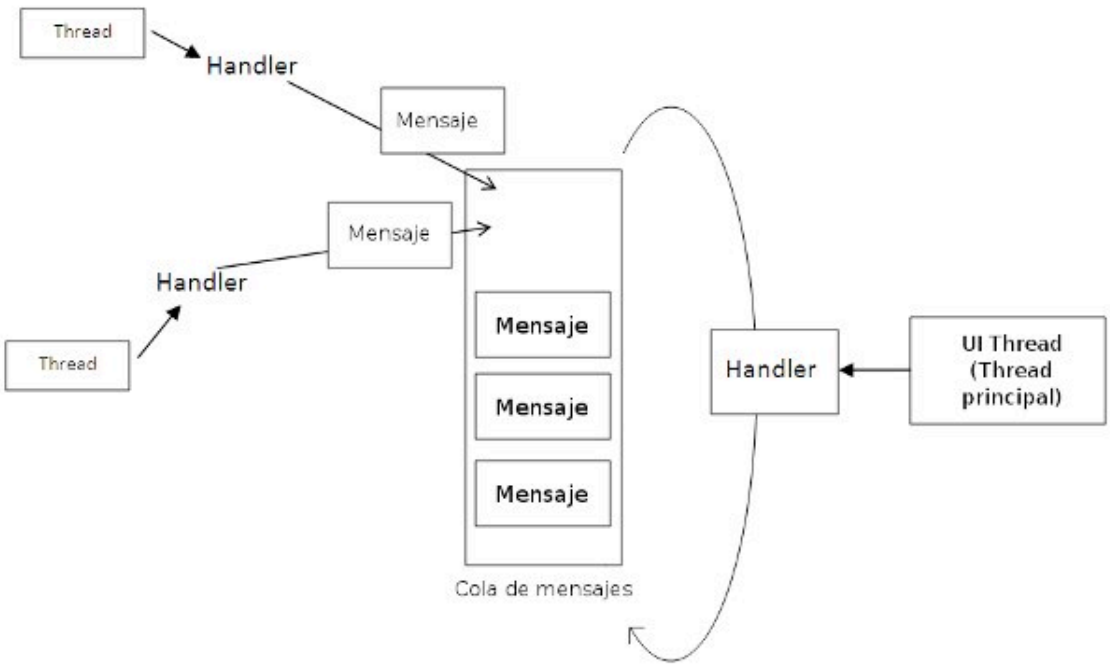
Recordemos que una llamada a un procedimiento es asíncrona cuando no tenemos garantías de que el código llamado se ejecute antes de que la llamada finalice.

```
class MyHandler extends Handler
{
public void processMessage (Message message)
{
switch message.what
{
...
}
}
}

Handler myHandler = new MyHandler();
…
Message message = new Message();
message.what = 1;
myHandler.sendMessage(message);
```

Cómo visualizar datos en una aplicación multi-flujo

Como hemos indicado, **sólo el hilo principal puede acceder a la UI**, por lo que será éste el único que podrá interactuar con los diferentes elementos que conformen las diferentes ventanas de nuestra aplicación.



De hecho, es una buena práctica que el hilo principal se encargue solamente de procesar los diferentes eventos que genere el usuario y/o la propia aplicación, iniciando hilos nuevos para implementar las operaciones de entrada/salida y/o

aquellas operaciones de proceso de datos que no sean triviales, teniendo en cuenta que el coste (en tiempo) de crear un nuevo hilo, aunque no es elevado, no es despreciable.

Para ello definiremos una instancia de la clase *Handler* en el hilo principal que se encargará de procesar (asíncronamente) los mensajes que le envían los diferentes hilos indicando que han finalizado sus tareas, procediendo a redibujar vistas, añadir o eliminar elementos gráficos, etc en el hilo principal.

La clase Runnable

En Java no existe la herencia múltiple, por lo que cuando extendemos una clase de tipo Thread perdemos la posibilidad de hacer que extienda otra, de ahí que los threads se usen sólo para implementar operaciones que empiezan y acaban, ya que cuando la función *run* finaliza también se acaba el objeto que la define.

Para evitar esto, se creó la clase *Runnable*, que no extiende una clase, sino que permite implementar métodos de acceso a la misma, lo que permite una mayor versatilidad.

Así, la clase *Thread* se ocupa habitualmente de implementar operaciones complejas en tareas, mientras que la clase *Runnable* implementa tareas completas. Es decisión del programador elegir cuándo utilizar una u otra.

En el próximo capítulo veremos cómo capturar los eventos (de usuario y de sistema), algunos de los cuales ya hemos introducido brevemente en sesiones anteriores.

Mientras tanto, puedes descargar la versión número 3 de la aplicación que estamos desarrollando, en la que comprobarás cómo obtenemos la lista de archivos de un directorio en un hilo separado y cómo enviamos un mensaje al hilo principal cuando esta lista ya está completa, procediendo éste a actualizar la lista que se muestra al usuario.

CAPÍTULO 6

Captura de eventos

En sesiones anteriores de nuestro curso Android te hemos explicado cómo crear tu primera aplicación, así como la manera de optimizarla usando múltiples hilos de procesamiento; en esta ocasión te enseñaremos a recibir eventos del sistema y/o de otras aplicaciones.

En Android distinguimos dos tipos de eventos:

Los relacionados con la interfaz gráfica, como por ejemplo los relacionados con el ciclo de vida de una actividad y/o los clics en los diferentes elementos que conforman una ventana, y que son programables mediante el propio lenguaje Java.

El resto de eventos, que no se refieren a la interfaz gráfica, y que no son directamente tratables mediante las funciones y procedimientos estándar de Java, como por ejemplo la detección de la llegada de un nuevo mensaje de texto o del apagado de la pantalla.

Eventos de la aplicación

Como indicamos en el capítulo anterior, el lenguaje de programación Java permite a una clase incorporar las funciones y procedimientos de otra mediante la utilización de las directivas *extends* (extiende, hereda) o *implements* (implementa).

Las *interfaces* son clases abstractas que se usan para definir métodos de entrada (callbacks) **que se codifican en la clase que los implementa.**

Las interfaces son utilizadas por el sistema para implementar los eventos (generalmente los clics en botones, eventos de los temporizadores, etc), y pueden

implementarse a través de una clase o de forma anónima, tal como se indica en los ejemplos que siguen.

Implementación a través de una clase

```
class MyClass implements OnClickListener {  MyClass()  {    Button button;    ...
button.setOnClickListener(this);  }   public void onClick(View view)  {    ...  } }
```

Implementación anónima

```
class MyClass implements OnClickListener {  MyClass()  {    Button button;    ...
button.setOnClickListener(new OnClickListener()    {      public void onClick(View view)
{       ...      }    });  } }
```

Otros eventos

Los eventos externos a la propia aplicación se implementan a través de una clase denominada *BroadcastReceiver*, la cual implementa el método *onReceive*, que es el punto de acceso a los objetos de dicha clase.

