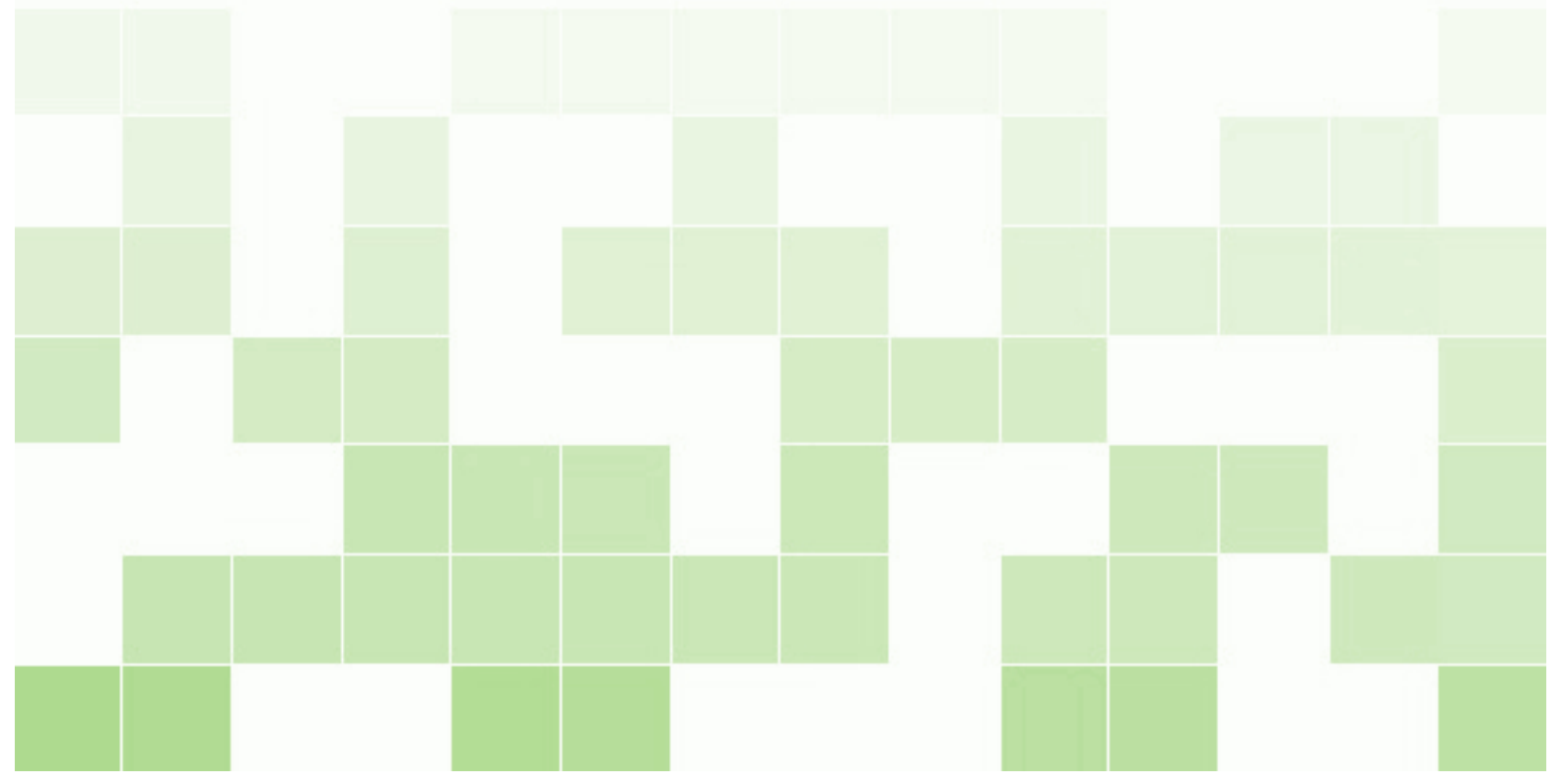




Programación Android

Alejandro Alcalde

elbaultdelprogramador.com



Copyright © 2013 Alejandro Alcalde

PUBLISHED BY L^AT_EX

ELBAULDELPROGRAMADOR.COM

Programación Android por Alejandro Alcalde se encuentra bajo una Licencia Creative Commons Reconocimiento-SinObraDerivada 3.0 Unported.

Créditos de las imágenes de capítulos a Benjamin Warth.

First printing, March 2013

ANDROID



Índice general

1	Hola Mundo	5
1.1	Introducción	5
1.2	Creando el proyecto	5
1.3	Componentes del proyecto	5
1.4	Profundizando en el “Hola Mundo”	7
2	Fundamentos	9
2.1	Conceptos básicos	9
2.2	Componentes de las aplicaciones	9
2.3	Intents	11
2.4	AndroidManifest	11
2.5	Actividades y tareas	11
2.6	Procesos e Hilos	12
2.7	Ciclo de vida de los componentes	12
2.8	Limpieza de Procesos	15
2.9	Ejemplo: Trabajar con Actividades y pasar parámetros entre ellas	15
3	Interfaz Gráfica	19
3.1	Conceptos básicos	19
3.2	Tipos de Layouts	20
3.3	Componentes gráficos y eventos	23
3.4	Adaptadores	26
3.5	Menús	33
3.6	Diálogos y Notificaciones	37

3.7	Estilos y Temas	41
4	Recursos	45
4.1	Usando recursos	46
4.2	Recursos string	47
4.3	Recursos Layout	48
4.4	Sintaxis de los recursos	50
4.5	Recursos compilados y no compilados	51
4.6	Arrays de strings	51
4.7	Plurales	52
4.8	Trabajar con recursos XML arbitrarios	54
4.9	Trabajar con recursos RAW	55
4.10	Trabajar con recursos Assets	56
4.11	Estructura del directorio de recursos	56
4.12	Recursos y cambios de configuración	57
5	StrictMode	61
5.1	Introducción	61
5.1.1	Ejemplo de uso	64
	Bibliography	65
	Books	65
	Articles	65
	Index	67



1 — Hola Mundo

1.1 Introducción

Como dije, voy a comenzar a escribir tutoriales sobre programación Android. Antes de comenzar es necesario tener configurado correctamente eclipse con el Android SDK, que se puede encontrar en este mismo blog, mediante el primer videotutorial de una entrada que publiqué hace tiempo, o simplemente buscando en google.

Antes de empezar, quiero comunicar que todas las entradas relacionadas con los tutoriales de Android los colocaré en la página Android[Pro1], así mismo, todo lo que he aprendido sobre Android ha sido debido al libro Pro Android 3 [lbr11], libro que considero todo programador Android debería poseer en su biblioteca.

En este capítulo vamos a empezar directamente con el típico Hola Mundo:

1.2 Creando el proyecto

Arrancamos eclipse con todo configurado correctamente y vamos a Archivo → nuevo → Proyecto Android:

Después de esto se nos mostrará un diálogo para configurar el proyecto, debemos introducir:

- El nombre del proyecto. en este caso Hola Mundo
- Donde queremos crear el proyecto (normalmente dentro del workspace)
- Versión Android a la que irá destinada la aplicación, en este caso Android 2.2
- Nombre de la aplicación (El que se mostrará al usuario una vez instalada, Hola Mundo)
- El Nombre del paquete que se usa como espacio de nombres y estructura de organización del código, “app.tutorial.holaMundo”
- Marcamos la opción *Crear Actividad* para que eclipse cree la clase que se lanzará al ejecutar la aplicación. Normalmente a esta clase se le llama *MainActivity*
- Versión Mínima del SDK es la versión mínima necesaria del SDK para ejecutar la aplicación, a menor número, la aplicación correrá en más terminales, pero no podremos usar las últimas características de Android
- Una vez rellenado todo, le damos a finalizar

1.3 Componentes del proyecto

Los proyectos de Android siguen una estructura fija de carpetas que debemos respetar. Podemos ver esta estructura con la vista *Package Explorer* que proporciona eclipse:

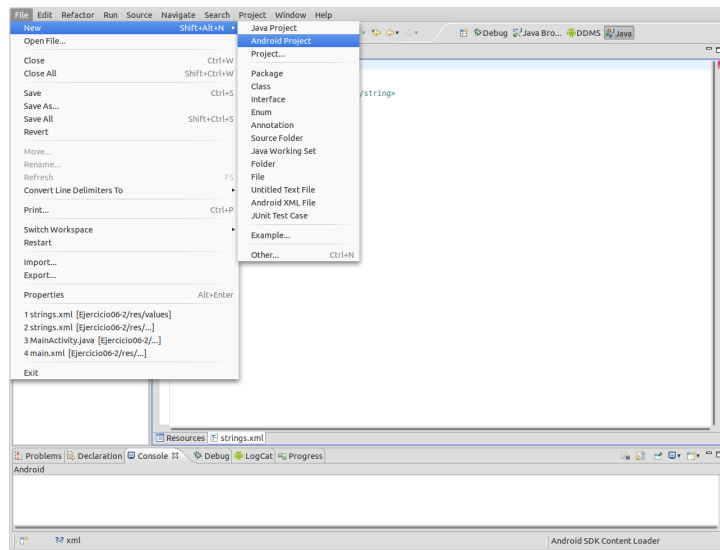


Figura 1.1: Crear un proyecto

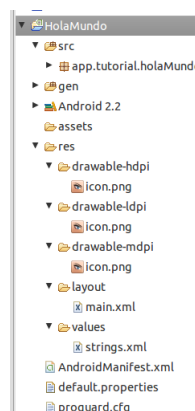


Figura 1.3: Seleccionando plataforma

Carpeta src (de fuentes)

Esta carpeta contiene el código fuente organizado en paquetes. Aquí irán las clases java de nuestra aplicación.

Carpeta gen (archivos generados)

Aquí van los archivos que genera el compilador en sus pasadas, como el archivo de recursos *R*, esta carpeta normalmente no se debe tocar.

Carpeta assets (De recursos varios)

Almacena recursos que pueda necesitar nuestra aplicación, como ficheros de música etc. Podremos acceder a ellos fácilmente con la clase del sistema *AssetManager*

Carpeta de recursos (res)

Esta carpeta es una de la que más vamos a usar junto con *src*, contiene todos los recursos necesarios para la aplicación. Todos los archivos de esta carpeta son indexados por el compilador y se genera el fichero de recursos *R*, que nos permite acceder a ellos de una forma rápida. Está dividida en subcarpetas:

- **anim:** Ficheros XML para la definición de Animaciones.
- **color:** Ficheros XML de definición de colores.
- **drawable:** Ficheros bitmap(.png, .9.png, .jpg, .gif) o XML con contenidos que se dibujarán (fondos, botones etc).
- **layout:** Ficheros XML que definen la capa de interfaz de usuario.

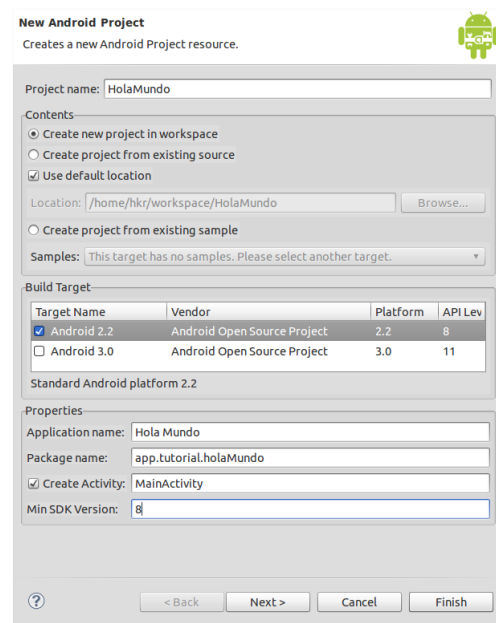


Figura 1.2: Seleccionando plataforma

- **menu:** Ficheros XML con la definición de los menús de la aplicación.
- **raw:** Binarios que no se pueden colocar en las otras carpetas.
- **values:** Ficheros XML para la definición de estilos, cadenas de texto para localización etc.
- **xml:** Ficheros XML que pueden ser accedidos en tiempo de ejecución.

Algunas carpetas pueden tener varias versiones para adaptarse a diferentes tamaños de pantallas, idiomas etc.

El archivo *Manifest (AndroidManifest.xml)*

Todos los proyectos tienen un archivo como este, en él se detallan las características principales (módulos, permisos, nombre, icono...).

Ahora que hemos explicado la estructura de un proyecto Android, veamos el ejemplo *Hola Mundo* al detalle.

1.4 Profundizando en el “Hola Mundo”

```
MainActivity.java
1 package app.tutorial.holaMundo;
2
3 import android.app.Activity;
4 import android.os.Bundle;
5
6 public class MainActivity extends Activity {
7     /** Called when the activity is first created. */
8     @Override
9     public void onCreate(Bundle savedInstanceState) {
10         super.onCreate(savedInstanceState);
11         setContentView(R.layout.main);
12     }
13 }
```

Al crear el proyecto dimos nombre a una Actividad (MainActivity), estas clases son las encargadas de mostrar las interfaz gráfica al usuario, deben extender de la clase *Activity*.

Al crear una actividad Android llama a su método *onCreate()* que hace lo necesario para mostrar

la pantalla al usuario. Tal y como está la actividad al crear el proyecto. Hace una llamada a `setContentView()`, que tiene como parámetro el identificador de una vista ya creada. Por lo tanto, *R.layout.main* referencia a un archivo xml situado en la carpeta *./res/layout* (ficheros de definición de pantalla).

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3      android:orientation="vertical"
4      android:layout_width="fill_parent"
5      android:layout_height="fill_parent"
6      >
7      <TextView
8          android:layout_width="fill_parent"
9          android:layout_height="wrap_content"
10         android:text="@string/hello"
11     />
12 </LinearLayout>

```

En este archivo se define una pantalla en la que los elementos se agruparán de forma lineal (LinearLayout) y con un componente de texto (TextView). Al componente de texto le fijamos el texto a mostrar con la referencia `@string/hello` (valor del item en *./res/values/strings.xml*).

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <resources>
3      <string name="nombre">Nombre:</string>
4      <string name="app_name">Ejercicio06-1</string>
5      <string name="apellidos">Apellidos:</string>
6      <string name="direccion">Dirección:</string>
7      <string name="localidad">Localidad:</string>
8      <string name="provincia">Provincia:</string>
9      <string name="pais">País:</string>
10 </resources>

```

Para que la aplicación funcione es necesario crear el AndroidManifest:

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3      package="app.tutorial.holaMundo" android:versionCode="1"
4      android:versionName="1.0">
5      <uses-sdk android:minSdkVersion="8" />
6
7      <application android:icon="@drawable/icon" android:label="@string/app_name">
8          <activity android:name=".MainActivity" android:label="@string/app_name">
9              <intent-filter>
10                 <action android:name="android.intent.action.MAIN" />
11                 <category android:name="android.intent.category.LAUNCHER" />
12             </intent-filter>
13          </activity>
14      </application>
15 </manifest>

```

En este archivo se definen el paquete por defecto, datos de versión, icono (mediante una referencia). El nombre de la aplicación (otra referencia al fichero strings.xml). Después se define el comportamiento de la aplicación. Se añaden dos filtros para que la actividad que definimos anteriormente sea usada como principal (*android.intent.action.MAIN*) y para que sea incluida en el menú de aplicaciones (*android.intent.category.LAUNCHER*)



2 — Fundamentos

2.1 Conceptos básicos

Hemos visto que un proyecto Android está formado por varias carpetas estructuradas, pero lo que se instala en los dispositivos es un fichero con extensión *.apk* (*application package*). Estos ficheros se generan con la herramienta apk (En el directorio tools del SDK) al terminar de compilar.

Las aplicaciones en Android tienen su propio entorno seguro de ejecución:

- Cada aplicación se ejecuta en su propio proceso Linux. El sistema lo crea cuando ejecutamos la aplicación y lo destruye cuando no se use pasado un rato o cuando el Sistema necesite recursos para otra aplicación
- Cada proceso se ejecuta en su propia máquina virtual, de esta manera está aislada del resto. De esta forma ante cualquier fallo en la aplicación solo afecta a su máquina virtual, no al resto.
- A cada aplicación se le asigna un identificador de usuario (uid) distinto. Los permisos de los archivos que refieren a la aplicación (caché, datos etc) son solo accesibles por dicho usuario. Es posible asignar un mismo uid a dos aplicaciones para que compartan una misma máquina virtual y recursos.

2.2 Componentes de las aplicaciones

La característica principal de Android es la reutilización de componentes de una aplicación por otra.

Por ejemplo, imaginemos que estamos desarrollando una aplicación que almacena datos de libros junto con una fotografía de su portada. En lugar de tener que escribir el código para capturar o seleccionar la imagen de la portada, podemos pasar el control a la aplicación de la cámara del teléfono, o a la galería, así una vez tomemos una foto o seleccionemos una imagen de la galería se nos devuelve el control a nuestra aplicación con la imagen seleccionada.

Para poder realizar estas operaciones, estamos obligados a dividir nuestras aplicaciones en módulos independientes que solo realicen una tarea concreta.

Veamos ahora otro ejemplo, muchos terminales tienen la opción de compartir algo en las redes sociales, por ejemplo Twitter, un módulo claramente definido de esta aplicación es por ejemplo la opción de *“enviar un mensaje o tweet”*, si seguimos la filosofía de dividir nuestras aplicaciones en módulos, la función de enviar un mensaje sería una actividad independiente que recibe como

parámetro el mensaje a enviar, si no recibe parámetro se mostrará el formulario para escribirlo. Dicha actividad usará la API de Twitter para enviar el mensaje y finalmente cerrará la actividad devolviendo el control a la aplicación que la llamó. De esta forma, y con los filtros adecuados en el *AndroidManifest.xml*, cada aplicación que quiera compartir algo en twitter llamará a esta actividad pasándole como parámetro el mensaje.

Con esto llegamos a la conclusión de que las aplicaciones Android no tienen un punto de entrada y otro de salida, podemos definir todos los que necesitamos. Para realizar todas estas operaciones, Android proporciona cuatro tipos de *componentes básicos*:

Actividades (Activity)

Son las encargadas de mostrar la interfaz de usuario e interactuar con él. Responden a los eventos generados por el usuario (pulsar botones etc). Heredan de la clase *Activity*.

El aspecto de la actividad se aplica pasando un objeto *View* (Encargado de dibujar una parte rectangular en la pantalla, pueden contener más objetos *View*, además todos los componentes de la interfaz (botones, imágenes etc) heredan de *View*) al método *Activity setContentView()*, que es el método encargado de dibujar la pantalla. Normalmente las vistas ocupan toda la pantalla, pero se pueden configurar para que se muestren como flotantes. Las actividades también pueden llamar a componentes que se mostrarán sobre su *View* (como diálogos o menús).

Por cada pantalla distinta hay una actividad distinta, normalmente las aplicaciones tienen una actividad fijada como punto de entrada. Por ejemplo:

Una aplicación que lee el correo tendrá las siguientes actividades:

- *RecibidosActivity*: muestra el listado de mensajes recibidos.
- *LeerMensajeActivity*: Muestra el contenido de un mensaje.
- *CrearMensajeActivity*: recibe como parámetro los datos necesarios, si no hay, muestra el formulario para rellenarlos y envía el mensaje.

Para esta aplicación definimos como punto de entrada *recibidosActivity* y *CrearMensajeActivity*, para que otras aplicaciones puedan reutilizarlas.

Servicios

No tienen interfaz visual y se ejecutan en segundo plano, se encargan de realizar tareas que deben continuar ejecutándose cuando nuestra aplicación no está en primer plano. Todos los servicios extienden de la clase *Service*.

Continuando con el ejemplo anterior, la aplicación de correo tendrá un servicio que comprobará y descargará nuevos correos. Es posible lanzar o conectar con un servicio en ejecución con la interfaz que proporciona la clase *Service*.

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

Receptores de mensajes de distribución (BroadcastReceiver)

Simplemente reciben un mensaje y reaccionan ante él, extienden de la clase *BroadcastReceiver*, no tienen interfaz de usuario, pero pueden lanzar Actividades como respuesta a un evento o usar *NotificationManager* para informar al usuario.

Android habitualmente lanza muchas notificaciones de sistema (llamadas entrantes, nuevos correos, nuevos sms etc). Si ponemos como ejemplo la aplicación del correo mencionada anteriormente, esta tendría un *BroadcastReceiver* escuchando el mensaje *nuevo_correo*, que lanzaría el servicio cada vez que detectara uno. Cuando esto sucediera, se mandaría un aviso a la barra del sistema para alertar al usuario.

Proveedores de contenido (ContentProvider)

Ponen un grupo de datos a disposición de distintas aplicaciones, extienden de la clase *ContentProvider* para implementar los métodos de la interfaz, pero para acceder a esta interfaz se ha de usar una clase llamada *ContentResolver*.

Con esta clase se permite acceder al sistema de ficheros, bases de datos SQLite o cualquier otra fuente de datos unificada.

Un lector de correo podría disponer de un *ContentProvider* para acceder a la bandeja de entrada y los datos del mensaje.

2.3 Intents

Las Actividades, Servicios y *BroadcastReceiver* se activan a través de mensajes asíncronos llamados *intent*. Un intent es un objeto que contiene todos los datos del mensaje.

Hay tres métodos para activar cada uno de los componentes:

- Las actividades se muestran pasando un Intent al método *Context.startActivity()* o *Activity.startActivityForResult()*. Una vez lanzada la actividad, dentro de la misma podemos abrir el objeto Intent para obtener los parámetros usando el método *getIntent()*
- Para lanzar servicios o interactuar con ellos pasaremos el intent al método *Context.startService()*. Para analizar el Intent dentro del proceso usaremos *onBind()*.
- Para pasar los intents a un mensaje de difusión debemos pasar el Intent al método *Context.sendBroadcast()*, *Context.sendOrderedBroadcast()* o *Context.sendStickyBroadcast()*, así el intent se entregará a todas las clases *BroadcastReceiver* que estén escuchando, y podrán analizarlo con *onReceive()*.

2.4 AndroidManifest

Para poder usar un componente, además de crearlo extendiendo de su clase correspondiente, es necesario definirlo en el *AndroidManifest.xml*. Este archivo podemos editarlo directamente como XML, o con el formulario que nos facilita eclipse.

Cada componente tiene su propia etiqueta xml:

- *<activity>*: Para actividades.
- *<service>*: Para servicios.
- *<receiver>*: Para receptores de mensajes de difusión.
- *<provider>*: Para proveedores de contenidos.
- *<intent-filter>*: Para categorizar componentes, así cuando se les llame no hay que saber el nombre del intent, android lo elige basandose en su categoría y parámetros.

2.5 Actividades y tareas

Las actividades conforme se van ejecutando van apilándose en una pila. Cuando finalizamos una actividad, con el método *finish()* o con la tecla atrás del teléfono, la actividad se extrae de la pila, quedando encima de la pila la actividad que se abrió anteriormente.

Si ejecutamos una actividad varias veces sin cerrarla, ésta aparecerá en la pila tantas veces como la hayamos ejecutado. La pila de actividades se envía al segundo plano cuando la aplicación pierde el foco, y vuelve al primer plano cuando la aplicación vuelve a tomar el control.

Podemos modificar este comportamiento con *flags* que pasamos al objeto Intent a partir de las propiedades de la actividad descritas en el *AndroidManifest*.

Si una pila de tareas se abandona por el usuario durante un periodo de tiempo y el sistema necesita más recursos, se limpia la pila de actividades (excepto la actividad principal), este comportamiento se puede modificar en el manifiesto:

Atributo	Función
<i>alwaysRetainTaskState</i>	Si vale true, se mantiene la pila aunque se abandone durante mucho tiempo
<i>clearTaskOnLaunch</i>	Si es true, se limpia la pila (excepto la actividad principal) cada vez que se lleve al segundo plano
<i>finishOnTaskLaunch</i>	Similar a la anterior, pero solo se aplica a la actividad con este atributo fijado a true.

Cuadro 2.1: Opciones Manifest.xml

2.6 Procesos e Hilos

Como cada aplicación se ejecuta en un proceso Linux distinto, todos los componentes y procesos de dicha aplicación corren en el mismo hilo. Esto se puede modificar con el atributo *process* de cada componente (activity, provider, receiver y service). En la etiqueta *application* del manifest podemos poner este atributo para que sea aplicado a todos sus elementos.

Para gestionar tareas pesadas podemos usar hilos para ejecutar dichas tareas en un hilo aparte (ejecutarlas en segundo plano). Para llevar a cabo esta operación usaremos el objeto *Thread* de java, aunque Android proporciona otros objetos para facilitar el trabajo, como *Handler*, *AsyncTask* o *Looper*. (entre otros).

2.7 Ciclo de vida de los componentes

Cada componente tiene un ciclo de vida distinto. Las superclases de estos componentes tienen métodos *callback* que les permiten reaccionar ante un cambio de estado. Cada método callback está obligado a llamar al mismo método de su padre.

Ciclo de vida de los Receiver

Tienen un ciclo de vida muy corto, ya que se activan al producirse un mensaje de difusión, que capturan con el método callback:

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

Si este método se está ejecutando, se considera el Receiver activo. Esto supone un problema si la tarea a ejecutar es pesada y la lanzamos a un hilo aparte. Cuando termine el método, Android considera que el receiver está inactivo, pero el hilo sigue ejecutándose y puede ser eliminado de la pila, interrumpiendo la tarea.

Se soluciona lanzando un servicio desde este método que se encarga de hacer las tareas pesadas, así, al tener el servicio su propio ciclo de vida, seguirá ejecutándose, aunque el receiver se considere inactivo.

Ciclo de vida de los ContentProvider

Su ciclo es muy corto, permanecen activos mientras sean referenciados por un ContentResolver.

Ciclo de vida de las Actividades

Tiene tres estados:

Activo: La actividad se encuentra en primer plano (Encima de la pila de tareas) e interactuando con el usuario.

Pausado: La actividad sigue siendo visible para el usuario, pero ha perdido el foco. Por ejemplo que se haya mostrado un cuadro de dialogo delante de nuestra actividad. Debemos guardar el estado de la interfaz y los datos de esta actividad antes de entrar en este estado, ya que podríamos perderlos si el sistema necesita más recursos de memoria.

Parado: La actividad no es visible para el usuario, queda a disposición del sistema para borrarla de la pila en caso de necesitar memoria.

La clase Activity dispone de métodos que se llaman cada vez que ésta cambia de estado, para permitirnos realizar tareas como guardar los datos antes de cambiar de estado, y cargar la actividad más rápido la próxima vez que se muestre:

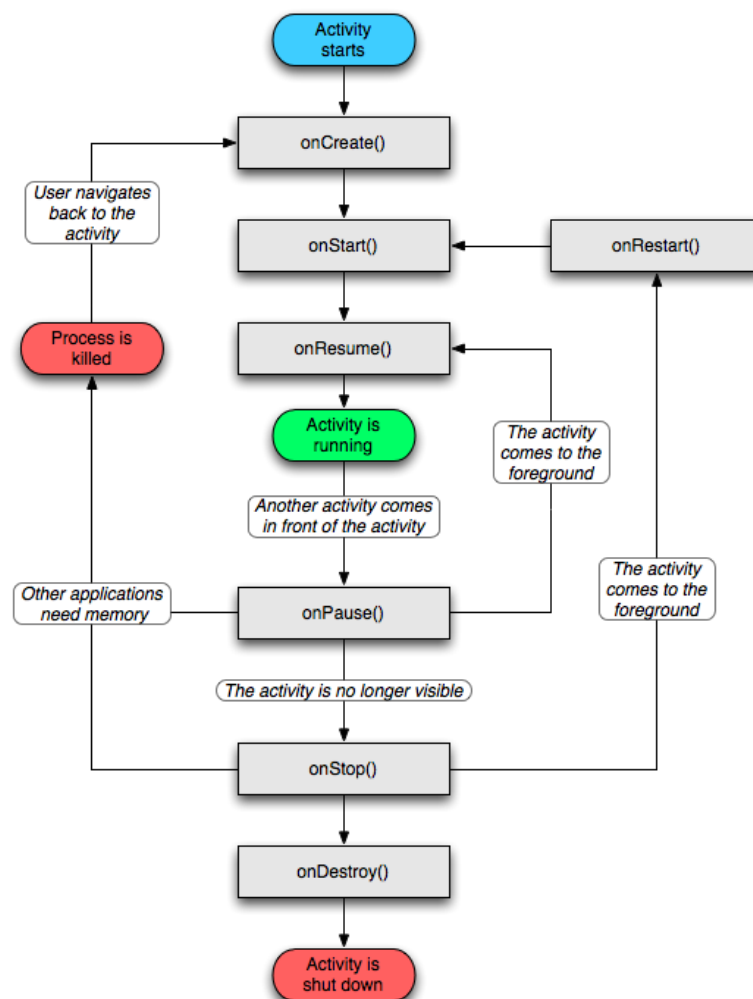


Figura 2.1: Ciclo de vida de una Activity

- ***onCreate(Bundle savedInstanceState):*** Este método se llama al crear la actividad. Siempre se sobrescribe para configurar la vista, crear adaptadores, rellenar los objetos con sus valores etc. Puede recibir como parámetro el estado anterior de la actividad para que podamos restaurarla.
- ***onPause():*** Es llamado justo antes de que se traiga a primer plano otra actividad. Aquí es donde debemos guardar los datos para no perder la información de la actividad si esta es sacada de la pila. Dentro de este método también se suele parar las tareas

pesadas que consuman CPU.

- **onStop():** Es llamado cuando la actividad se va a ocultar durante un largo periodo de tiempo. Si el sistema necesita recursos, puede que este método no sea llamado, por lo que es recomendable guardar los datos en el método *onPause()*.
- **onDestroy():** Último en llamarse antes de destruir la actividad. Puede llamarse a través del método *finish()* o llamarlo el sistema para conseguir más memoria. Para saber quién lo llamó, podemos usar *isFinishing()*.

Un ejemplo de uso de estos métodos puede ser cuando tenemos una aplicación que carga datos de internet, deberíamos cargar los datos al inicio de la actividad, una vez descargados, guardaríamos el estado de la actividad para que si es destruida, no sea necesario volver a descargar los datos.

Para realizar esta operación usaríamos el método *onSaveInstanceState()*, que crearía un *Bundle* con los datos necesarios que pasaríamos al método *onCreate()* la segunda vez que cargáramos la actividad, sin necesidad de volver a descargar los datos.

Ciclo de vida de Service

Los servicios se pueden usar de dos formas, dependiendo de como lo lancemos, su ciclo será uno u otro.

- Si lo lanzamos con *startService()* se ejecutará hasta que termine. Los servicios se configuran en el método *onCreate()* y se liberan en el *onDestroy()*. Podemos terminar un servicio externamente con *Context.stopService()* o dentro del mismo servicio con *Service.stopSelf()* o *Service.stopSelfResult()*.
 - Si lo lanzamos con *Context.bindService()* podremos interactuar con él mediante una interfaz que el servicio debe exportar. Terminaremos el servicio con *Context.unbindService()*.
- A continuación el diagrama con el ciclo de vida de los servicios:

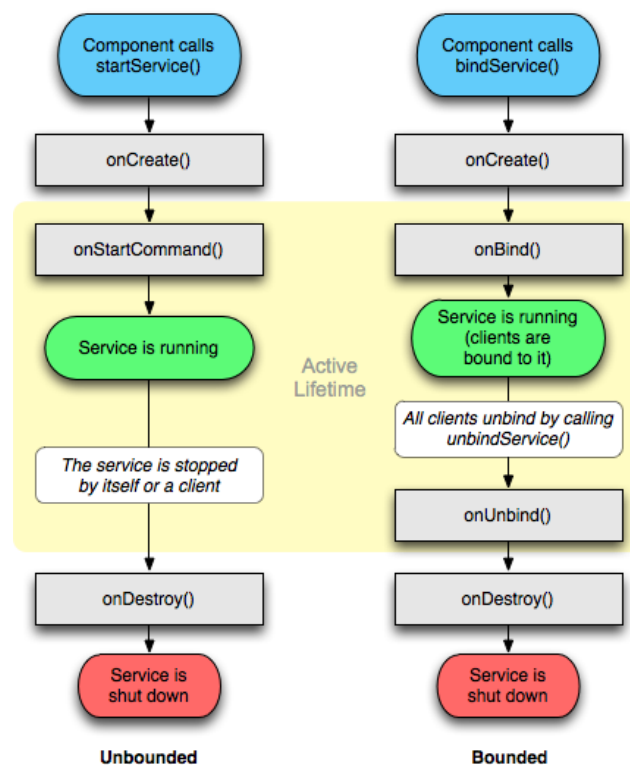


Figura 2.2: Ciclo de vida de los servicios

2.8 Limpieza de Procesos

Android va destruyendo componentes inactivos para liberar memoria, pero los elimina teniendo en cuenta cual es el de menor importancia:

- Los primeros en ser eliminados son los procesos vacíos (Son aplicaciones cerradas que se mantienen en memoria para cargar rápidamente la aplicación la proxima vez que se abra.)
- Procesos en segundo plano, estos son las aplicaciones que ya han ejecutado su método *onStop()*, Android confecciona una lista con los procesos en este estado y elimina en primer lugar el más antiguo.
- Despues elimina los procesos de servicio. (si sigue necesitando más memoria.)
- Si aún necesita más memoria, elimina los procesos pausados.
- Si con esto sigue necesitando, finalmente elimina el proceso en primer plano.

Es muy importante implementar bien los métodos de estado, para evitar perden información.

2.9 Ejemplo: Trabajar con Actividades y pasar parámetros entre ellas

En el primer capítulo, vimos como crear nuestro primer proyecto en Android, el conocido Hola Mundo, en esta entrada, vamos a ver como crear varias actividades y cómo hacer que se pasen parámetros las unas a las otras. El proyecto con este ejemplo está disponible para su descarga (Comentado paso a paso): Intents y Bundles. Voy a explicar un poco por encima que hace cada fichero del proyecto:

```

./res/layout/main.xml
1 <?xml version="1.0" encoding="utf-8"?>
2 <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3     android:orientation="vertical"
4     android:layout_width="fill_parent"
5     android:layout_height="fill_parent">
6
7     <TextView android:id="@+id/textView1"
8     android:layout_width="fill_parent"
9     android:layout_height="wrap_content"
10    android:text="@string/hello" />
11
12    <Button android:id="@+id/button1"
13    android:layout_width="fill_parent"
14    android:layout_height="wrap_content"
15    android:text="@string/cadena1" />
16
17    <Button android:id="@+id/button2"
18    android:layout_width="fill_parent"
19    android:layout_height="wrap_content"
20    android:text="@string/cadena2" />
21 </LinearLayout>

```

En este layout principal vamos a añadir dos botones que nos servirán para lanzar las nuevas actividades que creemos después.

```

./res/layout/segundaActividad.xml
1 <?xml version="1.0" encoding="utf-8"?>
2 <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3     android:orientation="vertical"
4     android:layout_width="fill_parent"
5     android:layout_height="fill_parent">
6
7     <TextView android:id="@+id/textView1"
8     android:layout_width="fill_parent"
9     android:layout_height="wrap_content"
10    android:text="@string/cadena1" />
11
12    <TextView android:id="@+id/params"
13    android:layout_width="fill_parent"

```

```

14         android:layout_height="wrap_content"
15         android:text="@string/hello" />
16
17         <Button android:id="@+id/boton"
18             android:layout_width="fill_parent"
19             android:layout_height="wrap_content"
20             android:text="@string/cadena1" />
21
22     </LinearLayout>

```

Este layout vamos a usarlo para mostrar los parámetros que pasemos de una actividad a otra.

```

./src/MainActivity.java
1 package com.elbauldelprogramador.actividades;
2
3 import android.app.Activity;
4 import android.content.Intent;
5 import android.os.Bundle;
6 import android.view.View;
7 import android.view.View.OnClickListener;
8 import android.widget.Button;
9 import android.widget.Toast;
10
11 public class MainActivity extends Activity {
12
13     protected static final int REQUEST_CODE = 10;
14
15     @Override
16     public void onCreate(Bundle savedInstanceState) {
17         super.onCreate(savedInstanceState);
18         setContentView(R.layout.main);
19
20         // Capturamos los objetos gráficos que vamos a usar
21         Button button1 = (Button) findViewById(R.id.button1);
22         Button button2 = (Button) findViewById(R.id.button2);
23
24         // Fijamos un evento onclick para el button1, cada vez que
25         // lo pulsemos se llamará a este método (que abrirá una actividad)
26         button1.setOnClickListener(new OnClickListener() {
27             public void onClick(View v) {
28                 Intent intent =
29                     new Intent(MainActivity.this, Activity1.class);
30                 startActivity(intent);
31             }
32         });
33
34         //button2 pasará parámetros a otra actividad, y los devolverá
35         button2.setOnClickListener(new OnClickListener() {
36             public void onClick(View v) {
37                 Intent intent = new Intent(MainActivity.this, ParametrosActivity.class);
38
39                 // damos valor al parámetro a pasar
40                 intent.putExtra("param1", "valor del parámetro 1 (viene de MainActivity)");
41                 /*
42                  * Inicia una actividad que devolverá un resultado cuando
43                  * haya terminado. Cuando la actividad termina, se llama al método
44                  * onActivityResult() con el requestCode dado.
45                  * El uso de un requestCode negativo es lo mismo que llamar a
46                  * startActivity(intent) (la actividad no se iniciará como una
47                  * sub-actividad).
48                  */
49                 startActivityForResult(intent, REQUEST_CODE);
50             }
51         });
52     }
53
54     /*
55     * Éste método se llama cuando la actividad que iniciamos con un
56     * startActivityForResult finaliza, dandi el REQUEST_CODE con el
57     * que llamó, el resultCode se devuelve, junto con algunos datos

```

```

58     * adicionales, el resultCode será RESULT_CANCELED si la actividad
59     * devuelve eso explícitamente, si no devuelve ningún resultado o
60     * si la operación finalizó de forma inesperada.
61     */
62     @Override
63     protected void onActivityResult(int requestCode, int resultCode, Intent data) {
64         super.onActivityResult(requestCode, resultCode, data);
65         if (requestCode == REQUEST_CODE) {
66             // cogemos el valor devuelto por la otra actividad
67             String result = data.getStringExtra("result");
68             // enseñamos al usuario el resultado
69             Toast.makeText(this, "ParametrosActivity devolvió: "
70                 + result, Toast.LENGTH_LONG).show();
71         }
72     }
73 }

```

En esta clase vamos a modificar el comportamiento de los botones, añadiéndoles listeners para cuando el usuario haga click en ellos.

Por último, vamos a crear otras dos actividades, la primera (./src/activity1.java), no va a hacer nada, solo mostrarse. La segunda (./src/ParametrosActivity.java), va a recibir un parámetro y devolver otro.

```

----- ./src/activity1.java -----
1 package com.elbaultdelprogramador.actividades;
2
3 import android.app.Activity;
4 import android.os.Bundle;
5 import android.view.View;
6 import android.view.View.OnClickListener;
7 import android.widget.Button;
8 import android.widget.TextView;
9
10 public class Activity1 extends Activity {
11     /** Called when the activity is first created. */
12     @Override
13     public void onCreate(Bundle savedInstanceState) {
14         super.onCreate(savedInstanceState);
15         setContentView(R.layout.segunda_actividad);
16
17         // Capturamos los objetos gráficos que vamos a usar
18         TextView text = (TextView) findViewById(R.id.textView1);
19         Button button = (Button) findViewById(R.id.boton);
20
21         // Agregamos al textView un texto
22         text.setText(R.string.cadenal);
23
24         // Cambiamos el texto al botón
25         button.setText(R.string.salir);
26
27         // Evento onclick del botón, cuando se pulse,
28         // cerramos la actividad
29         button.setOnClickListener(new OnClickListener() {
30             public void onClick(View v) {
31                 finish();
32             }
33         });
34     }
35 }

```

```

----- ./src/ParametrosActivity.java -----
1 package com.elbaultdelprogramador.actividades;
2
3 import android.app.Activity;
4 import android.content.Intent;
5 import android.os.Bundle;
6 import android.view.View;
7 import android.view.View.OnClickListener;

```

```
8 import android.widget.Button;
9 import android.widget.TextView;
10
11 public class ParametrosActivity extends Activity {
12     private static final int OK_RESULT_CODE = 1;
13     @Override
14     protected void onCreate(Bundle savedInstanceState) {
15         // TODO Auto-generated method stub
16         super.onCreate(savedInstanceState);
17         setContentView(R.layout.segunda_actividad);
18
19         // Capturamos los objetos gráficos que vamos a usar
20         TextView text = (TextView) findViewById(R.id.textView1);
21         Button button = (Button) findViewById(R.id.boton);
22         TextView params = (TextView) findViewById(R.id.params);
23
24         text.setText(R.string.cadena2);
25
26         button.setText(R.string.salir);
27
28         //Al pulsar el botón cerramos la ventana y volveremos a la anterior
29         button.setOnClickListener(new OnClickListener() {
30             public void onClick(View v) {
31                 //Cierra la actividad y la saca de la pila
32                 returnParams();
33             }
34         });
35
36         // Mostramos los parámetros recibidos de la actividad mainActivity
37         Bundle receiveParams = getIntent().getExtras();
38         params.setText(receiveParams.getString("param1"));
39     }
40
41     protected void returnParams() {
42         Intent intent = new Intent();
43         intent.putExtra("result", "Valor devuelto por ParametrosActivity");
44         setResult(OK_RESULT_CODE, intent);
45         finish();
46     }
47 }
```



3 — Interfaz Gráfica

3.1 Conceptos básicos

Todos los componentes de la interfaz de usuario de Android descienden de la clase *View*. Dichos objetos están organizados en forma de árbol y pueden contener nuevos objetos *View*, permitiendo crear interfaces muy completas.

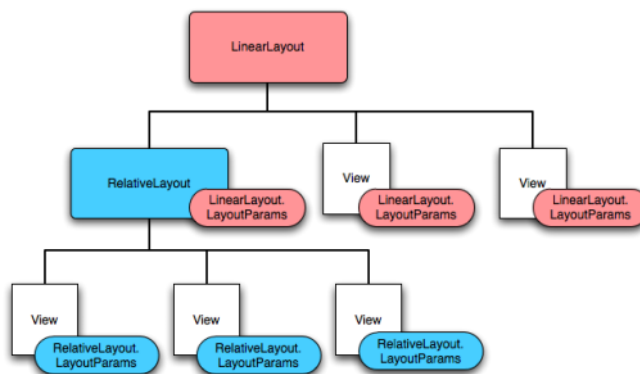


Figura 3.1: Layouts

Los objetos *View* se pueden definir de dos maneras:

- Mediante un fichero XML colocado dentro del directorio *res/layout*, que es el que usaremos normalmente.
- En tiempo de ejecución, muy útil para crear nuestros propios componentes *View*.

Para dibujar la interfaz, el sistema necesita que le pasemos el objeto *View* raíz, para ir descendiendo por cada uno de sus nodos y presentar al usuario toda la interfaz. El método encargado de esto es *Activity setContentView()*.

Android se encarga de dibujar los elementos llamando primero al método *draw()* de cada vista, podríamos decir que cada vista se dibuja a sí misma. El proceso de dibujo se hace en dos veces. Inicialmente se llama al método *measure(int, int)*, que define el tamaño de cada objeto *View*, posteriormente se llama al método *layout(int, int, int, int)*, que posiciona el objeto dentro de la vista.

Para que Android sepa dibujar correctamente los objetos, tenemos que pasarle algunos datos, como son la altura y anchura. Para eso nos servimos de la clase *LayoutParams*, que puede tomar los siguientes valores:

- Un número.
- La constante *MATCH_PARENT*, que indica que la vista debe intentar ser tan grande como su padre, quitando el padding.
- La constante *WRAP_CONTENT*, para que intente ser lo suficientemente grande para mostrar su contenido, mas el padding.

También nos podemos servir de la clase *View.MeasureSpec*, para especificar el tamaño y cómo deben ser posicionadas.

- *AT_MOST*, el padre fija un tamaño mínimo para el hijo. El hijo(y los descendientes de éste) tienen que ocupar por lo menos ese tamaño.
- *EXACTLY*, el padre impone un tamaño exacto al hijo.
- *UNSPECIFIED*, el padre fija el tamaño deseado del hijo.

Un atributo imprescindible es el *id*(de tipo entero). Que sirve para identificar únicamente a un objeto View. Cuando lo declaramos mediante xml podemos referenciarlo a través de la clase de recursos R, usando una @.

android:id="@+id/nombreID": Crea un nuevo atributo en la clase R llamado nombreID.

android:id="@id/nombreID": Hace referencia a un id ya existente asociado a la etiqueta 'nombreID'.

android:id="@android:id/list": Referencia a una etiqueta definida en la clase R del sistema llamada 'list'.

Los objetos View pueden tener otros muchos atributos, como padding, colores, imágenes, fondos, márgenes etc.

Context

Si ya has programado algo en Android, o has visto alguno de los ejemplos, probablemente hayas visto que muchos métodos referidos a la vista piden como parámetro un objeto de tipo context. Es una interfaz para la información global de la aplicación. A través de él podemos acceder a recursos, clases y operaciones, como lanzar actividades, manejar intents etc.

Podemos acceder al contexto de diferentes formas en función de donde nos encontremos:

- Con el método *getContext()*.
- Las actividades implementan esta interfaz, por lo que haciendo referencia a ellas mismas, con (*this*) o *NombreActivity.this*, estaremos referenciando el contexto.
- Usando otros métodos como *getApplicationContext()* o *getApplication()*.

3.2 Tipos de Layouts

Los layouts nos permiten posicionar cada objeto gráfico en el lugar que queramos de la pantalla, es decir, nos permite diseñar el aspecto gráfico que va a tener nuestra pantalla. Los layouts son de tipo ViewGroup, una subclase de View.

Existen varios tipos de Layouts para Android, vamos a ver los más comunes:

FrameLayout

Este tipo de Layout es el más básico, coloca a sus objetos hijos en la parte superior izquierda de la pantalla.

FrameLayout.xml


```

1 <?xml version="1.0" encoding="utf-8"?>
2 <FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
3     android:layout_width="fill_parent"
4     android:layout_height="fill_parent">
5
6     <TextView
7         android:layout_width="fill_parent"
8         android:layout_height="wrap_content"
9         android:text="@string/hello"/>
10
11     <TextView
12         android:layout_width="fill_parent"
13         android:layout_height="wrap_content"
14         android:text="@string/app_name"/>
15
16 </FrameLayout>

```

Como se puede apreciar en el resultado, si hay más de un hijo, los objetos se amontonan unos encima de otros.



Figura 3.2: FrameLayout

LinearLayout

Este tipo de layout coloca sus hijos unos detrás de otros, también comenzando por la esquina superior izquierda de la pantalla. Podemos colocarlos alineados horizontalmente o verticalmente mediante su propiedad `android:orientation="horizontal | vertical"`.

