Plataforma Android

Índice

1. Introducción a Android

1.1. Android

Android es un sistema operativo de código abierto para dispositivos móviles, se programa principalmente en Java, y su núcleo está basado en Linux.

1.1.1. Historia

Antiguamente los dispositivos empotrados sólo se podían programar a bajo nivel y los programadores necesitaban entender completamente el hardware para el que estaban programando.

En la actualidad los sistemas operativos abstraen al programador del hardware. Un ejemplo clásico es Symbian. Pero este tipo de plataformas todavía requieren que el programador escriba código C/C++ complicado, haciendo uso de librerías propietarias. Especiales complicaciones pueden surgir cuando se trabaja con hardware específico, como GPS, trackballs, pantallas tácticles, etc.

Java ME abstrae completamente al programador del hardware, pero las limitaciones impuestas por la máquina virtual le restringen mucho su libertad a la hora de acceder al hardware del dispositivo.

Esta situación motivó la aparición de Android, cuya primera version oficial (la 1.1) se publicó en febrero de 2009. Esto coincidió con la proliferación de smartphones con pantallas táctiles.

Desde entonces han ido apareciendo versiones nuevas del sistema operativo, desde la 1.5 llamada Cupcake y que se basaba en el núcleo de Linux 2.6.27, hasta la versión que usaremos en este módulo, la versión 2.3, conocida como Gingerbread y basada en el núcleo 2.6.35.x. Cada versión del sistema operativo tiene un nombre inspirado en la repostería, siguiendo un orden alfabético con respecto al resto de versiones de Android (Cupcake, Donut, Eclair, Froyo, Gingerbread, Honeycomb, etc).

1.1.2. Open Source

Android - tanto el sitema operativo, como la plataforma de desarrollo - están liberados bajo la licencia de Apache. Esta licencia permite a los fabricantes añadir sus propias extensiones propietarias, sin tener que ponerlas en manos de la comunidad de software libre.

Al ser de open source, Android hace posible:

- la existencia de una gran comunidad de desarrollo, gracias a sus completas APIs y

documentación ofrecida
- desarrollar desde cualquier plataforma (Linux, Mac, Windows, etc)
- su uso en cualquier tipo de dispositivo móvil
- que cualquier fabricante pueda diseñar un dispositivo que trabaje con Android, incluso adaptando o extendiendo el sistema para satisfacer las necesidades de su dispositivo concreto
- un gran valor añadido para los fabricantes de dispositivos: las empresas se ahorran el coste de desarrollar un sistema operativo completo desde cero
- un gran valor añadido para los desarrolladores: éstos se ahorran tener que programar APIs, entornos gráficos, aprender acceso a dispositivos hardware particulares, etc.

Android está formado por los siguientes componentes:

- núcleo basado en el de Linux para el manejo de memoria, procesos y hardware (se trata de una rama independiente de la rama principal, de manera que las mejoras introducidas no se incorporan en el desarrollo del núcleo de GNU/Linux)
- bibliotecas open source para el desarrollo de aplicaciones, incluyendo SQLite, WebKit, OpenGL y manejador de medios
- entorno de ejecución para las aplicaciones Android. La máquina virtual Dalvik y las bibliotecas específicas dan a las aplicaciones funcionalidades específicas de Android
- un framework de desarrollo que pone a disposición de las aplicaciones los servicios del sistema como el manejador de ventanas, de localización, proveedores de contenidos, sensores y telefonía
- SDK (kit de desarrollo de software) que incluye herramientas, plug-in para Eclipse, emulador, ejemplos y documentación
- interfaz de usuario útil para pantallas táctiles y otros tipos de dispositivos de entrada, como por ejemplo, teclado y trackball
- aplicaciones preinstaladas que hacen que el sistema operativo sea útil para el usuario desde el primer momento. Cabe destacar que cuenta con las últimas versiones de Flash Player
- muy importante es la existencia del Android Market, y más todavía la presencia de una comunidad de desarrolladores que publican allí sus aplicaciones, tanto de pago como gratuitas. De cara al usuario, el verdadero valor del sistema operativo está en las aplicaciones que se puede instalar

El principal responsable del desarrollo de Android es la Open Handset Alliance, un consorcio de varias compañías que tratan de definir y establecer una serie de estándares abiertos para dispositivos móviles. El consorcio cuenta con decenas de miembros que se pueden clasificar en varios tipos de empresas:

- operadores de telefonía móvil
- fabricantes de dispositivos
- fabricantes de procesadores y microelectrónica
- compañías de software
- compañías de comercialización

Por lo tanto, Android no es "de Google" como se suele decir, aunque Google es una de las empresas con mayor participación en el proyecto.

Por último concluimos esta sección considerando algunas cuestiones éticas. Uno de los aspectos más positivos de Android es su carácter de código abierto. Gracias a él, tanto fabricantes como usuarios se ven beneficiados y tanto el proceso de programación de dispositivos móviles como su fabricación se acelera. Todos salen ganando.

Otra consecuencia de que sea de código abierto es la mantenibilidad. Los fabricantes que venden dispositivos con Android tienen el compromiso de que sus aparatos funcionen. Si apareciera algún problema debido al sistema operativo (no nos referimos a que el usuario lo estropee, por supuesto) el fabricante, en última instancia, podría abrir el código fuente, descubrir el problema y solucionarlo. Esto es una garantía de éxito muy importante.

Por otro la seguridad informática también se ve beneficiada por el código abierto, como ha demostrado la experiencia con otros sistemas operativos abiertos frente a los propietarios.

Hoy en día los dispositivos móviles cuentan con hardware que recoge información de nuestro entorno: cámara, GPS, brújula y acelerómetros. Además cuentan con constante conexión a Internet, a través de la cuál diariamente circulan nuestros datos más personales. El carácter abierto del sistema operativo nos ofrece una transparencia con respecto al uso que se hace de esa información. Por ejemplo, si hubiera la más mínima sospecha de que el sistema operativo captura fotos sin preguntarnos y las envía, a los pocos días ya sería noticia.

Esto no concierne las aplicaciones que nos instalamos. Éstas requieren una serie de permisos antes de su instalación. Si los aceptamos, nos hacemos responsables de lo que la aplicación haga. Es necesario aclarar que esto no es un problema de seguridad,ya que los problemas de seguridad realmente surgen si se hace algo sin el consentimiento ni conocimiento del usuario.

No todo son aspectos éticos positivos, también abundan los preocupantes.

Los teléfonos con Android conectan nuestro teléfono con nuesto ID de google. Hay una transmisión periódica de datos entre Google y nuestro terminal: correo electrónico, calendario, el tiempo, actualizaciones del Android Market, etc. En este sentido, el usuario depende de Google (ya no dependemos sólo de nuestro proveedor de telefonía móvil). Además, la inmensa mayoría de los servicios que Google nos ofrece no son de código abierto. Son gratuitas, pero el usuario desconoce la suerte de sus datos personales dentro de dichas aplicaciones.

El usuario puede deshabilitar la localización geográfica en su dispositivo, y también puede indicar que no desea que ésta se envíe a Google. Aún así, seguimos conectados con Google por http, dándoles información de nuestra IP en cada momento. De manera indirecta, a través del uso de la red, ofrecemos información de nuestra actividad diaria. Si bien el usuario acepta estas condiciones de uso, la mayoría de los usuarios ni se paran a

pensar en ello porque desconocen el funcionamiento del sistema operativo, de los servicios de Google, y de Internet en general.

Lógicamente, nos fiamos de que Google no hará nada malo con nuestros datos, ya que no ha habido ningún precedente.

1.2. Aplicaciones Android

Las aplicaciones Android están compuestas por un conjunto heterogéneo de componentes enlazados mediante un archivo llamado `AndroidManifest.xml` que los describe e indica cómo interactúan. Este archivo también contiene metainformación acerca de la aplicación, como por ejemplo los requerimientos que debe cumplir la plataforma sobre la que se ejecuta.

Una aplicación Android estará compuesta por los siguientes componentes (no necesariamente todos ellos):

- **Actividades**. Las actividades son la capa de presentación de la aplicación. Cada pantalla a mostrar en la aplicación será una subclase de la clase `Activity`. Las actividades hacen uso de componentes de tipo `View` para mostrar elementos de la interfaz gráfica que permitan mostrar datos y reaccionar ante la entrada del usuario.
- **Servicios**. Los servicios son componentes que se ejecutan en el background de la aplicación, ya sea actualizando fuentes de información, atendiendo a diversos eventos, o activando la visualización de notificaciones en una actividad. Se utilizan para llevar a cabo procesamiento que debe ser realizado de manera regular, incluso en el caso en el que nuestras actividades no sean visibles o ni siquiera estén activas.
- **Proveedores de contenidos**. Permiten almacenar y compartir datos entre aplicaciones. Los dispositivos Android incluyen de serie un conjunto de proveedores de contenidos nativos que permiten acceder a datos del terminal, como por ejemplo los contactos o el contenido multimedia.
- **Intents**. Los intents constituyen una plataforma para el paso de mensajes entre aplicaciones (y también dentro de una misma aplicación). Emitiendo un intent al sistema declara la intención de tu aplicación de que se lleve a cabo una determinada acción. El sistema será el encargado de decidir quién lleva a cabo las acciones solicitadas.
- **Receptores**. Permiten a tu aplicación hacerse cargo de determinadas acciones solicitadas mediante intents. Los receptores iniciarán automáticamente la aplicación para responder a un intent que se haya recibido, haciendo que sean ideales para la creación de aplicaciones guiadas por eventos.
- **Widgets**. Se trata de componentes visuales que pueden ser añadidos a la ventana principal de Android.
- **Notificaciones.** Las notificaciones permiten comunicarse con el usuario sin necesidad de robar el foco de la aplicación activa actualmente o de interrumpir a la actividad actual. Pror ejemplo, cuando un dispositivo recibe un mensaje de texto, avisa al usuario mediante luces, sonidos o mostrando algún icono.

1.2.1. El archivo Manifest

Cada proyecto Android debe contener un archivo llamado `AndroidManifest.xml` en la carpeta raíz de su jerarquía de carpetas. Este archivo permite establecer la estructura de la aplicación y su metainformación, así como sus componentes o sus requisitos.

Incluye un nodo por cada uno de los componentes de una aplicación (actividades, servicios, proveedores de contenidos, etc.) y mediante el uso de Intents y permisos determina cómo interactuan unos con otros o con otras aplicaciones. También incluye atributos para especificar la metainformación asociada a la aplicación, como su icono, por ejemplo.

Veamos como ejemplo el archivo `AndroidManifest` de un proyecto Android recién creado:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
   package="es.ua.jtech.android"
   android:versionCode="1"
   android:versionName="1.0">
   <uses-sdk android:minSdkVersion="9" />

   <application android:icon="@drawable/icon"
                  android:label="@string/app_name">
      <activity android:name=".NombreProyectoActivity"
                  android:label="@string/app_name">
         <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
         </intent-filter>
      </activity>
   </application>
</manifest>
```

El elemento raíz del fichero es `<manifest>`, cuyo atributo `package` tendrá como valor el nombre del paquete del proyecto. El atributo `versionCode` permite al desarrollador indicar el número de versión actual de la aplicación. Este valor lo usará el desarrollador para comparar entre versiones. Por otra parte, el atributo `versionName` contendrá la cadena que describirá la versión y que sí que se mostrará a los usuarios.

El elemento `<manifest>` contendrá otros elementos que definirán diferentes componentes de la aplicación, opciones de seguridad y requisitos. Uno de estos nodos es `<uses-sdk>`. Este elemento permite especificar, entre otras cosas, cuál debe ser la versión mínima del sistema Android que debe encontrarse instalada en el dispositivo donde se vaya a ejecutar la aplicación mediante el parámetro `minSdkVersion`. Es importante que le asignemos un valor a este parámetro. En caso contrario se asignará un valor por defecto y nuestra aplicación puede sufrir un error en tiempo de ejecución si se intenta acceder a un elemento de la API de Android no soportada. En este ejemplo el valor del atributo es 9, lo cual se corresponde con la versión de Android 2.3.1. Obsérvese que la versión del SDK no se corresponde con la versión de la plataforma y que no se puede derivar una de la otra. Para saber qué versión del SDK se corresponde con cada versión de Android

podemos consultar `http://developer.android.com/guide/appendix/api-levels.html`.

El archivo Manifest sólo puede contener un elemento `<application>`. Se utiliza para establecer la metainformación de la aplicación (el nombre de la aplicación, su icono asociado, etc.). En este ejemplo tanto el valor del atributo `icon` como el del atributo `label`, que hacen referencia respectivamente al icono y al nombre de la aplicación, hacen referencia a sendos recursos de la aplicación. Más adelante en esta sesión hablaremos de los recursos.

El elemento `<application>` deberá contener un elemento de tipo `<activity>` por cada actividad presente en nuestra aplicación. Su atributo `name` contendrá el nombre de la clase de la actividad. Esto es importante, porque intentar iniciar una actividad que no esté listada en este fichero hará que se produzca un error en tiempo de ejecución. Cada elemento `<activity>` podrá contener a su vez un elemento de tipo `<intent-filter>` que permita especificar a qué Intents puede responder la actividad. En este ejemplo utilizamos este campo para indicar que nuestra única actividad es además la actividad principal, y por lo tanto la que se deberá mostrar al iniciar la aplicación.

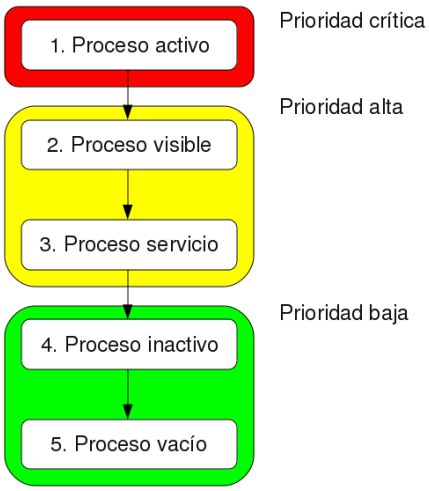
Conforme avancemos en éste y otros módulos iremos añadiendo nuevos elementos al archivo `AndroidManifest.xml`.

1.2.2. El ciclo de ejecución de una aplicación Android

Al contrario que en otros entornos, las aplicaciones Android tienen muy poco control sobre su propio ciclo de ejecución. Los componentes de una aplicación Android deberían estar atentos a los cambios producidos en el estado de la misma y reaccionar a los mismos como corresponda, estando especialmente preparados para el caso de una finalización repentina de la ejecución de la aplicación.

Por defecto cada aplicación Android se ejecutará en su propio proceso, cada uno con su propia instancia asociada de Dalvik (la máquina virtual de Android). Android administra sus recursos de manera agresiva, haciendo todo lo posible para que el dispositivo siempre responda a la interacción del usuario, lo cual puede llevar a que muchas aplicaciones dejen de ejecutarse de manera repentina, muchas veces incluso sin un aviso previo, con el objetivo de liberar recursos para aplicaciones de mayor prioridad. Estas aplicaciones de mayor prioridad suelen ser normalmente aquellas que están interactuando con el usuario en ese preciso instante.

El orden en el que los procesos de las aplicaciones son detenidos viene determinado por la prioridad de las mismas, la cual a su vez es equivalente a la prioridad de su componente de mayor prioridad. Cada uno de los estados en los que se puede encontrar una aplicación se resume en la siguiente figura y se detalla a continuación:



Ciclo de ejecución de los procesos en Android

- **Procesos activos:** son aquellos procesos que contienen aplicaciones que se encuentran interactuando con el usuario en ese preciso instante. Android liberará recursos para intentar que estos procesos activos siempre respondan sin latencia. Los procesos activos sólo serán detenidos como último recurso.
- **Procesos visibles**: procesos visibles pero inactivos, ya sea porque sus correspondientes aplicaciones se están mostrando detrás de otras o porque no están respondiendo a ninguna entrada del usuario. Esto sucede cuando una Actividad se encuentra parcialmente oculta por otra actividad, ya sea porque ésta última es transparente o no ocupa toda la pantalla. Estos procesos son detenidos tan solo bajo condiciones extremas.
- **Procesos asociados a servicios en ejecución**: los servicios permiten que exista procesamiento sin necesidad de que exista una interfaz de usuario visible. Debido a que estos servicios no interactúan directamente con el usuario, reciben una prioridad ligeramente inferior a la de los procesos visibles. Sin embargo se siguen considerando procesos activos y no serán detenidos a menos que sea estrictamente necesario.
- **Procesos inactivos**: se trata de procesos que albergan actividades que ni son visibles ni se encuentran realizando un procesamiento en este momento, y que además no están ejecutando ningún servicio. El orden en el que se detendrán estos procesos vendrá determinado por el tiempo que éstos llevan inactivos desde la última vez que fueron visibles, de mayor a menor.
- **Procesos vacíos**: son el resultado del intento de Android de retener aplicaciones en memoria una vez que éstas han terminado a modo de caché. Con esto se consigue que al lanzar de nuevo la aplicación se requiera menos tiempo.

1.3. Recursos

Se suele considerar una buena práctica de programación mantener todos los recursos de la

aplicación que no sean código fuente separados del propio código, como imágenes, cadenas de texto, etc. Android permite externalizar recursos de diversos tipos, no sólo los comentados anteriormente, sino que recursos más complejos como los `layouts`, o lo que es lo mismo, la especificación de la interfaz gráfica de las diferentes actividades.

Una ventaja adicional de externalizar los recursos es que se trata de un mecanismo simple para proporcionar valores diferentes a estos recursos dependiendo del hardware o del idioma del usuario. Si lo hacemos todo de manera correcta, será Android el encargado, al iniciar una actividad, de seleccionar los recursos adecuados.

1.3.1. Creación de recursos

Todos los recursos de la aplicación se almacenan bajo la carpeta `res/` del proyecto. Dentro de esta carpeta encontrarmos diferentes subcarpetas para distintos tipos de recursos.

Existen nueve tipos principales de recursos que tendrán su propia subcarpeta: valores simples, `Drawables`, `layouts`, animaciones, etilos, menús, `searchables`, XML y recursos `raw`. Al compilar nuestra aplicación estos recursos serán incluidos en el paquete *apk* que puede ser instalado en el dispositivo.

Durante el proceso de compilación se generará también una clase `R` que contendrá referencias a cada uno de los recursos. Esto nos permitirá referenciar a los recursos desde nuestro código fuente. En los ejercicios veremos ejemplos de esto. Por otra parte, a lo largo del curso iremos haciendo uso de algunos de estos recursos que se han comentado anteriormente. Iremos aprendiendo su uso sobre la marcha.

1.3.2. Unas pocas palabras sobre la creación de recursos para diferentes idiomas y configuraciones de hardware

En Android es posible preparar nuestra aplicación para poder ser ejecutada haciendo uso de diferentes idiomas o de configuraciones de hardware. Para ello definiremos archivos de recursos específicos. Android escogerá en tiempo de ejecución el archivo o archivos de recursos adecuados. Para conseguirlo definimos una estructura paralela de directorios dentro de la carpeta `res`, haciendo uso del guión `-` para indicar las diferentes alternativas de recursos que se están proporcionando. En el siguiente ejemplo se hace uso de una estructura de carpetas que permita tener valores por defecto para las cadenas, así como cadenas para el idioma francés y el francés de Canadá:

```
Project/
   res/
      values/
         strings.xml
      values-fr/
         strings.xml
      values-fr-rCA/
         strings.xml
```

Aparte de poder definir recursos para diferentes lenguajes utilizando especificadores como los que acabamos de ver (en, en-rUS, es, etc.), podría ser interesante conocer algunos otros referidos a hardware. En el siguiente listado se proporcionan algunos ejemplos:

- **Tamaño de pantalla:** `small` (para resoluciones menores que HVGA), `medium` (resoluciones al menos hasta HVGA pero menores que VGA) y `large` (para VGA o resoluciones mayores).
- **Orientación de la pantalla**: los valores pueden ser `port` (formato vertical), `land` (formato horizontal) o `square`.
- **Anchura/longitud de la pantalla**: usaremos `long` o `notlong` para recursos pensados especialmente para pantalla ancha (por ejemplo, para WVGA usaríamos `long` y para QVGA `notlong`).
- **Densidad de la pantalla**: medida en puntos por pulgada (*dots per inch* o *dpi*). Lo mejor es usar los valores `ldpi` para baja densidad (120dpi), `mdpi` para media densidad (160dpi) y `hdpi` para alta densidad (240dpi).

Aviso:

Si no se encuentra un directorio de recursos que se corresponda con la configuración del dispositivo en la que se está ejecutando la aplicación, se lanzará una excepción al intentar acceder al recurso no encontrado. Para evitar esto se debería incluir una carpeta por defecto para cada tipo de recurso, sin ninguna especificación de idioma, configuración de la pantalla, etc.

1.4. Actividades

Para crear las diferentes ventanas de interfaz de nuestra aplicación deberemos crear subclases de `Activity`. Cada actividad contendrá objetos de la clase `View` que permitirán mostrar los diferentes elementos gráficos de la interfaz gráfica así como añadir interactividad. En este sentido, se podría interpretar que cada actividad es como si fuera un formulario.

Deberemos añadir una nueva actividad por cada pantalla que queramos que pueda mostrar nuestra aplicación. Esto incluye la ventana principal de nuestra aplicación, la primera que se mostrará al iniciar nuestro programa, y desde la cual podremos acceder a todas las demás. Para movernos entre pantallas comenzaremos una nueva actividad (o volveremos a una anterior desde otra previamente ejecutada). La mayoría de las actividades están diseñadas para ocupar toda la pantalla, pero es posible crear actividades "flotantes" o semitransparentes.

1.4.1. Creando actividades

Para crear una nueva actividad añadimos a nuestra aplicación una subclase de `Activity`. Dentro de esta clase se deberá definir la interfaz gráfica de la actividad e implementar su funcionalidad. El esqueleto básico de una actividad se muestra a continuación:

```
package es.ua.jtech.android;

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

public class MiActividad extends Activity {
   /** Método invocado al crearse la actividad **/
   @Override
   public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
   }
}
```

Para añadir la actividad a la aplicación no basta con crear la clase correspondiente, sino que además deberemos registrarla en el Manifest. Para ello añadiremos un nuevo nodo `<activity>` dentro del elemento `<application>`. Esto es importante, porque una actividad que no haya sido incluida en el Manifest de esta forma no podrá ser mostrada por nuestra aplicación. Los atributos de `<activity>` permiten incluir información sobre su icono, los permisos que necesita, los temas que utiliza, etc. A continuación tenemos un ejemplo de este tipo de elemento:

```
<activity android:label="@string/app_name"
            android_name=".MiActividad">
</activity>
```

Como parte del contenido del elemento `<activity>` incluiremos los nodos `<intent-filter>` necesarios para indicar los `Intent` a los que escuchará nuestra aplicación y por lo tanto a los que reaccionará. Los `Intent` serán tratados más adelante, pero es necesario destacar que para que una actividad sea marcada como actividad principal (y por lo tanto, como la primera actividad que se ejecutará al inicarse nuestra aplicación) debe incluir el elemento `<intent-filter>` tal cual se muestra en el siguiente ejemplo:

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

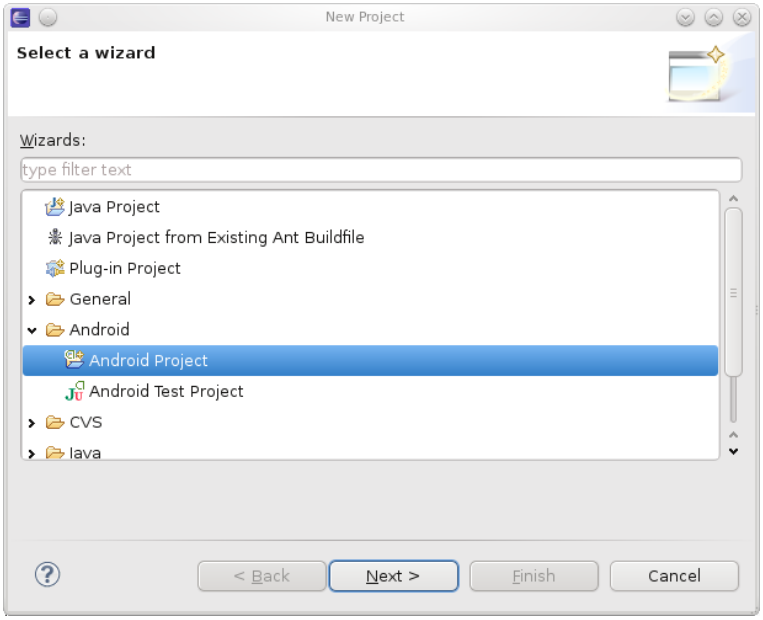
1.5. Nuestra primera aplicación

En esta sección veremos cómo unir todo lo visto anteriormente para crear nuestra primera aplicación Android. La aplicación constará de una única actividad, que mostrará un botón cuya etiqueta será un número. Cada vez que pulsemos el botón se incrementará el valor del número.

1.5.1. Creando el proyecto

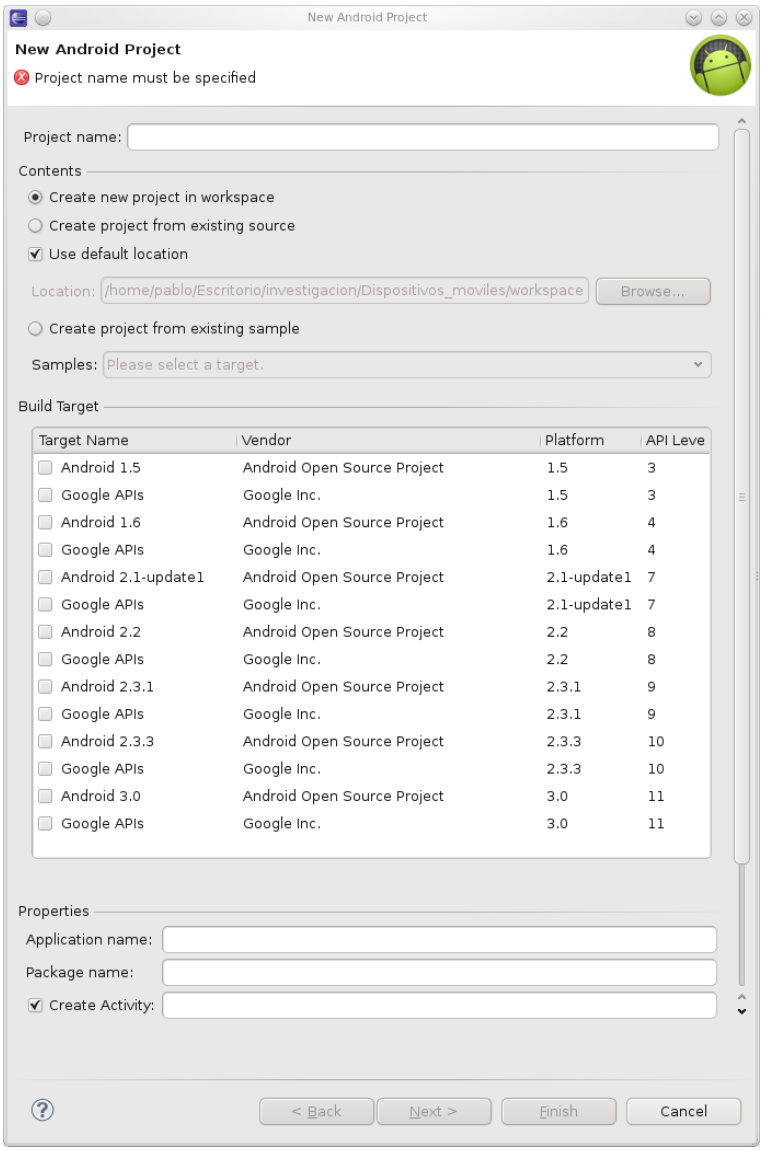
En primer lugar vamos a crear el proyecto. Para ello, en Eclipse, seleccionamos la opción `Project` dentro del submenú `New...` que podemos encontrar bajo el menú `File`. Entre

todos los tipos de proyecto a crear, seleccinamos `Android Project` dentro de la categoría `Android`.



Seleccionando el tipo de proyecto a crear (proyecto Android)

A continuación se mostrará una ventana en la que deberemos cuáles son las opciones del proyecto, como por ejemplo la versión del SDK de Android que utilizaremos, su nombre, su actividad principal, etc.



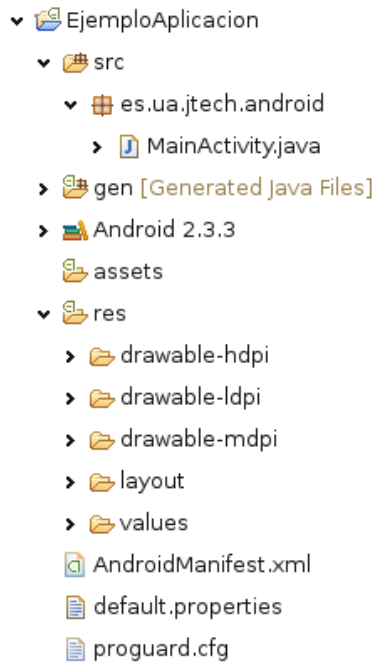
Dándole valor a los parámetros del proyecto

En esta ventana de opciones rellenaremos los siguientes campos (el resto de campos se dejarán con su valor por defecto):

- **Project name:** EjemploAplicacion
- **Build target:** Android 2.3.3
- **Application name:** Ejemplo de aplicación
- **Package name:** es.ua.jtech.android
- **Create activity:** MainActivity

Tras pulsar `Finish` se creará el proyecto `EjemploAplicacion` y podremos consultar su estructura de ficheros y directorios en el explorador de paquetes que aparece en la parte

izquierda de la interfaz de Eclipe. En el directorio raíz del proyecto tendremos el archivo `AndroidManifest.xml` con toda la información acerca de la aplicación. También veremos qeu se ha creado un paquete llamado `es.ua.jtech.android` dentro de la carpeta `src`. Dicho paquete contiene tan solo la definición de una clase, la de la actividad `MainActivity`. Obsérvese como también se ha generado la carpeta `res` que servirá para almacenar los recursos de la aplicación, y la carpeta `gen`, que contiene código autogenerado, y que por lo tanto nosotros no debemos tocar.



El contenido de nuestro nuevo proyecto

El archivo `AndroidManifest.xml` contiene el siguiente código:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
            package="es.ua.jtech.android"
            android:versionCode="1"
            android:versionName="1.0">
   <uses-sdk android:minSdkVersion="10" />

   <application android:icon="@drawable/icon"
                  android:label="@string/app_name">
      <activity android:name=".MainActivity"
                  android:label="@string/app_name">
         <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
         </intent-filter>
      </activity>

   </application>
</manifest>
```

Obsérvese el uso del elemento <uses-sdk>, con cuyo atributo android:minSdkVersion especifica la versión mínima del sistema Android que se le va a exigir al dispositivo móvil de destino para poder ejecutar la aplicación. El elemento <application> tiene dos atributos que permiten indicar el icono que se va a usar en la aplicación (atributo android:icon) y el nombre de la misma, el cual se mostrará en la pantalla del dispositivo (atributo android:label). El valor de ambos atributos tiene una sintaxis bastante peculiar. Esto es así porque lo que se está indicando es que el valor de dichos recursos se obtendrá a partir de los recursos de la aplicación, lo cual es algo que explicaremos un poco más adelante. Por último, el archivo Manifest indica que la aplicación constará de una única actividad, de nombre MainActivity, que además será la actividad principal (tal como se especifica por medio del uso del elemento <intent-filger>, tal como se ha explicado anteriormente).

1.5.2. Definiendo los recursos de la aplicación

Vamos ahora a echarle un vistazo al contenido de algunos de los recursos creados con la aplicación. Recuerda que todos los recursos se guardan bajo la carpeta res. Dentro de esta carpeta existe a su vez otra llamada values, y en su interior se almacena el archivo strings.xml. Este fichero sirve para almacenar las cadenas de caracteres (y otros elementos similares) que utilizaremos en nuestra aplicación, como por ejemplo las que se van a mostrar en la interfaz de las distintas actividades. El objetivo de disponer de un fichero strings.xml es poder crear aplicaciones independientes del idioma, de tal forma que podríamos disponer de diferentes carpetas *values* con sus correspondientes ficheros *strings.xml*, permitiendo al sistema operativo del dispositivo móvil escoger entre una u otra dependiendo del idioma en el que se encuentre configurado. El contenido del archivo strings.xml una vez creado el proyecto es el siguiente:

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

El archivo inicialmente creado contiene dos elementos de tipo <string>. El contenido de cada uno de esos elementos es una cadena de caracteres, y a cada una de ellas se le asigna un identificador mediante el atributo name. Fíjate en que el identificador de la segunda cadena, que se corresponde con el nombre que le dimos a la aplicación en las opciones del proyecto, tiene como identificador app_name. Y este es precisamente el identificador que se usaba en el archivo Manifest para indicar cuál era la etiqueta de nuestra actividad y nuestra aplicación. Así que, por ejemplo, cuando en el archivo Manifest estamos dándole al atributo android:lagel del elemento <activity> el valor @string/app_name, en realidad estamos indicando que queremos que el valor de dicho atributo sea la cadena de caracteres almacenada en el archivo strings.xml cuya etiqueta sea app_name.