Si el programador desea que un evento concreto, por ejemplo el que se activa cuando el sistema acaba de iniciarse y está preparado para ejecutar programas, **se active incluso si la aplicación que lo recibe no se está ejecutando, debe declararlo en el manifiesto**, lo que indicará al sistema este hecho y lo preparará para ejecutarlo cuando sea necesario.

```
<manifest ...>  <application ...>   <receiver
android:name=".BootCompletedBroadcastReceiver">     <intent-filter>      <action
android:name="android.intent.action.BOOT_COMPLETED" />     </intent-filter>
</receiver>  </application> </manifest>
```

El programador es el encargado de incorporar a su proyecto las clases que implementan cada uno de los BroadcastReceiver que declare (en el ejemplo anterior habría que definir la clase *BootCompletedBroadcastReceiver*, que extendería las propiedades de la clase *BroadcastReceiver*).

Adicionalmente, debe tenerse en cuenta que **algunos eventos son de acceso privilegiado, por lo que necesitan** que el usuario suministre a la aplicación **una serie de permisos** cuando ésta se instala, y que también deben enumerarse en el manifiesto.

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

No obstante, también es posible definir un *receiver* que monitorice eventos sólo cuando la aplicación, o un determinado servicio o actividad, se encuentran en

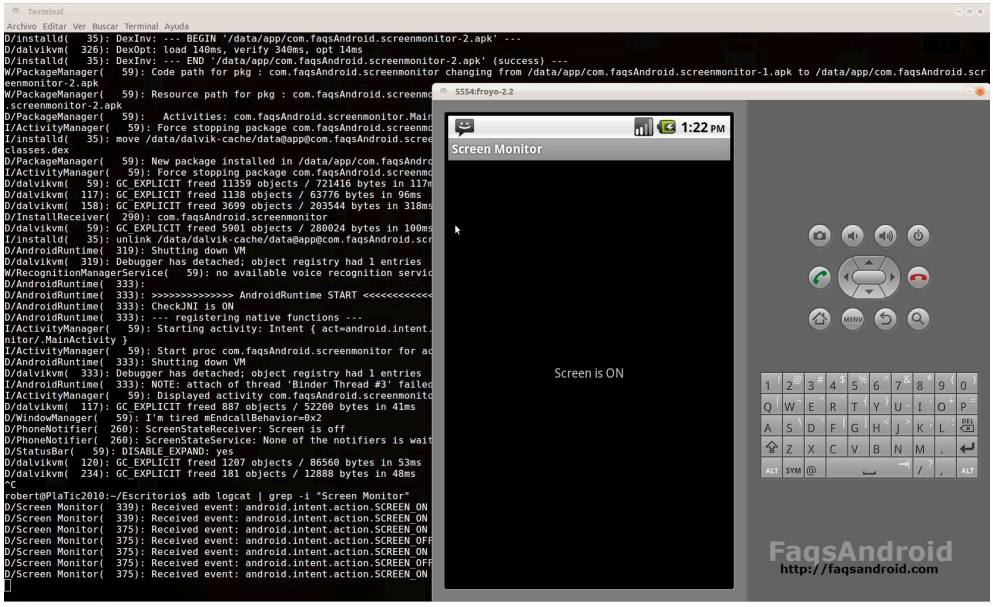
ejecución, para lo cual debe implementarse la correspondiente instancia de la clase *BroadcastReceiver* e instanciarse tal como se indica en el ejemplo.

```
class MyActivity extends Activity {   class MyActivityBroadcastReceiver extends
BroadcastReceiver   {     public void onReceive(Context context, Intent intent)    {      ...
}  }    private BroadcastReceiver iBroadcastReceiver = new
MyActivityBroadcastReceiver();    public void onResume()   {     super.onResume();
registerReceiver(iBroadcastReceiver, new IntentFilter(Intent.ACTION_SCREEN_OFF));
}   public void onPause()   {     unregisterReceiver(iBroadcastReceiver);
super.onPause();   } }
```

Nótese que los eventos empezarán a monitorizarse a partir de la llamada al procedimiento *registerReceiver* (presente en las clases *Activity* y *Service*) y dejarán de monitorizarse tras la llamada a *unregisterReceiver* (que debe realizarse forzosamente antes de que el objeto sea destruido, para evitar errores posteriores).

Recepción de parámetros

El procedimiento *onReceive* de la clase *BroadcastReceiver* se ejecuta cuando se activa el evento para el cual ha sido programado, recibiendo como parámetros el *contexto* de la aplicación en el que se ejecuta y un *intent* que contiene el tipo de evento activado (*intent.getAction*), así como el resto de parámetros asociados al mismo (accesibles usando las funciones *intent.getBooleanExtra*, *intent.getIntExtra*, etc).



En cuanto a la aplicación de éstos a nuestro programa de ejemplo, no hemos encontrado la posibilidad de aplicarlos, **por lo que hemos creado un proyecto nuevo, en el que detectaremos cuándo se enciende o apaga la pantalla de nuestro dispositivo**. Os recomendamos que hagáis un seguimiento del archivo de *log*, **para lo cual podéis usar el comando "*adb logcat*"**.

CAPÍTULO 7

Comunicación con otras Apps

En el capítulo anterior de nuestro curso Android vimos cómo hacer que nuestra aplicación interactúe con los eventos del sistema para, por ejemplo, ejecutar una acción cuando la pantalla se apaga, o se enciende.

En esta ocasión, usaremos el mismo principio para conseguir que dos aplicaciones diferentes se envíen mensajes a través del sistema.

En Android, y en la mayoría de los sistemas operativos modernos, **las aplicaciones se ejecutan en espacios de memoria separados**, lo que aumenta la seguridad de éste al impedir que una aplicación malintencionada, o no, afecte al funcionamiento de otra.

Para las contadas ocasiones en las que necesitemos enviar mensajes entre aplicaciones, el sistema provee de una interfaz que permite tanto la transmisión como la recepción automática de los mismos, siempre y cuando la aplicación que recibe el mensaje esté preparada para ello.

Enviar un mensaje

Para la transmisión del mensaje utilizaremos un objeto que los seguidores de este curso ya conocen, el *intent*, que permite el paso de información a través de la *cola de eventos de Android*.

