Programación de Móviles

Publicación 1.0

Oscar Gomez

26 de January de 2015

Índice general

Índice:

CAPÍTULO 1

Análisis de tecnologías para aplicaciones en dispositivos móviles:

1.1 Puntos iniciales

- En primer lugar, en http://10.8.0.253 se puede encontrar un servidor.
- Los apuntes también se podrán encontrar a diario en http://oscarmaestre.github.io
- Existen fotocopias con la programación, criterios, etc... en la mesa del profesor. En cualquier caso, están colgadas en la página del centro http://www.iesmaestredecalatrava.es (buscar el apartado "Presentaciones")
- Si se desea acceder a algún fichero individual de los apuntes puede hacerse en las página siguiente
  - https://github.com/OscarMaestre/Moviles
  - https://github.com/OscarMaestre/ServiciosYProcesos

1.2 Sistemas operativos para dispositivos móviles. Características.

El desarrollo para la telefonía móvil es un campo que se encuentra en plena expansión. El número de teléfonos no deja de crecer y las necesidades de programación de los mismos tampoco. En ese sentido existen diversas plataformas de desarrollo a tener en cuenta al empezar a programar.

- Android: es el más numeroso de lejos. La mayor parte del mercado usa esta plataforma. El hecho de que Google ofrezca *completamente gratis* el sistema operativo para los fabricantes y el entorno para los programadores lo ha hecho crecer hasta desbancar a su competidor. Google solo se ocupa de la venta de apps y ese es su nicho de beneficios.
- iOS: Utiliza una filosofía completamente distinta que es controlar todo el proceso desde el desarrollo hasta la distribución de aplicaciones. Ese control se hace por medio del pago de licencias y de la unicidad de la distribución.

- Windows Phone: es el sistema de Microsoft que ha lanzado más o menos recientemente y que aún está por ver si consigue una cuota de mercado significativa o no.
- FirefoxOS: se centra principalmente en mercados emergentes (con el objetivo a largo plazo de ganar cuota de mercado) su filosofía es la misma del software libre.
- Bada: Es la plataforma de Samsung creada exclusivamente para sus teléfonos. Es muy poco usada fuera de Corea del Sur.
- Symbian: surge de un antiguo sistema creado para las PDA, está prácticamente descatalogado en el desarrollo para telefonía móvil.
- Tizen: una plataforma relativamente nueva. Al igual que Android es gratis y además tiene el respaldo de Intel.
- Jolla: surge en los países nórdicos con una filosofía similar al software libre pero con la salvedad de que, de momento, solo se ejecuta en sus teléfonos (que son de gama alta)

En cuanto a la tecnología hay diferencias sustanciales entre ellas:

- Android: pensado principalmente para ser programado en Java (aunque se puede llegar a usar C++ con un kit aparte).
- iOS: usa Objective-C que lo separa mucho del resto de plataformas. El entorno exige el pago de una licencia, el SO exige una licencia y el poner aplicaciones a la venta exige otra.
- Windows Phone: usa Visual Studio que es una herramienta muy potente y usa la plataforma .NET.
- FirefoxOS: usa HTML y Javascript.
- Symbian y otros también permiten el uso de HTML y JS.
- Tizen permite dos opciones: C++ o HTML/JS
- Jolla/Sailfish: usa C++.
- Bada usa C++.

En este curso se usara Android con Java como lenguaje de desarrollo.

1.3 Limitaciones

Programar un teléfono móvil implica preparar nuestro programa para situaciones que no son de importancia en los ordenadores de escritorio o que incluso no existen.

- Desconexión: un teléfono puede perder el fluido eléctrico sin aviso o perder la conexión de red de forma repentina.
- Seguridad: un teléfono puede ser accesible desde cualquier punto del planeta lo que puede poner en grave riesgo la privacidad del usuario.
- Memoria: la cantidad de memoria de estos dispositivos es mucho más reducida que los equipos de escritorio.

- Consumo batería: la cantidad de código que se ejecuta implica disminuir la cantidad de batería del usuario.
- Almacenamiento: la cantidad de espacio para almacenar ficheros en estos dispositivos es muy variable y a veces prácticamente inexistente.

1.4 Entornos integrados de trabajo.

En este apartado vamos a hablar de los distintos entornos que se pueden usar para programar teléfonos móviles.

- Eclipse.
- XCode para iOS.
- Android Studio.
- NetBeans.
- Visual Studio (para Windows Phone).
- ¿Visual Kaffe?
- Línea de comandos.
- Appcelerator.

1.5 Android Studio

Android Studio es el entorno oficial de programación de Google. Como tal, es probablemente la herramienta del futuro si bien tiene diversas desventajas.

- Aún está en fase beta y según los documentos de instalación aún puede cambiar mucho sin previo aviso.
- Requiere una máquina más potente que los otros: de no ser así el proceso de edición-compilación-ejecución se vuelve demasiado lento.

1.6 Eclipse

Eclipse ha sido desde los comienzos la herramienta ofrecida por Google para programar en Android. De hecho, en su página se ofrece el "Android Developer Tools Bundle" que contiene absolutamente todo lo necesario para trabajar.

Truco: Aún así, despues de descargarlo, aún tendremos que bajar e instalar la(s) plataforma(s) Android para las cuales queramos programar, lo que en sitios con una línea de baja velocidad aún requerirá un tiempo apreciable.

En este manual será la herramienta que se utilizará para los ejemplos.

1.7 La línea de comandos

La línea de comandos es el entorno más ligero. Además ofrece grandes ventajas en cuanto a la automatización de tareas, y de hecho Google ofrece el kit de desarrollo adaptado a la línea de comandos. El inconveniente principal es que algunos desarrolladores no están muy acostumbrados a ella.

1.8 El primer proyecto

Cuando se instala el Android Developer Bundle y se arranca Eclipse podremos utilizar un pequeño asistente para crear la primera aplicación. Para ello, en el menú `File-New` elegiremos la opción `Android Application Project`, mostrándonos una ventana que debería ser parecida a la siguiente figura.

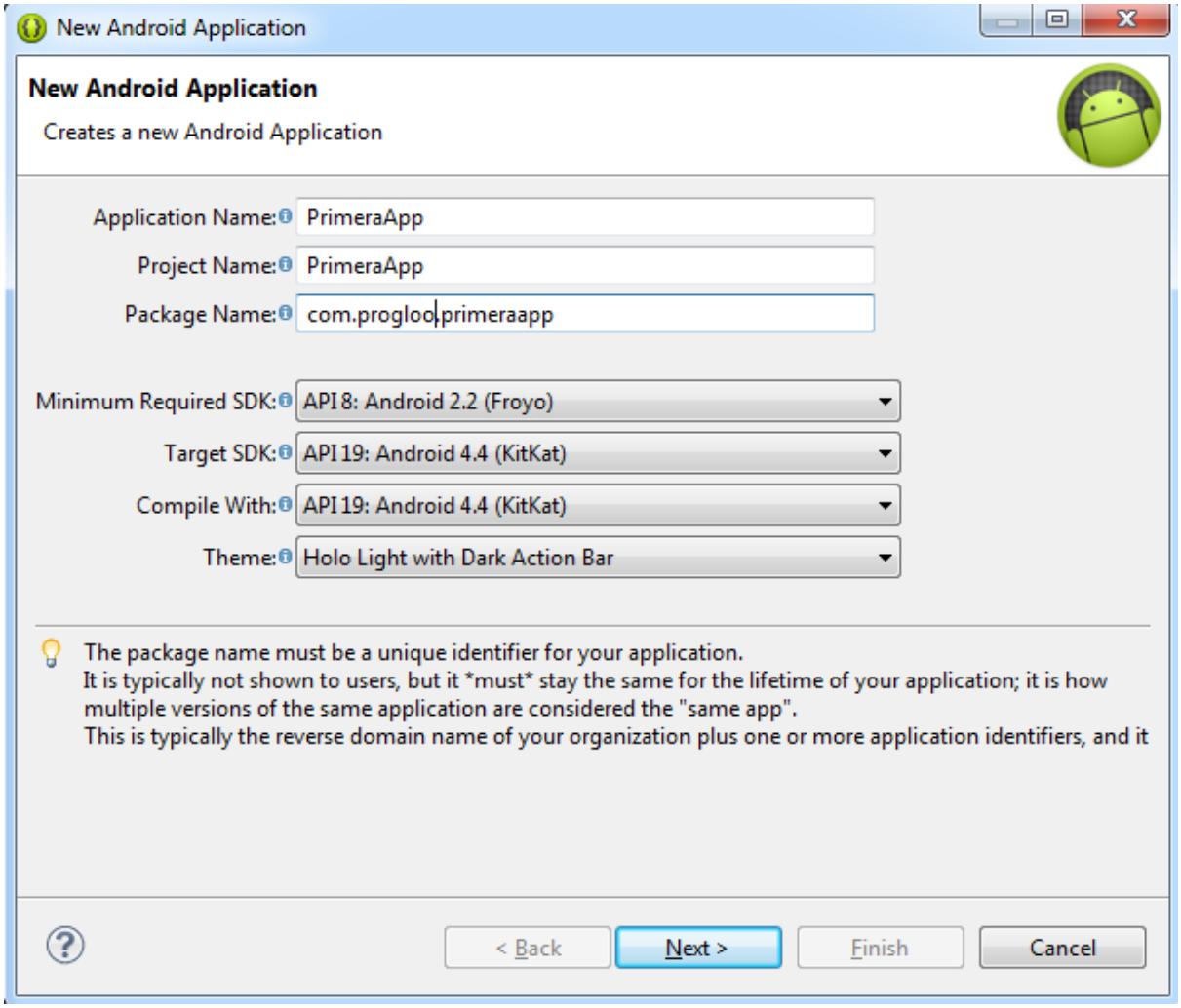


Figura 1.1: Datos iniciales de la aplicación Android

En ella deberemos prestar atención a los siguientes elementos:

- *Minimum required SDK* : es la versión de Android mínima que necesitará en su móvil/tablet quien desee instalar la aplicación. Si se tiene la tentación de poner la versión 1.0 se debe tener en cuenta que también se dispondrán de menos clases y métodos para construir la app. La versión 8 (Android 2.1) es un valor razonable a día de hoy.
- *Target SDK* : es la versión de Android para la cual hemos optimizado la aplicación. En todo este manual se usará la versión 19 de Android (o Android 4.4)
- *Compile with* : Android tiene varias versiones y podemos utilizar una versión posterior para optimizar una aplicación orientada a un Android más antiguo. Sin embargo, normalmente no lo haremos y usaremos la misma versión que en el Target SDK, es decir, la 19.
- *Theme* : las aplicaciones pueden tener diversos temas o "skins". Google ofrece algunos estilos predeterminados, pero no haremos especial hincapie en el diseño, solo en la programación. Usaremos el estilo por defecto "Holo Light".

Despues de haber rellenado estos datos podremos ver algo como esto:

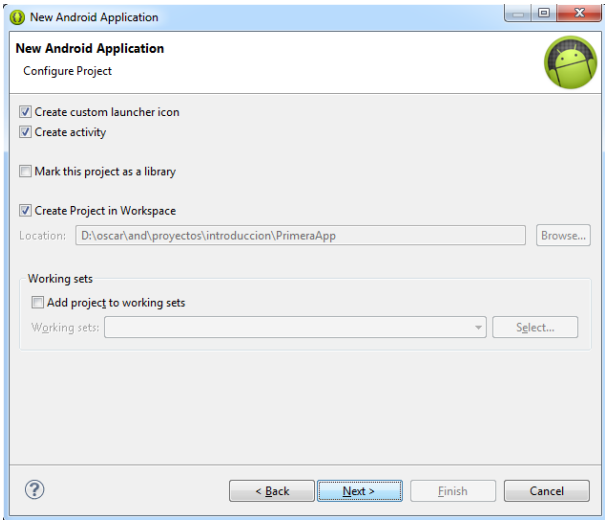


Figura 1.2: Opciones específicas del proyecto

Aquí podremos indicar si queremos crear una biblioteca en lugar de una aplicación, si deseamos que se cree una actividad en blanco y si queremos ponerlo en el directorio de trabajo predeterminado. Se dejarán las opciones por defecto.

1.9 Descargando plataformas

Una vez hecho esto se debería instalar alguna versión del kit de desarrollo Android para empezar a programar. Para ello, se debe arrancar el gestor de plataformas Android mediante el menú de Eclipse `Window->SDK Manager`

El SDK Manager hace unas cuantas recomendaciones bastante prácticas: normalmente intentará instalar la última versión de Android más algunas herramientas útiles.

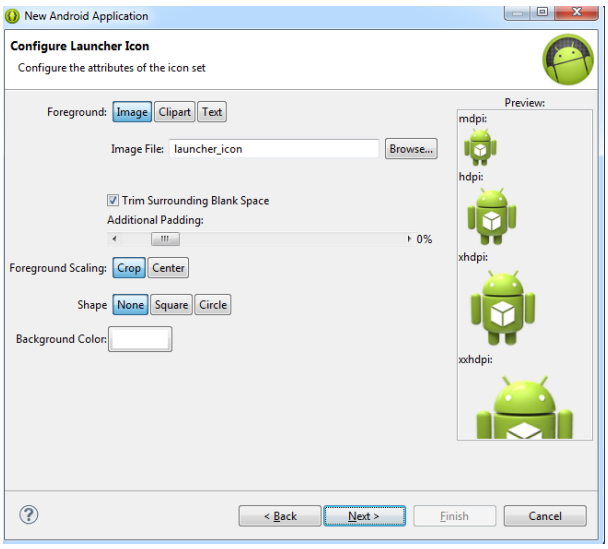


Figura 1.3: Personalizando el icono

Esta ventana permite elegir algunas opciones sobre el icono de la aplicación:

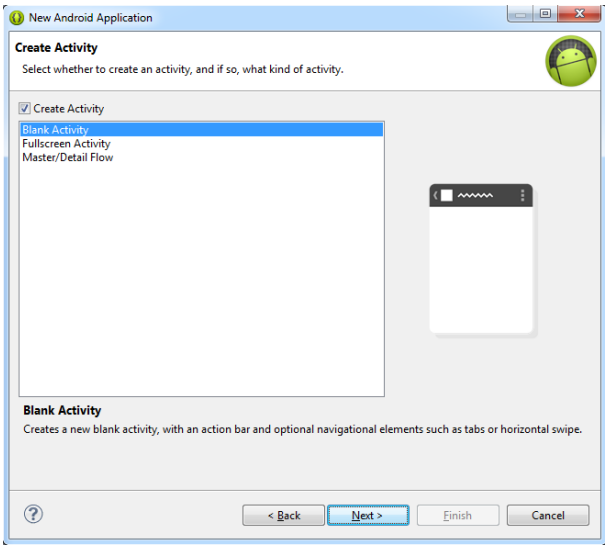


Figura 1.4: Tipo de actividad

Aquí se puede elegir qué tipo de actividad se desea. En general, usaremos una actividad en blanco.

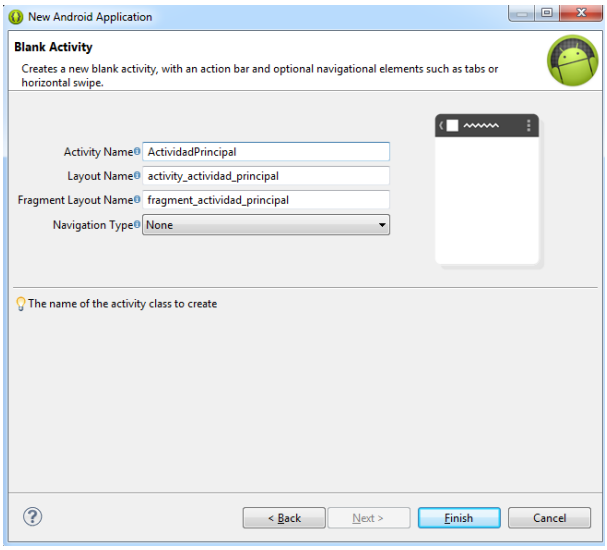


Figura 1.5: Datos de la actividad

En este último paso se indicará el nombre de la clase que contendrá la actividad principal de la aplicación. Usaremos el nombre `ActividadPrincipal` y terminaremos el asistente.

