



MANUAL PROGRAMACIÓN ANDROID

SALVADOR GÓMEZ OLIVER

WWW.SGOLIVER.NET

Versión 3.0

ATENCIÓN: DOCUMENTO DE MUESTRA

Esto sólo es un fragmento de muestra del Curso de Programación Android publicado en la web de [sgoliver.net](http://www.sgoliver.net)

Si desea más información sobre el curso completo puede acceder a su página oficial mediante el siguiente enlace:

http://www.sgoliver.net/blog/?page_id=2935

Versión 3.0 // Junio 2013

Este curso también está disponible **online**.

Es posible que exista una versión más reciente de este documento o que puedas encontrar contenido web actualizado.

Para más información te recomiendo que visites la web oficial del curso:

http://www.sgoliver.net/blog/?page_id=2935

INDICE DE CONTENIDOS

PRÓLOGO.....	6
¿A QUIÉN VA DIRIGIDO ESTE LIBRO?.....	7
LICENCIA.....	7

I. Conceptos Básicos

Entorno de desarrollo Android.....	9
Estructura de un proyecto Android.....	15
Componentes de una aplicación Android.....	25
Desarrollando una aplicación Android sencilla.....	26

II. Interfaz de Usuario

Layouts.....	42
Botones.....	48
Imágenes, etiquetas y cuadros de texto.....	51
Checkboxes y RadioButtons.....	55
Listas Desplegables.....	58
Listas.....	62
Optimización de listas.....	67
Grids.....	70
Pestañas.....	72
Controles personalizados: Extender controles.....	76
Controles personalizados: Combinar controles.....	79
Controles personalizados: Diseño completo.....	86
Fragments.....	92
Action Bar: Funcionamiento básico.....	102
Action Bar: Tabs.....	106

III. Widgets

Widgets básicos.....	112
Widgets avanzados.....	116

IV. Menús

Menús y Submenús básicos.....	127
Menús Contextuales.....	131
Opciones avanzadas de menú.....	136

V. Tratamiento de XML

Tratamiento de XML con SAX.....	143
Tratamiento de XML con SAX Simplificado.....	151
Tratamiento de XML con DOM.....	154
Tratamiento de XML con XmlPull.....	158
Alternativas para leer/escribir XML (y otros ficheros).....	160

VI. Bases de Datos

Primeros pasos con SQLite.....	165
Insertar/Actualizar/Eliminar registros de la BD.....	170
Consultar/Recuperar registros de la BD.....	172

VII. Preferencias en Android

Preferencias Compartidas.....	176
Pantallas de Preferencias.....	178

VIII. Localización Geográfica

Localización Geográfica Básica.....	188
Profundizando en la Localización Geográfica.....	193

IX. Mapas en Android

Preparativos y ejemplo básico.....	200
Opciones generales del mapa.....	210
Eventos, marcadores y dibujo sobre el mapa.....	215

X. Ficheros en Android

Ficheros en Memoria Interna.....	223
Ficheros en Memoria Externa (Tarjeta SD).....	226

XI. Content Providers

Construcción de Content Providers.....	231
Utilización de Content Providers.....	239

XII. Notificaciones Android

Notificaciones Toast.....	245
Notificaciones de la Barra de Estado.....	249
Cuadros de Diálogo.....	251

XIII. Tareas en Segundo Plano

Hilos y Tareas Asíncronas (Thread y AsyncTask).....	259
IntentService.....	266

XIV. Acceso a Servicios Web

Servicios Web SOAP: Servidor.....	271
Servicios Web SOAP: Cliente.....	279
Servicios Web REST: Servidor.....	290
Servicios Web REST: Cliente.....	297

XV. Notificaciones Push

Introducción a Google Cloud Messaging.....306

Implementación del Servidor.....310

Implementación del Cliente Android.....316

XVI. Depuración en Android

Logging en Android.....325

PRÓLOGO

Hay proyectos que se comienzan sin saber muy bien el rumbo exacto que se tomará, ni el destino que se pretende alcanzar. Proyectos cuyo único impulso es el día a día, sin planes, sin reglas, tan solo con el entusiasmo de seguir adelante, a veces con ganas, a veces sin fuerzas, pero siempre con la intuición de que va a salir bien.

El papel bajo estas líneas es uno de esos proyectos. Nació casi de la casualidad allá por 2010. Hoy, varios años después, sigue más vivo que nunca.

A pesar de llevar metido en el desarrollo para Android casi desde sus inicios, en mi blog [sgoliver.net] nunca había tratado estos temas, pretendía mantenerme fiel a su temática original: el desarrollo bajo las plataformas Java y .NET. Surgieron en algún momento algunos escarceos con otros lenguajes, pero siempre con un ojo puesto en los dos primeros.

Mi formación en Android fue en inglés. No había alternativa, era el único idioma en el que, por aquel entonces, existía buena documentación sobre la plataforma. Desde el primer concepto hasta el último tuve que aprenderlo en el idioma de Shakespeare. A día de hoy esto no ha cambiado mucho, la buena documentación sobre Android, la buena de verdad, sigue y seguirá aún durante algún tiempo estando en inglés, pero afortunadamente son ya muchas las personas de habla hispana las que se están ocupando de ir equilibrando poco a poco esta balanza de idiomas.

Y con ese afán de aportar un pequeño granito de arena a la comunidad hispanohablante es como acabé decidiendo dar un giro, quien sabe si temporal o permanente, a mi blog y comenzar a escribir sobre desarrollo para la plataforma Android. No sabía hasta dónde iba a llegar, no sabía la aceptación que tendría, pero lo que sí sabía es que me apetecía ayudar un poco a los que como yo les costaba encontrar información básica sobre Android disponible en su idioma.

Hoy, gracias a todo vuestro apoyo, vuestra colaboración, vuestras propuestas, y vuestras críticas (de todo se aprende) éste es un proyecto con varios años ya de vida. Más de 300 páginas, más de 50 artículos, y sobre todo cientos de comentarios de ánimo recibidos.

Y este documento no es un final, es sólo un punto y seguido. Este libro es tan solo la mejor forma que he encontrado de mirar atrás, ordenar ideas, y pensar en el siguiente camino a tomar, que espero sea largo. Espero que muchos de vosotros me acompañéis en parte de ese camino igual que lo habéis hecho en el recorrido hasta ahora.

Muchas gracias, y que comience el espectáculo.

¿A QUIÉN VA DIRIGIDO ESTE LIBRO?

Este manual va dirigido a todas aquellas personas interesadas en un tema tan en auge como la programación de aplicaciones móviles para la plataforma Android. Se tratarán temas dedicados a la construcción de aplicaciones nativas de la plataforma, dejando a un lado por el momento las aplicaciones web. Es por ello por lo que el único requisito indispensable a la hora de utilizar este manual es tener conocimientos bien asentados sobre el lenguaje de programación Java y ciertas nociones sobre aspectos básicos del desarrollo actual como la orientación a objetos.

LICENCIA

© **Salvador Gómez Oliver**. Todos los derechos reservados.

Queda prohibida la reproducción total o parcial de este documento, así como su uso y difusión, sin el consentimiento previo de su autor.

Por favor, respeta los derechos de autor. Si quieres emplear alguno de los textos o imágenes de este documento puedes solicitarlo por correo electrónico a la siguiente dirección: [sgoliver.net @ gmail.com](mailto:sgoliver.net@gmail.com)

1

Conceptos Básicos

I. Conceptos Básicos

Entorno de desarrollo Android

En este apartado vamos a describir los pasos básicos para disponer en nuestro PC del entorno y las herramientas necesarias para comenzar a programar aplicaciones para la plataforma Android.

No voy a ser exhaustivo, ya existen muy buenos tutoriales sobre la instalación de Eclipse y Android, incluida la [documentación oficial](#) de la plataforma. Además, si has llegado hasta aquí quiero suponer que tienes unos conocimientos básicos de Eclipse y Java, por lo que tan sólo enumeraré los pasos necesarios de instalación y configuración, y proporcionaré los enlaces a las distintas herramientas. Vamos allá.

Paso 1. Descarga e instalación de Java.

Si aún no tienes instalado ninguna versión del JDK (*Java Development Kit*) puedes descargar la última versión desde la [web de Oracle](#).

Java SE Downloads



En el momento de escribir este manual la versión más reciente disponible es la 7 update7, que podremos descargar para nuestra versión del sistema operativo, en mi caso la versión para Windows 64 bits.

Product / File Description	File Size	Download
Linux x86	120.62 MB	jdk-7u7-linux-i586.rpm
Linux x86	92.86 MB	jdk-7u7-linux-i586.tar.gz
Linux x64	118.8 MB	jdk-7u7-linux-x64.rpm
Linux x64	91.59 MB	jdk-7u7-linux-x64.tar.gz
Mac OS X	143.46 MB	jdk-7u7-macosx-x64.dmg
Solaris x86	135.4 MB	jdk-7u7-solaris-i586.tar.Z
Solaris x86	91.86 MB	jdk-7u7-solaris-i586.tar.gz
Solaris x64	22.51 MB	jdk-7u7-solaris-x64.tar.Z
Solaris x64	14.95 MB	jdk-7u7-solaris-x64.tar.gz
Solaris SPARC	135.69 MB	jdk-7u7-solaris-sparc.tar.Z
Solaris SPARC	95.15 MB	jdk-7u7-solaris-sparc.tar.gz
Solaris SPARC 64-bit	22.75 MB	jdk-7u7-solaris-sparcv9.tar.Z
Solaris SPARC 64-bit	17.47 MB	jdk-7u7-solaris-sparcv9.tar.gz
Windows x86	88.36 MB	jdk-7u7-windows-i586.exe
Windows x64	90 MB	jdk-7u7-windows-x64.exe

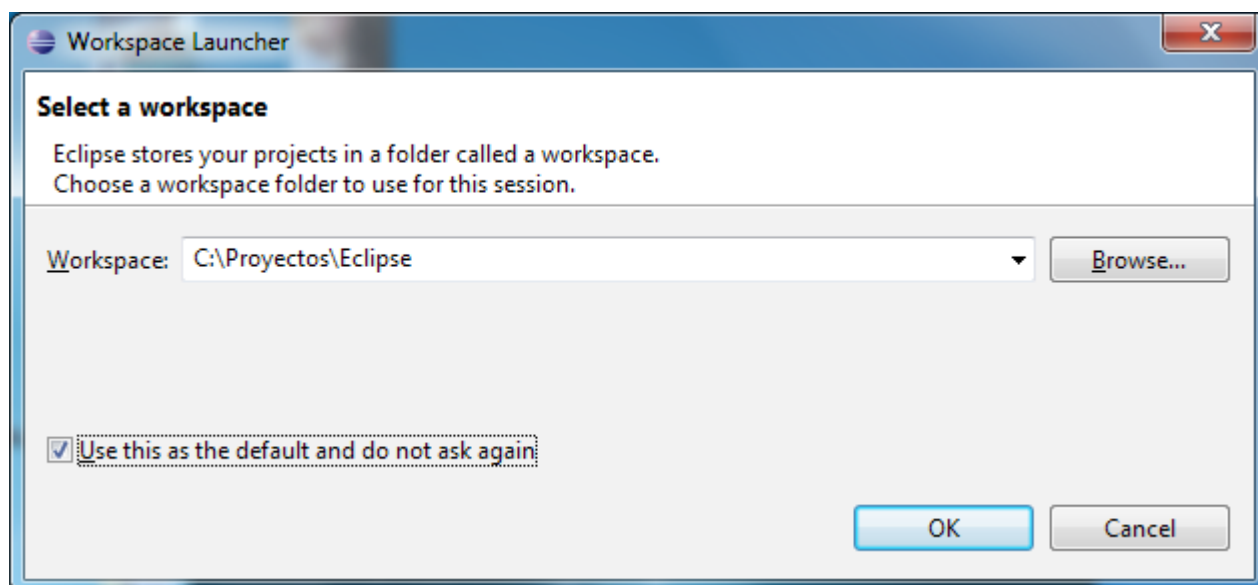
La instalación no tiene ninguna dificultad ya que es un instalador estándar de Windows donde tan sólo hay que aceptar las opciones que ofrece por defecto.

Paso 2. Descarga e instalación de Eclipse.

Si aún no tienes instalado Eclipse, puedes descargar la última versión, la 4.2.1 [Eclipse Juno SR1] en la última revisión de este texto, desde [este enlace](#). Recomiendo descargar la versión *Eclipse IDE for Java Developers*, y por supuesto descargar la versión apropiada para tu sistema operativo (Windows/Mac OS/Linux, y 32/64 bits). Durante el curso siempre utilizaré Windows 64 bits.

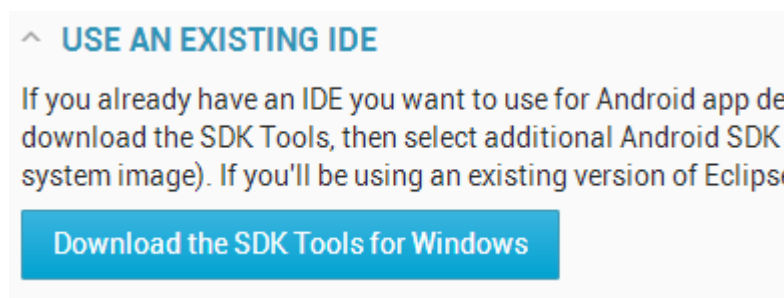


La instalación consiste simplemente en descomprimir el zip descargado en la ubicación deseada. Para ejecutarlo accederemos al fichero eclipse.exe dentro de la ruta donde hayamos descomprimido la aplicación, por ejemplo `c:\eclipse\eclipse.exe`. Durante la primera ejecución de la aplicación nos preguntará cuál será la carpeta donde queremos almacenar nuestros proyectos. Indicaremos la ruta deseada y marcaremos la check "Use this as the default" para que no vuelva a preguntarlo.



Paso 3. Descargar el SDK de Android.

El SDK de la plataforma Android se puede descargar desde [aquí](#) (en el momento de revisar este texto la última versión es la r21, que funciona perfectamente con Eclipse 4.2.1). Una vez descargado, bastará con ejecutar el instalador estándar de Windows.



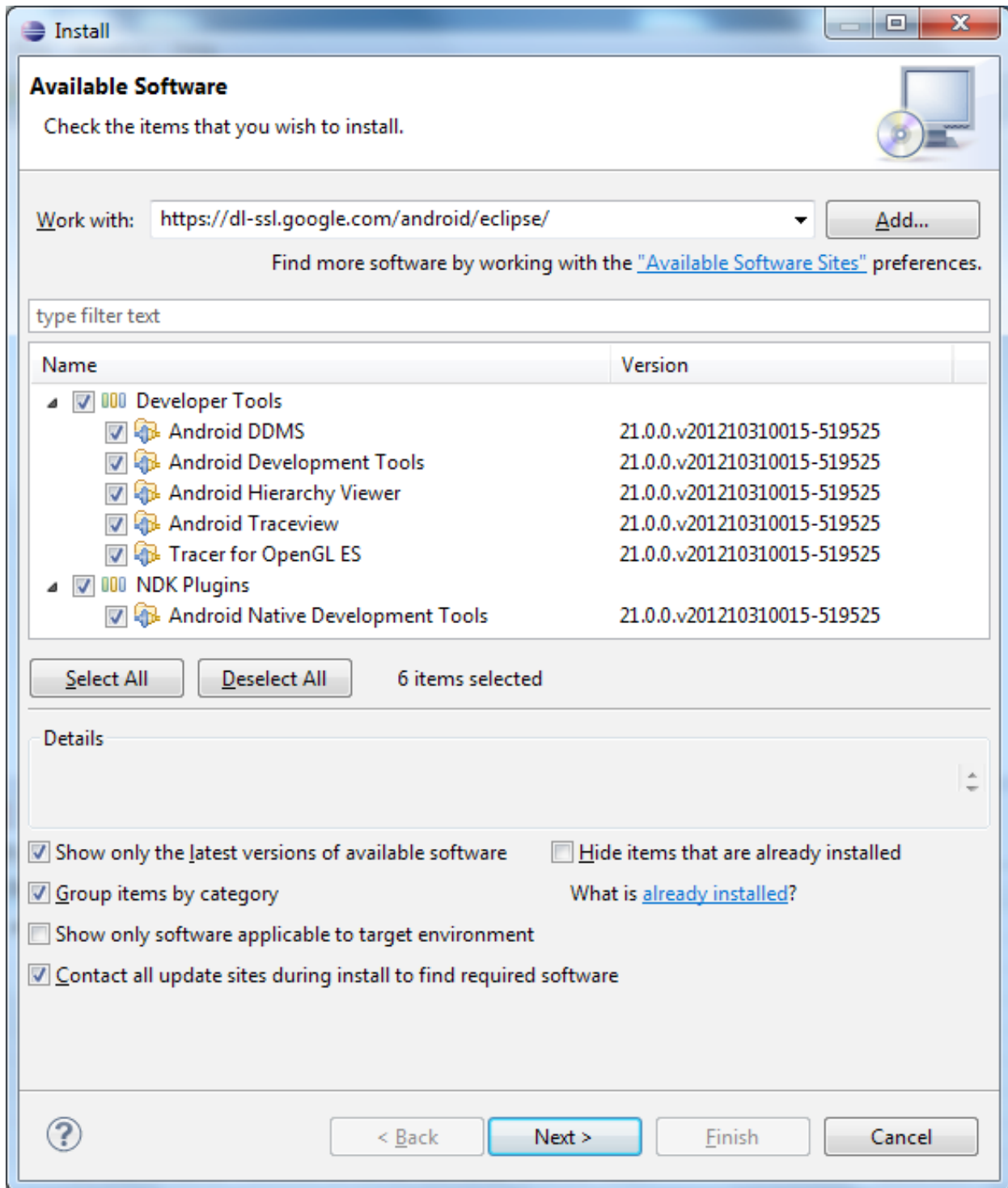
Paso 4. Descargar el plugin de Android para Eclipse.

Google pone a disposición de los desarrolladores un plugin para Eclipse llamado *Android Development Tools* (ADT) que facilita en gran medida el desarrollo de aplicaciones para la plataforma. Podéis descargarlo

mediante las opciones de actualización de Eclipse, accediendo al menú "Help / Install new software..." e indicando la siguiente URL de descarga:

<https://dl-ssl.google.com/android/eclipse/>

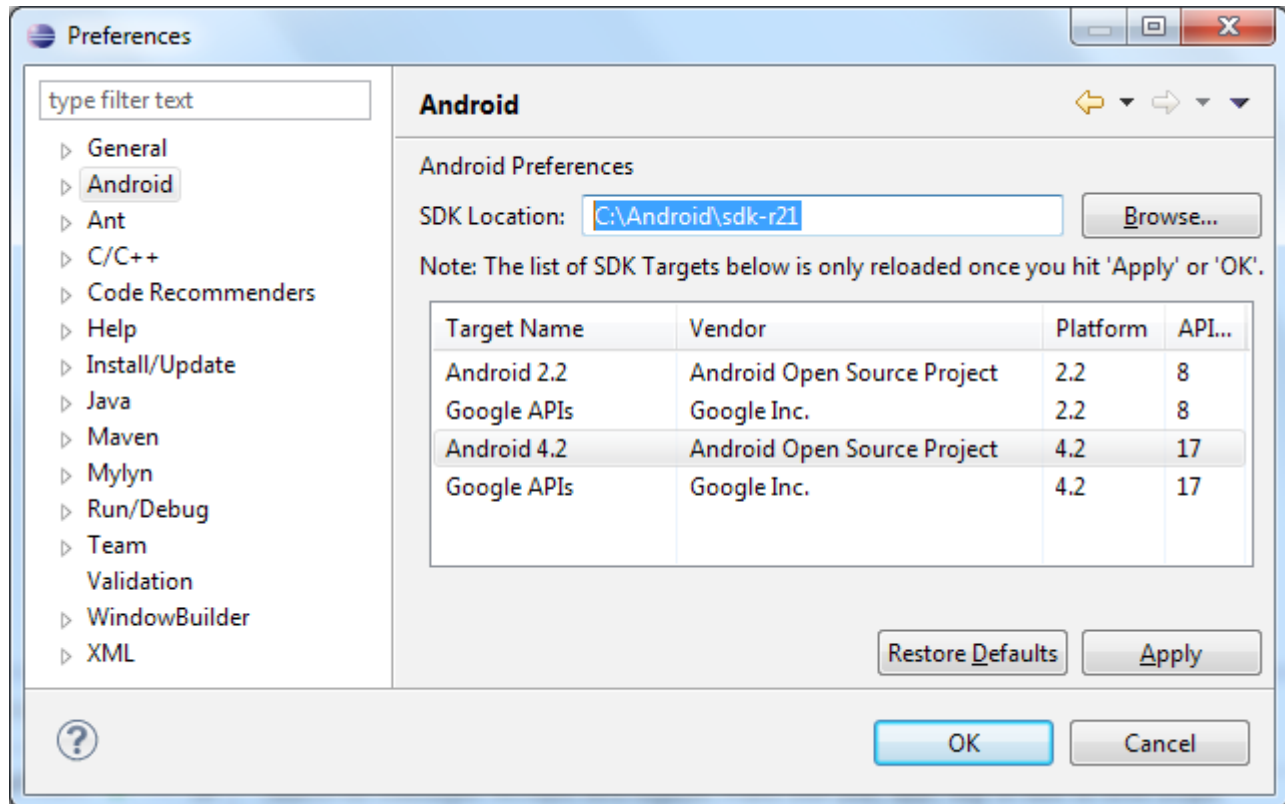
Seleccionaremos los dos paquetes disponibles "Developer Tools" y "NDK Plugins" y pulsaremos el botón "Next>" para comenzar con el asistente de instalación.



Durante la instalación Eclipse te pedirá que aceptes la licencia de los componentes de Google que vas a instalar y es posible que aparezca algún mensaje de *warning* que simplemente puedes aceptar para continuar con la instalación. Finalmente el instalador te pedirá que reinicies Eclipse.

Paso 5. Configurar el plugin ADT.