Al inicializar el objeto simplemente deberemos tener en cuenta que deberemos indicar el nombre del paquete y la clase que debe recibir el mensaje, así como la acción concreta que implementa el mensaje, que permitirá al receptor obtener correctamente los parámetros que incluyamos.

```
Intent intent = new Intent();  intent.setClassName(“NOMBRE_DEL_PAQUETE”,
“NOMBRE_DE_LA_CLASE”);  intent.setAction(“NOMBRE_DE_LA_ACCIÓN”);
intent.putExtra(“NOMBRE_DEL_PARÁMETRO_1”, “VALOR_DEL_PARÁMETRO_1”); …
intent.putExtra(“NOMBRE_DEL_PARÁMETRO_N”, “VALOR_DEL_PARÁMETRO_N”);
```

Finalmente, enviaremos el mensaje usando el procedimiento asíncrono *sendBroadcast*, que incorpora la superclase *Context*, y por ende también las clases *Activity* y *Service*, que son las que habitualmente tratan estos temas

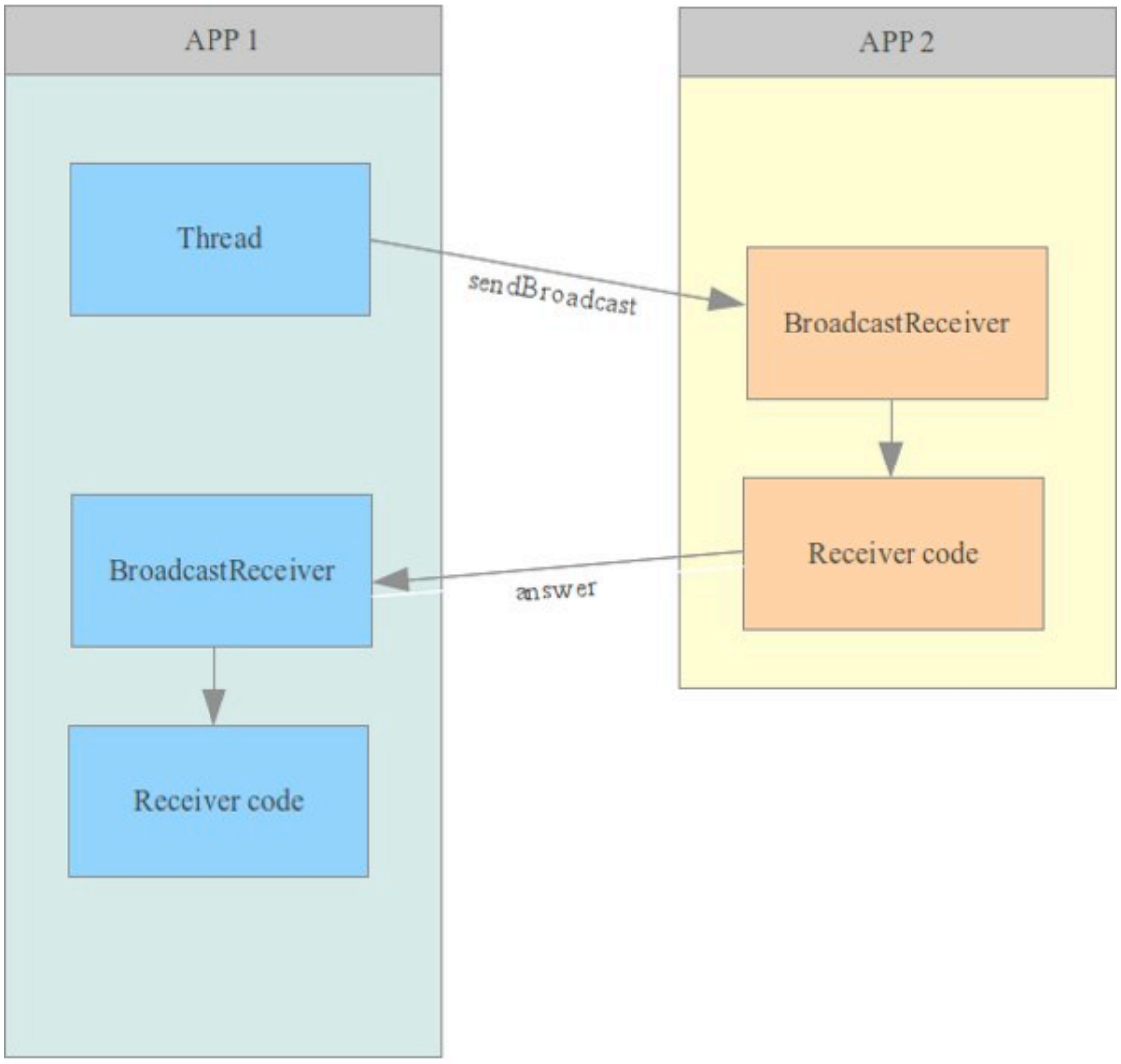
```
context.sendBroadcast(intent);
```

Recibir un mensaje

Para la recepción del mensaje deberemos usar el mecanismo indicado en la sesión anterior para la recepción de eventos del sistema, es decir, definir la acción en el manifiesto de nuestra aplicación, asociándolo a la misma clase que se usa al enviarlo, tal como se indica a continuación.

<receiver android:name="NOMBRE_DE_LA_CLASE" android:exported="true">
<action android:name="NOMBRE_DE_LA_ACCIÓN" /> </receiver>

Tal como se indica en el ejemplo anterior, no deberemos olvidar añadir el atributo *exported* que indica que la clase puede ser activada desde otras aplicaciones, tal como es el caso.



Devolución de resultados

En caso que la aplicación que recibe el mensaje deba devolver algún tipo de resultado, deberemos utilizar el mismo mecanismo descrito anteriormente (creación de un nuevo *intent* y envío a la primera aplicación, en la cual habremos definido un *BroadcastReceiver* que se encargará de procesar el resultado, de forma asíncrona).

Activar otras aplicaciones

En ocasiones puede resultar útil que vuestra aplicación active otras aplicaciones, bien se trate de aplicaciones de sistema o de aplicaciones que hayáis descargado de la *Play Store*, o de alguna otra tienda oficial.

En general Android no ofrece una interfaz de activación para las aplicaciones de sistema, por lo que un método que funciona correctamente en una versión puede no funcionar en la siguiente actualización del sistema, no obstante, los ejemplos que os proporcionamos a continuación se han mostrado funcionales.