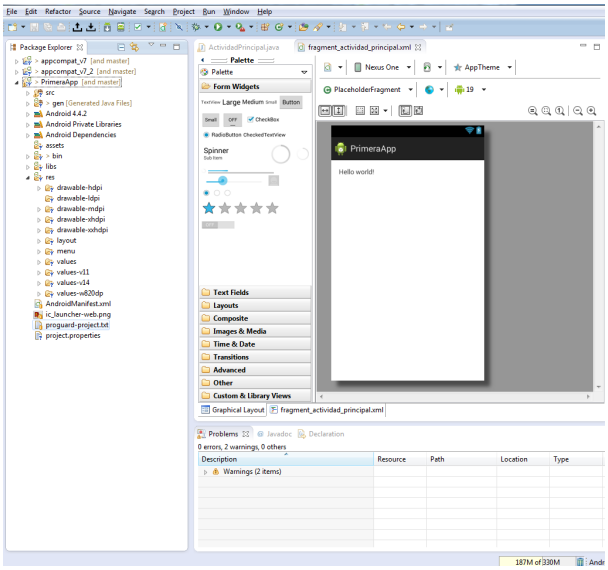


Figura 1.6: Un proyecto vacío de Android

El asistente terminará y se nos mostrará el entorno de Eclipse.

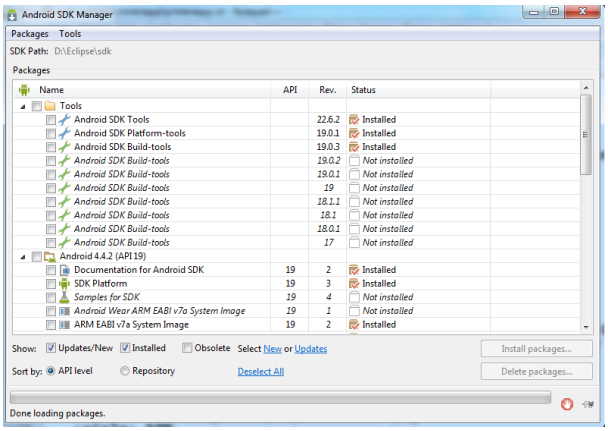


Figura 1.7: Administrador de plataformas Android

Advertencia: Una de las herramientas que se descargará es *Intel x86 Emulator Accelerator* o HAXM. Esta herramienta de Intel permite acelerar la ejecución del emulador de Android en microprocesadores Intel que tengan activada en su BIOS la opción de aceleración. Puede ser necesario habilitar esta opción en la BIOS (probablemente en alguna opción con el nombre `Enable Intel VT-x` o similar). El uso de HAXM es **MUY RECOMENDABLE**. Por otro lado, el SDK Manager descarga, pero no descomprime ni instala HAXM. Se debe buscar el ZIP en el directorio de instalación y ejecutarlo.

En líneas generales se necesitarán:

- Todos los archivos de la última plataforma
- El driver USB, que permitirá ejecutar nuestros programas en un móvil/tablet conectado por USB al equipo
- El driver HAXM
- La biblioteca de soporte de Android: permite que programas con una versión moderna se ejecuten en algunas plataformas más antiguas, entre otras cosas.
- Las *build-tools* o herramientas de compilación.
- Las *platform-tools* o herramientas específicas de la plataforma.
- Las *Android tools*, herramientas específicas de Android

1.10 Creando emuladores

Cuando se haya completado el paso anterior, se podrán crear *Android Virtual Devices* o AVDs o emuladores. Se pueden crear dispositivos con diferentes características como se muestra a continuación.

En primer lugar, se debe elegir la opción `Window-Android Virtual Device Manager`, con lo que ser verá una herramienta que permite crear emuladores.

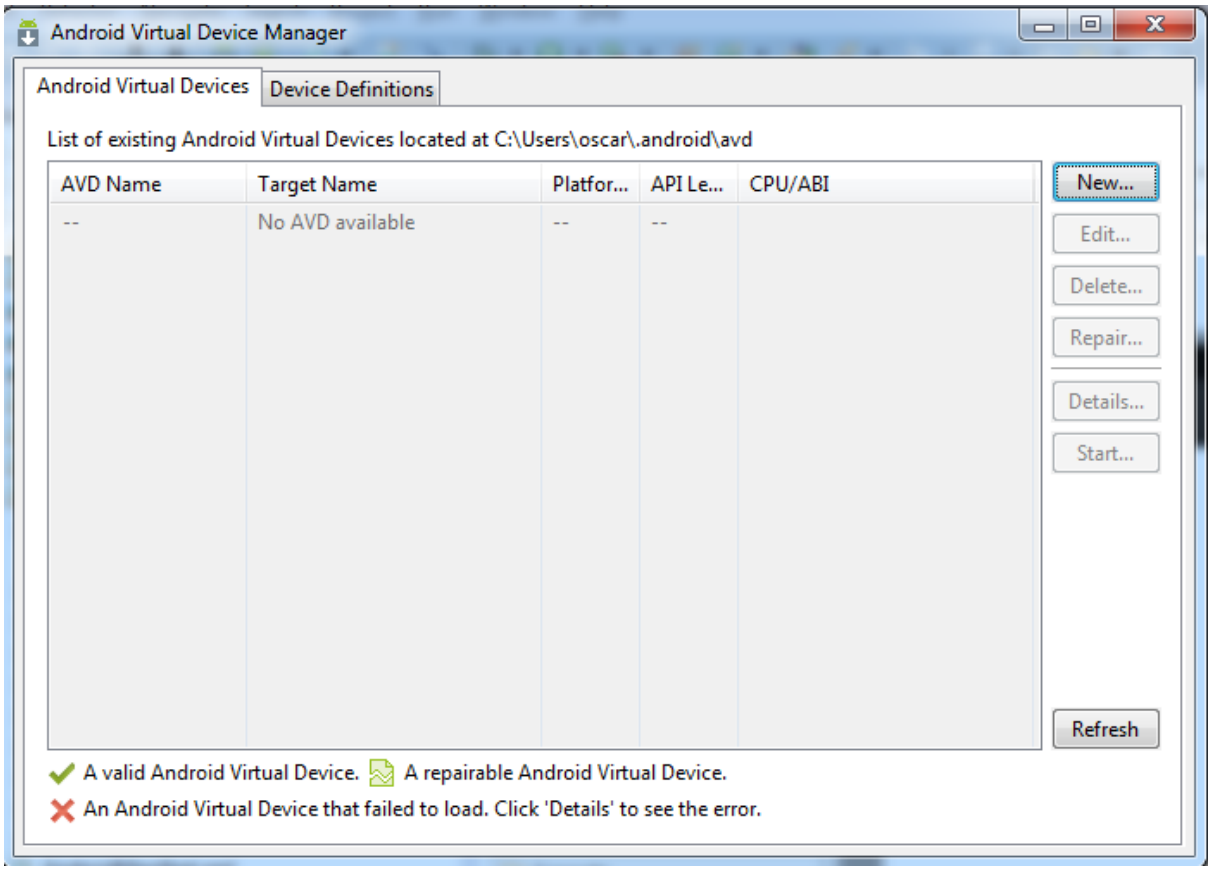


Figura 1.8: El Android Virtual Device Manager

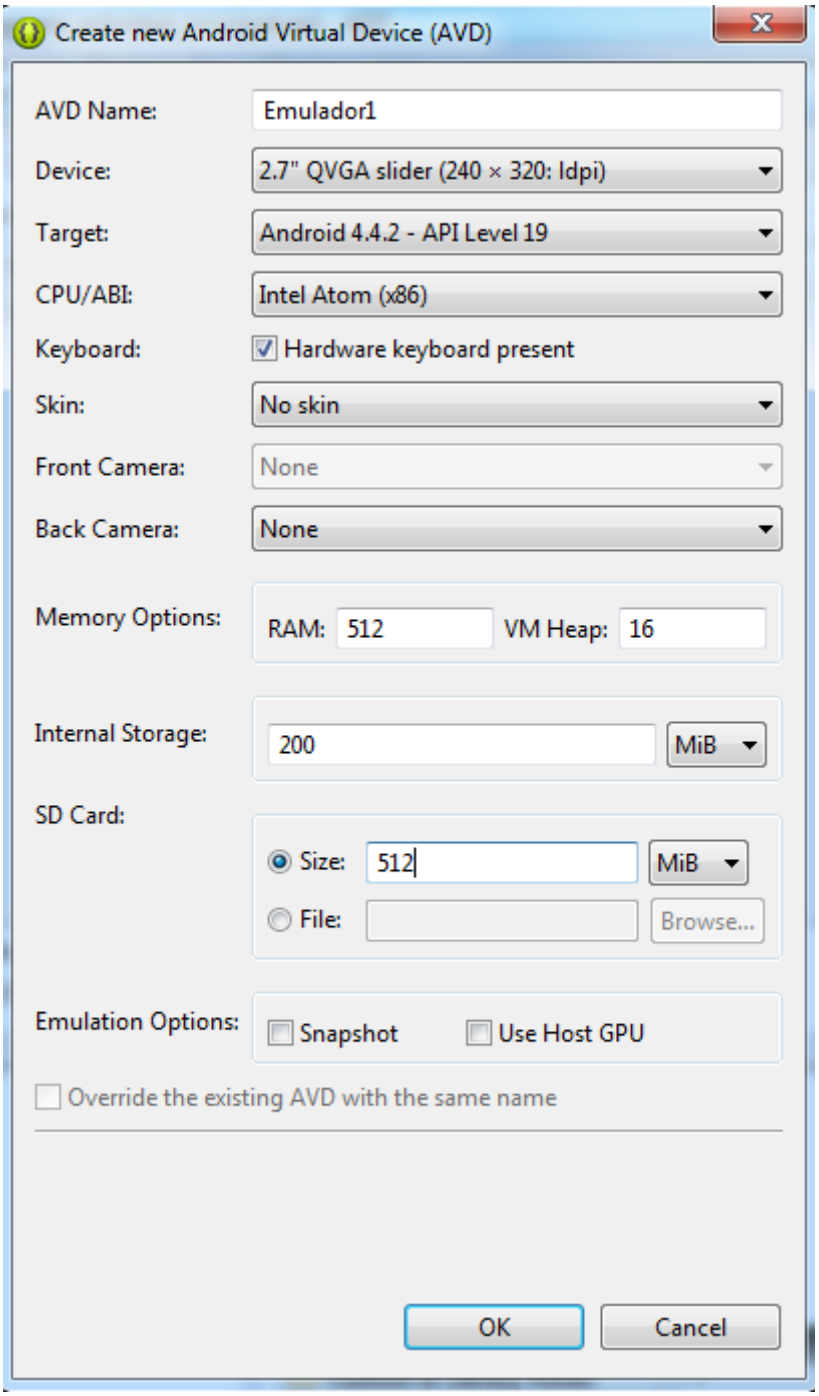


Figura 1.9: Creando dispositivos

Una vez arrancado se podrá crear un nuevo dispositivo con el botón *New*. Se recomienda mantener estas opciones.

1.11 Arrancando el programa

Una vez que se tiene el emulador creado, se puede arrancar con el botón Start, y despues arrancar el proyecto vacío Android de Eclipse. Para ello, una posibilidad es hacer click con el botón derecho en el proyecto que vemos a la izquierda de Eclipse y elegir el menu `Run As-Android Application`. Debería arrancarse la aplicación en el emulador y ver el resultado.

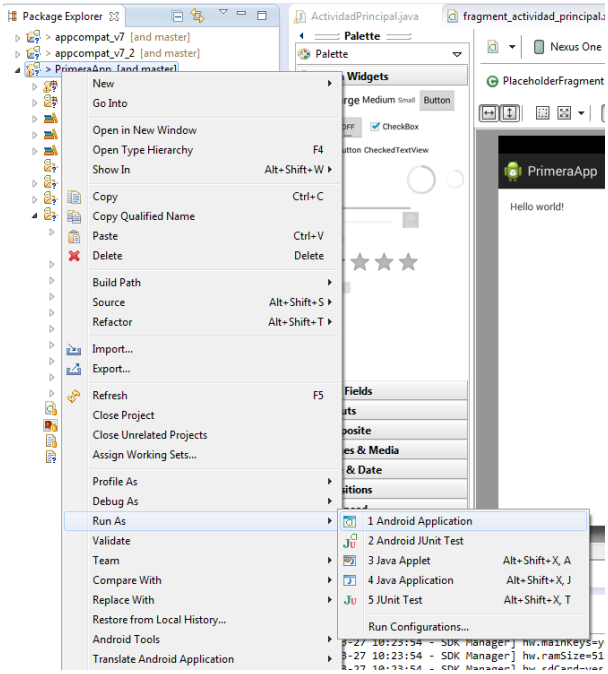


Figura 1.10: Ejecutando el primer proyecto

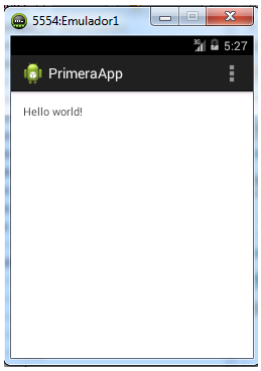


Figura 1.11: Emulador ejecutando la primera app

1.12 Módulos para el desarrollo de aplicaciones móviles.

En este curso, realmente solo necesitaremos Java para crear apps. Sin embargo, existen un montón de bibliotecas que permiten acelerar el desarrollo para diversos lenguajes y distintas tareas. Solo por nombrar algunos mencionaremos:

- Unity para desarrollar juegos.
- JQuery para Javascript.
- Bibliotecas para tareas muy específicas como la seguridad SSL y similares.

1.13 Emuladores.

A la hora de probar un app suele ser posible utilizar un emulador cargado en el sistema operativo que facilite la tarea de depurar la aplicación.

En Android, Google proporciona un sistema completo de emulación basado en máquinas virtuales (no usa VirtualBox sino un programa similar llamado QEMU).

El sistema de emulación permite crear dispositivos de características muy variadas para probar nuestra app en distintos entornos. Google denomina a estos dispositivos Android Virtual Devices (o AVDs)

- Se puede modificar el tamaño y la resolución.
- La memoria RAM y espacio en tarjeta SD.
- Se puede poner o quitar cámara.
- Existen dispositivos predefinidos por Google que permiten crear emuladores muy rápidamente.
- También se pueden clonar dispositivos para hacer solo una modificación de forma rápida.
- Una característica de interés es que *si se dispone de una tarjeta gráfica con aceleración* se puede activar una casilla llamada "Host GPU" que permite acelerar la emulación.
- Se puede obligar al emulador a que "recuerde" el estado en que se quedó para así continuar donde nos hubiésemos quedado el último día. Esta opción se llama instantánea o *snapshot*.

Si el equipo de escritorio es un Intel se puede instalar el Hardware Accelerated eXecution Manager o HAXM que permite acelerar la emulación. En el directorio `sdk/extras/intel` se puede encontrar un archivo ZIP que contiene un EXE que instala el HAXM. Se recomienda encarecidamente instalarlo en casa y, si es necesario, habilitar la tecnología VT en la BIOS.

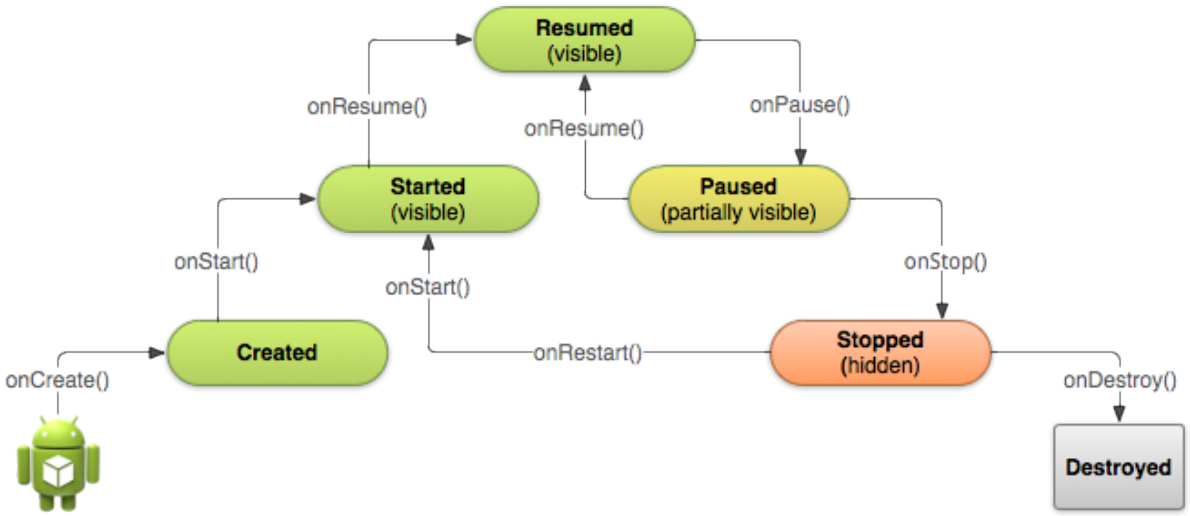


Figura 1.12: Pasos en la ejecución de una app (imagen tomada de Google).