```

----- LinearLayout.xml -----
1 <?xml version="1.0" encoding="utf-8"?>
2 <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3     android:orientation="horizontal"
4     android:layout_width="fill_parent"
5     android:layout_height="fill_parent">
6
7     <TextView
8         android:layout_width="wrap_content"
9         android:layout_height="wrap_content"
10        android:text="@string/app_name"
11        android:background="#0ff"/>
12
13    <TextView
14        android:layout_width="wrap_content"
15        android:layout_height="wrap_content"
16        android:text="@string/hello"

```

```

17         android:background="#ff0"/>
18
19 </LinearLayout>

```

En este caso, he puesto un fondo de color a cada texto (con la propiedad `android:background`) para diferenciarlo bien, y he usado la orientación horizontal, de haber usado la orientación vertical, los textos aparecerían uno debajo del otro:

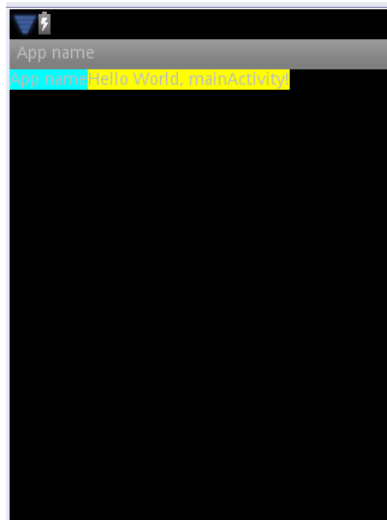


Figura 3.3: LinearLayout

RelativeLayout

Este Layout permite que coloquemos los elementos en un lugar con respecto a la posición de otro, es decir, colocar un botón a la derecha de un texto, o centrarlo en la pantalla, o por ejemplo, colocar un texto encima de tal elemento y a la derecha de este otro.

Para conseguir esto, *RelativeLayout* proporciona propiedades como *android:layout_toRightOf* o *android:layout_alignLeft*, que toman como valores los identificadores de los objetos, o valores booleanos.

```

----- RelativeLayout.xml -----
1 <?xml version="1.0" encoding="utf-8"?>
2 <RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
3     android:orientation="horizontal"
4     android:layout_width="fill_parent"
5     android:layout_height="fill_parent">
6
7     <TextView
8         android:layout_width="wrap_content"
9         android:layout_height="wrap_content"
10        android:text="@string/app_name"
11        android:background="#0ff"
12        android:layout_centerInParent="true"
13        android:id="@+id/text1"/>
14
15    <TextView
16        android:id="@+id/text2"
17        android:layout_width="wrap_content"
18        android:layout_height="wrap_content"
19        android:text="@string/hello"
20        android:background="#ff0"
21        android:layout_below="@id/text1"/>
22
23 </RelativeLayout>

```

Como vemos, hemos centrado el texto1 en la pantalla con `android:layout_centerInParent="true"` y hemos puesto debajo del texto1 al texto2 con `android:layout_below="@id/text1"`.



Figura 3.4: RelativeLayout

Para saber más acerca de todos los tipos de layouts que hay podéis visitar <http://developer.android.com/guide/topics/ui/layout-objects.html>

3.3 Componentes gráficos y eventos

Ya hemos visto que todos los componentes visuales descienden del objeto View, que proporciona una interfaz para que podemos interactuar con ellos.

Para que nuestras aplicaciones sean funcionales, necesitamos responder a los eventos que el usuario dispare al interactuar con nuestro programa, en Android, esto se consigue mediante los *Listeners*, que serán llamados cada vez que se produzca un evento. Por ejemplo, un listener muy común será `setOnClickListener()`, que responderá cada vez que se pulse sobre la vista a la que se lo aplicamos, como un botón, o una imagen. Hay muchos tipos de listener, `setKeyListener()` (Para eventos de teclado), `setOnItemClickListener()` (Para eventos al seleccionar un elemento de una lista) etc etc.

En los ejemplos mostrados hasta ahora, solo hemos visto objetos de tipo *TextView*, vamos a ver unos cuantos más (En cada ejemplo pondré la definición XML del objeto, y su manipulación mediante código):

Button

Botones simples, para realizar acciones al pulsar sobre ellos.

```

button.xml
1 <Button
2   android:layout_width="wrap_content"
3   android:layout_height="wrap_content"
4   android:text="Pulsame"
5   android:layout_centerInParent="true"
6   android:id="@+id/button1"/>

Button.java
1 //Recoger el botón en una variable para usarlo
2 final Button button1 = (Button) findViewById(R.id.button1);
  
```

```

3
4 button1.setOnClickListener(new OnClickListener() {
5
6     @Override
7     public void onClick(View arg0) {
8         Toast.makeText(
9             button1.getContext()
10            , "Me has pulsado " + ++contador + " veces."
11            , Toast.LENGTH_SHORT)
12            .show();
13     }
14 });

```

En este caso, hemos declarado una variable como miembro de la clase, (*public int contador = 0;*), para que cada vez que pulsemos el botón nos salga un mensaje con el número de veces que lo hemos pulsado:

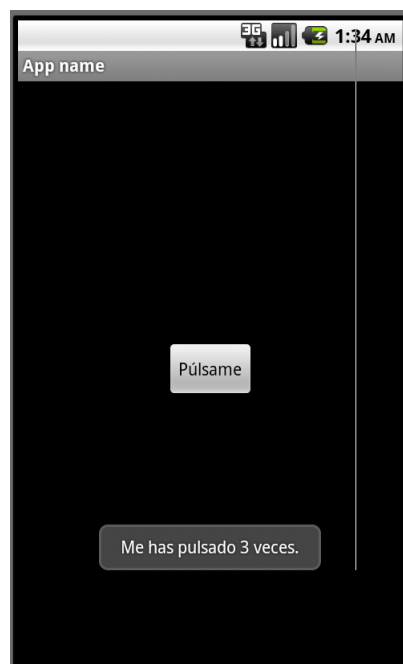


Figura 3.5: Ejemplo botones

EditText

Son campos de texto en los que el usuario puede escribir.

```

----- EditText.xml -----
1 <EditText
2     android:layout_width="200dip"
3     android:layout_height="wrap_content"
4     android:layout_above="@id/button1"
5     android:id="@+id/editText1"
6     android:layout_centerInParent="true"/>
-----
----- EditText.java -----
1 final EditText editText1 = (EditText) findViewById(R.id.editText1);
2
3 editText1.setOnKeyListener(new OnKeyListener() {
4
5     @Override
6     public boolean onKey(View arg0, int arg1, KeyEvent arg2) {
7         if (arg1 == KeyEvent.KEYCODE_ENTER) {

```

```

8      Toast.makeText (
9          editText1.getContext ()
10         , "Escribiste: " + editText1.getText ()
11         , Toast.LENGTH_SHORT)
12         .show ();
13         return true;
14     }
15     return false;
16 }
17 });

```

Lo que hemos hecho con este EditText, es fijarle un *onKeyListener*, que comprobará (con el if), que hemos pulsado la tecla enter, y si es cierto, mostrar el texto escrito:

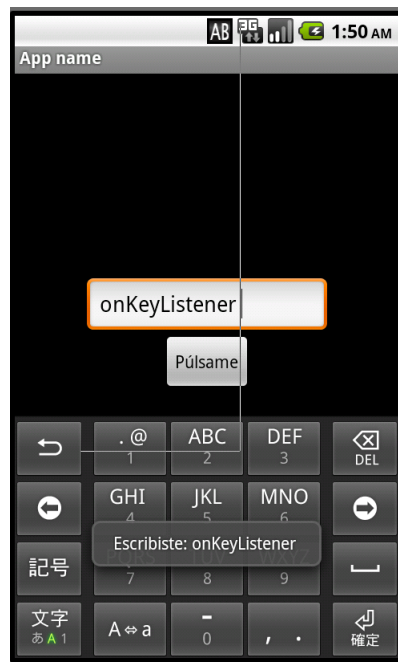


Figura 3.6: Ejemplo EditText

ImageView

Nos permite mostrar imágenes en la pantalla.

```

----- ImageView.xml -----
1 <ImageView
2     android:id="@+id/imageView"
3     android:layout_width="wrap_content"
4     android:layout_height="wrap_content"
5     android:src="@drawable/icon"/>
-----
----- ImageView.java -----
1 final ImageView imageView1 = (ImageView) findViewById(R.id.imageView);
2 imageView1.setImageResource(R.drawable.icon);
-----

```

El icono es el que viene por defecto al crear un proyecto. Este es el resultado:

CheckBox

Es un tipo de botón con dos estados, activo o inactivo, prácticamente tiene el mismo comportamiento de un botón, una de sus características es que podemos comprobar si el botón está activo o no:

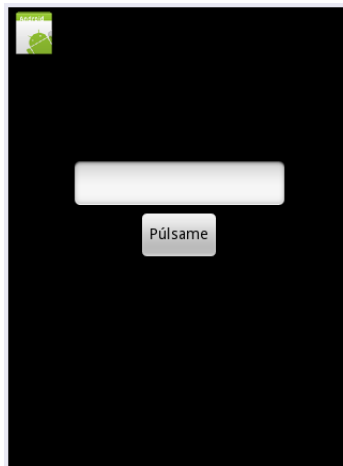


Figura 3.7: Ejemplo ImageView

CheckBox.xml

```

1 <CheckBox
2   android:layout_height="wrap_content"
3   android:layout_width="wrap_content"
4   android:text="CheckBox"
5   android:layout_centerInParent="true"
6   android:layout_below="@id/button1"
7   android:id="@+id/checkBox1" />

```

CheckBox.java

```

1 final CheckBox checkBox1 = (CheckBox) findViewById(R.id.checkBox1);
2 checkBox1.setOnCheckedChangeListener(new OnCheckedChangeListener() {
3     @Override
4     public void onCheckedChanged(CompoundButton arg0, boolean checked) {
5         if (checked)
6             Toast.makeText(checkBox1.getContext(),
7                             "Activo", Toast.LENGTH_LONG).show();
8         else
9             Toast.makeText(checkBox1.getContext(),
10                            "Inactivo", Toast.LENGTH_SHORT).show();
11     }
12 });

```

En este caso, hemos usado como listener `onCheckedChanged`, que se ejecutará cada vez que el estado del checkbox cambie.

Estos son los componentes gráficos básicos, también disponemos de `RadioButton`, `ToggleButton` (Parecidos a los `checkbox`, pero con una luz que se ilumina al estar activos, y con la característica de que el texto cambia dependiendo de su estado, aunque esto se puede conseguir con el `checkbox` fácilmente).

En general con echar un vistazo a los métodos y listeners de cada componente, y con la documentación que ofrece javadoc en eclipse, lograremos entender como funciona cada uno, y podremos usarlos fácilmente.

3.4 Adaptadores

Un objeto Adaptador actúa como puente entre un `AdapterView` y los datos de una Vista (View). El adaptador permite el acceso a los elementos de datos, éste también es responsable de crear una vista para cada elemento en la colección de datos.

Se puede decir, que los adaptadores son colecciones de datos, que asignamos a una vista para que ésta los muestre, por ejemplo, podemos crear un `ArrayAdapter` a partir de un array de string

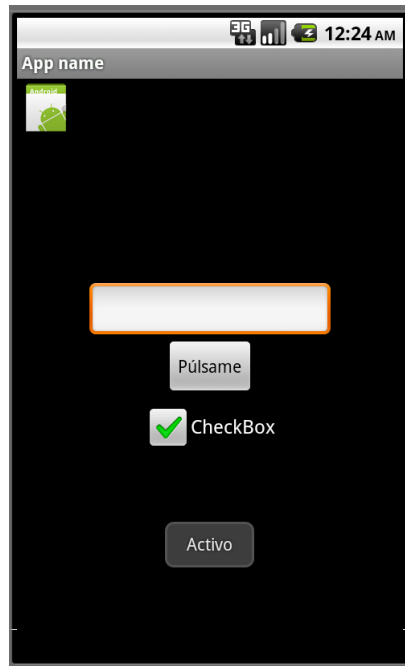


Figura 3.8: Ejemplo CheckBox

ya creado y con datos, y asignar este adaptador a un ListView, así, el ListView mostrará los datos del array.

Mediante el uso de Adapters definimos una forma común de acceder a colecciones de datos. Para que quede más claro este concepto, vamos a verlo mediante un ejemplo: Primero creamos el layout, que va a contener un ListView con un Id ya definido por android, y un TextView también con un id ya definido.

Adapter.xml

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3     android:orientation="vertical"
4     android:layout_width="fill_parent"
5     android:layout_height="fill_parent"
6     >
7     <ListView
8         android:layout_width="fill_parent"
9         android:layout_height="fill_parent"
10        android:id="@android:id/list" />
11
12     <TextView
13         android:layout_width="fill_parent"
14         android:layout_height="fill_parent"
15         android:id="@android:id/empty"
16         android:text="Lista vacía"/>
17
18 </LinearLayout>

```

Ahora, el código donde creamos el adaptador, y lo asociamos al ListView:

Adapter.java

```

1 package app.elbaultdelprogramador.adapters;
2
3 import android.app.Activity;
4 import android.os.Bundle;
5 import android.widget.ArrayAdapter;
6 import android.widget.ListAdapter;
7 import android.widget.ListView;

```

```

8
9 public class AdaptadoresActivity extends Activity {
10     /** Called when the activity is first created. */
11     @Override
12     public void onCreate(Bundle savedInstanceState) {
13         super.onCreate(savedInstanceState);
14         setContentView(R.layout.main);
15
16         //Array que asociaremos al adaptador
17         String[] array = new String[] {
18             "Elemento 1"
19             , "Elemento 2"
20             , "Elemento 3"
21             , "Elemento 4"
22             , "Elemento 5"
23             , "Elemento 6"
24         };
25
26         //Creación del adaptador, vamos a escoger el layout
27         //simple_list_item_1, que los mostr
28         ListAdapter adaptador = new ArrayAdapter(
29             this, android.R.layout.simple_list_item_1, array);
30
31         //Asociamos el adaptador a la vista.
32         ListView lv = (ListView) findViewById(android.R.id.list);
33         lv.setAdapter(adaptador);
34     }
35 }

```

Como vemos, al crear el `arrayAdapter`, tenemos que pasar tres parámetros, el contexto, un layout que se usará para dibujar cada ítem (en este caso `simple_list_item_1`, que ya viene definido por android), más adelante veremos como crear los nuestros propios, y como tercer parámetro la colección de datos.

Ahora, veamos cómo fijar un evento onclick para cada elemento de la lista.

```

AdapterEvent.java
1 //Evento que se disparará al pulsar en un elemento de la lista
2 lv.setOnItemClickListener(new OnItemClickListener() {
3
4     @Override
5     public void onItemClick(AdapterView arg0, View arg1, int arg2,
6         long arg3) {
7
8         ListAdapter la = (ListAdapter) arg0.getAdapter();
9
10        Toast.makeText(
11            arg1.getContext()
12            , la.getItem(arg2).toString()
13            , Toast.LENGTH_LONG)
14            .show();
15
16    };
17 });

```

Para realizar este tipo de cosas, android proporciona una clase llamada `ListActivity`, este tipo de clase necesita que exista una vista con el id ya definido por Android `@android:id/list` y otra con el id `@android:id/empty` (Tal y como lo definimos en nuestro layout), así, si el adaptador que le asignamos a la lista no tiene datos, se mostrará al usuario la vista empty, el código quedaría de la siguiente manera:

```

ListActivity.java
1 package app.elbauldelprogramador.adapters;
2
3 import android.app.ListActivity;
4 import android.os.Bundle;
5 import android.view.View;

```

```
6 import android.widget.ArrayAdapter;
7 import android.widget.ListAdapter;
8 import android.widget.ListView;
9 import android.widget.Toast;
10
11 public class AdaptadoresActivity extends ListActivity {
12     /** Called when the activity is first created. */
13     @Override
14     public void onCreate(Bundle savedInstanceState) {
15         super.onCreate(savedInstanceState);
16         setContentView(R.layout.main);
17
18         //Array que asociaremos al adaptador
19         String[] array = new String[] {
20             "Elemento 1"
21             , "Elemento 2"
22             , "Elemento 3"
23             , "Elemento 4"
24             , "Elemento 5"
25             , "Elemento 6"
26         };
27
28         //Creación del adaptador, vamos a escoger el layout
29         //simple_list_item_1, que los mostr
30         ListAdapter adaptador = new ArrayAdapter(
31             this, android.R.layout.simple_list_item_1, array);
32
33         //Asociamos el adaptador a la vista.
34         ListView lv = (ListView) findViewById(android.R.id.list);
35         lv.setAdapter(adaptador);
36
37     }
38     @Override
39     protected void onListItemClick(ListView l, View v,
40         int position, long id) {
41         super.onListItemClick(l, v, position, id);
42
43         Toast.makeText(
44             AdaptadoresActivity.this
45             , l.getItemAtPosition(position).toString()
46             , Toast.LENGTH_LONG)
47             .show();
48     }
49 }
```