Activar la bandeja de entrada de mensajes de texto

```
Intent intent = new Intent(); intent.setClassName("com.android.mms",
"com.android.mms.ui.ConversationList"); intent.setAction(Intent.ACTION_MAIN);
intent.addCategory(Intent.CATEGORY_DEFAULT);
intent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK  |
Intent.FLAG_ACTIVITY_SINGLE_TOP |          Intent.FLAG_ACTIVITY_CLEAR_TOP |
Intent.FLAG_ACTIVITY_NO_HISTORY |
Intent.FLAG_ACTIVITY_CLEAR_WHEN_TASK_RESET);
```

Activar el registro de llamadas

```
Intent intent = new Intent(); intent.setAction(Intent.ACTION_VIEW);
intent.setType(CallLog.Calls.CONTENT_TYPE);
intent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK  |
Intent.FLAG_ACTIVITY_SINGLE_TOP |          Intent.FLAG_ACTIVITY_CLEAR_TOP |
Intent.FLAG_ACTIVITY_NO_HISTORY |
Intent.FLAG_ACTIVITY_CLEAR_WHEN_TASK_RESET);
```

Podéis encontrar información sobre la activación de otras aplicaciones de sistema en *Google*, para lo que os recomendamos la siguiente búsqueda: "*open NOMBRE_APLICACIÓN +intent*" o en el **foro de desarrolladores stackoverflow.com**

En cuanto a las aplicaciones de usuario, y pese a que las interfases de acceso también pueden variar, el mecanismo es el siguiente:

1. Decompilar la aplicación para acceder al manifiesto de la misma.
2. Abrir el manifiesto con un editor de texto.

3. Los posibles puntos de entrada a la misma son aquellos que tengan asignado el atributo exported, tal como hemos indicado anteriormente.

Lógicamente, habrá que perder algún tiempo en comprobar los efectos que tiene la activación de cada uno de ellos, hasta encontrar el que hace lo que queremos que haga, si es que lo hay.

Algunas aplicaciones, como por ejemplo el calendario, generan mensajes que pueden ser capturados por otras aplicaciones, lo que permite no sólo su monitorización, sino también su sustitución, según el caso.

Dado que estos mensajes no son parte del núcleo de Android, no es posible obtener documentación de Google sobre este tema, por lo que su detección no es fácil y se realiza mediante la **monitorización de los logs de sistema** y el seguimiento del código fuente de las propias aplicaciones, que es posible obtener en **diversos repositorios, como el de CyanogenMOD**.

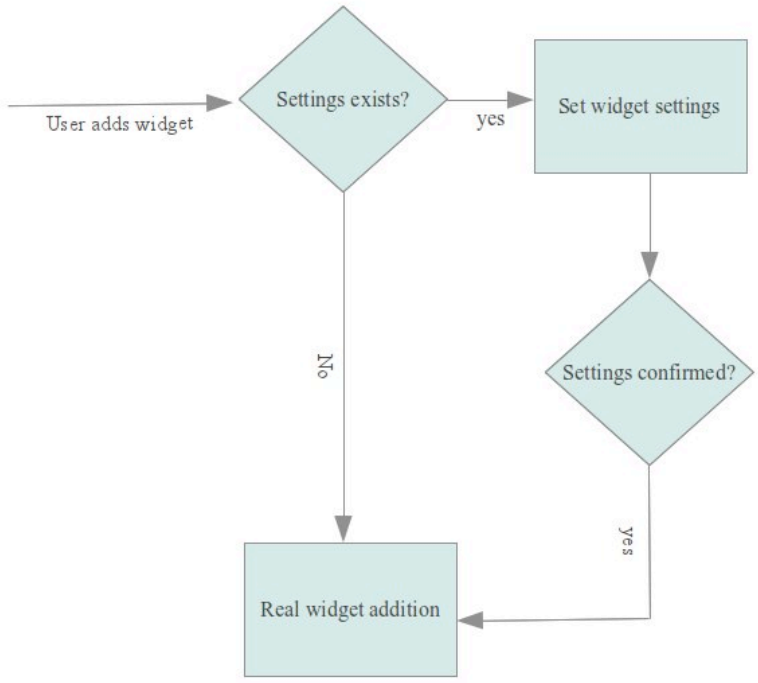
CAPÍTULO 8

Widgets

En los dos últimos capítulos de nuestro curso Android hemos explicado como el programador puede hacer que las aplicaciones capturen los eventos que genera el sistema, u otras aplicaciones, así como la manera en que deben generarse los eventos para que puedan ser capturados por otras aplicaciones, lo que se apartaba un poco del tema principal del curso pero que considerábamos interesante para entender el funcionamiento de Android.

Nos disponemos ahora a continuar desarrollando nuestro **explorador de archivos, al que en este capítulo proveeremos de un widget** que permitirá al usuario crear accesos directos a un directorio concreto, lo que nos permitirá no sólo explicaros cómo se crea y dibuja el widget, sino también cómo se visualiza la página de configuración del mismo.

En Android, **un widget es un cuadro gráfico que se posiciona en el escritorio**, y en algunos casos también en la pantalla de bloqueo, **y que permite a las aplicaciones mostrar información, a menudo dinámica, y a los usuarios acceder a funciones específicas de ésta.**



Para poder diseñar un widget es necesario responder primero a las siguientes preguntas:

- Qué queremos mostrar en el widget,
- Se puede configurar la información que aparece en el widget,
- Cuando se refrescará la información del widget,
- Cómo interactuará el usuario con el widget, si es que eso es posible,

Además, y como veremos a continuación, será necesario tener en cuenta al iniciar el proceso de actualización que **el usuario puede haber añadido más de un widget.**

Qué queremos mostrar en el widget

El contenido del widget se define de igual manera que el contenido de una actividad (ventana), es decir, **mediante un layout** que contendrá los diferentes elementos de éste.

Asimismo, también deberemos definir el widget en el manifiesto de la aplicación, tal como se indica a continuación.

La acción capturada (*android.appwidget.action.APPWIDGET_UPDATE*) sirve para indicar al sistema que la clase debe recibir los eventos de actualización del widget, tal y como ya indicamos en clases anteriores.

Adicionalmente, deberemos indicar las características básicas del widget, tales como el tamaño, el tiempo tras el cual debe actualizarse el widget y el layout inicial en el archivo *res/xml/widget.xml* (o el nombre que hayamos indicado en el manifiesto).

Se puede configurar la información que aparece en el widget

En caso que nuestro widget sea configurable, deberemos indicarlo en el archivo *res/xml/widget.xml*, lo cual forzará la creación de una actividad inicial en la que el usuario introduzca los parámetros de configuración de aquel.

Aunque podemos usar cualquier mecanismo para almacenar los parámetros de configuración del widget, lo más conveniente es utilizar la clase SharedPreferences, cuyo funcionamiento ya conocéis.

Adicionalmente, deberemos tener en cuenta que internamente Android implementa la llamada a la actividad de configuración mediante la ejecución del procedimiento *onActivityResult*, por lo que si la actividad de configuración acaba con un código de ejecución (*resultCode*) diferente de *Activity.RESULT_OK*, la adición del widget se cancelará.