1.14 Ciclo de vida

1.15 Configuraciones y perfiles

1.16 Tamaños y densidades

Como ya se ha mencionado, la plataforma Android establece diversas categorías de dispositivo en función del tamaño y la densidad/resolución:

- En tamaños se distingue entre *small* , *normal*, *large* y *xlarge*.
- En densidades se distingue entre *ldpi*, *mdpi*, *hdpi* y *xhdpi*.

Truco: Un cambio en la orientación del dispositivo **también se considera un cambio en el tamaño del dispositivo**.

	Low density (120), *ldpi*	Medium density (160), *mdpi*	High density (240), *hdpi*	Extra high density (320), *xhdpi*
Small screen	**QVGA (240x320)**		480x640	
Normal screen	**WQVGA400 (240x400)** **WQVGA432 (240x432)**	**HVGA (320x480)**	**WVGA800 (480x800)** **WVGA854 (480x854)** 600x1024	640x960
Large screen	**WVGA800** (480x800)** **WVGA854** (480x854)**	**WVGA800* (480x800)** **WVGA854* (480x854)** 600x1024		
Extra Large screen	1024x600	**WXGA (1280x800)**† 1024x768 1280x768	1536x1152 1920x1152 1920x1200	2048x1536 2560x1536 2560x1600

Figura 1.13: Tamaños de pantalla (imagen tomada de Google)

1.17 Directorios

Para que nuestra aplicación ofrezca soporte a todas estas variantes tan solo se deben utilizar distintos directorios `layout` dentro del subdirectorio `res`. Así, si queremos crear una configuración de interfaz diferente para pantallas grandes podemos crear un subdirectorio `res/layout-large` que contenga un interfaz diferente optimizada para pantallas grandes.

Como puede verse, la clave consiste en utilizar directorios `layout-xxx` donde *xxx* pueden ser una serie de sufijos.

- `res/layout` es el directorio que se usará para el interfaz por defecto que asume orientación vertical.
- `res/layout-large` para pantallas grandes.
- `res/layout-xlarge` para pantallas muy grandes.
- `res/layout-large-land` para pantallas muy grandes giradas para estar en horizontal (landscape).

1.18 Imágenes

Cuando se tiene la previsión de que la aplicación se va a ejecutar en muchos dispositivos diferentes se deben crear diferentes versiones de las imágenes usadas.

Lo ideal es disponer de las imágenes en formato vectorial y utilizar las siguientes escalas:

- Para resoluciones *mdpi*, la imagen a escala 1.
- Para *ldpi*, la imagen a escala 0.75
- Para *hdpi*, se escala a 1.5
- Para *xhdpi*, la escala debe ser 2.
- Y así sucesivamente.

Normalmente ya no es necesario poner nada para *ldpi* por dos motivos.

1. En la actualidad suponen un porcentaje muy pequeño de los dispositivos.
2. Android puede hacer la escala automáticamente.

Aunque en este manual se habla en general de Android 4 conviene no perder de vista las plataformas anteriores. Google mantiene una pequeña tabla con los porcentajes de uso de las diversas versiones de Android ya que crear nuestra aplicación *exclusivamente para cierta versión y las posteriores* hará que nos autoexcluyamos de una porción del mercado que puede ser muy significativa.

Version	Codename	API	Distribution
2.2	Froyo	8	1.1%
2.3.3 - 2.3.7	Gingerbread	10	17.8%
3.2	Honeycomb	13	0.1%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	14.3%
4.1.x	Jelly Bean	16	34.4%
4.2.x		17	18.1%
4.3		18	8.9%
4.4	KitKat	19	5.3%

Data collected during a 7-day period ending on April 1, 2014.
Any versions with less than 0.1% distribution are not shown.

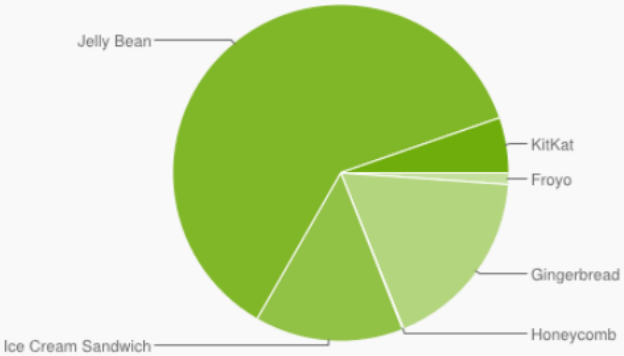


Figura 1.14: Porcentajes de uso de Android (tomada de Google el 4-4-2014)

1.19 Ejercicios

1. Crea una aplicación que se vea de tres formas distintas en función de que la pantalla sea normal, grande o muy grande
2. Haz que la aplicación anterior muestre datos sobre la plataforma sobre la que se está ejecutando. (Pista, deberás implementar *forzosamente* un método `protected void onStart()`)

Para resolver estos ejercicios necesitarás leer los apartados siguientes sobre directorios y recursos.

1.20 Solución Ejercicio 1

1.20.1 Enunciado

Crea una aplicación que se vea de tres formas distintas en función de que la pantalla sea normal, grande o muy grande .

Para poder ver los resultados deberemos tener en primer lugar tres emuladores, que tengan, los tamaños que necesitamos.

1.20.2 Implementación

Una vez creados los AVDs crearemos el proyecto, en el cual Eclipse nos creará automáticamente el directorio `res/layout`. Crearemos dos directorios más:

	Low density (120), *ldpi*	Medium density (160), *mdpi*	High density (240), *hdpi*	Extra high density (320), *xhdpi*
Small screen	**QVGA (240x320)**		480x640	
Normal screen	**WQVGA400 (240x400)** **WQVGA432 (240x432)**	**HVGA (320x480)**	**WVGA800 (480x800)** **WVGA854 (480x854)** 600x1024	640x960
Large screen	**WVGA800** (480x800)** **WVGA854** (480x854)**	**WVGA800* (480x800)** **WVGA854* (480x854)** 600x1024		
Extra Large screen	1024x600	**WXGA (1280x800)**† 1024x768 1280x768	1536x1152 1920x1152 1920x1200	2048x1536 2560x1536 2560x1600

Figura 1.15: Tamaños de pantalla reconocidos (imagen tomada de Google)

1. `res/layout-large` donde pondremos los ficheros con la interfaz definido para pantallas grandes.
2. `res/layout-xlarge` para pantallas muy grandes. Si no se tiene memoria suficiente es posible que un emulador con estas características tarde mucho en arrancar e incluso que sufra "cuelgues".

Crearemos un interfaz cualquiera como por ejemplo este:

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
        xmlns:tools="http://schemas.android.com/tools"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:paddingBottom="@dimen/activity_vertical_margin"
        android:paddingLeft="@dimen/activity_horizontal_margin"
        android:paddingRight="@dimen/activity_horizontal_margin"
        android:paddingTop="@dimen/activity_vertical_margin"
        tools:context="com.example.adapptable.ActividadPrincipal$PlaceholderFrag

        <TextView
                android:id="@+id/textView1"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:layout_alignParentBottom="true"
                android:layout_alignParentRight="true"
                android:text="@string/textoPlataformas" />

</RelativeLayout>
```

Como se puede apreciar, solo contiene un cuadro de texto que debe aparecer **en la esquina inferior derecha** del dispositivo.

Una vez hecho esto, pondremos en los otros directorios alguna variación de este fichero, como por ejemplo estas dos:

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

```
        xmlns:tools="http://schemas.android.com/tools"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:paddingBottom="@dimen/activity_vertical_margin"
        android:paddingLeft="@dimen/activity_horizontal_margin"
        android:paddingRight="@dimen/activity_horizontal_margin"
        android:paddingTop="@dimen/activity_vertical_margin"
        tools:context=
        "com.example.adapptable.ActividadPrincipal$PlaceholderFragment" >

        <TextView
                android:id="@+id/textView1"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:layout_centerHorizontal="true"
                android:layout_centerVertical="true"
                android:text="@string/textoPlataformas" />
        <!--El cuadro debe aparecer
        en el centro de la pantalla-->

</RelativeLayout>

<RelativeLayout xmlns:android=
        "http://schemas.android.com/apk/res/android"
        xmlns:tools="http://schemas.android.com/tools"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:paddingBottom="@dimen/activity_vertical_margin"
        android:paddingLeft="@dimen/activity_horizontal_margin"
        android:paddingRight="@dimen/activity_horizontal_margin"
        android:paddingTop="@dimen/activity_vertical_margin"
        tools:context=
        "com.example.adapptable.ActividadPrincipal$PlaceholderFragment" >

        <TextView
                android:id="@+id/textView1"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:layout_alignParentTop="true"
                android:layout_centerHorizontal="true"
                android:text="@string/textoPlataformas" />
        <!--El cuadro debe aparecer
        centrado en la parte superior-->
</RelativeLayout>
```

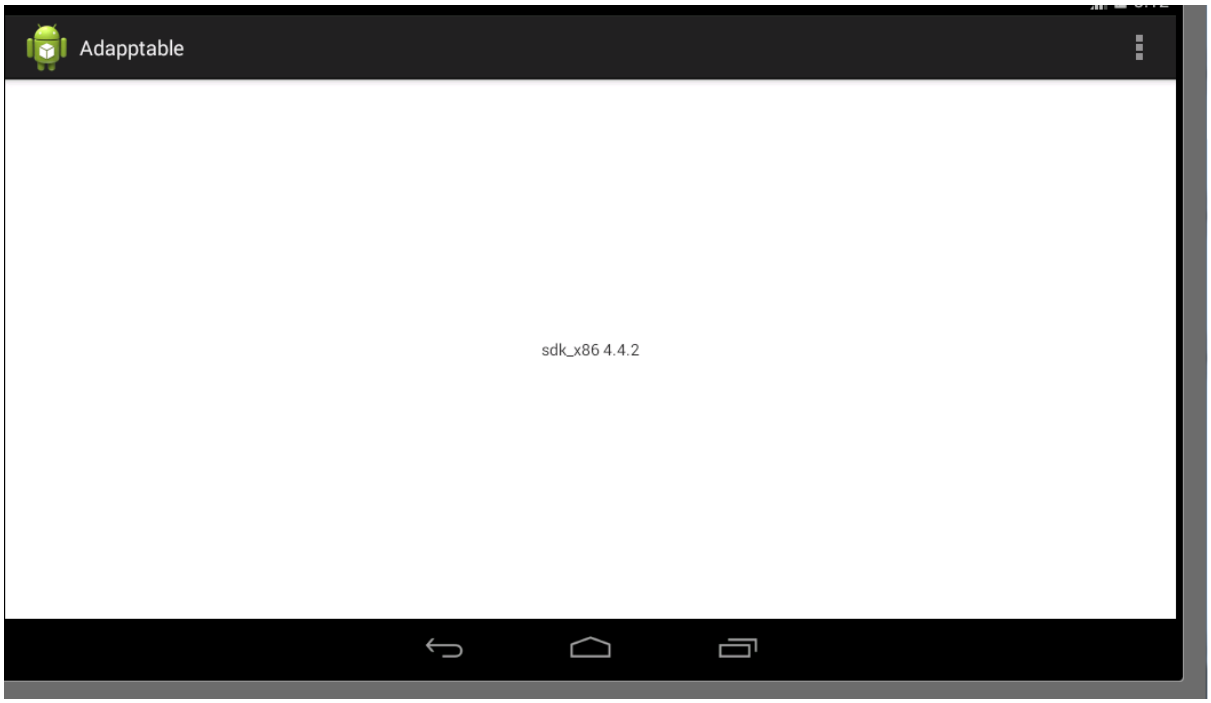


Figura 1.16: Vista de la aplicación en una tablet.

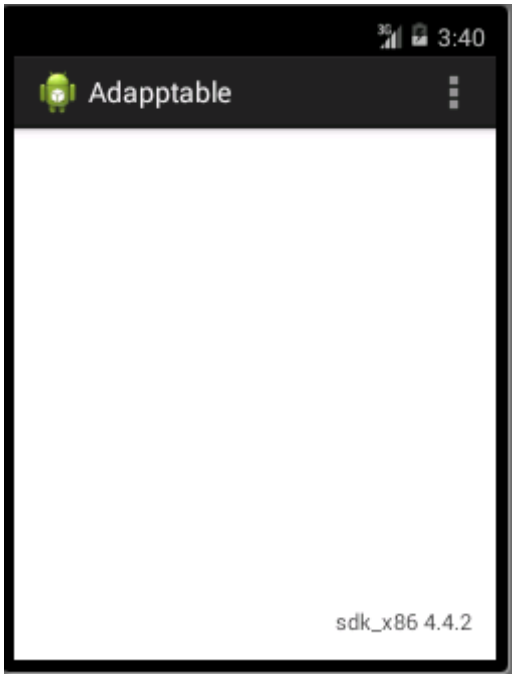


Figura 1.17: Vista de la aplicación en un móvil.

1.21 Directorios

Como ya se ha introducido anteriormente ciertos elementos que pueden cambiar no deberían estar dentro del código, sino en *recursos* (es decir, en ficheros externos que puedan cargarse en tiempo de ejecución). Un ejemplo muy elemental son las cadenas: si queremos ofrecer soporte a varios idiomas, es mejor tener todas las cadenas en un fichero, de forma que si queremos traducir la aplicación, bastará con traducir dicho fichero y hacer que la aplicación cargue distintos ficheros en función el idioma.

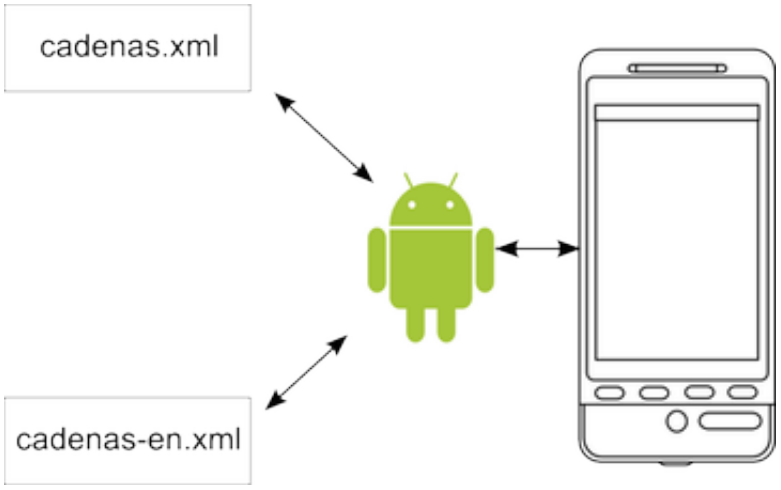


Figura 1.18: Un mismo código fuente, distintas cadenas (Imagen de shokunin).

Veamos un ejemplo muy simple. Supongamos que la aplicación saluda al usuario en el momento del arranque. Podríamos usar este código Java.

```
public class Actividad{
        private final String saludo="Hola";
}
```

Sin embargo, al hacerlo así, la traducción de la aplicación se vuelve muy compleja. Sin embargo, podemos almacenar las cadenas en un fichero de recursos como `strings.xml` de esta forma.

```
<string id="saludo">Hola</string>
```

Si ahora en el código Java cargamos la cadena (en pseudocódigo)...