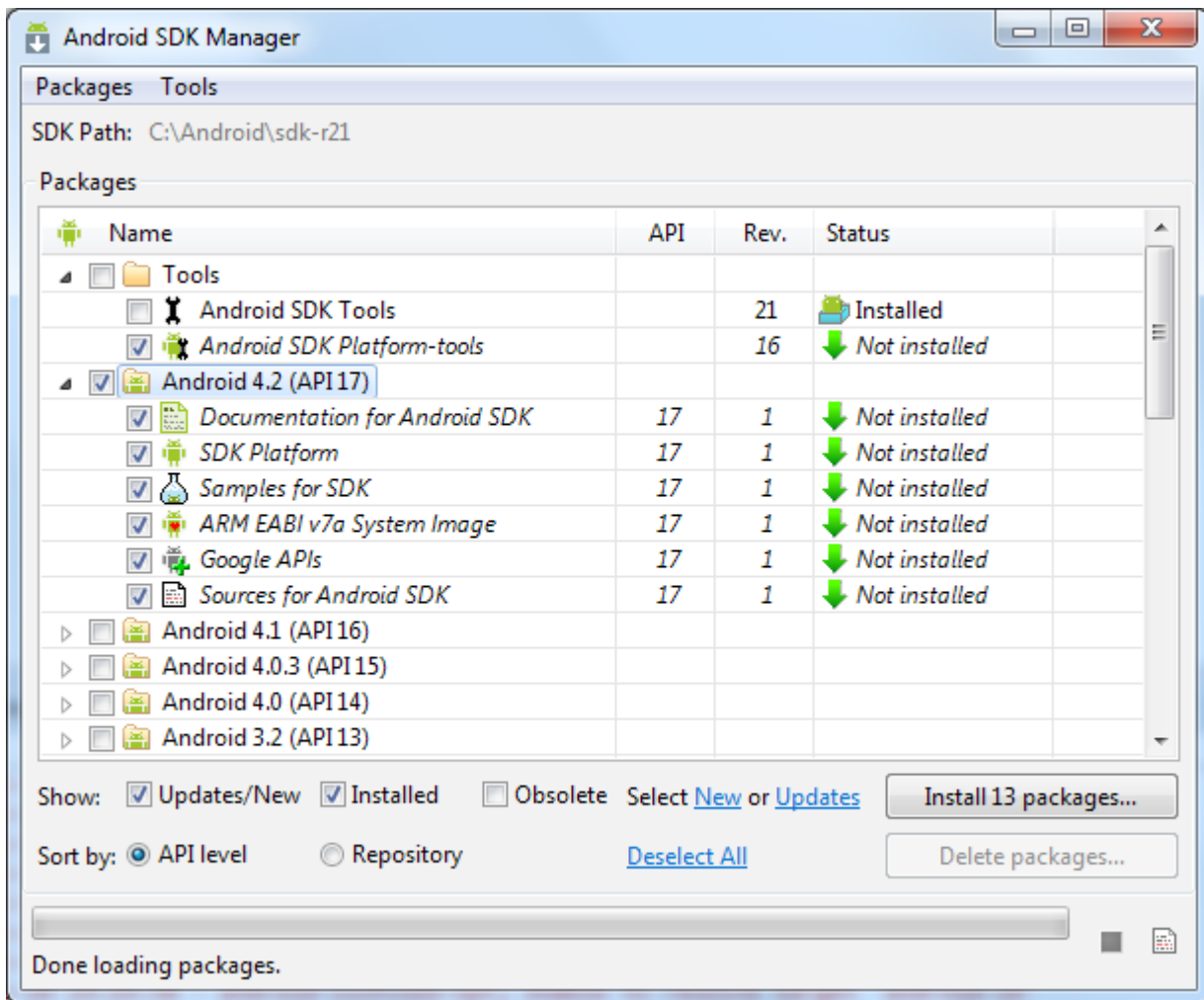
Una vez instalado el *plugin*, tendremos que configurarlo indicando la ruta en la que hemos instalado el SDK de Android. Para ello, iremos a la ventana de configuración de Eclipse (Window / Preferences...), y en la sección de Android indicaremos la ruta en la que se ha instalado. Finalmente pulsaremos OK para aceptar los cambios. Si aparece algún mensaje de *warning* aceptamos sin más, ya que se son problemas que se solucionarán en el siguiente paso.



Paso 6. Instalar las Platform Tools y los Platforms necesarios.

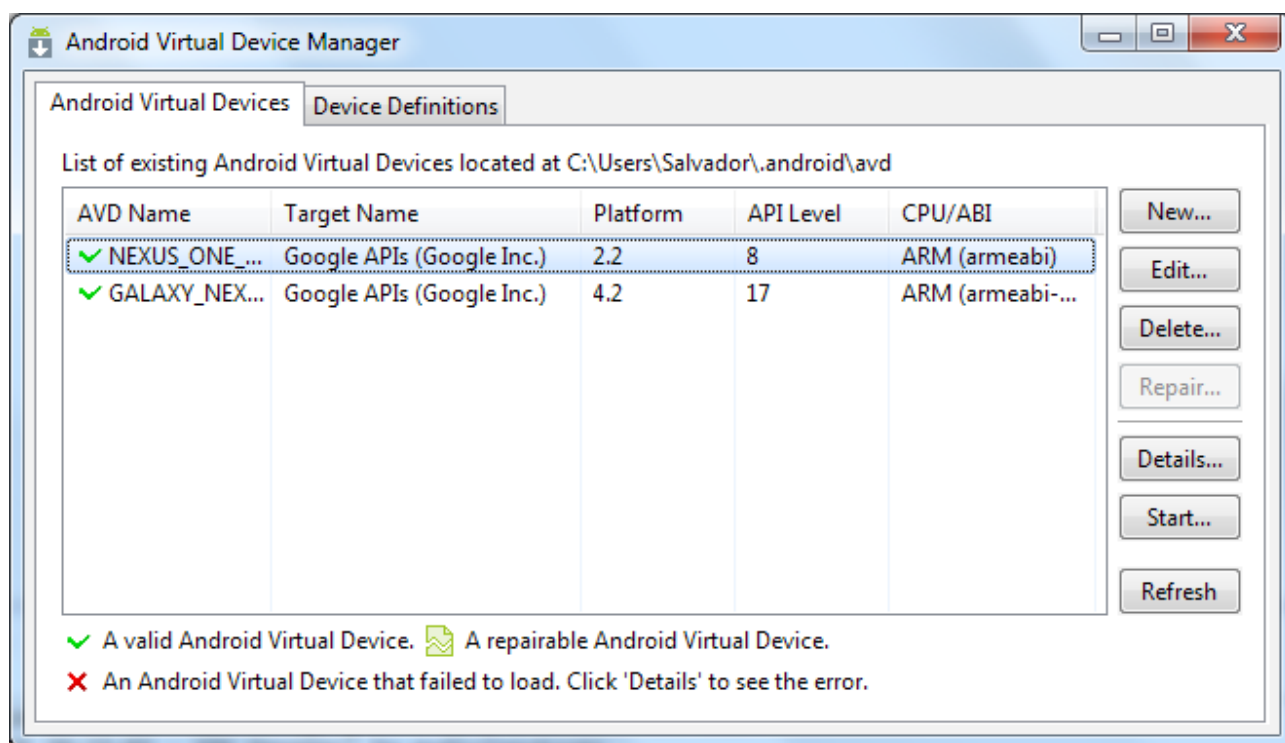
Además del SDK de Android comentado en el paso 2, que contiene las herramientas básicas para desarrollar en Android, también deberemos descargar las llamadas *Platform Tools*, que contiene herramientas específicas de la última versión de la plataforma, y una o varias plataformas (*SDK Platforms*) de Android, que no son más que las librerías necesarias para desarrollar sobre cada una de las versiones concretas de Android. Así, si queremos desarrollar por ejemplo para Android 2.2 tendremos que descargar su plataforma correspondiente. Mi consejo personal es siempre instalar al menos 2 plataformas: la correspondiente a la última versión disponible de Android, y la correspondiente a la mínima versión de Android que queremos que soporte nuestra aplicación.

Para ello, desde Eclipse debemos acceder al menú "Window / Android SDK Manager". En la lista de paquetes disponibles seleccionaremos las "Android SDK Platform-tools", las plataformas "Android 4.2 (API 17)" y "Android 2.2 (API 8)", y el paquete extra "Android Support Library", que es una librería que nos permitirá utilizar en versiones antiguas de Android características introducidas por versiones más recientes. Pulsaremos el botón "Install packages..." y esperaremos a que finalice la descarga.

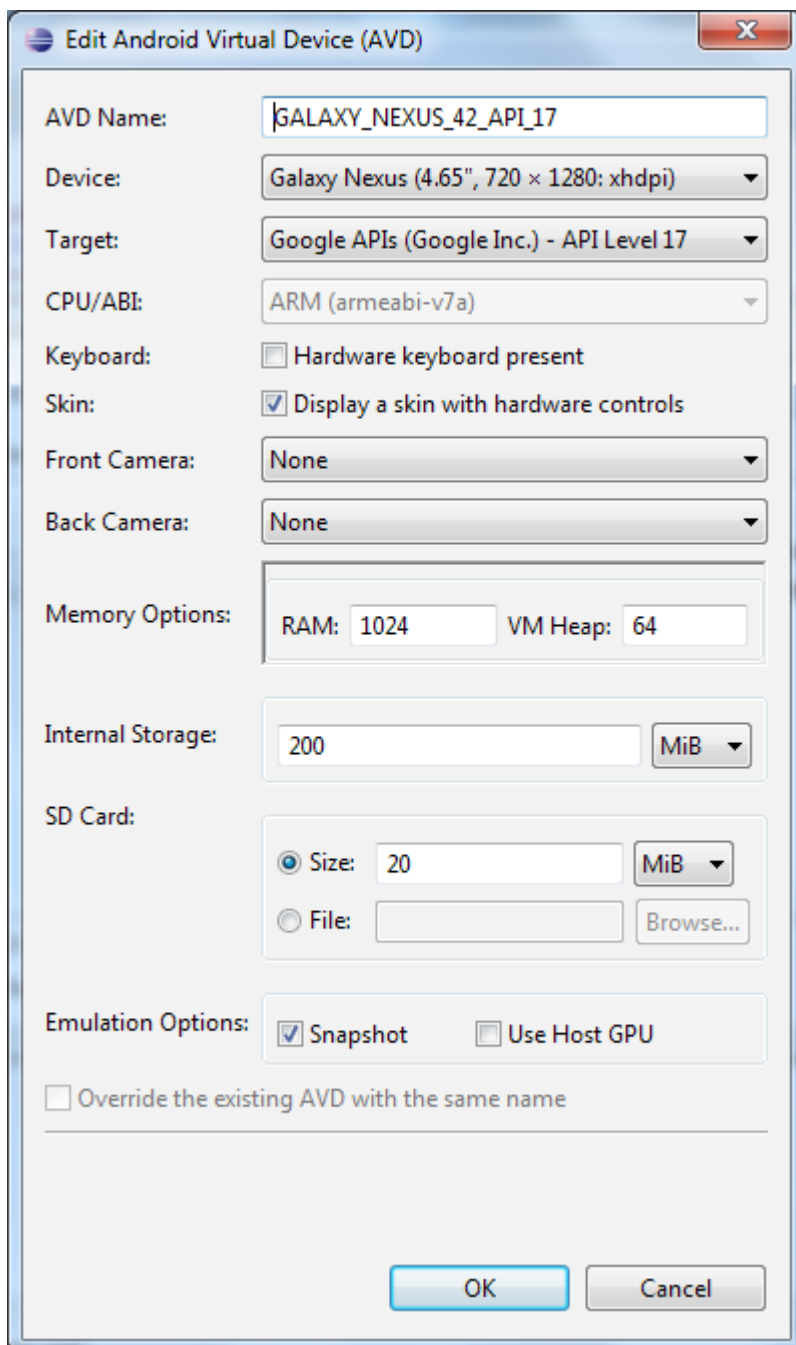


Paso 7. Configurar un AVD.

A la hora de probar y depurar aplicaciones Android no tendremos que hacerlo necesariamente sobre un dispositivo físico, sino que podremos configurar un emulador o dispositivo virtual (*Android Virtual Device*, o AVD) donde poder realizar fácilmente estas tareas. Para ello, accederemos al AVD Manager (menú Window / AVD Manager), y en la sección *Virtual Devices* podremos añadir tantos AVD como se necesiten (por ejemplo, configurados para distintas versiones de Android o distintos tipos de dispositivo). Nuevamente, mi consejo será configurar al menos dos AVD, uno para la mínima versión de Android que queramos soportar, y otro para la versión más reciente disponible.



Para configurar el AVD tan sólo tendremos que indicar un nombre descriptivo, la versión de la plataforma Android que utilizará, y las características de hardware del dispositivo virtual, como por ejemplo su resolución de pantalla o el tamaño de la tarjeta SD. Además, marcaremos la opción "*Snapshot*", que nos permitirá arrancar el emulador más rápidamente en futuras ejecuciones.

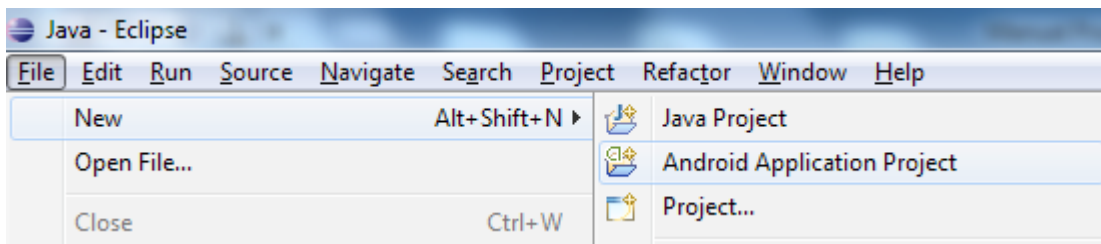


Y con este paso ya tendríamos preparadas todas las herramientas necesarias para comenzar a desarrollar aplicaciones Android. En próximos apartados veremos como crear un nuevo proyecto, la estructura y componentes de un proyecto Android, y crearemos una aplicación sencilla para poner en práctica todos los conceptos aprendidos.

Estructura de un proyecto Android

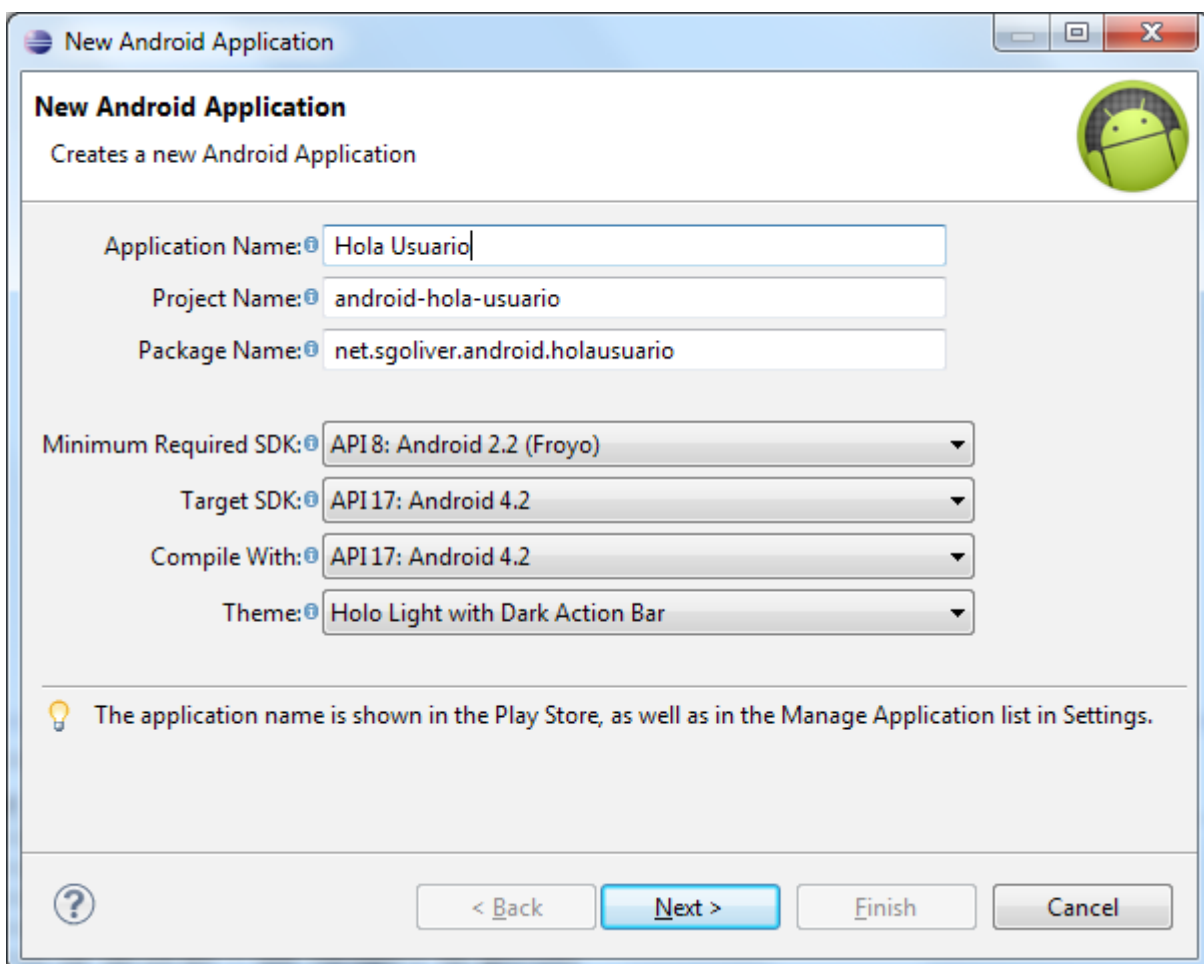
Para empezar a comprender cómo se construye una aplicación Android vamos a crear un nuevo proyecto Android en Eclipse y echaremos un vistazo a la estructura general del proyecto creado por defecto.

Para crear un nuevo proyecto abriremos Eclipse e iremos al menú File / New / Android Application Project.

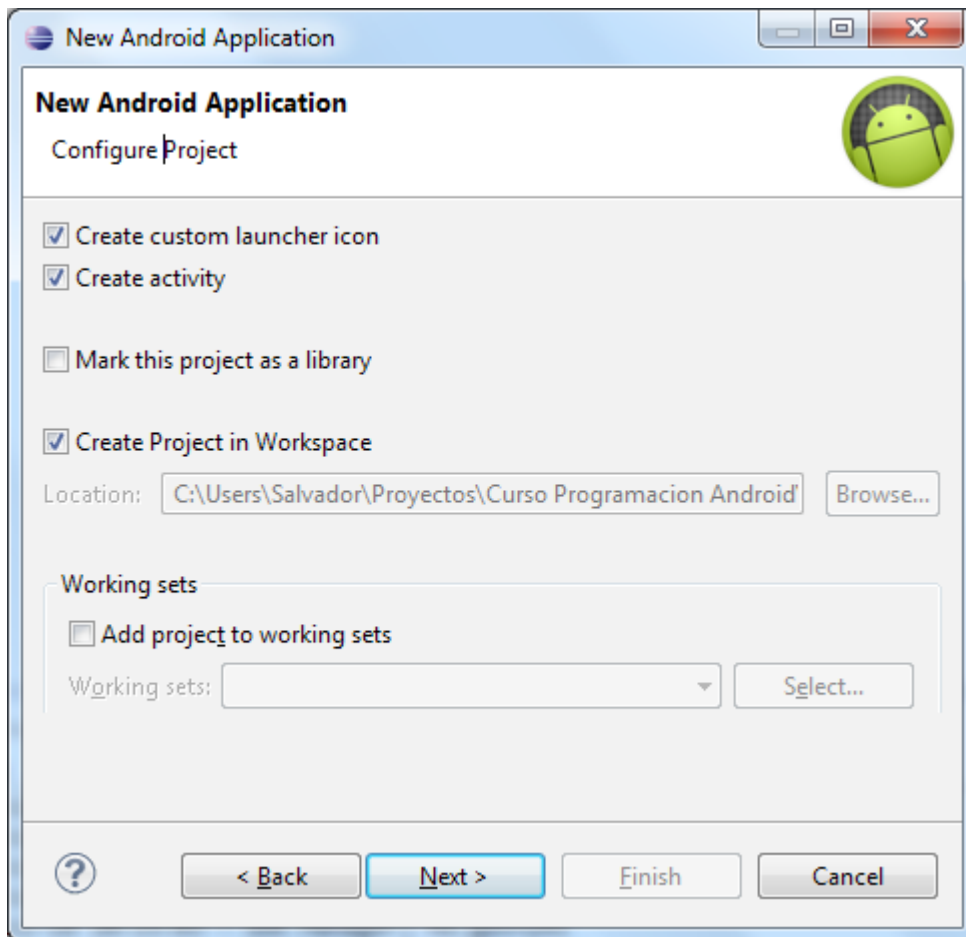


De esta forma iniciaremos el asistente de creación del proyecto, que nos guiará por las distintas opciones de creación y configuración de un nuevo proyecto.

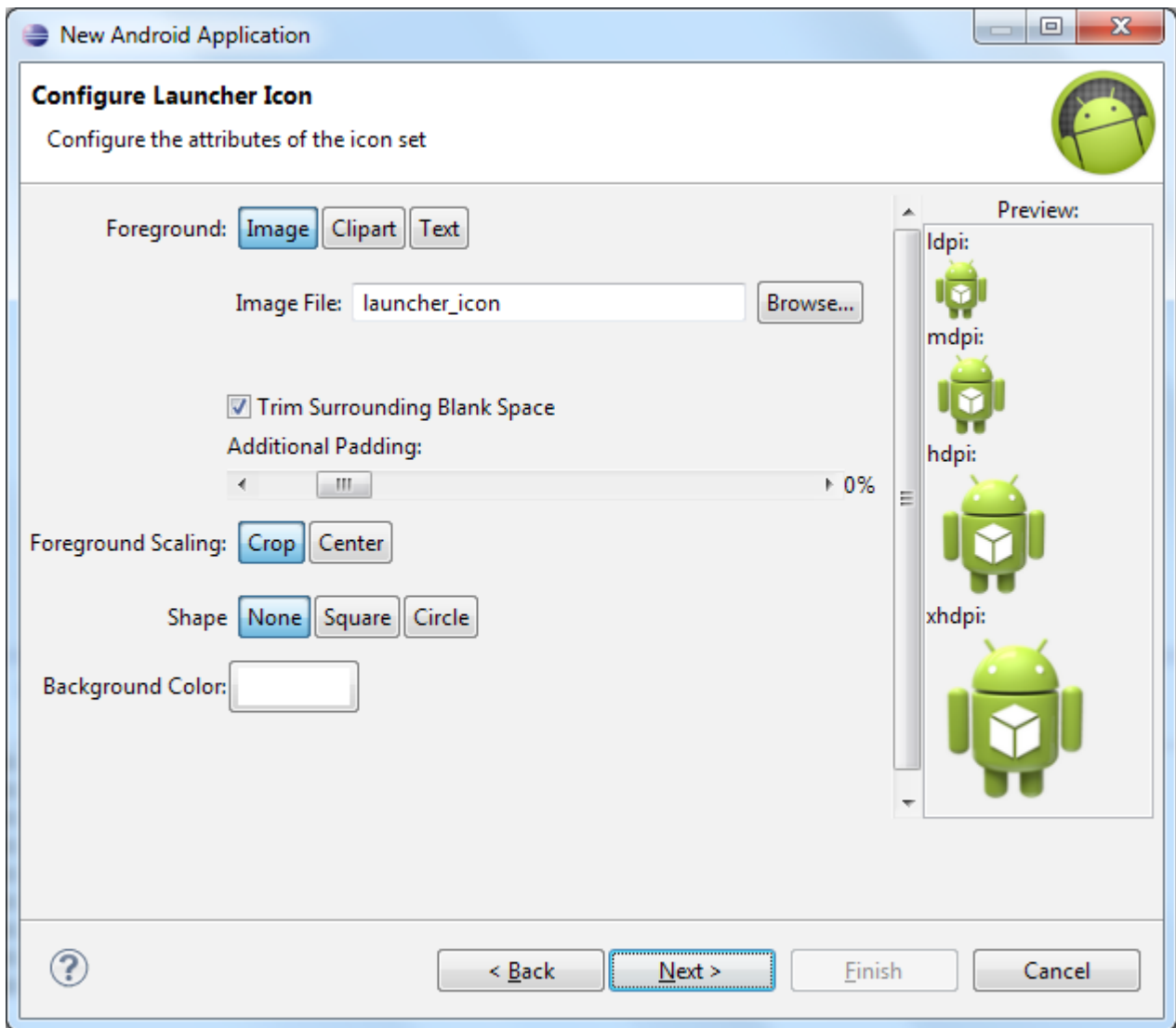
En la primera pantalla indicaremos el nombre de la aplicación, el nombre del proyecto y el paquete java que utilizaremos en nuestras clases java. Tendremos que seleccionar además la mínima versión del SDK que aceptará nuestra aplicación al ser instalada en un dispositivo (*Minimum Required SDK*), la versión del SDK para la que desarrollaremos (*Target SDK*), y la versión del SDK con la que compilaremos el proyecto (*Compile with*). Las dos últimas suelen coincidir con la versión de Android más reciente. El resto de opciones las dejaremos con los valores por defecto.