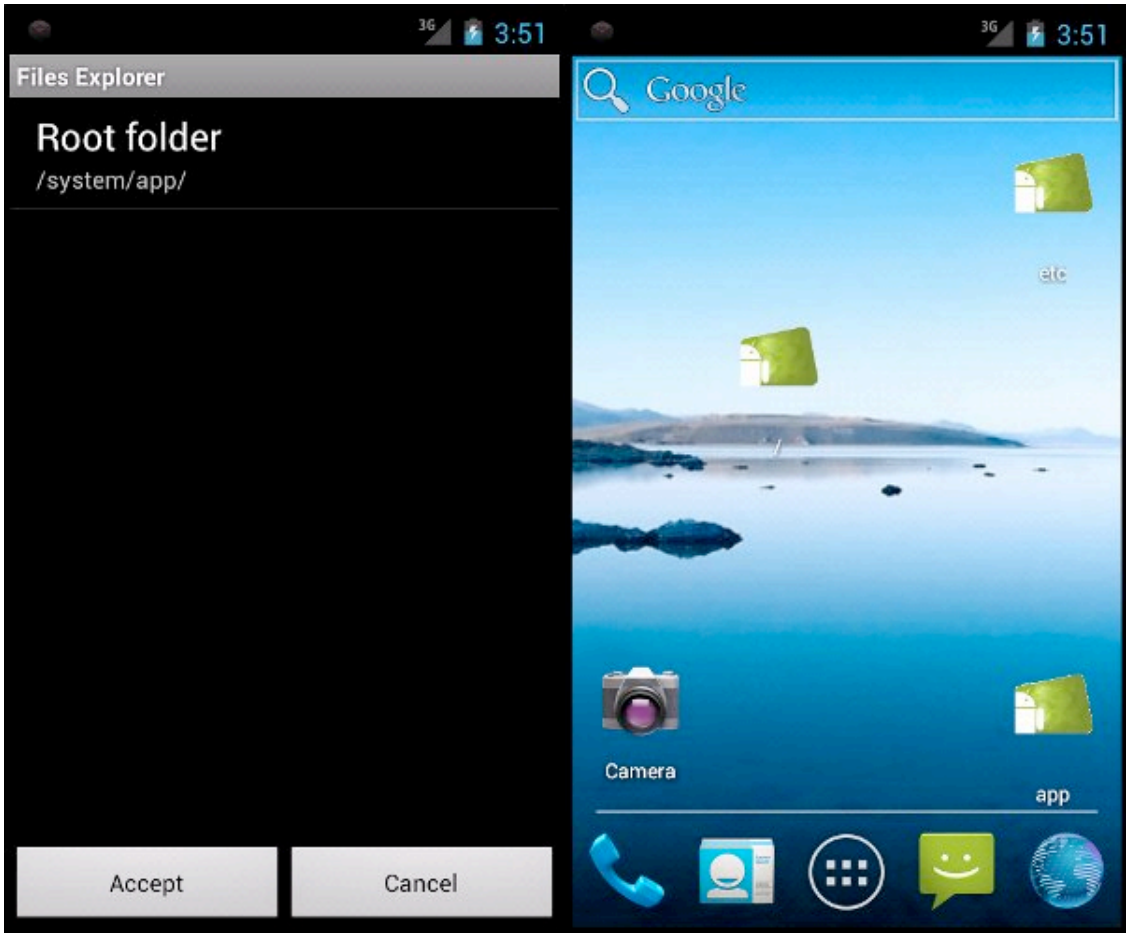
Cuándo se refrescará la información del widget

El widget se refresca cuando se activa el evento *android.appwidget.action.APPWIDGET_UPDATE*, que deberemos capturar convenientemente en el código que implemente nuestro widget (ver procedimiento *onUpdate* de la clase *Widget* en el ejemplo).

Es posible forzar la activación de dicho evento mediante el broadcast correspondiente, tal y como se muestra en el código de ejemplo.

Cómo interactuará el usuario con el widget, si es que eso es posible

Los widgets son objetos dinámicos en el sentido que su contenido puede ser modificado dinámicamente por la aplicación, mediante la captura del evento *APPWIDGET_UPDATE* enunciado en los párrafos anteriores.



No obstante, lo más probable es que nos interese la interacción con el usuario, habitualmente mediante el evento *ONCLICK* sobre la vista que contiene a nuestro widget, para lo cual deberemos usar el procedimiento *setOnClickPendingIntent*, que usaremos para que al clicar el widget se genere un broadcast, se active una ventana, etc.

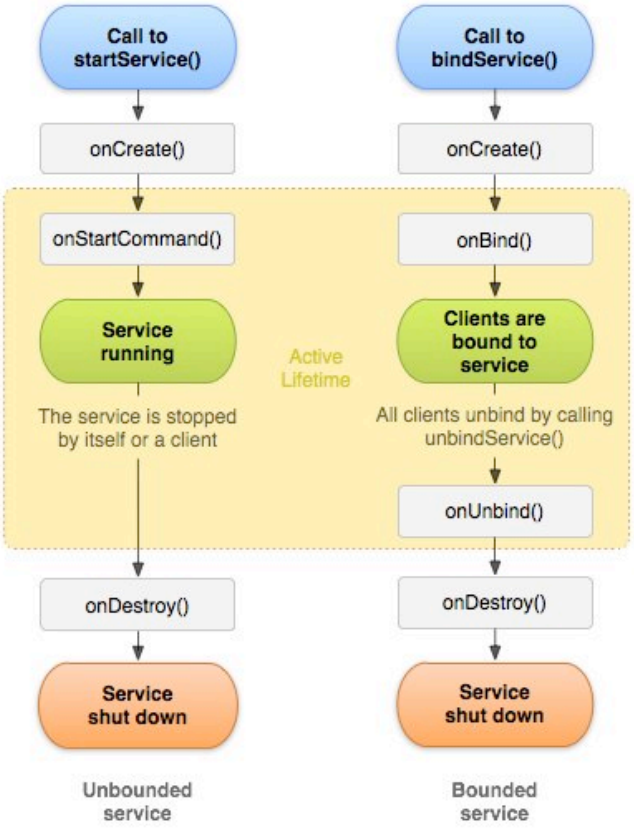
CAPÍTULO 9

Servicios

Un servicio es una sub-aplicación que se ejecuta en segundo plano, normalmente sin interacción con el usuario, y que realiza una serie de tareas dependiendo de la activación de determinados eventos externos (aquellos para cuya atención ha sido programado), tales como la recepción de un correo electrónico, la activación de un temporizador, etc.

En Android los servicios se ejecutan en el contexto del proceso al que pertenecen, que no es más que aquel que los ha creado, por lo que pueden acceder a cualquiera de los objetos creados por éste, a excepción de los relacionados con la interfase gráfica, que tal como sucedía con los Threads, son accesibles unicamente desde el flujo principal.