```
public void saludar(){
        String saludo=R.string.saludo;
}
```

...ahora la traducción es muy sencilla, ya que basta con tener otro fichero en la aplicación con las cadenas en inglés:

```
<string id="saludo">Hello</string>
```

Y no habrá que tocar nada del código Java. De hecho, Android compilar los recursos para que sean fácilmente accesibles desde código Java. En este capítulo se analiza como usar los

recursos.

1.22 Tipos de recursos

Todos los recursos se definen en forma de XML y *deben* ir dentro de uno de estos subdirectorios que hay dentro de `res`:

- `res/animator`: contiene los ficheros XML que especifican animaciones. Se verá más sobre animaciones más adelante.
- `res/anim`: aquí se pondrán unos tipos especiales de animación llamadas "tween animation" que permiten a Android generar la animación a partir de información tal como "punto inicial", "punto final" y "duración de la animación".
- `res/color`: define los colores de nuestra aplicación.
- `res/drawable`: para especificar los archivos de imagen usados (en formatos .png, .9.png, .jpg y .gif)
- `res/layout`: para indicar la colocación de recursos en pantalla en los distintos tamaños de pantalla.
- `res/menu`: define los menús de aplicación.
- `res/raw`: recursos almacenados en formato binario. Pueden cargarse con `Resources.openRawResource()`
- `res/values`: ficheros XML que contienen valores simples como números, cadenas o incluso colores. Aunque en realidad aquí se pueden usar los nombres de fichero que queramos la costumbre es usar estos nombres:
  - arrays.xml: permite crear vectores de recursos.
  - colors.xml: para colores.
  - dimens.xml: para especificar tamaños.
  - strings.xml: para cadenas.
  - styles.xml: para estilos
- `res/xml`: aquí se almacena cualquier otro fichero XML que se desee. Los ficheros en este directorio pueden cargarse usando `Resources.getXML()`

Existe una última posibilidad para almacenar recursos, que es usar el directorio `assets` (no es `res/assets`) sin embargo, Android no compila dichos recursos automáticamente. Deben cargarse con la clase AssetsManager.

1.23 Indicando recursos alternativos

Por ejemplo, ya sabemos que el archivo `res/values/strings.xml` contiene las cadenas que se mostrarán por defecto. Si esas cadenas están en español y deseamos indicar que se

carguen otras cadenas para el idioma inglés se deben indicar modificadores para el directorio `values`.

- `res/values-en/strings.xml` indicaría el fichero de cadenas para el idioma inglés.
- `res/values-fr/strings.xml` para francés.
- Se puede usar cualquier código ISO 639-1 para indicar el idioma.

Los modificadores se pueden añadir a cualquier subdirectorio de los vistos antes, además se pueden poner varios a la vez pero siempre respetando este orden:

1. MCC (Mobile Country Code o código de país) y MNC (Mobile Network Code o código de red). Pueden consultarse las distintas redes y países en Wikipedia . Por ejemplo para indicar un recurso específico de un teléfono Android usado en territorio español se usaría `mcc214` y para indicar específicamente un recurso en un Android que accede desde Movistar se usaría `mcc214-mnc07`.
2. Idioma y región: se usa un código de páis ISO 639-1 que puede o no ir seguido de una "r" y un código de región. Así el modificador "en" indica idioma inglés y "fr" francés, pero "fr-rFR" indica francés de Francia y "fr-rCA" francés de Canadá. Pueden consultarse los códigos de país en Wikipedia
3. Dirección de lectura: `ldrtl` para cuando el idioma del dispositivo se lee de derecha a izquierda (right-to-left) y `ldltr` para lectura de izquierda a derecha. Obsérvese que ya podriamos indicar un fichero `res/values-mcc214-fr-rCA/strings.xml` para indicar los textos que debe usar un teléfono Android con su idioma puesto a francés (de Canadá) que sin embargo usa una red española. Sin embargo `res/values-fr-mcc214/strings.xml` estaría mal ya que aunque el idioma y el territorio son correctos los hemos puesto al revés (sería `res/values-mcc214-fr`)
4. Anchura mínima del dispositivo: se usa `swNdp` donde N es el número mínimo de puntos que debe tener la anchura de la pantalla. También puede indicarse este valor en el `AndroidManifest.xml` con el atributo `android:requiresSmallestWidthDp`. Si indicamos varios directorios, Android escogerá siempre el valor de N más pequeño y cercano a la anchura del dispositivo **independientemente de si la pantalla se gira o no**. Algunos valores típicos son:
   - sw320dp: para pantallas de 240x320 (ldpi), de 320x480 (mdpi) o de 480x800 (hdpi)
   - sw480dp: para 480x800 (mdpi)
   - sw600dp: para 600x1024 (mdpi)
5. Anchura disponible: el sufijo `wNdp` indica la anchura que la aplicación necesita **teniendo en cuenta si la pantalla se gira** (esta es la diferencia con respecto al anterior).
6. Altura disponible: el sufijo `hNdp` indica la altura que la aplicación necesita.
7. Tamaño de pantalla: pueden usarse los sufijos siguientes:
   - `small` de aproximadamente 320x426
   - `normal` aproximadamente 320x470

- `large` de unos 480x640
- `xlarge` con un tamaño de 720x960 (normalmente tablets)

8. Aspecto de la pantalla: `long` para pantallas WQVGA, WVGA, FWVGA y `notlong` para QVGA, HVGA, and VGA. No tiene nada que ver con la orientación de la pantalla.
9. Orientación de la pantalla: `port` (portrait) para cuando la pantalla está en vertical y `land` (landscape) para cuando está en horizontal.
10. Modo del interfaz de usuario:
    - `car` cuando el dispositivo está en un coche.
    - `desk` en un escritorio
    - `television`
    - `appliance` el dispositivo es una herramienta y no tiene pantalla.
11. Modo nocturno: `night` y `notnight` dependiendo de si el dispositivo está en modo nocturno o no.
12. Densidad de pixeles: (la escala entre los principales tamaños es 3:4:6:8)
    - ldpi: pantallas de baja densidad, aproximadamente 120dpi.
    - mdpi: densidad media, unos 160dpi.
    - hdpi: alta densidad, unos 240dpi.
    - xhdpi: densidad "extra-alta", unos 320dpi.
    - nodpi: Usado para recursos para los que no queremos que Android haga el escalado.
    - tvdpi: unos 213 (entre `mdpi` y `hdpi`)
13. Tipo de pantalla: `finger` para dispositivos táctiles y `notouch` para los demás.
14. Disponibilidad de teclado:
    - `keysexposed`: hay teclado hardware.
    - `keyshidden`: hay teclado hardware pero no está disponible y además *no hay teclado software*.
    - `keyssoft`: hay teclado software.
15. Método de entrada: `nokeys` cuando no hay teclado hardware, `qwerty` si hay un teclado hardware y `12key` para teclados hardware de 12 teclas.
16. Disponibilidad de teclas de navegación:
    - `nonav`: no se puede navegar con teclas.
    - `dpad`: hay un pad direccional.
    - `trackball`: hay un trackball.
    - `wheel`: hay un ratón con rueda (poco habitual).
17. Versión de la plataforma Android: `v3`, `v4`, `v9` etc...

1.24 Ejercicio

¿Como debería llamarse un directorio que contuviera recursos específicos para un teléfono en portugués que esté usándose en Francia con el operador Bouygues Telecom y que fuera un dispositivo de una resolución hdpi?

Respuesta: hay que ir nombrando el directorio con los sufijos correctos en el orden correcto. En este caso sería `values-mcc208-mnc20-pt-hdpi`.

1.25 Tamaños y densidades

En la tabla siguiente, tomada de la documentación oficial de Google pueden verse los tamaños y densidades aproximados de los distintos tipos de pantalla que podemos encontrar.

	Low density (120), *ldpi*	Medium density (160), *mdpi*	High density (240), *hdpi*	Extra high density (320), *xhdpi*
Small screen	**QVGA (240x320)**		480x640	
Normal screen	**WQVGA400 (240x400)** **WQVGA432 (240x432)**	**HVGA (320x480)**	**WVGA800 (480x800)** **WVGA854 (480x854)** 600x1024	640x960
Large screen	**WVGA800** (480x800)** **WVGA854** (480x854)**	**WVGA800* (480x800)** **WVGA854* (480x854)** 600x1024		
Extra Large screen	1024x600	**WXGA (1280x800)**† 1024x768 1280x768	1536x1152 1920x1152 1920x1200	2048x1536 2560x1536 2560x1600

Figura 1.19: Tamaños de pantalla reconocidos (imagen tomada de Google)

1.26 Accediendo a los recursos

Cuando se crea un recurso puede accederse al mismo por medio de la clase especial `R` la cual es creada por la herramienta `aapt`. Dicha herramienta toma todos los recursos y crea distintas subclases para facilitar el uso de dichos recursos. Las clases creadas son:

- R.drawable: para acceder a archivos de imagen.
- R.id: para acceder al id de un control.
- R.layout: para cargar disposiciones de controles.
- R.string: para acceder a cadenas.

Aí, por ejemplo si un archivo de imagen ubicado en `res/drawable/icono.png` quiere ponerse de fondo en algún control se usará la sentencia:

```
control.setBackgroundDrawableResource(
        R.drawable.icono);
```

Por otro lado, si deseamos usar un recurso XML en otro archivo XML se puede hacer usando la siguiente estructura:

1. Empezar siempre por `@`
2. Si se desea acceder a un recurso en otro paquete poner el nombre seguido de `:`, como `com.ejemplo:`.
3. Despues se indica el tipo de recurso, `string`, `drawable`...
4. Despues se indica el nombre del recurso.

Supongamos que tenemos un fichero genérico con distintas definiciones de recursos como este:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
   <color name="color_corporativo">#f00</color>
   <string name="saludo">Hola</string>
</resources>
```

Y que deseamos usar este color y este texto en interfaz. El XML sería así:

```
<?xml version="1.0" encoding="utf-8"?>
<EditText
        xmlns:android=
          "http://schemas.android.com/apk/res/android"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:textColor="@color/color_corporativo"
        android:text="@string/saludo" />
```

Por último mencionar que Android dispone de muchos otros recursos a los cuales se puede acceder usando el prefijo `android`. Así, por ejemplo, Android define un interfaz para elementos en una lista que podemos usar con este código (obsérvese que Android llama a este interfaz `simple_list_item_1`):

```
setListAdapter(
        new ArrayAdapter<String>(
                this,
                android.R.layout.simple_list_item_1,
                vector
        )
);
```

1.27 Gestión de cambios durante la ejecución

Android puede decidir reiniciar nuestra app por diversos motivos:

- El usuario ha cambiado el idioma.

- La pantalla ha rotado por ejemplo de vertical a horizontal.
- Se ha conectado un teclado
- Etc...

En estos casos, Android llamará a nuestro método `onDestroy` y despues a `onCreate`. Sin embargo, un usuario podría llegar a perder trabajo, por lo cual una actividad puede usar si lo desea dos métodos que Android llamará.

- `onSaveInstanceState`: podemos implementarlo para guardar el trabajo que haya hecho.
- `onRestoreInstanceState`: Android puede usarlo para restaurar el estado.

Sin embargo, ¿qué ocurre si esto implica grabar y cargar grandes cantidades de datos?: podría ocurrir que la aplicación se ralentizara, dando una pobre experiencia de usuario. En este caso hay dos opciones:

1. Retener un objeto en memoria durante la reinicialización.
2. Gestionar el cambio por nosotros mismos tomando el control de Android.

En cualquier caso, Android suele recordar automáticamente los elementos de la interfaz de usuario, por lo que lo que llamamos "reiniciar" la actividad en realidad no es un reinicio absoluto.

1.28 Reteniendo objetos en memoria

El método `onCreate` de una actividad siempre acepta un objeto `Bundle` en el que puede estar el estado anterior de nuestra actividad. Sin embargo, este objeto no está diseñado para almacenar grandes cantidades de datos.

Por todo ello, la clase Fragment permite ejecutar `setRetainInstance(true)` en el método `onCreate` y así evitar la destrucción y re-creación de la actividad.

1.29 Gestionando el cambio

Se puede implementar el método ‘‘ onConfigurationChanged()‘‘ para gestonar los cambios por nosotros mismos.

Peligro: Implementar este método debería ser *la última opción* ya que tendremos que reaplicar todos los cambios por nosotros mismos (cargar cadenas, interfaces, etc...)

1.30 Ejercicios

1. ¿Qué diferencia hay entre `sw320dp` y `w320dp`?.

2. Si una imagen mide 30x30 en el tamaño `ldpi`, ¿cuanto medirá en `mdpi`, `hdpi` y `xhdpi`?.

1.31 Solución ejercicio

1.31.1 Enunciado

Si una imagen mide 30x30 en el tamaño ‘‘ldpi‘‘, ¿cuanto medirá en ‘‘mdpi‘‘, ‘‘hdpi‘‘ y ‘‘xhdpi‘‘?.

1.31.2 Solución

Dado que la escala es 3:4:6:8 el tamaño será:

- 40x40 en `mdpi`.
- 60x60 en `hdpi`.
- 80x80 en `xhdpi`.
- Como plus, en las televisiones, el escalado es 1.33*mdpi por lo que una televisón el tamaño sería 54.2x54.2 (40*1.33)

1.32 Solución ejercicio

1.32.1 Enunciado

¿Qué diferencia hay entre ‘‘sw480dp‘‘ y ‘‘w480dp‘‘?.

1.32.2 Solución

La diferencia es que el primero exige que el dispositivo tenga una anchura mínima de 480 puntos. En el caso de un dispositivo de 320 de alto por 480 de ancho, la aplicación funcionará. Si el dispositivo se gira, con lo que tendría 480 de alto por 320 de alto, seguirá funcionando, ya que quizá el programador haya preparado el interfaz para auto-adaptarse a una anchura más pequeña.

Sin embargo, `w480dp` funcionaría correctamente si el dispositivo está en una posición de 320 de alto por 480 de ancho, pero *no aceptará* el giro que deje el terminal en un tamaño de 320 de ancho por 480 de alto.

1.33 Modificación de aplicaciones existentes.

Cuando hay que modificar una aplicación existente se pueden tener dos situaciones:

1. Se dispone del código fuente: en ese caso se debería empezar por analizar el diagrama UML de la aplicación, el modelo E/R y toda la documentación de la que disponga el proyecto. Despues se debe leer el código por encima analizando primero el código de la interfaz (los ficheros XML de los directorios layout). A continuación se puede ejecutar en el simulador y analizar los logs y despues se puede empezar a ampliar/corregir la aplicación yendo método a método, e implementando las pruebas de unidad que se necesiten.
2. No se dispone del código fuente: de entrada en un proyecto de tamaño mediano puede que ya sea imposible hacer ninguna modificación. Sin embargo, Java ofrece un "descompilador" que intenta reconstruir el código fuente a partir del fichero de aplicación.

1.34 Ejercicio final

1.34.1 Enunciado

Examina la documentación de Google sobre Android y analiza la aplicación de ejemplo.

1.34.2 Solución

La aplicación de ejemplo sobre Fragments utiliza los fragmentos para construir una aplicación lectora de noticias. La idea general es tener dos fragmentos, uno donde se puede elegir el título de la noticia y otro donde se muestre la noticia completa.

Usando Fragments se pueden diseñar dos interfaces de usuario:

- En uno de ellos, adaptado a tablets, los dos fragmentos aparecen a la vez en pantalla: cuando en uno se selecciona la noticia, en la misma pantalla se ve el texto de dicha noticia.
- En el otro interfaz, adaptado a móviles, se muestra una sola actividad con solo un fragmento. Cuando se selecciona una noticia *la actividad principal de selección desaparece* y queda oculta por la actividad que contiene el fragmento que muestra noticias. Esta solución es muy razonable al no poder mostrar tantas cosas en una pantalla tan pequeña.

También merece la pena comentar el concepto de *backstack*. El *backstack* es la pila de actividades a través de la cual podemos retroceder usando el botón "Atrás" de Android. Podemos tomar el control de dicha *pila* para decidir si *apilamos* las noticias, o una noticia sustituye a otra.

1.35 Fechas del examen

Miércoles 15 de octubre 3 últimas horas (de 12'00 a 14'45)

CAPÍTULO 2

Programación de aplicaciones para dispositivos móviles

2.1 Herramientas y fases de construcción.

La principal herramienta para programar aplicaciones Android es Eclipse aunque hoy en día está empezando a ser reemplazada por Android Studio.

En primer lugar se debe hacer un análisis de los permisos que deberá necesitar nuestra aplicación.

Todos los permisos que requiera nuestra aplicación se indican en el principal fichero del proyecto: `AndroidManifest.xml`

El archivo `AndroidManifest.xml` es un archivo imprescindible en cualquier aplicación Android, debe tener siempre ese nombre y debe estar en el directorio raíz del proyecto. Este fichero sirve para lo siguiente:

- Identifica el paquete Java de la aplicación, que se usará como identificador único de la misma.
- Describe los componentes de la aplicación: actividades, servicios, etc...
- Determina qué procesos alojarán componentes de la aplicación.
- Declara los permisos que debe tener la aplicación para acceder al hardware o al software del sistema.
- Declara los permisos que otros componentes deben tener para interactuar con los componentes de nuestra aplicación.
- Identifica las clases `Instrumentation` que se usarán para monitorizar el rendimiento. Normalmente esto solo se hace mientras estamos en pruebas, después se elimina.
- Indica la versión mínima de Android que se necesita para ejecutar nuestra app.
- Indica las bibliotecas con las que enlaza nuestro programa.

Un `AndroidManifest.xml` tiene esta estructura:

```
<?xml version="1.0" encoding="utf-8"?>

<manifest>

        <uses-permission />
        <permission />
        <permission-tree />
        <permission-group />
        <instrumentation />
        <uses-sdk />
        <uses-configuration />
        <uses-feature />
        <supports-screens />
        <compatible-screens />
        <supports-gl-texture />

        <application>

                <activity>
                        <intent-filter>
                                <action />
                                <category />
                                <data />
                        </intent-filter>
                        <meta-data />
                </activity>

                <activity-alias>
                        <intent-filter> . . . </intent-filter>
                        <meta-data />
                </activity-alias>

                <service>
                        <intent-filter> . . . </intent-filter>
                        <meta-data/>
                </service>

                <receiver>
                        <intent-filter> . . . </intent-filter>
                        <meta-data />
                </receiver>

                <provider>
                        <grant-uri-permission />
                        <meta-data />
                        <path-permission />
                </provider>

                <uses-library />

        </application>
```

```
</manifest>
```

2.2 <uses-permission>

Indica que la app necesita que se le conceda un cierto permiso para poder ser instalada y ejecutada. El permiso se indica en el atributo `android:name` con un valor como `android.permission.CAMERA`.