El resultado de este código es el siguiente, para una adaptador con datos:

Y para un adaptador sin datos:

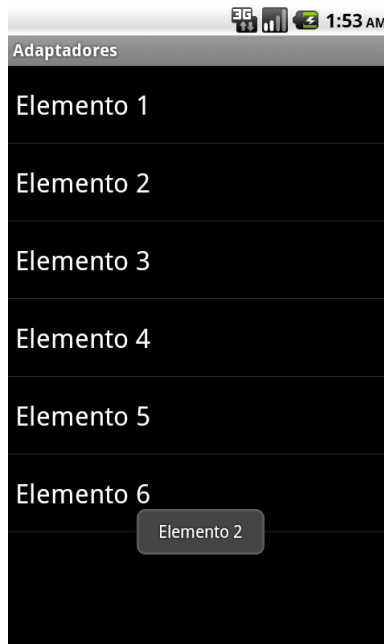


Figura 3.9: Ejemplo ListActivity

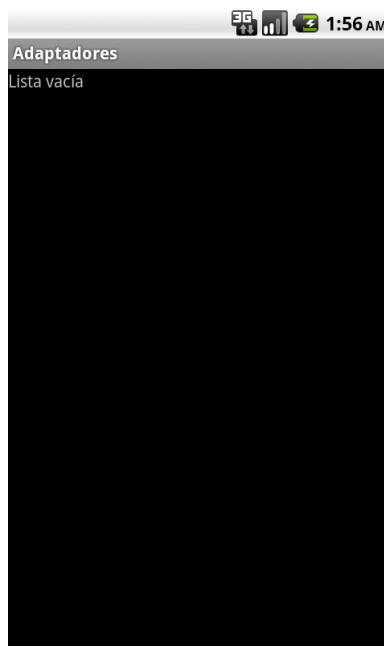


Figura 3.10: Ejemplo ListActivity vacío

Ahora vamos a ver cómo crear los nuestros propios. Para explicar el funcionamiento nos vamos a basar en un ejemplo que listará las distintas versiones de Ubuntu. Lo primero que debemos hacer es crearnos una clase que almacenará los datos de cada una de las versiones:

```
----- VersionesUbuntu.java -----  
1 package app.elbauldelprogramador.adapters2;  
2  
3 public class VersionesUbuntu {  
4  
5     private String nombre;  
6     private String version;
```

```

7   private int logotipo;
8
9   public VersionesUbuntu(String nombre, String version, int logotipo) {
10      this.nombre = nombre;
11      this.version = version;
12      this.logotipo = logotipo;
13  }
14
15  public void setNombre(String nombre) { this.nombre = nombre; }
16
17  public String getNombre() { return nombre; }
18
19  public void setVersion(String version) { this.version = version; }
20
21  public String getVersion() { return version; }
22
23  public void setLogotipo(int logotipo) { this.logotipo = logotipo; }
24
25  public int getLogotipo() { return logotipo; }
26
27 }

```

A continuación vamos a crear el adaptador desde cero, siendo necesario extender de la clase *BaseAdapter*:

```

----- VersionesUbuntuAdapter.java -----
1  import android.widget.TextView;
2
3  public class VersionesUbuntuAdapter extends BaseAdapter{
4
5      private ArrayList<VersionesUbuntu> listadoVersiones;
6      private LayoutInflater lInflater;
7
8      public VersionesUbuntuAdapter(Context context,
9          ArrayList<VersionesUbuntu> versiones) {
10
11          this.lInflater = LayoutInflater.from(context);
12          this.listadoVersiones = versiones;
13      }
14
15      @Override
16      public int getCount() { return listadoVersiones.size(); }
17
18      @Override
19      public Object getItem(int arg0)
20          { return listadoVersiones.get(arg0); }
21
22      @Override
23      public long getItemId(int arg0) { return arg0; }
24
25      @Override
26      public View getView(int arg0, View arg1, ViewGroup arg2) {
27          ContenedorView contenedor = null;
28
29          if (arg1 == null) {
30              arg1 = lInflater.inflate(R.layout.lista_versiones_ubuntu, null);
31
32              contenedor = new ContenedorView();
33              contenedor.nombreVersion =
34                  (TextView) arg1.findViewById(R.id.nomVersion);
35              contenedor.numeroVersion =
36                  (TextView) arg1.findViewById(R.id.numVersion);
37              contenedor.logoVersion =
38                  (ImageView) arg1.findViewById(R.id.logo);
39
40              arg1.setTag(contenedor);
41          } else
42              contenedor = (ContenedorView) arg1.getTag();
43
44          VersionesUbuntu versiones = (VersionesUbuntu) getItem(arg0);

```

```

45     contenedor.nombreVersion.setText(versiones.getNombre());
46     contenedor.numeroVersion.setText(versiones.getVersion());
47     contenedor.logoVersion.setImageResource(versiones.getLogotipo());
48
49     return arg1;
50 }
51
52 class ContenedorView{
53     TextView nombreVersion;
54     TextView numeroVersion;
55     ImageView logoVersion;
56 }
57 }

```

Lo que hace esta clase es lo siguiente, en su constructor, carga un objeto *LayoutInflater*, que infla los objetos XML del layout para que podamos usarlos, convirtiéndolos en un objeto java. También tenemos tres métodos sencillos (*getCount*, *getItem*, *getItemId*), que se encargan de devolver el número de elementos de la colección, un elemento en concreto y el identificador de un elemento. El método *getView()* se invoca cada vez que hay que dibujar la lista. En este caso, para la lista, hemos usado un layout personalizado, que es el siguiente:

```

----- mainlayout.xml -----
1  <?xml version="1.0" encoding="utf-8"?>
2  <RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
3      android:layout_width="fill_parent"
4      android:layout_height="fill_parent">
5
6
7      <ImageView
8          android:id="@+id/logo"
9          android:src="@drawable/ocelot"
10         android:layout_alignParentLeft="true"
11         android:padding="5px"
12         android:layout_width="128px"
13         android:layout_height="128px"/>
14
15     <TextView
16         android:id="@+id/nomVersion"
17         android:layout_toRightOf="@id/logo"
18         android:layout_width="fill_parent"
19         android:layout_height="wrap_content"
20         android:paddingTop="15dip"
21         android:text="VersionNom" />
22
23     <TextView
24         android:id="@+id/numVersion"
25         android:layout_below="@id/nomVersion"
26         android:layout_toRightOf="@id/logo"
27         android:layout_width="fill_parent"
28         android:layout_height="wrap_content"
29         android:text="VersionNum" />
30
31 </RelativeLayout>

```

Por último, tan solo queda usar el adaptador que hemos creado en los pasos anteriores en nuestra Actividad principal, en este caso, una *ListActivity*:

```

----- MainActivity.java -----
1  package app.elbaultdelprogramador.adapters2;
2
3  import java.util.ArrayList;
4
5  import android.app.ListActivity;
6  import android.os.Bundle;
7
8  public class Adaptadores2Activity extends ListActivity {
9      /** Called when the activity is first created. */

```



```

10  @Override
11  public void onCreate(Bundle savedInstanceState) {
12      super.onCreate(savedInstanceState);
13      setContentView(R.layout.main);
14
15      ArrayList<VersionesUbuntu> versiones =
16          new ArrayList<VersionesUbuntu>();
17
18      versiones.add(
19          new VersionesUbuntu("Lucid Lynx", "10.4 LTS", R.drawable.lucid));
20
21      versiones.add(
22          new VersionesUbuntu("Maverick Meerkat", "10.10", R.drawable.u1010));
23
24      versiones.add(
25          new VersionesUbuntu("Natty Narwhal", "11.04", R.drawable.natty));
26
27      versiones.add(
28          new VersionesUbuntu("Oneiric Ocelot", "11.10", R.drawable.ocelot));
29
30      VersionesUbuntuAdapter adaptador = new VersionesUbuntuAdapter(
31          Adaptadores2Activity.this, versiones);
32      setListAdapter(adaptador);
33  }
34  }

```

Ya solo resta ejecutarlo y ver el resultado:

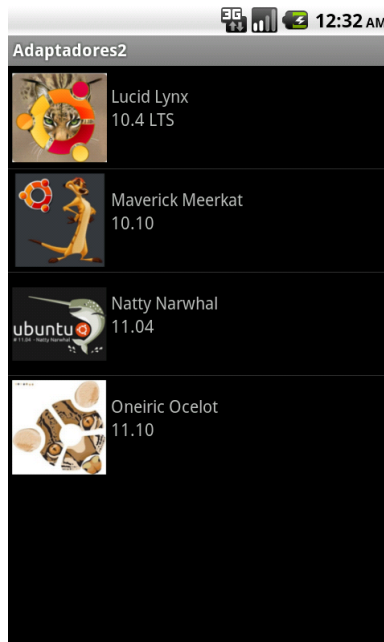


Figura 3.11: Ejemplo BaseAdapter

3.5 Menús

Los menús en las aplicaciones son algo que encontramos frecuentemente, de hecho, casi todos los terminales Android tienen un botón específico para desplegarlos.

Existen distintos tipo de menús:

- **Options Menu:** El menú típico, que se despliega al pulsar la tecla menú, que se divide en dos grupos:
 - **Icon menu:** Muestra un menú con iconos, 6 elementos como máximo.

- **Expanded Menu:** Se usa cuando hay más de 6 elementos, mostrando un elemento con la palabra 'Más'.
- **Context Menu:** Menús contextuales desplegados al realizar una pulsación larga en una View.
- **Submenús:** Menús desplegados al pulsar sobre un elemento de otro menú.

Options Menu

Lo más simple y sencillo es definir los menús en XML, colocado en `./res/menu`, para este ejemplo he definido el siguiente menú, que contiene dos elementos, un About y un Quit:

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <menu
3      xmlns:android="http://schemas.android.com/apk/res/android">
4      <item
5          android:id="@+id/about"
6          android:icon="@drawable/about"
7          android:title="About App">
8          <menu xmlns:android="http://schemas.android.com/apk/res/android">
9              <item
10                 android:id="@+id/submenu"
11                 android:title="Submenú de &quot;About App&quot;"/>
12             </menu>
13          </item>
14          <item
15              android:id="@+id/quit"
16              android:title="Quit App"
17              android:icon="@drawable/quit"/>
18  </menu>

```

Bien, voy a explicar un poco la estructura de este menú, Empezamos declarando el menú con la etiqueta `<menu>`, que contendrá todos sus elementos bajo la etiqueta `<item>`, en este caso, también tenemos un submenu, que se declara igual que el menú principal.

Los atributos de cada elemento son su identificador, el icono a mostrar y el título.

Para poder usar este menú, necesitamos inflarlo (Convertir el fichero XML en un objeto java), para hacer esto, hay que llamar a `MenuInflater.inflate()`, el código siguiente infla el fichero xml anterior en el método callback `onCreateOptionsMenu()`.

```

1  @Override
2  public boolean onCreateOptionsMenu(Menu menu) {
3      MenuInflater inflater = getMenuInflater();
4      inflater.inflate(R.menu.ejemplo_menu, menu);
5      return true;
6  }

```

Ahora, tenemos que responder a las acciones del usuario cuando pulse algún elemento de nuestro menú, para ello vamos a sobrescribir el método `onOptionsItemSelected()`:

```

1  @Override
2  public boolean onOptionsItemSelected(MenuItem item) {
3      switch (item.getItemId()) {
4          case R.id.about:
5              Toast.makeText(
6                  MenuActivity.this,
7                  "Ejemplo Menús App",
8                  Toast.LENGTH_LONG)
9                  .show();
10             return true;
11
12             case R.id.quit:

```

```

13     finish();
14     return true;
15
16     default:
17         return super.onOptionsItemSelected(item);
18     }
19
20 }

```

Context Menu

Los menús contextuales son similares a los menús mostrados al hacer click con el botón derecho de un ratón en un PC, para crearlos, debemos sobrescribir el método *onCreateContextMenu()*, donde inflaremos el archivo xml.

```

MainActivity.java
1 @Override
2 public void onCreateContextMenu(ContextMenu menu, View v,
3     ContextMenuInfo menuInfo) {
4     super.onCreateContextMenu(menu, v, menuInfo);
5     MenuInflater inflater = getMenuInflater();
6     inflater.inflate(R.menu.ejemplo_menu, menu);
7 }

```

Al igual que en los options menu, tenemos que responder a las acciones del usuario:

```

MainActivity.java
1 @Override
2 public boolean onContextItemSelected(MenuItem item) {
3     switch (item.getItemId()) {
4         case R.id.about:
5             Toast.makeText(
6                 MenuActivity.this,
7                 "Ejemplo Menús App",
8                 Toast.LENGTH_LONG)
9                 .show();
10            return true;
11
12            case R.id.quit:
13                finish();
14                return true;
15
16            default:
17                return super.onContextItemSelected(item);
18        }
19    }

```

Pero este menú contextual no se va a mostrar, ya que tenemos que asociarlo para que se lance al realizar una pulsación prolongada sobre una view, en este caso un botón:

```

final Button boton = (Button) findViewById(R.id.button1);
registerForContextMenu(boton);

```

Aquí dejo algunas capturas de pantalla de la aplicación:

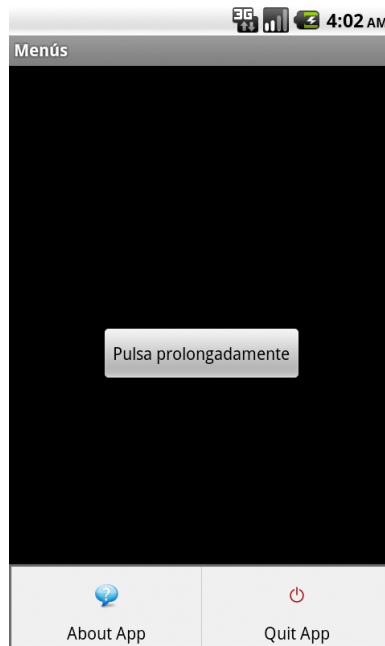


Figura 3.12: Ejemplo OptionMenu

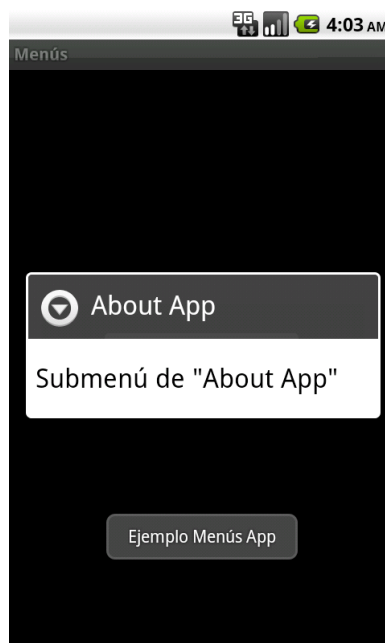


Figura 3.13: Ejemplo OptionMenu

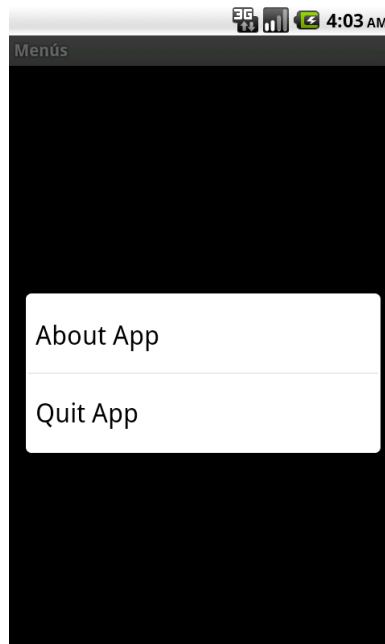


Figura 3.14: Ejemplo OptionMenu

Podéis encontrar más información sobre Menús en la página oficial de Android.

3.6 Diálogos y Notificaciones

En ocasiones hay que mostrar mensajes al usuario para informarle del estado de la aplicación, o del estado de las operaciones que se estén llevando a cabo. Android dispone de tres tipos de notificaciones:

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

Notificaciones Toast

Una notificación Toast es un mensaje que se muestra superpuesto en la pantalla. Solo ocupa el espacio necesario para mostrar la alerta, mientras tanto, la actividad que estaba visible puede seguir usándose. Este tipo de notificaciones se muestran durante un periodo de tiempo y desaparecen, no permiten interactuar con ellas. Debido a que un Toast se crea mediante un servicio en segundo plano, puede aparecer aunque la aplicación no esté visible.

A lo largo del manual se han usado mucho, la cabecera de la función es:

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

Para pasar el contexto, tenemos varias posibilidades, `NombreActividad.this`, o `getApplicationContext()`. Para fijar la duración del mensaje, usamos una de las dos constantes predefinidas, `Toast.LENGTH_SHORT` ó `Toast.LENGTH_LONG`. En este caso, vamos a crear un layout personalizado para mostrar el Toast:

```
_____ toast_layout.xml _____  
1 <?xml version="1.0" encoding="utf-8"?>  
2 <LinearLayout  
3     xmlns:android="http://schemas.android.com/apk/res/android"  
4     android:id="@+id/toastLayout"  
5     android:orientation="horizontal"
```

```

6  android:layout_width="match_parent"
7  android:layout_height="match_parent"
8  android:padding="10dp"
9  android:background="#DAAA">
10
11  <ImageView
12      android:layout_width="48px"
13      android:layout_height="48px"
14      android:src="@drawable/ok"
15      android:padding="5dp"
16      android:id="@+id/ok"/>
17
18  <TextView
19      android:id="@+id/textview"
20      android:layout_width="wrap_content"
21      android:layout_height="fill_parent"
22      android:text="Toast con layout personalizado"
23      android:textColor="#fff"
24      android:gravity="center_vertical|center_horizontal"/>
25
26 </LinearLayout>

```

Hay que asignar un id al LinearLayout, que usaremos posteriormente. También hemos creado un ImageView para mostrar un icono, y un TextView para mostrar el mensaje.

El siguiente paso es inflar este layout desde el código:

```

ToastMainActivity.java
1  LayoutInflater inflater = getLayoutInflater();
2  View layout = inflater.inflate(
3      R.layout.toast_layout
4      , (ViewGroup) findViewById(R.id.toastLayout));
5
6  Toast toast = new Toast(getApplicationContext());
7  toast.setGravity(Gravity.CENTER_VERTICAL, 0, 0);
8  toast.setDuration(Toast.LENGTH_LONG);
9  toast.setView(layout);
10 toast.show();

```

Listo, al ejecutar la aplicación tendremos un Toast como este:

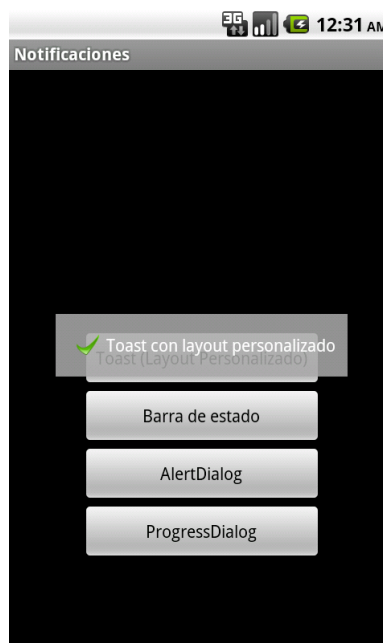


Figura 3.15: Ejemplo Toast

Para saber más acerca de los mensajes toast puedes visitar: <http://developer.android.com/guide/topics/ui/notifiers/toasts.html>

Notificaciones en la barra de estado

Este tipo de notificaciones muestran un icono en la barra de estado, cuando desplegamos esta barra, veremos el icono acompañado de un texto descriptivo indicando que ha pasado algo (Como que hemos recibido un nuevo mensaje, o un correo electrónico).

Los pasos necesarios para crear este tipo de notificaciones son, usar el gestor de notificaciones del sistema (NotificationManager) y posteriormente crear un objeto Notification en el que configuraremos nuestra notificación. Vamos a ver como hacerlo

```

NotificationMainActivity.java
1 NotificationManager mNotificationManager =
2   (NotificationManager) getSystemService(Context.NOTIFICATION_SERVICE);
3
4 //Agregando el icono, texto y momento para lanzar la notificación
5 int icon = R.drawable.ok;
6 CharSequence tickerText = "Notification Bar";
7 long when = System.currentTimeMillis();
8
9 Notification notification = new Notification(icon, tickerText, when);
10 Context context = getApplicationContext();
11 CharSequence contentTitle = "Notificación en barra";
12 CharSequence contentText = "Mensaje corto de la notificación";
13
14 //Agregando sonido
15 notification.defaults |= Notification.DEFAULT_SOUND;
16 //Agregando vibración
17 notification.defaults |= Notification.DEFAULT_VIBRATE;
18
19 Intent notificationIntent = new Intent(this, NotificacionesActivity.class);
20 PendingIntent contentIntent = PendingIntent.getActivity(
21   this
22   , 0
23   , notificationIntent
24   , 0);
25 notification.setLatestEventInfo(context, contentTitle, contentText, contentIntent);
26
27 mNotificationManager.notify(HELLO_ID, notification);

```

El resultado es el siguiente: Al igual que los Toast, se puede crear un layout personalizado, para más información visita: <http://developer.android.com/guide/topics/ui/notifiers/notifications.html>

Diálogos

Por último, vamos a ver los diálogos, que son ventanas que se muestran delante de las actividades, y pueden recibir acciones del usuario, hay varios tipos:

- AlertDialog
- ProgressDialog
- DatePickerDialog
- TimePickerDialog

Si necesitamos un Diálogo que no sea uno de los de arriba, podemos extender de la clase *Dialog*, y crear el nuestro propio.

La clase Activity implementa métodos para gestionar los diálogos, son:

- **onCreateDialog(int):** Encargado de crear el diálogo.
- **onPrepareDialog(int):** Llamado justo antes de mostrarlo.

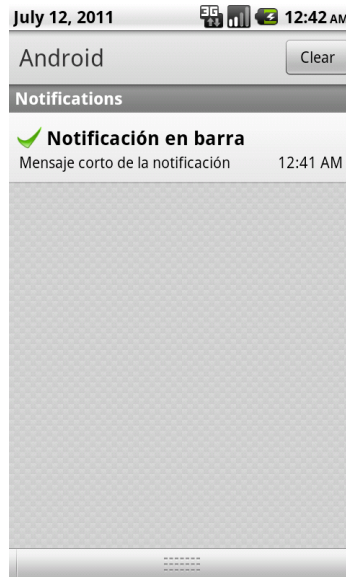


Figura 3.16: Ejemplo Notificaciones

- *showDialog(int)*: Para mostrarlo.
- *dismissDialog(int)*: cierra el diálogo, guardando su estado.
- *removeDialog(int)*: cierra el dialogo eliminándolo por completo.

Vamos a ver un ejemplo de AlertDialog, que preguntará si queremos salir de la aplicación:

```

DialogMainActivity.java
1 AlertDialog.Builder dialog = new AlertDialog.Builder(this);
2
3 dialog.setMessage("¿Salir?");
4 dialog.setCancelable(false);
5 dialog.setPositiveButton("Si", new DialogInterface.OnClickListener() {
6
7     @Override
8     public void onClick(DialogInterface dialog, int which) {
9         NotificacionesActivity.this.finish();
10    }
11 });
12 dialog.setNegativeButton("No", new DialogInterface.OnClickListener() {
13
14     @Override
15     public void onClick(DialogInterface dialog, int which) {
16         dialog.cancel();
17    }
18 });
19 dialog.show();

```

También vamos a ver un ProgressDialog, indefinido (Que nunca termina).

```

ProgressDialog.show(
    NotificacionesActivity.this
    , "ProgressDialog"
    , "Ejemplo dialogo de progreso"
    , true
    , true);

```

Los dos últimos parámetros son para que el diálogo sea indeterminado, y para que se pueda cerrar con la flecha de "atrás" del terminal.



Figura 3.17: Ejemplo Dialogos

3.7 Estilos y Temas

Un estilo es una colección de propiedades que especifican que aspecto ha de tener un objeto View o una ventana. Con los estilos podemos definir propiedades como la altura, relleno, color del texto, fondo etc. Los estilos en Android comparten la filosofía de las hojas de estilo en cascada (CSS), permitiendo separar el diseño del contenido.

Como podemos comprobar con este ejemplo, el código queda mucho más limpio usando estilos:

Sin estilos:

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

Con estilos:

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

Definiendo estilos

Los estilos se definen en un fichero de recursos distinto al layout, colocado en el directorio */res/values*. Éstos ficheros no tienen porqué tener un nombre en concreto, pero por convención se suelen llamar *styles.xml* ó *themes.xml*. El nodo raíz de estos archivos debe ser *<resources>*.

Para cada estilo que queramos definir, hay que añadir un elemento *<style>* y un atributo *name* para ese estilo (es obligatorio), después, añadiremos un elemento *<item>* para cada propiedad del estilo, que debe tener obligatoriamente el atributo *name* que declara la propiedad del estilo y su valor. Los valores para *<item>* pueden ser una palabra clave, valor hexadecimal, una referencia a un recurso u otro valor dependiendo de la propiedad del estilo. Veamos un ejemplo:

```
<style name="CodeFont" parent="@android:style/TextAppearance.Medium">
  <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>
```

En tiempo de compilación, los elementos se convierten en un recurso que podremos referenciar posteriormente mediante el atributo `name` del estilo, como vimos en el primer ejemplo (`@style/CodeFont`).

El atributo *parent* es opcional y especifica el ID de otro estilo del cual queremos heredar sus propiedades, pudiendo así sobreescribirlas.

Herencia

Como acabamos de ver, el atributo `parent` sirve para heredar propiedades de otros estilos, podemos heredar tanto de estilos del sistema como de los nuestros propios:

Del sistema:

```
<style name="CodeFont" parent="@android:style/TextAppearance">
  <item name="android:textColor">#00FF00</item>
</style>
```

De nuestros propios estilos:

```
<style name="CodeFont.Red">
  <item name="android:textColor">#FF0000</item>
</style>
<style name="CodeFont.Red.Big">
  <item name="android:textSize">30sp</item>
</style>
```

En este caso, no usamos el atributo `parent`, ya que estamos usando un estilo propio, también se puede apreciar que podemos heredar cuantas veces queramos, como es el caso de (`CodeFont.Red.Big`)

Aplicar estilos y temas a la interfaz gráfica

Hay dos formas de aplicar estilos a la UI:

- A una View individual, añadiendo el atributo `style` a un elemento del layout.
- A una aplicación o actividad completa, mediante el atributo `android:theme` del elemento `<activity>` o `<application>` en el Android manifest.

Como vimos al principio, para aplicar un estilo a una View concreta usamos `style="@style/NombreDelEstilo`. Para aplicar un tema a una actividad o aplicación usaremos:

```
<application android:theme="@style/CustomTheme">
```

Para aplicarlos sobre actividades, usamos:

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

En este caso, estos temas ya vienen predefinidos, y se ven así, respectivamente: Para saber más acerca de los estilos y temas visite la página oficial: [Applying Styles and Themes](#).

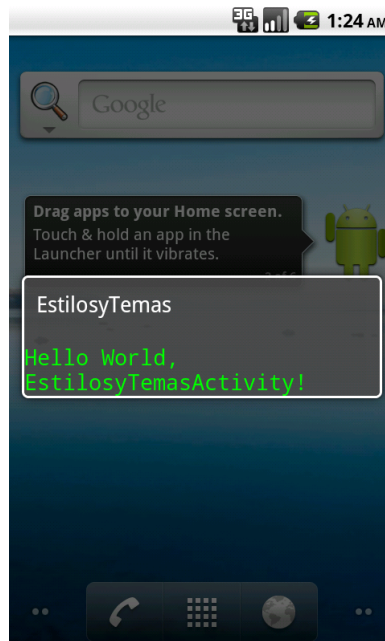


Figura 3.18: Ejemplo Temas

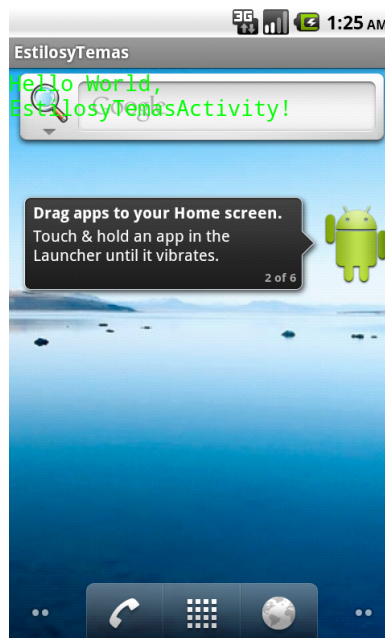


Figura 3.19: Ejemplo Temas

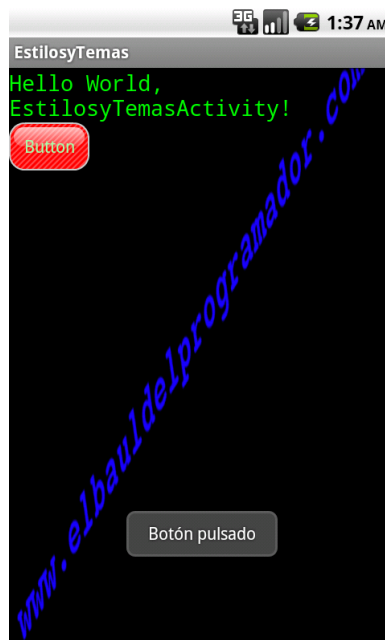


Figura 3.20: Ejemplo Temas



4 — Recursos

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

Separar los recursos permite proporcionar alternativas para dar soporte a las distintas configuraciones de dispositivos, como idiomas o tamaños de pantalla. Para conseguir compatibilidad con las diferentes configuraciones, debemos organizar los recursos en el directorio `.res` de nuestros proyectos, dentro de subdirectorios para agruparlos por tipo y configuración.

Un recurso puede usarse por defecto (Se mostrará en cualquier dispositivo, independientemente de su configuración), o pueden especificarse recursos alternativos (Que se mostrarán en los dispositivos para configuraciones determinadas), veámoslo con unas imágenes de ejemplo:

Dos dispositivos distintos, usando recursos por defecto:

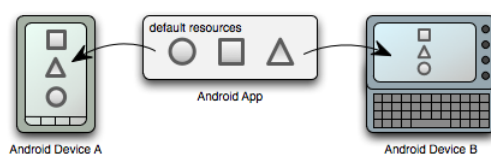


Figura 4.1: Recursos por defecto

Dos dispositivos distintos, usando recursos alternativos:

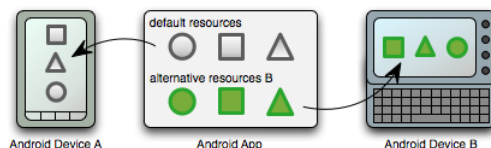


Figura 4.2: Recursos por alternativos

Por ejemplo, podemos crear iconos más pequeños para que sean mostrados en los terminales con pantallas más pequeñas o diseñar una disposición de pantalla diferente para cuando la

aplicación se esté ejecutando en modo apaisado, para conseguir esto, simplemente creamos una carpeta de recursos añadiendo el sufijo que indica la situación en la que debe usarse.

Para el caso del idioma, crearíamos un archivo xml con las cadenas traducidas a dicho idioma, en este caso inglés, y lo colocaríamos dentro de `.res/values-en/strings.xml`.

Para el caso del layout personalizado cuando la pantalla esté en modo apaisado, meteríamos nuestro layout dentro de `res/layout-land/`.

Para saber más acerca de los tipos de sufijos que se pueden usar, visita Providing Resources en la página oficial de Android.

4.1 Usando recursos

A todos los recursos que colocamos en las subcarpetas de `.res/` se puede acceder a través de la clase R de nuestro proyecto. Esta clase R la genera el comando `aapt` en una pasada anterior a la compilación (Eclipse, por defecto, la va generando continuamente conforme cambiamos los recursos). Contiene todos los identificadores de recursos para poder referenciarlos.

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

Así pues los ID de recurso están compuestos de: Clase R que contienen todos los recursos. Subclase de recurso, cada grupo tiene la suya (`drawable`, `string`, `style`, `layout`...). Nombre del recurso que, según el tipo, será: el nombre del fichero sin la extensión o el atributo xml “`android:name`” si es un valor sencillo (cadena, estilo, etc.). Tenemos dos formas de acceder a los recursos definidos en la clase R: En el código, accediendo a las propiedades de la clase R directamente (`R.string.nombre`). En los ficheros XML, usando una notación especial: `@grupo_recursos/ nombre_recurso`, es decir, el recurso anterior se accedería con `@string/nombre`. Si lo que queremos es acceder a un recurso definido por el sistema antepondremos el prefijo `android`:

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

Referenciando atributos de estilo

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

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

Así por ejemplo si queremos colocar un texto pequeño usaremos:

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

Si queremos, también podemos utilizar nuestros propios atributos. Primero lo definimos con un tag “`attr`” dentro de `.res/values/attr.xml`:

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

Ahora ya podemos usar esa propiedad en nuestros estilos. Primero definimos un estilo de texto llamado “`TituloRojo`”, y luego lo aplicamos al atributo que hemos creado llamado “`Cabecera`”. Obsérvese que como es un atributo propio, no usamos el espacio de nombres “`android`”:

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

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <FrameLayout
3  xmlns:android="http://schemas.android.com/apk/res/android"
4      android:layout_width="fill_parent"
5      android:layout_height="fill_parent">
6      <TextView
7          android:layout_width="fill_parent"
8          android:layout_height="fill_parent"
9          android:text="No hay datos disponibles"
10         style="?attr/cabecera" />
11  </FrameLayout>