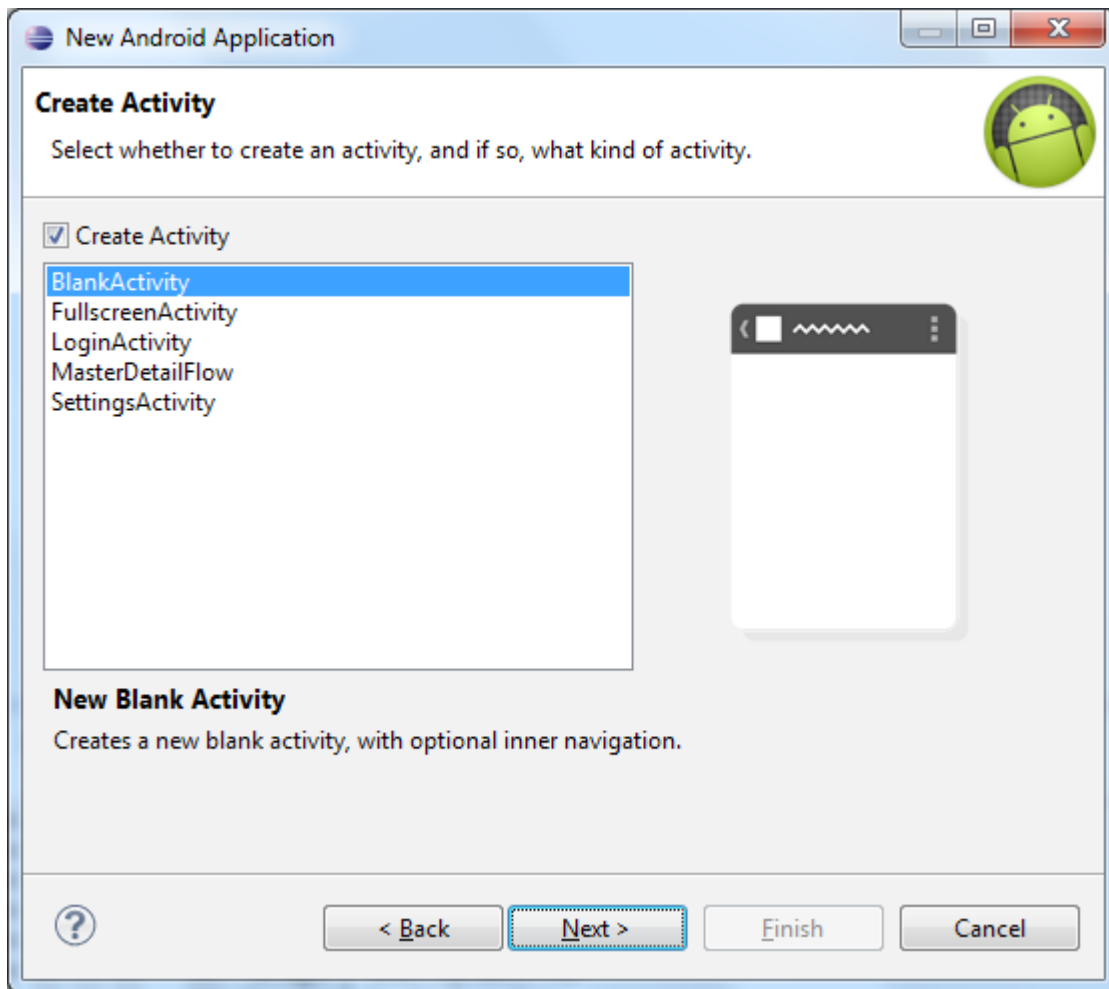
Al pulsar el botón *Next*, accederemos al segundo paso del asistente, donde tendremos que indicar si durante la creación del nuevo proyecto queremos crear un icono para nuestra aplicación (*Create custom launcher icon*) y si queremos crear una actividad inicial (*Create activity*). También podremos indicar si nuestro proyecto será del tipo Librería (*Mark this Project as a library*). Por ahora dejaremos todas las opciones marcadas por defecto como se ve en la siguiente imagen y pulsamos *Next*.



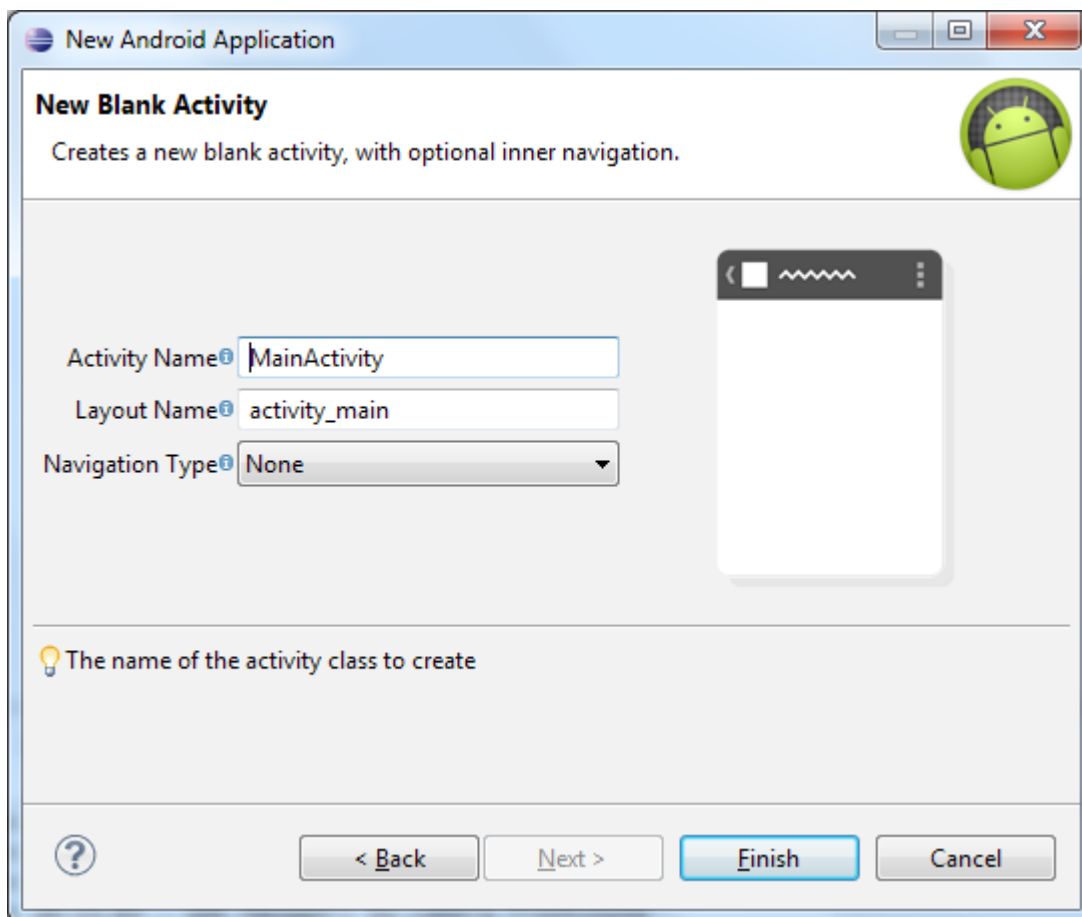
En la siguiente pantalla del asistente configuraremos el icono que tendrá nuestra aplicación en el dispositivo. No nos detendremos mucho en este paso ya que no tiene demasiada relevancia por el momento. Tan sólo decir que podremos seleccionar la imagen, texto o dibujo predefinido que aparecerá en el icono, el margen, la forma y los colores aplicados. Por ahora podemos dejarlo todo por defecto y avanzar al siguiente paso pulsando *Next*.



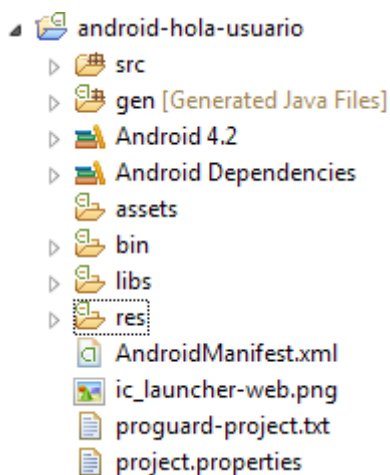
En la siguiente pantalla del asistente elegiremos el tipo de Actividad principal de la aplicación. Entenderemos por ahora que una *actividad* es una "ventana" o "pantalla" de la aplicación. En este paso también dejaremos todos los valores por defecto, indicando así que nuestra pantalla principal será del tipo *BlankActivity*.



Por último, en el último paso del asistente indicaremos los datos de esta actividad principal que acabamos de elegir, indicando el nombre de su clase java asociada y el nombre de su *layout xml* (algo así como la interfaz gráfica de la actividad, lo veremos más adelante).



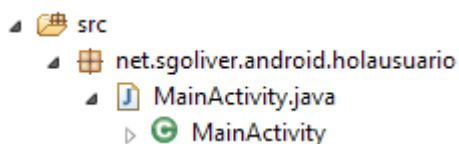
Una vez configurado todo pulsamos el botón *Finish* y Eclipse creará por nosotros toda la estructura del proyecto y los elementos indispensables que debe contener. En la siguiente imagen vemos los elementos creados inicialmente para un nuevo proyecto Android:



En los siguientes apartados describiremos los elementos principales de esta estructura.

Carpeta /src/

Esta carpeta contendrá todo el código fuente de la aplicación, código de la interfaz gráfica, clases auxiliares, etc. Inicialmente, Eclipse creará por nosotros el código básico de la pantalla (*Activity*) principal de la aplicación, que recordemos que en nuestro caso era *MainActivity*, y siempre bajo la estructura del paquete java definido.



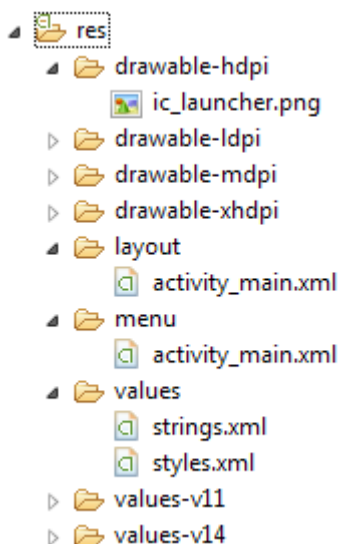
Carpeta /res/

Contiene todos los ficheros de recursos necesarios para el proyecto: imágenes, vídeos, cadenas de texto, etc. Los diferentes tipos de recursos se distribuyen entre las siguientes subcarpetas:

Carpeta	Descripción
/res/drawable/	Contiene las imágenes [y otros elementos gráficos] usados en por la aplicación. Para definir diferentes recursos dependiendo de la resolución y densidad de la pantalla del dispositivo se suele dividir en varias subcarpetas: <ul style="list-style-type: none">➤ /drawable-ldpi (densidad baja)➤ /drawable-mdpi (densidad media)➤ /drawable-hdpi (densidad alta)➤ /drawable-xhdpi (densidad muy alta)
/res/layout/	Contiene los ficheros de definición XML de las diferentes pantallas de la interfaz gráfica. Para definir distintos <i>layouts</i> dependiendo de la orientación del dispositivo se puede dividir en dos subcarpetas: <ul style="list-style-type: none">➤ /layout (vertical)➤ /layout-land (horizontal)
/res/anim/ /res/animator/	Contienen la definición de las animaciones utilizadas por la aplicación.
/res/color/	Contiene ficheros XML de definición de colores según estado.
/res/menu/	Contiene la definición XML de los menús de la aplicación.
/res/values/	Contiene otros ficheros XML de recursos de la aplicación, como por ejemplo cadenas de texto (<i>strings.xml</i>), estilos (<i>styles.xml</i>), colores (<i>colors.xml</i>), arrays de valores (<i>arrays.xml</i>), etc.
/res/xml/	Contiene otros ficheros XML de datos utilizados por la aplicación.
/res/raw/	Contiene recursos adicionales, normalmente en formato distinto a XML, que no se incluyan en el resto de carpetas de recursos.

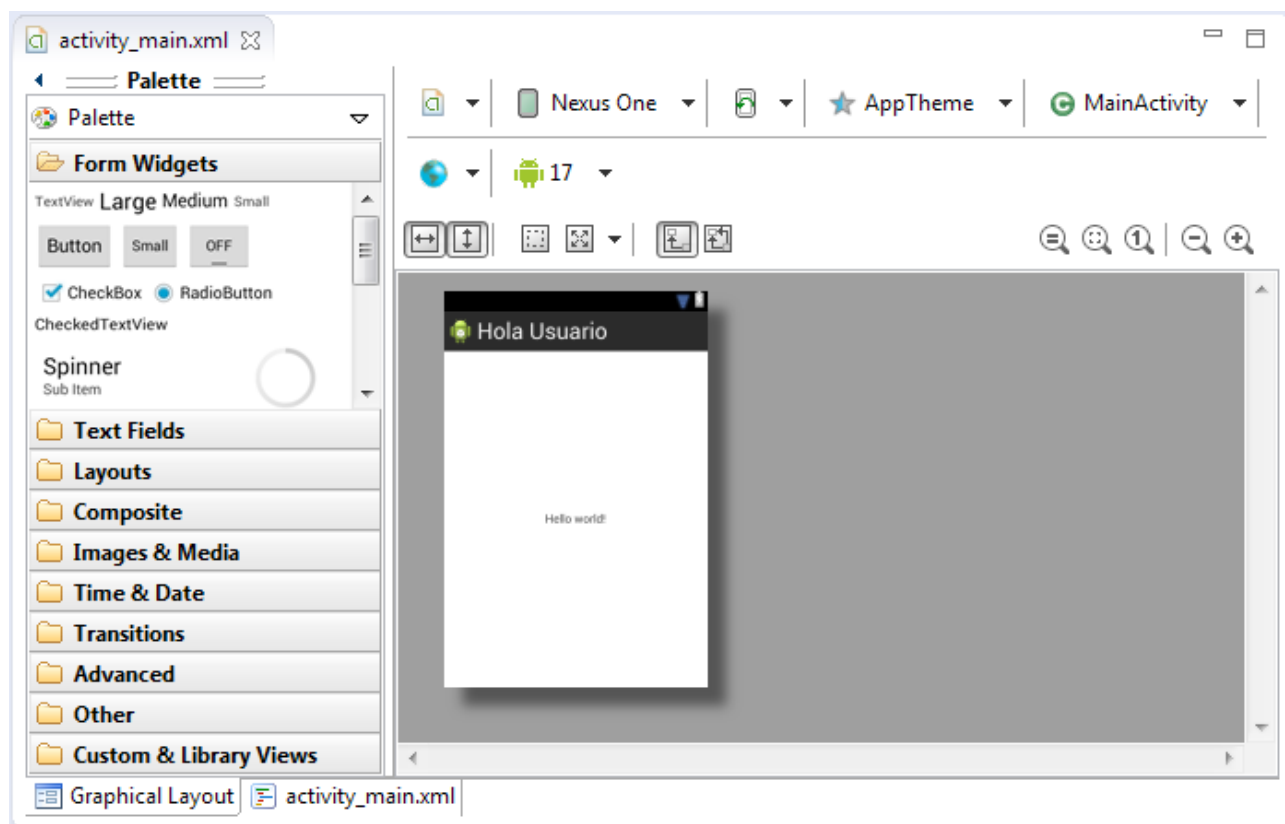
No todas estas carpetas tienen por qué aparecer en cada proyecto Android, tan sólo las que se necesiten. Iremos viendo durante el curso que tipo de elementos se pueden incluir en cada una de estas carpetas.

Como ejemplo, para un proyecto nuevo Android, se crean por defecto los siguientes recursos para la aplicación:



Como se puede observar, existen algunas carpetas en cuyo nombre se incluye un sufijo adicional, como por ejemplo "values-v11" y "values-v14". Estos, y otros sufijos, se emplean para definir recursos independientes para determinados dispositivos según sus características. De esta forma, por ejemplo, los recursos incluidos en la carpeta "values-v11" se aplicarían tan sólo a dispositivos cuya versión de Android sea la 3.0 (API 11) o superior. Al igual que el sufijo "-v" existen otros muchos para referirse a otras características del terminal, puede consultarse la lista completa en la [documentación oficial del Android](#).

Entre los recursos creados por defecto, cabe destacar el layout "activity_main.xml", que contiene la definición de la interfaz gráfica de la pantalla principal de la aplicación. Si hacemos doble clic sobre el fichero Eclipse nos mostrará esta interfaz en su editor gráfico (tipo arrastrar y soltar) y como podremos comprobar, en principio contiene tan sólo una etiqueta de texto centrada en pantalla con el mensaje "Hello World!".



Durante el curso no utilizaremos demasiado este editor gráfico, sino que modificaremos la interfaz de nuestras pantallas manipulando directamente el fichero XML asociado (al que se puede acceder pulsando

sobre la pestaña inferior derecha, junto la solapa "Graphical Layout" que se observa en la imagen. En este caso, el XML asociado sería el siguiente:



```
activity_main.xml
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity" >

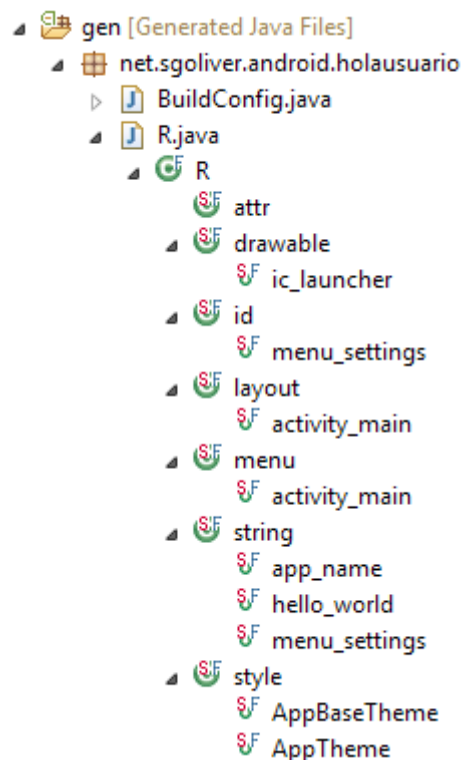
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerHorizontal="true"
        android:layout_centerVertical="true"
        android:text="@string/hello_world" />

</RelativeLayout>
```

Esto en principio puede parecer mucho más complicado que utilizar el editor gráfico, pero por el contrario [además de no ser nada complicado en realidad] nos permitirá aprender los entresijos de Android más rápidamente.

Carpeta /gen/

Contiene una serie de elementos de código **generados automáticamente** al compilar el proyecto. Cada vez que generamos nuestro proyecto, la maquinaria de compilación de Android genera por nosotros una serie de ficheros fuente java dirigidos al control de los recursos de la aplicación. Importante: dado que estos ficheros se generan automáticamente tras cada compilación del proyecto es importante que no se modifiquen manualmente bajo ninguna circunstancia.



```
gen [Generated Java Files]
├── net.sgoliver.android.holausuario
│   ├── BuildConfig.java
│   └── R.java
│       ├── R
│       │   ├── attr
│       │   ├── drawable
│       │   │   ├── ic_launcher
│       │   ├── id
│       │   │   ├── menu_settings
│       │   ├── layout
│       │   │   ├── activity_main
│       │   ├── menu
│       │   │   ├── activity_main
│       │   ├── string
│       │   │   ├── app_name
│       │   │   ├── hello_world
│       │   │   ├── menu_settings
│       │   ├── style
│       │   │   ├── AppBaseTheme
│       │   │   ├── AppTheme
```

A destacar sobre todo el fichero que aparece desplegado en la imagen anterior, llamado `R.java`, donde se define la clase `R`.

Esta clase `R` contendrá en todo momento una serie de constantes con los ID de todos los recursos de la aplicación incluidos en la carpeta `/res/`, de forma que podamos acceder fácilmente a estos recursos desde nuestro código a través de este dato. Así, por ejemplo, la constante `R.drawable.ic_launcher` contendrá el ID de la imagen `"ic_launcher.png"` contenida en la carpeta `/res/drawable/`. Veamos como ejemplo la clase `R` creada por defecto para un proyecto nuevo:

```
package net.sgoliver.android.holausuario;

public final class R {
    public static final class attr {
    }
    public static final class drawable {
        public static final int ic_launcher=0x7f020001;
    }
    public static final class id {
        public static final int menu_settings=0x7f070000;
    }
    public static final class layout {
        public static final int activity_main=0x7f030000;
    }
    public static final class menu {
        public static final int activity_main=0x7f060000;
    }
    public static final class string {
        public static final int app_name=0x7f040000;
        public static final int hello_world=0x7f040001;
        public static final int menu_settings=0x7f040002;
    }
    public static final class style {
        /**
         * Base application theme, dependent on API level. This theme is replaced
         * by AppBaseTheme from res/values-vXX/styles.xml on newer devices.
         *
         * Theme customizations available in newer API levels can go in
         * res/values-vXX/styles.xml, while customizations related to
         * backward-compatibility can go here.
         *
         * Base application theme for API 11+. This theme completely replaces
         * AppBaseTheme from res/values/styles.xml on API 11+ devices.
         *
         * API 11 theme customizations can go here.
         *
         * Base application theme for API 14+. This theme completely replaces
         * AppBaseTheme from BOTH res/values/styles.xml and
         * res/values-v11/styles.xml on API 14+ devices.
         *
         * API 14 theme customizations can go here.
         */
        public static final int AppBaseTheme=0x7f050000;
        /** Application theme.
         * All customizations that are NOT specific to a particular API-level can go
         * here.
         */
        public static final int AppTheme=0x7f050001;
    }
}
```

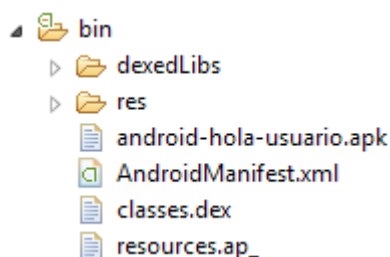

Carpeta /assets/

Contiene todos los demás ficheros auxiliares necesarios para la aplicación (y que se incluirán en su propio paquete), como por ejemplo ficheros de configuración, de datos, etc.