La otra cadena se utiliza para indicar qué texto aparecerá en el cuadro de texto que se muestra por defecto en la actividad inicial creada cuando iniciamos un nuevo proyecto. Se

referencia en el archivo `main.xml` de la carpeta `layouts` en los recursos. De momento la ignoramos.

En la carpeta de recursos de la aplicación disponemos también de tres carpetas para almacenar elementos gráficos (elementos `drawable`) a tres diferentes resoluciones: alta (`drawable-hdpi`), media (`drawable-mdpi`) y baja (`drawable-ldpi`). Al crear el proyecto todas estas carpetas contienen un archivo con el mismo nombre: `icon.png`. Dicho archivo contendrá el icono de nuestra aplicación. En el archivo Manifest dábamos al atributo `android:icon` del elemento `<application>` el valor `@drawable/icon`. Esto significa que queremos que se utilice como icono de la aplicación la imagen almacenada con el nombre `icon` (obsérvese como se omite la extensión del fichero) en las carpetas `drawable`. Deberemos pues proporcionar a la aplicación tres copias de la misma imagen `icon.png` pero a tres resoluciones diferentes, que deberemos guardar en la carpeta correspondiente (`drawable-hdpi`, `drawable-mdpi` o `drawable-ldpi`). El sistema Android será el encargado de mostrar una imagen u otra según la resolución de la pantalla del dispositivo.

Un últimpo tipo de recursos que nos puede ser útil, sobre todo al principio del curso, son los `layouts`. Se trata de ficheros XML almacenados en la carpeta `layout` de los recursos de la aplicación, y que nos permitirán definir la disposición de las vistas en las interfaces gráficas de cada una de las actividades de la aplicación. Aprenderemos más sobre este tema en posteriores sesiones.

1.5.3. La actividad principal

Finalmente vamos a examinar el código de nuestra actividad principal, a la que hemos llamado `MainActivity`, y vamos a modificar dicho código para añadir la funcionalidad deseada a nuestra aplicación. El código de la actividad `MainActivity` es el siguiente:

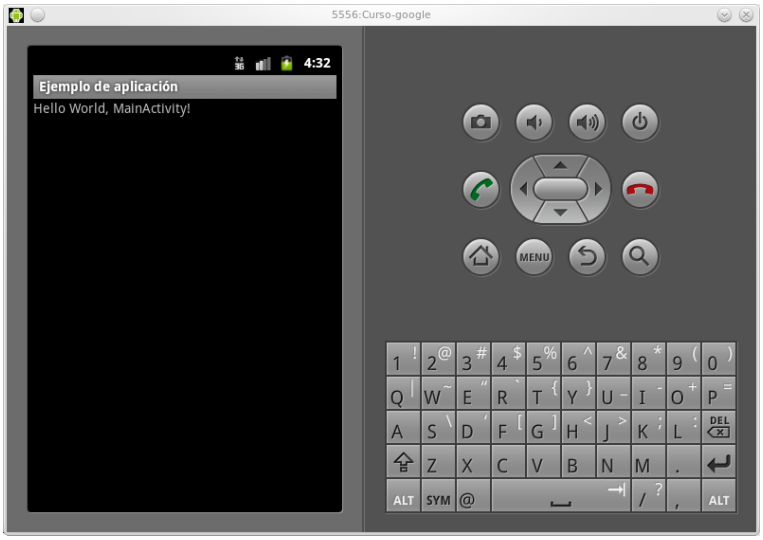
```
package es.ua.jtech.android;

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

public class MainActivity extends Activity {

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

La clase `MainActivity` hereda de la clase `Activity`, que entre otras cosas establece cómo debe ser el comportamiento de una actividad durante el ciclo de ejecución de la aplicación. El único método implementado inicialmente es una sobrecarga del método `onCreate()`, que se podría interpretar como un constructor: contiene el código que se ejecutará al crearse la actividad. De momento lo único que contiene es una llamada al método `onCreate()` de la clase padre, y una llamada al método `setContentView()` que

utiliza el archivo `main.xml` de la carpeta `layout` de los recursos para definir qué vistas tendrá la interfaz gráfica de la aplicación.

El parámetro de `setContentView()` es un **identificador de recurso**, y es un ejemplo de la forma que tendremos de referenciar a los diferentes recursos desde nuestro código Android. El objeto R se genera automáticamente por el SDK a partir de los recursos de la aplicación. Así pues, al utilizar el valor `R.layout.main` estamos accediendo al identificador asociado al archivo `main.xml` dentro de la carpeta `layout` de los recursos.

Si ejecutamos ahora nuestra aplicación, el aspecto de la misma será el que se muestra a continuación. Lo único que contiene la interfaz de la actividad `MainActivity` es un campo de texto (una vista de la clase `TextView`) que muestra la cadena del archivo `strings.xml` cuyo identificador es `hello`.



Aspecto de nuestra aplicación antes de realizar ninguna modificación

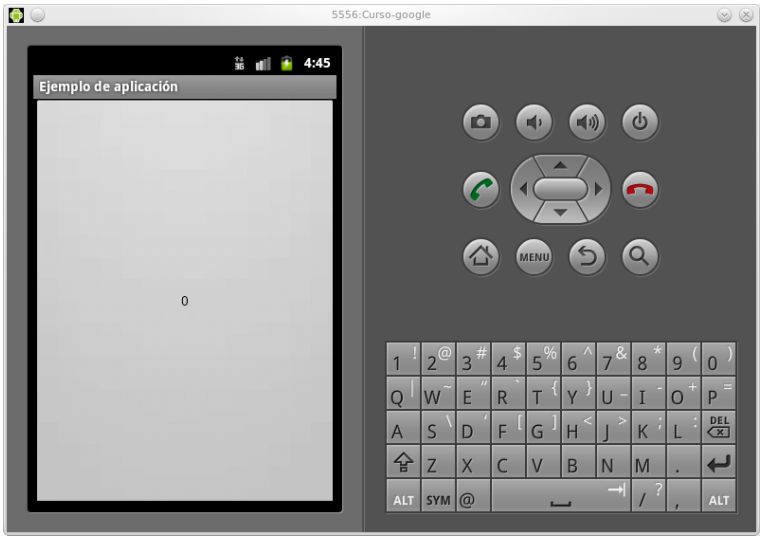
Veamos ahora cómo añadir un botón. Más adelante veremos una forma más correcta de hacerlo, pero de momento añadiremos el botón por medio de código. Nuestro método `onCreate()` debería quedar de la siguiente forma:

```
@Override
public void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   boton = new Button(this);
   boton.setText(R.string.texto_boton);
   setContentView(boton);
}
```

donde boton es un objeto de la clase `Button` definida como un atributo de la clase y donde `R.string.texto_boton` hace referencia a una cadena con identificador `text_boton` que habremos añadido al final del archivo `strings.xml` de los recursos de la aplicación:

```
<string name="texto_boton">0</string>
```

El aspecto que tendrá ahora nuestra aplicación será el siguiente:



Nuestra aplicación tras añadir un botón

Sólo queda añadirle funcionalidad al botón, de tal forma que cada vez que se pulse, se sume 1 al valor que contiene. Esto lo haremos añadiendo al botón un manejador del evento `onClick`:

```
boton.setOnClickListener(new OnClickListener() {
   public void onClick(View v) {
      int numero = Integer.parseInt(boton.getText().toString());
      numero++;
      boton.setText(new Integer(numero).toString());
   }
});
```

El código completo de la actividad `MainActivity` sería el siguiente:

```
package es.ua.jtech.android;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;

public class MainActivity extends Activity {
    Button boton;

        /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        boton = new Button(this);
        boton.setText(R.string.texto_boton);
        setContentView(boton);

        boton.setOnClickListener(new OnClickListener() {
                        public void onClick(View v) {
                                int numero =
```

```
Integer.parseInt(boton.getText().toString());
                                numero++;
                                boton.setText(new
Integer(numero).toString());
                            }
        });
    }
}
```

1.6. El ciclo de ejecución de una actividad

Terminamos esta primera sesión incidiendo nuevamente en la administración que lleva a cabo Android de sus diferentes elementos ejecutables. Hemos hablado anteriormente en esta sesión de la ejecución de aplicaciones; veamos ahora cómo se administra la ejecución de las diferentes actividades dentro de una aplicación.

Conforme se produce la ejecución de una determinada aplicación irá modificandose el estado de sus correspondientes actividades. El estado de una actividad servirá para determinar su prioridad en el contexto de su aplicación padre. Y esto es importante, porque hemos de recordar que la prioridad de una aplicación, y por lo tanto, la probabilidad de que dicha aplicación sea detenida en el caso en el que sea necesario liberar recursos del sistema, dependerá de cuál sea la de su actividad de mayor prioridad.

1.6.1. Pilas de actividades

El estado de cada actividad viene determinado por su posición en la pila de actividades, una colección de tipo `last-in-first-out` que contiene todas las actividades de la aplicación actualmente en ejecución. Cuando comienza una nueva actividad, aquella que se encontrara mostrándose en ese momento se mueve al tope de la pila. Si el usuario pulsa el botón que permite volver a la actividad anterior o se cierra la actividad que se estuviera mostrando en ese determinado momento, la actividad que se encontrara en el tope de la pila sale de ella y pasa a ser la actividad activa.

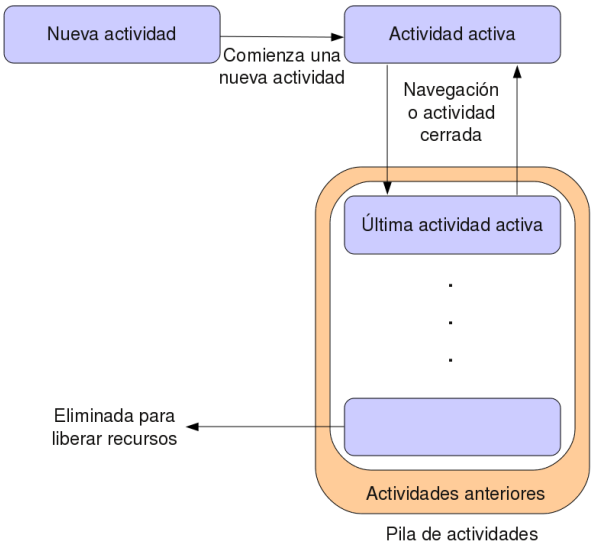


Diagrama de la pila de actividades

La pila de actividades será también usada por el sistema en el caso en el que se quiera determinar la prioridad de una actividad a partir de la prioridad de sus diferentes actividades.

Conforme las actividades se crean o destruyen van entrando o saliendo de la pila. Al hacerlo pueden ir transitando entre cuatro diferentes estados:

- **Activa**: se trata de la actividad que se está ejectuando en ese momento: es visible, tiene el foco de la aplicación y es capaz de recibir datos por parte del usuario. Android tratará por todos los medios de mantener esta actividad en ejecución, deteniendo cualquier otra actividad en la pila siempre que sea necesario.
- **En pausa**: se trata del estado en el que se encuentra una actividad cuando ésta está activa pero no dispone del foco. Este estado se puede alcanzar por ejemplo cuando se encuentra situada por debajo de otra transparente o que no ocupe toda la pantalla. Una actividad en pausa recibe el mismo tratamiento que una actividad activa, con la única diferencia de que no recibe eventos relacionados con la entrada de datos.
- **Detenida**: este es el estado en el que se encuentra una actividad que no es visible en ese momento. La actividad permanece en memoria, manteniendo toda su información asociada. Sin embargo, ahora la actividad podría ser escogida para ser eliminada de la memoria en el caso en el que se requieran recursos en otra parte del sistema. Por eso es importante almacenar los datos de una actividad y el estado de su interfaz de usuario cuando ésta pasa a estar detenida.
- **Inactiva:** una actividad estará inactiva si se ha terminado su ejecución o si todavía no se ha iniciado durante la ejecución de la aplicación. Las actividades inactivas han sido extraídas de la pila de actividades y deben ser reiniciadas para poder ser mostradas y utilizadas.

Todo este proceso debe ser transparente al usuario. No debería haber ninguna diferencia entre actividades pasando a un estado activo desde cualquiera de los otros estados. Por lo tanto puede ser interesante hacer uso de los diferentes manejadores de eventos relativos al cambio de estado de una actividad para almacenar los datos de la misma cuando pasa a estar detenida o inactiva y para volver a leerlos cuando ésta pasa a estar activa. Para manejar estos diferentes eventos podemos sobrecargar las siguientes funciones:

```
// Equivalente a un constructor
// Recibe un objeto conteniendo el estado de la interfaz de usuario
// guardada en la anterior llamada a onSaveInstanceState
public void onCreate(Bundle savedInstaceState)

// Se puede utilizar en lugar de la anterior durante el proceso
// de restaurar el estado de la interfaz de usuario
public void onRestoreInstanceState(Bundle savedInstanceState)

// Llamada cuando la actividad pasa a estar visible
public void onStart()

// Llamada antes de cualquier llamada a onStart, excepto la primera vez
public void onRestart()

// Cuando una actividad pasa a estar activa
public void onResume()

// Cuando una actividad deja de estar activa
public void onPause()

// Inmediatamente antes de llamar a onPause
public void onSaveInstanceState(Bundle savedInstanceState)

// Llamada cuando la actividad deja de estar visible
public void onStop()

// Equivalente a un destructor
public void onDestroy()
```

2. Introducción a Android - Ejercicios

2.1. Nuestra primera aplicación Android

En este primer ejercicio practicaremos los pasos vistos en clase para la creación de una nueva aplicación en Android, prestando especial atención al tema de los recursos, el cual puede ser descuidado si no se le presta la suficiente atención desde el principio.

- Creamos en primer lugar un proyecto Android cuyo nombre será **HolaMundo**. Escogeremos la versión 2.3.1 de la plataforma, le daremos a la aplicación el nombre *Hola Mundo*, y crearemos una actividad principal de nombre `HolaMundoActivity.java`, que se encontrará dentro del paquete `es.ua.jtech.android.holamundo`.
- Ejecuta la aplicación en el emulador y comprueba que funciona correctamente. Recuerda que el AVD creado debe tener instalada al menos la versión 2.3.1 de la plataforma para que nuestra aplicación pueda ser ejecutada.
- La interfaz de nuestra aplicación contiene una vista de tipo `TextView`, que mostrará el mensaje *Hello World, HolaMundoActivity!*. Dicha interfaz está definida en los recursos de la aplicación, dentro del archivo `main.xml` de la carpeta `/res/layout/`. Abre el fichero y mira la definición de la vista `TextView`. El elemento XML que la representa contiene un atributo `android:text` que indica el texto a mostrar en pantalla. El valor de ese atributo hace referencia a una cadena definida en el archivo `strings.xml` de la carpeta `/res/values/`. Cambia la cadena en dicho archivo para que el texto mostrado por pantalla sea *Hola Mundo*.
- Vamos a cambiar también el icono de nuestra aplicación. Cualquier proyecto nuevo en Android incluirá un icono por defecto definido en tres diferentes resoluciones, que podrá encontrarse en las carpetas `drawable-hdpi`, `drawable-mdpi` y `drawable-ldpi` dentro de la carpeta de recursos. En cada una de estas carpetas encontraremos un fichero llamado *icon.png* con la resolución correspondiente (alta, media y baja, respectivamente). El sistema escogerá qué icono mostrar en función de la resolución de la pantalla. En las plantillas de los ejercicios se han incluido tres archivos .png conteniendo un icono alternativo para nuestra aplicación. El icono se encuentra almacenado en tres carpetas a distintas resoluciones. El nombre de los archivos es *icono_alternativo.png*. Guarda los ficheros en las carpetas correspondientes dentro de los recursos de la aplicación, y modifica el archivo Manifest del proyecto para que se utilice ese nuevo icono.
- Comprueba con varios AVDs que hagan uso de diferentes resoluciones que el icono se muestra correctamente en el menú de aplicaciones.
- Ahora haremos los cambios necesarios para permitir a nuestra aplicación mostrar el texto en varios idiomas. Para ello creamos una nueva carpeta dentro de la carpeta de recursos a la que llamaremos *values-EN*. Dentro de dicha carpeta crearemos un nuevo fichero *strings.xml*. Este fichero será una copia de *values/strings.xml*, con la única diferencia de que el texto *Hola Mundo* será sustituido por *Hello World*. Al hacer esto,

nuestra aplicación mostrará el texto en inglés cuando el dispositivo esté configurado para mostrar los textos en ese idioma, y en español en el resto de casos.
- Prueba a cambiar la configuración de idioma de tu AVD y observa los resultados.

Una vez hecho todo lo anterior vamos a ver cómo podemos modificar el texto de la vista `TextView` desde el código fuente. Esto es algo que se verá en detalle en posteriores sesiones. Debemos seguir los siguientes pasos:

- En primer lugar añadimos una nueva función de nombre `devuelveEntero()` que no recibirá ningún parámetro y devolverá el entero 0.
- Para poder acceder al `TextView`, éste debe tener asignado un identificador. Añadimos el siguiente atributo al elemento `TextView` dentro del archivo `main.xml`:

```
android:id="@+id/texto"
```

- A partir de este momento podremos acceder al `TextView` en nuestro código por medio del identificador `R.id.texto`.
- Al final del método `onCreate()` creamos una variable para almacenar el objeto que representará el `TextView` en nuestro código y le añadimos el valor devuelto por la función `devuelveEntero()` utilizando el siguiente código:

```
TextView texto = (TextView)findViewById(R.id.texto);
texto.append(" " + new Integer(devuelveEntero()).toString());
```

Nota:
Los elementos de la interfaz gráfica de una actividad son accedidos en nuestro código mediante un identificador de recurso.

2.2. Depuración en Android por medio de LogCat

Entre las plantillas de la sesión encontrarás el proyecto Fibonacci. Este proyecto consta de una única actividad cuya interfaz gráfica está compuesta por un elemento de tipo `TextView` y dos botones. El `TextView` muestra los dos primeros elementos de la serie de Fibonacci. La serie de Fibonacci se define recursivamente de la siguiente forma:

- El elemento en la posición **n** se calcula como la suma de los elementos en las posiciones **n-1** y **n-2**
- Los dos primeros elementos de la serie valen ambos 1

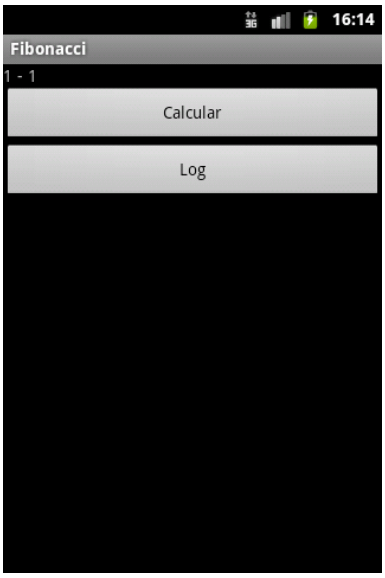


Diagrama de la pila de actividades

En primer lugar haremos los cambios necesarios para que cada vez que se pulse el botón superior se vayan añadiendo elementos de la serie de Fibonacci al `TextView`:

- Crea dos atributos de la clase `FibonacciActivity` para almacenar los dos términos anteriores de la serie de Fibonacci. Estos valores serán usados para calcular un nuevo término cuando se pulse el botón. Estos atributos serán de tipo entero, se llamarán `termino1` y `termino2`, y en ambos casos su valor inicial será 1.
- Para poder añadir un manejador para el botón superior primero hemos de asignarle un identificador en el archivo `main.xml` de la carpeta `layout`. Edita el archivo y añade el siguiente atributo al primero de los botones:

```
android:id="@+id/botonCalcular"
```

- Ahora ya podemos acceder al botón desde el código Java. Añade el siguiente código al método `onCreate()`:

```
Button botonCalcular = (Button)findViewById(R.id.botonCalcular);
botonCalcular.setOnClickListener(new OnClickListener() {
        public void onClick(View v) {
                // Manejador para el click del botón

        }
});
```

- Dentro del manejador del botón haz los cambios necesarios para añadir al texto mostrado por pantalla el siguiente término de la serie de Fibonacci. Cada vez que se pulse el botón se deberá añadir al `TextView` el siguiente elemento de la serie y actualizar los atributos `termino1` y `termino2` como corresponda.

A continuación tienes una captura de pantalla que muestra el estado de la aplicación tras haber pulsado unas cuantas veces el botón superior:

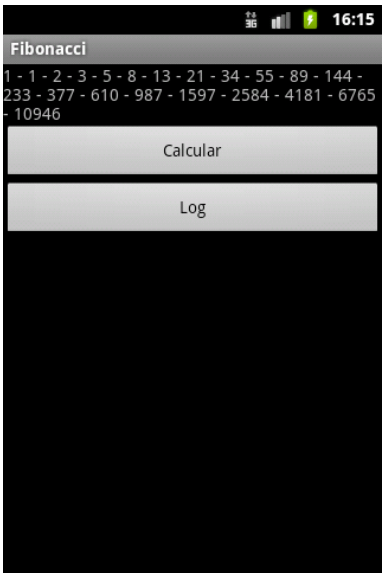


Diagrama de la pila de actividades

El siguiente paso será practicar el uso del sistema de logging de Android (*LogCat*) para depurar nuestras aplicaciones. *LogCat* consiste en un registro a través del cual el SDK va mostrando el estado de las aplicaciones ejecutadas en el dispositivo, excepciones, y otros mensajes. Dentro de Eclipse es posible visualizar el contenido de *LogCat* a partir de la pestaña situada en la parte inferior de la ventana. Si no aparece allí podremos hacer que se muestre de nuevo seleccionando la opción *Window->Show View->Other*, y dentro de la ventana que aparecerá a continuación, la opción LogCat dentro de la carpeta Android.

Para mostrar textos en *LogCat* deberemos hacer uso de alguno de los métodos estáticos proporcionados por la clase Log a tal efecto. Los más utilizados habitualmente son el método `e` y el método `i`. Ambos métodos reciben dos cadenas como parámetro. La primera de ellas se mostrará en el LogCat como un identificador de la aplicación que está lanzando el mensaje en el *LogCat*, mientras que la segunda cadena será el mensaje en sí mismo. La diferencia entre ambos métodos es que el primero se utiliza para lanzar mensajes de error, mientras que el segundo se utiliza para lanzar mensajes de información. Si se examina el *LogCat* desde la interfaz de Eclipse ambos tipos de mensaje se verán de diferente color.

Haremos los siguientes cambios a nuestra aplicación:

- Añadir un manejador para el botón inferior, siguiendo el ejemplo que se ha proporcionado previamente para el botón superior.
- Hacer que al pulsar dicho botón se muestre el valor de las variables `termino1` y `termino2` en el LogCat. Se puede usar cualquiera de los dos métodos vistos (`e` o `i`).
- Observa el funcionamiento de tu aplicación. Pulsa el botón inferior tras haber pulsado varias veces el botón superior y comprueba qué mensajes aparecen en el *LogCat*.

2.3. Estados de ejecución

Ya hemos visto que una actividad en Android puede pasar por diferentes estados de ejecución dependiendo de, por ejemplo, si se encuentra o no visible en ese momento. También hemos visto el concepto de pila de actividades.

En las plantillas de la sesión tienes el proyecto *EstadosEjecucion*, el cual incluye dos actividades: `Actividad1` y `Actividad2`. La actividad principal es `Actividad1`. Esta actividad contiene un botón. Vamos a hacer los cambios necesarios para que al pulsar dicho botón la actividad `Actividad2` pase a estar en primer plano. Para ello vamos a hacer uso de algunos conceptos que veremos en la siguiente sesión, así que de momento no es necesario tener un conocimiento exhaustivo de lo que estamos haciendo. Sigue los siguientes pasos:

- Crea un manejador para el botón, siguiendo los pasos vistos en ejercicios anteriores.
- Dentro del manejador del botón crea un objeto de la clase `Intent` de la siguiente forma:

```
Intent intent = new Intent(Actividad1.this, Actividad2.class);
```

- Los `Intent` son un elemento clave en las aplicaciones Android. Aprenderemos más sobre ellos en la siguiente sesión. Mientras tanto lo único que debéis saber es que necesitáis un `Intent` para llamar a una nueva actividad.
- Haz una llamada a `startActivity` dentro del manejador del botón, pasando como parámetro el `Intent` creado:

```
Actividad1.this.startActivity(intent);
```

- La llamada a `startActivity` hará que `Actividad2` pase a estar activa en primer plano. La actividad `Actividad1` pasará entonces a la pila de actividades, a la espera de que vuelva a estar visible. Si pulsamos le botón para volver atrás en nuestro dispositivo, la actividad `Actividad2` será destruida, mientras que la actividad `Actividad1` saldrá de la pila de actividades para volver a estar visible y activa.
- Ejecuta y prueba tu aplicación. Comprobarás que ésta no funciona correctamente. Haz los cambios necesarios en el archivo Manifest para que el problema desaparezca.

A continuación haremos uso de los `Toast`, otro mecanismo de comunicación entre la aplicación y el usuario. Mediante un `Toast` podremos mostrar un mensaje flotante sobre la aplicación, para avisar al usuario de que se ha producido algún evento. Para mostrar un Toast, deberemos hacer una llamada de la siguiente manera:

```
Context context = getApplicationContext();
CharSequence text = "Ejemplo de Toast";
int duration = Toast.LENGTH_SHORT; // También podría ser Toast.LENGTH_LONG

Toast toast = Toast.makeText(context, text, duration);
toast.show();
```

Nosotros usaremos `Toast` para comprobar el funcionamiento de los manejadores de

evento para el cambio de estado de las actividades. En concreto, tanto en `Actividad1` como en `Actividad2`, haremos que los siguientes manejadores muestren un `Toast`: `onCreate`, `onStart`, `onResume`, `onPause`, `onStop` y `onDestroy`. El `Toast` mostrará el nombre del método y de la actividad. Prueba la aplicación y navega entre las dos actividades, comprobando qué métodos entran en funcionamiento.

Aviso:

No olvides llamar al método correspondiente de la superclase en cada uno de los manejadores anteriores (por ejemplo, no olvides incluir dentro del método `onStart()` una llamada a `super.onStart()`.

2.4. Esperando el resultado de otra actividad (*)

En el ejercicio anterior hemos visto cómo lanzar una actividad desde cualquier otra mediante el método `startActivity`. Otra forma de lanzar actividades en Android es hacer uso del método `startActivityForResult`. La actividad que hace una llamada a `startActivityForResult` quedará a la espera de que la actividad llamada termine y devuelva un determinado resultado. Vamos a probarlo con una aplicación sencilla basada en el ejemplo del contador visto en clase.

En las plantillas de la sesión se incluye el proyecto `ActividadesResult`. Este proyecto consta de dos actividades, `ActividadPrincipal` y `ContarActivity`. La primera de ellas contiene una vista de tipo `TextView` que inicialmente muestra el valor cero, y un botón para lanzar la segunda actividad. La segunda actividad contiene el ejemplo visto en clase con un botón que muestra el número de veces que éste se ha pulsado. El objetivo del ejercicio es conseguir que al volver de la segunda actividad a la pimera mediante el uso del botón de volver atrás de nuestro dispositivo, se muestre en el `TextView` el número mostrado sobre el botón en la segunda actividad.

Los cambios a realizar serán los siguientes:

- Añade un manejador para el botón de la actividad `ActividadPrincipal`. Desde este manejador haremos uso de `startActivityForResult` para mostrar la actividad `ContarActivity`:

```
Intent intent = new Intent(ActividadPrincipal.this, ContarActivity.class);
// El segundo parámetro de la siguiente función
// es un identificador de petición
// Podría ser definido como una constante
startActivityForResult(intent, 1);
```

- Añade el siguiente código para que cambie el texto del `TextView` de la actividad principal al volver desde la segunda actividad:

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent
data) {
        super.onActivityResult(requestCode, resultCode, data);
        if(resultCode==RESULT_OK && requestCode==1){
```

```
                String msg = data.getStringExtra("contador");
                TextView texto = (TextView)findViewById(R.id.texto);
                texto.setText(msg);
        }
}
```

- El método `onActivityResult` es llamado cuando una actividad que fue comenzada con un `Intent` termina. Debemos comprobar antes de hacer nada que el evento se ha producido debido a la terminación de la actividad que se lanzó con un indicador de petición igual a 1 (que fue el valor que se pasó como segundo parámetro en nuestro `startActivityForResult`).
- Por último, en la actividad `ContarActivity`, añadimos el siguiente código dentro del manejador del evento del botón, tras la instrucción que cambia la etiqueta del botón. El código crea un Intent para que sea enviado a `ActividadPrincipal` en el caso en el que se vuelva a ella. Este código podría ser escrito en otra parte de la clase, pero lo dejaremos aquí de momento. Trataremos este tema en más detalle en la siguiente sesión.

```
Intent intent = new Intent();
intent.putExtra("contador", boton.getText());
setResult(RESULT_OK, intent);
```

- Prueba la aplicación. Si no has hecho ningún cambio más ésta no funcionará. Haz los cambios necesarios en el Manifest para que se solucione el problema.

3. Intents y navegación entre actividades

En esta segunda sesión hablaremos en primer lugar de un concepto muy importante en el desarrollo de aplicaciones Android: el de `Intent`. Veremos que entre sus múlitples usos se encuentra el de iniciar nuevas actividades, por lo que también le prestaremos especial atención a cómo se puede gestionar la forma en la que se navega entre las diferentes actividades de una aplicación, ampliando la información que se dio en la sesión anterior sobre este tema.

3.1. Intents

Los `Intents` son uno de los elementos más característico en el desarrollo de aplicaciones para Android. Los `Intents` permiten intercambiar datos entre aplicaciones o componentes de aplicaciones, como por ejemplo las actividades. También pueden ser usados para iniciar actividades o servicios. Otra posible aplicación de los `Intents` es solicitar al sistema que se realice una determinada acción con ciertos datos; el propio Android se encargará de buscar la aplicación más cualificada para realizar el trabajo.

Por lo tanto, los `Intents` se convierten un mecanismo para la transmisión de mensajes que puede ser utilizado tanto en el seno de una única aplicación como para comunicar aplicaciones entre sí. Los posibles usos de los `Intents` son:

- Solicitar que se inicie una actividad o servicio para llevar a cabo una determinada acción, pudiéndose añadir o no datos a la solicitud. Se trata de una solicitud implícita (no especificamos la actividad o servicio a iniciar, sino que la tarea que queremos que se lleve a cabo)
- Anunciar al resto del sistema que se ha producido un determinado evento (como cambios en la conexión a Internet o el nivel de carga de la batería), de tal forma que las actividades que estén preparadas para reaccionar ante determinado evento puedan realizar una determinada operación.
- Iniciar un servicio o actividad de manera explícita.

El uso de `Intents` es un principio fundamental en el desarrollo de aplicaciones para Android. Permite el desacoplamiento de componentes de la aplicación, de tal forma que cualquiera de ellos pueda ser sustituido fácilmente. También permite de manera simple extender la funcionalidad de nuestras aplicaciones, reutilizando actividades presentes en aplicaciones de terceros o incluso aplicaciones nativas de Android.

3.1.1. Usar Intents para lanzar actividades

Los `Intents` pueden ser utilizados para iniciar o navegar entre actividades. Para iniciar una actividad debemos hacer una llamada a la función `startActivity`, pasando como parámetro un `Intent`:

```
startActivity(intent);
```

El `Intent` puede o bien especificar de manera explícita el nombre correspondiente a la clase de la actividad que queremos iniciar, o bien indicar una acción que queremos que sea resuelta por una actvidad, sin especificar cuál. En este segundo caso el sistema escogerá la actividad a ejecutar de manera dinámica en tiempo de ejecución, buscando aquella que permita llevar a cabo la tarea solicitada de la manera más apropiada. Nuestra actividad no obtendrá ninguna notificación cuando la actividad recién iniciada finalice. Para ello deberemos utilizar otra función que veremos en más detalle más adelante.

Veamos en primer lugar como iniciar una Actividad de manera **explícita**. Para ello debemos crear un nuevo `Intent` al que se le pase como parámetro el contexto de la aplicción y la clase de la actividad a ejecutar:

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

Una llamada a la función `finish` (sin parámetros) en la nueva actividad o la pulsación del botón *BACK* de nuestro dispositivo hará que la nueva actividad se cierre y se elimine de la pila de actividades.

La otra forma de iniciar una actividad es mediante un `Intent` **implícito**, en el que se solicita que un componente anónimo de una aplicación sin determinar se encargue de satisfacer una petición concreta. Esto básicamente significa que somos capaces de solicitar al sistema que se inicie una actividad o servicio para realizar una acción sin necesidad de conocer previamente qué actividad, o incluso aplicación, se encargará de ello. Para crear un `Intent` implícito indicamos la acción a realizar y, opcionalmente, la URI de los datos sobre los que queremos que se lleve a cabo la acción. También es posible añadir datos adicionales al `Intent`, conocidos como *extras*.