También puede llevar un atributo `android:maxSdkVersion` con el que se indica la versión máxima de Android donde es necesario pedir el permiso. Se usa en los casos en los que un permiso deja de existir. Eclipse suele rellenar este valor con la misma versión que usamos para el desarrollo.

Un posible valor:

```
<uses-permission
  android:name="android.permission.WRITE_EXTERNAL_STORAGE"
  android:maxSdkVersion="18" />
```

Después de construir el `AndroidManifest.xml` se debería realizar un boceto de como va a ser el interfaz. Aunque se indique de esta forma, el `AndroidManifest.xml` puede volver a modificarse en el futuro. Hay aplicaciones que permiten elaborar el "wireframe" de nuestro interfaz, pero Eclipse también puede ayudar mucho en esta tarea.

2.3 Interfaces de usuario. Clases asociadas.

2.3.1 Ejercicio

Usando el diseñador de Eclipse construye un interfaz como el siguiente. Recuerda editar los ID de los controles (para poder tener en el código nombres más fáciles de recordar) y pon nuevos textos a los controles (para que la aplicación sea fácil de traducir)

La aplicación calcula una pensión de una forma muy sencilla: si se ha cotizado durante el mínimo de años exigidos por la ley, se tiene una pensión equivalente al 90 % del sueldo actual. Si no ha sido así se tiene una pensión del 75 % del sueldo actual.

Aunque es una funcionalidad que todavía no se va a implementar, la app podrá enviar un SMS con el resultado (y un anuncio de nuestra empresa) a otro número. Esto implica hacer que la aplicación exija pedir ese permiso en el `AndroidManifest.xml`.

2.3.2 Código Java

El código Java sería algo así (faltan unas líneas)

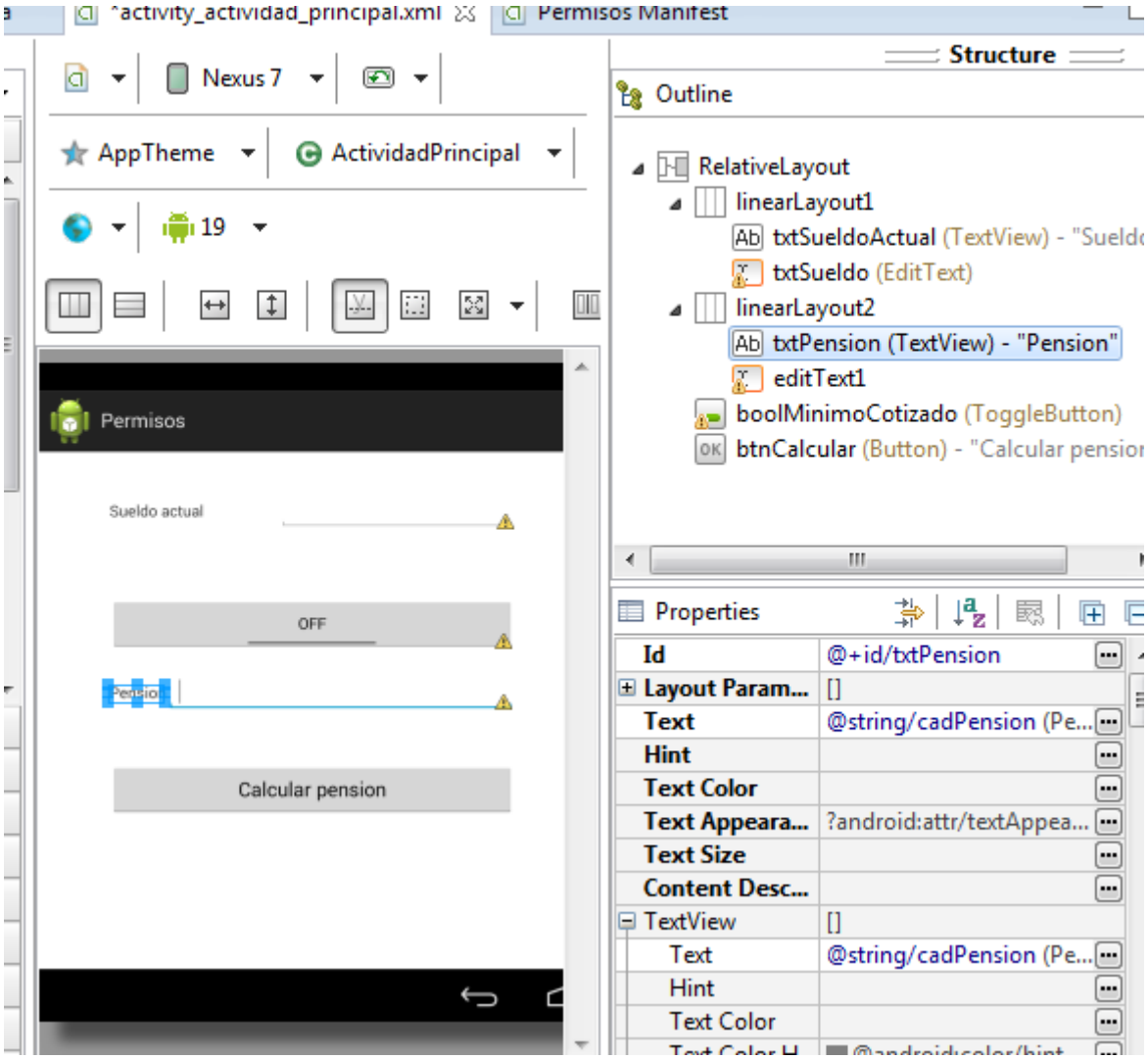


Figura 2.1: Interfaz de la aplicación

```
public void calcularPension(View controlPulsado){
                EditText control;
                control=(EditText)
                                this.findViewById(R.id.txtSueldoActual);

                Editable cadPension=control.getEditableText();
                if (cadPension.toString().equals("")) return ;
                Double sueldoActual;
                sueldoActual=
                                Double.parseDouble(cadPension.toString());

                ToggleButton togMinimo;
                togMinimo=(ToggleButton)
                                this.findViewById(R.id.togMinimo);
                Double pensionResultado;
                if (togMinimo.isChecked()){
                        pensionResultado=sueldoActual*0.9;
                }
                else {
                        pensionResultado=sueldoActual*0.75;
                }

                EditText txtPensionResultado;
                txtPensionResultado=(EditText)
                                this.findViewById(R.id.txtPensionResultado);
                txtPensionResultado.setText(pensionResultado.toString());
        }
```

2.4 Sobre el diseño de interfaces

Cuando se diseña un interfaz lo normal es ir insertando los controles en "layouts" que a su vez van dentro de otros. El objetivo es poder modificar un bloque de controles sin afectar a los demás.

Todo control Android puede manipularse de dos formas:

- Indicando su tamaño en los parámetros `width` y `height`. Se podría indicar el tamaño en puntos (mala idea porque el control no se redimensiona automáticamente) pero también se pueden indicar otras dos posibilidades:
  - `wrap_content`: significa más o menos "adáptate al mínimo posible".
  - `match_parent`: "agrándate y adáptate al tamaño de tu contenedor padre".
- Indicando qué proporción ocupa con respecto a sus controles del mismo contenedor. Esto se hace modificando el atributo `weight` y poniendo luego el `width` o el `height` a `0dp`.

2.4.1 Ejercicio

Crear una aplicación que permita al usuario practicar el cálculo mental con operaciones sencillas (sumas y restas) con números pequeños (de 1 a 99). Cuando el usuario introduce un resultado se le dice si acierta o no y se genera una nueva operación al azar.

```
package com.ies.calculus;

import java.util.Random;

import android.support.v7.app.ActionBarActivity;
import android.os.Bundle;
import android.util.Log;
import android.view.Menu;
import android.view.MenuItem;
import android.view.View;
import android.widget.EditText;
import android.widget.TextView;


public class ActividadPrincipal extends ActionBarActivity {
        int num1;
        int num2;
        String operacion;
        @Override
        protected void onCreate(Bundle savedInstanceState) {
                super.onCreate(savedInstanceState);
                setContentView(R.layout.activity_actividad_principal);
                generarOperacion();
        }


        @Override
        public boolean onCreateOptionsMenu(Menu menu) {
                // Inflate the menu; this adds items to the action bar if it is
                getMenuInflater().inflate(R.menu.actividad_principal, menu);
                return true;
        }

        @Override
        public boolean onOptionsItemSelected(MenuItem item) {
                // Handle action bar item clicks here. The action bar will
                // automatically handle clicks on the Home/Up button, so long
                // as you specify a parent activity in AndroidManifest.xml.
                int id = item.getItemId();
                if (id == R.id.action_settings) {
                        return true;
                }
                return super.onOptionsItemSelected(item);
        }
```

```
private void escribirNumeroEnTextView(
                int num, int id){
        TextView tv=(TextView)
                        this.findViewById(id);
        tv.setText(""+num);
}
private void generarOperacion(){
        Random generador=new Random();
        num1=generador.nextInt(100);
        num2=generador.nextInt(100);
        escribirNumeroEnTextView(
                        num1, R.id.tvOperando1);
        escribirNumeroEnTextView(
                        num2, R.id.tvOperando2);
        //Para generar la op. matemática
        //escogeremos un valor al azar de un vector
        String[] ops={"+", "-"};

        int posAzar=generador.nextInt(ops.length);
        operacion=ops[posAzar];
        TextView tvOperando=
                        (TextView)this.findViewById(R.id.tvOperador);
        tvOperando.setText(operacion);
}
public void comprobar(View control){
        EditText txtResultado=(EditText)
                        this.findViewById(R.id.txtResultado);
        String resultado=txtResultado.getText().toString();
        TextView tvMensajes=(TextView)
                        this.findViewById(R.id.tvMensajes);
        if (resultado.equals("")){
                tvMensajes.setText("Resultado incorrecto");
                generarOperacion();
                return ;
        }
        int resultCalculado=0;
        switch (operacion.charAt(0)){
                case '+':{
                        resultCalculado=num1+num2;
                        break;
                }
                case '-':{
                        resultCalculado=num1-num2;
                        break;
                }
        }
        int resultadoIntroducido=
                        Integer.parseInt(resultado);
        if (resultadoIntroducido==resultCalculado){
                tvMensajes.setText("¡Correcto!");
        } else {
```

```
                    tvMensajes.setText("¡MAL!");
                }
                txtResultado.setText("");
                generarOperacion();
        }
}
```

2.4.2 Resumen de los contenedores Android

En la imagen siguiente puede apreciarse la variedad de contenedores que ofrece Android:

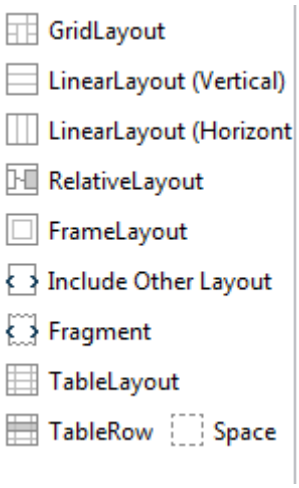


Figura 2.2: Contenedores Android

2.5 Actividades

Una actividad es un programa diseñado no solo para llamar a otros programas sino que también puede ofrecer sus servicios en Android para que otros programas les llamen a ellos.

El objetivo básico es comprender la forma de comunicar actividades en Android.

2.5.1 Actividad receptora de información

Una actividad típica debería estar preparada para recibir parámetros de una forma similar a esta:

```
public static String parametroNombre=
                "com.ies.actividades1.nombrePersona";
protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        this.setContentView(R.layout.actividad_mostrar_nombres);
```

```
        Intent intentPasado=this.getIntent();
        String nombrePasado=
                        intentPasado.getStringExtra(
                ActividadMostrarNombres.parametroNombre
        );

        TextView txtNombreAMostrar;
        txtNombreAMostrar=
                        (TextView) this.findViewById(R.id.txtNombreMostrado);
        txtNombreAMostrar.setText(nombrePasado);
}
```

2.5.2 Actividad llamadora

Una actividad que desee invocar a otro necesitará "pasar parámetros" de una forma similar a esta:

```
    public void pasarNombre(View control){
    EditText txtNombre;
    txtNombre=(EditText) findViewById(R.id.txtNombre);
    String nombre=txtNombre.getText().toString();
    Intent iMostrarNombre;
    //Indicamos quien es el llamador e
    //indicamos
    iMostrarNombre=new Intent(
                    this, ActividadMostrarNombres.class);
    iMostrarNombre.putExtra
            (ActividadMostrarNombres.parametroNombre
                            , nombre);
    //Se lanza el intent
    this.startActivity(iMostrarNombre);

}
```

2.6 Construcción de actividades

Para crear una actividad desde cero necesitamos hacer dos cosas

1. Crear el interfaz XML (Eclipse puede que no añada un `id` a dicho interfaz, si no lo ha hecho añadirlo a mano)
2. Crear una clase Java que herede de `Activity`. Dicha clase Java necesita que añadamos algo: el `onCreate` contendrá ahora el código que procesa el `Intent` que nos pasen y también usaremos `setContentView` para cargar un fichero de interfaz o *layout*.

Dentro de la actividad suele ser buena política definir los nombres de los parámetros utilizando como prefijo el nombre del paquete:

```
public class
        ActividadCalculadora extends Activity {

        public static String nombreNum1=
                        "com.ies.actividades2.num1";
        public static String nombreNum2=
                        "com.ies.actividades2.num2";
        public static String nombreOp=
                        "com.ies.actividades2.op";
        @Override
        protected void onCreate(Bundle savedInstanceState) {
                // TODO Auto-generated method stub
                super.onCreate(savedInstanceState);

        }
}
```

2.7 Ejemplo: llamadas entre actividades

Supongamos que deseamos tener una actividad que acepta recibir dos números y un operando. Tras la recepción se efectuará la operación matemática y se mostrará el resultado en un interfaz distinto de la actividad llamadora.

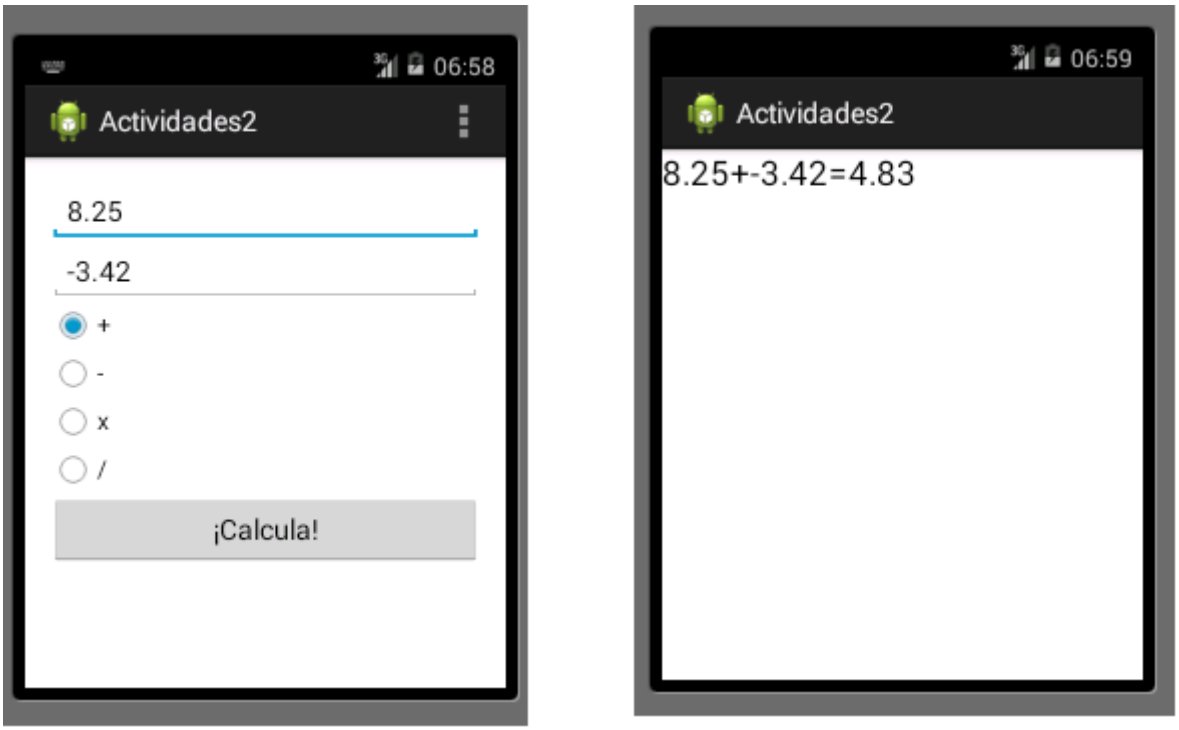


Figura 2.3: Aplicación con dos actividades

2.7.1 Actividad calculadora

Esta actividad carga el interfaz XML y despues procesa el `Intent` para determinar qué operación debe ejecutar.