La diferencia entre los recursos incluidos en la carpeta `/res/raw/` y los incluidos en la carpeta `/assets/` es que para los primeros se generará un ID en la clase `R` y se deberá acceder a ellos con los diferentes métodos de acceso a recursos. Para los segundos sin embargo no se generarán ID y se podrá acceder a ellos por su ruta como a cualquier otro fichero del sistema. Usaremos uno u otro según las necesidades de nuestra aplicación.

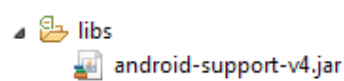
Carpeta /bin/

Ésta es otra de esas carpetas que en principio no tendremos por qué tocar. Contiene los elementos compilados de la aplicación y otros ficheros auxiliares. Cabe destacar el fichero con extensión `".apk"`, que es el ejecutable de la aplicación que se instalará en el dispositivo.



Carpeta /libs/

Contendrá las librerías auxiliares, normalmente en formato `".jar"` que utilizemos en nuestra aplicación Android.



Fichero AndroidManifest.xml

Contiene la definición en XML de los aspectos principales de la aplicación, como por ejemplo su identificación (nombre, versión, icono, ...), sus componentes (pantallas, mensajes, ...), las librerías auxiliares utilizadas, o los permisos necesarios para su ejecución. Veremos más adelante más detalles de este fichero.

Y con esto todos los elementos principales de un proyecto Android. No pierdas de vista este proyecto de ejemplo que hemos creado ya que lo utilizaremos en breve como base para crear nuestra primera aplicación. Pero antes, en el siguiente apartado hablaremos de los componentes software principales con los que podemos construir una aplicación Android.

Componentes de una aplicación Android

En el apartado anterior vimos la estructura de un proyecto Android y aprendimos dónde colocar cada uno de los elementos que componen una aplicación, tanto elementos de software como recursos gráficos o de datos. En éste nuevo post vamos a centrarnos específicamente en los primeros, es decir, veremos los distintos tipos de componentes de software con los que podremos construir una aplicación Android.

En Java o .NET estamos acostumbrados a manejar conceptos como ventana, control, eventos o servicios como los elementos básicos en la construcción de una aplicación.

Pues bien, en Android vamos a disponer de esos mismos elementos básicos aunque con un pequeño cambio en la terminología y el enfoque. Repasemos los componentes principales que pueden formar parte de una aplicación Android [Por claridad, y para evitar confusiones al consultar documentación en inglés, intentaré traducir lo menos posible los nombres originales de los componentes].

Activity

Las actividades (*activities*) representan el componente principal de la interfaz gráfica de una aplicación Android. Se puede pensar en una actividad como el elemento análogo a una ventana o pantalla en cualquier otro lenguaje visual.

View

Las vistas (*view*) son los componentes básicos con los que se construye la interfaz gráfica de la aplicación, análogo por ejemplo a los *controles* de Java o .NET. De inicio, Android pone a nuestra disposición una gran cantidad de controles básicos, como cuadros de texto, botones, listas desplegables o imágenes, aunque también existe la posibilidad de extender la funcionalidad de estos controles básicos o crear nuestros propios controles personalizados.

Service

Los servicios son componentes sin interfaz gráfica que se ejecutan en segundo plano. En concepto, son similares a los servicios presentes en cualquier otro sistema operativo. Los servicios pueden realizar cualquier tipo de acciones, por ejemplo actualizar datos, lanzar notificaciones, o incluso mostrar elementos visuales (p.ej. actividades) si se necesita en algún momento la interacción con del usuario.

Content Provider

Un *content provider* es el mecanismo que se ha definido en Android para compartir datos entre aplicaciones. Mediante estos componentes es posible compartir determinados datos de nuestra aplicación sin mostrar detalles sobre su almacenamiento interno, su estructura, o su implementación. De la misma forma, nuestra aplicación podrá acceder a los datos de otra a través de los *content provider* que se hayan definido.

Broadcast Receiver

Un *broadcast receiver* es un componente destinado a detectar y reaccionar ante determinados mensajes o eventos globales generados por el sistema (por ejemplo: "Batería baja", "SMS recibido", "Tarjeta SD insertada", ...) o por otras aplicaciones (cualquier aplicación puede generar mensajes (*intents*, en terminología Android) broadcast, es decir, no dirigidos a una aplicación concreta sino a cualquiera que quiera escucharlo).

Widget

Los *widgets* son elementos visuales, normalmente interactivos, que pueden mostrarse en la pantalla principal (*home screen*) del dispositivo Android y recibir actualizaciones periódicas. Permiten mostrar información de la aplicación al usuario directamente sobre la pantalla principal.

Intent

Un *intent* es el elemento básico de comunicación entre los distintos componentes Android que hemos descrito anteriormente. Se pueden entender como los mensajes o peticiones que son enviados entre los distintos componentes de una aplicación o entre distintas aplicaciones. Mediante un *intent* se puede mostrar una actividad desde cualquier otra, iniciar un servicio, enviar un mensaje *broadcast*, iniciar otra aplicación, etc.

En el siguiente apartado empezaremos ya a añadir y modificar algo de código, analizando al detalle una aplicación sencilla.

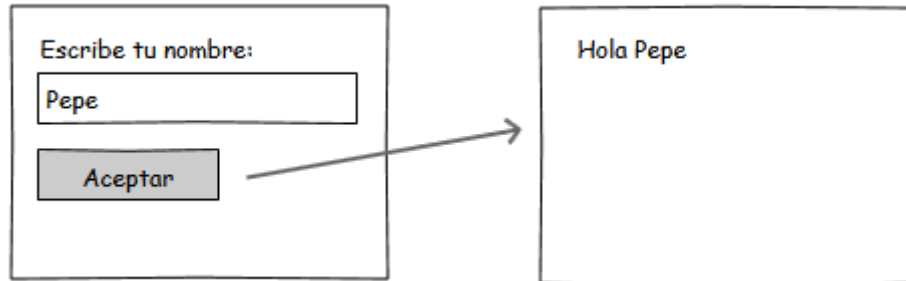
Desarrollando una aplicación Android sencilla

Después de instalar nuestro entorno de desarrollo para Android y comentar la estructura básica de un proyecto y los diferentes componentes software que podemos utilizar ya es hora de empezar a escribir algo de código. Y como siempre lo mejor es empezar por escribir una aplicación sencilla.

En un principio me planteé analizar en este capítulo el clásico *Hola Mundo* pero más tarde me pareció que

se iban a quedar algunas cosas básicas en el tintero. Así que he versionado a mi manera el *Hola Mundo* transformándolo en algo así como un *Hola Usuario*, que es igual de sencilla pero añade un par de cosas interesantes de contar. La aplicación constará de dos pantallas, por un lado la pantalla principal que solicitará un nombre al usuario y una segunda pantalla en la que se mostrará un mensaje personalizado para el usuario. Así de sencillo e inútil, pero aprenderemos muchos conceptos básicos, que para empezar no está mal.

Por dibujarlo para entender mejor lo que queremos conseguir, sería algo tan sencillo como lo siguiente:



Vamos a partir del proyecto de ejemplo que creamos en un apartado anterior, al que casualmente llamamos *HolaUsuario*. Como ya vimos Eclipse había creado por nosotros la estructura de carpetas del proyecto y todos los ficheros necesarios de un *Hola Mundo* básico, es decir, una sola pantalla donde se muestra únicamente un mensaje fijo.

Lo primero que vamos a hacer es diseñar nuestra pantalla principal modificando la que Eclipse nos ha creado por defecto. Aunque ya lo hemos comentado de pasada, recordemos dónde y cómo se define cada pantalla de la aplicación. En Android, el diseño y la lógica de una pantalla están separados en dos ficheros distintos. Por un lado, en el fichero `/res/layout/activity_main.xml` tendremos el diseño puramente visual de la pantalla definido como fichero XML y por otro lado, en el fichero `/src/paquete.java/MainActivity.java`, encontraremos el código java que determina la lógica de la pantalla.

Vamos a modificar en primer lugar el aspecto de la ventana principal de la aplicación añadiendo los controles (*views*) que vemos en el esquema mostrado al principio del apartado. Para ello, vamos a sustituir el contenido del fichero `activity_main.xml` por el siguiente:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/LinearLayout1"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <TextView
        android:id="@+id/LblNombre"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/nombre" />

    <EditText
        android:id="@+id/TxtNombre"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:inputType="text" />
```

```

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

</LinearLayout>

```

En este XML se definen los elementos visuales que componen la interfaz de nuestra pantalla principal y se especifican todas sus propiedades. No nos detendremos mucho por ahora en cada detalle, pero expliquemos un poco lo que vemos en el fichero.

Lo primero que nos encontramos es un elemento `LinearLayout`. Los *layout* son elementos no visibles que determinan cómo se van a distribuir en el espacio los controles que incluyamos en su interior. Los programadores java, y más concretamente de *Swing*, conocerán este concepto perfectamente. En este caso, un `LinearLayout` distribuirá los controles simplemente uno tras otro y en la orientación que indique su propiedad `android:orientation`, que en este caso será "vertical".

Dentro del *layout* hemos incluido 3 controles: una etiqueta (`TextView`), un cuadro de texto (`EditText`), y un botón (`Button`). En todos ellos hemos establecido las siguientes propiedades:

- `android:id`. ID del control, con el que podremos identificarlo más tarde en nuestro código. Vemos que el identificador lo escribimos precedido de "@+id/". Esto tendrá como efecto que al compilarse el proyecto se genere automáticamente una nueva constante en la clase `R` para dicho control. Así, por ejemplo, como al cuadro de texto le hemos asignado el ID `TxtNombre`, podremos más tarde acceder al él desde nuestro código haciendo referencia a la constante `R.id.TxtNombre`.
- `android:layout_height` y `android:layout_width`. Dimensiones del control con respecto al layout que lo contiene. Esta propiedad tomará normalmente los valores "wrap_content" para indicar que las dimensiones del control se ajustarán al contenido del mismo, o bien "match_parent" para indicar que el ancho o el alto del control se ajustará al ancho o alto del layout contenedor respectivamente.

Además de estas propiedades comunes a casi todos los controles que utilizaremos, en el cuadro de texto hemos establecido también la propiedad `android:inputType`, que indica qué tipo de contenido va a albergar el control, en este caso texto normal (valor "text"), aunque podría haber sido una contraseña (`textPassword`), un teléfono (`phone`), una fecha (`date`),

Por último, en la etiqueta y el botón hemos establecido la propiedad `android:text`, que indica el texto que aparece en el control. Y aquí nos vamos a detener un poco, ya que tenemos dos alternativas a la hora de hacer esto. En Android, el texto de un control se puede especificar directamente como valor de la propiedad `android:text`, o bien utilizar alguna de las cadenas de texto definidas en los recursos del proyecto (como ya vimos, en el fichero `strings.xml`), en cuyo caso indicaremos como valor de la propiedad `android:text` su identificador precedido del prefijo "@string/". Dicho de otra forma, la primera alternativa habría sido indicar directamente el texto como valor de la propiedad, por ejemplo en la etiqueta de esta forma:

```

<TextView
    android:id="@+id/LblNombre"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Escribe tu nombre:" />

```

Y la segunda alternativa, la utilizada en el ejemplo, consistiría en definir primero una nueva cadena de texto en el fichero de recursos `/res/values/strings.xml`, por ejemplo con identificador "nombre" y valor "Escribe tu nombre:"

```

<resources>

    . . .

    <string name="nombre">Escribe tu nombre:</string>

    . . .

</resources>

```

Y posteriormente indicar el identificador de la cadena como valor de la propiedad `android:text`, siempre precedido del prefijo `@string/`, de la siguiente forma:

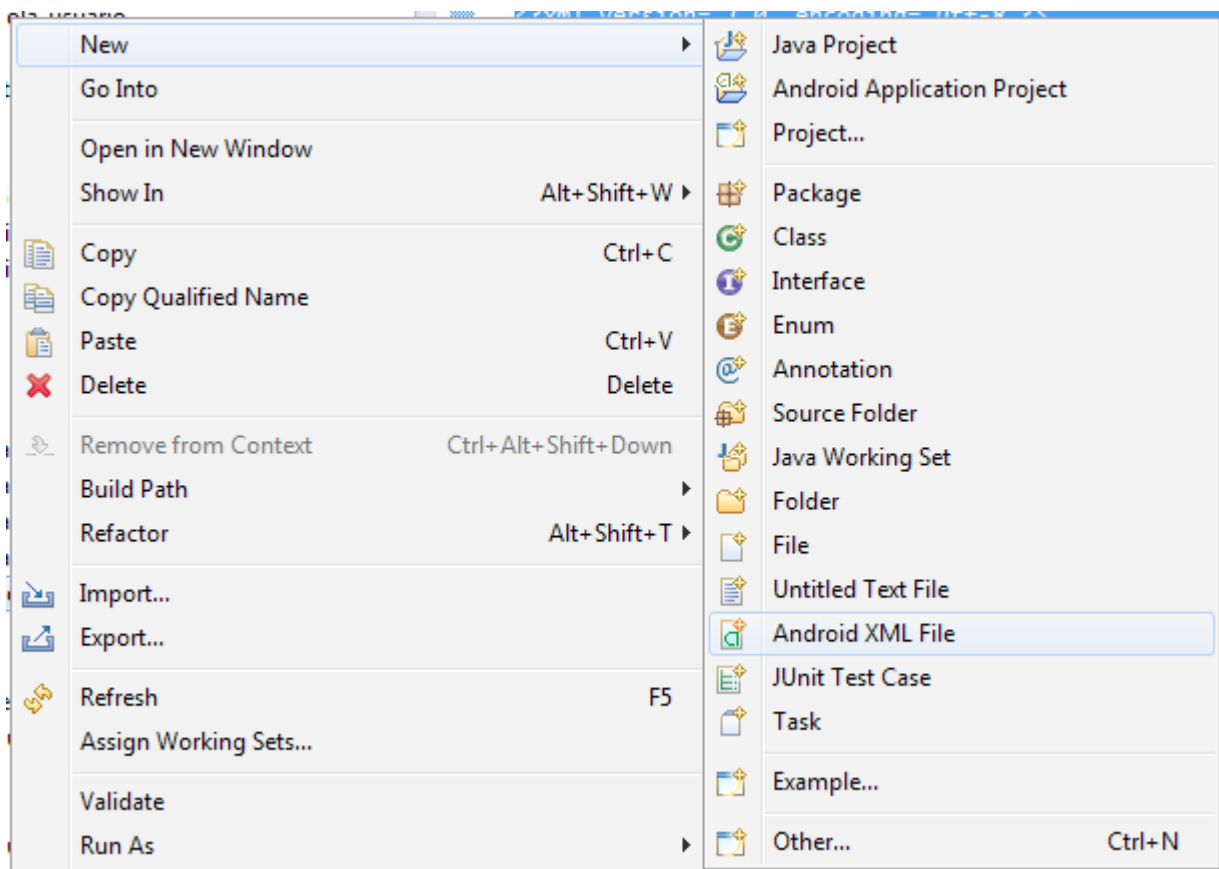
```

<TextView
    android:id="@+id/LblNombre"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/nombre" />

```

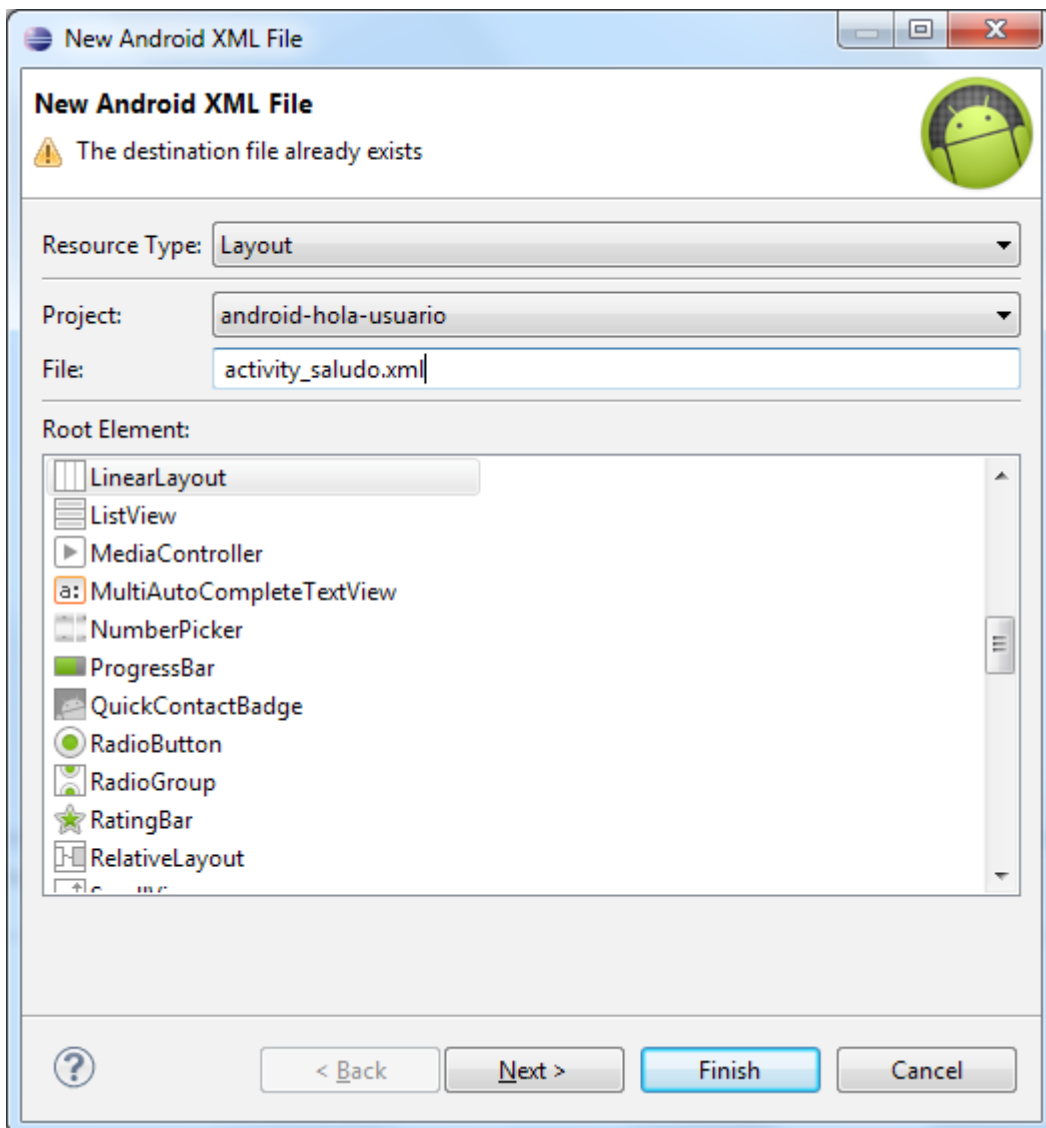
Esta segunda alternativa nos permite tener perfectamente localizadas y agrupadas todas las cadenas de texto utilizadas en la aplicación, lo que nos podría facilitar por ejemplo la traducción de la aplicación a otro idioma. Con esto ya tenemos definida la presentación visual de nuestra ventana principal de la aplicación. De igual forma definiremos la interfaz de la segunda pantalla, creando un nuevo fichero llamado `activity_saludo.xml`, y añadiendo esta vez tan solo una etiqueta (`TextView`) para mostrar el mensaje personalizado al usuario.

Para añadir el fichero, pulsaremos el botón derecho del ratón sobre la carpeta de recursos `/res/layout` y pulsaremos la opción `"New Android XML file"`.



En el cuadro de diálogo que nos aparece indicaremos como tipo de recurso `"Layout"`, indicaremos el nombre

del fichero (con extensión ".xml") y como elemento raíz seleccionaremos `LinearLayout`. Finalmente pulsamos *Finish* para crear el fichero.



Eclipse creará entonces el nuevo fichero y lo abrirá en el editor gráfico, aunque como ya indicamos, nosotros accederemos a la solapa de código para modificar directamente el contenido XML del fichero.

Para esta segunda pantalla el código que incluiríamos sería el siguiente:

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

    <TextView
        android:id="@+id/TxtSaludo"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="" />

</LinearLayout>
```

Una vez definida la interfaz de las pantallas de la aplicación deberemos implementar la lógica de la misma.

Como ya hemos comentado, la lógica de la aplicación se definirá en ficheros java independientes. Para la pantalla principal ya tenemos creado un fichero por defecto llamado `MainActivity.java`. Empecemos por comentar su código por defecto:

```
package net.sgoliver.android.holausuario;

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

public class MainActivity extends Activity {

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

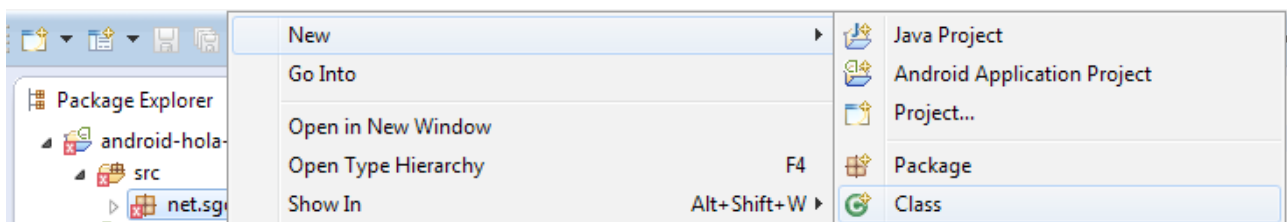
    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        getMenuInflater().inflate(R.menu.activity_main, menu);
        return true;
    }
}
```

Como ya vimos en un apartado anterior, las diferentes pantallas de una aplicación Android se definen mediante objetos de tipo `Activity`. Por tanto, lo primero que encontramos en nuestro fichero java es la definición de una nueva clase `MainActivity` que extiende a `Activity`. El único método que modificaremos de esta clase será el método `onCreate()`, llamado cuando se crea por primera vez la actividad. En este método lo único que encontramos en principio, además de la llamada a su implementación en la clase padre, es la llamada al método `setContentView(R.layout.activity_main)`. Con esta llamada estaremos indicando a Android que debe establecer como interfaz gráfica de esta actividad la definida en el recurso `R.layout.activity_main`, que no es más que la que hemos especificado en el fichero `/res/layout/activity_main.xml`. Una vez más vemos la utilidad de las diferentes constantes de recursos creadas automáticamente en la clase `R` al compilar el proyecto.