Cuando se utiliza este método será el propio sistema el que, en tiempo de ejecución, decidirá qué actividad es la más adecuada para realizar la acción solicitada sobre los datos suministrados. Esto podría incluso permitir que nuestra aplicación use funcionalidades de otras, sin saber exactamente qué otra aplicación es la que nos está ayudando. Por ejemplo, para que se puedan hacer llamadas de teléfono desde nuestra aplicación podríamos o bien implementar una nueva actividad para ello, o utilizar un `Intent` implícito que solicite que se lleve a cabo la tarea de realizar la llamada a un número de teléfono representado por una URI:

```
Intent intent = new Intent(Intent.ACTION_DIAL, Uri.parse("tel:666666666");
startActivity(intent);
```

En el caso en el que existan varias actividades igualmente capaces de llevar a cabo la tarea se le permitirá escoger al usuario cuál de ellas utilizar por medio de una ventana de diálogo. Los componentes y aplicaciones nativos de Android tienen la misma prioridad que el resto, por lo que pueden ser completamente reemplazados por nuevas actividades qeu declaren que pueden llevar a cabo las mismas acciones.

3.1.2. Obtener información de subactividades

Una actividad iniciada por medio de la función `startActivity` es independiente de su actividad padre y por lo tanto no propocionará ningún tipo de información a ésta cuando finalice. En Android otra posibilidad es iniciar una actividad como una *subactividad* que esté conectada a su actividad padre. Una subactividad que finalice provocará siempre la activación de un evento en su actividad padre.

> **Nota:**
> Una subactividad no es más que una actividad que se ha iniciado de manera diferente. Cualquier actividad registrada en el *Manifest* de la aplicación puede ser iniciada como una subactividad.

Para lanzar una subactividad usaremos el método `startActivityForResult`, que funciona de manera muy parecida a `startActivity`, pero con una diferencia muy importante. Además de pasarle un `Intent` con una petición explícita o implícita de inicio de actividad, se le pasará también como parámetro un *código de petición*. Este código será un valor entero que será utilizado más tarde para identificar cuál de las subtareas es la que ha finalizado:

```
private static final int CODIGO_ACTIVIDAD = 1;

Intent intent = new Intent(this, MiOtraActividad.class);
startActivityForResult(intent, CODIGO_ACTIVIDAD);
```

Cuando la subactividad esté preparada para terminar, llamaremos a `setResult` antes de la llamada `finish` para devolver un resultado a la actividad padre. El método `setResult` requiere dos parámetros: el código de resultado y el propio resultado, que se representará mediante un `Intent`. Generalmente el código de resultado tendrá el valor `Activity.RESULT_OK` o `Activity.RESULT_CANCELED`, aunque podríamos utilizar nuestros propios códigos de resultado, ya que este primer parámetro puede tomar como valor cualquier número entero. El `Intent` devuelto a la clase padre contiene normalmente una URI que hace referencia a una determinada pieza de información y una colección de elementos extra usados para devolver información adicional. Veamos un ejemplo:

```
botonOk.setOnClickListener(new View.onClickListener() {
        public void onClick(View view) {
                Uri dato = Uri.parse("content://contactos/" +
id_contacto_seleccionado);

                Intent resultado = new Intent(null, dato);
                resultado.putExtra(DATOS_CORRECTOS, datosCorrectos);
                resultado.putExtra(TELEFONO_SELECCIONADO,
telefonoSeleccionado);
                setResult(RESULT_OK, resultado);
                finish();
        }
});

botonCancelar.setOnClickListener(new View.OnClickListener() {
        public void onClick(View view) {
                setResult(RESULT_CANCELED, null);
```

```
                finish();
        }
});
```

Como se puede observar es posible que una actividad devuelva diferentes resultados a la actividad padre desde diferentes puntos del código. En el caso de que la actividad se cierre debido a que se pulsó el botón *BACK* en el dispositivo móvil o en el caso en el que se haga una llamada a `finish` sin haberla hecho previamente a `setResult`, el código de resultado devuelto a la actividad padre será `RESULT_CANCELED` y el `Intent` devuelto contendrá el valor `null`.

El método `putExtra` de la clase `Intent` está sobrecargado. El primer parámetro será siempre una cadena que permita identificar el dato concreto, mientras que el segundo podrá ser una cadena, un carácter, un entero, un vector, etc.

Cuando una subactividad finaliza se ejecutará el manejador del evento `onActivityResult` de la clase padre. Deberemos sobrecargar este método para poder hacer algo con los datos devueltos por una subactividad. Dicho método recibirá tres parámetros, el código de petición que fue usado para lanzar la subactividad que acaba de finalizar, el código de resultado establecido por la subactividad para indicar su resultado antes de finalizar (recuerda que este valor puede ser un entero cualquiera, aunque lo más normal es que sea `Activity.RESULT_OK` o `Activity.RESULT_CANCELED`), y el `Intent` utilizado por la subactividad para empaquetar los datos a devolver. Este `Intent` puede incluir una URI que represente una determinada pieza de información y también una serie de elementos extra. A continuación se muestra un ejemplo de implementación de este método:

```
private static final int PRIMERA_ACTIVIDAD = 1;
private static final int SEGUNDA_ACTIVIDAD = 2;

@Override
public void onActivityResult(int requestCode,
                                                    int resultCode,
                                                    Intent data) {
        super.onActivityResult(requestCode, resultCode, data);

        switch(requestCode) {
                case PRIMERA_ACTIVIDAD:
                        if (resultCode == Activity.RESULT_OK) {
                                Uri dato = data.getData();
                                boolean datosCorrectos =
data.getBooleanExtra(DATOS_CORRECTOS, false);
                                String telefono =
data.getStringExtra(TELEFONO_SELECCIONADO);
                        }
                        break;
                case SEGUNDA_ACTIVIDAD:
                        if (resultCode == Activity.RESULT_OK) {
                                // Hacer algo
                        }
                        break;
        }
}
```

3.1.3. Responder peticiones de Intents

Como se ha comentado anteriormente, los `Intents` implícitos permiten solicitar la realización de una determinada acción, sin especificar explícitamente qué actividad deberá encargarse de llevarla a cabo. El sistema escoge la más adecuada. Algunas de las acciones que podemos especificar se indican a continuación. Muchas de ellas serán llevadas a cabo por actividades nativas del sistema:

- `ACTION_ANSWER`: abre una actividad que maneje llamadas de teléfono entrantes.
- `ACTION_CALL`: muestra un marcador de teléfonos e inmediatamente inicia una llamada usando el número indicado en la URI del `Intent`.
- `ACTION_DELETE`: inicia una actividad que permita eliminar los datos referenciados por la URI del `Intent`.
- `ACTION_DIAL`: muestra un marcador de teléfonos, con un número premarcado (aquel pasado mediante la URI del `Intent`).
- `ACTION_EDIT`: solicita la ejecución de una actividad que permita la edición de los datos referenciados por la URI del `Intent`.
- `ACTION_INSERT`: solicita la ejecución de una actividad que permita la inserción de nuevos elementos en el `Cursor` especificado por la URI. El concepto de `Cursor` será tratado más adelante en el curso.
- `ACTION_PICK`: lanza una actividad que permita escoger un elemento del Proveedor de Contenidos especificado por la URI del `Intent`. La aplicación seleccionada dependerá del tipo de dato que se desee escoger. Por ejemplo, usando `content://contacts/people` como URI hará que se lance la lista de contactos nativa del dispositivo. El concepto de Proveedor de Contenidos será explicado más adelante durante el curso.
- `ACTION_SEARCH`: lanza una actividad que permita realizar una búsqueda. El término a buscar deberá ser incluido en el `Intent` como un extra usando la clave `SearchManager.QUERY` como clave.
- `ACTION_SENDTO`: lanza una actividad que permita enviar un mensaje al contacto especificado mediante la URI del `Intent`.
- `ACTION_SEND`: lanza una actividad que sea capaz de enviar los datos especificados mediante el `Intent`.
- `ACTION_VIEW`: es la acción más común. Se solicita que los datos referenciados por la URI del `Intent` sean visualizados de la manera más adecuada posible.
- `ACTION_WEB_SEARCH`: abre una actividad que realiza una búsqueda en Internet basada en la URI del `Intent`.

En el caso en el que deseemos que nuestra actividad sea capaz de responder a un determinado tipo de `Intent` implícito, deberemos hacer uso de un *Intent Filter*. Los *Intent Filters* son utilizados como medio para registrar actividades en el sistema como capaces de realizar una determinada acción sobre unos datos concretos. Con un *Intent FIlter* se anuncia al resto del sistema que nuestra aplicación puede responder a peticiones de otras aplicaciones instaladas en el dispositivo.

Para registrar una actividad como manejador potencial de un determinado tipo de `Intent` añadimos un elemento `intent-filter` a su correspondiente nodo `activity` en el *Manifest* de la aplicación:

```
<activity android:name=".MiActividad" android:label="Mi Actividad">
        <intent-filter>
                <action
android:name="es.ua.jtech.intent.action.HAZ_ALGO"/>
                <category android:name="android.intent.category.DEFAULT"/>
                <category
android:name="android.intent.category.ALTERNATIVE_SELECTED"/>
                <data android:mimeType="vnd.miaplicacion.cursor.item/*"/>
        </intent-filter>
</activity>
```

Dentro del elemento `intent-filter` incluiremos los siguientes nodos:

- `action`: usa el atributo `android:name` para especificar el nombre de la acción que la actividad es capaz de realizar. Todos los *Intent Filter* deben tener un y sólo un nodo `action`. El nombre de una acción debería ser una cadena lo suficientemente descriptiva. Lo más correcto es utilizar un sistema de nombres basado en la nomenclatura de paquetes de Java.
- `category`: usa el atributo `android:name` para especificar bajo que circunstancias la actividad atenderá la solicitud. Cada elemento `intent-filter` puede contener múltiples elementos `category`. Los valores estándar de Android son los siguientes:
  - `ALTERNATIVE`: este valor indica que la actividad debería estar disponible como una alternativa de la acción por defecto realizada para el tipo de datos manejado. Por ejemplo, si la acción por defecto para un contacto es visualizarlo, nuestra actividad podría anunciar que es capaz de editarlo y presentarse como una alternativa para tratar ese tipo de dato. Esta alternativa se muestra normalmente en el menú de opciones de la actividad.
  - `SELECTED_ALTERNATIVE`: similar a la categoría anterior, pero para el caso de actividades que muestran un listado de elementos entre los que el usuario puede escoger, de tal forma que se pueda presentar una alternativa a la acción que se pueda realizar sobre el elemento seleccionado.
  - `BROWSABLE`: la acción sólo está disponible para un navegador web. Cuando un `Intent` se lance desde un navegador web siempre incluirá la categoría `BROWSABLE`.
  - `DEFAULT`: esto permite marcar una actividad como la acción por defecto para el tipo de datos especificado.
  - `GADGET`: esto indica que nuestra actividad puede ser ejecutada empotrada dentro de otra.
  - `HOME`: si especificamos esta categoría sin especificar una acción, estamos haciendo que la actividad represente una alternativa a la pantalla de inicio nativa.
  - `LAUNCHER`: usar esta categoría hará que nuestra actividad aparezca en el lanzador de aplicaciones de Android.
- `data`: este elemento permite especificar sobre qué tipos de datos puede actuar nuestra actividad. Se pueden incluir varios elementos de este tipo. Se puede utilizar cualquier

combinación de los siguientes atributos:

- `android:host` especifica un nombre de dominio válido.
- `android:mimeType` permite especificar el tipo de datos que nuestra actividad puede manejar.
- `android:path` específica una ruta válida para la URI.
- `android:port` especifica puertos válidos para el host indicado.

3.1.4. Acceder al Intent dentro de una actividad

Cuando una actividad es iniciada por medio de un `Intent`, ésta necesita conocer qué acción realizar y sobre qué datos. Para ello es necesario acceder al `Intent` con el que se hizo la petición. Una forma de hacer esto es utilizar del método `getIntent` (lo cual se hará normalmente dentro del método `onCreate`):

```
@Override
public void onCreate(Bundle icicle) {
        super.onCreate(icicle);
        setContentView(R.layout.main);

        Intent intent = getIntent();
}
```

Para acceder a la acción y a los datos asociados al `Intent` haremos uso de los métodos `getData` y `getAction`. Los elementos extra pueden ser obtenidos por medio a llamadas a las funciones `get[TIPO]Extra`, donde `[TIPO]` es un determinado tipo de datos (String, Boolean, etc.):

```
String accion = intent.getAction();
Uri datos = intent.getData();
```

3.2. Navegación

Ya vimos en la sesión anterior que una aplicación suele estar compuesta de varias actividades, e introdujimos el concepto de *pila de actividades*. Dicha pila está compuesta por todas las actividades de nuestra aplicación que se han ido ejecutando. Cada vez que iniciamos una actividad ésta pasa a estar activa y en primer plano, haciendo que la que lo estuviera hasta ese momento pasara a ocupar el tope de la pila. Una vez que la actividad en primer plano termina, se destruye, y aquella que estuviera en el tope de la pila la abandona para pasar a ser la actividad activa.

Sin embargo, también es posible que una actividad inicie otra que forme parte de una aplicación distinta. Por ejemplo, si nuestra aplicación desea tomar una fotografía, se define un `Intent` para solicitar la realización de dicha acción. A continuación, una actividad de cualquier otra aplicación instalada en el dispositivo y que ha declarado previamente que se puede ocupar de este tipo de tareas se inicia, pasando a primer plano. Una ves que se toma la fotografía, nuestra actividad vuelve a primer plano y continua su ejecución. Para el usuario la sensación es como si la actividad encargada de tomar la

fotografía formara parte de nuestra propia aplicación. A pesar de que las actividades en ejecución puedan pertenecer a diferentes aplicaciones, Android se encarga de navegar entre ellas de manera transparente al usiario manteniendo ambas actividades en una misma *tarea*.

Una **tarea** es una colección de actividades con las que el usuario interacciona para realizar una acción determinada. Cada tarea podría interpretarse como una aplicación independiente. Todas las actividades de una tarea se organizan en una pila de actividades. Toda tarea tiene, por lo tanto, su propia pila de actividades. Cuando el usuario selecciona el icono de una aplicación en el menú de aplicaciones, la tarea de dicha aplicación pasa a primer plano. Si no existiera una tarea para la aplicación, porque por ejemplo ésta no haya sido utilizada recientemente, se crea entonces una nueva tarea y la actividad principal de la aplicación pasa a estar asociada a la misma, activa y visible.

A partir de este momento las actividades de la aplicación se manejan mediante la pila de actividades tal como se explicó en la sesión anterior. Es realmente importante que recordemos cómo se realiza la navegación entre las actividades pertencientes a una tarea: cuando una actividad deja de estar activa pasa al tope de la pila; cuando la actividad activa termina, entonces la actividad en el tope pasa a estar activa.

Nota:

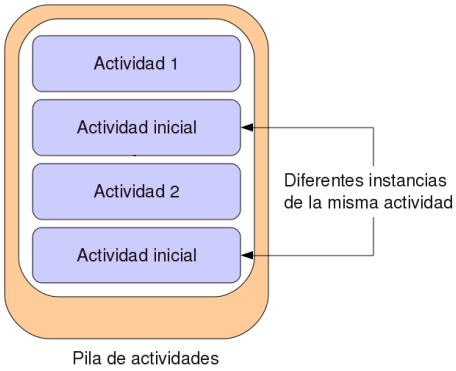
Lo anterior quiere decir que las actividades presentes en la pila nunca son reordenadas. Éstas son tan solo introducidas en el tope de la pila o extraídas del tope.

Una vez que todas las actividades son eliminadas de la pila de actvidades, la tarea correspondiente deja de existir. Una tarea se maneja como un bloque único que deja de estar en primer plano cuando el usuario inicia una nueva tarea o va a la pantalla de inicio de Android. Mientras la tarea no esté en primer plano, todas sus actividades están detenidas, pero en principio se mantienen intactas. De esta forma, cuando la tarea pasa a primer plano, el usuario puede seguir usándola como si nada hubiera pasado.

Nota:

Hemos de tener en cuenta que si el número de tareas que no se encuentra en primer plano crece, es posible que el sistema necesite eliminar algunas de sus actividades cuando se vea falto de recursos.

Debido a que nunca se reordenan las actividades en la pila de actividades, es posible que más de una instancia de una actividad se encuentre en ella. Esto puede suceder, por ejemplo, si más de una actividad de nuestra aplicación permite iniciar una actividad concreta. Cuando esto sucede, varias copias de la actividad se encuentran en la pila, por lo que cuando vamos pulsando el botón *BACK* de nuestro dispositivo cada instancia de dicha actividad se mostrará en el orden en el que fueron abiertas (cada una, por supuesto, con su propio estado de la interfaz). Este comportamiento está representado en la siguiente figura:



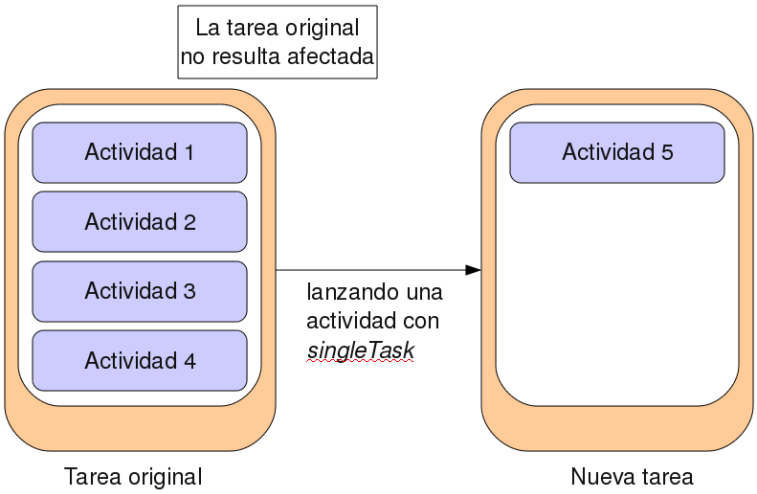
Ejemplo de varias instancias de una actividad en una pila

En esta sección veremos cómo modificar el comportamiento por defecto de Android a la hora de navegar entre actividades. Será posible, por ejemplo, hacer que una actividad sólo pueda ser instanciada una vez. O conseguir que una actividad se inicie en una nueva tarea en lugar de ser colocada en la tarea actualmente en ejecución. O incluso eliminar todas las actividades almacenadas en la pila de actividades cuando se lanza una determinada.

3.2.1. El atributo launchmode

Uno de los atributos del elemento `activity` en el *Manifest* de la aplicación es `launchmode`. Los diferentes valores de `launchmode` especifican cómo debería ser lanzada una actividad con respecto a su tarea. Los cuatro posibles valores de este atributo son:

- `standard`: se trata del modo por defecto. Cuando se lanza la actividad se crea una nueva instancia de la misma dentro de la tarea desde la que se inició y se le envía el `Intent`. La actividad puede ser instanciada múltiples veces; cada instancia puede estar en diferentes tareas, y también es posible disponer de varias copias de la actividad en una misma tarea.
- `singleTop`: si ya existía una instancia de la actividad en el tope de la pila de la tarea actual, se le envía el `Intent` a dicha instancia a través de una llamada a su método `onNewIntent`, en lugar de crear una nueva instancia. La actividad podría ser instanciada múltiples veces, tanto en diferentes como en la misma tarea; la única diferencia es que si la actividad ya se encontraba activa no se creará ninguna nueva copia de la misma. Esto evidentemente va a ser útil en el caso de actividades que puedan iniciarse a si mismas.
- `singleTask`: el sistema crea una nueva tarea y la nueva copia de la actividad pasa a ser la actividad activa en dicha tarea. Sin embargo, si ya existía una instancia de la actividad en una tarea separada, en lugar de crear una nueva copia se le envía a la ya existente el `Intent` por medio de una llamada a su método `onNewIntent`. Sólo puede existir una instancia de la actividad. Se podría interpretar el resultado como si se hubiera iniciado una nueva aplicación independiente de la anterior.



Ejemplo de singleTask

- `singleInstance`: el comportamiento es el mismo que en el caso de `singleTask`, con la diferencia de que el sistema nunca lanza ninguna otra actividad en la tarea que contiene la instancia recién creada. La actividad será siempre el único miembro de su tarea. Todas las actividades iniciadas por ésta se abrirán en una tarea nueva. Este valor se podría utilizar por ejemplo para una actividad que fuera a cumplir el papel de navegador web y que fuera a ser notificada con `Intents` desde diferentes tareas. Sólo existiría una copia de este navegador en la mamoria, que sería una aplicación totalmente independiente sin ninguna actividad adicional, y que en su historial incluiría todas las páginas que se hubieran visualizado a partir de diferentes aplicaciones.

Ejemplo de singleInstance

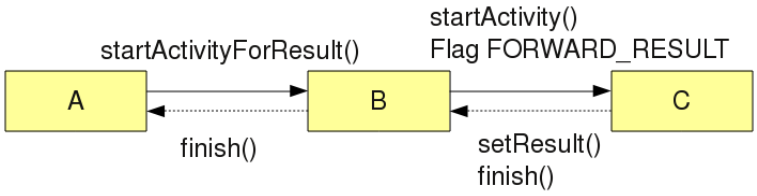
Nota:

Todos los comportamientos indicados anteriormente pueden ser sustituidos por otros cuando se añaden ciertos flags al `Intent` que lanza la actividad, los cuales se verán en la siguiente sección.

3.2.2. Modificar el lanzamiento de actividades mediante Intents

Además de mediante el método visto en la sección anterior, es posible modificar la forma en la que una actividad se asocia a una determinada tarea incluyendo un determinado flag en el `Intent` pasado como parámetro al método `startActivity`. Los flags que se pueden utilizar son los siguientes:

- `FLAG_ACTIVITY_NEW_TASK`: produce el mismo comportamiento que asignar el valor `singleTask` al atributo `launchMode`.
- `FLAG_ACTIVITY_SINGLE_TOP`: produce el mismo comportamiento que al asignar el valor `singleTop` al atributo `launchMode`.
- `FLAG_ACTIVITY_CLEAR_TOP`: si ya existe una copia de la actividad que se quiere ejecutar en la tarea actual, en lugar de lanzar una nueva copia se destruyen todas las actividades sobre ella en la pila de actvidades, y se le envía el `Intent` a dicha copia por medio de su método `onNewIntent()`.
- `FLAG_ACTIVITY_REORDER_TO_FRONT`: si existe una instancia de la actividad en la pila la llevará al primer plano haciéndola activa, sin necesidad de crear una nueva instancia de la misma.
- `FLAG_ACTIVITY_EXCLUDE_FROM_RECENTS`: cuando se lanza una actividad en una nueva tarea, ésta normalmente se suele mostrar en la lista de actividades recientemente inciadas, a la cual se puede acceder mediante una pulsación larga del botón *HOME*. Usar este flag evitará que se produzca este comportamiento.
- `FLAG_ACTIVITY_FORWARD_RESULT`: ya sabemos que para que una subactividad pueda comunicarse con su actividad padre por medio del método `setResult` es necesario que ésta haya hecho uso del método `startActivityForResult`. Supongamos ahora el caso mostrado en la siguiente figura, en el que la actividad A inicia la actividad B, que a su vez inicia la actividad C. Si quisiéramos que el resultado devuelto de C a B por medio de `setResult` fuera también devuelto a A, deberíamos hacerlo manualmente volviendo a incluir una llamada a `setResult` en B. Para evitar tener que hacer esto podemos usar este flag. Para que esto suceda la actividad B debe haber sido lanzada con este flag y además **debe haber lanzado a la actividad C con `startActivity`** (y no `startActivityForResult`).



Ejemplo de FLAG_ACTIVITY_FORWARD_RESULT

- `FLAG_ACTIVITY_MULTIPLE_TASK`: este flag no tiene efecto a menos que se utilice conjuntamente con `FLAG_ACTIVITY_NEW_TASK`. Con este flag siempre se crea una nueva instancia de la actividad que será colocada en el tope de la pila de una nueva tarea.

- `FLAG_ACTIVITY_NO_ANIMATION`: este flag desactiva la animación que normalmente se suele mostrar cuando se pasa de una actividad a otra. Esto podría ser utilizado por ejemplo en el caso de lanzar una actividad que a su vez va a lanzar otra sin ninguna interacción por parte del usuario. El resultado sería que se produciría una única animación de transición entre actividades en lugar de dos.

3.2.3. Afinidad entre actividades

La afinidad es un medio para indicar a qué tarea prefiere pertenecer una actividad. Todas las actividades de una misma aplicación tienen por defecto afinidad con el resto. Así que por defecto todas las actividades de una misma aplicación prefieren pertenecer a la misma tarea. Sin embargo, este comportamiento se puede modificar, de tal forma que actividades definidas en diferentes aplicaciones puedan compartir afinidad o que actividades definidas en una misma aplicación puedan tener diferentes afinidades y por lo tanto asignarse a diferentes tareas. Para ello se debe hacer uso del atributo `taskAffinity` del elemento `activity` en el *Manifest* de la aplicación.

El valor del atributo `taskAffinity` debe ser una cadena, que debe ser diferente del nombre del paquete por defecto definido en el elemento `manifest` dentro del *Manifest* de la aplicación. Esto es así porque ese nombre es el que la aplicación que utiliza por defecto para definir la afinidad de sus tareas.

La afinidad entre actividades entra en juego cuando se presenta cualquiera de las siguientes circunstancias:

- Cuando el `Intent` que inicia una nueva actividad contiene el flag `FLAG_ACTIVITY_NEW_TASK`. Cuando esto sucede normalmente se crea una nueva tarea para la actividad, pero esto no tiene por qué ser así. Si ya existía una tarea en la que sus actividades comparten la misma afinidad que la nueva actividad, ésta se inicia en dicha tarea. En otro caso se crea una nueva tarea. Hemos de tener cuidado con esto. Si al usar este flag se crea una nueva tarea para nuestra actividad, deberemos proporcionar algún medio para poder volver a mostrar la actividad en el caso en el que el usuario vuelva a la pantalla inicial de Android por medio del botón correspondiente de su dispositivo. Una forma de hacerlo sería por ejemplo incluir un icono en alguna parte para poder volver a ella.
- Cuando el atributo `allowTaskReparenting` de la actividad tiene el valor `"true"`. En este caso la actividad podría moverse de la tarea que inicia a la tarea por la que tenga afinidad, cuando esta tarea pase a primer plano.

Nota:

Si un paquete `.apk` contiene más de una *aplicación* desde el punto de vista del usuario, quizá sea interesante hacer uso de `taskAffinity` para asignar diferentes tareas a las actividades correspondientes a cada *aplicación*

3.2.4. Limpiar la pila de actividades

Si una tarea deja de estar en primer plano durante un periodo prolongado de tiempo, el sistema eliminará todas las actividades de la tarea excepto la actividad que deba pasar a activa cuando la tarea vuelva al primer plano. Por lo tanto, cuando se vuelva a activar la tarea, tan sólo se restaura dicha actividad. Esto es así porque el sistema supone que tras estar mucho tiempo sin utilizar una tarea probablemente se haya abandonado para hacer cualquier otra cosa. Sin embargo, existen ciertos atributos del elemento `activity` en el *Manifest* de la aplicación que permiten modificar este comportamiento:

- `alwaysRetainTaskState`: si se asigna el valor `true` a este atributo para la actividad en el tope de la pila de la tareas, el proceso descrito anteriormente nunca tiene lugar. La tarea mantiene todas sus actividades incluso tras un largo periodo de tiempo.
- `clearTaskOnLaunch`: si se asigna el valor `true` a este atributo para la actividad en el tope de la pila de tareas, se produce el efecto contrario que en el caso del atributo anterior; tan pronto como la tarea deje de estar en primer plano se eliminarán todas las actividades de la pila excepto aquella en el tope de la misma. Cada vez que se abandona la tarea y se vuelve a ella ésta se encontrará en su estado inicial, incluso si ha sido sólo durante un corto periodo de tiempo.
- `finishOnTaskLaunch`: este atributo cumple una función parecida a la de `clearTaskOnLaunch`, pero operando a nivel de una sóla actividad y no de toda la tarea. Si su valor es `true`, tan pronto la tarea deje de estar en primer plano se destruirá la actividad.
- `noHistory`: este atributo también se aplica a una actividad individual, y es muy parecido al anterior, pero no igual. En este caso, si su valor es `true` para una determinada actividad, la actividad se destruirá *cuando deje de estar en primer plano* (la actividad, no su tarea). Esto quiere decir que la actividad nunca llegará a estar en la pila de tareas.

3.3. Esquemas típicos de navegación

Una vez estudiados todos los conceptos anteriores, describimos brevemente en esta sección algunos esquemas básicos de navegación en Android, viendo que sucede cuando pulsamos los botones *HOME*, *BACK* o cambiamos de una aplicación a otra. Esto nos permitirá comprender mejor cómo debemos plantear la navegación en nuestras aplicaciones y que consideraciones debemos tomar a la hora de diseñarlas.

3.3.1. Iniciando una aplicación desde la pantalla inicial de Android

La pantalla inicial de Android es el punto a partir del cual podremos lanzar muchas de nuestras aplicaciones (algunas aplicaciones sólo pueden ser lanzadas desde otras). Cuando el usuario pulsa sobre un icono en el lanzador de aplicaciones (o sobre su correspondiente icono en la pantalla inicial), se lanza la actividad principal de dicha

aplicación; ésta se pone en primer plano y puede captar la entrada del usuario. La actividad correspondiente a la pantalla de inicio permanecerá en segundo plano, detenida, a la espera de que el usuario pulse la tecla *HOME* en su dispositivo.

3.3.2. Abandonar una actividad con los botones BACK y HOME

El efecto que abandonar una actividad tendrá sobre ésta dependerá de cómo lo haga el usuario. Por ejemplo, pulsar el botón *BACK* del dispositivo hará que por defecto la actividad en primer plano se destruya y se muestre la anterior en la pila de actividades correspondiente. En el caso en el que la pila quede vacía, se elimina la tarea asociada y se muestra la primera de las actividades de la siguiente tarea. En el caso en el que no hubiera ninguna otra tarea en ejecución simplemente se vuelve a mostrar la pantalla de inicio de Android. Esto quiere decir que si cerramos una aplicación por medio del botón *BACK*, ésta volverá a ser mostrada en su estado inicial si se vuelve a lanzar, porque fue destruida.

Sin embargo, si se pulsa *HOME* en lugar de *BACK*, se vuelve a la pantalla incial, pasando la actividad que estuviera activa a segundo plano pero **sin destruirla**. El efecto que producirá ejecutar de nuevo la aplicación será simplemente volver a llevar a primer plano la tarea correspondiente, con lo que la actividad correspondiente se volvería a mostrar en el estado en el que se encontrara.

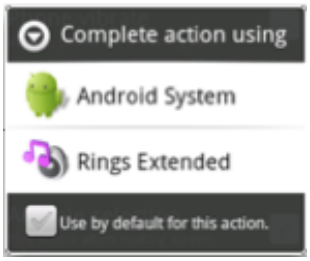
Esto permite la posibilidad de ejecución **multitarea** en Android. Esto se produce si al volver a la pantalla inicial de Android por medio del botón *HOME* se decide lanzar una aplicación distinta a la que ha quedado fuera del primer plano. Cada aplicación sería lanzada en una tarea diferente. Ahora seríamos perfectamente capaces de pasar de una aplicación a la otra, pasando las tareas correspondientes de primer a segundo plano y viceversa.

Existen excepciones para este comportamiento. Algunas actividades se vuelven a mostrar en su estado inicial tras volverlas a colocar en primer plano tras haber dejado de estar activas previamente. Esto puede pasar por ejemplo con los contactos. Si se selecciona un contacto para ver sus detalles y se cierra la aplicación de contactos, al volver a iniciarla se mostrará de nuevo la lista de contactos, y no los detalles del contacto que se estuviera consultando en el momento de cerrar la aplicación. Por otra parte, no todas las actividades tienen que ser destruidas tras pulsar el botón *BACK*. Por ejemplo, en el caso de un reproductor de música, éste podría seguir en funcionamiento en segundo plano mientras la música sigue sonando aunque se hubiera pulsado el botón *BACK*.

3.3.3. Reutilizar y reemplazar actividades

Cuando una actividad A lanza una actividad B perteneciente a una aplicación distinta, diremos que A reutiliza B. Esto sucede como ya hemos visto cuando A no es capaz de realizar una determinada tarea y confía en B para ello. A la hora de diseñar una actividad es importante dedicar un tiempo a pensar cómo podría reutilizar otras o cómo funcionaría a la hora de ser reutilizada por actividades de terceros.

El caso contrario al anterior se conoce como reemplazar una actividad. Esto sucede cuando una actividad A está más capacitada para llevar a cabo una tarea que otra actividad B. En este caso A y B tienen un nivel de equivalencia tal que A podría ser utilizada en lugar de B. Esto contrasta con el caso anterior en el que A y B son actividades totalmente diferentes que se complementan. Por ejemplo, supongamos que el usuario se ha descargado una aplicación que puede sustituir a la aplicación nativa para los tonos de teléfono. Cuando el usuario intenta acceder a la configuración de los tonos de teléfono a través de las opciones de configuración, verá un cuadro de diálogo que le permitirá escoger entre esta nueva aplicación o la nativa de Android, con una opción para recordar la opción escogida en el futuro.