También define el nombre de los parámetros en **constantes** que tanto el llamador como ella pueden usar (y así evitar el cortar y pegar).

```
public class ActividadCalculadora extends Activity {

        public static String nombreNum1=
                        "com.ies.actividades2.num1";
        public static String nombreNum2=
                        "com.ies.actividades2.num2";
        public static String nombreOp=
                        "com.ies.actividades2.op";
        @Override
        protected void onCreate(Bundle savedInstanceState) {
                // TODO Auto-generated method stub
                super.onCreate(savedInstanceState);
                this.setContentView(R.layout.actividad_secundaria);
                Intent intento=this.getIntent();
                float num1=
                                intento.getFloatExtra(
                                                ActividadCalculadora.nombreNum1
                                                ,
                                                0.0f);
                float num2=intento.getFloatExtra(
                                ActividadCalculadora.nombreNum2
                                ,
                                0.0f);
                String op=
                                intento.getStringExtra(
                                                ActividadCalculadora.nombreOp);

                float resultado=this.calcular(num1, op, num2);

                String cadResultado=
                                num1+op+num2+"="+resultado;
                TextView tvResultado;
                tvResultado=(TextView) findViewById(R.id.tvResultado);
                tvResultado.setText(cadResultado);
        }
        public float calcular(float n1, String op, float n2){
                float resultado=0.0f;

                if (op.equals("+")){
                        return n1+n2;
                }
                if (op.equals("-")){
                        return n1-n2;
```

```
                }
                return resultado;
        }
}
```

2.7.2 Actividad llamadora

```
public class ActividadPrincipal extends ActionBarActivity {

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


        @Override
        public boolean onCreateOptionsMenu(Menu menu) {
                // Inflate the menu; this adds items to the action bar if it is
                getMenuInflater().inflate(R.menu.actividad_principal, menu);
                return true;
        }

        @Override
        public boolean onOptionsItemSelected(MenuItem item) {
                // Handle action bar item clicks here. The action bar will
                // automatically handle clicks on the Home/Up button, so long
                // as you specify a parent activity in AndroidManifest.xml.
                int id = item.getItemId();
                if (id == R.id.action_settings) {
                        return true;
                }
                return super.onOptionsItemSelected(item);
        }



        /* Dado un id de recurso este método nos
         * devuelve el texto que hay dentro
         */
        public float getNumero(int id){
                EditText control;
                control=(EditText) findViewById(id);
                String cadena=control.getText().toString();
                float f=Float.parseFloat(cadena);
                return f;
        }
        public void lanzarActCalculadora(
                        float f1, float f2, String op
```

```
                ){
            Intent intento=new Intent(this, ActividadCalculadora.class);
            intento.putExtra(
                        ActividadCalculadora.nombreNum1,
                        f1);
            intento.putExtra(
                        ActividadCalculadora.nombreNum2,
                        f2);
            intento.putExtra(
                        ActividadCalculadora.nombreOp,
                        op);
            this.startActivity(intento);
        }
        public void calcular(View control){
            RadioButton rbSuma;
            rbSuma=(RadioButton) findViewById(R.id.radSuma);
            if (rbSuma.isChecked()){
                float f1=this.getNumero(R.id.txtNum1);
                float f2=this.getNumero(R.id.txtNum2);
                lanzarActCalculadora(f1,f2,"+");
            }
        }
}
```

2.7.3 Modificación del `AndroidManifest.xml`

Se debe añadir esta actividad en el `AndroidManifest.xml`

```
        <activity
            android:name=".ActividadCalculadora">
</activity>
```

2.8 Ejercicio

Crear una aplicación con dos actividades donde una de ellas permita introducir un texto y un número y la otra reciba ambos valores. La segunda truncará los n primeros caracteres de la cadena y los mostrará en pantalla.

2.8.1 Actividad inicial

```
public class ActPeticionTexto extends ActionBarActivity {

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

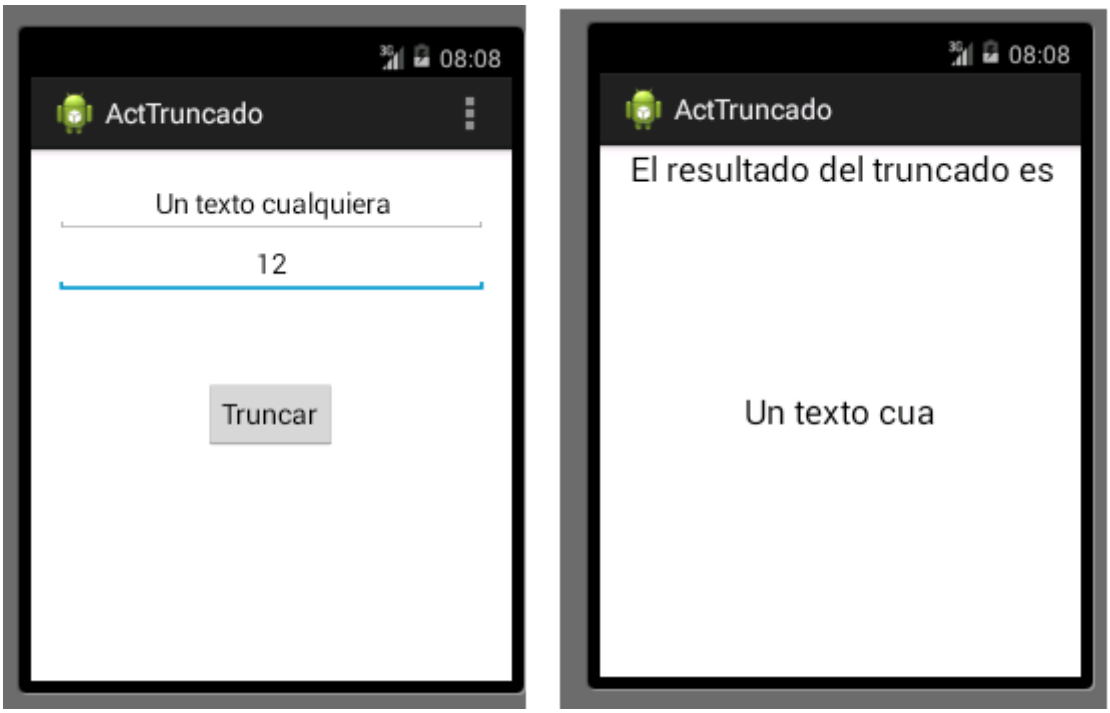


Figura 2.4: Ejemplo de funcionamiento del truncado

```
}


@Override
public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the action bar if it is
        getMenuInflater().inflate(R.menu.act_peticion_texto, menu);
        return true;
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
        // Handle action bar item clicks here. The action bar will
        // automatically handle clicks on the Home/Up button, so long
        // as you specify a parent activity in AndroidManifest.xml.
        int id = item.getItemId();
        if (id == R.id.action_settings) {
                return true;
        }
        return super.onOptionsItemSelected(item);
}
private String getCadena(int id){
        EditText controlTexto=
                        (EditText) this.findViewById(id);
        return controlTexto.getText().toString();
}

public void truncar(View control){
```

```
                Intent intento=new Intent(this,ActividadTruncadora.class);
                String textoEscrito=
                                getCadena(R.id.txtTexto);
                String textoNumCaracteres=
                                getCadena(R.id.txtNumero);
                int numCaracteres=Integer.parseInt(
                                textoNumCaracteres);
                intento.putExtra(
                                ActividadTruncadora.nombreCadena,
                                textoEscrito);
                intento.putExtra(
                                ActividadTruncadora.nombreNumCaracteres,
                                numCaracteres);
                this.startActivity(intento);
        }
}
```

2.8.2 Actividad truncadora

```
public class ActividadTruncadora extends Activity {
        public static String nombreCadena=
                        "com.ies.truncado.nombreCadena";
        public static String nombreNumCaracteres=
                        "com.ies.truncado.nombreNumCaracteres";
        @Override
        protected void onCreate(Bundle savedInstanceState) {
                // TODO Auto-generated method stub
                super.onCreate(savedInstanceState);
                this.setContentView(R.layout.act_truncado);
                Intent intRecibido=this.getIntent();
                String cad=intRecibido.getStringExtra(
                                ActividadTruncadora.nombreCadena
                                );
                int numCaracteres=      intRecibido.getIntExtra(
                        ActividadTruncadora.nombreNumCaracteres,
                        0);

                String cadTruncada=this.truncar(cad, numCaracteres);
                Log.d("Truncado", "Resultado:"+cadTruncada);
                TextView tvTextoTruncado=
                                (TextView) this.findViewById(R.id.tvResultado);
                tvTextoTruncado.setText(cadTruncada);
        }

        /* Recorta los num primeros caracteres*/
        private String truncar (String cad, int num){
                /* Si el usuario intenta poner
                 * un valor más grande que la propia
                 * longitud de la cadena, reducimos el numero
```

```
                 */
                if (num>cad.length()){
                        num=cad.length();
                }
                return cad.substring(0, num);
        }
}
```

2.9 Ejercicio

Crear una aplicación que permita simular un juego de apuestas a la ruleta.

El usuario puede apostar de 1 a 100 euros y parte con un saldo inicial ficticio de 1000 euros.

La ruleta tiene 37 números (del uno al 36 más el 0, que será un caso especial) y el usuario puede apostar de dos formas:

- Puede apostar a par o impar: si apuesta por ejemplo 2 euros a "Par" y sale por ejemplo, el 18, ganará un 50 % más, es decir los dos euros se multiplican por 0'5 y ganará un euro. Si pierde, pierde los dos euros.
- Puede apostar a que el número está en la primera docena (del 1 al 12) en la segunda docena (del 13 al 24) o en la tercera docena (del 25 al 36). Si por ejemplo apostamos 3 euros a la primera docena y sale por ejemplo el 7 multiplicamos por 0,66 los 3 euros y obtendremos 2 euros de beneficio. Si perdemos perdemos los 3 euros que apostamos.
- El 0 significa que la banca gana. No importa si la apuesta se hizo a "Par" o a "Primera docena". Perderemos todo lo que apostamos.

2.10 Bases de datos y almacenamiento.

Android ofrece 5 posibilidades a la hora de almacenar datos:

- Preferencias.
- Almacenamiento interno.
- Almacenamiento externo.
- Bases de datos SQLite.
- Almacenamiento en la nube.

2.10.1 Preferencias compartidas

Dentro de las preferencias se puede almacenar cualquier tipo de datos básicos: `String`, `int`, `floats`, `ints` y `longs`. Dentro de nuestra actividad podemos usar dos tipos de preferencias

- Preferencias compartidas: lo usaremos cuando queramos manejar muchos ficheros de preferencias, debiendo indicar siempre un nombre de fichero.
- Preferencias únicas. Si solo queremos un fichero de preferencias para la actividad no tendremos que indicar ningún nombre de fichero.

Para escribir valores usaremos el objeto `SharedPreferences` de esta forma

```
public class Actividad1 extends Activity{
    private static String ficheroPrefs="misPrefs.prf";
    private MODO_FICHERO=MODE_PRIVATE
    protected void onCreate(Bundle estado){
        SharedPreferences prefs=getSharedPrefs(
                ficheroPrefs, MODO_FICHERO);
        SharedPreferences.Editor editor=
                prefs.edit();
        editor.putString("nombreUsuario", "pepe");
        /* ¡NO HAY QUE OLVIDAR EL COMMIT!*/
        editor.commit();
    }
```

Un fichero se puede crear de varias maneras:

- `MODE_PRIVATE`
- `MODE_WORLD_READABLE`
- `MODE_WORLD_WRITABLE`
- `MODE_MULTI_PROCESS`: Lo usaremos cuando queramos indicar que muchos ficheros van a cambiar a la vez el fichero en forma `MODE_PRIVATE | MODE_MULTI_PROCESS`.

Por ejemplo, supongamos una aplicación que desea guardar un texto como el nombre de usuario que está almacenado en un control `EditText`. El código para almacenar sería algo así:

```
    String ficheroPrefs="nombre_usuario";
    String claveUltimoUsuario="ultimo_usuario";

    public void guardar(View control){
    EditText txtNombre=
                    (EditText) findViewById(R.id.txtNombre);
    String cadena=
                    txtNombre.getText().toString();
    SharedPreferences gestorPrefs;


    gestorPrefs=this.getSharedPreferences(
                    ficheroPrefs, MODE_PRIVATE);

    SharedPreferences.Editor editor;
    editor=gestorPrefs.edit();

    editor.putString(claveUltimoUsuario,
```

```
                    cadena);

    /* Si no hay commit, no se cierra
     * la transacción->No se almacenará
     */
    editor.commit();
    Log.d("Almacen:", "Cadena almacenada");
}
```

2.10.2 Almacenamiento interno

Los ficheros creados aquí son privados a nuestra aplicación. Ni siquiera el usuario puede acceder a ellos (salvo en caso de teléfonos *rooteados*)

Para almacenar haremos algo como esto:

```
String fichero  =   "saludo.txt";
String mensaje  =   "Hola mundo";
FileOutputStream fos=openFileOutput(fichero, MODE_PRIVATE);
fos.write(mensaje.getBytes());
fos.close();
```

2.10.3 Almacenamiento externo

Implica solicitar permisos como `READ_EXTERNAL_STORAGE` o `WRITE_EXTERNAL_STORAGE`.

El almacenamiento puede estar o no disponible, se debería comprobar con algo como:

```
String estado =
        Environment.getExternalStorageState();
if (Environment.MEDIA_MOUNTED.equals(state)) {
        /* Podemos escribir y además leer*/
        return SE_PUEDE_ESCRIBIR;
}
if (Environment.MEDIA_MOUNTED.equals(estado) ||
Environment.MEDIA_MOUNTED_READ_ONLY.equals(state)) {
        return SOLO_SE_PUEDE_LEER;
}
/* Si llegamos aquí no se puede hacer nada*/
return NO_SE_PUEDE_HACER_NADA;
```

Como vemos, la clave está en la clase `Environment` que nos ofrece diversos métodos y constantes para acceder a directorios de la tarjeta.

Por ejemplo, el código siguiente ilustra como conseguir crear un subdirectorio en el directorio estándar de imágenes:

public static String	DIRECTORY_ALARMS	Standard directory in which to place any audio files that should be in the list of alarms that the user can select (not as regular music).
public static String	DIRECTORY_DCIM	The traditional location for pictures and videos when mounting the device as a camera.
public static String	DIRECTORY_DOCUMENTS	Standard directory in which to place documents that have been created by the user.
public static String	DIRECTORY_DOWNLOADS	Standard directory in which to place files that have been downloaded by the user.
public static String	DIRECTORY_MOVIES	Standard directory in which to place movies that are available to the user.
public static String	DIRECTORY_MUSIC	Standard directory in which to place any audio files that should be in the regular list of music for the user.
public static String	DIRECTORY_NOTIFICATIONS	Standard directory in which to place any audio files that should be in the list of notifications that the user can select (not as regular music).
public static String	DIRECTORY_PICTURES	Standard directory in which to place pictures that are available to the user.
public static String	DIRECTORY_PODCASTS	Standard directory in which to place any audio files that should be in the list of podcasts that the user can select (not as regular music).
public static String	DIRECTORY_RINGTONES	Standard directory in which to place any audio files that should be in the list of ringtones that the user can select (not as regular music).

Figura 2.5: Directorios estándar