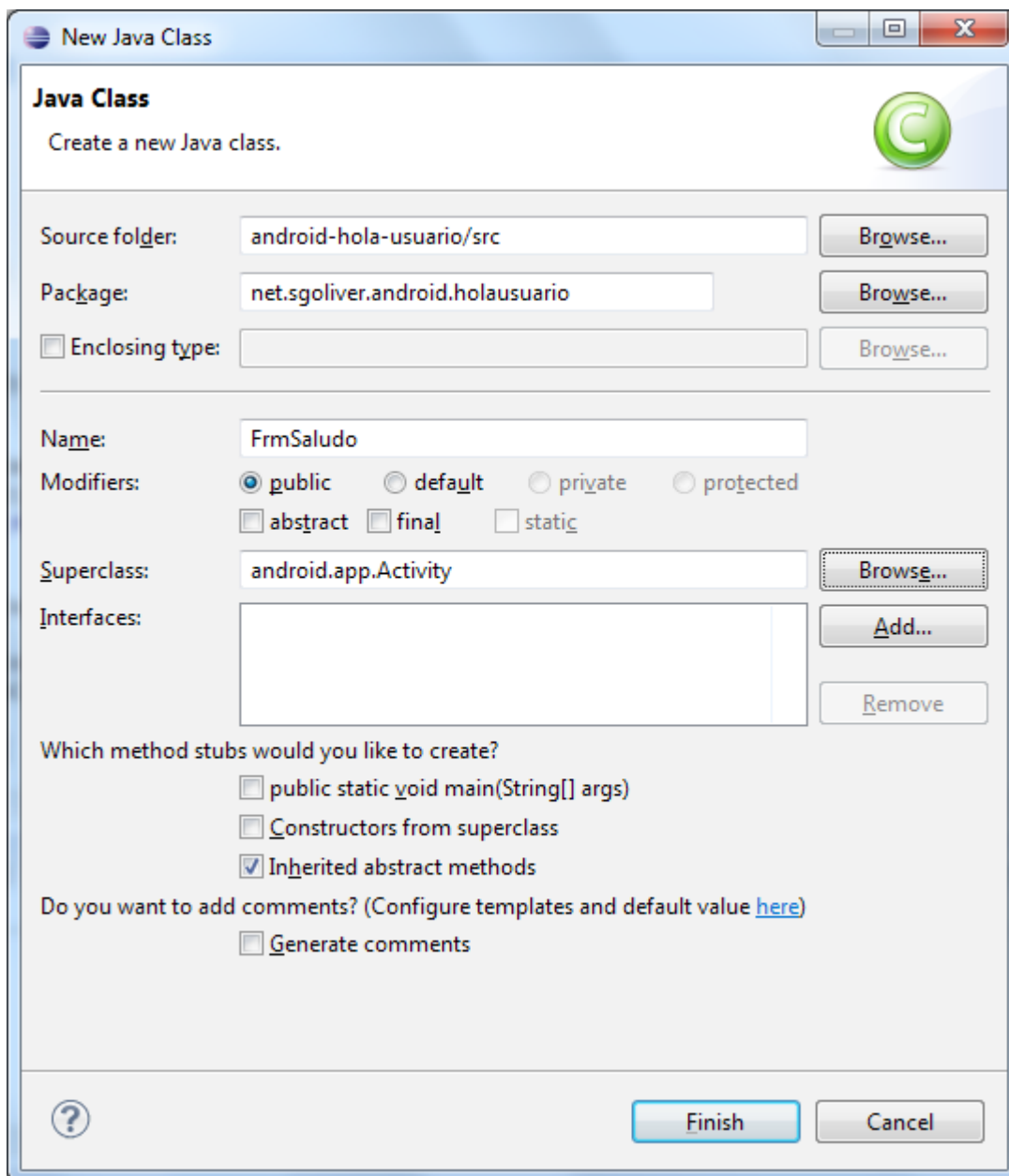
Además del método `onCreate()`, vemos que también se sobrescribe el método `onOptionsItemSelected()`, que se utiliza para definir menús en la aplicación. Por el momento no tocaremos este método, más adelante en el curso nos ocuparemos de este tema.

Ahora vamos a crear una nueva actividad para la segunda pantalla de la aplicación análoga a ésta primera, para lo que crearemos una nueva clase `FrmSaludo` que extienda también de `Activity` y que implemente el método `onCreate()` pero indicando esta vez que utilice la interfaz definida para la segunda pantalla en `R.layout.activity_saludo`.

Para ello, pulsaremos el botón derecho sobre la carpeta `/src/tu.paquete.java/` y seleccionaremos la opción de menú `New / Class`.



En el cuadro de diálogo que nos aparece indicaremos el nombre (*Name*) de la nueva clase y su clase padre (*Superclass*) como `android.app.Activity`.



Pulsaremos *Finish* y Eclipse creará el nuevo fichero y lo abrirá en el editor de código java.

Modificaremos por ahora el código de la clase para que quede algo análogo a la actividad principal:

```
package net.sgoliver.android.holausuario;

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

public class FrmSaludo extends Activity {

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


NOTA: Todos los pasos anteriores de creación de una nueva pantalla (layout xml + clase java) se puede realizar también mediante un asistente de Eclipse al que se accede mediante el menú contextual "New / Other... / Android / Android Activity". Sin embargo, he preferido explicarlo de esta forma para que quedaran claros todos los pasos y elementos necesarios.

Sigamos. Por ahora, el código incluido en estas clases lo único que hace es generar la interfaz de la actividad. A partir de aquí nosotros tendremos que incluir el resto de la lógica de la aplicación.

Y vamos a empezar con la actividad principal `MainActivity`, obteniendo una referencia a los diferentes controles de la interfaz que necesitemos manipular, en nuestro caso sólo el cuadro de texto y el botón. Para ello utilizaremos el método `findViewById()` indicando el ID de cada control, definidos como siempre en la clase `R`. Todo esto lo haremos dentro del método `onCreate()` de la clase `MainActivity`, justo a continuación de la llamada a `setContentView()` que ya comentamos:

```
. . .

import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.EditText;

. . .

public class MainActivity extends Activity {

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

        //Obtenemos una referencia a los controles de la interfaz
        final EditText txtNombre = (EditText)findViewById(R.id.TxtNombre);
        final Button btnHola = (Button)findViewById(R.id.BtnHola);

        . . .

    }
}

. . .
```

Como vemos, hemos añadido también varios `import` adicionales para tener acceso a todas las clases utilizadas.

Una vez tenemos acceso a los diferentes controles, ya sólo nos queda implementar las acciones a tomar cuando pulsemos el botón de la pantalla. Para ello, continuando el código anterior, y siempre dentro del método `onCreate()`, implementaremos el evento `onClick` de dicho botón, veamos cómo:

```

. . .

import android.content.Intent;

. . .

public class MainActivity extends Activity {

    @Override
    public void onCreate(Bundle savedInstanceState) {

        . . .

        //Obtenemos una referencia a los controles de la interfaz
        final EditText txtNombre = (EditText)findViewById(R.id.TxtNombre);
        final Button btnHola = (Button)findViewById(R.id.BtnHola);

        //Implementamos el evento "click" del botón
        btnHola.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                //Creamos el Intent
                Intent intent =
                    new Intent(MainActivity.this, FrmSaludo.class);

                //Creamos la información a pasar entre actividades
                Bundle b = new Bundle();
                b.putString("NOMBRE", txtNombre.getText().toString());

                //Añadimos la información al intent
                intent.putExtras(b);

                //Iniciamos la nueva actividad
                startActivity(intent);
            }
        });
    }

}

. . .

```

Como ya indicamos en el apartado anterior, la comunicación entre los distintos componentes y aplicaciones en Android se realiza mediante *intents*, por lo que el primer paso será crear un objeto de este tipo. Existen varias variantes del constructor de la clase `Intent`, cada una de ellas dirigida a unas determinadas acciones. En nuestro caso particular vamos a utilizar el intent para llamar a una actividad desde otra actividad de la misma aplicación, para lo que pasaremos a su constructor una referencia a la propia actividad *llamadora* (`MainActivity.this`), y la clase de la actividad *llamada* (`FrmMensaje.class`).

Si quisiéramos tan sólo mostrar una nueva actividad ya tan sólo nos quedaría llamar a `startActivity()` pasándole como parámetro el intent creado. Pero en nuestro ejemplo queremos también pasarle cierta información a la actividad llamada, concretamente el nombre que introduzca el usuario en el cuadro de texto de la pantalla principal. Para hacer esto vamos a crear un objeto `Bundle`, que puede contener una lista de pares *clave-valor* con toda la información a pasar entre las actividades. En nuestro caso sólo añadiremos un dato de tipo `String` mediante el método `putString(clave, valor)`. Tras esto añadiremos la información al intent mediante el método `putExtras(bundle)`.

Con esto hemos finalizado ya actividad principal de la aplicación, por lo que pasaremos ya a la secundaria.

Comenzaremos de forma análoga a la anterior, ampliando el método `onCreate` obteniendo las referencias a los objetos que manipularemos, esta vez sólo la etiqueta de texto. Tras esto viene lo más interesante, debemos recuperar la información pasada desde la actividad principal y asignarla como texto de la etiqueta. Para ello accederemos en primer lugar al intent que ha originado la actividad actual mediante el método `getIntent()` y recuperaremos su información asociada (objeto `Bundle`) mediante el método `getExtras()`.

Hecho esto tan sólo nos queda construir el texto de la etiqueta mediante su método `setText(texto)` y recuperando el valor de nuestra clave almacenada en el objeto `Bundle` mediante `getString(clave)`.

```
package net.sgoliver.android.holausuario;

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

public class FrmSaludo extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_saludo);

        //Localizar los controles
        TextView txtSaludo = (TextView)findViewById(R.id.TxtSaludo);

        //Recuperamos la información pasada en el intent
        Bundle bundle = this getIntent().getExtras();

        //Construimos el mensaje a mostrar
        txtSaludo.setText("Hola " + bundle.getString("NOMBRE"));
    }
}
```

Con esto hemos concluido la lógica de las dos pantallas de nuestra aplicación y tan sólo nos queda un paso importante para finalizar nuestro desarrollo. Como ya indicamos en un apartado anterior, toda aplicación Android utiliza un fichero especial en formato XML (`AndroidManifest.xml`) para definir, entre otras cosas, los diferentes elementos que la componen. Por tanto, todas las actividades de nuestra aplicación deben quedar convenientemente recogidas en este fichero. La actividad principal ya debe aparecer puesto que se creó de forma automática al crear el nuevo proyecto Android, por lo que debemos añadir tan sólo la segunda.

Para este ejemplo nos limitaremos a incluir la actividad en el XML mediante una nueva etiqueta `<Activity>`, indicar el nombre de la clase java asociada como valor del atributo `android:name`, y asignarle su título mediante el atributo `android:label`, más adelante veremos que opciones adicionales podemos especificar. Todo esto lo incluiremos justo debajo de la definición de la actividad principal dentro del fichero `AndroidManifest.xml`:

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

    . . .

    <activity
        android:name=".MainActivity"
        android:label="@string/title_activity_main" >
```

```

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

    <activity android:name=".FrmSaludo"
        android:label="@string/title_activity_saludo" >

    </activity>

</application>

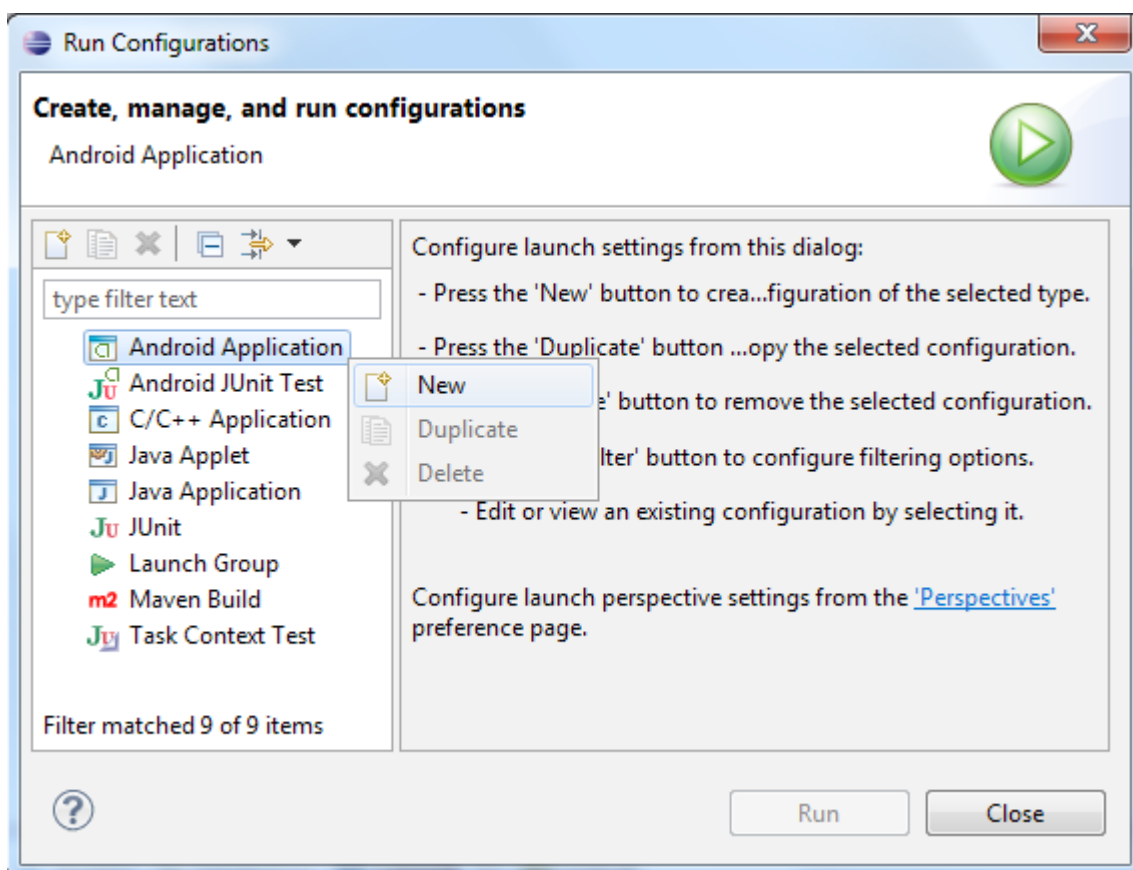
</manifest>

```

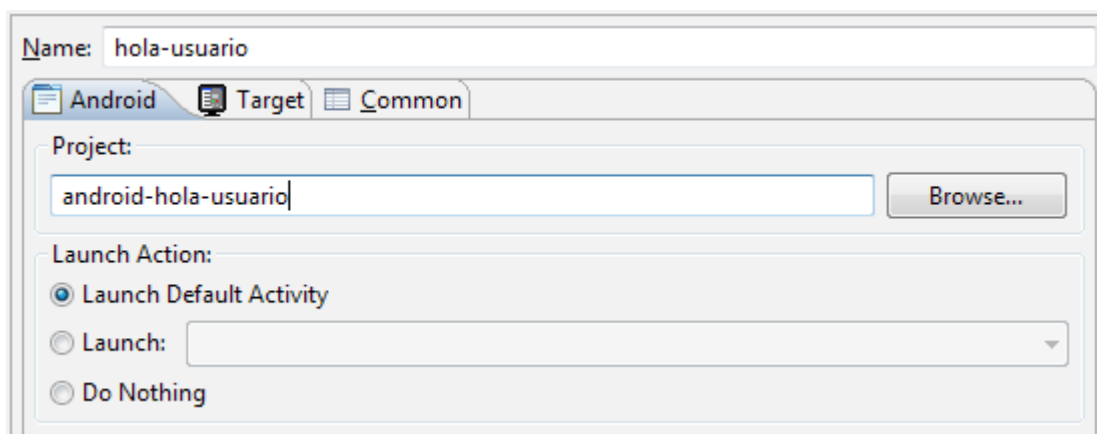
Como vemos, el título de la nueva actividad lo hemos indicado como referencia a una nueva cadena de caracteres, que tendremos que incluir como ya hemos comentado anteriormente en el fichero `/res/values/strings.xml`

Llegados aquí, y si todo ha ido bien, deberíamos poder ejecutar el proyecto sin errores y probar nuestra aplicación en el emulador. La forma de ejecutar y depurar la aplicación en Eclipse es análoga a cualquier otra aplicación java, pero por ser el primer capítulo vamos a recordarla.

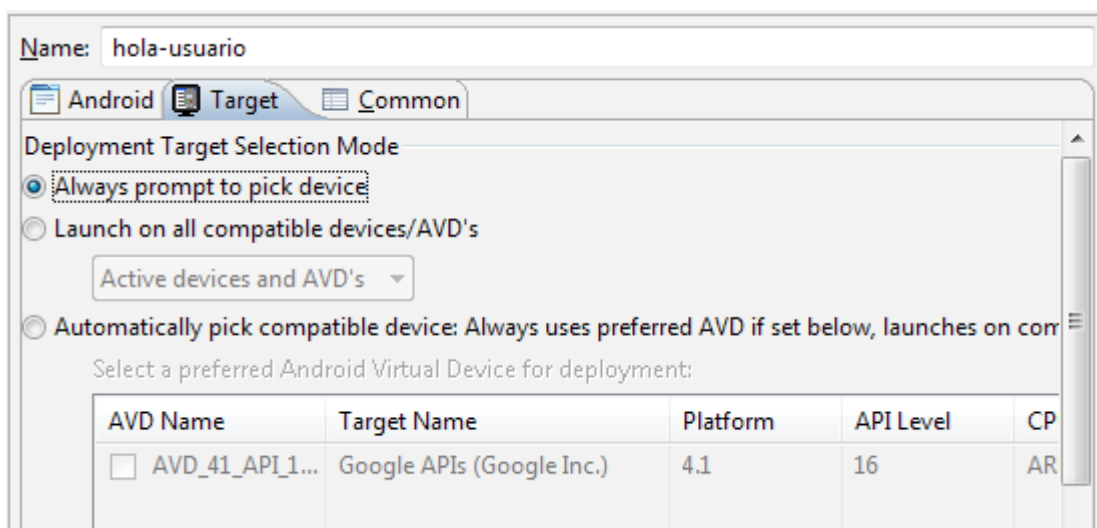
Lo primero que tendremos que hacer será configurar un nuevo "perfil de ejecución". Para ello accederemos al menú "Run/ Run Configurations..." y nos aparecerá la siguiente pantalla.



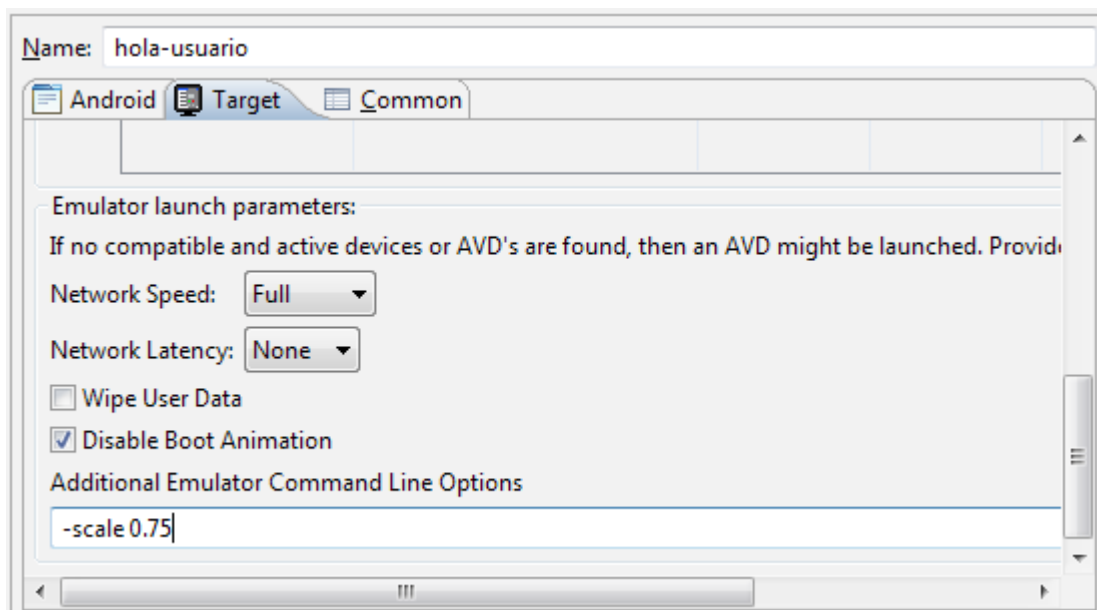
Sobre la categoría "Android Application" pulsaremos el botón derecho y elegiremos la opción "New" para crear un nuevo perfil para nuestra aplicación. En la siguiente pantalla le pondremos un nombre al perfil, en nuestro ejemplo "hola-usuario", y en la pestaña "Android" seleccionaremos el proyecto que queremos ejecutar.