Tal y como se aprecia en la imagen inferior, el ciclo de vida de un servicio se sitúa entre la activación de los eventos onCreate y onDestroy, garantizando el sistema que una activación del evento onCreate asociado a un servicio se complementará mediante la activación del evento onDestroy correspondiente.



Aunque no es obligatorio, lo habitual es que el evento *onCreate* asociado a un servicio se encargue de la inicialización de un objeto de tipo *BroadcastReceiver* que se encargará de monitorizar los eventos (externos o internos) que activen o desactiven el servicio, mientras que el evento *onDestroy* se encargará de destruir el mencionado objeto.

Definición de un servicio

Al igual que ocurre con las actividades, los servicios deben definirse en el *manifiesto* de nuestra aplicación, tal como se indica a continuación, teniendo en cuenta que la clase que implemente el servicio deberá heredar de la superclase *Service*.

<service android:name=".ServiceClassName" />

Instanciación de un servicio

Al igual que ocurre con las actividades, **no pueden cohexistir en el sistema dos servicios de la misma clase**, por lo que antes de instanciar un servicio deberemos comprobar si éste ya se está ejecutando, procediendo a activarlo mediante un evento o a instanciarlo según sea el caso, tal como se muestra en el ejemplo.

boolean is_running = serviceRunning(); Intent intent = is_running ? new Intent() : new Intent(context, service_class); intent.setAction(action); if (is_running) context.sendBroadcast(intent); else context.startService(intent);

Destrucción de un servicio

La activación de un evento provoca que una porción de código asociada a un servicio se ejecute, pero éste permanece residente en memoria, a la espera de que el mismo u otro evento se active, hasta que el servicio es explicitamente destruido, lo cual puede hacerse mediante la llamada *stopService* (a la que se le pasa como parámetro el tipo de servicio a deterner) o mediante la llamada al procedimiento *stopSelf* asociada al servicio.

protected synchronized static void stopService(String service_name, Service service) { Debug.d(service_name, "Stopping service"); service.stopSelf(); } protected synchronized static void stopService(Context context, String service_name, Classservice_class) { Debug.d(service_name, "Stopping service"); context.stopService(new Intent(context, service_class)); }

Activando un evento en un servicio

Tal como hemos indicado con anterioridad, lo habitual es que todo servicio lleve asociado un objeto *BroadcastReceiver* que sirve como punto de entrada al servicio.

Esta instancia de la clase *BroadcastReceiver* implementa el código asociado al procesamiento de los eventos externos (del sistema) que atiende el servicio, así como

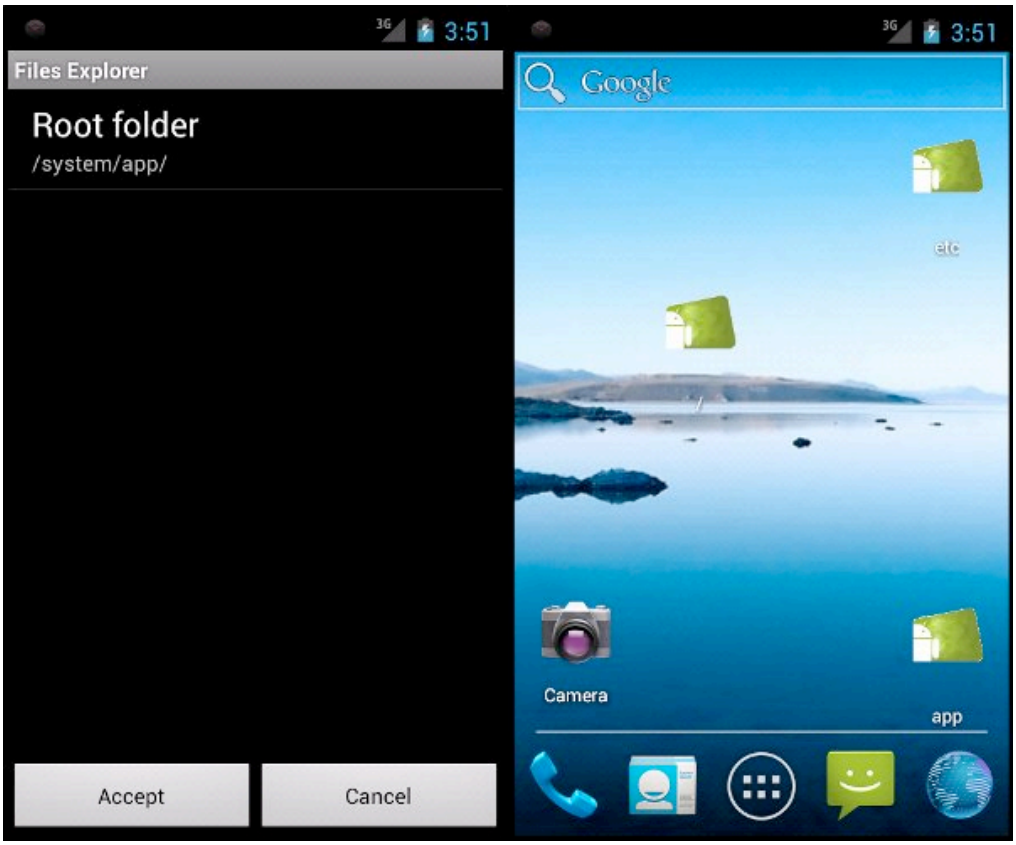
el código asociado a los eventos de la propia aplicación que pueden modificar el comportamiento del servicio, que suelen asociarse a eventos externos que se procesan parcialmente en otros contextos o a cambios en los parámetros de configuración de la aplicación.

Aunque no es obligatorio, es una buena práctica definir una operación estática asociada al servicio para cada uno de los puntos de entrada al mismo, de forma que sea en ese procedimiento en el que se compruebe si éste se encuentra o no en ejecución y se proceda a su creación o activación, según el caso.

```
public static synchronized void screenIsOff(Context context) {   boolean is_running =
isRunning();   Intent intent = is_running ? new Intent() : new Intent(context,
ScreenStateService.class);   intent.setAction(INTERNAL_ACTION_SCREEN_OFF);   if
(is_running) context.sendBroadcast(intent);   else context.startService(intent); }
```

En el ejemplo anterior, el *BroadcastReceiver* asociado al servicio captura los eventos de tipo *INTERNAL_ACTION_SCREEN_OFF*, lo que provoca la ejecución de la instancia del servicio, mientras que si el servicio no existe se procede a crearlo mediante el procedimiento *startService*.

Nótese que el procedimiento *screenIsOff* se define como *static*, lo que permite que sea ejecutado fuera del contexto de un servicio, y como *synchronized*, lo que impide que otro *Thread* ejecute cualquier otro procedimiento de la clase marcado como *synchronized* hasta que la ejecución de éste acabe, lo que garantiza acceso exclusivo a la clase mientras el servicio se crea.