Reemplazo de actividades

4. Intents y navegación entre actividades - Ejercicios

4.1. Intents implícitos

En este primer ejercicio vamos a modificar una aplicación que se proporciona en las plantillas de la sesión y que se llama *Implicitos*. La interfaz de la actividad principal de esta aplicación está compuesta por tres botones. El objetivo es que al pulsar cada uno de ellos se lance un `Intent` implícito que sea atendido por alguna de las actividades nativas del sistema.

Interfaz de la aplicación Implicitos

Crea un manejador para todos los botones, de tal forma que cada uno de ellos se encargue de solicitar la realización de la acción indicada. No olvides inicializar los `Intents` con los datos requeridos para que se pueda llevar a cabo la tarea solicitada. Los datos que necesitarás para cada tipo de acción son los siguientes:

- `ACTION_DIAL`: una URI correspondiente a un teléfono. Se puede construir a partir de una cadena con la sintaxis `"tel:[TELÉFONO]"` (donde [TELÉFONO] es un número de teléfono) mediante el método `Uri.parse(String)`.
- `ACTION_WEB_SEARCH`: un parámetro extra que será una cadena, cuyo identificador será `SearchManager.QUERY` y cuyo contenido será el texto a buscar en Google.
- `ACTION_VIEW`: en este caso utilizaremos la acción para visualizar nuestra lista de contactos. Para ello debemos añadir al `Intent` la URI `content://contacts/people/`.

Nota:

Cuando estés ejecutando el programa, abre el historial de tareas con una pulsación larga de la tecla *HOME*. ¿Puedes ver en este listado alguna de las actividades lanzadas para resolver tus peticiones?

4.2. Intents explícitos

Se te proporciona un proyecto en las plantillas de la sesión de nombre *Explicitos*. La aplicación consta de dos actividades: *Principal* y *Secundaria*. La actividad *Principal* contiene en su interfaz cuatro botones. Al lado de cada uno de estos botones hay un elemento de tipo `TextView`, cuyo valor inicial es cero. Sea cual sea el botón que se pulse se mostrará siempre la actividad *Secundaria*. Esta actividad leerá del `Intent` recibido el valor numérico situado al lado del botón a partir del cual se lanzó desde *Principal* y le sumará uno, mostrándose en la pantalla del dispositivo. Al pulsar el botón de *Secundaria* se deberá volver a *Principal*. En *Principal* se actualizará el valor a la derecha del botón que se pulsó para llamar a *Secundaria*, tomando el valor que se estaba mostrando en dicha actividad antes de pulsar su botón *Volver*.

La siguiente figura muestra un ejemplo de uso, de izquierda a derecha y de arriba a abajo. En primer lugar se pulsa el segundo botón, con lo que se muestra la actividad *Secundaria* mostrando como valor 1 (ya que el valor a la derecha del segundo botón era 0). Cuando pulsamos en volver, el `Intent` de respuesta de *Secundaria* devuelve ese valor 1 que *Principal* utiliza para actualizar el valor a la derecha del segundo botón. Luego se repite el proceso pulsando el tercer botón, y por último se muestra el resultado al volver a utilizar el segundo botón.

Ejemplo de uso de la aplicación de Intents explícitos

Nota:

Comprueba que ocurre al volver a lanzar la actividad tanto si vuelves a la pantalla de inicio de Android por medio de la tecla *BACK* como si lo haces por medio de la tecla *HOME* de tu dispositivo.

Nota:

Para cambiar el texto mostrado por un elemento de tipo `TextView` debes hacer uso de su método `setText`, que recibe como parámetro un dato de tipo `CharSequence`.

4.3. Navegación

Otro proyecto que te proporcionamos con las plantillas de la sesión es *Navegacion*. Este proyecto consta de cuatro actividades: *A*, *B*, *C* y *D*. Cada actividad contiene un botón etiquetado con el nombre del resto de actividades de la aplicación. Vamos a modificar la manera en la que se navega entre estas actividades, de tal forma que vamos a suponer que existe una actividad principal (actividad *A*) y una serie de actividades secundarias (actividades *B*, *C* y *D*). Cuando se vuelva a la actividad *A* desde cualquiera de las demás, se deberá conseguir que dicha actividad sea la única que quede en la pila. Sigue los siguientes pasos:

- Añade manejadores de evento para los botones de la actividad *A*, de tal forma que

cada uno de ellos lance la actividad con la que está etiquetado dicho botón.
- Repite estos mismos pasos con las actividades *B*, *C* y *D*.
- Haz los cambios necesarios para que nunca haya más de una instancia de la actividad *A* en la pila de actividades. Investiga cual de los flags debes añadir al `Intent` a la hora de lanzar la actividad.
- Haz los cambios necesarios para que al pulsar el botón *BACK* estando en la actividad *B*, *C* o *D* se vuelva siempre a la actividad *A*. Al pulsar el botón *BACK* desde la actividad *A* se saldrá de la aplicación. Esto quiere decir que durante la ejecución de la aplicación su tarea contendrá o tan sólo una actividad (la actividad *A*) o como mucho dos (la actividad *A* y cualquiera de entre las otras tres).

Nota:

Para poner a uno algún flag de un objeto de la clase `Intent` hacemos uso del método `setFlags`, que recibe como parámetro uno o más identificadores de flag separados por el operador `|` (*OR* binario)

4.4. Actividades en diferentes tareas (*)

Se te proporciona el proyecto *Tareas* con dos actividades: *Principal* y *Web*. La actividad *Principal* contiene un botón que servirá para lanzar la actividad *Web*. La actividad *Web* contiene una vista de tipo *WebView*, que trataremos en sesiones posteriores, y que se encarga de mostrar contenido web.

Sigue los siguientes pasos:

- Haz los cambios necesarios para que se pueda lanzar la actividad *Web* desde la actividad *Principal*.
- Lanza la aplicación y pulsa el botón para mostrar la actividad *Web*. Pulsa ahora el botón *HOME* del dispositivo, con lo que volverás a la pantalla principal de Android. A continuación mantén pulsado el botón *HOME* hasta que aparezca el listado de tareas recientes. Selecciona *Tareas*; al hacerlo se deberá mostrar de nuevo la actividad *Web*. Para salir de la aplicación deberás pulsar dos veces el botón *BACK* del dispositivo. Ese es el comportamiento normal de una aplicación Android.
- Haz los cambios necesarios para que cuando se lance la actividad *Web* lo haga en una tarea nueva. Además, debes conseguir que sea la única actividad de esta nueva tarea. Escoge el valor adecuado para el atributo *launchMode*.
- Vamos a repetir el experimento anterior. Ejecuta la aplicación, y pulsa el botón para lanzar la actividad *Web*. Ahora pulsa *HOME*, y vuelve a enviar *Tareas* a primer plano seleccionándo la aplicación en el listado que aparece al mantener durante un tiempo pulsada la tecla *HOME*. Verás que en esta ocasión también se mostrará la última actividad activa, la actividad *Web*. La diferencia es que ahora sólo hace falta pulsar *BACK* una vez para abandonar la aplicación.

¿Qué ha sucedido? Las actividades *Principal* y *Web* se encontraban en tareas distintas. Al

volver a la actividad *Web* y darle a *BACK* hemos destruido su tarea asociada, pero no hemos vuelto a la tarea que contiene la actividad *Principal*.

Nota:

Los valores `standard` y `singleTop` del atributo `launchMode` son los más habituales en aplicaciones Android. Por otra parte, `singleTask` y `singleInstance` simplemente no son adecuados para la inmensa mayoría de aplicaciones. Sea cual sea el valor de `launchMode` que se utilice, es conveniente hacer pruebas de navegación en nuestra aplicación, observando cómo se comporta conforme vamos pulsando el botón *BACK*.

- Veamos como lanzar correctamente la actividad *Web* en una tarea independiente sin confundir al usuario. En el *Manifest* de la aplicación dale como valor al atributo *taskAffinity* de la actividad *Web* la cadena `"es.ua.jtech.android.tareas2"`.
- Ejecuta la aplicación y pulsa el botón para mostrar la actividad *Web*. Ahora pulsa el botón *HOME*. Si ahora pulsas el botón *HOME* durante un rato, aparecerán dos tares con el nombre *Tareas* en el listado. Una es la tarea que contiene la actividad *Principal* y la otra es la tarea que contiene la actividad *Web*.
- Aun así todo sigue siendo un poco confuso, porque las dos tareas tienen el mismo nombre de aplicación. Cambia en el *Manifest* el valor del atributo `android:label` de la tarea *Web* a *Mi Navegador*.
- Ahora, cuando repitas el experimento anterior y muestres el listado de aplicaciones recientes verás las aplicaciones correspondientes a las dos tareas mostrándose con nombres diferentes.
- Activemos ahora el flag `FLAG_ACTIVITY_EXCLUDE_FROM_RECENTS` en el `Intent` que utilizamos para lanzar *Web* desde principal. Ejecuta la aplicación, pulsa el botón para lanzar la actividad *Web*, y pulsa el botón *HOME*. Haz ahora una pulsación prolongada del botón *Home*. ¿Aparece la tarea *Mi Navegador* en el listado de tareas recientes?

5. Introducción al diseño de interfaces gráficas en Android

En esta sesión comenzaremos a tratar el tema de la interfaz gráfica de las aplicaciones en Android. Pero antes de empezar hemos de tener en cuenta que la terminología de Android referida a interfaces gráficas puede resultar un poco extraña al principio. Examinemos alguno de los conceptos que trataremos en mayor profundidad a lo largo de la sesión:

- Las **Vistas** son la base del desarrollo de interfaces gráficas en Android. Todos los elementos gráficos, comunmente llamados *widgets* o *componentes* en otros entornos, son subclases de `View`. Otros elementos más complejos, como agrupaciones de vistas o layouts también heredan de esta clase. Así pues, en Android, un botón será una vista, así como también un campo de edición de texto.
- Los **Grupos de Vistas** son extensiones de la vista genérica. Subclases de `ViewGroup`, pueden contener múltiples vistas hijas. Los layouts son extensiones de los grupos de vistas.
- Los **Layouts** son medios por los que organizar vistas en la pantalla del dispositivo. Existen diferentes tipos de *layout* que nos permitirán disponer los elementos de la pantalla de diversas maneras.
- Cada una de las pantallas o ventanas de nuestra aplicación se corresponderá con una **Actividad**. Las actividades ya han sido tratadas en detalle en sesiones anteriores. En Android son el equivalente de los formularios. Para mostrar una interfaz de usuario la operación que se debe realizar es asignar una vista (normalmente un layout) a una actividad.

5.1. Vistas

Todos los componentes visuales en Android son una subclase de la clase `View`. No se les llama *widgets* para no confundirlos con las aplicaciones de tipo widget que se pueden mostrar en la ventana inicial de Android. En sesiones anteriores ya hemos conocido algunas vistas: el botón (clase `Button`) y la etiqueta de texto (clase `TextView`).

5.1.1. Crear interfaces de usuario con vistas

Cuando se inicia una actividad, lo hace con una pantalla temporalmente vacía sobre la que deberemos colocar todos los elementos gráficos de su interfaz. Para asignar el interfaz de usuario se hace uso del método `setContentView`, pasando como parámetro una instancia de la clase `View` o de alguna de sus subclases. Este método también acepta como parámetro un identificador de recurso, correspondiente a un archivo XML con una descripción de interfaz gráfica. Este segundo método suele ser el más habitual.

Usar recursos de tipo layout permite separar la capa de presentación de la capa lógica, proporcionando la suficiente flexibilidad para cambiar la interfaz gráfica sin necesidad de modificar ni una línea de código. Definir la interfaz gráfica por medio de recursos en

lugar de mediante código permite también especificar diferentes layouts en función de diferentes configuraciones de hardware o incluso que se pueda modificar la interfaz en tiempo de ejecución cuando se produzca algún evento, como por ejemplo cambiar la orientación de la pantalla.

En el siguiente código vemos como inicializar la interfaz gráfica a partir de un layout definido en los recursos de la aplicación. Normalemente el recurso consistirá en un archivo XML almacenado en la carpeta `/res/layout/`. En este ejemplo se supone que se está haciendo uso del archivo *milayout.xml*, el cual se encuentra precisamente en dicha carpeta:

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

        setContentView(R.layout.milayout);
        TextView miTexto = (TextView)findViewById(R.id.texto);
}
```

En el ejemplo anterior también se puede observar cómo acceder desde el código de la actividad a cualquiera de las vistas definidas en el layout, por medio de una llamada al método `findViewById`. En este caso se crea una instancia de la clase `TextView`, que se corresponderá con el elemento `TextView` del layout *milayout*, con lo que podremos acceder a la vista desde el código para modificar el texto que muestra, poder asociarle manejadores de eventos, etc. El método `findViewById` recibe como parámetro el identificador de la vista en el layout. El identificador de una vista se especifica mediante su atributo `android:id` y tiene la siguiente sintaxis: `android:id="@+id/[IDENTIFICADOR]"`, donde `[IDENTIFICADOR]` es el identificador que le queremos asignar al elemento.

Como se ha comentado anteriormente, la otra alternativa para crear la interfaz gráfica de una actividad es construirla directamente desde el código fuente. En el siguiente ejemplo vemos cómo asignar una interfaz gráfica compuesta únicamente por un `TextView` a una actividad cualquiera:

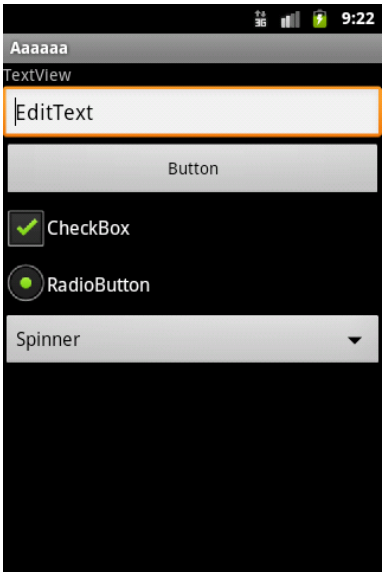
```
@Override
public void onCreate(Bundle sagedInstaceState) {
        super.onCreate(savedInstanceState);

        textView texto = new TextView(this);
        setContentView(texto);

        texto.setText("Hola Mundo!");
}
```

El mayor inconveniente es que el método `setContentView` tan solo acepta como parámetro una única vista, así que si queremos una interfaz más compleja deberemos pasar como parámetro algún layout que deberemos haber construido manualmente en el código.

5.1.2. Las vistas de Android

Android proporciona una gran variedad de vistas básicas para poder diseñar interfaces de usuario de manera sencilla. Usando estas vistas podremos desarrollar aplicaciones consistentes visualmente con las del resto del sistema, aunque también es posible modificarlas o ampliar sus funcionalidades. Algunas de las vistas más utilizadas son:

- **TextView**: una etiqueta de sólo lectura. Permite texto multilínea, formateado de cadenas, etc.
- **EditText**: una caja de texto editable. Permite texto multilínea, texto flotante, etc.
- **ListView**: un grupo de vistas que crea y administra una lista vertical de vistas, correspondiéndose cada una de ellas a una fila de la lista.
- **Spinner**: un cuadro de selección que permite seleccionar un elemento de una lista de posibles opciones. Se muestra como un botón que al ser pulsado visualiza el listado de posibles opciones.
- **Button**: un botón normal.
- **CheckBox**: un botón con dos posibles estados representado por un recuadro que puede estar marcado o no.
- **RadioButton**: un grupo de botones con dos posibles estados. Uno de estos grupos muestra al usuario un conjunto de opciones de las cuales sólo una puede estar activa simultáneamente.
- **SeekBar**: una barra de desplazamiento, que permite escoger visualmente un valor dentro de un rango entre un valor inicial y final.



Vistas básicas de Android

Estos son sólo unos ejemplos. Android pone a nuestra disposición vistas más avanzadas, incluyendo selectores de fecha, cajas de texto con autocompletado, mapas, galerías y pestañas. En *http://developer.android.com/guide/tutorials/views/index.html* se puede consultar un listado de los widgets disponibles en Android.

5.2. Layouts

Los layouts son subclases de `ViewGroup` utilizadas para posicionar diferentes vistas en nuestra interfaz gráfica. Los layouts se pueden anidar con el objetivo de conseguir crear layouts más complejos. Algunos de los layouts disponibles en Android son:

- **FrameLayout**: es el layout más simple. Lo único que hace es añadir cada vista a la esquina superior izquierda. Si se añaden varias vistas se van colocando una encima de la otra, de tal forma que las vistas superiores ocultan a las inferiores.
- **LinearLayout**: alinea diferentes vistas ya sea en una línea horizontal o una línea vertical. Un `LinearLayout` vertical consiste en una columna de vistas, mientras que un `LinearLayout` horizontal no es más que una fila de vistas. Este layout permite establecer un peso mediante la propiedad `weight` a cada elemento, lo que controlará el tamaño relativo de cada elemento en la vista.
- **RelativeLayout**: es el layout nativo más flexible, permitiendo definir la posición de cada una de sus vistas de manera relativa a la posición de los demás o a los bordes de la pantalla.
- **TableLayout**: permite disponer un conjunto de vistas en una rejilla formada por varias filas y columnas. Es posible permitir que las filas o columnas crezcan o disminuyan de tamaño.
- **Gallery**: muestra una lista horizontal de vistas en la que se puede navegar mediante un scroll.

La documentación de Android describe en detalle las características y propiedades de cada layout. Esta documentación se puede consultar en *http://developer.android.com/guide/topics/ui/layout-objects.html*.

5.2.1. Utilizando layouts

La manera habitual de hacer uso de layouts es mediante un fichero XML definido como un recurso de la aplicación, en la carpeta */res/layout/*. Este fichero debe contener un elemento raíz, el cual podrá contener de manera anidada tantos layouts y vistas como sea necesario. A continuación se muestra un ejemplo de layout en el que un `TetView` es colocado sobre un `EditText` usando un `LinearLayout` vertical:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="vertical"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent">
        <TextView
                android:layout_width="fill_parent"
                android:layout_height="wrap_content"
                android:text="Introduce un texto"
        />
        <EditText
                android:layout_width="fill_parent"
                android:layout_height="wrap_content"
                android:text="Escribe el texto aquí"
```

```
        />
</LinearLayout>
```

Para cada elemento se debe suministrar un valor para los atributos `layout_width` y `layout_height`. Se podría utilizar una altura o anchura exacta en píxeles, pero no es aconsejable, ya que nuestra aplicación podría ejecutarse en terminales con diferentes resoluciones o tamaños de pantalla. Los valores más habituales para estos atributos, y que además permiten que haya independencia del hardware, son los usados en este ejemplo: `fill_parent` y `wrap_content`.

El valor `wrap_content` limita el tamaño de una vista al mínimo requerido para poder mostrar sus contenidos. Por su parte, el valor `fill_parent` expande la vista para que ocupe todo el tamaño disponible en su vista padre (o en la pantalla, si se trata del elemento raíz). En el ejemplo anterior el layout situado como elemento raíz ocupará toda la pantalla. Las dos vistas dentro del layout ocuparán toda la anchura disponible, mientras que su altura será tan sólo la necesaria para mostrar sus respectivos textos.

En el caso en el que sea estrictamente necesario por algún motivo (ya que es algo que se desaconseja en general) es posible implementar un layout en el propio código fuente. Cuando asignamos vistas a layouts por medio de código, es importate hacer uso de `LayoutParameters` por medio del método `setLayoutParams`, o pasándolos como parámetro en la llamada a `addView`, tal como se muestra en el siguiente ejemplo:

```
LinearLayout ll = new LinearLayout(this);
ll.setOrientation(LinearLayout.VERTICAL);

TextView texto = new TextView(this);
EditText edicion = new EditText(this);

text.setText("Introduce un texto");
edicion.setText("Escribe el texto aquí");

int lHeight = LinearLayout.LayoutParams.FILL_PARENT;
int lWidth = LienarLayout.LayoutParams.WRAP_CONTENT;

ll.addView(texto, new LinearLayout.LayoutParams(lHeight, lWidth));
ll.addView(edicion, new LinearLayout.layoutParams(lHeight, lWidth));

setContentView(ll);
```

5.2.2. Optimizar layouts

El proceso mediante el cual se rellena la pantalla correspondiente a una actividad de elementos gráficos es costoso computacionalmente. El añadir un nuevo layout o una nueva vista puede tener un gran impacto en la latencia a la hora de navegar por las diferentes actividades de nuestra aplicación.

Suele ser una buena práctica, en general, utilizar layouts tan simples como sea posible. Para conseguir una interfaz más eficiente podemos seguir los siguientes consejos:

- Evitar anidamientos innecesarios: no introduzcas un layout dentro de otro a menos que sea estrictamente necesario. Un `LinearLayout` dentro de un `FrameLayout`,

usando ambos el valor `fill_parent` para su altura y su anchura, no producirá ningún cambio en el aspecto final de la interfaz y su único efecto será aumentar el coste computacional de construir dicha interfaz. Buscar layouts redundantes, sobre todo si has hecho varios cambios en la interfaz.
- Evitar usar demasiadas vistas: cada vista que se añade a la interfaz requiere recursos y tiempo de ejecución.
- Evitar anidamientos profundos: debido a que los layouts pueden ser anidados de cualquier manera y sin limitaciones es fácil caer en la tentación de construir estructuras complejas, con muchos anidamientos. Aunque no exista un límite para el nivel de anidamiento, deberemos intentar que sea lo más bajo posible.

Para ayudarnos en la tarea de optimizar nuestros layouts disponemos del comando `layoutopt`, incluido en el SDK de Android y que puede ser ejecutado en la línea de comandos. Para analizar un layout o conjunto de layouts ejecutamos el comando pasándole como parámetro el nombre de un recurso de tipo layout o una carpeta de recursos de este tipo. El comando nos mostrará diferentes recomendaciones para mejorar nuestros layouts.

5.3. Uso básico de vistas y layouts

En esta sección veremos algunos detalles sobre cómo utilizar algunas de las vistas y layouts proporcionados por Android. Se debe tener en cuenta que esto no es más que una introducción, y que es posible encontrar una descripción más detallada de todos estos elementos en la documentación de Android.

5.3.1. TextView

Se corresponde con una etiqueta de texto simple, que sirve evidentemente para mostrar un texto al usuario. Su atributo más importante es `android:text`, cuyo valor indica el texto a mostrar por pantalla. También pueden ser de interés los atributos `android:textColor` y `android:textSize`. Una curiosidad es que es posible hacer el `TextView` editable por medio del atributo booleano `android:editable`; sin embargo, esta vista no está preparada para este tipo de acción. Sólo se podrá editar texto a través de su subclase `EditText`.

Dentro del código los métodos más utilizados son `setText` y `appendText`, que permiten modificar el texto del `TextView` o añadir texto adicional durante la ejecución del programa. En ambos casos el parámetro recibido será una cadena. Para acceder a su texto usaremos el método `getText`.

5.3.2. EditText

`EditText` no es más que una subclase de `TextView` que está preparada para la edición de texto. Al pulsar sobre la vista la interfaz de Android mostrará un teclado para poder

introducir nuestros datos. En el código se maneja igual que un `TextView` (por medio de los métodos `getText` o `setText`, por ejemplo).

5.3.3. Button

Esta vista representa un botón normal y corriente, con un texto asociado. Para cambiar el texto del botón usaremos su atributo `android:text`.

La forma más habitual de interactuar con un botón es pulsarlo para desencadenar un evento. Para que nuestra aplicación realice una determinada acción cuando el botón sea pulsado, deberemos implementar un manejador para el evento *OnClick*. Veamos un ejemplo. Supongamos una actividad cuyo layout viene definido por el siguiente fichero XML:

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

En el método `onCreate` de la actividad podríamos incluir el siguiente código para que cuando se pulsara el botón se modificara el texto del `TextView`:

```
public class MiActividad extends Activity {
    protected void onCreate(Bundle icicle) {
        super.onCreate(icicle);

        setContentView(R.layout.miLayout);

                TextView texto = (TextView)findViewById(R.id.texto);
        Button boton = (Button)findViewById(R.id.boton);
        boton.setOnClickListener(new OnClickListener() {
            public void onClick(View v) {
                texto.setText("Botón pulsado");
            }
        });
    }
}
```

También es posible definir el manejador para el click del ratón en el propio fichero XML, a través del atributo `android:onClick`. El valor de este atributo será el nombre del método que se encargará de manejar el evento. Según la especificación del botón definido en el siguiente ejemplo:

```
<Button
     android:layout_height="wrap_content"
     android:layout_width="wrap_content"
     android:text="@string/textoBoton"
     android:onClick="manejador" />
```

cada vez que se pulse el botón se lanzará el método `manejador`, cuya cabecera deberá ser la siguiente:

```
public void manejador(View vista) {
        // Hacer algo
}
```

5.3.4. CheckBox

Se trata de un botón que puede encontrarse en dos posibles estados: seleccionado o no seleccionado. Este tipo de botones, al contrario que en el caso de los pertenecientes a la vista `RadioButton`, son totalmente independientes unos de otros, por lo que varios de ellos pueden encontrarse seleccionados al mismo tiempo. Dentro del código se manejan individualmente. Los métodos más importantes para manejar vistas de tipo `CheckBox` son `isChecked`, que devuelve un booleano indicando si el botón está seleccionado, y `setChecked`, que recibe un valor booleano como parámetro y sirve para indicar el estado del botón.

Veamos otro ejemplo. Supongamos que el layout de nuestra actividad se define de la siguiente forma:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="vertical"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent">
        <TextView
                android:id="@+id/texto"
                android:layout_width="fill_parent"
                android:layout_height="wrap_content"
                android:text="Texto"
        />
        <CheckBox
                android:id="@+id/micheckbox"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:text="Púlsame"
                android:checked="false"
        />
</LinearLayout>
```

Obsérvese como se hace uso del atributo `android:text` del `CheckBox` para establecer el texto que acompañará al botón. También se ha hecho uso del atributo `android:checked` para establecer el estado inicial del botón. Para hacer que nuestra actividad reaccione a la pulsación del `CheckBox` implementamos el manejador del evento *OnClick* para el mismo:

```
public class MiActividad extends Activity {
        TextView texto;
        CheckBox miCheckbox;
```

```
     protected void onCreate(Bundle icicle) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
                texto = (TextView)findViewById(R.id.texto);
                miCheckbox = (CheckBox)findViewById(R.id.micheckbox);
                miCheckbox.setOnClickListener(new OnClickListener() {
                    public void onClick(View v) {
                        // Realizamos una acción u otra según si el botón
está pulsado o no
                        if (miCheckbox.isChecked()) {
                            texto.setText("activado");
                        } else {
                            texto.setText("desactivado");
                        }
                    }
                });
        }
}
```

5.3.5. RadioButton

Un `RadioButton`, al igual que un `CheckButton`, tiene dos estados (seleccionado y no seleccionado). La diferencia con el anterior es que una vez que el botón está en el estado de seleccionado no puede cambiar de estado por intervención directa del usuario. Los elementos de tipo `RadioButton` suelen agruparse normalmente en un elemento de tipo `RadioGroup`, de tal forma que la activación de uno de los botones del grupo supondrá la desactivación de los demás.

En el siguiente ejemplo vemos como definir un grupo formado por dos botones de radio en el archivo XML de layout de una actividad:

```
<RadioGroup
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:orientation="vertical">
      <RadioButton android:id="@+id/radio_si"
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:text="Sí" />
      <RadioButton android:id="@+id/radio_no"
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:text="No" />
</RadioGroup>
```

Para que se realice una determinada acción al pulsar uno de los botones de radio deberemos implementar el manejador del evento *OnClick*:

```
public class MiActividad extends Activity {
        boolean valor = true;
        RadioButton botonSi, botonNo;

     protected void onCreate(Bundle icicle) {
             super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

             botonSi = (RadioButton)findViewById(R.id.radio_si);
             botonNo = (RadioButton)findViewById(R.id.radio_no);
```

```
                botonSi.setOnClickListener(new OnClickListener() {
                    public void onClick(View v) {
                        valor = true;
                    }
                });

                botonNo.setOnClickListener(new OnClickListener() {
                    public void onClick(View v) {
                        valor = false;
                    }
                });
        }
}
```

5.3.6. Spinner

Un `Spinner` es una vista que permite escoger uno de entre una lista de elementos. Es el equivalente a un cuadro de selección de un formulario normal. Construir un `Spinner` en Android requiere varios pasos, que vamos a ver a continuación.

- En primer lugar añadimos el `Spinner` a nuestro layout. Los dos atributos específicos de este tipo de vista que hemos añadido al ejemplo son `drawSelectorOnTop`, que dibujará un control encima de la opción seleccionada, y `android:prompt`, que indica la cadena de texto a mostrar cuando todavía no se ha seleccionado ninguna opción. Obsérvese como en este caso su valor se ha definido a partir de una cadena definida en el archivo *strings.xml* de los recursos de la aplicación. Esto es así porque no es posible asignarle una cadena como valor a este atributo.

```
<Spinner
        android:id="@+id/spinner"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:drawSelectorOnTop="true"
        android:prompt="@string/eligeopcion"
    />
```

- Añadimos la cadena para el *prompt* en */res/values/strings.xml*:

```
<string name="eligeopcion">Elige!</string>
```

- A continuación debemos indicar cuáles serán las opciones disponibles en el `Spinner`. Esto se hará también mediante los recursos de la aplicación. En concreto, crearemos un fichero en */res/values/* al que llamaremos *arrays.xml* y que contendrá lo siguiente:

```
<resources>
        <string-array name="opciones">
                <item>Mensual</item>
                <item>Trimestral</item>
                <item>Semestral</item>
                <item>Anual</item>
        </string-array>
</resources>
```

- Finalmente añadiremos el código necesario en el método `onCreate` de la actividad para que se asocien las opciones al `Spinner`. Para ello se utiliza un objeto de la clase `ArrayAdapter`, que asocia cada cadena de nuestro `string-array` a una vista que será

mostrada en el `Spinner`. Al método `createFromResource` le pasamos el contexto de la aplicación, el identificador del `string-array` y un identificador para indicar el tipo de vista que queremos asociar a cada elemento seleccionable. El método `setDropDownViewResource` se utiliza para definir el tipo de layout que se utilizará para mostrar la lista completa de opciones. Finalmente se asocia el adaptador al `Spinner`:

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

    Spinner s = (Spinner) findViewById(R.id.spinner);
    ArrayAdapter adaptador = ArrayAdapter.createFromResource(
            this, R.array.opciones, android.R.layout.simple_spinner_item);
    adaptador.setDropDownViewResource(
                    android.R.layout.simple_spinner_dropdown_item);
    s.setAdapter(adaptador);
}
```

Con respecto a los eventos relacionados con el `Spinner`, hemos de tener en cuenta que Android no soporta el manejo de eventos para sus elementos individuales. Si intentamos crear un manejador para el evento *OnClick* para cada opción por separado, por ejemplo, se producirá un error en tiempo de ejecución. En lugar de ello deberemos definir un manejador de evento global para el `Spinner`, que tendrá el siguiente aspecto:

```
spinner.setOnItemSelectedListener(new OnItemSelectedListener() {
        public void onItemSelected(AdapterView<?> arg0, View arg1,
                                int arg2, long arg3) {
                // Hacer algo
        }

        public void onNothingSelected(AdapterView<?> arg0) {
                // Hacer algo
        }
});
```

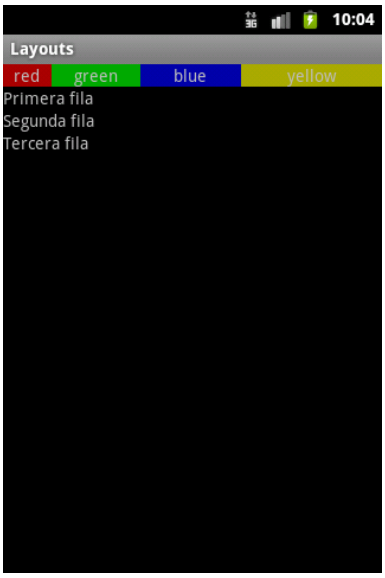
5.3.7. LinearLayout

Como se ha comentado anteriormente se trata de un layout que organiza sus componentes en una única fila o una única columna, según éste sea horizontal o vertical, respectivamente. Para establecer la orientación podemos utilizar o bien el atributo `android:orientation` en la definición del layout en el archivo XML (tomando como valor `horizontal` o `vertical`, siendo el primero el valor por defecto) o mediante el método `setOrientation` desde el código.