```

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <resources>
3  <style name="MiTema" parent=
4      "@android:style/Theme.Light">
5      <item name="android:windowBackground">
6          @drawable/fondo
7      <item name="cabecera">
8          @style/TituloRojo</item>
9  </style>
10 <style name="TituloRojo"
11     parent="@android:style/TextAppearance.Large">
12     <item name="android:textColor">#FF0000</item>
13     <item name="android:textStyle">bold</item>
14 </style>
15 </resources>

```

Como ya hemos visto, los recursos juegan un papel muy importante en la arquitectura Android. Un recurso en Android es un archivo (como un fichero de música) o un valor (como el título de un Diálogo) que está ligado a una aplicación ejecutable. Estos archivos están ligados a un ejecutable de tal manera que podemos cambiarlos sin necesidad de recompilar la aplicación.

Los ejemplos de recursos más familiares son cadenas de texto, colores e imágenes. En lugar de escribir las cadenas de texto en el código fuente, usamos sus IDs. Esta indirección nos permite cambiar el valor de la cadena sin tener que cambiar el código fuente.

Existen mucho recursos en Android, que vamos a ver a lo largo del libro. Empezaremos por un recurso muy común, los string:

4.2 Recursos string

Android permite definir strings en uno o más archivos XML de recursos. Estos archivos están bajo el directorio `./res/values`. El nombre del archivo XML para este tipo de recurso puede ser cualquiera, pero por convención se suele llamar *strings.xml*. Veamos un ejemplo de este fichero:

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <resources>
3      <string name="hello">Hello</string>
4      <string name="app_name">Hello_app</string>
5  </resources>

```

Cuando este archivo se crea o modifica, el plugin ADT de eclipse automáticamente creará o actualizará una clase java de nuestra aplicación llamada *R.java* alojada en el directorio `./gen`, que contiene los IDs únicos para las dos cadenas que acabamos de crear.

Para el fichero `strings.xml` que acabamos de crear, tendremos lo siguiente en la clase *R*:


```
package nombre.de.nuestro.paquete;

public final class R {
    //.. otras entradas dependiendo de tu proyecto y aplicacion

    public static final class string {
        //.. otras entradas dependiendo de tu proyecto y aplicacion
        public static final int app_name=0x7f040001;
        public static final int hello=0x7f040000;
        //.. otras entradas dependiendo de tu proyecto y aplicacion
    }
    //.. otras entradas dependiendo de tu proyecto y aplicacion
}
```

Como vemos como primero R.java define una clase principal en el paquete raíz: *public final class R*. Después, define una clase interna llamada *public static final class string*. R.java crea esta clase estática interna como espacio de nombres para guardar los IDs de los recursos string. La definición de los dos *public static final int* llamados *app_name* y *hello* son los IDs de los recursos que representan nuestras cadenas de texto. Podemos usar estos IDs en cualquier lugar de nuestro código mediante *R.string.hello* o *R.string.app_name*.

La estructura del fichero *string.xml* es muy fácil de seguir. Tenemos un elemento raíz llamado *<resources>*, seguido por uno o más elementos hijos *<string>*. Cada elemento *<string>* tiene una propiedad llamada *name* que será la que usará como identificador R.java.

Para comprobar que se permiten varios recursos de string en el directorio values, vamos a crear otro fichero llamado *strings1.xml* con lo siguiente:

```
./values/string1.xml
1 <?xml version="1.0" encoding="utf-8"?>
2 <resources>
3   <string name="hello1">Hello1</string>
4   <string name="app_name1">Hello_app1</string>
5 </resources>
```

Ahora, el plugin ADT de eclipse se encargará de actualizar el fichero R.java, que contendrá lo siguiente:

```
package nombre.de.nuestro.paquete;

public final class R {
    //.. otras entradas dependiendo de tu proyecto y aplicacion

    public static final class string {
        //.. otras entradas dependiendo de tu proyecto y aplicacion
        public static final int app_name=0x7f040001;
        public static final int hello=0x7f040000;
        public static final int app_name1=0x7f040006;
        public static final int hello1=0x7f040005;
        //.. otras entradas dependiendo de tu proyecto y aplicacion
    }
    //.. otras entradas dependiendo de tu proyecto y aplicacion
}
```

4.3 Recursos Layout

En Android, cada pantalla de una aplicación habitualmente se carga desde un fichero XML que actúa de recurso. Un recurso layout es un recurso clave que se usa en Android para componer la UI de nuestra aplicación. Vamos a considerar el segmento de código siguiente como ejemplo de una actividad en Android.

```
LayoutMainActivity.java
1 public class PrincipalActivity extends Activity {
2     /** Called when the activity is first created. */
```

```

3     @Override
4     public void onCreate(Bundle savedInstanceState) {
5         super.onCreate(savedInstanceState);
6         setContentView(R.layout.main);
7     }
8 }

```

La línea `setContentView(R.layout.main);` señala que hay una clase estática llamada *R.layout* y, que dentro de esa clase hay una constante entera llamada *main* que apunta a una vista definida por un fichero de recursos layout XML. El nombre del fichero XML es *main.xml*, el cual debe estar en el directorio *./res/layout*. El contenido de este fichero es el siguiente:

```

mainlayout.xml
1 <?xml version="1.0" encoding="utf-8"?>
2 <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3     android:orientation="vertical"
4     android:layout_width="fill_parent"
5     android:layout_height="fill_parent"
6 >
7     <TextView
8         android:id="@+id/text1"
9         android:layout_width="fill_parent"
10        android:layout_height="wrap_content"
11        android:text="@string/hello" />
12    <Button
13        android:id="@+id/b1"
14        android:layout_width="fill_parent"
15        android:layout_height="wrap_content"
16        android:text="@string/hello"/>
17
18 </LinearLayout>

```

Vamos a ver la composición de este fichero, como elemento raíz tenemos un elemento llamado `<LinearLayout>`, que contiene un *TextView* seguido de un *Button*. Un *LinearLayout* coloca a sus hijos uno detrás de otro vertical u horizontalmente.

Para cada pantalla que queramos hacer, necesitaremos ficheros layout distintos, mejor dicho, cada layout debe estar en un fichero, como *./res/layout/screen1.xml* y *./res/layout/screen2.xml*.

Por cada archivo de layout que tengamos en *./res/layout*, se generará una entrada en R.java. Por ejemplo, si tenemos estos dos archivos, *file1.xml* y *file2.xml*, en R.java aparecerá:

```

// ...
public static final class layout {
    public static final int file1=0x7f030000;
    public static final int file2=0x7f030001;
}
// ..

```

Las vistas definidas en estos layout, como el *TextView* son accesibles mediante código java a través sus IDs de recursos generadas en R.java.

```

TextView tv = (TextView) this.findViewById(R.id.text1);
tv.setText("Texto para el TextView");

```

En este ejemplo, hemos localizado el *TextView* usando el método *findViewById()* de la clase *Activity*. La constante *R.id.text1* corresponde al ID definido para el *TextView* en el fichero XML, que creamos de la siguiente manera:

```

<TextView
    android:id="@+id/text1"
    . . .
/>

```

El valor del atributo *id*, indica que la constante llamada *text1* será usada para identificar únicamente a esa vista. El signo + de *+id/text1* significa que el ID *text1* será creado si no existe.

4.4 Sintaxis de los recursos

Independientemente del tipo de recurso (de string y layout son los dos que hemos visto hasta ahora), Todos los recursos Android están identificados (o referenciados) por sus ids en código fuente Java.

La sintaxis que usamos para crear un ID de recurso en un fichero XML se llama *resource-reference syntax* (sintaxis de referencia a recurso). La sintaxis del atributo id que vimos anteriormente (`@+id/text1`) tiene la siguiente estructura formal:

```
@[package:]type/name
```

El tipo (*type*), corresponde a uno de los espacios de nombres de tipos de recursos disponible en R.java, que son algunos de los siguientes:

- R.drawable
- R.id
- R.layout
- R.string
- R.attr
- R.plural
- R.array

En sintaxis de referencia a recursos XML, estos tipos se nombrarían así:

- drawable
- id
- layout
- string
- attr
- plurals
- string-array

La parte *name* en `@[package:]type/name` corresponde al nombre dado al recurso (text1 en nuestro ejemplo anterior), este nombre, también será representado como una constante entera en R.java.

Si no se especifica ningún paquete en `@[package:]` (de ahí que en la representación formal de la sintaxis aparezca entre corchetes, que quiere decir que es opcional); el par tipo/nombre será resuelto basándose en los recursos locales y en el archivo R.java local de nuestra aplicación.

Si especificamos *android:type/name*, el id referenciado será resuelto usando el paquete android y específicamente a través del archivo android.R.java. Podemos usar cualquier nombre de paquete java en el lugar de `@[package:]` para localizar el archivo R.java correcto y resolver la referencia al recurso en cuestión. Ahora que conocemos la sintaxis, vamos a analizar unos ejemplos. Nótese que la parte izquierda del ID android:id no es parte de la sintaxis. "android:id" simplemente indica que vamos a crear un id para un control como lo es el *TextView*.

EjemploId.xml

```

1 <TextView android:id="text">
2 <!-- Error de compilación, como id no tomará cadenas de texto sin formato. -->
3
4 <TextView android:id="@text">
5 <!-- Sintaxis incorrecta. No dispone de tipo y nombre-->
6 <!-- debería ser @id/text, @+id/text o @string/string1-->
7 <!-- obtendremos el siguiente error: "No resource type specified"-->
8
9 <TextView android:id="@id/text">
10 <!-- Error: No hay ningún recurso que coincida con el id "text"-->
11 <!-- a no ser que lo hayamos definido nosotros mismos con anterioridad.-->
12
```

```
13 <TextView android:id="@android:id/text">
14 <!-- Error: el recurso no es público-->
15 <!-- lo que indica que no hay tal identificación en android.R.id-->
16 <!-- Por supuesto esto será válido si el archivo android R.java definió
17     un id con este nombre.-->
18
19 <TextView android:id="@+id/text">
20 <!-- Correcto: crea un id llamado text en el paquete R.java local.-->
```

En la sintaxis “@+id/text”, el signo + tiene un significado especial. Le dice a Android que el ID puede no existir aún y, que en ese caso, cree uno nuevo y lo llame text.

4.5 Recursos compilados y no compilados

Android soporta principalmente dos tipos de recursos: archivos XML y archivos raw (como imágenes, audio y vídeo). Incluso dentro de los archivos XML, has podido ver que en algunos casos los recursos están definidos como valores dentro del archivo XML (las cadenas de texto, por ejemplo). En otras ocasiones, un archivo XML es un recurso por sí mismo, como los Layout.

Podemos encontrar dos tipos de archivos XML: uno se compilará a un formato binario y el otro se copiará tal como es al dispositivo. Como ejemplo podemos poner los ficheros XML de recursos string y los ficheros de layout, ambos se compilarán a un formato binario antes de formar parte del paquete de instalación. Estos ficheros XML tienen formatos predefinidos en los que los nodos XML se traducen a ids.

Por supuesto, también se pueden elegir archivos XML para guardarlos con su formato propio, estos archivos no serán interpretados y se les asignará un ID de recurso para poder identificarlos. Sin embargo, podemos querer que estos ficheros también se compilen a un formato binario. Para lograrlo, hay que colocarlos bajo el directorio `./res/xml`. En tal caso, deberemos usar los lectores XML que proporciona Android para acceder a los nodos del fichero.

Si colocamos cualquier tipo de archivo, incluyendo ficheros XML, bajo `./res/raw`, no se compilarán a un formato binario. Deberemos entonces, para leerlos, usar Stream. Los archivos de audio y vídeo entran en esta categoría también.

Hay que destacar, que al formar parte el directorio rar de `./res/*`, incluso estos archivos raw de audio y vídeo serán localizados como todos los demás recursos.

Como vimos, los recursos se almacenan en varios subdirectorios según su tipo, a continuación vemos algunos de los subdirectorios más importantes de la carpeta res:

- anim: Archivos compilados de animaciones.
- drawable: Bitmaps
- layout: definición de vistas y UI.
- values: Arrays, colores, dimensiones, strings y estilos.
- xml: ficheros XML arbitrarios.
- raw: ficheros raw no compilados.

El compilador de recursos AAPT (*Android Asset Packaging Tool*), compila todos los recursos excepto los raw y lo coloca en el fichero .apk (Android package) final. Este fichero es el que contiene el código de las aplicaciones Android y los recursos, similar a los .jar de Java. Estos ficheros son los que usamos para instalar aplicaciones en Android.

4.6 Arrays de strings

Se pueden definir arrays de strings como recursos en cualquier archivo bajo el subdirectorio `./res/values`. Para definirlos, usaremos un nodo XML llamado string-array. Este nodo es un hijo

de resources, al igual que el nodo string. A continuación, vamos a ver como crear un array de strings:

```
<resources>
<string-array name="test_array">
  <item>uno</item>
  <item>dos</item>
  <item>tres</item>
</string-array>
</resources>
```

Una vez definido el recurso, podemos usarlo en el código Java de la siguiente manera:

```
//Accedemos al objeto 'Recursos' desde la Activity
Resources res = nombre-de-la-actividad.getResources();
String strings[] = res.getStringArray(R.array.test_array);

//Mostramos el array
for (String s: strings)
    Log.d("ejemplo", s);
```

4.7 Plurales

Los recursos Plurals son un conjunto de strings. Estos strings representan una forma de escribir cantidades numéricas, por ejemplo, cuantos huevos hay en una cesta. Vamos a ver un ejemplo:

```
Hay 1 huevo
Hay 2 huevos
Hay 10 huevos
```

Como puedes notar, las frases son iguales para los números 2 y 10. Sin embargo, la frase para 1 huevo es diferente. Android permite representar esta variación con el recurso llamado plurals. En el siguiente ejemplo vemos como se representan estas dos variaciones.

```
<resources>
<plurals name="huevos_en_una_cesta">
  <item quantity="one">Hay 1 huevo</item>
  <item quantity="other">Hay %d huevos</item>
</plurals>
</resources>
```

Las dos variaciones se representan como dos cadenas de texto diferentes bajo el mismo plural. Ahora, podemos usarlo desde el código Java para mostrar correctamente la cadena correspondiente a la cantidad de huevos. El primer parámetro de *getQuantityString()* es el ID del plural. El segundo selecciona la cadena a usar. Cuando el valor de la cantidad es 1, usamos la cadena tal como es. Cuando el valor es distinto a 1, debemos pasar un tercer parámetro que será el valor a colocar en el lugar de %d. Siempre deberá haber al menos 3 parámetros si usamos cadenas formateadas en los plurales.

```
Resources res = tu_actividad.getResources();
String s1 = res.getQuantityString(R.plurals.huevos_en_una_cesta, 0, 0);
String s2 = res.getQuantityString(R.plurals.huevos_en_una_cesta, 1, 1);
String s3 = res.getQuantityString(R.plurals.huevos_en_una_cesta, 2, 2);
String s4 = res.getQuantityString(R.plurals.huevos_en_una_cesta, 10, 10);
```

Con este código, cada cantidad se mostrará con su correcta cadena de texto.