Primera activación de un servicio tras su instanciación

Al instanciar un servicio, y justo después de activarse el evento *onCreate*, se produce la activación del evento *onStartCommand*, que recibe como parámetro un *Intent* que puede procesarse usando el *BroadcastReceiver* que asociemos al servicio.

```
public int onStartCommand(Intent intent, int flags, int start_id) {
iBroadcastReceiver.onReceive(this, intent);   return super.onStartCommand(intent,
flags, start_id); }
```

Y hasta aquí las características principales de los servicios, **que podrás ver en detalle en nuestro programa de ejemplo**, que hemos adaptado para que el repintado del widget se asocie a uno, lo que en condiciones reales permitiría ejecutar procedimientos y cálculos complejos ya que, como recordarás, **el tiempo de proceso máximo asociado a los eventos del sistema es de cinco segundos**.

CAPÍTULO 10

Obtener permiso de superusuario (root)

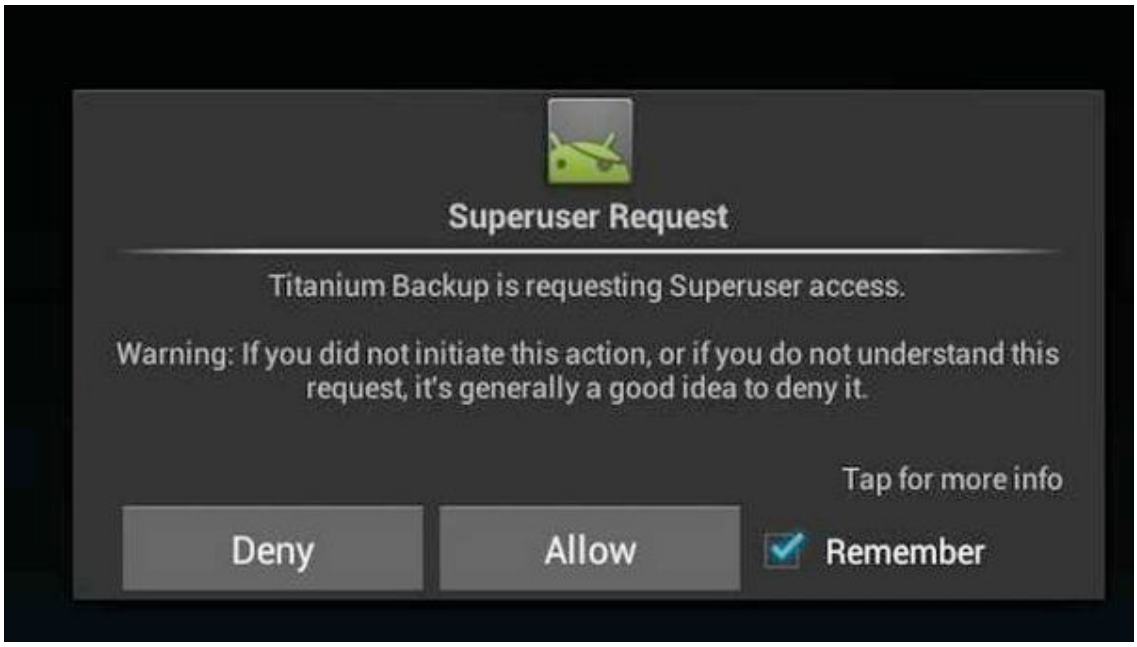
En el curso Android que acabamos en este capítulo hemos explicado las estructuras básicas que os permitirán implementar vuestras aplicaciones, pasando por las actividades y los threads, y sin olvidar los servicios, los permisos y el paso de mensajes entre aplicaciones.

En este bloque, y pese a que probablemente no os será de utilidad más que en unos pocos casos, queremos explicaros cómo conseguir privilegios de administrador en vuestras aplicaciones, lo que os permitirá desarrollar aplicaciones que se ejecuten sin las restricciones impuestas por el sistema.

Como hemos indicado en muchas ocasiones en nuestros posts de índole general, **Android es un sistema estructurado en capas, cada una de las cuales interacciona con las capas superior e inferior e incorpora un nivel de abstracción diferente.**

En concreto, **las aplicaciones se ejecutan sobre Dalvik, la máquina virtual que incorpora Android** y que se encarga de la ejecución de las aplicaciones, incluyendo tanto las que pertenecen al sistema como al usuario, y delegando en el sistema la gestión de procesos y memoria.

Dalvik ejecuta las aplicaciones en el contexto del usuario al que pertenecen, que se crea en el momento en que éstas se instalan y esto no puede cambiarse, **lo que imposibilita la inclusión de código privilegiado directamente en las aplicaciones y obliga a la creación de un proceso separado**, como veremos más adelante.



Cómo funciona el root

La adquisición del root en terminales Android es realmente ingeniosa, y se divide en dos partes claramente diferenciadas.

- **Por un lado un programa que pertenece al usuario root y que tiene activo el bit SUID**, lo cual permite que su ejecución se realice usando el código de usuario al que pertenece el programa (root en este caso) y no el código de usuario que realiza la llamada (el asignado a nuestra aplicación), y que lo único que hace es ejecutar el programa que recibe como parámetro, habitualmente un shell, una vez comprobado que el usuario ha decidido permitirlo,
- **Por otro lado una aplicación, con interfaz gráfica, y que no dispone de ningún privilegio especial**, que se encarga de mostrar un diálogo al usuario cada vez que una aplicación desea obtener privilegios de superusuario y que simplifica la gestión de dichos permisos, la visualización de logs, etc.

Aplicaciones como SuperUser y SuperSU implementan esta interfaz gráfica, todo y que, bajo determinadas situaciones, son capaces de actualizar y/o descargar también el archivo binario asociado.

Obtención del root en una aplicación

Como hemos indicado anteriormente, **la obtención de los permisos de superusuario no puede hacerse en el contexto del proceso que se está ejecutando**, sino que deberemos ejecutar un proceso externo, que será el que ejecute el código privilegiado y del que deberemos capturar la salida para comprobar si la ejecución ha sido correcta o no.

Pese a que podemos ejecutar un comando cada vez, **lo más habitual es que el comando que se ejecute sea una shell, lo que permitirá la ejecución de varios comandos de forma secuencial**, pudiendo capturar la salida de cada uno de ellos de forma separada.