Un ejemplo de layout de este tipo sería:

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

  <LinearLayout
```

```
      android:orientation="horizontal"
      android:layout_width="fill_parent"
      android:layout_height="fill_parent">
      <TextView
          android:text="red"
          android:gravity="center_horizontal"
          android:background="#aa0000"
          android:layout_width="wrap_content"
          android:layout_height="fill_parent"
          android:layout_weight="1"/>
      <TextView
          android:text="green"
          android:gravity="center_horizontal"
          android:background="#00aa00"
          android:layout_width="wrap_content"
          android:layout_height="fill_parent"
          android:layout_weight="2"/>
      <TextView
          android:text="blue"
          android:gravity="center_horizontal"
          android:background="#0000aa"
          android:layout_width="wrap_content"
          android:layout_height="fill_parent"
          android:layout_weight="3"/>
      <TextView
          android:text="yellow"
          android:gravity="center_horizontal"
          android:background="#aaaa00"
          android:layout_width="wrap_content"
          android:layout_height="fill_parent"
          android:layout_weight="4"/>
  </LinearLayout>

  <LinearLayout
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    <TextView
        android:text="Primera fila"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"/>
    <TextView
        android:text="Segunda fila"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"/>
    <TextView
        android:text="Tercera fila"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"/>
  </LinearLayout>

</LinearLayout>
```



Ejemplo de LinearLayout

En este ejemplo podemos ver el uso de un par de atributos interesantes. En primer lugar tenemos `android:layout-weight`, que va a permitir que las diferentes vistas dentro del `LinearLayout` ocupen una porción del espacio disponible proporcional a su peso. Por otra parte tenemos `android:gravity`, que permite alinear el texto de una vista. En este caso se ha centro el texto horizontalmente.

5.3.8. TableLayout

Se trata de un layout que organiza sus elementos como una rejilla dividida en filas y columnas. Se compone de un conjunto de elementos de tipo `TableRow` que representan a las diferentes filas de la tabla. Cada fila a su vez se compone de un conjunto de vistas. Existe la posibilidad de tener filas con diferente número de vistas; en este caso la tabla tendrá tantas columnas como celdas tenga la fila que contenga más vistas.

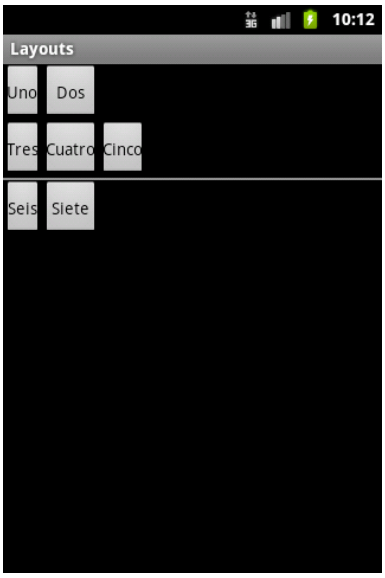
La anchura de una columna vendrá definida por la fila que contenta la vista de máxima anchura en dicha columna. Esto quiere decir que no tenemos libertad para asignar valor al atributo `android:layout_width`, que debería valer `match_parent`. Sí que podemos definir la altura, aunque lo más habitual será utilizar el valor `wrap_content` para el atributo `android:layout_height`. Si no se indica lo contrario esos serán los valores por defecto.

Aunque no podamos controlar la anchura de una columna, si que podemos tener columnas de anchura dinámica, usando los métodos `setColumnShrinkable()` y `setColumnStretchable()`. El primero permitirá disminuir la anchura de una columna con tal de que su correspondiente tabla pueda caber correctamente en su elemento padre. Por su parte, el segundo permitirá aumentar la anchura de una columna para hacer que la tabla pueda ocupar todo el espacio que tenga disponible. Hay que tener en cuenta que se

pueden utilizar ambos métodos para una misma columna; en este caso la anchura de la columna siempre se incrementará hasta que se utilice todo el espacio disponible, pero nunca más. Otra opción disponible es ocultar una columna por medio del método `setColumnCollapsed`.

El siguiente ejemplo muestra un `TableLayout` definido como recurso de la aplicación por medio de un fichero XML. Como se puede observar en el código se ha introducido un elemento `View` cuya única función es de servir de separador visual entre la segunda y la tercera fila:

```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    <TableRow>
        <Button
            android:text="Uno"
            android:padding="3dip" />
        <Button
            android:text="Dos"
            android:padding="3dip" />
    </TableRow>
        <TableRow>
        <Button
            android:text="Tres"
            android:padding="3dip" />
        <Button
            android:text="Cuatro"
            android:padding="3dip" />
        <Button
            android:text="Cinco"
            android:padding="3dip" />
    </TableRow>
    <View
        android:layout_height="2dip"
        android:background="#FF909090" />
    <TableRow>
        <Button
            android:text="Seis"
            android:padding="3dip" />
        <Button
            android:text="Siete"
            android:padding="3dip" />
    </TableRow>
</TableLayout>
```



Ejemplo de TableLayout

5.3.9. RelativeLayout

Este layout es muy flexible; permite indicar la posición de un elemento en función de otros y de los bordes de la pantalla. Enumerar todos sus posibles atributos alargaría innecesariamente esta sección, así que mostraremos un ejemplo que deje claro su funcionamiento, dejando al lector la tarea de acudir a la documentación de Android en el caso de que necesite profundizar más:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    <TextView
        android:id="@+id/etiqueta"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="Escribe"/>
    <EditText
        android:id="@+id/entrada"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:background="@android:drawable/editbox_background"
        android:layout_below="@id/etiqueta"/>
    <Button
        android:id="@+id/ok"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@id/entrada"
        android:layout_alignParentRight="true"
        android:layout_marginLeft="10dip"
        android:text="OK" />
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_toLeftOf="@id/ok"
```

```
        android:layout_alignTop="@id/ok"
        android:text="Cancelar" />
</RelativeLayout>
```

Ejemplo de RelativeLayout

5.4. Interfaces independientes de densidad y resolución

Durante el primer año de vida del sistema Android tan solo existía un único terminal que hiciera uso de este sistema, por lo que a la hora de diseñar una interfaz gráfica no era necesario tener en mente diferentes configuraciones de hardware. A principios del 2010 se produjo una avalancha en el mercado de nuevos dispositivos basados en Android. Una consecuencia de esta variedad fue un aumento también en las posibles configuraciones de resolución o densidad.

A la hora de diseñar una interfaz gráfica para nuestra aplicación Android es importante tener en cuenta que ésta podría llegar a ejecutarse en una gran variedad de configuraciones hardware distintas, con diferentes resoluciones (HVGA, QVGA o WVGA) y densidades de pantalla (3.2 pulgadas, 3.7, 4, etc.). Los tablets cada vez están más de moda, y es posible que otro tipo de dispositivos incluyan un sistema Android en el futuro.

En esta sección trataremos algunas cuestiones relativas a cómo diseñar nuestras interfaces gráficas teniendo todo esto en cuenta.

5.4.1. Múltiples archivos de recurso

En la primera sesión del módulo de Android tratamos el tema de los recursos de la aplicación. Vimos que para determinados tipos de recurso era posible crear una estructura

paralela de directorios que permitiera almacenar recursos a utilizar con diferentes configuraciones de hardware. En este apartado vemos diferentes sufijos que podemos añadir al nombre de la carpeta *layout* o *drawable* dentro de los recursos de la aplicación para definir diferentes interfaces según la configuración de nuestra pantalla.

- **Tamaño de la pantalla**: tamaño de la pantalla relativa a un terminal de tamaño "estándar":
  - `small`: una pantalla con un tamaño menor de 3.2 pulgadas.
  - `medium`: para dispositivos con un tamaño de pantalla típico.
  - `large`: para pantallas de un tamaño significativamente mayor que la de un terminal típico, como la pantalla de una tablet o un netbook.
- **Densidad**: se refiere a la densidad en píxeles de la pantalla. Normalmente se mide en puntos por pulgada (*dots per inch* o *dpi*). Se calcula en función de la resolución y el tamaño físico de la pantalla:
  - `ldpi`: usado para almacenar recursos para baja densidad pensados para pantallas con densidades entre 100 y 140dpi.
  - `mdpi`: usado para pantallas de densidad media entre 140 y 180dpi.
  - `hdpi`: usado para pantalla de alta densidad, entre 190 y 250dpi.
  - `nodpi`: usado para recursos que no deberían ser escalados, sea cual sea la densidad de la pantalla donde van a ser mostrados.
- **Relación de aspecto** (aspect ratio): se trata de la relación de la altura con respecto a la anchura de la pantalla:
  - `long`: usado para pantallas que son mucho más anchas que las de los dispositivos estándar.
  - `notlong`: usado para terminales con una relación de aspecto estándar.

Cada uno de estos sufijos puede ser utilizado de manera independiente o en combinación con otros. A continuación tenemos un par de ejemplos de carpetas de recursos para layouts que hacen uso de alguno de los listados anteriormente:

```
/res/layout-small-long/
/res/layout-large/
/res/drawable-hdpi/
```

5.4.2. Indicar las configuraciones de pantalla soportadas por nuestra aplicación

En el caso de alguna aplicación puede que no sea posible optimizar la interfaz para todas y cada una de las configuraciones de pantalla existentes. En estos casos puede ser útil utilizar el elemento `supports-screens` en el *Manifest* de la aplicación para especificar en qué pantallas puede ésta ser ejecutada. En el siguiente código se muestra un ejemplo:

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

Un valor de `false` en alguno de estos atributos forzará a Android a mostrar la interfaz por medio de un modo de compatibilidad que consistirá básicamente en realizar un escalado. Esto no suele funcionar siempre de manera correcta, haciendo aparecer elementos extraños o artefactos en la interfaz.

5.4.3. Prácticas a seguir para conseguir interfaces independientes de la resolución

En esta sección resumimos algunas técnicas que podremos utilizar en nuestras aplicaciones para que puedan ser visualizadas de manera correcta en la inmensa mayoría de las configuraciones hardware. Pero antes de empezar hemos de tener en cuenta que la consideración más importante a tener en cuenta es que nunca debemos presuponer cuál será el tamaño de la pantalla del terminal en el que se ejecutará nuestra aplicación. La regla de oro es crear layouts para diferentes clases de pantallas (pequeña, normal y grande) y para diferentes resoluciones (baja, media y alta).

La primera regla es **no utilizar nunca tamaños absolutos basados en número de píxeles**. Esto se aplica tanto a layouts, como a vistas y fuentes. En particular se debería evitar usar el `AbsoluteLayout`, que se basa en la especificación de la disposición de sus elementos a partir de coordenadas en píxeles. Haciendo uso del `RelativeLayout` se pueden generar interfaces suficientemente complejas, evitando el uso de coordenadas absolutas, por lo que es una mejor solución.

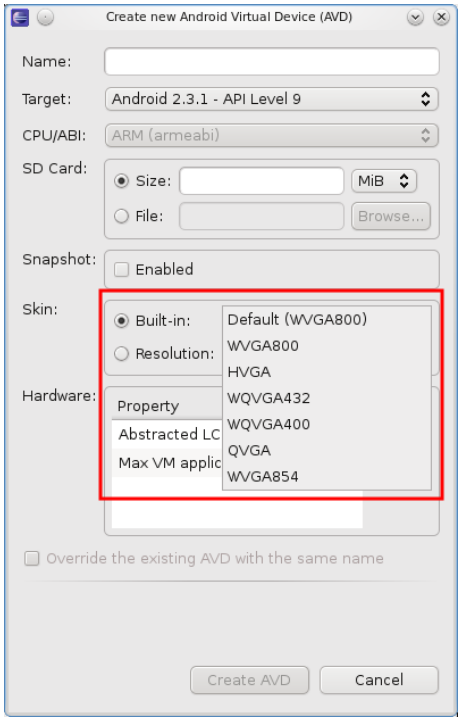
A la hora de determinar el tamaño de una vista o un layout utilizaremos valores como `fill_parent` o `wrap_content`. También podría ser posible asignar valores a los atributos `layout-width` y `layout-height` medidos en píxeles independientes de la densidad (dp) o en píxeles independientes de la escala (sp). Esto mismo podría ser aplicado a las fuentes. Las medidas independientes de la densidad o de la escala son un medio por el que se puede especificar el tamaño de los componentes de nuestra interfaz de tal forma que al mostrarse en diferentes configuraciones hardware se escalen para seguir mostrando el mismo aspecto. Por ejemplo, un dp equivale a un píxel en una pantalla de 160dpi. Una línea de 2dp de anchura aparecerá como una línea de 3 píxeles de anchura en una pantalla de 240dpi.

Por supuesto no debemos olvidar **utilizar sufijos** para crear una estructura paralela de directorios de recursos, tal como se explicó en la sección anterior. Como mínimo se deberían definir las siguientes carpetas con sus recursos correspondientes:

```
/res/drawable-ldpi/
/res/drawable-mdpi/
/res/drawable-hdpi/
/res/layout-small/
/res/layout-normal/
/res/laout-large/
```

Por último, no debemos olvidar **probar nuestra aplicación en la mayor variedad de dispositivos posibles**. En este sentido podemos hacer uso de la emulación y de Android SDK, ya que es poco práctico hacer estas pruebas en dispositivos reales. Hemos de recordar que a la hora de crear un dispositivo virtual para probar nuestras aplicaciones

Android podemos especificar diferentes configuraciones de pantalla, tanto en cuanto a resolución como en cuanto a densidad. La forma más sencilla de probar diferentes configuraciones es hacer uso de los skins que incorpora el AVD Manager del SDK. Crea un dispositivo virtual para cada opción de skin disponible y prueba tu aplicación en todos ellos.

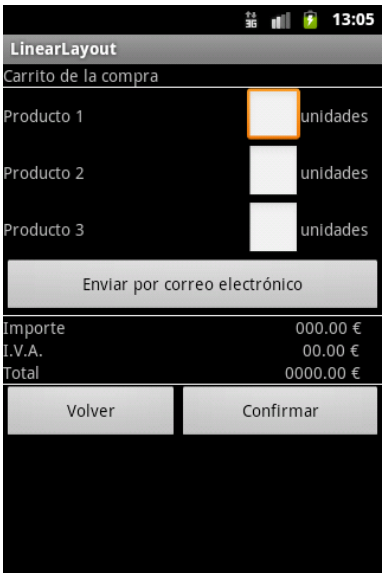


Selección de skins en el AVD del SDK de Android

6. Introducción al diseño de interfaces gráficas en Android - Ejercicios

6.1. LinearLayout

Crea una aplicación llamada *LinearLayout*. La aplicación contendrá una única actividad, llamada *Principal*, cuya interfaz gráfica estará contruida exclusivamente a partir de layouts de tipo `LinearLayout` y deberá ser lo más parecida posible a la mostrada en la siguiente imagen.



Interfaz gráfica de la aplicación LinearLayout

Nota:

Las líneas han sido creadas por medio de elementos `View` a los que se les ha asignado una altura de *1dip* mediante el atributo `android:layout_height` y un color de fondo *#FFFFFF* mediante el atributo `android:background`.

6.2. Colores

Creamos ahora una nueva aplicación; en este caso su nombre será *Colores*. La interfaz de su única actividad (también llamada *Principal*) contendrá dos grupos de botones de radio y un checkbox, además de un elemento de tipo `TextView` en la parte superior.

El primer grupo de botones de radio servirá para modificar el color de fondo del elemento `TextView`, mientras que el segundo permitirá modificar el color del texto. Con respecto al checkbox, éste indicará la presencia o no de texto. Con el checkbox activado el texto se mostrará; en caso contrario no se mostrará texto en el `TextView` y tan sólo será visible su

color de fondo.

El estado inicial de los elementos de la interfaz gráfica de la actividad *Principal* será el mostrado en la siguiente imagen:

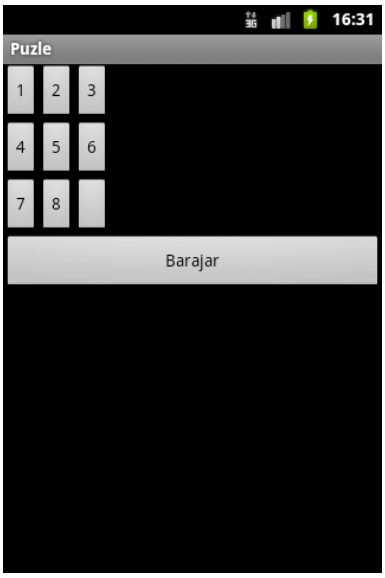
Interfaz gráfica de la aplicación Colores

Nota:

En el código podemos cambiar el color de fondo y de texto de un `TextView` con los métodos `setBackgroundColor` y `setTextColor`. Ambos aceptan como parámetro un entero, que puede ser cualquiera de las constantes estáticas definidas en la clase `Color`.

6.3. Puzle (*)

En este ejercicio implementaremos una versión en Android de un conocido tipo de puzle. En este caso crearemos una nueva aplicación llamada *Puzle* que como en casos anteriores tan sólo contendrá una actividad, de nombre *Principal*. Su interfaz gráfica, que deberá utilizar al menos un `TableLayout`, tendrá el siguiente aspecto:



Interfaz gráfica de la aplicación Puzle

Como se puede observar, tenemos nueve botones formando una rejilla de 3x3 y numerados del 1 al 8 (uno de los botones no tiene etiqueta). Primero debemos añadir manejadores a los botones de tal forma que si se pulsa un botón adyacente ortogonalmente (es decir, no en diagonal) al botón sin etiqueta, se intercambie la etiqueta entre ambos.

En la parte inferior de la interfaz hay otro botón etiquetado como *Barajar*. Al pulsarlo se deberán realizar cincuenta movimientos aleatorios del puzle para desordenarlo, partiendo de la situación en la que se encontrara el puzle en ese momento, y siguiendo las especificaciones del párrafo anterior: en cada iteración intercambiamos la etiqueta del botón vacío con la de algún botón adyacente ortogonalmente.

6.4. Ciudades

En este ejercicio practicaremos con los elementos de tipo `Spinner`. La aplicación *Ciudades* contendrá una única actividad de nombre *Principal*. La interfaz de dicha actividad estará compuesta por un `TextView` y dos elementos de tipo `Spinner`. El primero de ellos permitirá escoger entre tres países cualquiera (inicialmente ningún país estará seleccionado). El segundo permitirá escoger una ciudad según el país seleccionado en el anterior. Cada vez que se seleccione un país en el primer `Spinner` deberán cambiar las opciones del segundo, mostrando dos ciudades del país seleccionado.

La ciudad seleccionada en el segundo `Spinner` aparecerá en el `TextView` de la parte superior.

Para completar el ejercicio debes seguir los siguientes pasos:

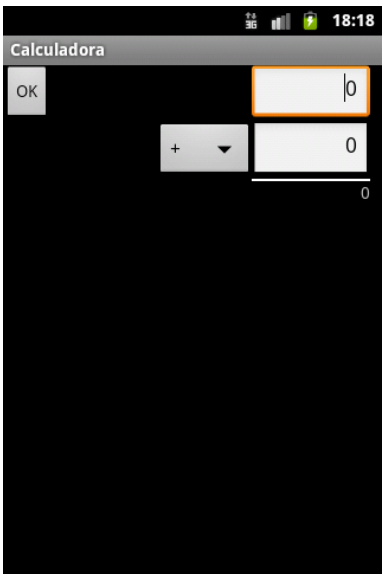
- Añade el `TextView` y los dos `Spinner` al recurso layout de la aplicación, sin olvidar

añadir a estos dos últimos su correspondiente atributo `android:prompt` (con los textos "Selecciona país" y "Selecciona ciudad" respectivamente).

- Crea las opciones para los `Spinner` en el archivo *arrays.xml* de los recursos. Elige tú mismo el nombre de los países y de las ciudades. Recuerda que debes crear tres conjuntos diferentes de opciones (tres elementos de tipo `string-array`) para el segundo `Spinner`, ya que las ciudades a escoger cambiarán según el pais escogido en el primero.
- Rellena el primer `Spinner` con sus correspondientes opciones.
- Rellena el segundo `Spinner` con las ciudades correspondintes al primer país. Esto debes hacerlo así porque siempre que inicies la actividad será el primer país el que se encuentre seleccionado.
- Asígnale al TextView como valor inicial el nombre de la primera ciudad, pues será la que se encontrará seleccionada al iniciar la actividad.
- Añade un manejador al `Spinner` de paises para que cada vez que se seleccione una opción se muestren las opciones adecuadas en el `Spinner` de ciudades.
- Añade un manejador al `Spinner` de ciudades para que cada vez que se seleccione una opción se muestre en el `TextView`. Para obtener el texto correspondiente a la opción seleccionada en el `Spinner` puedes utilizar el método `getSelectedItem` del mismo. Una vez hecho esto puedes llamar al método `toString` para obtener la cadena correspondiente.

6.5. Calculadora sencilla

El objetivo de este ejercicio es implementar una calculadora sencilla. La aplicación *Calculadora* contendrá una única actividad de nombre *Principal*, vuya interfaz gráfica tendrá el siguiente aspecto:



Interfaz gráfica de la aplicación Calculadora

Como se puede observar nuestra calculadora es bastante limitada. Tan solo acepta dos operandos (que se podrán introducir en los dos `EditText`) y cuatro operaciones seleccionables con el `Spinner`: +, -, * y /. En el `TextView` inferior deberá aparecer el resultado de la operación cuando se pulse el botón *Ok*.

A la hora de diseñar la interfaz se ha utilizado un `RelativeLayout`. Los atributos más importantes utilizados han sido: `layout_alignParentRight`, `layout_below`, `align_marginRight`, `android:inputType="number"` para los `EditText` y `android:gravity="right"` para el `TextView` y los `EditText`.

7. Menús, listas y barras de progreso

En esta sesión seguimos hablando de interfaces gráficas en Android, presentando algunos elementos que requieren un estudio especial y un poco más profundo.

7.1. Barras de progreso

Una de las características esenciales que debe presentar cualquier aplicación móvil es la ausencia de latencia a la hora de responder al usuario. Debemos evitar que el tiempo de espera entre que el usuario interaccione con la aplicación y obtenga su respuesta sea demasiado alto. Una alta latencia podría incluso podrucir el cierre de la aplicación.

La forma más adecuada de evitar que se produzca latencia, sobre todo a la hora de realizar conexiones de red, es el uso de `AsyncTasks`. Este concepto será explorado en futuros módulos del curso. Otra posibilidad, que es la que vamos a explorar en esta sesión, es utilizar barras de progreso para informar al usuario del tiempo restante para completar una determinada operación.

En esta sección veremos dos formas posibles de incorporar un componente de este tipo a nuestra aplicación.

7.1.1. Barra de progreso circular

Utilizaremos esta barra de progreso cuando el tiempo que requerirá una tarea es indeterminado o desconocido a priori. La aplicación mostrará un elemento gráfico animado. Simplemente lo incorporamos al layout y lo mostramos cuando sea necesario. En el layout podríamos añadir un elemento `ProgressBar` de la siguiente forma:

```
<ProgressBar
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/progressbar"
/>
```

Una vez añadido al layout en la posición adecuada tan solo será cuestión de mostarlo u ocultarlo cuando sea necesario. Para ello usamos el método `setVisibility`, pasando como parámetro el valor `View.VISIBLE` o `View.INVISIBLE`.

Barra de progreso circular

7.1.2. Barra de progreso lineal

Al contrario que en el caso de la barra de progreso circular, la barra de progreso lineal será utilizada cuando sí conozcamos el tiempo o número de pasos que va a requerir la finalización de una determinada tarea. Como el objetivo va a ser que la barra de progreso se vaya actualizando conforme se va avanzando en la tarea a completar, deberemos incorporar algún mecanismo para ir modificando el estado del componente gráfico. Esto se hará mediante hilos.

Podemos incorporar una barra de progreso lineal a un layout mediante un elemento de tipo `ProgressBar` a cuyo atributo `style` le hemos asignado el valor que se puede ver en el siguiente ejemplo:

```
<ProgressBar
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    style="?android:attr/progressBarStyleHorizontal"
    android:id="@+id/progreso"
    android:max="100"
/>
```

El atributo `android:max` establece, evidentemente, el máximo valor que puede tomar la barra de progreso. Ten en cuenta que no existe un atributo para establecer el valor mínimo. El valor inicial será por lo tanto siempre 0. Si queremos un rango de valores con otro valor inicial, deberemos sumar siempre ese valor inicial a cualquier valor leído desde el `ProgressBar`.

A continuación se muestra un ejemplo muy sencillo que ilustra el uso del `ProgressBar` lineal:

```
public class AndroidProgressBar extends Activity {

        ProgressBar progreso;
        int miProgreso = 0;

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

                progreso=(ProgressBar)findViewById(R.id.progreso);

                new Thread(myThread).start();
            }

        private Runnable myThread = new Runnable(){

                  public void run() {
                           while (miProgreso<100){
                            try{
myHandle.sendMessage(myHandle.obtainMessage());
                                     Thread.sleep(1000);
                            } catch(Throwable t){}
                           }
                }

                Handler myHandle = new Handler(){
                      @Override
                         public void handleMessage(Message msg) {
                          miProgreso++;
```

```
                            progreso.setProgress(miProgreso);
                        }
                    };

    };
}
```

Como se puede observar, la actualización de la barra de progreso se realiza mediante una instancia de `Runnable`. Esto crea un hilo aparte que se encarga de dicha actualización mientras se realizan otras tareas. En este caso la actividad no hace nada; simplemente hace una llamada a `Thread.sleep` para ir aumentando el valor de la barra de progreso de uno en uno cada segundo.

Barra de progreso lineal

7.1.3. SeekBar

Un `SeekBar` es un tipo de vista que permite seleccionar un valor entero como entrada para una determinada actividad. Se trata de una extensión del `ProgressBar` a la que se le ha añadido un control arrastable con el que el usuario puede interaccionar. El valor asociado al `SeekBar` dependerá de la posición en la que se encuentre dicho control.

Para incorporar un `SeekBar` a nuestro layout podemos añadir el siguiente código:

```
<SeekBar android:id="@+id/seek"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:max="100"
        android:progress="2" />
```

donde `android:max` indica el valor máximo, como en el caso de un `ProgressBar`. También, como en ese caso, no es posible establecer un valor mínimo; éste será siempre 0. El atributo `android:progress` proviene de la clase `ProgressBar` y establece el incremento en el valor representado por el `SeekBar` tras cada movimiento del control arrastable.

Seekbar

Ahora es posible ejecutar la actividad e interactuar con la vista. Pero sin manejar sus eventos asociados dicha vista es bastante inútil. Veamos algunos eventos que nos permitan que se realice algo con el valor del `SeekBar`. En concreto vamos a ver un ejemplo muy sencillo en el que un `TextView` mostrará el valor de un `SeekBar`. Tomemos como ejemplo el siguiente layout:

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

```
                        android:layout_width="fill_parent"
                        android:layout_height="wrap_content" />
                <SeekBar android:id="@+id/seek"
                        android:layout_width="fill_parent"
                        android:layout_height="wrap_content"
                        android:max="100"/>
</LinearLayout>
```

El evento para el que vamos a crear un manejador es `onProgressChanged`, que se lanzará cuando el usuario arrastre el control del `SeekBar` cambiando su valor. Lo que haremos en dicho manejador es simplemente modificar el valor del `TextView`. Obsérvese que el manejador ha sido creado haciendo que la actividad implemente la interfaz `SeekBar.OnSeekBarChangeListener`:

```
public class EjemploSeekBar extends Activity implements
SeekBar.OnSeekBarChangeListener{

    SeekBar seek;
    TextView valor;

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

        seek = (SeekBar)findViewById(R.id.seek);
                valor = (TextView)findViewById(R.id.valor);
        seek.setOnSeekBarChangeListener(this);
    }

    public void onProgressChanged(SeekBar seekBar, int progress,
        boolean fromTouch) {
                valor.setText(new Integer(progress).toString());
    }

    public void onStartTrackingTouch(SeekBar seekBar) {
                // Hacer algo
        }

        public void onStopTrackingTouch(SeekBar seekBar) {
                // Hacer algo
        }

}
```

El valor del `SeekBar` se ha obtenido en este ejemplo directamente de uno de los parámetros del manejador del evento. Para leer o modificar el valor del `SeekBar` se pueden utilizar también los métodos `setProgress` y `getProgress` que se heredan de la clase `ProgressBar`.

Otros eventos que quizá podrían sernos útiles son `onStartTrackingTouch`, que se lanza en el momento en el que el usuario pulsa sobre el control arrastrable del `SeekBar`, y `onStopTrackingTouch`, que se lanza en el momento en el que se deja de pulsar dicho control.

7.2. Listas

Las listas son un tipo de vista muy importante en Android; son muy utilizadas para mostrar datos al usuario. Una lista muestra un conjunto de elementos ordenados verticalmente, a través de los cuales se puede navegar por medio de una barra de scroll. Seleccionar un elemento normalmente tendrá como consecuencia otra acción, como por ejemplo lanzar una nueva actividad. En esta sección aprenderemos a crear listas en Android y veremos cómo manejar sus diferentes elementos.

7.2.1. ListActivity

Podemos introducir un `ListView` directamente en un layout como haríamos normalmente con cualquier otro tipo de vista en Android:

```
<ListView
        android:id="@+id/lista"
        android:layout_width="fill_parent"
        android.layout_height="wrap_content">
</ListView>
```

Sin embargo, si el único componente gráfico que va a mostrar nuestra actividad es una lista, es mucho más recomendable hacer que dicha actividad herede de `ListActivity`. Hacer esto simplifica mucho el manejo de listas. Aparte de disponer de todas las funcionalidades de una actividad normal dispondremos de otras relacionadas con listas como por ejemplo un método predefinido para manejar el caso en el que se pulse un elemento de la misma.

Una actividad de tipo `ListActivity` contendrá un adaptador de tipo `ListAdapter` que se encargará de manejar los datos de la lista. Este adaptador deberá ser inicializado en el método `onCreate` de la actividad por medio del método `setListAdapter`. Android proporciona diferentes layouts para asociarlos a los diferentes elementos de un adaptador. Además tendremos la oportunidad de implementar el manejador del evento `onListItemClicked` para realizar alguna acción cuado se escoja una opción de la lista.

7.2.2. Adaptadores

Un `ListView` obtiene los datos a mostrar a través de un adaptador, que debe ser una subclase de `BaseAdapter` y proporciona un modelo de datos que convierta los datos en diferentes elementos de la lista. Android dispone de dos adaptadores estándar: `ArrayAdapter` y `CursorAdapter`. El primero permite manejar datos basados en arrays, como por ejemplo datos almacenados en un `ArrayList`, mientras que el segundo permite manejar datos provenientes de una base de datos. También podemos crear nuestro propio adaptador creando una subclase de `BaseAdapter`.

El método más importante de un adaptador es `getView`, el cual es llamado por cada línea del `ListView` para determinar qué debe ser mostrado en cada fila de la lista y cómo. Normalmente una lista tendrá más elementos de los que caben en la pantalla, por lo que no todos los elementos serán visibles simultáneamente. Es por ello que existe un parámetro llamado `convertView` en `getView` de tal forma que se puedan reutilizar filas y

rellenar sus vistas asociadas (cada fila de la lista será una vista) con los nuevos datos a mostrar.

7.2.3. Nuestra primera ListActivity

Veamos primero cómo crear una `ListActivity` sencilla basada en los elementos que proporciona Android por defecto: utilizaremos un adaptador de tipo `ArrayAdapter` y un layout para las filas predefinido en Android.

En primer lugar creamos un nuevo proyecto que contendrá una única actividad, a la que llamaremos *MiLista*. Esta actividad será una subclase de `ListActivity`. El aspecto que tendrá es el siguiente. Fíjate en que no se llama a `setContentView` desde el método `onCreate` y que por lo tanto no es necesario modificar para nada el archivo de layout *main.xml* que se crea por defecto entre los recursos de la aplicación:

```
public class MiLista extends ListActivity {
        public void onCreate(Bundle icicle) {
                super.onCreate(icicle);
        }
}
```

A continuación vamos a crear un adaptador que sirva para rellenar el listado con los elementos almacenados en un array de cadenas. Debemos añadir lo siguiente al final del método `onCreate`:

```
String[] valores = new String[] {"C", "Java", "C++", "Python", "Perl",
        "PHP", "Haskell", "Eiffel", "Lisp",
        "Pascal", "Cobol", "Prolog"};
ArrayAdapter<String> adaptador = new ArrayAdapter<String>(this,
        android.R.layout.simple_list_item_1,valores);
setListAdapter(adaptador);
```

El adaptador es de tipo `ArrayAdapter`. Al constructor le pasamos como parámetro el contexto de la aplicación, el identificador de un layout definido por defecto en Android para mostrar los elementos de la lista (`android.R.layout.simple_list_item_1`) y el array de cadenas. Por último asociamos el adaptador a la lista. Ahora podríamos ejecutar la aplicación y comprobar que efectivamente se muestran los elementos de la lista.