```
String miDir="mis_imgs";
File file =
        new File (
        Environment.getExternalStoragePublicDirector
                (Environment.DIRECTORY_PICTURES),
                miDir
);
if (!file.mkdirs()) {
Log.e("Error", "No se pudo crear "+miDir);
}
```

2.11 Ejercicio

En los juegos de apuestas, todo jugador siempre desea saber el punto en el que debió dar marcha atrás, sin embargo, no siempre es fácil recordar cual fue.

Para facilitar esto se desea modificar el programa de simulación de la ruleta para que se vaya almacenando todo el historial de apuestas en un fichero llamado `historial.txt`.

En dicho historial deberíamos ir viendo el saldo, el tipo de apuesta que hizo el usuario (si fue a par, si fue a la segunda docena...), el número que salió al apostar y el estado en que quedó el saldo. Estas operaciones deben almacenarse cada vez que el usuario hace una apuesta del tipo que sea.

2.12 Bases de datos

En Android es perfectamente posible utilizar bases de datos relacionales con prácticamente todas sus características: tablas, claves primarias y ajenas, consultas, etc... El corazón de este sistema es *SQLite <http://sqlite.org>* un gestor de bases de datos pensado para dispositivos reducidos y con versiones para prácticamente todas las plataformas.

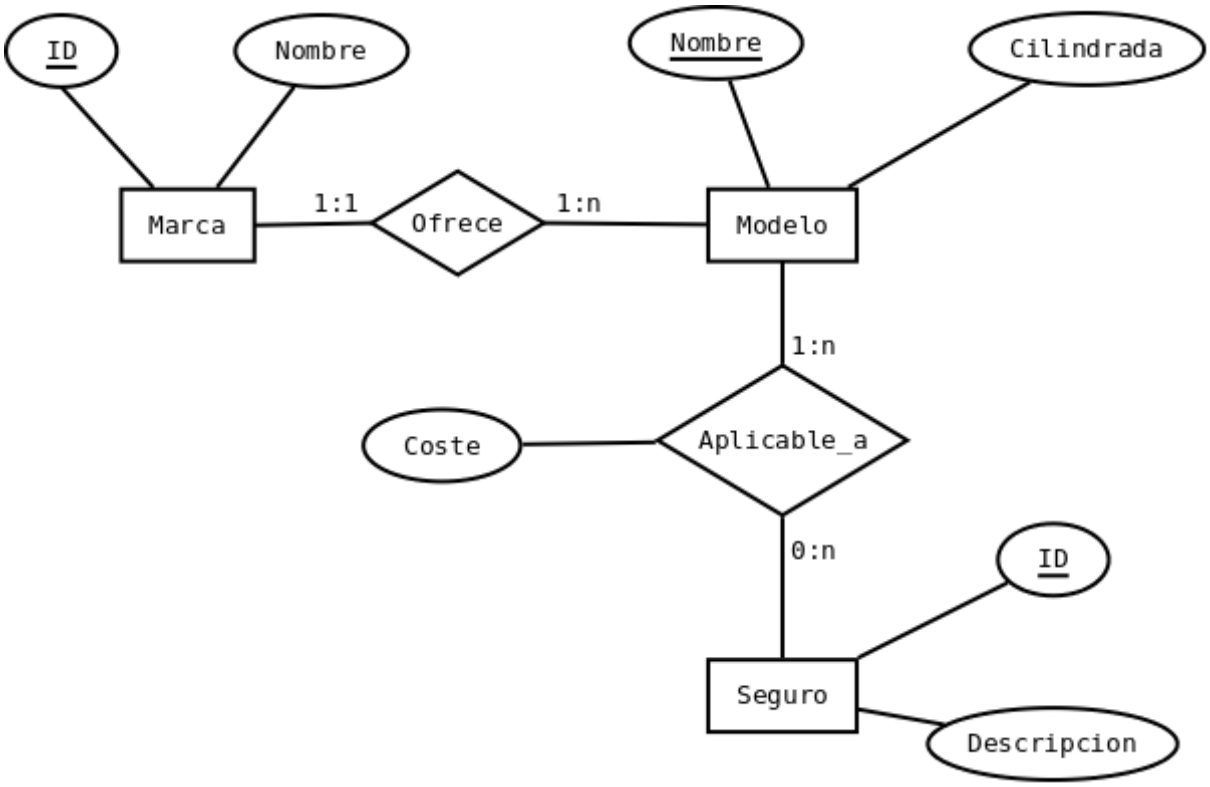
Para operar con bases de datos la documentación oficial de Google aconseja utilizar *clases contrato*. En dichas clases se almacenarán los nombres de las tablas, campos y demás, con el fin de facilitar el mantenimiento. Aunque no es obligatorio es muy aconsejable implementar el interfaz `BaseColumns`. De hecho hay muchos casos en los que Android espera clases que implementen dicho interfaz.

Supongamos que deseamos almacenar información técnica sobre modelos de automóvil y los costes asociados de un seguro: supongamos que hay coches de muchas marcas y modelos, y que para cada uno de ellos se puede contratar un seguro de uno de estos tipos (aunque se desea poder tener más tipos de seguro en el futuro):

- Seguro obligatorio.
- Seguro lunas+incendio sin franquicia.
- Seguro lunas+incendio con franquicia.
- Seguro todo riesgo.

No todos los seguros se ofrecen para todos los coches y de hecho podría haber coches para los cuales la compañía no ofrece ningún seguro. No todos los tipos de seguro tienen el mismo coste para todos los coches. Todo seguro tiene una validez medida en días y siempre es la misma para un cierto tipo de seguro. Por ejemplo, todos los seguros obligatorios tienen una validez de 365 días y todos los "todo riesgo" de 90. Los seguros del tipo "lunas+incendio" tienen todos una validez de 180 días.

Toda marca tiene un nombre y un código único, todo modelo tiene un nombre único y una cilindrada. Todo seguro tiene un un código único y una descripción.



Con esto el SQL que necesitaríamos por ejemplo para la tabla `Marcas` sería algo así:

```
create table marcas
(
    id integer primary key,
    nombre varchar(40)
);

insert into marca values (1, "Ford");
insert into marca values (2, "Renault");
```

Y la clase contrato Java asociada a esta entidad sería:

```
public class MarcasContrato implements BaseColumns {
        public static final String
                NOMBRE_TABLA="marcas";
        public static final String
                NOMBRE_COL_ID="id";
        public static final String
```

```
                NOMBRE_COL_NOMBRE="nombre";
}
```

2.12.1 Bases de datos SQLite

Supongamos que deseamos crear una base de datos sobre seguros de coches. Un primer elemento necesario sería una tabla donde se almacenen las marcas (cada una llevará un ID).

```
create table marcas (
        id      integer primary key,
        marca   char(30)
);

insert into marcas values (1, 'Ford');
insert into marcas values (2, 'Renault');
```

Para manejar la creación y procesado de esta base de datos Android ofrece la clase `SQLiteOpenHelper` de la cual se puede heredar de esta manera:

```
public class BD extends SQLiteOpenHelper {

        private String sqlCreacion=
            "create table marcas(id integer primary key," +
                "nombre varchar(40));\n" ;
        private String insert1="insert into marca values (1, \"Ford\")";
        private String insert2="insert into marca values (2, \"Renault\");";
        public BD(Context context, String name,
            CursorFactory factory, int version) {
                super(context, name, factory, version);
        }

        @Override
        public void onCreate(SQLiteDatabase db) {
                db.execSQL(sqlCreacion);
                db.execSQL(insert1);
                db.execSQL(insert2);

        }
}
```

Y podemos crear un objeto de la clase BD simplemente instanciándolo

Advertencia: Cuando se hacen pruebas en el simulador es posible que el fichero de base de datos no aparezca hasta que no intentemos leer o escribir datos de él. Aparte de eso, el fichero suele estar en el directorio `/data/data/<paquete>` *pero las ubicaciones pueden cambiar*.

2.12.2 Datos y cursores

Un objeto del tipo `SQLiteOpenHelper` nos puede devolver un objeto `SQLiteDatabase` que tiene los métodos necesarios para acceder a cursores, hacer consultas y recorrer los datos. El código siguiente muestra un ejemplo:

```
BD gestorBD=new BD(this, "seguros.db", null, 1);
SQLiteDatabase bd=gestorBD.getReadableDatabase();
Cursor cursor=bd.rawQuery("select id, nombre from marcas", null);
cursor.moveToFirst();
int posID=cursor.getColumnIndex(MarcasContrato.NOMBRE_COL_ID);
int posNombreMarca=
    cursor.getColumnIndex(MarcasContrato.NOMBRE_COL_NOMBRE);
while (!cursor.isAfterLast()){
    String numero=cursor.getString(posID);
    String marca=cursor.getString(posNombreMarca);
    Log.d("Marca:", numero+":"+marca);
    cursor.moveToNext();
}
cursor.close();
```

2.13 Ejercicio: ampliación de la BD

Ampliar la base de datos para que exista una tabla "Modelos" que incluya un par de modelos de cada marca (tiene que haber claves ajenas).

- Marca: Ford, Modelo: Focus
- Marca: Ford, Modelo: Mondeo
- Marca: Renault, Modelo: Megane
- Marca: Renault, Modelo: Kangoo

Hacer un programa que recupere todos los modelos de coche junto con sus marcas y los muestre en pantalla.

2.14 Solución a la ampliación de la BD

En primer lugar, habría que crear el SQL que permita tener la segunda tabla con la clave ajena:

```
create table marcas
(
    id integer primary key,
    nombre varchar(40)
);

insert into marcas values (1, "Ford");
insert into marcas values (2, "Renault");
```

```
create table modelos
(
    id_modelo integer primary key,
    id_marca integer,
    nombre varchar(40),
    foreign key (id_marca) references marcas (id)
);
insert into modelos values(1, 1, 'Focus');
insert into modelos values(2, 1, 'Mondeo');
insert into modelos values(3, 2, 'Megane');
insert into modelos values(4, 2, 'Kangoo');
```

En lugar de insertar todo el código SQL **se puede crear el archivo de base de datos en un ordenador** e insertarlo en el proyecto despues, por desgracia Android no ofrece un soporte cómodo para hacer esto, ya que una vez instalada la app tenemos que copiar el fichero de base de datos al terminal para despues abrirlo.

2.14.1 Creación dinámica del interfaz

Un problema fundamental en este ejercicio es que no sabemos a priori cuantos controles poner en la aplicación: **el interfaz se tiene que crear dinámicamente**

Supongamos que simplemente deseamos crear un ListView en el que se muestren simplemente los modelos de coche. Se pueden utilizar un par de clases útiles para conseguir lo que queramos de la forma siguiente:

- Metemos los nombres en un vector de Strings.
- Crearemos un fichero de layout para indicar como se mostrará cada modelo. Este layout debe tener un TextView (Android encontrará automáticamente el TextView y en él insertará cada nombre de modelo).
- Usamos la clase `ArrayAdapter`, que "convierte" cada elemento de nuestro *array* en un elemento del ListView. Le indicaremos el fichero de layout que debe crearse *para cada modelo individual*.
- Indicamos al ListView que use ese `ArrayAdapter`.

Para conseguir que la clase BD nos devuelve un vector de Strings le podemos añadir el siguiente método:

```
public String[] getArrayModelos(){
    String[] vectorResultado;
    SQLiteDatabase bd=this.getReadableDatabase();
    Cursor cursor=bd.rawQuery("select nombre from modelos", null);
    vectorResultado=new String[cursor.getCount()];
    cursor.moveToFirst();
    int pos=0;
    while (!cursor.isAfterLast()){
            vectorResultado[pos]=cursor.getString(0);
            Log.d("D",  cursor.getString(0));
```

```
            cursor.moveToNext();
            pos++;
        }
        return vectorResultado;
    }
```

Y para que ahora la Actividad cree el interfaz dinámicamente podemos hacer algo como esto:

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_actividad_principal);

    /* Recuperamos los modelos*/
    String[] modelosCoche=gestorBD.getArrayModelos();
    /* Cada modelo de coche se insertará en el fichero
     * de layout que tiene un textview donde
     * se pondrá el nombre del modelo */
    ArrayAdapter<String> adaptador=
                    new ArrayAdapter(this,
                                     R.layout.modelo, R.id.tvNombreModelo,
                                     modelosCoche);
    /* El Listview de nuestro interfaz cargará
     * los datos a partir de ese adaptador */
    ListView lvModelos=(ListView) this.findViewById(R.id.lvModelos);
    lvModelos.setAdapter(adaptador);
}
```

2.14.2 Respondiendo al evento click

Un problema que ocurre es que aunque alguien haga click en algún modelo no ocurre nada. Para conseguir que la actividad procese el evento podemos hacer que la actividad implemente el interfaz `AdapterView.OnItemClickListener`.

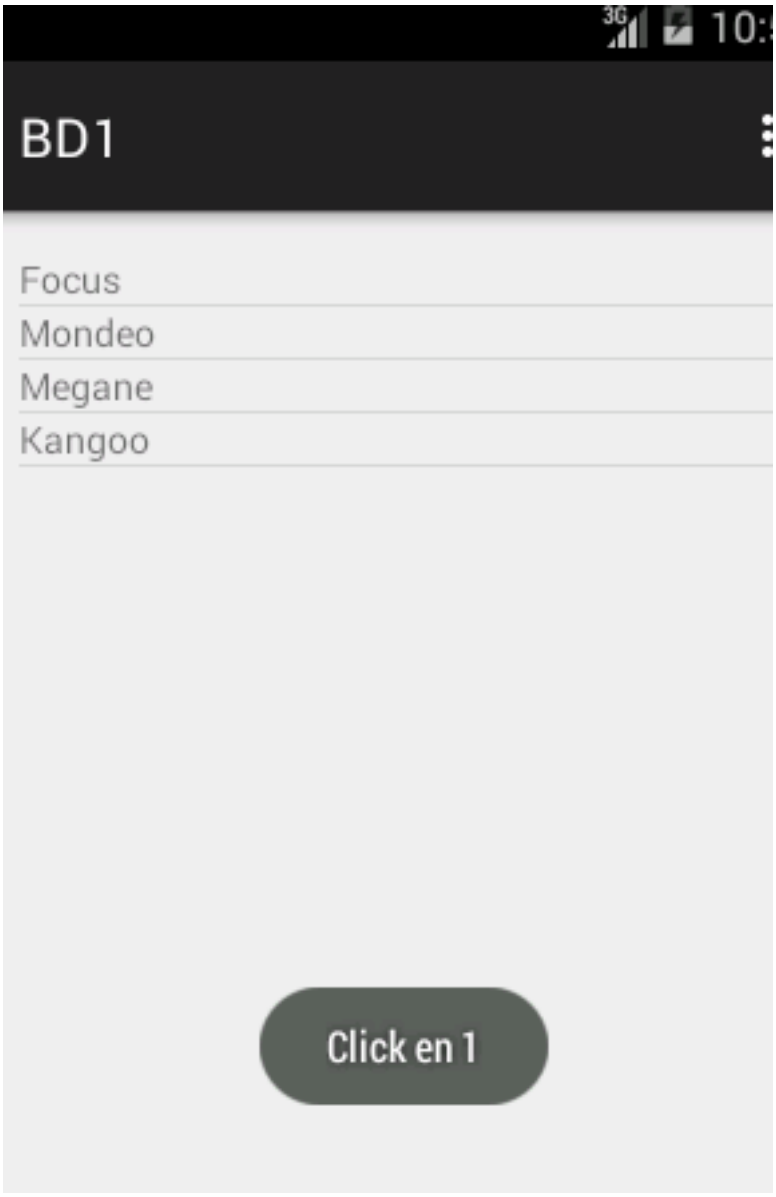
> **Advertencia:** Al hacer esto debemos asegurarnos de poner a `false` el atributo `Focusable` de los controles. En concreto deberemos ir al fichero de layout que sirve de plantilla para cada control y si por ejemplo hay un `EditText` (que podría apoderarse del evento click) modificar su `Focusable` como hemos dicho. Se debe hacer esto para todos los controles.