Esta forma de actuar permite, además, que **la solicitud de los permisos de superusuario** (por medio de la aplicación **SuperUser** o **SuperSU**) **se realice** una única vez, **cuando se inicia el shell**, y no cada vez que se ejecuta un comando.

```
iProcess = Runtime.getRuntime().exec("su -c sh");
```

Capturando la salida del proceso privilegiado

Todo proceso Unix estándar dispone de dos canales por los que expulsa el resultado de las operaciones que ejecuta: el canal de salida, que es aquel en el que se imprimen el resultado de las operaciones print, write y, en general, aquellas funciones y procedimientos que imprimen resultados **y, el canal de errores**, en el que se escriben los códigos y mensajes de error y advertencias.

Adicionalmente, es necesario abrir un canal que permita enviar los comandos al proceso privilegiado que acabamos de crear, y que en Unix se conoce como **canal de entrada del proceso**.

```
iDataOutputStream = new DataOutputStream(iProcess.getOutputStream());
iDataInputStream = new DataInputStream(iProcess.getInputStream());
iDataErrorStream = new DataInputStream(iProcess.getErrorStream());
```

Enviando comandos al proceso privilegiado

Una vez abierto el canal de entrada estándar del proceso privilegiado, la ejecución de comandos se realiza cada vez que enviamos una línea de texto por dicho canal.

```
public boolean execute(String command) {   iDataOutputStream.writeBytes(command + “\n”);   iDataOutputStream.flush(); }
```

Recibiendo resultados del proceso privilegiado

Del mismo modo, la recepción de resultados se realiza leyendo de los canales de salida y errores, tal como se muestra más abajo.

```
public List getStandardOutput() {   List results = new ArrayList();   String line;   while (true)   {     line = iDataInputStream.readline();     if (line == null) break;   results.add(line);   }   return results; }   public List getStandardErrorsOutput() {   List results = new ArrayList();   String line;   while (true)   {     line = iDataErrorStream.readline();     if (line == null) break;     results.add(line);   }   return results; }
```

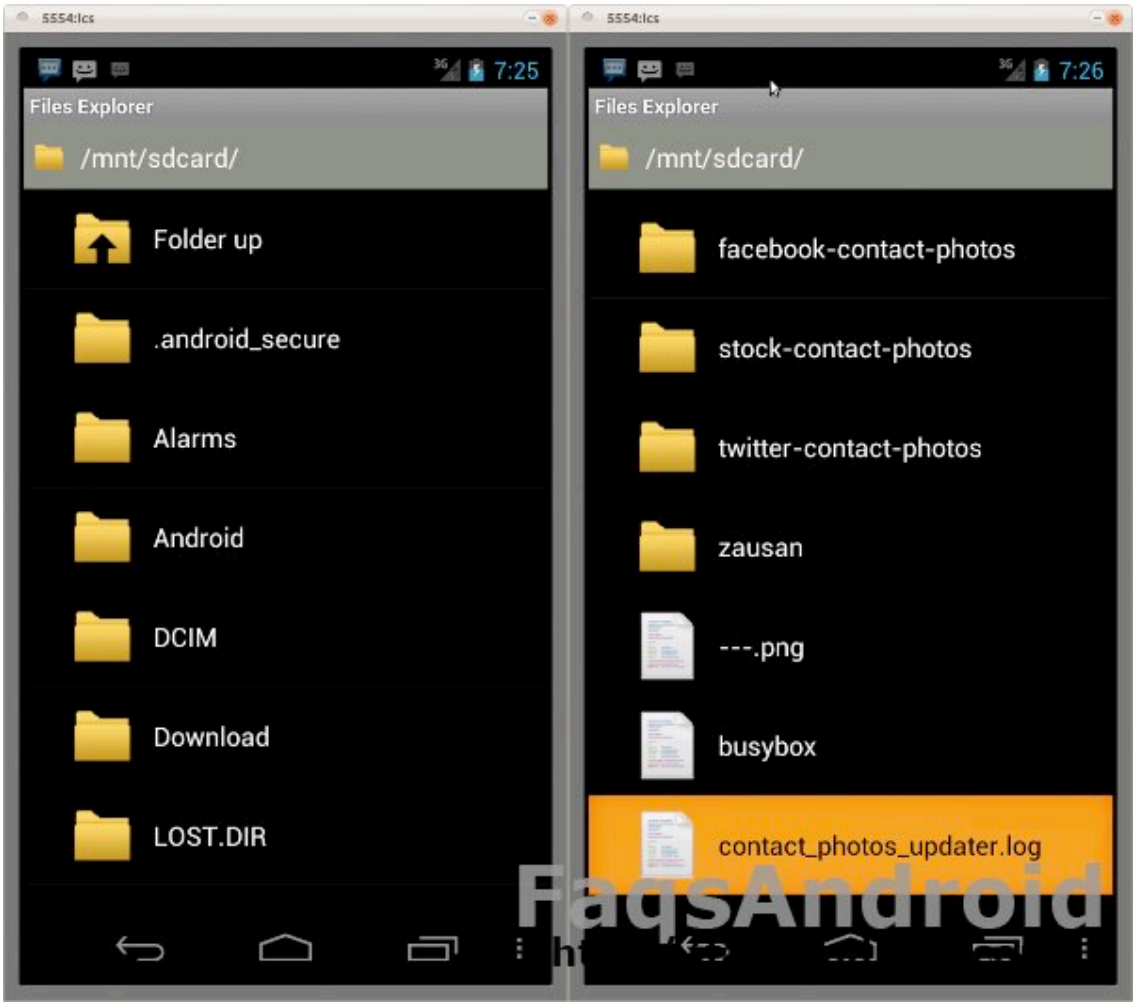
Comprobación de los permisos

El método más sencillo para comprobar si la obtención de los permisos de superusuario ha sido exitosa es la ejecución del comando *id*, que nos proporcionará información sobre el usuario que está ejecutando la aplicación y que debería ser 0 en caso afirmativo.

```
execute(“id”);   List output = getStandardOutput();   if ((output == null) || (output.isEmpty()))     throw new Exception("Can't get root access or denied by user");   if (! output.toString().contains("uid=0"))     throw new Exception("Root access rejected by user or device isn't rooted");
```

Acabar la sesión

Una vez que se han ejecutado todos los comandos necesarios, será conveniente finalizar la sesión de shell, mediante la ejecución del comando *exit*, que provocará la finalización del proceso asociado y el cierre de los canales de comunicaciones asociados al mismo.



Con esto damos por finalizado nuestro curso Android, no sin antes agradeceros vuestra atención y animaros a **descargaros la última versión del mismo**, que incluye el código completo necesario para conseguir permisos de superusuario en vuestras aplicaciones.

El **CURSO DE PROGRAMACIÓN EN ANDROID PARA PRINCIPIANTES** ha sido desarrollado por Robert P. para **FaqsAndroid.com** del **Grupo bemoob**.

Este obra está bajo una **Licencia Creative Commons Atribución-NoComercial-SinDerivadas 3.0 Unported**.

by