Ejemplo de ListActivity

El siguiente paso podría ser implementar un manjeador para el evento `onListItemClick`. En nuestro caso vamos a hacer que cada vez que se seleccione un elemento de la lista se muestre un `Toast` (ver ejercicios de la primera sesión) mostrando el elemento seleccionado. Añademos el siguiente código dentro de la clase *MiLista*:

```
@Override
protected void onListItemClick(ListView l, View v, int position, long id)
{
        String elemento = (String)getListAdapter().getItem(position);
        Toast.makeText(this, elemento + " seleccionado",
Toast.LENGTH_LONG).show();
}
```

7.2.4. Listas con un layout personalizado

Un siguiente paso lógico podría ser estudiar cómo podemos modificar el layout que se utilizará para visualizar cada elemento de una lista, en lugar de aplicar un layout por defecto, como se hizo en el ejemplo anterior. Ten en cuenta que al crear un layout personalizado éste se asignará a todos y cada uno de los elementos de la lista, por lo que todos ellos se mostrarán con el mismo aspecto. En este ejemplo añadiremos en cada fila una imagen; para ello utilizaremos el archivo *icon.png* que viene incluido por defecto en los recursos de cualquier aplicación Android que creamos desde Eclipse.

Creamos un nuevo fichero llamado *layoutfila.xml* en la carpeta */res/layout/* de nuestra aplicación. Dicho fichero tendrá el siguiente contenido:

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

```
        android:id="@+id/icon"
        android:layout_width="22px"
        android:layout_height="22px"
        android:layout_marginLeft="4px"
        android:layout_marginRight="10px"
        android:layout_marginTop="4px"
        android:src="@drawable/icon">
    </ImageView>

    <TextView
        android:id="@+id/etiqueta"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@+id/etiqueta"
        android:textSize="20px">
    </TextView>
</LinearLayout>
```

El anterior fichero define el layout de cada fila de la lista. Cada una se compondrá de un `LinearLayout` vertical que a su vez contendrá dos elementos: un `ImageView` y el `TextView` donde mostraremos los elementos de la lista. A este `TextView` le hemos dado como valor de su atributo `android:id` el valor `"@+id/etiqueta"`. Será necesario que recordemos esto al inicializar el adaptador.

Modificamos ahora el código de nuestra actividad para que quede de la siguiente forma. El único cambio que estamos realizando es indicar al constructor del adaptador que queremos utilizar nuestro propio fichero *layoutfila.xml* para especificar cómo queremos mostrar los elementos de la lista por pantalla, y que en cada fila la cadena se mostrará en la vista cuyo identificador es `etiqueta` (lo cual se corresponde con el `TextView` en dicho archivo de layout).

```
String[] valores = new String[] {"C", "Java", "C++", "Python", "Perl",
        "PHP", "Haskell", "Eiffel", "Lisp",
        "Pascal", "Cobol", "Prolog"};
ArrayAdapter<String> adaptador = new ArrayAdapter<String>(this,
        R.layout.layoutfila, R.id.etiqueta, valores);
setListAdapter(adaptador);
```

Layout personalizado para los elementos de una lista

7.2.5. El evento onItemLongClick

Otro evento relativo a las listas para el que podemos añadir un manejadir es `onItemLongClick`. Este evento se disparará cuando el usario mantenga pulsado un elemento de la lista durante un periodo de tiempo más prolongado de lo habitual. El manejador se asociará a la lista contenida por el `ListActivity` por medio de una llamada al método `setOnItemLongClickListener`. Esto va a requerir acceder a la vista `ListView` contenida por el `ListActivity`; esto puede hacerse por medio del método `getListView`. A continuación se muestra un código de ejemplo que podría ser añadido al que hemos estado mostrando hasta el momento, al final del método `onCreate`:

```
ListView list = getListView();
list.setOnItemLongClickListener(new OnItemLongClickListener() {
        public boolean onItemLongClick(AdapterView<?> parent, View view,
        int position, long id) {
                Toast.makeText(MiLista.this,
                        "Elemento nº " + position + " pulsado",
                        Toast.LENGTH_LONG).show();
                // Devolvemos el valor falso para evitar que se dispare
                // también el evento onListItemClick
                return true;
        }
});
```

7.2.6. Selección múltiple

Otra de las posibilidades de las listas en Android es permitir que haya varios elementos seleccionados simultáneamente. Para poder indicar el modo de selección de la lista utilizamos el método `setChoiceMode`, pasando como parámetro cualquiera de los dos valores que se presentan en el siguiente ejemplo:

```
ListView listView = getListView();
// Selección múltiple:
listView.setChoiceMode(ListView.CHOICE_MODE_MULTIPLE);
// Selección única:
listView.setChoiceMode(ListView.CHOICE_MODE_SINGLE);
```

Para poder acceder a los elementos seleccionados utilizamos o bien el método `getCheckedItemPosition` para el caso de selección única o bien el método `getCheckedItemPositions` para el caso de selección múltiple. Incluso es posible, si se ha establecido un identificador para los elementos, hacer uso del método `getCheckedItemIds`, el cual devolverá el identificador de los elementos seleccionados.

7.2.7. Modificando el layout de un ListActivity

Es posible modificar el layout de una actividad que herede de `ListActivity`. Podríamos definir por ejemplo un layout para el `ListActivity` compuesto por dos elementos `TextView` y una `ListView` entre ambos. Estos `TextView` se podrían utilizar como cabecera o pie de la lista. Si se hace esto hemos de tener en cuenta que se debe asignar al `ListView` el identificador `android:id="@android:id/list"`, ya que éste es precisamente el identificador que busca el `ListView` para visualizar su lista asociada.

7.3. Menús

Los menús son una herramienta estándar en las aplicaciones Android que permite añadir más funcionalidad sin necesidad de sacrificar el valioso y limitado espacio del que disponemos en la pantalla para mostrar elementos. Cada actividad puede tener asociado su propio menú, el cual se mostrará cuando se pulse el botón de menú del dispositivo.

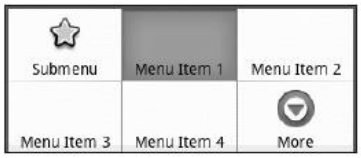
Android también permite la creación de menús contextuales, los cuales pueden ser asociados a una vista cualquiera. El menú contextual de una vista se activa habitualmente cuando ésta vista tiene el foco y se pulsa la pantalla durante al menos tres segundos.

Estos dos tipos de menú soportan la inclusión de submenús, iconos, atajos de teclado y la inclusión de elementos como checkboxes y botones de radio.

7.3.1. El sistema de menús de Android

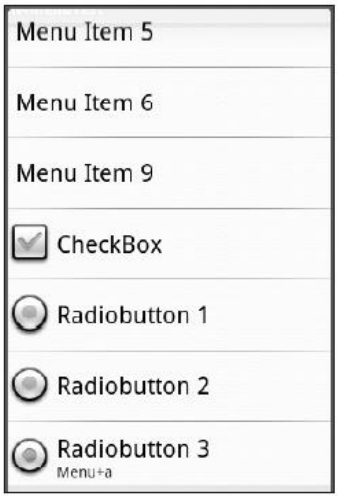
Android dispone de un sistema de menús para las aplicaciones basado en tres etapas. Este sistema está optimizado para pantallas de pequeño tamaño:

- **Menú de iconos**: se trata de un menú compacto que aparece en pantalla cuando se pulsa el botón correspondiente del dispositivo. El menú muestra los iconos y el texto asociados a un número limitado de opciones (seis como máximo). Normalmente se suelen utilizar iconos en tonos de gris. Este menú no muestra checkboxes, botones de radio o atajos. Si el menú de la actividad contiene más de seis elementos aparecerá una opción que permitirá mostrar un menú extendido con las opciones adicionales. Pulsando el botón *BACK* en el dispositivo cerramos el menú.



Menú de iconos

- **Menú extendido**: este menú se muestra cuando el menú de iconos tiene demasiadas opciones y se elecciona la opción correspondiente para mostrar los elementos adicionales. Este menú muestra un listado por el que nos podemos mover con una barra de scroll que tan solo contendrá aquellas opciones que no se pudieron mostrar en el menú de iconos. Además, se permite la visualización de atajos, checkboxes y botones de radio. Sin embargo, no se mostrará ningún icono. Pulsando el botón *BACK* en el dispositivo volveremos al menú de iconos.



Menú extendido

- **Submenús**: Android muestra cada submenú en una ventana flotante. Un submenú se visualiza junto con su nombre, y puede contener checkboxes, botones de radio o atajos. Hemos de tener en cuenta que Android no soporta la creación de submenús anidados, por lo que no se podrá añadir un submenú a otro ya existente (intentar esto provocará una excepción). En este caso tampoco se muestran los iconos de las opciones. Al pulsar el botón *BACK* del dispositivo volvemos al menú desde el cual se hizo la llamada al submenú.

7.3.2. Definir el menú de una actividad

Para definir el menú de una actividad se debe sobrecargar su método `onCreateOptionsMenu`. La primera vez que se vaya a mostrar el menú de la actividad se llamará a este método, el cual recibe como parámetro un objeto de la clase `Menu`. A la hora de sobrecargar el método no debemos olvidar llamar al método correspondiente de la

superclase, pues éste se encargará de incluir de manera automática opciones adicionales cuando esto sea necesario.

Para añadir opciones a un `Menu` utilizamos el método `add`. Se deben especificar los siguientes datos para cada nuevo elemento de un menú:

- Un entero conocido como identificador de grupo que permita separar los elementos del menú en tareas de procesamiento por lotes u ordenación.
- Un identificador único (un entero) para cada elemento del menú. Esto servirá más adelante en el manejador de eventos adecuado determinar cuál de las opciones de un menú fue seleccionada. Se suele seguir la convención de declarar cada identificador como un atributo privado estático dentro de la actividad correspondiente. Otra opción podría ser utilizar la constante `Menu.FIRST` e ir incrementando este valor en cada elemento posterior.
- Un entero que indique el orden en el que los elementos serán añadidos al menú.
- El texto a mostrar en la opción, ya sea en forma de cadena o ya sea en forma de identificador de recurso (en este caso, el identificador debería corresponderse a una cadena definida en el archivo *strings.xml* de los recursos de la aplicación).

Una vez que terminemos de añadir todas las opciones del menú deberemos devolver `true`. A continuación se muestra un ejemplo en el que se añade una opción al menú de una actividad:

```
static final private int MI_ELEMENTO = Menu.FIRST;

@Override
public boolean onCreateOptionsMenu(Menu menu) {
        super.onCreateOptionsMenu(menu);
        // Identificador de grupo
        int groupId = 0;
        // Identificador único del evento. Utilizado en
        // el manejador de evento correspondiente
        int menuItemId = MI_ELEMENTO;
        // Posición del elemento en el menú
        int menuItemOrder = Menu.NONE;
        // Texto que mostrará el menú de opciones
        int menuItemText = R.string.menu_item;
        // Creamos el elemento con todos estos datos
        MenuItem menuItem = menu.add(groupId, menuItemId,
                                                menuItemOrder,
menuItemText);
        return true;
}
```

Podemos mantener una referencia al menú creado en `onCreateOptionsMenu` para utilizarla en otras partes del código. Esta referencia será válida hasta que se vuelva a invocar este método. También podemos guardar una referencia a los diferentes objetos `MenuItem` de un menú, aunque también es posible acceder a ellos mediante una llamada al método `findItem` de la clase `Menu`.

7.3.3. Personalizar elementos de menús

En esta sección resumimos algunas de las opciones de las que disponemos a la hora de

diseñar los elementos de nuestros menús.

- **Checkboxes y botones de radio**: estos elementos tan sólo pueden ser visualizados en menús extendidos y en submenús. Para hacer que una opción pase a ser de tipo checkbox utilizamos el método `setCheckable`.Una vez hecho eso podemos controlar su estado mediante el método `setChecked`. Con respecto a los botones de radio, éstos deben ser organizados en grupos. Sólo uno de los botones del grupo de botones de radio podrá estar activo en un momento determinado; al seleccionar cualquiera de ellos, áquel que estuviera seleccionado dejará de estarlo. Para crear un grupo de botones de radio le asignamos el mismo identificador de grupo a todos ellos y a continuación llamamos al método `Menu.setGroupCheckable`, pasando como parámetro dicho identificador de grupo y dándole al parámetro `exclusive` el valor `true`. El siguiente código muestra un ejemplo en el que se añade un elemento de tipo checkbox y un grupo de tres botones de radio:

```
// Creamos un elemento de tipo checkbox
menu.add(0, ELEMENTO_CHECKBOX, Menu.NONE, "CheckBox").setCheckable(true);

// Creamos un grupo de botones de radio
menu.add(GRUPO_BR, BOTONRADIO_1, Menu.NONE, "Opción 1");
menu.add(GRUPO_BR, BOTONRADIO_2, Menu.NONE, "Opción 2");
menu.add(GRUPO_BR, BOTONRADIO_3, Menu.NONE, "Opción 3").setChecked(true);
menu.setGroupCheckable(GRUPO_BR, true, true);
```

- **Atajos de teclado**: es posible asociar un atajo de teclado a un determinado elemento de un menú por medio del método `setShortcut`. Cada llamada a este método requiere en realidad dos teclas, una para poder se usada con el teclado numérico y otra en el caso en el que se esté utilizando un teclado completo. En ninguno de los casos se distinguirá entre mayúsculas y minúsculas:

```
// Añadimos un atajo de teclado a esta opción del menú: '0' en el
// caso de utilizar el teclado numérico, o 'b' en el caso de
// utilizar un teclado completo
menuItem.setShortcut('0','b');
```

- **Texto resumido**: el método `setTitleCondensed` se puede utilizar para especificar el texto asociado a una opción de menú cuando ésta se muestra en el menú de iconos. Teniendo en cuenta que en dicho menú no es posible mostrar checkboxes, a veces este texto se utiliza para indicar el estado de una determinada opción.

```
menuItem.setTitleCondensed("Texto corto");
```

- **Iconos**: una de las propiedades de los elementos del menú es su icono, que se tratará del identificador de un recurso de tipo *Drawable*. Los iconos se muestran únicamente en el menú de iconos, es decir, no se pueden visualizar ni en un menú extendido ni en un submenú. Como se ha comentado anteriormente se suele adoptar la convención de utilizar imágenes en tonos de gris para los iconos de las opciones de los menús.

```
menuItem.setIcon(R.drawable.icono_opcion);
```

- **Manejador de evento click**: también es posible crear un manejador para el evento de pulsar sobre una opción del menú. Aunque esto sea así, por motivos de eficiencia se

desaconseja utilizar esta aproximación; es preferible hacer uso del método `onOptionsItemSelected`, tal como se mostrará una sección posterior.

```
menuItem.setOnMenuItemClickListener(new OnMenuItemClickListener() {
        public boolean onMenuItemClick(MenuItem _menuItem) {
                [ ... hacer algo, devolver true si todo correcto ... ]
                return true;
        }
});
```

- **Intents**: el `Intent` asociado a una opción de menú se activará cuando el evento de seleccionar dicha opción no es manejado ni por `MenuItemClickLister` ni por `onOptionsItemSelected`. Al activarse el `Intent` se hará una llamada a `startActivity` pasando como parámetro dicho `Intent`.

```
menuItem.setIntent(new Intent(this, OtraActividad.class));
```

7.3.4. Actualización dinámica de opciones

Sobrecargar el método `onPrepareOptionsMenu` permite modificar un `Menu` según el estado actual de la aplicación de manera inmediatamente anterior a que dicho menú sea mostrado por pantalla. Esto permite realizar operaciones como añadir o eleminar opciones de manera dinámica, modificar la visibilidad de los diferentes elementos de un menú, o modificar el texto.

Para modificar un determinado elemento de un menú tenemos dos opciones: o bien nos guardamos una referencia al elemento en `onCreateOptionsMenu` cuando éste se crea, o bien hacemos uso del método `findItem` de la clase `Menu`. Este segundo método es el que se utiliza en el siguiente ejemplo:

```
@Override
public boolean onPrepareOptionsMenu(Menu menu) {
        super.onPrepareOptionsMenu(menu);

        MenuItem menuItem = menu.findItem(ELEMENTO_MENU);
        [ ... modificar el elemento del menú ... ]
        return true;
}
```

7.3.5. Manejo de la selección de elementos

Android maneja toda la actividad relacionada con la selección de opciones de un menú a través de un único manejador de evento: `onOptionsItemSelected`. El elemento del menú que fue seleccionado se pasa a la función como parámetro. Para decidir qué acción realizar, debemos omprar el valor del método `getItemId` del parámetro recibido con aquellos identificadores que utilizamos a la hora de rellenar el menú con opciones:

```
public boolean onOptionsItemSelected(MenuItem elemento) {
        super.onOptionsItemSelected(elemento);

        // Comprobamos qué elemento del menú fue seleccionado
        switch (elemento.getItemId()) {
                // Comparamos con los identificadores definidos en
```

```
                // nuestra actividad
                case (ELEMENTO_MENU):
                [ ... hacer algo ... ]
                return true;
        }

        // Devolvemos false si no hemos hecho nada con el
        // elemento seleccionado
        return false;
}
```

7.3.6. Submenús

Tradicionalmente, en otro tipo de aplicaciones, los submenús se muestran como un árbol jerárquico de opciones. Android presenta un planteamiento alternativo con el objetivo de simplificar la navegación a través de menú en dispositivos cuya pantalla tiene un tamaño reducido. En lugar de una estructura en forma de árbol, un submenú se muestra como una venta flotante.

Para añadir un submenú utilizamos el método `addSubMenu`, el cual recibe los mismos parámetros usados para crear un nuevo elemento de menú mediante el método `add`, lo cual quiere decir que podemos especificar un grupo, un identificador y un texto. También podemos hacer uso de los métodos `setHeaderIcon` y `setIcon` para especificar el icono a mostrar en la cabecera del submenú o en su opción correspondiente en el menú de iconos, respectivamente.

Dentro de un submenú podemos tener cualquier tipo de elementos que también puedan ser añadidos a un menú de iconos o un menú extendido. La única restricción que debemos tener en cuenta es que en Android no es posible tener submenús anidados, es decir, submenús que contengan alguna opción que a su vez se corresponda con un submenú.

El siguiente código muestra un ejemplo en el que dentro de `onCreateMenuOptions` se añade un submenú al menú principal, estableciendo el icono de su cabecera:

```
SubMenu sub = menu.addSubMenu(0, 0, Menu.NONE, "Submenú");
sub.setHeaderIcon(R.drawable.icon);
sub.setIcon(R.drawable.icon);

MenuItem elementoSubmenu = sub.add(0, 0, Menu.NONE, "Elemento submenú");
```

7.3.7. Menús contextuales

Una vista puede tener asociada un menú contextual. Si esa vista tiene el foco y se mantiene pulsada la pantalla durante tres segundos, el menú contextual aparecerá. Podemos definir menús contextuales y añadirles opciones de la misma forma que se hace con el menú de una actividad.

Para crear un menú contextual sobrecargamos el método `onCreateContextMenu` de la actividad y registramos las vistas de dicha actividad que deben usar el menú por medio del método `registerForContextMenu`. Primero vemos como registrar una vista:

```
@Override
public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        EditText vista = new EditText(this);
        setContentView(vista);
        registerForContextMenu(vista);
}
```

Una vez que se ha registrado una vista, el método `onCreateContextMenu` será invocado la primera vez que el menú contextual de dicha vista deba ser mostrado. Para que todo funcione correctamente sobrecargamos dicho método, comprobando para qué vista fue lanzado, de tal forma que rellenemos el menú con las opciones correctas.

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

        super.onCreateContextMenu(menu, v, menuInfo);

        menu.setHeaderTitle("Menú contextual");

        menu.add(0, menu.FIRST, Menu.NONE,
                "Elemento 1").setIcon(R.drawable.menu_item);
        menu.add(0, menu.FIRST+1, Menu.NONE,
                "Elemento 2").setCheckable(true);
        menu.add(0, menu.FIRST+2, Menu.NONE,
                "Elemento 3").setShortcut('3','3');
        SubMenu sub = menu.addSubMenu("Submenú");
        sub.add("Elemento de submenú");
}
```

Obsérvese como es posible utilizar con un objeto `ContextMenu` el método `add` de la misma forma que se utiliza con el menú de una actividad, con lo que es posible asociar elementos a un menú contextual como se haría en el caso del menú de una actividad. Esto incluye por supuesto la posibilidad de añadir submenús. Sin embargo, hemos de tener en cuenta que los iconos no se mostrarán.

El evento correspondiente a la selección de un elemento del menú contextual se maneja de la misma forma que en el caso del menú de una actividad. Se puede asociar un `Intent` o un manejador de evento a cada elemento individual, o bien se puede sobrecargar el método `onContextItemSelected` para manejar el evento para el menú en su conjunto, lo cual suele ser una opción más recomendada por cuestiones de eficiencia. El aspecto del manejador sería el siguiente:

```
@Override
public boolean onContextItemSelected(MenuItem item) {

        super.onContextItemSelected(item);
        [ ... hacer algo ... ]
        return false;
}
```

7.3.8. Definiendo menús como recursos

La alternativa más sencilla a la hora de diseñar el menú de nuestra actividad es por medio

de recursos XML. Como en el caso de los layouts, esto nos permite crear menús para diferentes configuraciones de hardware o idiomas. Por ejemplo, una opción podría ser mostrar menús con menos elementos en el caso de pantallas más pequeñas.

Los archivos XML para definir menús se deben almacenar dentro de la carpeta de los recursos de la aplicación, en */res/menu/*. Cada menú se debe crear en un fichero separado. El nombre del fichero pasará a ser el identificador del recurso.

El archivo contendrá un elemento raíz `menu`, que a su vez contendrá varios elementos `item` para especificar cada elemento del menú. Los atributos el elemento `item` son usados para indicar las propiedades del elemento, como su texto asociado, su icono, atajo de teclado, etc. Crear un submenú es tan sencillo como añadir un elemento `menu` en el interior de un elemento `item`.

El siguiente ejemplo muestra cómo definir el menú contextual de ejemplo de la sección anterior como un recurso XML:

```
<menu
                xmlns:android="http://schemas.android.com/apk/res/android"
                android:name="Menú contextual">
        <item
                android:id="@+id/elemento01"
                android:icon="@drawable/elemento_menu"
                android:title="Elemento 1">
        </item>
        <item
                android:id="@+id/elemento02"
                android:checkable="true"
                android:title="Elemento 2">
        </item>
        <item
                android:id="@+id/elemento03"
                android:numericShortcut="3"
                android:alphabeticShortcut="3"
                android:title="Elemento 3">
        </item>
        <item
                android:id="@+id/elemento04"
                android:title="Submenú">
                <menu>
                        <item>
                                android:id="@+id/elemento05"
                                android:title="Elemento de submenú">
                        </item>
                </menu>
        </item>
</menu>
```

Para asociar un menú definido como recurso a una actividad debemos añadir el siguiente código a ésta última:

```
public boolean onCreateOptionsMenu(Menu menu){
        MenuInflater inflater = getMenuInflater();
        // En este caso se usa el fichero de recurso menu.xml
        inflater.inflate(R.menu.menu, menu);
        return true
}
```

8. Menús, listas y barras de progreso - Ejercicios

8.1. Barra de progreso lineal

Crea una aplicación llamada *BarraProgreso* conteniendo una única actividad llamada *Principal*, cuya interfaz gráfica sea la siguiente:

Interfaz de la aplicación BarraProgreso

Como puedes observar, la interfaz se compone de dos cuadros de edición de texto, un botón y una barra de progreso lineal. Los valores de los dos cuadros de edición marcan un valor inicial y un valor final para el proceso que se lanzará al pulsar el botón. Al pulsar el botón, una propiedad de la actividad llamada `cuenta` deberá ir avanzando desde el valor inicial hasta el valor final, haciendo una pausa de un segundo tras cada incremento.

Mientras se realice este progreso tanto el botón como los cuadros de edición de texto deberán estar deshabilitados, y la barra de progreso deberá ir mostrando la evolución del mismo, avanzando tras cada incremento. Cuando finalice el proceso, todas las vistas deberán volver a estar habilitadas.

Por último, antes de comenzar el progreso, hemos de asegurarnos de que el valor inicial sea menor que el valor final. En caso contrario el proceso no se lanzará y se deberá mostrar un `Toast` con un mensaje de error.

Nota:

Para habilitar o deshabilitar una vista invocamos a su método `setEnabled` pasándole como parámetro `true` o `false`, respectivamente.

Nota:

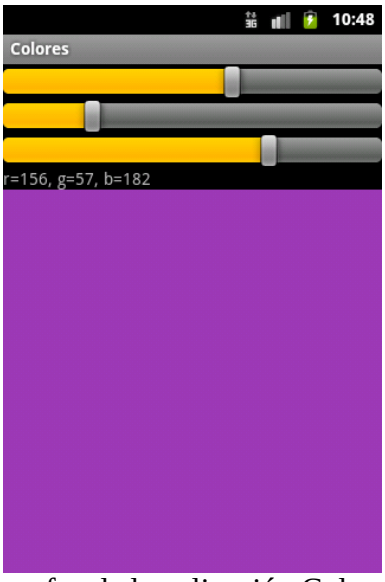
El valor máximo de una barra de progreso se puede establecer desde el código con el método `setMax` de `ProgressBar`.

Aviso:

Si intentamos modificar algún atributo de alguna de las vistas desde el método `run` del elemento `Runnable` se producirá una excepción. Sólo podremos modificar la interfaz gráfica desde el `Handler`.

8.2. Selección de color

En este ejercicio vamos a crear una aplicación basada en vistas de tipo `SeekBar` llamada *Colores*. La aplicación permitirá visualizar colores para diferentes valores de las tres componentes básicas rojo (r), verde (v) y azul (a) de manera dinámica. El aspecto de la interfaz será el siguiente:



Interfaz de la aplicación Colores

Cada uno de los tres `SeekBar` permitirá modificar el valor de cada una de las componentes básicas. El valor de estos `SeekBar` será como mínimo de 0 y como máximo de 255. Debajo de ellos se mostrará mediante un `TextView` el valor de cada componente de color. Finalmente, tendremos un último `TextView` sin texto asociado y cuya propiedad `android:layout_height` valdrá `fill_parent`. El color de fondo de éste se obtendrá a partir de la mezcla de la proporción indicada de los tres colores básicos.

Nota:

El color de fondo de un elemento `TextView` puede ser modificado con su método `setBackground`, que debe recibir como parámetro un color. Para indicar el color utilizamos la clase `Color`, y más concretamente el método estático `Color.rgb()`, que recibe tres parámetros: el valor de rojo, verde y azul.

8.3. Lista de tareas

El objetivo de este ejercicio será crear una sencilla aplicación llamada *ListaTareas* para gestionar una lista de tareas. La aplicación tendrá una única actividad, que heredará de `ListActivity`, y en cuya interfaz tan sólo habrá un cuadro de edición de texto, un botón y una vista de tipo `ListView`. Cada vez que pulsemos el botón, el texto introducido en el cuadro de edición se añadirá a la lista como una nueva lista de tareas y el contenido de dicho cuadro de edición se borrará para dejar paso a la nueva tarea. Tienes una figura mostrando un ejemplo de la aplicación en ejecución al final del enunciado del ejercicio.

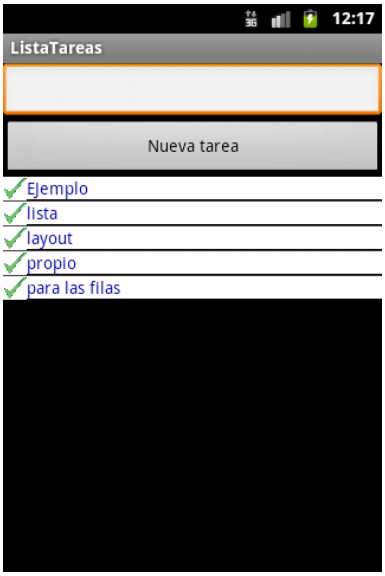
Para completar el ejercicio debes seguir los sigiuentes pasos:

- Crea la aplicación *ListaTareas*, con una actividad llamada *Principal* que debe heredar de `ListActivity`.
- Modifica el layout de la actividad para que incluya un `EditText` y un `Button` en el borde superior, además de un componente `ListView`. Recuerda para que dicho `ListView` pueda ser manejado por la actividad, su identificador debe ser `android:id="@android:id/list"`.
- Añade a la clase *Principal* un `ArrayList` de cadenas, que servirá para almacenar las tareas que el usuario vaya introduciendo. Inicializalo en el método `onCreate` con el operador `new`, pero de momento no introduzcas ninguna cadena.
- Añade a la clase principal un objeto de tipo `ArrayAdapter<String>`. Inicialízalo y asocialo a la lista de la actividad. Al inicializar el adaptador, recuerda que el primer parámetro debe ser el contexto de la aplicación (puedes usar `this`), para el segundo utilizaremos el identificador del layout por defecto para las filas `android.R.layout.simple_list_item_1`, y el tercero será el `AraryList` que creaste en el punto anterior.
- Añade un manejador al botón. Cada vez que se pulse el botón se debe incorporar el texto del `EditText` al `ArrayList` de cadenas y luego limpiar el `EditText`.
- Por último, en ese mismo manejador, invoca al método `notifyDataSetChanged` del adaptador para que la lista se actualice cada vez que se introduzca un elemento en el `ArrayList`

Interfaz de la aplicación ListaTareas

8.4. Modificando el aspecto de la lista de tareas (*)

Crea un layout personalizado para las filas de la lista de la aplicación *ListaTareas*. Cada elemento de la lista debe mostrarse con texto azul oscuro sobre fondo blanco. A la izquierda de cada elemento de la lista debe aparecer la imagen *check.png* que se te proporciona en las plantillas de la sesión. Tanto la anchura como la altura de la imagen será de 20dip.



Interfaz de la aplicación ListaTareas con un layout propio para las filas

8.5. Menú contextual (*)

Basándonos también en el ejercicio de la lista de tareas, el siguiente paso que vamos a seguir es añadir un menú contextual. Tras realizar una pulsación larga sobre alguno de los elementos de la lista de tareas se debe mostrar un menú contextual con dos posibles operaciones: eliminar el elemento de la lista de tareas o duplicarlo, colocando la nueva copia de la tarea al final.

Para que todo funcione correctamente debes asociar el menú contextual al `ListView` y desde allí acceder al elemento seleccionado, en lugar de intentar asociar un menú contextual a cada uno de los elementos individuales de la lista.

El menú contextual de la lista de tareas

Nota:

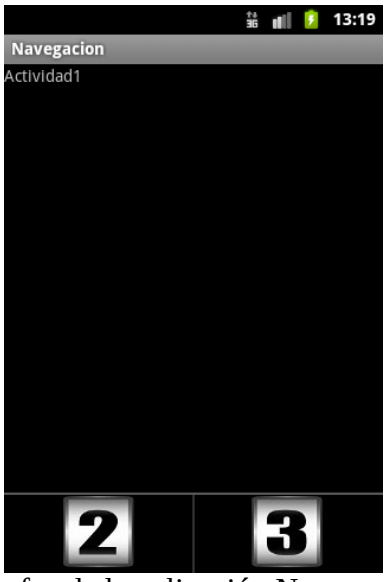
Para saber dentro de `onContextItemSelected` para qué elemento de la lista se mostró el menú contextual, debemos introducir en dicha función la línea `AdapterView.AdapterContextMenuInfo info = (AdapterView.AdapterContextMenuInfo)item.getMenuInfo();`. Una vez hecho esto, podremos acceder al entero `info.position` para obtener esta información.

8.6. Lanzando actividades desde un menú

En este ejercicio vamos a crear una aplicación llamada *Navegacion* para navegar entre tres actividades. Las tres actividades tendrán de nombre *Actividad1*, *Actividad2* y *Actividad3*. La interfaz gráfica de cada una de ellas consistirá únicamente en un `TextView` mostrnado su nombre. Desde cada una de ellas se podrá acceder a las otras dos mediante

un menú de iconos. Esto quiere decir que el menú de cada actividad será distinto y se compondrá de dos únicas opciones.

En las plantillas de la sesión se incluyen tres archivos gráficos llamados *icono1.png*, *icono2.png* y *icono3.png* que deberemos utilizar en los menús.



Interfaz de la aplicación Navegacion

Nota:

Asigna al atributo `android:noHistory` de cada actividad en el *Manifest* de la aplicación el valor `true`.

9. Drawables, estilos y temas

En esta sesión trataremos varias herramientas de Android que nos permitirán personalizar la interfaz de nuestras aplicaciones. En primer lugar hablaremos de los *Drawables*, recursos de tipo imagen que pueden ser usados de varias formas distintas: para mostrarlos tal cual en una actividad, para definir el fondo de una determinada vista, etc. A cotinuación hablaremos de estilos y temas en Android. Los estilos y temas siguen la misma filosofía que el lenguaje CSS en el caso del desarrollo web: separar el diseño del contenido. De esta forma seríamos capaces de modificar el aspecto de un conjunto de vistas o el aspecto global de la aplicación sin necesidad de modificar ningún archivo de código o de layout.

9.1. Elementos drawables

Un *drawable* es un tipo de recurso que puede ser dibujado en pantalla. Podremos utilizarlos para especificar el aspecto que van a tener los diferentes componentes de la interfaz, o partes de éstos. Estos *drawables* podrán ser definidos en XML o de forma programática. Entre los diferentes tipos de *drawables* existentes encontramos:

- **Color**: Rellena el lienzo de un determinado color.
- **Gradiente**: Rellena el lienzo con un gradiente.
- **Forma** (*shape*): Se pueden definir una serie de primitivas geométricas básicas como *drawables*.
- **Imágen** (*bitmap*): Una imagen se comporta como *drawable*, ya que podrá ser dibujada y referenciada de la misma forma que el resto.
- **Nine-patch**: Tipo especial de imagen PNG que al ser escalada sólo se escala su parte central, pero no su marco.
- **Animación**: Define una animación por fotogramas, como veremos más adelante.
- **Capa** (*layer list*): Es un *drawable* que contiene otros *drawables*. Cada uno especificará la posición en la que se ubica dentro de la capa.
- **Estados** (*state list*): Este *drawable* puede mostrar diferentes contenidos (que a su vez son *drawables*) según el estado en el que se encuentre. Por ejemplo sirve para definir un botón, que se mostrará de forma distinta según si está normal, presionado, o inhabilitado.
- **Niveles** (*level list*): Similar al anterior, pero en este caso cada *item* tiene asignado un valor numérico (nivel). Al establecer el nivel del *drawable* se mostrará el *item* cuyo nivel sea mayor o igual que el indicado.
- **Transición** (*transition*): Nos permite mostrar una transición de un *drawable* a otro mediante un fundido.
- **Inserción** (*inset*): Ubica un *drawable* dentro de otro, en la posición especificada.
- **Recorte** (*clip*): Realiza un recorte de un *drawable*.
- **Escala** (*scale*): Cambia el tamaño de un *drawable*.

Todos los *drawables* derivan de la clase `Drawable`. Esta nos permite que todos ellos puedan ser utilizados de la misma forma, independientemente del tipo del que se trate. Se puede consultar la lista completa de *drawables* y su especificación en la siguiente dirección:

http://developer.android.com/guide/topics/resources/drawable-resource.html

Por ejemplo, vamos a definir un *drawable* que muestre un rectángulo rojo con borde azul, creando un fichero XML de nombre `rectangulo.xml` en el directorio */res/drawable/*. El fichero puede tener el siguiente contenido:

```
<?xml version="1.0" encoding="utf-8"?> <shape
xmlns:android="http://schemas.android.com/apk/res/android"
        android:shape="rectangle">
         <solid android:color="#ff0000"/>
         <stroke android:width="2dp" android:color="#0000ff"
            android:dashWidth="10dp" android:dashGap="5dp"/>
</shape>
```

Podremos hacer referencia a este rectángulo desde el código mediante `R.drawable.rectangulo` y mostrarlo en la interfaz asignándolo a un componente de alto nivel como por ejemplo `ImageView`, o bien hacer referencia a él desde un atributo del XML mediante `@drawable/rectangulo`. Por ejemplo, podríamos especificar este *drawable* en el atributo `android:background` de la etiqueta `Button` dentro de nuestro *layout*, para que así el botón pase a tener como aspecto una forma rectangular de color rojo y con borde azul. De la misma forma podríamos darle al botón el aspecto de cualquier otro tipo de *drawable* de los vistos anteriormente. A continuación vamos a ver con más detalle los tipos de *drawables* más interesantes.

Nota

Como en el caso de otros recursos, podemos definir diferentes variantes del directorio de *drawables*, para así definir diferentes versiones de los *drawables* para los distintos tipos de dispositivos, por ejemplo según su densidad, tamaño, o forma de pantalla.

9.1.1. Colores

Los *ColorDrawables* son los *drawables* más sencilllos. Permiten definir una imagen simplemente a partir de un único color sólido. Se definen usando la etiqueta `color` en XML, dentro de la carpeta de *drawables* en los recursos de la aplicación. A continuación mostramos un ejemplo de drawable de color rojo

```
<color
        xmlns:android="http://schemas.android.com/apk/res/android"
        android:color="#FF0000"
/>
```

9.1.2. Formas

Ya hemos visto anteriormente un ejemplo de *drawable* de tipo forma, pero en esta sección examineramos este tipo de elementos en más detalle. Mediante el elemento `shape` podremos definir formas simples indicando sus dimensiones, fondo y bordes.

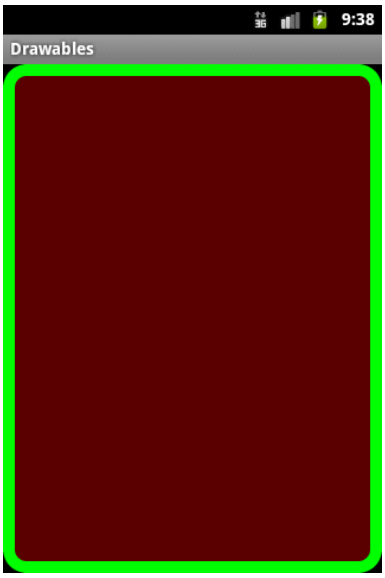
Cada forma será de un tipo determinado. El tipo se especifica mediante un atributo del elemento `shape`. Los posibles valores que este atributo `shape` podrá tomar son los siguientes:

- `oval`: como su propio nombre indica este tipo permitirá definir formas de tipo oval.
- `rectangle`: permite el uso de un subnodo `corners` con un atributo `radius` para establecer bordes redondeados.
- `ring`: podemos definir el radio interno y la anchura del anillo mediante los atributos `innerRadius` y `thickness`, respectivamente. También es posible establecer estos valores en función de la anchura del *drawable* por medio de los atributos `innerRadiusRatio` y `thicknessRatio`.

Para especificar cómo será el borde de la figura podemos utilizar el elemento `stroke`. En concreto, deberemos hacer uso de los atributos `width` y `color`. Otro nodo que podemos utilizar es `padding`, que permitirá desplazar la figura en el lienzo, para que no aparezca centrada. De todas formas, el nodo más habitual suele ser `solid`, que mediante el atributo `color` nos permite establecer el color de fondo.

A continuación mostramos un ejemplo de forma, el aspecto de la cual se puede ver en la siguiente figura:

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android"
android:shape="rectangle">
        <solid android:color="#f0600000"/>
        <stroke
                android:width="10dp"
                android:color="#00FF00"/>
        <corners
                android:radius="15dp" />
        <padding
                android:left="10dp"
                android:top="10dp"
                android:right="10dp"
        android:bottom="10dp"
        />
</shape>
```



Ejemplo de drawable: forma rectangular

9.1.3. Gradientes

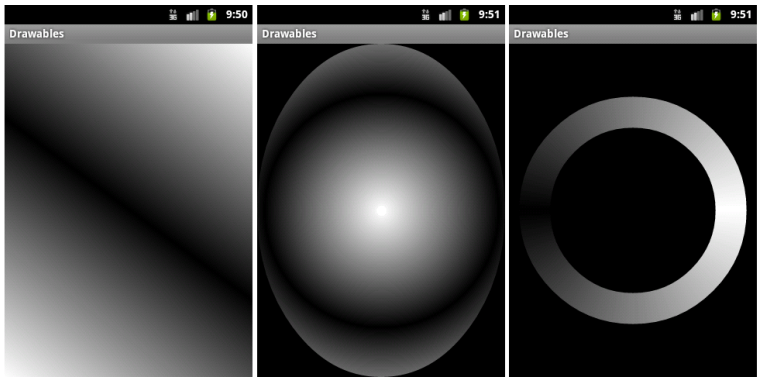
Un *GradientDrawable* permite incluir gradientes más o menos complejos en nuestra actividad. Se definen mediante una etiqueta `gradient` en XML. Cada *drawable* de este tipo requiere al menos el uso de los atributos `startColor` y `endColor`, que indican los dos colores entre los que se va a realizar la transición. También es posible el uso de un atributo `middleColor`, por lo que la transición se puede llegar a realizar entre tres colores diferentes. El atributo `type` permite definir el tipo de gradiente:

- `linear`: es el tipo por defecto. Muestra una transición directa desde `startColor` a `endColor`, con un ángulo que puede ser definido mediante el atributo `angle`.
- `radial`: dibuja un gradiente circular en el que se se produce una transición desde `startColor` en la parte más externa a `endColor` en el centro. Requiere el uso del atributo `gradientRadius` para indicar el radio del círculo a través del cual se producirá la transición. También es posible utilizar de manera opcional los atributos `centerX` y `centerY` para desplazar el centro del círculo. El atributo `gradientRadius` está definido en píxeles, por lo que será necesario definir un *drawable* de este estilo diferente para cada posible resolución de pantalla.
- `sweep`: el gradiente se muestra en el límite externo de la figura padre (normalmente un anillo).

El siguiente código muestra un ejemplo con tres gradientes diferentes. El resultado se puede contemplar en la siguiente figura:

```
<!-- Rectangulo con gradiente lineal -->
<?xml version="1.0" encoding="utf-8"?>
        <shape xmlns:android="http://schemas.android.com/apk/res/android"
                android:shape="rectangle"
                android:useLevel="false">
```

```
                <gradient
                        android:startColor="#ffffff"
                        android:endColor="#ffffff"
                        android:centerColor="#000000"
                        android:useLevel="false"
                        android:type="linear"
                        android:angle="45"
                        />
        </shape>
<!-- Oval gradiente radial -->
<?xml version="1.0" encoding="utf-8"?>
        <shape xmlns:android="http://schemas.android.com/apk/res/android"
                android:shape="oval"
                android:useLevel="false">
                <gradient
                        android:type="radial"
                        android:startColor="#ffffff"
                        android:endColor="#ffffff"
                        android:centerColor="#000000"
                        android:useLevel="false"
                        android:gradientRadius="300"
                />
        </shape>
<!-- Anillo con gradiente de tipo sweep -->
<?xml version="1.0" encoding="utf-8"?>
        <shape xmlns:android="http://schemas.android.com/apk/res/android"
                android:shape="ring"
                android:useLevel="false"
                android:innerRadiusRatio="3"
                android:thicknessRatio="8">
                <gradient
                        android:startColor="#ffffff"
                        android:endColor="#ffffff"
                        android:centerColor="#000000"
                        android:useLevel="false"
                        android:type="sweep"
                />
</shape>
```



Ejemplos de drawable a partir de gradientes

9.1.4. Imágenes

Las imágenes que introduzcamos en los directorios de recursos de tipo *drawable* (*/res/drawable/*) podrán ser tratadas igual que cualquier otro tipo de *drawable*. Por ejemplo, si introducimos en dicho directorio una imagen `titulo.png`, podremos hacer

referencia a ella en los atributos de los XML mediante `@drawable/titulo` (no se pone la extensión), o bien desde el código mediante `R.drawable.titulo`.

Las imágenes se encapsulan en la clase `Bitmap`. Los *bitmaps* pueden ser mutables o inmutables, según si se nos permite modificar el valor de sus pixels o no respectivamente.

Si el *bitmap* se crea a partir de un *array* de pixels, de un recurso con la imagen, o de otro *bitmap*, tendremos un *bitmap* inmutable.

Si creamos el *bitmap* vacío, simplemente especificando su altura y su anchura, entonces será mutable (en este caso no tendría sentido que fuese inmutable ya que sería imposible darle contenido). También podemos conseguir un *bitmap* mutable haciendo una copia de un *bitmap* existente mediante el método `copy`, indicando que queremos que el *bitmap* resultante sea mutable.

Para crear un *bitmap* vacío, a partir de un *array* de pixels, o a partir de otro *bitmap*, tenemos una serie de métodos estáticos `createBitmap` dentro de la clase `Bitmap`.

Para crear un *bitmap* a partir de un fichero de imagen (GIF, JPEG, o PNG, siendo este último el formato recomendado) utilizaremos la clase `BitmapFactory`. Dentro de ella tenemos varios métodos con prefijo `decode` que nos permiten leer las imágenes de diferentes formas: de un *array* de bytes en memoria, de un flujo de entrada, de un fichero, de una URL, o de un recurso de la aplicación. Por ejemplo, si tenemos una imagen (`titulo.png`) en el directorio de *drawables* podemos leerla como `Bitmap` de la siguiente forma:

```
Bitmap imagen = BitmapFactory.decodeResource(getResources(),
    R.drawable.titulo);
```

Al crear un *bitmap* a partir de otro, podremos realizar diferentes transformaciones (escalado, rotación, etc).

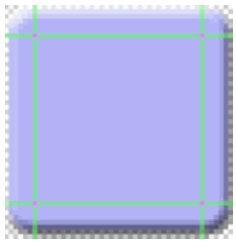
Una vez no se vaya a utilizar más el *bitmap*, es recomendable liberar la memoria que ocupa. Podemos hacer esto llamando a su método `recycle`.

9.1.5. Imágenes nine-patch

Como hemos comentado anteriormente, utilizaremos los *drawables* para especificar el aspecto que queremos que tengan los componentes de la interfaz. La forma más flexible de definir este aspecto es especificar una imagen propia. Sin embargo, encontramos el problema de que los componentes normalmente no tendrán siempre el mismo tamaño, sino que Android los "estirará" según su contenido y según los parámetros de *layout* especificados (es decir, si deben ajustarse a su contenido o llenar todo el espacio disponible). Esto es un problema, ya que si siempre especificamos la misma imagen como aspecto para estos componentes, al estirarla veremos que ésta se deforma, dando un aspecto terrible a nuestra aplicación, como podemos ver a continuación:

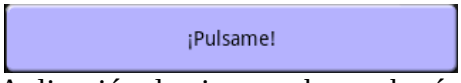
Imagen estirada

Sin embargo, tenemos un tipo especial de imágenes PNG llamadas *nine-patch* (llevan extensión `.9.png`), que nos permitirán evitar este problema. Normalmente la parte central de nuestros componentes es homogénea, por lo que no pasa nada si se estira. Sin embargo, los bordes si que contienen un mayor número de detalles, que no deberían ser deformados, especialmente las esquinas. Las imágenes *nine-patch* se dividen en 9 regiones: la parte central, que puede ser escalada en cualquier dirección, las esquinas, que nunca pueden escaladas, y los bordes, que sólo pueden ser escalados en su misma dirección (horizontal o vertical). A continuación vemos un ejemplo de dicha división:



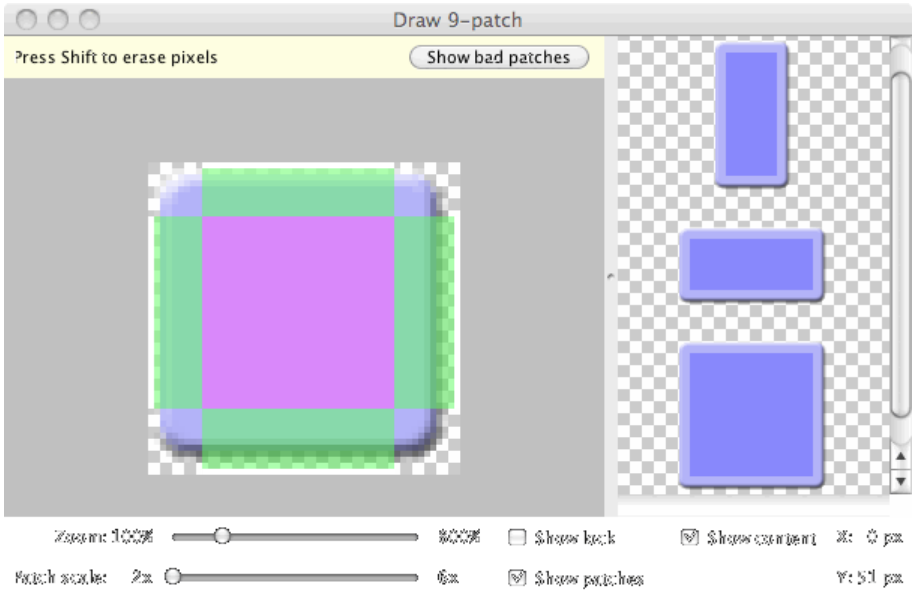
Parches de la imagen

Si ponemos una imagen de este tipo como *drawable* de fondo para un botón, veremos que siempre se mostrará con el aspecto correcto, independientemente de su contenido:



Aplicación de nine-patch a un botón

Podemos crear este tipo de imágenes con la herramienta `draw9patch` que podemos encontrar en el subdirectorio `tools` del SDK de Android. Lo único que necesitaremos es arrastrar el PNG que queramos tratar como *nine-patch*, y añadir una serie de píxeles en el marco de la imagen para marcar las regiones:



Herramienta draw9patch

La fila de píxeles superior y la columna izquierda indican las zonas de la imagen que son flexibles y que se pueden ampliar si es necesario repitiendo su contenido. En el caso de la fila superior, indica que se pueden estirar en la horizontal, mientras que los del lateral izquierdo corresponden a la vertical.

Opcionalmente podemos especificar en la fila inferior y en la columna derecha la zona que utilizaremos como contenido. Por ejemplo, si utilizamos la imagen como marco de un botón, esta será la zona donde se ubicará el texto que pongamos en el botón. Marcando la casilla *Show content* veremos en el lateral derecho de la herramienta una previsualización de la zona de contenido.

9.1.6. Lista de estados

Siguiendo con el ejemplo del botón, encontramos ahora un nuevo problema. Los botones no deben tener siempre el mismo aspecto de fondo, normalmente cambiarán de aspecto cuando están pulsados o seleccionados, sin embargo sólo tenemos la posibilidad de especificar un único *drawable* como fondo. Para poder personalizar el aspecto de todos los estados en los que se encuentra el botón tenemos un tipo de *drawable* llamado *state list drawable*. Se define en XML, y nos permitirá especificar un *drawable* diferente para cada estado en el que se puedan encontrar los componentes de la interfaz, de forma que en cada momento el componente mostrará el aspecto correspondiente a su estado actual.

Por ejemplo, podemos especificar los estados de un botón (no seleccionado, seleccionado, y pulsado) de la siguiente forma:

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

```
        <!-- presionado -->
    <item android:state_pressed="true"
          android:drawable="@drawable/boton_pressed" />
    <!-- seleccionado -->
    <item android:state_focused="true"
          android:drawable="@drawable/boton_selected" />
    <!-- no seleccionado -->
    <item android:drawable="@drawable/boton_normal" />
</selector>
```

Los *drawables* especificados para cada estado pueden ser de cualquier tipo (por ejemplo imágenes, *nine-patch*, o formas definidas en XML).

Un *drawable* similar es el de tipo *level list*, pero en este caso los diferentes posibles *drawables* a mostrar se especifican para un rango de valores numéricos. ¿Para qué tipos de componentes de la interfaz podría resultar esto de utilidad?

9.1.7. Definición por capas

Es posible crear un *drawable* a partir de la composición de capas de otros *drawables* diferentes. Para ello se utiliza el elemento `layer-list`. Este elemento contendrá subnodos de tipo `item`, cuyo atributo `drawable` indicará a qué *drawable* se está haciendo referencia. Los elementos se añadirán en orden, quedando el primer elemento en el fondo de la pila.

```
<?xml version="1.0" encoding="utf-8"?>
        <layer-list
xmlns:android="http://schemas.android.com/apk/res/android">
                <item android:drawable="@drawable/bottomimage"/>
                <item android:drawable="@drawable/image2"/>
                <item android:drawable="@drawable/image3"/>
                <item android:drawable="@drawable/topimage"/>
        </layer-list>
```

9.1.8. Animación por fotogramas

Este tipo de *drawable* nos permite definir una animación a partir de diferentes fotogramas, que deberemos especificar también como *drawables*, además del tiempo en milisegundos que durará el fotograma. Se definen en XML de la siguiente forma:

```
<animation-list
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:oneshot="false">
  <item android:drawable="@drawable/spr0" android:duration="50" />
  <item android:drawable="@drawable/spr1" android:duration="50" />
  <item android:drawable="@drawable/spr2" android:duration="50" />
</animation-list>
```

Además, la propiedad `oneshot` nos indica si la animación se va a reproducir sólo una vez o en bucle infinito. Al ponerla como `false` especificamos que se reproduzca de forma continuada.

Desde el código, podremos obtener la animación de la siguiente forma, considerando que

la hemos guardado en un fichero `animacion.xml`:

```
AnimationDrawable animFotogramas =
getResources().getDrawable(R.drawable.animacion);
```

De forma alternativa, podríamos haberla definido de forma programática de la siguiente forma:

```
BitmapDrawable f1 = (BitmapDrawable)getResources()
    .getDrawable(R.drawable.sprite0);
BitmapDrawable f2 = (BitmapDrawable)getResources()
    .getDrawable(R.drawable.sprite1);
BitmapDrawable f3 = (BitmapDrawable)getResources()
    .getDrawable(R.drawable.sprite2);
AnimationDrawable animFotogramas = new AnimationDrawable();

animFotogramas.addFrame(f1, 50);
animFotogramas.addFrame(f2, 50);
animFotogramas.addFrame(f3, 50);

animFotogramas.setOneShot(false);
```

Nota

La diferencia entre `Bitmap` y `BitmapDrawable` reside en que en el primer caso simplemente tenemos una imagen, mientras que en el segundo lo que tenemos es un *drawable* que encapsula una imagen, es decir, se le podrá proporcionar a cualquier componente que acepte *drawables* en general como entrada, y concretamente lo que dibujará será la imagen (`Bitmap`) que contiene.

Para que comience la reproducción deberemos llamar al método `start` de la animación:

```
animFotogramas.start();
```

De la misma forma, podemos detenerla con el método `stop`:

```
animFotogramas.stop();
```

Importante

El método `start` no puede ser llamado desde el método `onCreate` de nuestra actividad, ya que en ese momento el *drawable* todavía no está vinculado a la vista. Si lo que queremos es que se ponga en marcha nada más cargarse la actividad, el lugar idóneo para invocarlo es el evento `onWindowFocusChanged`. Lo recomendable será llamar a `start` cuando obtengamos el foco, y a `stop` cuando lo perdamos.

9.1.9. Definición programática

Vamos a suponer que tenemos un `ImageView` con identificador `visor` y un drawable de nombre `rectangulo`. Normalmente especificaremos directamente en el XML el *drawable* que queremos mostrar en el `ImageView`. Para ello deberemos añadir el atributo `android:src = "@drawable/rectangulo"` en la definición del `ImageView`.

Podremos también obtener una referencia a dicha vista y mostrar en ella nuestro

rectángulo especificando el identificador del *drawable* de la siguiente forma:

```
ImageView visor = (ImageView)findViewById(R.id.visor);
visor.setImageResource(R.drawable.rectangulo);
```

Otra alternativa para mostrarlo es obtener primero el objeto `Drawable` y posteriormente incluirlo en el `ImageView`:

```
Drawable rectangulo = this.getResources()
    .getDrawable(R.drawable.rectangulo);
visor.setImageDrawable(rectangulo);
```

Estas primitivas básicas también se pueden crear directamente de forma programática. En el paquete `android.graphics.drawable.shape` podemos encontrar clases que encapsulan diferentes formas geométricas. Podríamos crear el rectángulo de la siguiente forma:

```
RectShape r = new RectShape();
ShapeDrawable sd = new ShapeDrawable(r);
sd.getPaint().setColor(Color.RED);
sd.setIntrinsicWidth(100);
sd.setIntrinsicHeight(50);

visor.setImageDrawable(sd);
```

9.2. Estilos y temas

Los estilos permiten modificar el aspecto de una vista o una ventana. Con un estilo podemos definir propiedades como altura, padding, color de fuente, tamaño de fuente, y muchas más cosas. Un estilo se define en un archivo XML separado de la definición del layout. La filosofía de los estilos en Android es la misma que la del lenguaje CSS en el caso del diseño web: permiten separar el diseño del contenido.

Para crear un estilo simplemente utilizamos un elemento XML `style` con un atributo `name` y uno o más elementos `item`. El elemento `item` debe incluir un atributo `name` para especificar el atributo al que hace referencia (como color o fuente) y su contenido será el valor que se le asigna a dicho atributo.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
        <style name="NombreEstilo">
                <item name="NombreAtributo">value</item>
        </style>
</resources>
```

Utilizando un estilo como el anterior podríamos hacer que a siguiente definición de layout:

```
<TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:textColor="#00FF00"
    android:typeface="monospace"
    android:text="@string/hello" >
```

quede de la sigiente forma:

```
<TextView
        style="@style/EstiloFuente"
        android:text="@string/hello" />
```

siempre y cuando hubiéramos definido el estilo `EstiloFuente` en un archivo XML separado dentro de */res/values/* con el siguiente contenido:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
        <style name="EstiloFuente">
                <item name="android:layout_width">fill_parent</item>
        <item name="android:layout_height">wrap_content</item>
        <item name="android:textColor">#00FF00</item>
        <item name="android:typeface">monospace</item>
    </style>
</resources>
```

Un **tema** es un estilo que se aplica a una aplicación o actividad completa en lugar de a una vista individual. Cuando un estilo se aplica a una actividad, se aplicará a cada vista de la misma las propiedades que sean compatibles con ella. Por ejemplo, todas las propiedades relacionadas con el texto, como el color o el tamaño de la fuente, se aplicarían a las vistas que contuvieran texto. Más adelante veremos cómo aplicar un estilo como tema.

9.2.1. Herencia

La herencia es un mecanismo que nos va a permitir crear estilos fácilmente, extendiendo estilos base. Para ello hacemos uso del atributo `parent`. El siguiente código muestra un ejemplo de estilo base en el que se establecen ciertas propiedades para el texto, y cómo se aplica la herencia para construir otro estilo a partir de éste pero con el texto más pequeño:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
        <style name="TextoBase">
                <item name="android:textSize">14sp</item>
                <item name="android:textColor">#111</item>
        </style>
        <style name="TextoPequenyo" parent="TextoBase">
                <item name="android:textSize">8sp</item>
        </style>
</resources>
```

Se puede aplicar la herencia tantas veces como sea necesario. Por ejemplo, sería posible definir un nuevo estilo a partir del estilo `TextoPequenyo` definido en el ejemplo anterior, también mediante herencia.

9.2.2. Cómo aplicar un estilo

Existen dos formas de aplicar un estilo:

- A una vista individual, añadiendo un atributo `style` al elemento XML de la vista en

el archivo de layout correspondiente.
- A una aplicación o actividad de manera global, añadiendo el atributo `android:theme` o a un elemento `activity` o al elemento `application` dentro del *Manifest* de la aplicación.

Cuando se aplica un estilo a una vista determinada, ésta será la única a la que se aplicará. Por otra parte, si se aplica a un `ViewGroup`, hemos de tener en cuenta que los elementos `View` que contenga **no** heredarán las propiedades definidas en el estilo. Sin embargo, sería posible aplicar el estilo a todas las vistas, aunque para ello sería necesario aplicarlo como un tema.

9.2.3. Selección de un tema basada en la versión de la plataforma

Las versiones más modernas de Android incluyen nuevos temas que podríamos querer aplicar a nuestras aplicaciones, pero en ese caso tendríamos que intentar conservar la compatibilidad con versiones anteriores del sistema. Esto se puede conseguir mediante un tema que utilice herencia y carpetas de recursos con sufijos para escoger un archivo de estilo u otro según la versión.

Por ejemplo, nuestra aplicación podría declarar un estilo que herede del estilo ligero por defecto de Android. Para ello, crearíamos el archivo */res/values/styles.xml* y añadiríamos lo siguiente:

```
<style name="LightThemeSelector" parent="android:Theme.Light">
    ...
</style>
```

Por otra parte, podríamos desear que en el caso en el que el dispositivo estuviera ejecutando la versión 3.0 de Android, éste usara un tema especial de esta versión. Para ello podríamos crear el archivo */res/values-v11/styles.xml* con el siguiente contenido:

```
<style name="LightThemeSelector" parent="android:Theme.Holo.Light">
    ...
</style>
```

9.2.4. Utilizando los estilos y temas definidos en el sistema

La plataforma Android proporciona una gran cantidad de temas y estilos que podemos aplicar a nuestras propias aplicaciones. Se puede encontrar una referencia a todos estos estilos en la clase `R.style`. Para usar cualquier estilo definido dentro de dicha clase sustituimos cada subrayado por un punto. Por ejemplo, podríamos aplicar el estilo `Theme_NoTitleBar` de la siguiente forma: `"@android:style/Theme.NoTitleBar"`. Otros ejemplos podrían ser los siguientes, que producirían como resultado una actividad en forma de diálogo o en forma de ventana semitransparente respectivamente:

```
<activity android:theme="@android:style/Theme.Dialog">
<activity android:theme="@android:style/Theme.Translucent">
```

Desafortunadamente la clase `R.style` no está bien documentada y los estilos no están

descritos. Por lo tanto, la mejor manera de conocer cómo están definidos es mirando directamente el código fuente. El código fuente nos dirá qué propiedades se definen para cada estilo.

Para tener una referencia de qué propiedades o atributos podemos definir para cada estilo o tema (como `WindowBackground` o `TextAppearance`) podemos o bien consultar la clase `R.attr` o bien la descripción de la clase correspondiente a la vista que nos interese en la documentación de Android.

Nota:

Todo el código fuente se puede encontrar en *http://source.android.com/source/downloading.html*

10. Drawables, estilos y temas

10.1. Personalización del aspecto

Crea una nueva aplicación de Android llamada *Drawables* y haz que muestre un `TextView` que ocupe toda la anchura de la pantalla y que tenga el siguiente aspecto:

- Un marco de color azul con esquinas redondeadas.
- El marco debe tener un grosor de 5 density pixels.
- Se dejará un margen de 10 density pixels entre el marco y el texto que contiene
- Se dejará un margen de 5 density pixels entre los bordes de la pantalla y el marco. ¿Encontramos algún atributo que haga esto en la definición del drawable? ¿Y en el layout?
- Un relleno de gradiente lineal para el fondo, desde gris claro hasta blanco
- El texto será de color negro.

Ayúdate de la documentación oficial de Android para crear el recurso:

http://developer.android.com/guide/topics/resources/drawable-resource.html#Shape

10.2. Personalización de botones

Vamos a añadir un botón a la aplicación anterior debajo del `TextView`, que llenará todo el ancho. Se proporcionan en las plantillas de la sesión tres imágenes, una para cada uno de los estados del botón (*boton_focused.png*, *boton_normal.png* y *boton_pressed.png*). Se pide:

- Dar inicialmente al botón como aspecto la imagen correspondiente al estado sin pulsar.
- Crear un *state list drawable* para que el botón cambie su aspecto para cada estado.
- Convertir las imágenes a nine-patch y utilizar esta nueva versión en la aplicación.

Nos podemos ayudar de la documentación sobre *state list drawable*:

http://developer.android.com/guide/topics/resources/drawable-resource.html#StateList

10.3. Animación por fotogramas

Vamos a añadir a nuestra actividad, debajo del botón, una vista de tipo `ProgressBar` que muestre que se está realizando un progreso de forma indeterminada. Se pide:

- Ejecutar la aplicación y comprobar como aparece un círculo girando continuamente como indicativo de que se está realizando un progreso. Hacer que el indicador de progreso aparezca centrado en la horizontal (para ello podemos usar el atributo `layout_gravity`).

- Vamos ahora a personalizar la animación que figura en el progreso. En las plantillas de la sesión tenemos cuatro fotogramas de un armadillo caminando. Crea un *animation-list drawable* a partir de dichos fotogramas, que se reproduzca de forma continuada cambiando de fotograma cada 200ms.
- Reemplaza la animación por defecto del `ProgressBar` por la animación que hemos definido nosotros. Pista: se debe especificar como drawable, y el progreso continuo indeterminado (sin dar un porcentaje concreto) que queremos mostrar, se conoce en la API de Android como *indeterminate progress*. Busca entre los atributos de `ProgressBar` el que consideres adecuado.

Nos podemos ayudar de la documentación sobre animation list drawable:

http://developer.android.com/guide/topics/resources/animation-resource.html#Frame

10.4. Niveles

En este ejercicio vamos a añadir un `SeekBar` a nuestra aplicación *Drawables*, y haremos que se modifique su aspecto según su valor. Seguimos los siguientes pasos:

- Añadir el `SeekBar` debajo del `ProgressBar` del ejercicio anterior, haciendo que ocupe todo el ancho de la pantalla. Comprobar que se muestra correctamente y que al pulsar sobre él podemos mover el cursor. Por defecto, el nivel de la barra va de 0 a 10000.
- Personalizar el `SeekBar` para que el fondo de la barra cambie de color según el nivel seleccionado. Los colores que debe mostrar son:
  - Verde (#FF00FF00): Niveles de 0 a 2500
  - Amarillo (#FFFFFF00): Niveles de 2501 a 5000
  - Naranja (#FFFF8800): Niveles de 5001 a 7500
  - Rojo (#FFFF0000): Niveles de 7501 a 10000

Crea un `Dawable` de tipo rectángulo con esquinas redondeadas para cada estado. Introdúcelos en un *level list drawable*, y establécelo como `progressDrawable` en la barra.

Nos podemos ayudar de la documentación sobre level list drawable:

http://developer.android.com/guide/topics/resources/drawable-resource.html#LevelList

10.5. Estilos y temas (*)

En las plantillas de la sesión se te proporciona una aplicación llamada *Estilos* que contiene una única actividad. Su interfaz gráfica está compuesta por dos `TextView` y dos botones. Se han utilizado algunos atributos para modificar el aspecto de estos elementos. Mientras que algunos valores de estos atributos son compartidos por todas o algunas de las vista, otros son asignados exclusivamente a una.

En concreto:

- Todos los textos de la actividad aparecen de color rojo
- En el caso de los `TextView`, ambos tienen fondo de color blanco y una fuente de tipo *serif*
- Sin embargo, el tamaño del texto del segundo `TextView` es mayor que el del primero
- Con respecto a los botones, ambos comparten el mismo tamaño
- Sin embargo, el texto de un botón está alineado a la izquierda y el del otro a la derecha

Con tal de practicar lo aprendido en la sección de estilos y temas, deberemos definir lo siguiente:

- Un tema para los atributos compartidos por todos los elementos de la interfaz (texto de color rojo). ¿Se puede modificar el color del texto de los botones de esta forma? ¿Qué otro elemento de la interfaz gráfica a pasado a ser de color rojo?
- Un estilo para el primer `TextView` y un estilo heredado de éste para el segundo
- Un estilo para el primer botón y un estilo heredado de éste para el segundo
- Incorpora también los atributos `android:layout_height` y `android:layout_width` como parte de los estilos, de tal forma que en el archivo de layout los dos únicos atributos que tendrá cada vista serán `style` y `android:text`

11. Personalización de componentes

En esta sesión veremos cómo crear vistas personalizadas para nuestras aplicaciones Android. En primer lugar trataremos el tema de los componentes compuestos, que nos permitirán crear nuevas vistas a partir de la agrupación de vistas ya existentes. A continuación presentaremos los componentes propios, un mecanismo mediante el cual podremos crear nuestras propias vistas desde cero. Por último, mostraremos de manera resumida cómo extender la funcionalidad de las vistas ya existentes, como por ejemplo los `TextView`, para conseguir disponer de nuevos controles en nuestra aplicación sin tener que empezar desde cero.

11.1. Componentes compuestos

Un componente compuesto está formado por un conjunto de vistas hijas que se tratan de manera atómica, como si fueran una única vista. Cuando se crea un componente compuesto se deben definir los siguientes elementos: el layout o disposición, la apariencia y la interacción con cada una de las vistas que contiene.

11.1.1. Crear un componente compuesto

Para crear un componente compuesto definimos una subclase de algún `ViewGroup` (normalmente de un objeto de tipo *layout*). Escogeremos una clase de tipo *layout* que se adapte más a la forma en la que queremos disponer las vistas de nuestro nuevo componente. El esqueleto de la subclase creada será el siguiente:

```
public class MiComponente extends LinearLayout {
        public MiComponente(Context context) {
                super(context);
        }

        public MiComponente(Context context, AttributeSet atts) {
                super(context, atts);
        }
}
```

Como en el caso de las actividades, la mejor forma de definir la disposición de las vistas de un componente compuesto es mediante un archivo XML de *layout* en los recursos de la aplicación. En el siguiente listado se muestra un ejemplo de componente compuesto formado por un campo de edición de texto y un botón al que más tarde le proporcionaremos la funcionalidad de eliminar el texto contenido en el campo de edición:

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

```
                android:layout_height="wrap_content"
        />
        <Button
                android:id="@+id/button"
                android:layout_width="fill_parent"
                android:layout_height="wrap_content"
                android:text="Borrar"
        />
</LinearLayout>
```

Para usar este layout en nuestra nueva vista, sobrecargamos su constructor para que se inicialice su interfaz gráfica mediante el método `inflate` perteneciente al servicio de Android `LayoutInflate`, pasando como parámetro el identificador de recurso correspondiente al *layout*. En el siguiente código mostramos la clase `EdicionBorrable`, que se basa en el *layout* definido anteriormente.