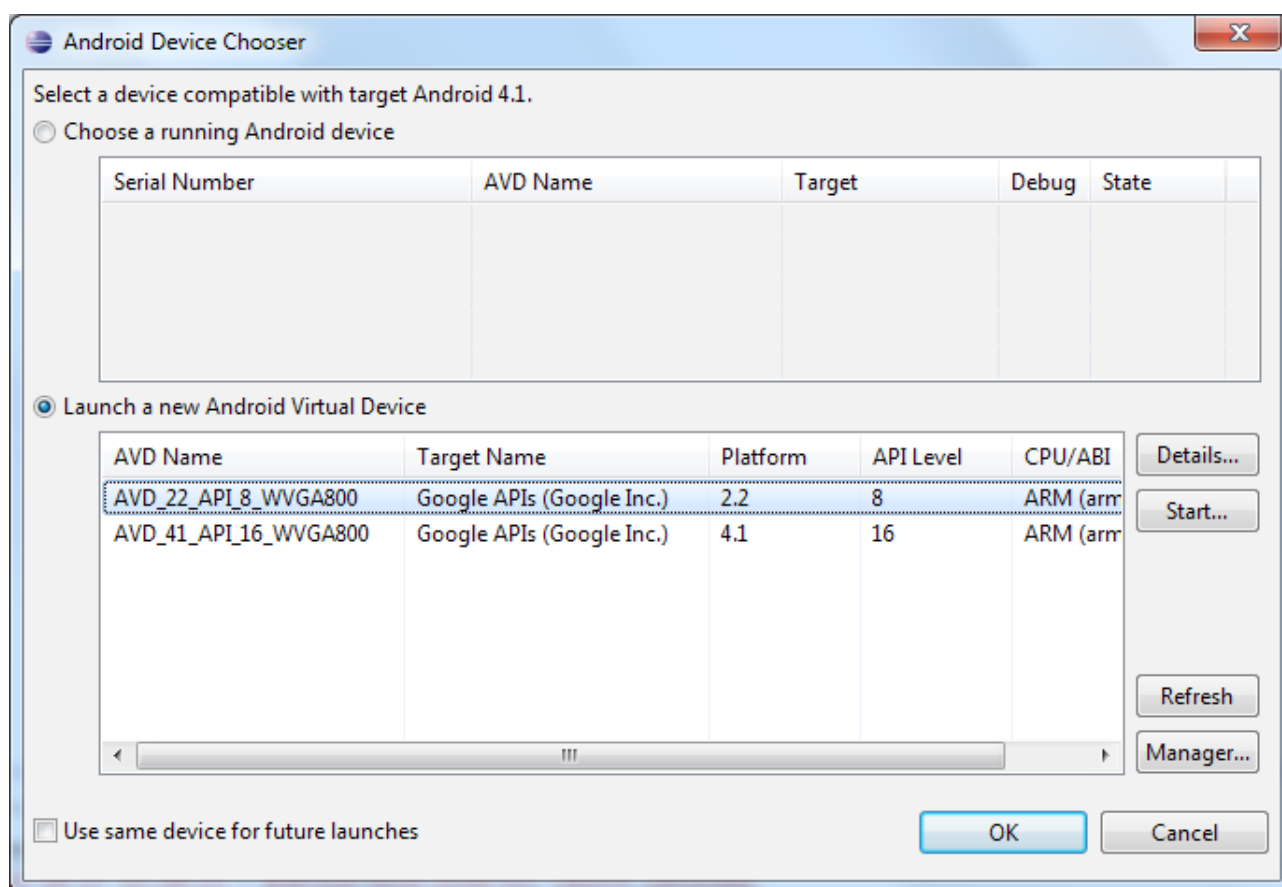
El resto de opciones las dejaremos por defecto y pasaremos a la pestaña "Target". En esta segunda pestaña podremos seleccionar el AVD sobre el que queremos ejecutar la aplicación, aunque suele ser práctico indicarle a Eclipse que nos pregunte esto antes de cada ejecución, de forma que podamos ir alternando fácilmente de AVD sin tener que volver a configurar el perfil. Para ello seleccionaremos la opción "Always prompt to pick device".



Un poco más abajo en esta misma pestaña es bueno marcar la opción "Disable Boot Animation" para acelerar un poco el primer arranque del emulador, y normalmente también suele ser necesario reducir, o mejor dicho escalar, la pantalla del emulador de forma que podamos verlo completo en la pantalla de nuestro PC. Esto se configura mediante la opción "Additional Emulator Command Line Options", donde en mi caso indicaré la opción "`-scale 0.75`", aunque este valor dependerá de la resolución de vuestro monitor y de la configuración del AVD.



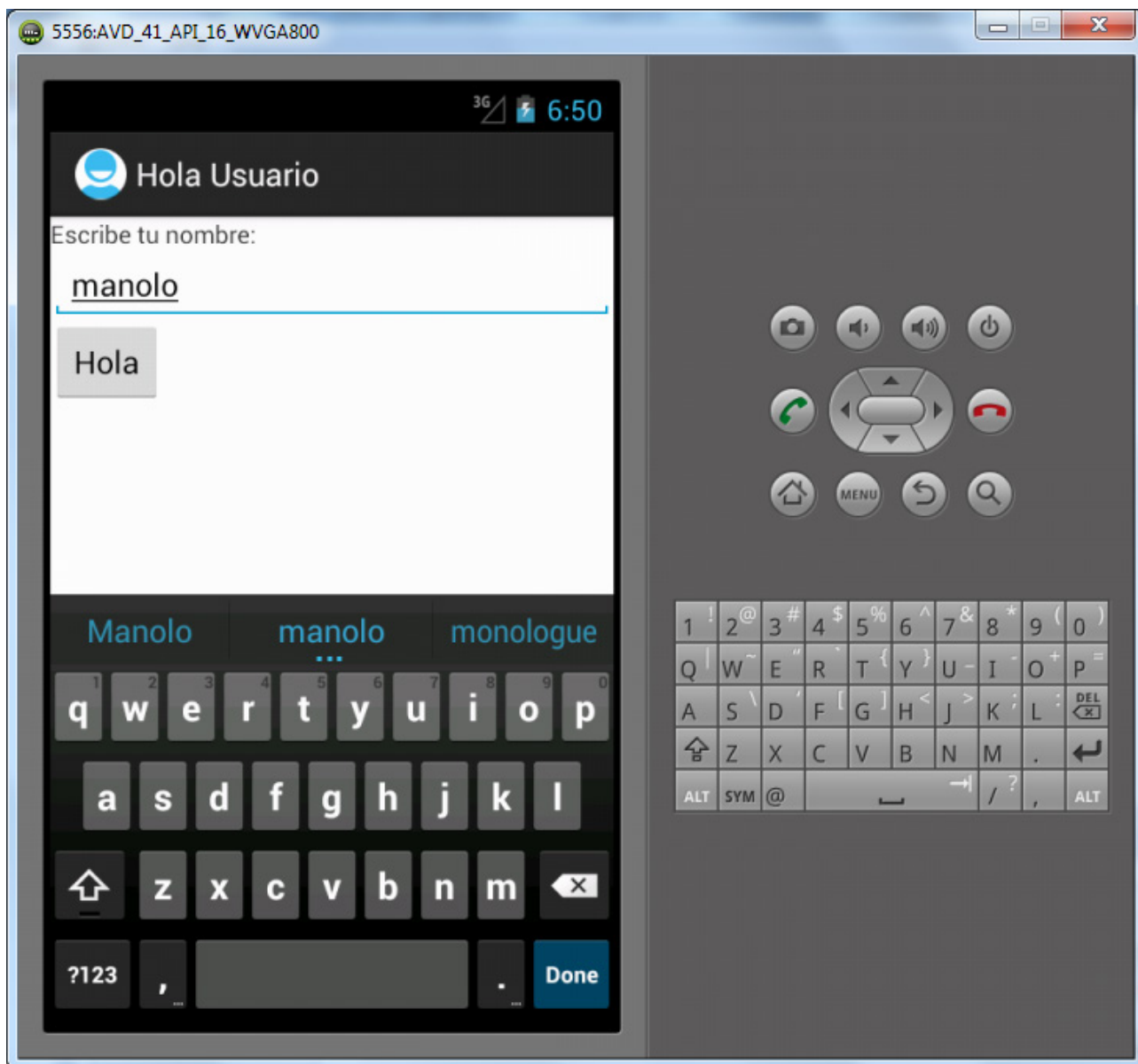
Tras esto ya podríamos pulsar el botón "Run" para ejecutar la aplicación en el emulador de Android. Eclipse nos preguntará en qué dispositivo queremos ejecutar y nos mostrará dos listas. La primera de ellas con los dispositivos que haya en ese momento en funcionamiento (por ejemplo si ya teníamos un emulador funcionando) y la siguiente con el resto de AVDs configurados en nuestro entorno. Elegiré en primer lugar el emulador con Android 2.2. Es posible que la primera ejecución se demore unos minutos, todo dependerá de las posibilidades de vuestro PC, así que paciencia.



Si todo va bien, tras una pequeña espera aparecerá el emulador de Android y se iniciará automáticamente nuestra aplicación. Podemos probar a escribir un nombre y pulsar el botón "Hola" para comprobar si el funcionamiento es el correcto.



Sin cerrar este emulador podríamos volver a ejecutar la aplicación sobre Android 4.2 seleccionando el AVD correspondiente. De cualquier forma, si vuestro PC no es demasiado potente no recomiendo tener dos emuladores abiertos al mismo tiempo.



Y con esto terminamos por ahora. Espero que esta aplicación de ejemplo os sea de ayuda para aprender temas básicos en el desarrollo para Android, como por ejemplo la definición de la interfaz gráfica, el código java necesario para acceder y manipular los elementos de dicha interfaz, y la forma de comunicar diferentes actividades de Android. En los apartados siguientes veremos algunos de estos temas de forma mucho más específica.



Puedes consultar y descargar el código fuente completo de este apartado accediendo a la página del curso en GitHub:

[curso-android-src/android-hola-usuario](https://github.com/curso-android-src/android-hola-usuario)

3

Widgets

III. Widgets

Widgets básicos

En los dos próximos capítulos vamos a describir cómo crear un *widget* de escritorio (*home screen widget*).

En esta primera parte construiremos un *widget* estático (no será interactivo, ni contendrá datos actualizables, ni responderá a eventos) muy básico para entender claramente la estructura interna de un componente de este tipo, y en el siguiente capítulo completaremos el ejercicio añadiendo una ventana de configuración inicial para el widget, añadiremos algún dato que podamos actualizar periódicamente, y haremos que responda a pulsaciones del usuario.

Como hemos dicho, en esta primera parte vamos a crear un *widget* muy básico, consistente en un simple marco rectangular negro con un mensaje de texto predeterminado ("*Mi Primer Widget*"). La sencillez del ejemplo nos permitirá centrarnos en los pasos principales de la construcción de un widget Android y olvidarnos de otros detalles que nada tienen que ver con el tema que nos ocupa (gráficos, datos, ...). Para que os hagáis una idea, éste será el aspecto final de nuestro widget de ejemplo:



Los pasos principales para la creación de un widget Android son los siguientes:

1. Definición de su interfaz gráfica (*layout*).
2. Configuración XML del widget (`AppWidgetProviderInfo`).
3. Implementación de la funcionalidad del widget (`AppWidgetProvider`), especialmente su evento de actualización.
4. Declaración del widget en el *Android Manifest* de la aplicación.

En el primer paso no nos vamos a detener mucho ya que es análogo a cualquier definición de layout de las que hemos visto hasta ahora en el curso. En esta ocasión, la interfaz del widget estará compuesta únicamente por un par de *frames* (`FrameLayout`), uno negro exterior y uno blanco interior algo más pequeño para

simular el marco, y una etiqueta de texto (`TextView`) que albergará el mensaje a mostrar. Veamos cómo queda el layout xml, que para este ejemplo llamaremos `miwidget.xml`:

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="#000000"
    android:padding="10dp"
    android:layout_margin="5dp" >

    <FrameLayout android:id="@+id/frmWidget"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:background="#FFFFFF"
        android:padding="5dp" >

        <TextView android:id="@+id/txtMensaje"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:textColor="#000000"
            android:text="@string/mi_primer_widget" />

    </FrameLayout>

</FrameLayout>
```

Cabe destacar aquí que, debido a que el layout de los widgets de Android está basado en un tipo especial de componentes llamados `RemoteViews`, no es posible utilizar en su interfaz todos los contenedores y controles que hemos visto en apartados anteriores sino sólo unos pocos básicos que se indican a continuación:

- Contenedores: `FrameLayout`, `LinearLayout`, `RelativeLayout` y `GridLayout` (éste último a partir de Android 4).
- Controles: `Button`, `ImageButton`, `ImageView`, `TextView`, `ProgressBar`, `Chronometer`, `AnalogClock` y `ViewFlipper`. A partir de Android 3 también podemos utilizar `ListView`, `GridView`, `StackView` y `AdapterViewFlipper`, aunque su uso tiene algunas particularidades. En este apartado no trataremos este último caso, pero si necesitas información puedes empezar por la [documentación oficial sobre el tema](#).

Aunque la lista de controles soportados no deja de ser curiosa (al menos en mi humilde opinión), debería ser suficiente para crear todo tipo de widgets.

Como segundo paso del proceso de construcción vamos a crear un nuevo fichero XML donde definiremos un conjunto de propiedades del widget, como por ejemplo su tamaño en pantalla o su frecuencia de actualización. Este XML se deberá crear en la carpeta `\res\xml` de nuestro proyecto.

En nuestro caso de ejemplo lo llamaremos `miwidget_wprovider.xml` y tendrá la siguiente estructura:

```
<?xml version="1.0" encoding="utf-8"?>
<appwidget-provider xmlns:android="http://schemas.android.com/apk/res/android"
    android:initialLayout="@layout/miwidget"
    android:minWidth="110dip"
    android:minHeight="40dip"
    android:label="@string/mi_primer_widget"
    android:updatePeriodMillis="3600000"
/>
```

Para nuestro widget estamos definiendo las siguientes propiedades:

- `initialLayout`: referencia al layout XML que hemos creado en el paso anterior.
- `minWidth`: ancho mínimo del widget en pantalla, en dp (*density-independent pixels*).
- `minHeight`: alto mínimo del widget en pantalla, en dp (*density-independent pixels*).
- `label`: nombre del widget que se mostrará en el menú de selección de Android.
- `updatePeriodMillis`: frecuencia de actualización del widget, en milisegundos.

Existen varias propiedades más que se pueden definir, por ejemplo el icono de vista previa del widget (`android:previewImage`, sólo para Android >3.0) o el indicativo de si el widget será redimensionable (`android:resizeMode`, sólo para Android >3.1) o la actividad de configuración del widget (`android:configure`). En el siguiente apartado utilizaremos alguna de ellas, el resto se pueden consultar en la documentación oficial de la clase `AppWidgetProviderInfo`.

Como sabemos, la pantalla inicial de Android se divide en un mínimo de 4×4 celdas (según el dispositivo pueden ser más) donde se pueden colocar aplicaciones, accesos directos y widgets. Teniendo en cuenta las diferentes dimensiones de estas celdas según el dispositivo y la orientación de la pantalla, existe una fórmula sencilla para ajustar las dimensiones de nuestro widget para que ocupe un número determinado de celdas sea cual sea la orientación:

- $\text{ancho_mínimo} = (\text{num_celdas} * 70) - 30$
- $\text{alto_mínimo} = (\text{num_celdas} * 70) - 30$

Atendiendo a esta fórmula, si queremos que nuestro widget ocupe por ejemplo un tamaño mínimo de 2 celdas de ancho por 1 celda de alto, deberemos indicar unas dimensiones de **110dp x 40dp**.

Vamos ahora con el tercer paso. Éste consiste en implementar la funcionalidad de nuestro widget en su clase java asociada. Esta clase deberá heredar de `AppWidgetProvider`, que a su vez no es más que una clase auxiliar derivada de `BroadcastReceiver`, ya que los widgets de Android no son más que un caso particular de este tipo de componentes.

En esta clase deberemos implementar los mensajes a los que vamos a responder desde nuestro widget, entre los que destacan:

- `onEnabled()`: lanzado cuando se crea la primera instancia de un widget.
- `onUpdate()`: lanzado periódicamente cada vez que se debe actualizar un widget, por ejemplo cada vez que se cumple el periodo de tiempo definido por el parámetro `updatePeriodMillis` antes descrito, o cuando se añade el widget al escritorio.
- `onDeleted()`: lanzado cuando se elimina del escritorio una instancia de un widget.
- `onDisabled()`: lanzado cuando se elimina del escritorio la última instancia de un widget.

En la mayoría de los casos, tendremos que implementar como mínimo el evento `onUpdate()`. El resto de métodos dependerán de la funcionalidad de nuestro widget. En nuestro caso particular no nos hará falta ninguno de ellos ya que el widget que estamos creando no contiene ningún dato actualizable, por lo que crearemos la clase, llamada `MiWidget`, pero dejaremos vacío por el momento el método `onUpdate()`. En el siguiente apartado veremos qué cosas podemos hacer dentro de estos métodos.

```
package net.sgoliver.android.widgets;

import android.appwidget.AppWidgetManager;
import android.appwidget.AppWidgetProvider;
import android.content.Context;

public class MiWidget extends AppWidgetProvider {
    @Override
    public void onUpdate(Context context,
        AppWidgetManager appWidgetManager,
        int[] appWidgetIds) {
        //Actualizar el widget
        //...
    }
}
```

El último paso del proceso será declarar el widget dentro del *manifest* de nuestra aplicación. Para ello, editaremos el fichero `AndroidManifest.xml` para incluir la siguiente declaración dentro del elemento `<application>`:

```
<application>
    ...
    <receiver android:name=".MiWidget" android:label="Mi Primer Widget">
        <intent-filter>
            <action android:name="android.appwidget.action.APPWIDGET_UPDATE" />
        </intent-filter>
        <meta-data
            android:name="android.appwidget.provider"
            android:resource="@xml/miwidget_wprovider" />
    </receiver>
</application>
```

El widget se declarará como un elemento `<receiver>` y deberemos aportar la siguiente información:

- Atributo `name`: Referencia a la clase java de nuestro widget, creada en el paso anterior.
- Elemento `<intent-filter>`, donde indicaremos los "eventos" a los que responderá nuestro widget, normalmente añadiremos el evento `APPWIDGET_UPDATE`, para detectar la acción de actualización.
- Elemento `<meta-data>`, donde haremos referencia con su atributo `resource` al XML de configuración que creamos en el segundo paso del proceso.

Con esto habríamos terminado de escribir los distintos elementos necesarios para hacer funcionar nuestro widget básico de ejemplo. Para probarlo, podemos ejecutar el proyecto de Eclipse en el emulador de Android, esperar a que se ejecute la aplicación principal (que estará vacía, ya que no hemos incluido ninguna funcionalidad para ella), ir a la pantalla principal del emulador y añadir nuestro widget al escritorio tal como lo haríamos en nuestro dispositivo físico.

- En **Android 2**: pulsación larga sobre el escritorio o tecla Menú, seleccionar la opción *Widgets*, y por último seleccionar nuestro Widget de la lista.
- En **Android 4**: accedemos al menú principal, pulsamos la pestaña *Widgets*, buscamos el nuestro en la lista y realizamos sobre él una pulsación larga hasta que el sistema nos deja arrastrarlo y colocarlo sobre el escritorio.

Con esto ya hemos conseguido la funcionalidad básica de un widget, es posible añadir varias instancias al escritorio, desplazarlos por la pantalla y eliminarlos enviándolos a la papelera.

En el próximo apartado veremos cómo podemos mejorar este widget añadiendo una pantalla de configuración inicial, mostraremos algún dato que se actualice periódicamente, y añadiremos la posibilidad de capturar eventos de pulsación sobre el widget.



Puedes consultar y descargar el código fuente completo de este apartado accediendo a la página del curso en GitHub:

[curso-android-src/android-widgets-1](https://github.com/curso-android-src/android-widgets-1)

Widgets avanzados

Ya hemos visto cómo construir un widget básico para Android, y prometimos que dedicaríamos un apartado adicional a comentar algunas características más avanzadas de este tipo de componentes. Pues bien, en este segundo apartado sobre el tema vamos a ver cómo podemos añadir los siguientes elementos y funcionalidades al widget básico que ya construimos:

- Pantalla de configuración inicial.
- Datos actualizables de forma periódica.
- Eventos de usuario.

Como sabéis, intento simplificar al máximo todos los ejemplos que utilizo en este curso para que podamos centrar nuestra atención en los aspectos realmente importantes. En esta ocasión utilizaré el mismo criterio, y las únicas características (aunque suficientes para demostrar los tres conceptos anteriores) que añadiremos a nuestro widget serán las siguientes:

1. Añadiremos una pantalla de configuración inicial del widget, que aparecerá cada vez que se añada una nueva instancia del widget a nuestro escritorio. En esta pantalla podrá configurarse únicamente el mensaje de texto a mostrar en el widget.
2. Añadiremos un nuevo elemento de texto al widget que muestre la hora actual. Esto nos servirá para comprobar que el widget se actualiza periódicamente.
3. Añadiremos un botón al widget, que al ser pulsado forzará la actualización inmediata del mismo.

Empecemos por el primer punto, la pantalla de configuración inicial del widget. Y procederemos igual que para el diseño de cualquier otra actividad Android, definiendo su layout xml. En nuestro caso será muy sencilla, un cuadro de texto para introducir el mensaje a personalizar y dos botones, uno para aceptar la configuración y otro para cancelar (en cuyo caso el widget no se añade al escritorio). En esta ocasión llamaremos a este layout "`widget_config.xml`". Veamos cómo queda:

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

    <TextView android:id="@+id/LblMensaje"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/mensaje_personalizado" />

    <EditText android:id="@+id/TxtMensaje"
        android:layout_height="wrap_content"
        android:layout_width="match_parent" />
```

```

<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="horizontal" >

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

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

</LinearLayout>
</LinearLayout>

```

Una vez diseñada la interfaz de nuestra actividad de configuración tendremos que implementar su funcionalidad en java. Llamaremos a la clase `WidgetConfig`, su estructura será análoga a la de cualquier actividad de Android, y las acciones a realizar serán las comentadas a continuación. En primer lugar nos hará falta el identificador de la instancia concreta del widget que se configurará con esta actividad. Este ID nos llega como parámetro del *intent* que ha lanzado la actividad. Como ya vimos en un apartado anterior del curso, este *intent* se puede recuperar mediante el método `getIntent()` y sus parámetros mediante el método `getExtras()`. Conseguida la lista de parámetros del intent, obtendremos el valor del ID del widget accediendo a la clave `AppWidgetManager.EXTRA_APPWIDGET_ID`. Veamos el código hasta este momento:

```

public class WidgetConfig extends Activity {
    private Button btnAceptar;
    private Button btnCancelar;
    private EditText txtMensaje;
    private int widgetId = 0;

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

        //Obtenemos el Intent que ha lanzado esta ventana
        //y recuperamos sus parámetros
        Intent intentOrigen = getIntent();
        Bundle params = intentOrigen.getExtras();

        //Obtenemos el ID del widget que se está configurando
        widgetId = params.getInt(
            AppWidgetManager.EXTRA_APPWIDGET_ID,
            AppWidgetManager.INVALID_APPWIDGET_ID);

        //Establecemos el resultado por defecto (si se pulsa el botón 'Atrás'
        //del teléfono será éste el resultado devuelto).
        setResult(RESULT_CANCELED);

        //...
    }
}

```

En el código también podemos ver como aprovechamos este momento para establecer el resultado por defecto a devolver por la actividad de configuración mediante el método `setResult()`. Esto es importante porque las actividades de configuración de widgets deben devolver siempre un resultado (`RESULT_OK` en caso de aceptarse la configuración, o `RESULT_CANCELED` en caso de salir de la configuración sin aceptar los cambios). Estableciendo aquí ya un resultado `RESULT_CANCELED` por defecto nos aseguramos de que si el usuario sale de la configuración pulsando el botón *Atrás* del teléfono no añadiremos el widget al escritorio, mismo resultado que si pulsáramos el botón "Cancelar" de nuestra actividad.

Como siguiente paso recuperamos las referencias a cada uno de los controles de la actividad de configuración:

```
//Obtenemos la referencia a los controles de la pantalla
btnAceptar = (Button)findViewById(R.id.BtnAceptar);
btnCancelar = (Button)findViewById(R.id.BtnCancelar);
txtMensaje = (EditText)findViewById(R.id.TxtMensaje);
```

Por último, implementaremos las acciones de los botones "Aceptar" y "Cancelar". En principio, el botón Cancelar no tendría por qué hacer nada, tan sólo finalizar la actividad mediante una llamada al método `finish()` ya que el resultado `CANCELED` ya se ha establecido por defecto anteriormente:

```
//Implementación del botón "Cancelar"
btnCancelar.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View arg0) {
        //Devolvemos como resultado: CANCELAR (RESULT_CANCELED)
        finish();
    }
});
```

En el caso del botón Aceptar tendremos que hacer más cosas:

1. Guardar de alguna forma el mensaje que ha introducido el usuario.
2. Actualizar manualmente la interfaz del widget según la configuración establecida.
3. Devolver el resultado `RESULT_OK` aportando además el ID del widget.