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_actividad_principal);

    /* Recuperamos los modelos*/
    String[] modelosCoche=gestorBD.getArrayModelos();
    /* Cada modelo de coche se insertará en el fichero
     * de layout que tiene un textview donde
     * se pondrá el nombre del modelo */
    ArrayAdapter<String> adaptador=
```

```
                    new ArrayAdapter(this,
                                        R.layout.modelo, R.id.tvNombreModelo,
                                        modelosCoche);
    /* El Listview de nuestro interfaz cargará
     * los datos a partir de ese adaptador */
    ListView lvModelos=(ListView) this.findViewById(R.id.lvModelos);
    lvModelos.setAdapter(adaptador);
    /* Activar la gestión de eventos*/
    lvModelos.setOnItemClickListener(this);
}
public void onItemClick(AdapterView<?> padre, View control, int posicion,
                    long id) {
    Toast.makeText(this, "Click en "+posicion, Toast.LENGTH_SHORT).show();
}
```

Esto se muestra en el emulador de esta forma



2.14.3 Creación dinámica de controles avanzada

El `ArrayAdapter` es útil pero solo sirve cuando manejamos cadenas, lo cual es un dato simple. Sin embargo, si queremos representar algo más complejo (por ejemplo, el nombre de modelo y el código de modelo) se necesita una clase más avanzada, que además pueda recuperar datos directamente de un `Cursor`: la clase `SimpleCursorAdapter`

Esta clase espera que pasemos cuatro cosas: * La clase padre: normalmente `this` * Un vector con los nombres de las columnas que vamos a extraer de la consulta. * Un vector con los `id` de recursos de controles donde se va a meter el valor de cada columna. * Flags que puedan modificar el comportamiento de la clase (normalmente usaremos 0).

El problema principal es que esta clase **espera que nuestro cursor tenga algún campo llamado _id** que actúe como identificador así que tendremos que reescribir nuestro SQL y pasarlo de esto .. code-block:: sql

    select id_modelo, nombre from modelos

a esto

```
select id_modelo as _id, nombre from modelos
```

Así que ahora un método que podría rellenar controles dinámicamente sería este:

```
public void rellenarControles(){
    SQLiteDatabase bd=gestorBD.getReadableDatabase();
    Cursor cursor=bd.rawQuery("select id_modelo as _id, nombre from modelos",nul
    String[] nombresColumnas={
                    BaseColumns._ID,
                    ModelosContrato.NOMBRE_COL_NOMBRE
    };
    int[] idTextViews={
                    R.id.tvIdModelo,
                    R.id.tvModelo
    };
    SimpleCursorAdapter adaptador=
                    new SimpleCursorAdapter(this,
                                    R.layout.modelos_avanzados,cursor,
                                    nombresColumnas, idTextViews,0);
    ListView lvModelos=(ListView) this.findViewById(R.id.lvModelos);
    lvModelos.setAdapter(adaptador);
    lvModelos.setOnItemClickListener(this);
}
```

Advertencia: El código para cargar datos o crear controles puede ser lento y podría llegar a bloquear el interfaz. Es recomendable delegar todo el código que pueda ser lento a una `AsyncTask`

2.15 Servicios en dispositivos móviles.

2.16 Proveedores de contenido.

Para poder echar un vistazo a un proveedor de contenidos muy utilizado, el de los contactos necesitaremos crear algunos contactos de prueba. Para leer desde proveedores de contenidos puede ser necesario activar permisos en el `AndroidManifest.xml`. Por ejemplo, para poder leer contactos es necesario el permiso `READ_CONTACTS`

2.16.1 Fundamentos

Un proveedor de contenidos es una clase Java que permite acceder a datos **como si esos datos estuvieran en una tabla** (aunque no estén). Para acceder a estos datos *se necesita la URI correcta* que habrá que buscar en la documentación.

Una URI es más o menos como una tabla. Se podrán acceder a sus datos haciendo "consultas", cuyos resultados podremos recorrer con un cursos y extraer los campos que deseemos.

2.16.2 Un ejemplo

La siguiente clase accede a diversa información de contactos. La URI de contactos es algo como "content://com.android.contacts/data".

Sin embargo, poner las URI directamente como un String es muy arriesgado, ya que la URI puede cambiar. Para evitar esto Android ofrece clases con constantes que permiten que nuestro código acceda a URIs sin que tengamos que preocuparnos por problemas futuros debidos a cambios en la URI. En Android la URI para los contactos es `ContactsContract.Data.CONTENT_URI`.

Esta "tabla" tiene diversos campos, que podemos extraer mediante sus "nombres de contrato", es decir constantes que nos protegen contra posibles cambios de nombre de campo que pueda haber en el futuro.

Un problema que ocurre a menudo es que la estructura de los proveedores de contenido puede ser muy extraña. Por ejemplo, a nivel interno, Android ofrece en esta tabla de datos filas para cada trozo de información de un contacto. Es decir, si hay un contacto llamado "Pepe Perez" con teléfono "555-123456", **veremos dos filas**, una para cada trozo.

Al recorrer la tabla de datos podemos examinar la columna MIMETYPE, que nos dirá lo que hay almacenado en DATA1, la columna que contiene la información relevante. En realidad hay alias que ofrecen nombres más significativos que DATA1, pero aún así no se debe olvidar consultar el MIMETYPE.

El ejemplo siguiente extrae todos los "trozos de datos" y nos muestra solo los email.

```
public class GestorContactos {
        Uri uriContactos;
        public GestorContactos(ContentResolver cr){
```

```
                uriContactos=
                        ContactsContract.Data.CONTENT_URI;
                Log.d("DEBUG", "Gestor contactos construido");
                Log.d("DEBUG", "La URI es:"+
                        uriContactos.toString());

                String[] campos={
                        ContactsContract.Data.DISPLAY_NAME
                };
                Cursor cursor=
                        cr.query(uriContactos, campos,
                                        null, null, null);
                int numDatos=cursor.getCount();
                Log.d("DEBUG", "Num datos:"
                        +numDatos);
                cursor.moveToFirst();
                while (!cursor.isAfterLast()){
                        int posData1=
                                        cursor.getColumnIndex(
                                                        ContactsContract.Data.DA
                        int posTipo=
                                        cursor.getColumnIndex(
                                                        ContactsContract.Data.MI
                                                        );
                        String tipo=
                                        cursor.getString(posTipo);
                        if (
                                tipo.equals(ContactsContract.CommonDataKinds.Ema
                        {
                                String data1=cursor.getString(posData1);
                                Log.d("DEBUG",
                                                "El data 1/email:"+data1);
                        }
                        cursor.moveToNext();
                }
        }
}
```

Otra forma de acceder a la información es la siguiente:

2.17 Ejercicio: proveedor de diccionario

Sabiendo que Android ofrece un proveedor para el diccionario del usuario, añadir algunas palabras a dicho diccionario.

2.17.1 Solución al diccionario

En realidad la clase `UserDictionary.Words` ofrece un método `addWord` que resuelve esta tarea. Sin embargo, probaremos a hacerlo manejando directamente el proveedor de contenidos. En concreto, para poder insertar valores se tiene que hacer uso de otra clase llamada `ContentValues`, dentro de la cual se pondrá la información que se desea almacenar en el proveedor.

La clase `UserDictionary.Words` requiere pasar lo siguiente:

- La palabra a incluir en el diccionario.
- La frecuencia con la que pensamos que aparece, siendo 1 el valor "muy poco probable que aparezca" y el 255 el "aparece con mucha frecuencia".
- A partir de la versión 16 de Android el programador puede, si lo desea, pasar un "atajo" es decir, una abreviatura que Android luego puede expandir automáticamente.

El código siguiente ilustra como insertar un valor:

```
public class GestorDiccionario {
        ContentResolver cr;
        Uri uriDiccionario;
        public GestorDiccionario(ContentResolver cr){
                this.cr=cr;
                uriDiccionario=android.provider.UserDictionary.Words.CONTENT_URI
                Log.d("DEBUG", "La URI es:"+uriDiccionario.toString());
        }
        public void insertarPalabra(String palabra, String atajo){
                /* Este objeto "empaqueta" los valores que queremoa
                 * insertar en una URI
                 */
                ContentValues objetoValores=new ContentValues();
                /* La frecuencia puede ir de 1 a 255 (siendo el 255 "muy frecuen
                 * En principio suponemos que nuestra palabra es poco frecuente
                 */
                objetoValores.put(UserDictionary.Words.FREQUENCY, 1);
                objetoValores.put(UserDictionary.Words.WORD, palabra);
                /* Los atajos solo se pueden insertar a partir
                 * de la version 16 de Android
                 */
                if (Build.VERSION.SDK_INT>=16)
                {
                        objetoValores.put(UserDictionary.Words.SHORTCUT, atajo);
                }
                cr.insert(uriDiccionario, objetoValores);
        }
}
```

2.18 Gestión de recursos y notificaciones.

2.19 Técnicas de animación y sonido.

A partir de su versión 3 (API 11) Android ofrece una enorme variedad de posibilidades para animar elementos. Aunque se puede hacer desde código, el método recomendado es usar XML. Una animación definida en XML puede aplicarse a cualquier elemento.

Una animación puede ser de tres tipos:

1. Animación de un valor: elemento `<animator>`
2. Animación de un objeto: elemento `<objectAnimator>`
3. Agrupamiento de animaciones de valor o de objeto: elemento `<set>`

La animación de un valor es tan simple como esto en Java:

```
ValueAnimator animacion = ValueAnimator.ofInt(100,200)
animation.setDuration(1000);
animation.start();
```

O en XML:

```
<animator android:valueFrom="100"
    android:valueTo="200"
    android:duration="1000">
</animator>
```

Las animaciones XML se almacenarán en el directorio `res/animator/<nombre_archivo>.xml`. Un problema es que la animación de un valor no hace nada: **hay que implementar listeners**. Si nuestra clase implementa el interfaz `ValueAnimator.AnimatorUpdateListener` y de él implementa el método `onAnimationUpdate(ValueAnimator animacion)`, este método se ejecutará automáticamente cada vez que toque "mostrar un nuevo cuadro de animación". Este problema se resuelve con los `ObjectAnimator`

2.19.1 Ejercicio: contador

Implementa un programa que tenga un boton y un cuadro de texto con un número. Cuando el usuario haga click en el botón el contador irá de 0 a 100. Usa animaciones de valores Java y luego hazlo con XML.

2.20 Contexto gráfico. Imágenes.

2.20.1 ¿Qué es OpenGL?

2.20.2 Usando OpenGL

En primer lugar se debe añadir al `AndroidManifest` esta línea, (por ejemplo, justo encima de `<application>`):

```
<uses-feature android:glEsVersion="0x00020000" android:required="true" />
```

En segundo lugar vamos a necesitar dos objetos

1. En algún punto del interfaz se debe añadir un objeto de la clase `GlSurfaceView`.Los gráficos se dibujarán en este objeto. A este objeto lo llamaremos "Superficie".
2. Se debe programar una clase que herede de `GLSurfaceView.Renderer`. Esta clase será la encargada de dibujar en la superficie definida antes. A este objeto lo llamaremos "Renderer".

Empecemos por crear una clase Java como la que mostramos aquí. Esta será nuestra superficie:

```
public class VistaGL extends GLSurfaceView {
    public VistaGL(Context context) {
            super(context);
    }
}
```

Ahora creamos el objeto encargado de realizar los dibujos:

```
import javax.microedition.khronos.egl.EGLConfig;
import javax.microedition.khronos.opengles.GL10;
import android.opengl.GLSurfaceView;

public class Dibujador implements GLSurfaceView.Renderer {
        @Override
        public void onSurfaceCreated(GL10 gl, EGLConfig config) {
        }
        @Override
        public void onSurfaceChanged(GL10 gl, int width, int height) {
        }
        @Override
        public void onDrawFrame(GL10 gl) {
        }
}
```

Toda clase que implemente el interfaz `GLSurfaceView.Renderer` tiene que implementar esos tres métodos:

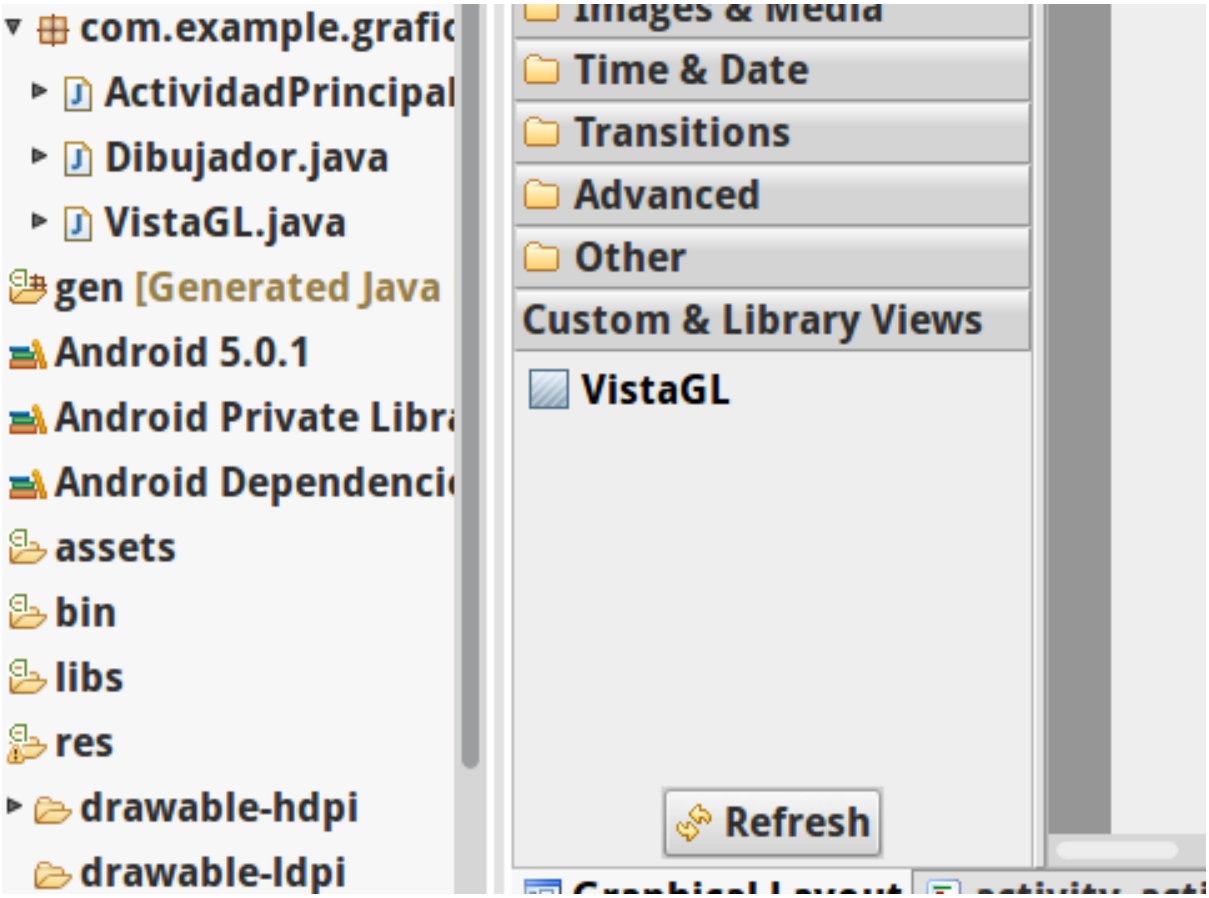
1. El método `onSurfaceCreated` se ejecutará una sola vez, al crear la superficie.

2. El método `onSurfaceChange` se ejecutará cuando haya algún cambio en la geometría de la superficie (habitualmente si rotamos la pantalla).
3. El método `onDrawFrame` se ejecuta continuamente (será aquí donde implementemos cambios en el dibujo, animaciones etc...)

Ahora hay que indicar a la superficie qué objeto se va a encargar de hacer los dibujos. Modificamos el código de la superficie:

```
public class VistaGL extends GLSurfaceView {
    Dibujador dibujador;
    public VistaGL(Context context) {
            super(context);
            dibujador=new Dibujador();
            this.setRenderer(dibujador);
    }
}
```

Aún no hemos añadido nada al interfaz de nuestra aplicación. Nuestra clase `VistaGL` puede añadirse como si fuera un control más, solo tenemos que revisar Eclipse y veremos que nuestra clase es seleccionable y se puede añadir, como si fuera un objeto predeterminado. Si no está pulsaremos el botón "Refresh".



2.21 Eventos del teclado.

2.22 Descubrimiento de servicios.

2.23 Persistencia.

2.24 Modelo de hilos.

2.25 Comunicaciones: clases asociadas. Tipos de conexiones.

2.26 Gestión de la comunicación inalámbrica.

2.27 Seguridad y permisos.

2.28 Envío y recepción de mensajes texto.

2.29 Envío y recepción de mensajería multimedia. Sincronización de contenido.

2.30 Manejo de conexiones HTTP y HTTPS.

2.31 Empaquetado y despliegue de aplicaciones para dispositivos móviles.

2.32 Centros de distribución de aplicaciones.

2.33 Documentación de aplicaciones de dispositivos móviles.

CAPÍTULO 3

Análisis de motores de juegos

3.1 Animación 2D y 3D.

3.2 Arquitectura del juego. Componentes.

3.3 Motores de juegos: Tipos y utilización.

3.4 Áreas de especialización, librerías utilizadas y lenguajes de programación

3.5 Componentes de un motor de juegos.

3.6 Librerías que proporcionan las funciones básicas de un Motor 2D/3D.

3.7 APIs gráficos 3D.

3.8 Estudio de juegos existentes.

3.9 Aplicación de modificaciones sobre juegos existentes.

CAPÍTULO 4

Desarrollo de juegos 2D y 3D

4.1 Entornos de desarrollo para juegos.

4.2 Integración del motor de juegos en entornos de desarrollo.

4.3 Conceptos avanzados de programación 3D.

4.4 Fases de desarrollo:

4.5 Propiedades de los objetos: luz, texturas, reflejos, sombras.

4.6 Aplicación de las funciones del motor gráfico. Renderización.

4.7 Aplicación de las funciones del grafo de escena.

4.8 Tipos de nodos y su utilización.

4.9 Asociación de sonidos a los eventos del juego.

4.10 Análisis de ejecución. Optimización del código.

4.11 Documentación de la fase de diseño y de desarrollo.

CAPÍTULO 5

Sistemas basados en localización

5.1 Tecnologías de localización (GPS, A-GPS,...).

5.2 Servicios de localización, mapas y geocodificación.

5.3 Emuladores para simular las ubicaciones.

5.4 Visualización la información geolocalizada.