```
public class EdicionBorrable extends LinearLayout {

        EditText editText;
        Button button;

        public EdicionBorrable(Context context) {
                super(context);

                // Creamos la interfaz a partir del layout
                String infService = Context.LAYOUT_INFLATER_SERVICE;
                LayoutInflater li;
                li =
(LayoutInflater)getContext().getSystemService(infService);
                li.inflate(R.layout.edicionborrable, this, true);

                // Obtenemos las referencias a las vistas hijas
                editText = (EditText)findViewById(R.id.editText);
                button = (Button)findViewById(R.id.button);
        }
}
```

11.1.2. Definir el componente compuesto mediante código

También es posible, como en el caso de las actividades, definir el *layout* de un componente compuesto mediante código. En el siguiente código mostramos cómo definir el *layout* de la clase `EdicionBorrable` sin utilizar un archivo XML:

```
public EdicionBorrable(Context context) {
        super(context);

        // Cambiamos la orientación del layout a vertical
        setOrientation(LinearLayout.VERTICAL);

        // Creamos las vistas hijas
        editText = new EditText(getContext());
        button = new Button(getContext());
        button.setText("Borrar");

        // Colocamos estas vistas en el control compuesto
        int lHeight = LayoutParams.WRAP_CONTENT;
        int lWidth = LayoutParams.FILL_PARENT;
        addView(editText, new LinearLayout.LayoutParams(lWidth, lHeight));
        addView(button, new LinearLayout.LayoutParams(lWidth, lHeight));
}
```

11.1.3. Añadir funcionalidad

Para añadir funcionalidad al componente compuesto debemos definir manejadores de eventos para sus vistas individuales. Esto se hará de la manera habitual. Por ejemplo, para hacer que en nuestro componente compuesto el botón borre el texto del cuadro de edición, lo único que tenemos que hacer es añadir el manejador adecuado al final del constructor, tras obtener la referencia a las vistas:

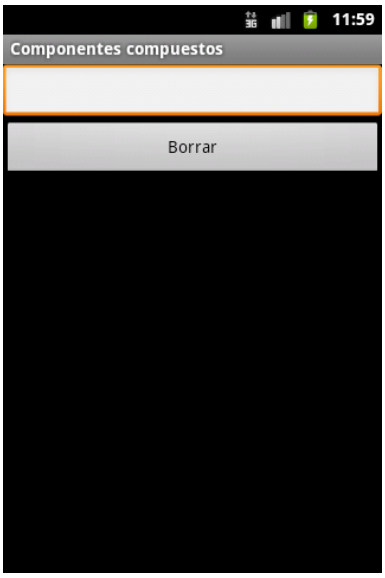
```
button.setOnClickListener(new Button.OnClickListener() {
        public void onClick(View v) {
                editText.setText("");
        }
});
```

11.1.4. Incluir el componente compuesto en nuestra actividad

Una vez definido el componente compuesto podemos añadirlo a la interfaz de nuestra actividad como si se tratara de una vista cualquiera. Por ejemplo, podemos añadir a nuestra aplicación una actividad cuya interfaz contendrá una única vista de tipo `EdicionBorrable`, con lo que su *layout* quedaría definido de la siguiente forma:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
<es.ua.jtech.android.compuestos.EdicionBorrable
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    />
</LinearLayout>
```

Obsérvese que a la hora de añadir nuestro componente `EdicionBorrable` se ha incluido el espacio de nombres, que en nuestro caso concreto es `es.ua.jtech.android.compuestos`. Por lo demás, nuestro componente se tratará como una vista normal y corriente.



Nuestro ejemplo de componente compuesto

11.2. Componentes propios

Si no hay ningún componente predefinido que se adapte a nuestras necesidades, podemos crear un nuevo tipo de vista (`View`) en la que especificaremos exactamente qué es lo que queremos dibujar en la pantalla. El primer paso consistirá en crear una subclase de `View` en la que sobrescribiremos el método `onDraw`, que es el que define la forma en la que se dibuja el componente.

```
public class MiVista extends View {
        public MiVista(Context context) {
                super(context);
        }

        @Override
        protected void onDraw(Canvas canvas) {
                // TODO Definir como dibujar el componente
        }
}
```

11.2.1. Lienzo y pincel

El método `onDraw` recibe como parámetro el lienzo (`Canvas`) en el que deberemos dibujar. En este lienzo podremos dibujar diferentes tipos de elementos, como primitivas geométricas, texto e imágenes.

Importante

No confundir el `Canvas` de Android con el `Canvas` que existe en Java ME/SE. En Java ME/SE el `Canvas` es un componente de la interfaz, que equivaldría a `View` en Android, mientras que el `Canvas` de Android es más parecido al objeto `Graphics` de Java ME/SE, que encapsula el contexto gráfico (o lienzo) del área en la que vamos a dibujar.

Además, para dibujar determinados tipos de elementos deberemos especificar también el tipo de pincel a utilizar (`Paint`), en el que especificaremos una serie de atributos como su color, grosor, etc.

Por ejemplo, para especificar un pincel que pinte en color rojo escribiremos lo siguiente:

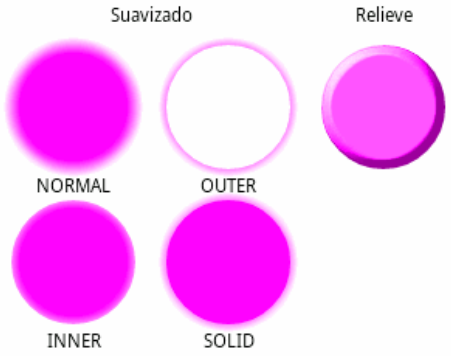
```
Paint p = new Paint();
p.setColor(Color.RED);
```

Las propiedades que podemos establecer en el pincel son:

- **Color plano**: Con `setARGB` o `setColor` se puede especificar el código ARGB del color o bien utilizar constantes con colores predefinidos de la clase `Color`.
- **Gradientes y shaders**: Se pueden rellenar las figuras utilizando shaders de gradiente o de bitmap. Para utilizar un *shader* tenemos el método `setShader`, y tenemos varios *shaders* disponibles, como distintos *shaders* de gradiente (`LinearShader`, `RadialShader`, `SweepShader`), `BitmapShader` para rellenar utilizando un mapa de bits como patrón, y `ComposeShader` para combinar dos *shaders* distintos.

Tipos de gradiente

- **Máscaras**: Nos sirven para aplicar un suavizado a los gráficos (`BlurMaskFilter`) o dar efecto de relieve (`EmbossMaskFilter`). Se aplican con `setMaskFilter`.

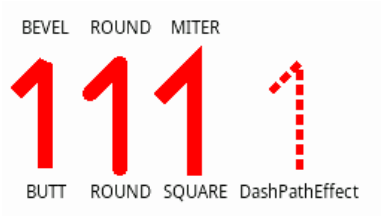


Mascaras de suavizado y relieve

- **Sombras**: Podemos crear efectos de sombra con `setShadowLayer`.
- **Filtros de color**: Aplica un filtro de color a los gráficos dibujados, alterando así su color original. Se aplica con `setColorFilter`.
- **Estilo de la figura**: Se puede especificar con `setStyle` que se dibuje sólo el trazo, sólo el relleno, o ambos.

Estilos de pincel

- **Estilo del trazo**: Podemos especificar el grosor del trazo (`setStrokeWidth`), el tipo de línea (`setPathEffect`), la forma de las uniones en las polilíneas (redondeada/`ROUND`, a inglete/`MITER`, o biselada/`BEVEL`, con `setStrokeJoin`), o la forma de las terminaciones (cuadrada/`SQUARE`, redonda/`ROUND` o recortada/`BUTT`, con `setStrokeCap`).



Tipos de trazo y límites

- **Antialiasing**: Podemos aplicar *antialiasing* con `setAntiAlias` a los gráficos para evitar el efecto *sierra*.
- **Dithering**: Si el dispositivo no puede mostrar los 16 millones de colores, en caso de haber un gradiente, para que el cambio de color no sea brusco, con esta opción (`setDither`) se mezclan pixels de diferentes colores para dar la sensación de que la transición entre colores es más suave.

Efecto dithering

- **Modo de transferencia**: Con `setXferMode` podemos cambiar el modo de transferencia con el que se dibuja. Por ejemplo, podemos hacer que sólo se dibuje encima de pixels que tengan un determinado color.
- **Estilo del texto**: Podemos también especificar el tipo de fuente a utilizar y sus atributos. Lo veremos con más detalle más adelante.

Una vez establecido el tipo de pincel, podremos utilizarlo para dibujar diferentes elementos en el lienzo, utilizando métodos de la clase `Canvas`.

En el lienzo podremos también establecer algunas propiedades, como el área de recorte (`clipRect`), que en este caso no tiene porque ser rectangular (`clipPath`), o transformaciones geométricas (`translate`, `scale`, `rotate`, `skew`, o `setMatrix`). Si queremos cambiar temporalmente estas propiedades, y luego volver a dejar el lienzo

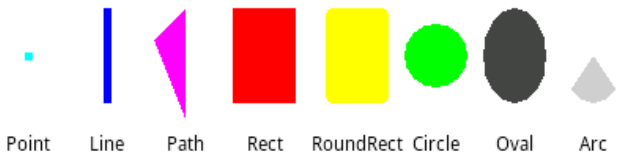
como estaba originalmente, podemos utilizar los métodos `save` y `restore`.

Vamos a ver a continuación como utilizar los métodos del lienzo para dibujar distintos tipos de primitivas geométricas.

11.2.2. Primitivas geométricas

En la clase `Canvas` encontramos métodos para dibujar diferentes tipos de primitivas geométricas. Estos tipos son:

- **Puntos**: Con `drawPoint` podemos dibujar un punto en las coordenadas X, Y especificadas.
- **Líneas**: Con `drawLine` dibujamos una línea recta desde un punto de origen hasta un punto destino.
- **Polilíneas**: Podemos dibujar una polilínea mediante `drawPath`. La polilínea se especificará mediante un objeto de clase `Path`, en el que iremos añadiendo los segmentos de los que se compone. Este objeto `Path` representa un contorno, que podemos crear no sólo a partir de segmentos rectos, sino también de curvas cuadráticas y cúbicas.
- **Rectángulos**: Con `drawRect` podemos dibujar un rectángulo con los límites superior, inferior, izquierdo y derecho especificados.
- **Rectángulos con bordes redondeados**: Es también posible dibujar un rectángulo con esquinas redondeadas con `drawRoundRect`. En este caso deberemos especificar también el radio de las esquinas.
- **Círculos**: Con `drawCircle` podemos dibujar un círculo dando su centro y su radio.
- **Óvalos**: Los óvalos son un caso más general que el del círculo, y los crearemos con `drawOval` proporcionando el rectángulo que lo engloba.
- **Arcos**: También podemos dibujar arcos, que consisten en un segmento del contorno de un óvalo. Se crean con `drawArc`, proporcionando, además de los mismos datos que en el caso del óvalo, los ángulos que limitan el arco.
- **Todo el lienzo**: Podemos también especificar que todo el lienzo se rellene de un color determinado con `drawColor` o `drawARGB`. Esto resulta útil para limpiar el fondo antes de empezar a dibujar.



Tipos de primitivas geométricas

A continuación mostramos un ejemplo de cómo podríamos dibujar una polilínea y un rectángulo:

```
Paint paint = new Paint();
paint.setStyle(Style.FILL);
paint.setStrokeWidth(5);
```

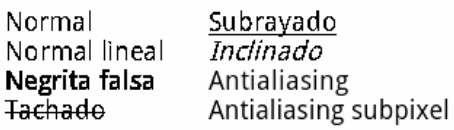
```
paint.setColor(Color.BLUE);

Path path = new Path();
path.moveTo(50, 130);
path.lineTo(50, 60);
path.lineTo(30, 80);

canvas.drawPath(path, paint);

canvas.drawRect(new RectF(180, 20, 220, 80), paint);
```

11.2.3. Cadenas de texto

Para dibujar texto podemos utilizar el método `drawText`. De forma alternativa, se puede utilizar `drawPosText` para mostrar texto especificando una por una la posición de cada carácter, y `drawTextOnPath` para dibujar el texto a lo largo de un contorno (`Path`).

Para especificar el tipo de fuente y sus atributos, utilizaremos las propiedades del objeto `Paint`. Las propiedades que podemos especificar del texto son:

- **Fuente**: Con `setTypeface` podemos especificar la fuente, que puede ser alguna de las fuentes predefinidas (*Sans Serif*, *Serif*, *Monoespaciada*), o bien una fuente propia a partir de un fichero de fuente. También podemos especificar si el estilo de la fuente será normal, cursiva, negrita, o negrita cursiva.
- **Tamaño**: Podemos establecer el tamaño del texto con `setTextSize`.
- **Anchura**: Con `setTextScaleX` podemos modificar la anchura del texto sin alterar la altura.
- **Inclinación**: Con `setTextSkewX` podemos aplicar un efecto de desencajado al texto, pudiendo establecer la inclinación que tendrán los carácteres.
- **Subrayado**: Con `setUnderlineText` podemos activar o desactivar el subrayado.
- **Tachado**: Con `setStrikeThruText` podemos activar o desactivar el efecto de tachado.
- **Negrita falsa**: Con `setFakeBoldText` podemos darle al texto un efecto de *negrita*, aunque la fuente no sea de este tipo.
- **Alineación**: Con `setTextAlign` podemos especificar si el texto de alinea al centro, a la derecha, o a la izquierda.
- **Subpixel**: Se renderiza a nivel de subpixel. El texto se genera a una resolución mayor que la de la pantalla donde lo vamos a mostrar, y para cada pixel real se habrán generado varios pixels. Si aplicamos *antialiasing*, a la hora de mostrar el pixel real, se determinará un nivel de gris dependiendo de cuantos pixels ficticios estén activados. Se consigue un aspecto de texto más suavizado.
- **Texto lineal**: Muestra el texto con sus dimensiones reales de forma lineal, sin ajustar los tamaños de los caracteres a la cuadrícula de pixels de la pantalla.
- **Contorno del texto**: Aunque esto no es una propiedad del texto, el objeto `Paint` también nos permite obtener el contorno (`Path`) de un texto dado, para así poder aplicar al texto los mismos efectos que a cualquier otro contorno que dibujemos.



Efectos del texto

Con esto hemos visto como dibujar texto en pantalla, pero para poderlo ubicar de forma correcta es importante saber el tamaño en pixels del texto a mostrar. Vamos a ver ahora cómo obtener estas métricas.

Las métricas se obtendrán a partir del objeto `Paint` en el que hemos definido las propiedades de la fuente a utilizar. Mediante `getFontMetrics` podemos obtener una serie de métricas de la fuente actual, que nos dan las distancias recomendadas que debemos dejar entre diferentes líneas de texto:

- `ascent`: Distancia que asciende la fuente desde la línea de base. Para texto con espaciado sencillo es la distancia que se recomienda dejar por encima del texto. Se trata de un valor negativo.
- `descent`: Distancia que baja la fuente desde la línea de base. Para texto con espaciado sencillo es la distancia que se recomienda dejar por debajo del texto. Se trata de un valor positivo.
- `leading`: Distancia que se recomienda dejar entre dos líneas consecutivas de texto.
- `bottom`: Es la máxima distancia que puede bajar un símbolo desde la línea de base. Es un valor positivo.
- `top`: Es la máxima distancia que puede subir un símbolo desde la línea de base. Es un valor negativo.

Los anteriores valores son métricas generales de la fuente, pero muchas veces necesitaremos saber la anchura de una determinada cadena de texto, que ya no sólo depende de la fuente sino también del texto. Tenemos una serie de métodos con los que obtener este tipo de información:

- `measureText`: Nos da la anchura en pixels de una cadena de texto con la fuente actual.
- `breakText`: Método útil para cortar el texto de forma que no se salga de los márgenes de la pantalla. Se le proporciona la anchura máxima que puede tener la línea, y el método nos dice cuántos carácteres de la cadena proporcionada caben en dicha línea.
- `getTextWidths`: Nos da la anchura individual de cada carácter del texto proporcionado.
- `getTextBounds`: Nos devuelve un rectángulo con las dimensiones del texto, tanto anchura como altura.

11.2.4. Imágenes

Podemos también dibujar en nuestro lienzo imágenes que hayamos cargado como `Bitmap`. Esto se hará utilizando el método `drawBitmap`.

También podremos realizar transformaciones geométricas en la imagen al mostrarla en el lienzo lienzo con `drawBitmap`, e incluso podemos dibujar el *bitmap* sobre una malla poligonal con `drawBitmapMesh`

11.2.5. Drawables

También podemos dibujar objetos de tipo *drawable* en nuestro lienzo, esta vez mediante el método `draw` definido en la clase `Drawable`. Esto nos permitirá mostrar en nuestro componente cualquiera de los tipos disponibles de *drawables*, tanto definidos en XML como de forma programática.

11.2.6. Medición del componente

Al crear un nuevo componente, además de sobreescribir el método `onDraw`, es buena idea sobreescribir también el método `onMeasure`. Este método será invocado por el sistema cuando vaya a ubicarlo en el *layout*, para asignarle un tamaño. Para cada dimensión (altura y anchura), nos pasa dos parámetros:

- **Tamaño**: Tamaño en píxeles solicitado para la dimensión (altura o anchura).
- **Modo**: Puede ser `EXACTLY`, `AT_MOST`, o `UNSPECIFIED`. En el primer caso indica que el componente debe tener exactamente el tamaño solicitado, el segundo indica que como mucho puede tener ese tamaño, y el tercero nos da libertad para decidir el tamaño.

Antes de finalizar `onMeasure`, deberemos llamar obligatoriamente a `setMeasuredDimension(width, height)` proporcionando el tamaño que queramos que tenga nuestro componente. Una posible implementación sería la siguiente:

```
@Override
protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
  int widthMode = MeasureSpec.getMode(widthMeasureSpec);
  int widthSize = MeasureSpec.getSize(widthMeasureSpec);
  int heightMode = MeasureSpec.getMode(heightMeasureSpec);
  int heightSize = MeasureSpec.getSize(heightMeasureSpec);

  int width = DEFAULT_SIZE;
  int height = DEFAULT_SIZE;

  switch(widthMode) {
    case MeasureSpec.EXACTLY:
      width = widthSize;
      break;
    case MeasureSpec.AT_MOST:
      if(width > widthSize) {
        width = widthSize;
      }
      break;
  }

  switch(heightMode) {
    case MeasureSpec.EXACTLY:
      height = heightSize;
      break;
    case MeasureSpec.AT_MOST:
      if(height > heightSize) {
```

```
        height = heightSize;
      }
      break;
  }

  this.setMeasuredDimension(width, height);
}
```

Podemos ver que tenemos unas dimensiones preferidas por defecto para nuestro componente. Si nos piden unas dimensiones exactas, ponemos esas dimensiones, pero si nos piden unas dimensiones como máximo, nos quedamos con el mínimo entre nuestra dimensión preferida y la que se ha especificado como límite máximo que puede tener.

11.2.7. Atributos propios

Si creamos un nuevo tipo de vista, es muy probable que necesitemos parametrizarla de alguna forma. Por ejemplo, si queremos dibujar una gráfica que nos muestre un porcentaje, necesitaremos proporcionar un valor numérico como porcentaje a mostrar. Si vamos a crear la vista siempre de forma programática esto no es ningún problema, ya que basta con incluir en nuestra clase un método que establezca dicha propiedad.

Sin embargo, si queremos que nuestro componente se pueda añadir desde el XML, será necesario poder pasarle dicho valor como atributo. Para ello en primer lugar debemos declarar los atributos propios en un fichero */res/values/attrs.xml*:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
        <declare-styleable name="Grafica">
                <attr name="percentage" format="integer"/>
        </declare-styleable>
</resources>
```

En el XML donde definimos el *layout*, podemos especificar nuestro componente utilizando como nombre de la etiqueta el nombre completo (incluyendo el paquete) de la clase donde hemos definido la vista:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
        xmlns:app=
"http://schemas.android.com/apk/res/es.ua.jtech.android.grafica"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
<es.ua.jtech.grafica.android.GraficaView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    app:percentage="60"
 />
</LinearLayout>
```

Podemos fijarnos en que para declarar el atributo propio hemos tenido que especificar el espacio de nombres en el que se encuentra, mediante el atributo `xmlns:app` del `LinearLayout`. En dicho espacio de nombres deberemos especificar el paquete que

hemos declarado en `AndroidManifest.xml` para la aplicación (en nuestro caso `es.ua.jtech.android.grafica`).

En el siguiente código vemos cómo acceder al valor de un atributo para nuestro componente propio desde el código:

```
public GraficaView(Context context) {
  super(context);
}

public GraficaView(Context context, AttributeSet attrs, int defStyle) {
  super(context, attrs, defStyle);
  this.init(attrs);
}

public GraficaView(Context context, AttributeSet attrs) {
  super(context, attrs);
  this.init(attrs);
}

private void init(AttributeSet attrs) {
  TypedArray ta = this.getContext().obtainStyledAttributes(attrs,
                                                R.styleable.Grafica);
  this.percentage = ta.getInt(R.styleable.Grafica_percentage, 0);
}
```

En primer lugar podemos ver que debemos definir todos los posibles constructores de las vistas, ya que cuando se cree desde el XML se invocará uno de los que reciben la lista de atributos especificados. Una vez recibamos dicha lista de atributos, deberemos obtener el conjunto de atributos propios mediante `obtainStyledAttributes`, y posteriormente obtener los valores de cada atributo concreto dentro de dicho conjunto.

11.2.8. Actualización del contenido

Es posible que en un momento dado cambien los datos a mostrar y necesitemos actualizar el contenido que nuestro componente está dibujando en pantalla. Podemos forzar que se vuelva a dibujar llamando al método `invalidate` de nuestra vista (`View`).

Esto podemos utilizarlo también para crear animaciones. De hecho, para crear una animación simplemente deberemos cambiar el contenido del lienzo conforme pasa el tiempo. Una forma de hacer esto es simplemente cambiar mediante un hilo o temporizadores propiedades de los objetos de la escena (como sus posiciones), y forzar a que se vuelva a redibujar el contenido del lienzo cada cierto tiempo.

Sin embargo, si necesitamos contar con una elevada tasa de refresco, como por ejemplo en el caso de un videojuego, será recomendable utilizar una vista de tipo `SurfaceView`.

11.3. Modificar vistas existentes

Existe una alternativa más simple para crear vistas propias: si ya existe una vista en Android que se parezca al nuevo control que deseamos crear, podemos simplemente extender la funcionalidad de dicho componente y modificar aquellas partes de su

comportamiento que deseamos modificar mediante sobrecarga. De esta forma podemos ahorrar también mucho código.

11.3.1. Extendiendo la funcionalidad de un TextView

El primer paso consistirá en crear una clase que herede de la clase de la vista cuya funcionalidad queremos extender. En el siguiente ejemplo mostramos el esqueleto de una clase llamada `MiTextView` y que heredará de `TextView`:

```
public class MiTextView extends TextView {

        public MiTextView (Context context, AttributeSet attrs, int
defStyle) {
                super(context, attrs, defStyle);
        }

        public MiTextView (Context context) {
                super(context);
        }

        public MiTextView (Context context, AttributeSet attrs) {
                super(context, attrs);
        }
}
```

El siguiente paso consistiría en sobrecargar aquellos manejadores de evento cuyo comportamiento queremos que difiera del de la clase base. El siguiente código amplia la clase `MiTextView` mostrada en el ejemplo anterior, añadiendo la sobrecarga de el manejador `onDraw`, lo cual permitirá modificar la forma en la que la vista se mostrará en la pantalla, y sobrecargando también el manejador `onKeyDown`:

```
public class MiTextView extends TextView {

        public MiTextView (Context context, AttributeSet ats, int
defStyle) {
                super(context, ats, defStyle);
        }

        public MiTextView (Context context) {
                super(context);
        }

        public MiTextView (Context context, AttributeSet attrs) {
                super(context, attrs);
        }

        @Override
        public void onDraw(Canvas canvas) {
                // Primero dibujamos en el canvas bajo el texto...

                // ... luego mostramos el texto de la manera habitual
                // haciendo uso de la clase base...
                super.onDraw(canvas);

                // ... y por último dibujamos cosas sobre el texto
        }

        @Override
        public boolean onKeyDown(int keyCode, KeyEvent keyEvent) {
```

```
                // Primero realizamos las acciones que sean oportunas
                // según la tecla pulsada...

                // ...y a continuación hacemos también uso de la clase
base
                return super.onKeyDown(keyCode, keyEvent);
        }
}
```

12. Personalización de componentes

12.1. Gráfica básica

En éste y los siguientes ejercicios vamos a crear un nuevo componente que muestre un gráfico de tipo tarta para indicar un determinado porcentaje. Dentro del círculo, aparecerá en rojo el sector correspondiente al porcentaje indicado, y en azul el resto. Para ello crearemos una nueva aplicación Android de nombre *Grafica*. En este primer ejercicio se pide crear una vista con la gráfica que muestre por defecto un porcentaje del 25%. Deberemos mostrarla por pantalla a partir de código, no de XML.

Nota:

Utiliza el método `getClipBounds` del `Canvas` para obtener las medidas del gráfico a dibujar. Este método devuelve un objeto `Rect` que puede ser pasado como parámetro al constructor de `RectF`.

12.2. Tamaño de la gráfica

Sobreescribe el método `onMeasure`, de forma que la gráfica del ejercicio anterior tenga un tamaño preferido de 50x50. Al ejecutarlo, ¿ha cambiado algo? ¿Por qué? (Pista: piensa en los parámetros de layout que se deben estar aplicando por defecto al introducir el componente mediante código).

12.3. Gráfica en XML y atributos propios

En primer lugar prueba a incluir la vista como XML en el layout por defecto definido para tu actividad principal.

Aviso:

En este punto es necesario haber definido todos los constructores de la superclase `View`, ya que al inicializar la gráfica desde XML podrá utilizar cualquiera de ellos según la información proporcionada. Échale un vistazo a la documentación de Android en http://developer.android.com/reference/android/view/View.html para saber cuáles son esos constructores.

A continuación añade un atributo llamado `porcentaje` a la vista. Haz los cambios necesarios para que el valor de este atributo modifique el aspecto de la gráfica. Prueba para diferentes valores de porcentaje.

Por último haz que la actividad muestre cuatro gráficas en una tabla de 2x2, con los siguientes porcentajes: 12.5, 37.5, 62.5 y 87.5

12.4. Modificar el aspecto de la gráfica

En este ejercicio modificaremos el aspecto de nuestro componente gráfica. En primer lugar vamos a hacer que la gráfica se muestre con un gradiente radial, con un color más oscuro en el centro y más claro en los bordes.

> **Nota:**
> El gradiente se establece con el método `setShader`, que toma como parámetro un objeto de tipo `Shader`. Busca entre sus subclases para encontrar el *shader* apropiado.

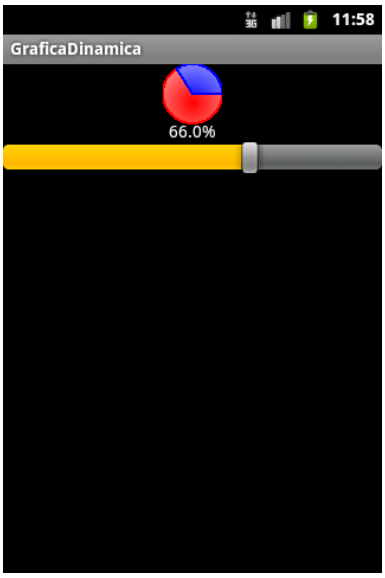
El siguiente paso será añadir un borde a la figura del mismo color que el central y con grosor 2. Para conseguir esto primero dibujaremos el relleno, y después el borde. Para ello podremos utilizar `setStyle`.

12.5. Texto (*)

Continuando con el código desarrollado en el ejercicio anterior, vamos ahora a mostrar también con texto el porcentaje que representa la gráfica. Este texto debe aparecer centrado horizontalmente y en la parte inferior, sin que se pueda llegar a cortar ninguna letra por el borde inferior de la vista. Añadir antialiasing al texto.

12.6. Gráfica dinámica

Creamos una nueva aplicación llamada *GraficaDinamica*. Dentro de esta aplicación definiremos un nuevo componente compuesto, que estará formado por el componente *Grafica* que hemos ido creando a lo largo de los ejercicios anteriores, y un `SeekBar`. La idea es que al modificar la posición del control del `SeekBar` (cuyos valores deben estar entre 0 y 100) se vaya modificando el porcentaje de la práctica. El control deberá tener el aspecto que se muestra a continuación:



Interfaz de la aplicación GraficaDinamica

Añade un control de este tipo al layout de la aplicación y pruébalo.

13. Roadmap

13.1. Libros

- Groovy in action, (Dierk König) - Ed: Manning
- Beginning Groovy and Grails, (Christopher M. Judd, Joseph Faisal Nusairat, James Shingler) - Ed: Apress
- The definitive guide to Grails (Graeme Keith Rocher) - Ed: Apress
- Groovy and Grails Recipes (Bashar Abdul-Jawad) - Ed: Apress
- Manual de desarrollo web con Grails (Nacho Brito) - Imaginarium Works

13.2. Enlaces

- Página oficial de Groovy
- API de Groovy
- Comunidad en castellano de Groovy
- Página oficial de Grails
- Podcast sobre Groovy & Grails
- Escuela de Groovy

13.3. Twitter

- Guillaume Laforge
- Graeme Rocher
- Neodevelop
- Domingo Suarez
- Andrés Almiray

13.4. Podcasts

- Grails.org.mx
- Grailspodcast

13.5. Proyectos Grails

- Uifi
- Jobsket
- goCMS
- Aloja