Para el primer punto nos ayudaremos de la API de Preferencias (para más información leer el capítulo dedicado a este tema). En nuestro caso, guardaremos una sola preferencia cuya clave seguirá el patrón "`msg_IdWidget`", esto nos permitirá distinguir el mensaje configurado para cada instancia del widget que añadamos a nuestro escritorio de Android.

El segundo paso indicado es necesario debido a que si definimos una actividad de configuración para un widget, será ésta la que tenga la responsabilidad de realizar la primera actualización del mismo en caso de ser necesario. Es decir, tras salir de la actividad de configuración no se lanzará automáticamente el evento `onUpdate()` del widget (sí se lanzará posteriormente y de forma periódica según la configuración del parámetro `updatePeriodMillis` del provider que veremos más adelante), sino que tendrá que ser la propia actividad quien fuerce la primera actualización. Para ello, simplemente obtendremos una referencia al widget manager de nuestro contexto mediante el método `AppWidgetManager.getInstance()` y con esta referencia llamaremos al método estático de actualización del widget `MiWidget.actualizarWidget()`, que actualizará los datos de todos los controles del widget (lo veremos un poco más adelante).

Por último, al resultado a devolver (`RESULT_OK`) deberemos añadir información sobre el ID de nuestro widget. Esto lo conseguimos creando un nuevo Intent que contenga como parámetro el ID del widget que recuperamos antes y estableciéndolo como resultado de la actividad mediante el método `setResult(resultado, intent)`. Por último llamaremos al método `finish()` para finalizar la actividad. Con estas indicaciones, veamos cómo quedaría el código del botón *Aceptar*:


```
//Implementación del botón "Aceptar"
btnAceptar.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View arg0) {
        //Guardamos el mensaje personalizado en las preferencias
        SharedPreferences prefs =
            getSharedPreferences("WidgetPrefs", Context.MODE_PRIVATE);
        SharedPreferences.Editor editor = prefs.edit();

        editor.putString("msg_" + widgetId,
            txtMensaje.getText().toString());
        editor.commit();

        //Actualizamos el widget tras la configuración
        AppWidgetManager appWidgetManager =
            AppWidgetManager.getInstance(WidgetConfig.this);
        MiWidget.actualizarWidget(WidgetConfig.this,
            appWidgetManager, widgetId);

        //Devolvemos como resultado: ACEPTAR (RESULT_OK)
        Intent resultado = new Intent();
        resultado.putExtra(AppWidgetManager.EXTRA_APPWIDGET_ID, widgetId);
        setResult(RESULT_OK, resultado);
        finish();
    }
});
```

Ya hemos terminado de implementar nuestra actividad de configuración. Pero para su correcto funcionamiento aún nos quedan dos detalles más por modificar. En primer lugar tendremos que declarar esta actividad en nuestro fichero `AndroidManifest.xml`, indicando que debe responder a los mensajes de tipo `APPWIDGET_CONFIGURE`:

```
<activity android:name=".WidgetConfig">
    <intent-filter>
        <action android:name="android.apwidget.action.APPWIDGET_CONFIGURE"/>
    </intent-filter>
</activity>
```

Por último, debemos indicar en el XML de configuración de nuestro widget (`xml\miwidget_wprovider.xml`) que al añadir una instancia de este widget debe mostrarse la actividad de configuración que hemos creado. Esto se consigue estableciendo el atributo `android:configure` del provider. Aprovecharemos además este paso para establecer el tiempo de actualización automática del widget al mínimo permitido por este parámetro (30 minutos) y el tamaño del widget a 3x2 celdas. Veamos cómo quedaría finalmente:

```
<appwidget-provider xmlns:android="http://schemas.android.com/apk/res/android"
    android:initialLayout="@layout/miwidget"
    android:minWidth="180dip"
    android:minHeight="110dip"
    android:label="@string/mi_primer_widget"
    android:updatePeriodMillis="3600000"
    android:configure="net.sgoliver.android.widgets.WidgetConfig"
/>
```

Con esto, ya tenemos todo listo para que al añadir nuestro widget al escritorio se muestre automáticamente la pantalla de configuración que hemos construido. Podemos ejecutar el proyecto en este punto y comprobar que todo funciona correctamente.

Como siguiente paso vamos a modificar el layout del widget que ya construimos en el apartado anterior para añadir una nueva etiqueta de texto donde mostraremos la hora actual, y un botón que nos servirá para forzar la actualización de los datos del widget:

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="#000000"
    android:padding="10dp"
    android:layout_margin="5dp" >

    <LinearLayout android:id="@+id/FrmWidget"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:background="#FFFFFF"
        android:padding="5dp"
        android:orientation="vertical">

        <TextView android:id="@+id/LblMensaje"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:textColor="#000000"
            android:text="" />

        <TextView android:id="@+id/LblHora"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:textColor="#000000"
            android:text="" />

        <Button android:id="@+id/BtnActualizar"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:textColor="#000000"
            android:text="@string/actualizar" />

    </LinearLayout>

</FrameLayout>
```

Hecho esto, tendremos que modificar la implementación de nuestro provider (*MiWidget.java*) para que en cada actualización del widget se actualicen sus controles con los datos correctos (recordemos que en el apartado anterior dejamos este evento de actualización vacío ya que no mostrábamos datos actualizables en el widget). Esto lo haremos dentro del evento *onUpdate()* de nuestro provider.

Como ya dijimos, los componentes de un widget se basan en un tipo especial de vistas que llamamos *Remote Views*. Pues bien, para acceder a la lista de estos componentes que constituyen la interfaz del widget construiremos un nuevo objeto *RemoteViews* a partir del ID del layout del widget. Obtenida la lista de componentes, tendremos disponibles una serie de métodos *set* (uno para cada tipo de datos básicos) para establecer las propiedades de cada control del widget. Estos métodos reciben como parámetros el ID del control, el nombre del método que queremos ejecutar sobre el control, y el valor a establecer. Además de estos métodos, contamos adicionalmente con una serie de métodos más específicos para establecer directamente el texto y otras propiedades sencillas de los controles *TextView*, *ImageView*, *ProgressBar* y *Chronometer*, como por ejemplo *setTextViewText(idControl, valor)* para establecer el texto de un control *TextView*. Pueden consultarse todos los métodos disponibles en la [documentación oficial](#) de la clase *RemoteViews*. De esta forma, si por ejemplo queremos establecer el texto del control cuyo id es *LblMensaje* haríamos lo siguiente:

```
RemoteViews controles = new RemoteViews(context.getPackageName(), R.layout.miwidget);
controles.setTextViewText(R.id.LblMensaje, "Mensaje de prueba");
```

El proceso de actualización habrá que realizarlo por supuesto para todas las instancias del widget que se hayan añadido al escritorio. Recordemos aquí que el evento `onUpdate()` recibe como parámetro la lista de widgets que hay que actualizar.

Dicho esto, creo que ya podemos mostrar cómo quedaría el código de actualización de nuestro widget:

```
@Override
public void onUpdate(Context context,
                    AppWidgetManager appWidgetManager,
                    int[] appWidgetIds) {

    //Iteramos la lista de widgets en ejecución
    for (int i = 0; i < appWidgetIds.length; i++)
    {
        //ID del widget actual
        int widgetId = appWidgetIds[i];

        //Actualizamos el widget actual
        actualizarWidget(context, appWidgetManager, widgetId);
    }
}

public static void actualizarWidget(Context context,
                                   AppWidgetManager appWidgetManager, int widgetId)
{
    //Recuperamos el mensaje personalizado para el widget actual
    SharedPreferences prefs =
        context.getSharedPreferences("WidgetPrefs", Context.MODE_PRIVATE);
    String mensaje = prefs.getString("msg_" + widgetId, "Hora actual:");

    //Obtenemos la lista de controles del widget actual
    RemoteViews controles =
        new RemoteViews(context.getPackageName(), R.layout.miwidget);

    //Actualizamos el mensaje en el control del widget
    controles.setTextViewText(R.id.LblMensaje, mensaje);

    //Obtenemos la hora actual
    Calendar calendario = new GregorianCalendar();
    String hora = calendario.getTime().toLocaleString();

    //Actualizamos la hora en el control del widget
    controles.setTextViewText(R.id.LblHora, hora);

    //Notificamos al manager de la actualización del widget actual
    appWidgetManager.updateAppWidget(widgetId, controles);
}
```

Como vemos, todo el trabajo de actualización para un widget lo hemos extraído a un método estático independiente, de forma que también podamos llamarlo desde otras partes de la aplicación (como hacemos por ejemplo desde la actividad de configuración para forzar la primera actualización del widget).

Además quiero destacar la última línea del código, donde llamamos al método `updateAppWidget()` del *widget manager*. Esto es importante y necesario, ya que de no hacerlo la actualización de los controles no se

reflejará correctamente en la interfaz del widget.

Tras esto, ya sólo nos queda implementar la funcionalidad del nuevo botón que hemos incluido en el widget para poder forzar la actualización del mismo. A los controles utilizados en los widgets de Android, que ya sabemos que son del tipo `RemoteView`, no podemos asociar eventos de la forma tradicional que hemos visto en múltiples ocasiones durante el curso. Sin embargo, en su lugar, tenemos la posibilidad de asociar a un evento (por ejemplo, el click sobre un botón) un determinado mensaje (*Pending Intent*) de tipo *broadcast* que será lanzado cada vez que se produzca dicho evento. Además, podremos configurar el widget (que como ya indicamos no es más que un componente de tipo *broadcast receiver*) para que capture esos mensajes, e implementar en su evento `onReceive()` las acciones necesarias a ejecutar tras capturar el mensaje. Con estas tres acciones simularemos la captura de eventos sobre controles de un widget.

Vamos por partes. En primer lugar hagamos que se lance un intent de tipo broadcast cada vez que se pulse el botón del widget. Para ello, en el método `actualizarWidget()` construiremos un nuevo Intent asociándole una acción personalizada, que en nuestro caso llamaremos por ejemplo `"net.sgoliver.android.widgets.ACTUALIZAR_WIDGET"`. Como parámetro del nuevo Intent insertaremos mediante `putExtra()` el ID del widget actual de forma que más tarde podamos saber el widget concreto que ha lanzado el mensaje (recordemos que podemos tener varias instancias del mismo widget en el escritorio). Por último crearemos el `PendingIntent` mediante el método `getBroadcast()` y lo asociaremos al evento `onClick` del control llamando a `setOnClickPendingIntent()` pasándole el ID del control, en nuestro caso el botón de "Actualizar". Veamos cómo queda todo esto dentro del método `actualizarWidget()`:

```
Intent intent = new Intent("net.sgoliver.android.widgets.ACTUALIZAR_WIDGET");
intent.putExtra(
    AppWidgetManager.EXTRA_APPWIDGET_ID, widgetId);

PendingIntent pendingIntent =
    PendingIntent.getBroadcast(context, widgetId,
        intent, PendingIntent.FLAG_UPDATE_CURRENT);

controles.setOnClickPendingIntent(R.id.BtnActualizar, pendingIntent);
```

También podemos hacer por ejemplo que si pulsamos en el resto del espacio del widget (el no ocupado por el botón) se abra automáticamente la actividad principal de nuestra aplicación. Se haría de forma análoga, con la única diferencia que en vez de utilizar `getBroadcast()` utilizaríamos `getActivity()` y el `Intent` lo construiríamos a partir de la clase de la actividad principal:

```
Intent intent2 = new Intent(context, MainActivity.class);
PendingIntent pendingIntent2 =
    PendingIntent.getActivity(context, widgetId,
        intent2, PendingIntent.FLAG_UPDATE_CURRENT);

controles.setOnClickPendingIntent(R.id.FrmWidget, pendingIntent2);
```

Ahora vamos a declarar en el *Android Manifest* este mensaje personalizado, de forma que el widget sea capaz de capturarlo. Para ello, añadiremos simplemente un nuevo elemento `<intent-filter>` con nuestro nombre de acción personalizado dentro del componente `<receiver>` que ya teníamos definido:

```

<receiver android:name=".MiWidget" android:label="Mi Primer Widget">
    <intent-filter>
        <action android:name="android.appwidget.action.APPWIDGET_UPDATE" />
    </intent-filter>
    <intent-filter>
        <action android:name="net.sgoliver.android.widgets.ACTUALIZAR_WIDGET"/>
    </intent-filter>
    <meta-data
        android:name="android.appwidget.provider"
        android:resource="@xml/miwidget_wprovider" />
</receiver>

```

Por último, vamos a implementar el evento `onReceive()` del widget para actuar en caso de recibir nuestro mensaje de actualización personalizado. Dentro de este evento comprobaremos si la acción del mensaje recibido es la nuestra, y en ese caso recuperaremos el ID del widget que lo ha lanzado, obtendremos una referencia al *widget manager*, y por último llamaremos a nuestro método estático de actualización pasándole estos datos.

```

@Override
public void onReceive(Context context, Intent intent) {
    if (intent.getAction().equals(
        "net.sgoliver.android.widgets.ACTUALIZAR_WIDGET")) {

        //Obtenemos el ID del widget a actualizar
        int widgetId = intent.getIntExtra(
            AppWidgetManager.EXTRA_APPWIDGET_ID,
            AppWidgetManager.INVALID_APPWIDGET_ID);

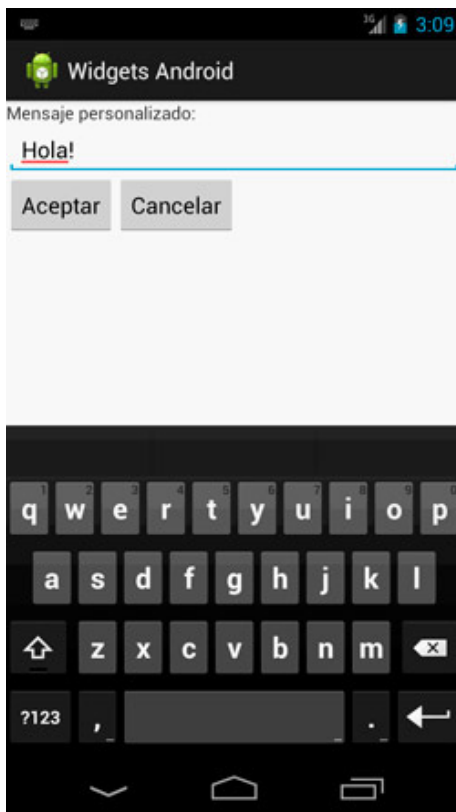
        //Obtenemos el widget manager de nuestro contexto
        AppWidgetManager widgetManager =
            AppWidgetManager.getInstance(context);

        //Actualizamos el widget
        if (widgetId != AppWidgetManager.INVALID_APPWIDGET_ID) {
            actualizarWidget(context, widgetManager, widgetId);
        }
    }
}

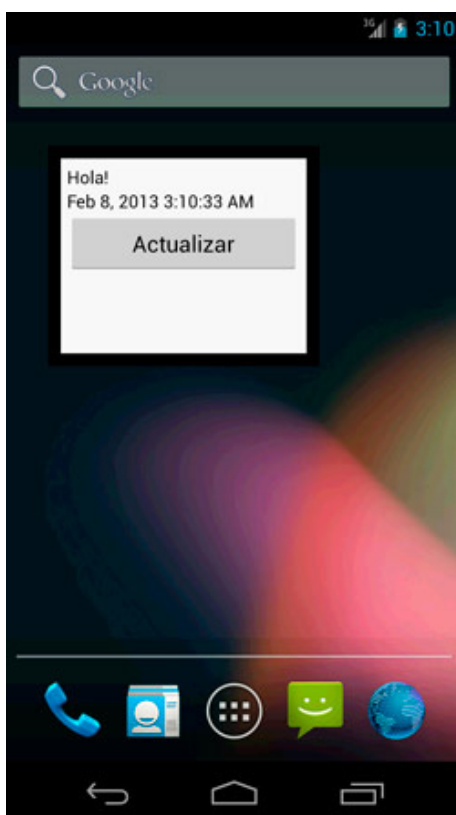
```

Con esto, por fin, hemos ya finalizado la construcción de nuestro widget Android y podemos ejecutar el proyecto de Eclipse para comprobar que todo funciona correctamente, tanto para una sola instancia como para varias instancias simultaneas.

Cuando añadamos el widget al escritorio nos aparecerá la pantalla de configuración que hemos definido:



Una vez introducido el mensaje que queremos mostrar, pulsaremos el botón *Aceptar* y el widget aparecerá automáticamente en el escritorio con dicho mensaje, la fecha-hora actual y el botón *Actualizar*.



Un comentario final, la actualización automática del widget se ha establecido a la frecuencia mínima que permite el atributo `updatePeriodMillis` del widget provider, que son 30 minutos. Por tanto es difícil y aburrido esperar para verla en funcionamiento mientras probamos el widget en el emulador. Pero funciona, os lo aseguro. De cualquier forma, esos 30 minutos pueden ser un periodo demasiado largo de tiempo

según la funcionalidad que queramos dar a nuestro widget, que puede requerir tiempos de actualización mucho más cortos (ojo con el rendimiento y el gasto de batería). Para solucionar esto podemos hacer uso de Alarmas, pero por ahora no nos preocuparemos de esto.



Puedes consultar y descargar el código fuente completo de este apartado accediendo a la página del curso en GitHub:

[curso-android-src/android-widgets-2](https://github.com/curso-android-src/android-widgets-2)

7

Preferencias

VII. Preferencias en Android

Preferencias Compartidas

Las preferencias no son más que datos que una aplicación debe guardar de algún modo para personalizar la experiencia del usuario, por ejemplo información personal, opciones de presentación, etc. En apartados anteriores vimos ya uno de los métodos disponibles en la plataforma Android para almacenar datos, como son las bases de datos SQLite. Las preferencias de una aplicación se podrían almacenar por su puesto utilizando este método, y no tendría nada de malo, pero Android proporciona otro método alternativo diseñado específicamente para administrar este tipo de datos: las *preferencias compartidas* o *shared preferences*. Cada preferencia se almacenará en forma de clave-valor, es decir, cada una de ellas estará compuesta por un identificador único (p.e. "email") y un valor asociado a dicho identificador (p.e. "prueba@email.com"). Además, y a diferencia de SQLite, los datos no se guardan en un fichero binario de base de datos, sino en ficheros XML como veremos al final de este apartado.

La API para el manejo de estas preferencias es muy sencilla. Toda la gestión se centraliza en la clase `SharedPreferences`, que representará a una colección de preferencias. Una aplicación Android puede gestionar varias colecciones de preferencias, que se diferenciarán mediante un identificador único. Para obtener una referencia a una colección determinada utilizaremos el método `getSharedPreferences()` al que pasaremos el identificador de la colección y un modo de acceso. El modo de acceso indicará qué aplicaciones tendrán acceso a la colección de preferencias y qué operaciones tendrán permitido realizar sobre ellas. Así, tendremos tres posibilidades principales:

- `MODE_PRIVATE`. Sólo nuestra aplicación tiene acceso a estas preferencias.
- `MODE_WORLD_READABLE`. Todas las aplicaciones pueden leer estas preferencias, pero sólo la nuestra puede modificarlas.
- `MODE_WORLD_WRITEABLE`. Todas las aplicaciones pueden leer y modificar estas preferencias.

Las dos últimas opciones son relativamente "peligrosas" por lo que en condiciones normales no deberían usarse. De hecho, se han declarado como obsoletas en la API 17 (Android 4.2).

Teniendo todo esto en cuenta, para obtener una referencia a una colección de preferencias llamada por ejemplo "MisPreferencias" y como modo de acceso exclusivo para nuestra aplicación haríamos lo siguiente:

```
SharedPreferences prefs =  
    getSharedPreferences("MisPreferencias", Context.MODE_PRIVATE);
```

Una vez hemos obtenido una referencia a nuestra colección de preferencias, ya podemos obtener, insertar o modificar preferencias utilizando los métodos `get` o `put` correspondientes al tipo de dato de cada preferencia. Así, por ejemplo, para obtener el valor de una preferencia llamada "email" de tipo `String` escribiríamos lo siguiente:

```
SharedPreferences prefs =  
    getSharedPreferences("MisPreferencias", Context.MODE_PRIVATE);  
  
String correo = prefs.getString("email", "por_defecto@email.com");
```

Como vemos, al método `getString()` le pasamos el nombre de la preferencia que queremos recuperar y un segundo parámetro con un valor por defecto. Este valor por defecto será el devuelto por el método `getString()` si la preferencia solicitada no existe en la colección. Además del método `getString()`, existen por supuesto métodos análogos para el resto de tipos de datos básicos, por ejemplo `getInt()`, `getLong()`, `getFloat()`, `getBoolean()`, ...

Para actualizar o insertar nuevas preferencias el proceso será igual de sencillo, con la única diferencia de que la actualización o inserción no la haremos directamente sobre el objeto `SharedPreferences`, sino sobre su objeto de edición `SharedPreferences.Editor`. A este último objeto accedemos mediante el método `edit()` de la clase `SharedPreferences`. Una vez obtenida la referencia al editor, utilizaremos los métodos `put` correspondientes al tipo de datos de cada preferencia para actualizar/insertar su valor, por ejemplo `putString(clave, valor)`, para actualizar una preferencia de tipo `String`. De forma análoga a los métodos `get` que ya hemos visto, tendremos disponibles métodos `put` para todos los tipos de datos básicos: `putInt()`, `putFloat()`, `putBoolean()`, etc. Finalmente, una vez actualizados/insertados todos los datos necesarios llamaremos al método `commit()` para confirmar los cambios. Veamos un ejemplo sencillo:

```
SharedPreferences prefs =
    getSharedPreferences("MisPreferencias", Context.MODE_PRIVATE);

SharedPreferences.Editor editor = prefs.edit();
editor.putString("email", "modificado@email.com");
editor.putString("nombre", "Prueba");
editor.commit();
```

¿Pero dónde se almacenan estas preferencias compartidas? Como dijimos al comienzo del apartado, las preferencias no se almacenan en ficheros binarios como las bases de datos SQLite, sino en ficheros XML. Estos ficheros XML se almacenan en una ruta que sigue el siguiente patrón:

```
/data/data/paquetejava/shared_prefs/nombre_coleccion.xml
```

Así, por ejemplo, en nuestro caso encontraríamos nuestro fichero de preferencias en la ruta:

```
/data/data/net.sgoliver.android.preferences1/shared_prefs/MisPreferencias.xml
```

Sirva una imagen del explorador de archivos del DDMS como prueba:

net.sgoliver.android.preferences1		2013-02-08	03:23	drwxr-x--x
└─ cache		2013-02-08	03:23	drwxrwx--x
└─ lib		2013-02-08	03:23	lrwxrwxrwx
└─ shared_prefs		2013-02-08	03:23	drwxrwx--x
MisPreferencias.xml	159	2013-02-08	03:23	-rw-rw----

Si descargamos este fichero desde el DDMS y lo abrimos con cualquier editor de texto veremos un contenido como el siguiente:

```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
  <string name="nombre">prueba</string>
  <string name="email">modificado@email.com</string>
</map>
```

En este XML podemos observar cómo se han almacenado las dos preferencias de ejemplo que insertamos anteriormente, con sus claves y valores correspondientes.

Y nada más, así de fácil y práctico. Con esto hemos aprendido una forma sencilla de almacenar determinadas opciones de nuestra aplicación sin tener que recurrir para ello a definir bases de datos SQLite, que aunque tampoco añaden mucha dificultad sí que requieren algo más de trabajo por nuestra parte.

Se aporta una pequeña aplicación de ejemplo para este apartado que tan sólo incluye dos botones, el primero de ellos para guardar las preferencias tal como hemos descrito, y el segundo para recuperarlas y mostrarlas en el log.

En una segunda parte de este tema dedicado a las preferencias veremos cómo Android nos ofrece otra forma de gestionar estos datos, que se integra además fácilmente con la interfaz gráfica necesaria para solicitar los datos al usuario.



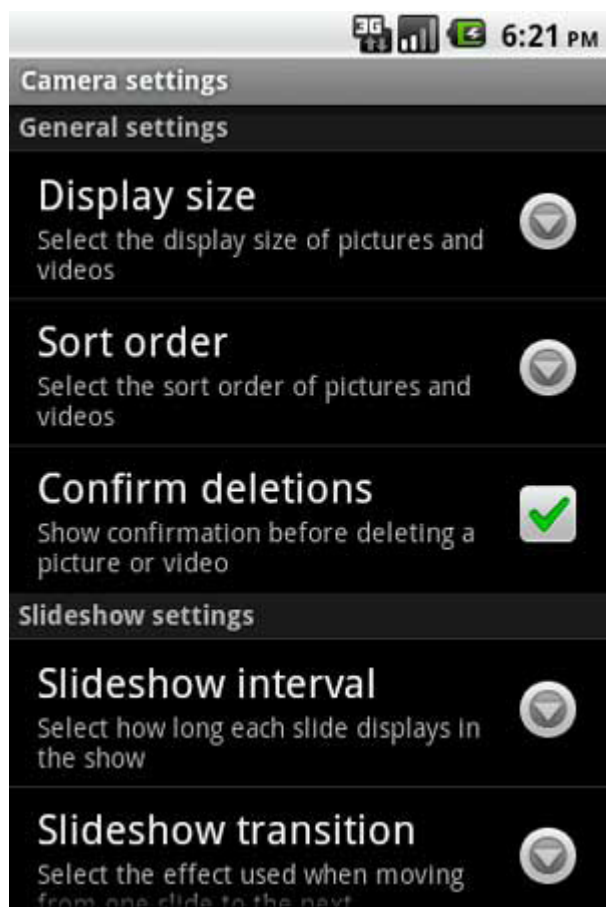
Puedes consultar y descargar el código fuente completo de este apartado accediendo a la página del curso en GitHub:

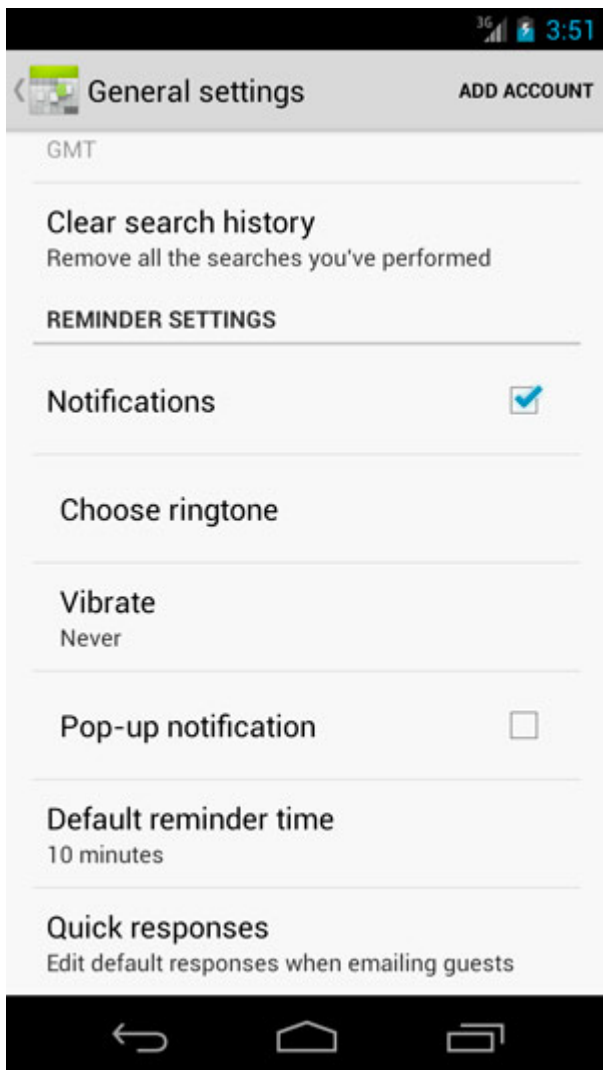
[curso-android-src/android-preferences-1](https://github.com/curso-android-src/android-preferences-1)

Pantallas de Preferencias

En el apartado anterior hemos hablado de las *Shared Preferences*, un mecanismo que nos permite gestionar fácilmente las opciones de una aplicación permitiéndonos guardarlas en XML de una forma transparente para el programador. Y vimos cómo hacer uso de ellas mediante código, es decir, creando nosotros mismos los objetos necesarios (*SharedPreferences*) y añadiendo, modificando y/o recuperando "a mano" los valores de las opciones a través de los métodos correspondientes (*getString()*, *putString()*, ...). Sin embargo, ya avisamos de que Android ofrece una forma alternativa de definir mediante XML un conjunto de opciones para una aplicación y crear por nosotros las pantallas necesarias para permitir al usuario modificarlas a su antojo. A esto dedicaremos este segundo apartado sobre preferencias.

Si nos fijamos en cualquier pantalla de preferencias estándar de Android veremos que todas comparten una interfaz común, similar por ejemplo a la que se muestra en las imágenes siguientes para Android 2.x y Android 4.x respectivamente:





Si atendemos por ejemplo a la primera imagen vemos cómo las diferentes opciones se organizan dentro de la **pantalla de opciones** en varias **categorías** ("*General Settings*" y "*Slideshow Settings*"). Dentro de cada categoría pueden aparecer varias opciones de diversos **tipos**, como por ejemplo de tipo checkbox ("*Confirm deletions*") o de tipo lista de selección ("*Display size*"). He resaltado las palabras "*pantalla de opciones*", "*categorías*", y "*tipos de opción*" porque serán estos los tres elementos principales con los que vamos a definir el conjunto de opciones o preferencias de nuestra aplicación. Empecemos.

Como hemos indicado, nuestra pantalla de opciones la vamos a definir mediante un XML, de forma similar a como definimos cualquier layout, aunque en este caso deberemos colocarlo en la carpeta `/res/xml`. El contenedor principal de nuestra pantalla de preferencias será el elemento `<PreferenceScreen>`. Este elemento representará a la pantalla de opciones en sí, dentro de la cual incluiremos el resto de elementos. Dentro de éste podremos incluir nuestra lista de opciones organizadas por categorías, que se representarán mediante el elemento `<PreferenceCategory>` al que daremos un texto descriptivo utilizando su atributo `android:title`. Dentro de cada categoría podremos añadir cualquier número de opciones, las cuales pueden ser de distintos tipos, entre los que destacan:

Tipo	Descripción
CheckBoxPreference	Marca seleccionable.
EditTextPreference	Cadena simple de texto.
ListPreference	Lista de valores seleccionables (exclusiva).
MultiSelectListPreference	Lista de valores seleccionables (múltiple).

Cada uno de estos tipos de preferencia requiere la definición de diferentes atributos, que iremos viendo en

los siguientes apartados.

CheckBoxPreference

Representa un tipo de opción que sólo puede tomar dos valores distintos: activada o desactivada. Es el equivalente a un control de tipo *checkbox*. En este caso tan sólo tendremos que especificar los atributos: nombre interno de la opción (`android:key`), texto a mostrar (`android:title`) y descripción de la opción (`android:summary`). Veamos un ejemplo:

```
<CheckBoxPreference
    android:key="opcion1"
    android:title="Preferencia 1"
    android:summary="Descripción de la preferencia 1" />
```

EditTextPreference

Representa un tipo de opción que puede contener como valor una cadena de texto. Al pulsar sobre una opción de este tipo se mostrará un cuadro de diálogo sencillo que solicitará al usuario el texto a almacenar. Para este tipo, además de los tres atributos comunes a todas las opciones (`key`, `title` y `summary`) también tendremos que indicar el texto a mostrar en el cuadro de diálogo, mediante el atributo `android:dialogTitle`. Un ejemplo sería el siguiente:

```
<EditTextPreference
    android:key="opcion2"
    android:title="Preferencia 2"
    android:summary="Descripción de la preferencia 2"
    android:dialogTitle="Introduce valor" />
```

ListPreference

Representa un tipo de opción que puede tomar como valor un elemento, y sólo uno, seleccionado por el usuario entre una lista de valores predefinida. Al pulsar sobre una opción de este tipo se mostrará la lista de valores posibles y el usuario podrá seleccionar uno de ellos. Y en este caso seguimos añadiendo atributos. Además de los cuatro ya comentados (`key`, `title`, `summary` y `dialogTitle`) tendremos que añadir dos más, uno de ellos indicando la lista de valores a visualizar en la lista y el otro indicando los valores internos que utilizaremos para cada uno de los valores de la lista anterior (Ejemplo: al usuario podemos mostrar una lista con los valores "*Español*" y "*Francés*", pero internamente almacenarlos como "*ESP*" y "*FRA*").

Estas listas de valores las definiremos también como ficheros XML dentro de la carpeta `/res/xml`. Definiremos para ello los recursos de tipos `<string-array>` necesarios, en este caso dos, uno para la lista de valores visibles y otro para la lista de valores internos, cada uno de ellos con su ID único correspondiente. Veamos cómo quedarían dos listas de ejemplo, en un fichero llamado "`codigospaíses.xml`":

```
<?xml version="1.0" encoding="utf-8" ?>
<resources>
    <string-array name="pais">
        <item>España</item>
        <item>Francia</item>
        <item>Alemania</item>
    </string-array>
    <string-array name="codigopais">
        <item>ESP</item>
        <item>FRA</item>
        <item>ALE</item>
    </string-array>
</resources>
```

En la preferencia utilizaremos los atributos `android:entries` y `android:entryValues` para hacer referencia a estas listas, como vemos en el ejemplo siguiente:

```
<ListPreference
    android:key="opcion3"
    android:title="Preferencia 3"
    android:summary="Descripción de la preferencia 3"
    android:dialogTitle="Indicar Pais"
    android:entries="@array/pais"
    android:entryValues="@array/codigopais" />
```

MultiSelectListPreference

[A partir de Android 3.0.x/Honeycomb] Las opciones de este tipo son muy similares a las `ListPreference`, con la diferencia de que el usuario puede seleccionar varias de las opciones de la lista de posibles valores. Los atributos a asignar son por tanto los mismos que para el tipo anterior.

```
<MultiSelectListPreference
    android:key="opcion4"
    android:title="Preferencia 4"
    android:summary="Descripción de la preferencia 4"
    android:dialogTitle="Indicar Pais"
    android:entries="@array/pais"
    android:entryValues="@array/codigopais" />
```

Como ejemplo completo, veamos cómo quedaría definida una pantalla de opciones con las 3 primeras opciones comentadas (ya que probaré con Android 2.2), divididas en 2 categorías llamadas por simplicidad "*Categoría 1*" y "*Categoría 2*". Llamaremos al fichero "`opciones.xml`".

```
<PreferenceScreen
    xmlns:android="http://schemas.android.com/apk/res/android">
    <PreferenceCategory android:title="Categoría 1">
        <CheckBoxPreference
            android:key="opcion1"
            android:title="Preferencia 1"
            android:summary="Descripción de la preferencia 1" />
        <EditTextPreference
            android:key="opcion2"
            android:title="Preferencia 2"
            android:summary="Descripción de la preferencia 2"
            android:dialogTitle="Introduce valor" />
    </PreferenceCategory>
    <PreferenceCategory android:title="Categoría 2">
        <ListPreference
            android:key="opcion3"
            android:title="Preferencia 3"
            android:summary="Descripción de la preferencia 3"
            android:dialogTitle="Indicar Pais"
            android:entries="@array/pais"
            android:entryValues="@array/codigopais" />
    </PreferenceCategory>
</PreferenceScreen>
```

Ya tenemos definida la estructura de nuestra pantalla de opciones, pero aún nos queda un paso más para poder hacer uso de ella desde nuestra aplicación. Además de la definición XML de la lista de opciones, debemos implementar una nueva actividad, que será a la que hagamos referencia cuando queramos mostrar nuestra pantalla de opciones y la que se encargará internamente de gestionar todas las opciones,

guardarlas, modificarlas, etc, a partir de nuestra definición XML.

Android nos facilita las cosas ofreciéndonos una clase de la que podemos derivar fácilmente la nuestra propia y que hace casi todo el trabajo por nosotros. Esta clase se llama `PreferenceActivity`. Tan sólo deberemos crear una nueva actividad (yo la he llamado `OpcionesActivity`) que extienda a esta clase, e implementar su evento `onCreate()` para añadir una llamada al método `addPreferencesFromResource()`, mediante el que indicaremos el fichero XML en el que hemos definido la pantalla de opciones. Lo vemos mejor directamente en el código:

```
public class OpcionesActivity extends PreferenceActivity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        addPreferencesFromResource(R.xml.opciones);
    }
}
```

Así de sencillo, nuestra nueva actividad, al extender a `PreferenceActivity`, se encargará por nosotros de crear la interfaz gráfica de nuestra lista de opciones según la hemos definido en el XML y se preocupará por nosotros de mostrar, modificar y guardar las opciones cuando sea necesario tras la acción del usuario.

Aunque esto continúa funcionando sin problemas en versiones recientes de Android, la API 11 trajo consigo una nueva forma de definir las pantallas de preferencias haciendo uso de fragments. Para ello, basta simplemente con definir la clase java del fragment, que deberá extender de `PreferenceFragment` y añadir a su método `onCreate()` una llamada a `addPreferencesFromResource()` igual que ya hemos visto antes.

```
public static class OpcionesFragment extends PreferenceFragment {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        addPreferencesFromResource(R.xml.opciones);
    }
}
```

Hecho esto ya no será necesario que la clase de nuestra pantalla de preferencias extienda de `PreferenceActivity`, sino que podrá ser una actividad normal. Para mostrar el fragment creado como contenido principal de la actividad utilizaríamos el *fragment manager* para sustituir el contenido de la pantalla (`android.R.id.content`) por el de nuestro fragment de preferencias recién definido:

```
public class SettingsActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        getFragmentManager().beginTransaction()
            .replace(android.R.id.content, new OpcionesFragment())
            .commit();
    }
}
```

Sea cual se la opción elegida para definir la pantalla de preferencias, el siguiente paso será añadir esta actividad al fichero `AndroidManifest.xml`, al igual que cualquier otra actividad que utilicemos en la aplicación.

```
<activity android:name=".OpcionesActivity"
    android:label="@string/app_name">
</activity>
```

Ya sólo nos queda añadir a nuestra aplicación algún mecanismo para mostrar la pantalla de preferencias. Esta opción suele estar en un menú (para Android 2.x) o en el menú de overflow de la action bar (para Android 3 o superior), pero por simplificar el ejemplo vamos a añadir simplemente un botón (`btnPreferencias`) que abra la ventana de preferencias.

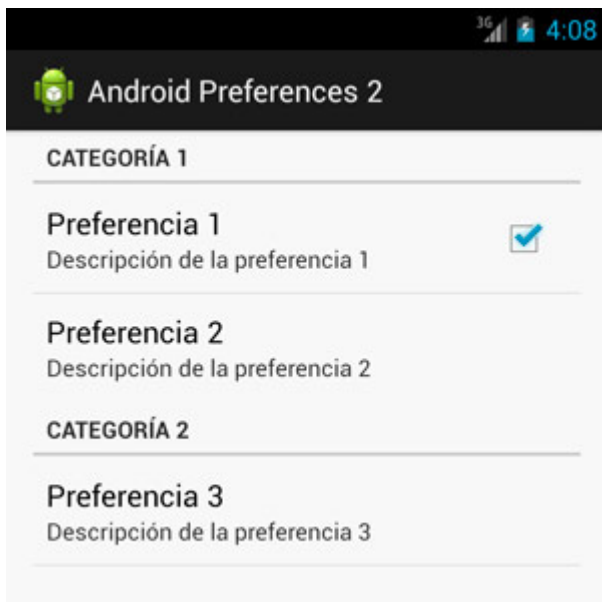
Al pulsar este botón llamaremos a la ventana de preferencias mediante el método `startActivity()`, como ya hemos visto en alguna ocasión, al que pasaremos como parámetros el contexto de la aplicación (nos vale con nuestra actividad principal) y la clase de la ventana de preferencias (`OpcionesActivity.class`).

```
btnPreferencias = (Button)findViewById(R.id.BtnPreferencias);

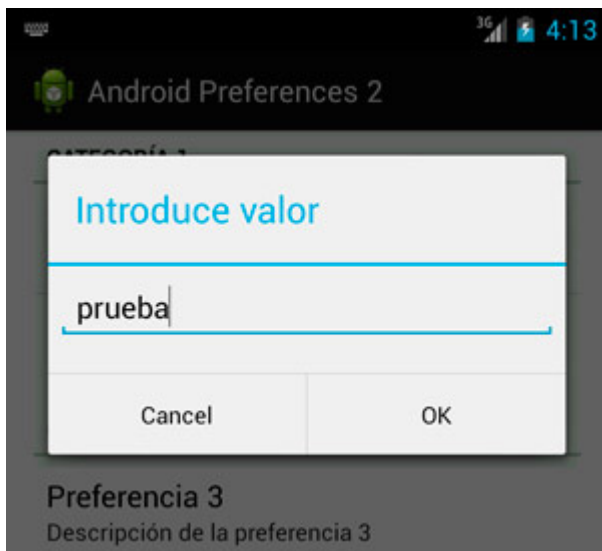
btnPreferencias.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
        startActivity(new Intent(MainActivity.this,
                                OpcionesActivity.class));
    }
});
```

Y esto es todo, ya sólo nos queda ejecutar la aplicación en el emulador y pulsar el botón de preferencias para mostrar nuestra nueva pantalla de opciones. Debe quedar como muestran las imágenes siguientes (para Android 2 y 4 respectivamente):

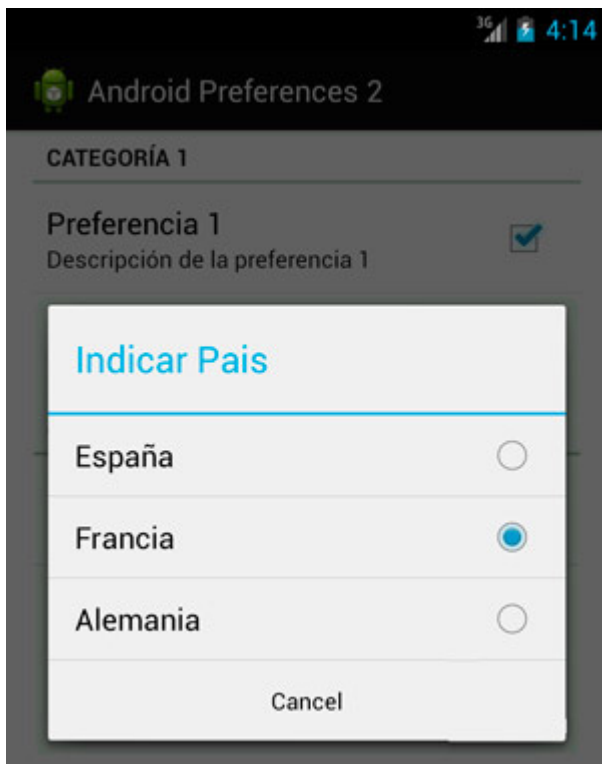




La primera opción podemos marcarla o desmarcarla directamente pulsando sobre la check de su derecha. La segunda, de tipo texto, nos mostrará al pulsarla un pequeño formulario para solicitar el valor de la opción.



Por último, la opción 3 de tipo lista, nos mostrará una ventana emergente con la lista de valores posibles, donde podremos seleccionar sólo uno de ellos.



Una vez establecidos los valores de las preferencias podemos salir de la ventana de opciones simplemente pulsando el botón Atrás del dispositivo o del emulador. Nuestra actividad `OpcionesActivity` se habrá ocupado por nosotros de guardar correctamente los valores de las opciones haciendo uso de la API de preferencias compartidas (*Shared Preferences*). Y para comprobarlo vamos a añadir otro botón (`btnObtenerOpciones`) a la aplicación de ejemplo que recupere el valor actual de las 3 preferencias y los escriba en el log de la aplicación.

La forma de acceder a las preferencias compartidas de la aplicación ya la vimos en el apartado anterior sobre este tema. Obtenemos la lista de preferencias mediante el método `getDefaultSharedPreferences()` y posteriormente utilizamos los distintos métodos `get()` para recuperar el valor de cada opción dependiendo de su tipo.

```
btnObtenerPreferencias.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
        SharedPreferences pref =
            PreferenceManager.getDefaultSharedPreferences(
                AndroidPrefScreensActivity.this);

        Log.i("", "Opción 1: " + pref.getBoolean("opcion1", false));
        Log.i("", "Opción 2: " + pref.getString("opcion2", ""));
        Log.i("", "Opción 3: " + pref.getString("opcion3", ""));
    }
});
```

Si ejecutamos ahora la aplicación, establecemos las preferencias y pulsamos el nuevo botón de consulta que hemos creado veremos cómo en el log de la aplicación aparecen los valores correctos de cada preferencia. Se mostraría algo como lo siguiente:

```
10-08 09:27:09.681: INFO/(1162): Opción 1: true
10-08 09:27:09.681: INFO/(1162): Opción 2: prueba
10-08 09:27:09.693: INFO/(1162): Opción 3: FRA
```

Y hasta aquí hemos llegado con el tema de las preferencias, un tema muy interesante de controlar ya que casi ninguna aplicación se libra de hacer uso de ellas. Existen otras muchas opciones de configuración de las pantallas de preferencias, sobre todo con la llegada de Android 4, pero con lo que hemos visto aquí podremos cubrir la gran mayoría de casos.



Puedes consultar y descargar el código fuente completo de este apartado accediendo a la página del curso en GitHub:

[curso-android-src/android-preferences-2](https://github.com/course-android-src/android-preferences-2)