Existen otras posibilidades que podemos aplicar al atributo *quantity* del elemento item. Para ello, recomiendo que lean el código fuente de Resources.java y PluralsRules.java para entenderlo correctamente. Aún así, dejo lo fundamental de estas dos ficheros para que entiendan bien el funcionamiento:

```
----- PluralsRules.java -----
1 abstract class PluralsRules {
2
3     static final int QUANTITY_OTHER = 0x0000;
```

```

4     static final int QUANTITY_ZERO = 0x0001;
5     static final int QUANTITY_ONE  = 0x0002;
6     static final int QUANTITY_TWO  = 0x0004;
7     static final int QUANTITY_FEW  = 0x0008;
8     static final int QUANTITY_MANY = 0x0010;
9
10    static final int ID_OTHER = 0x01000004;
11
12    abstract int quantityForNumber(int n);
13
14    final int attrForNumber(int n) {
15        return PluralRules.attrForQuantity(quantityForNumber(n));
16    }
17
18    static final int attrForQuantity(int quantity) {
19        // see include/utils/ResourceTypes.h
20        switch (quantity) {
21            case QUANTITY_ZERO: return 0x01000005;
22            case QUANTITY_ONE:  return 0x01000006;
23            case QUANTITY_TWO:  return 0x01000007;
24            case QUANTITY_FEW:  return 0x01000008;
25            case QUANTITY_MANY: return 0x01000009;
26            default:            return ID_OTHER;
27        }
28    }
29
30    static final String stringForQuantity(int quantity) {
31        switch (quantity) {
32            case QUANTITY_ZERO:
33                return "zero";
34            case QUANTITY_ONE:
35                return "one";
36            case QUANTITY_TWO:
37                return "two";
38            case QUANTITY_FEW:
39                return "few";
40            case QUANTITY_MANY:
41                return "many";
42            default:
43                return "other";
44        }
45    }
46
47    static final PluralRules ruleForLocale(Locale locale) {
48        String lang = locale.getLanguage();
49        if ("cs".equals(lang)) {
50            if (cs == null) cs = new cs();
51            return cs;
52        }
53        else {
54            if (en == null) en = new en();
55            return en;
56        }
57    }
58
59    private static PluralRules cs;
60    private static class cs extends PluralRules {
61        int quantityForNumber(int n) {
62            if (n == 1) {
63                return QUANTITY_ONE;
64            }
65            else if (n >= 2 && n <= 4) {
66                return QUANTITY_FEW;
67            }
68            else {
69                return QUANTITY_OTHER;
70            }
71        }
72    }
73

```

```

74     private static PluralRules en;
75     private static class en extends PluralRules {
76         int quantityForNumber(int n) {
77             if (n == 1) {
78                 return QUANTITY_ONE;
79             }
80             else {
81                 return QUANTITY_OTHER;
82             }
83         }
84     }
85 }

```

```

Resources.java
1 public CharSequence getQuantityText(int id, int quantity)
2     throws NotFoundException {
3     PluralRules rule = getPluralRule();
4     CharSequence res =
5         mAssets.getResourceBagText(id, rule.attrForNumber(quantity));
6     if (res != null) {
7         return res;
8     }
9     res = mAssets.getResourceBagText(id, PluralRules.ID_OTHER);
10    if (res != null) {
11        return res;
12    }
13    throw new NotFoundException("Plural resource ID #0x"
14        + Integer.toHexString(id)
15        + " quantity=" + quantity
16        + " item=" + PluralRules.stringForQuantity(
17            rule.quantityForNumber(quantity)));
18 }
19
20 private PluralRules getPluralRule() {
21     synchronized (mSync) {
22         if (mPluralRule == null) {
23             mPluralRule =
24                 PluralRules.ruleForLocale(mConfiguration.locale);
25         }
26         return mPluralRule;
27     }
28 }

```

En la mayoría de los idiomas normalmente hay dos posibles valores, "one" y "other", pero para el Checo, los valores son "one" para 1, "few" del 2 al 4 y "other" para el resto.

4.8 Trabajar con recursos XML arbitrarios

Además de los recursos estructurados que hemos ido viendo, Android permite usar archivos XML arbitrarios como recursos. Esto proporciona una forma rápida de referenciar los archivos basándose en su ID de recurso así como permitirnos localizar estos archivos de una manera sencilla. Como última ventaja, nos permite compilar y almacenar estos archivos en el dispositivo eficientemente.

Los ficheros XML que se lean de esta manera, tienen que almacenarse bajo el directorio `./res/xml`. A continuación vamos a ver un ejemplo:

```

miXML.xml
1 <rootelem1>
2     <subelem1>
3         Hello World from an xml sub element
4     </subelem1>
5 </rootelem1>

```

Como hace con cualquier otro fichero XML de recursos, el AAPT compila este fichero antes de colocarlo en el paquete de la aplicación. Ahora necesitamos usar una instancia de *XmlPullParser*

para poder parsear el archivo. Para obtener la instancia de *XmlPullParser* usando el siguiente código desde cualquier contexto (incluyendo una activity):

```
Resources res = activity.getResources();
XmlResourceParser xpp = res.getXml(R.xml.test);
```

El *XmlResourceParser* devuelto es una instancia de *XmlPullParser*, y también implementa *java.util.AttributeSet*. En el siguiente fragmento de código se muestra como leer el fichero:

EjemploLeerXML.java

```
1 private String getEventsFromAnXMLFile(Context activity)
2     throws XmlPullParserException, IOException
3 {
4     StringBuffer sb = new StringBuffer();
5     Resources res = activity.getResources();
6     XmlResourceParser xpp = res.getXml(R.xml.test);
7
8     xpp.next();
9     int eventType = xpp.getEventType();
10    while (eventType != XmlPullParser.END_DOCUMENT)
11    {
12        if(eventType == XmlPullParser.START_DOCUMENT)
13        {
14            sb.append("*****Start document");
15        }
16        else if(eventType == XmlPullParser.START_TAG)
17        {
18            sb.append("\nStart tag "+xpp.getName());
19        }
20        else if(eventType == XmlPullParser.END_TAG)
21        {
22            sb.append("\nEnd tag "+xpp.getName());
23        }
24        else if(eventType == XmlPullParser.TEXT)
25        {
26            sb.append("\nText "+xpp.getText());
27        }
28        eventType = xpp.next();
29    } //eof-while
30    sb.append("\n*****End document");
31    return sb.toString();
32 } //eof-function
```

Lo que hacemos en el código de arriba es obtener el *XmlPullParser*, usarlo para navegar a través de los elementos del archivo y usar métodos adicionales de *XmlPullParser* para acceder a detalles de los elementos XML. Para ejecutar este código, se debe crear un archivo XML como el mostrado anteriormente y llamar al método *getEventsFromAnXMLFile* desde cualquier menú o botón. Devolverá un string, el cual se podrá usar para mostrarlo por el Log usando el método de debug Log.d

4.9 Trabajar con recursos RAW

Los recursos Raw se colocan bajo el directorio *./res/raw*. Son recursos raw archivos como ficheros de audio, vídeo o archivos de texto que requieran localización o ser referenciados mediante IDs de recursos.

A diferencia de los archivos XML, colocados en *./res/xml*, estos archivos no se compilan, se mueven al paquete de la aplicación tal y como son. Sin embargo, a cada fichero se le asignará un identificador en la clase R.java. Si colocamos un archivo de texto en *./res/raw/test.txt*, podremos leerlo usando el código de abajo:

EjemploArchivosRaw.java

```
1 private String getStringFromRawFile(Context activity)
2     throws IOException
3 {
```



```

4     Resources r = activity.getResources();
5     InputStream is = r.openRawResource(R.raw.test);
6     String myText = convertStreamToString(is);
7     is.close();
8     return myText;
9 }
10
11 private String convertStreamToString(InputStream is)
12     throws IOException
13 {
14     ByteArrayOutputStream baos = new ByteArrayOutputStream();
15     int i = is.read();
16     while (i != -1)
17     {
18         baos.write(i);
19         i = is.read();
20     }
21     return baos.toString();
22 }

```

Los nombres de ficheros con el mismo nombre base generan un error en el plugin ADT de eclipse.

4.10 Trabajar con recursos Assets

Android ofrece más de un directorio en el que guardar ficheros que se incluirán en el paquete.: */assets*. Está en el mismo nivel que el directorio */res*, lo que significa que no es parte de los subdirectorios del mismo. A los archivos colocados en el directorio */assets* no se les generan IDs en R.java. Somos nosotros los que debemos especificar la ruta para leerlo. La ruta al fichero es una ruta relativa que comienza con */assets*. Debemos usar la clase *AssetManager* para acceder a estos ficheros, como se muestra en el código de abajo:

EjemploAssets.java

```

1 String getStringFromAssetFile(Context activity)
2     throws IOException
3 {
4     AssetManager am = activity.getAssets();
5     InputStream is = am.open("test.txt");
6     String s = convertStreamToString(is);
7     is.close();
8     return s;
9 }

```

4.11 Estructura del directorio de recursos

En resumen, en el siguiente listado muestra la estructura global del directorio de recursos:

```

/res/values/string.xml
        /colors.xml
        /dimens.xml
        /attrs.xml
        /styles.xml
/drawable/*.png
        /*.jpg
        /*.gif
        /*.9.png
/anim/*.xml
/layout/*.xml
/raw/*.xml
/xml/*.xml
/assets/./././

```

N Debido a que no se encuentra bajo el directorio */res*, solo el directorio */assets* puede contener una lista arbitraria de directorios. Cualquier otro directorio solo puede contener ficheros en ese nivel, y no mas subdirectorios

4.12 Recursos y cambios de configuración

Los recursos ayudan a la localización. Por ejemplo, podemos tener valores para strings que cambien en función del idioma configurado en el terminal. Los recursos Android generalizan esta idea para cualquier configuración del dispositivo, el idioma es tan solo otra configuración más. Otro ejemplo de cambios de configuración se da cuando el dispositivo cambia de posición (de vertical a horizontal o viceversa). El modo vertical se suele llamar portrait y el horizontal landscape.

Android permite elegir distintas configuraciones de layout basandose en el tipo de layout. Y ambos tendrán el mismo ID de recurso. Esto se consigue usando directorios diferentes para cada configuración. Vamos a verlo con un ejemplo:

```
/res/layout/main_layout.xml  
/res/layout-port/main_layout.xml  
/res/layout-land/main_layout.xml
```

Aunque hay tres archivos layout separados, solo se generará un único ID para ellos en R.java. Este ID será de la forma *R.layout.main_layout*. Sin embargo, cuando recuperemos el layout correspondiente a este ID, obtendremos el layout apropiado para la configuración actual del dispositivo.

En el ejemplo anterior, las extensiones de los directorios *-port* y *-land* se llaman *configuration qualifiers*. Estos clasificadores (qualifiers), se tienen que separar del directorio original añadiendo un guión (-) a su nombre. Los recursos para los que especificamos estos clasificadores de configuración se llaman recursos alternativos. Los recursos almacenados en el directorio de recursos sin los clasificadores de configuración se llaman recursos por defecto.

En la siguiente lista se muestran los clasificadores de configuración disponibles:

- mccAAA: AAA es el código del país del dispositivo
- mncAAA: AAA es el código de red
- en-rUS: Idioma y región
- small, normal, large, xlarge: Tamaño de pantalla.
- long, notlong: Tipo de pantalla
- port, land: Posición vertical o horizontal (portrait y landscape)
- car, desk: Tipo de accesorio.
- night, notnight: Día o noche
- ldpi, mdpi, hdpi, xdpi, nodpi: Densidad de pantalla.
- notouch, stylus, finger: Tipo de pantalla
- keyexposed, keysoft, keyshidden: Tipo de teclado
- nokeys, qwerty, 12key: Cantidad de teclas
- navexposed, navhidden: Teclas de navegación ocultas o al descubierto
- nonav, dpad, trackball, wheel: Tipo de navegación del dispositivo (trackball es la bolita que se usa como ratón)
- v3, v4, v7: Nivel API

Con los clasificadores mostrados arriba, podemos crear directorios de recursos como los siguientes:

```
/res/layout-mcc312-mnc222-en-rUs  
/res/layout-ldpi  
/res/layout-hdpi  
/res/layout-car
```

Para saber nuestra localización actual podemos ejecutar una aplicación que viene instalada en el emulador android. La encontramos en el menú de aplicaciones y se llama Custom Locale.

Dado un ID de recurso, Android usa un algoritmo para elegir el adecuado. Si deseas saber más acerca de este tema puedes visitar la siguiente dirección <http://developer.android.com/guide/topics/resources/providing-resources.html#AlternativeResources>, pero voy a dar unas reglas básicas.

La primera regla es que los clasificadores mostrados arriba están en orden de precedencia. Veamos un ejemplo de como funciona la precedencia:

```
/res/layout/main_layout.xml
/res/layout-port/main_layout.xml
/res/layout-en/main_layout.xml
```

En este listado, el archivo *main_layout.xml* está disponible para dos variaciones posibles. Una variación para el idioma y otra para la orientación. Veamos que layout se selecciona si tenemos el dispositivo en vertical.

Incluso si el dispositivo está en vertical, Android va a elegir el layout que reside en la carpeta layout-en, ya que en la lista de clasificadores que vimos anteriormente el idioma está por delante del modo portrait (Vertical). En la dirección dada anteriormente se encuentra todo esto explicado más detalladamente.

Ahora vamos a ver algunas reglas de precedencia con unos ejemplos de recursos strings. Pero los recursos string son recursos basados en ids individuales, mientras que los ids de layout se basan en los archivos. Para probar la precedencia con recursos string, vamos a definir cinco ids para las siguientes variaciones: default, en, en_us, port, en_port. Los cinco recursos son los siguientes:

- *teststring_all*: Este ID estará en todas las variaciones del directorio values, incluyendo el default.
- *testport_port*: Estará en las variaciones default y -port
- *t1_enport*: Estará en las variaciones default, -en y -port
- *t1_1_en_port*: Estará en las variaciones default y en -en-port
- *t2*: Solo en default

El siguiente código muestra las variaciones del directorio values:

```
TestRecursos.xml
1 <resources xmlns="http://schemas.android.com/apk/res/android">
2   <string name="teststring_all">teststring in root</string>
3   <string name="t1_enport">t1 in root</string>
4   <string name="t1_1_en_port">t1_1 in root</string>
5   <string name="t2">t2 in root</string>
6   <string name="testport_port">testport in root</string>
7 </resources>
8
9 // values-en/string_en.xml
10 <resources xmlns="http://schemas.android.com/apk/res/android">
11   <string name="teststring_all">teststring-en</string>
12   <string name="t1_enport">t1_en</string>
13   <string name="t1_1_en_port">t1_1_en</string>
14 </resources>
15
16 // values-en-rUS/string_en_us.xml
17 <resources xmlns="http://schemas.android.com/apk/res/android">
18   <string name="teststring_all">test-en-us</string>
19 </resources>
20
21 // values-port/string_port.xml
22 <resources xmlns="http://schemas.android.com/apk/res/android">
23   <string name="teststring_all">test-en-us-port</string>
24   <string name="testport_port">testport-port</string>
25   <string name="t1_enport">t1_port</string>
26   <string name="t1_1_en_port">t1_1_port</string>
27 </resources>
28
29 // values/string_en_port.xml
30 <resources xmlns="http://schemas.android.com/apk/res/android">
31   <string name="teststring_all">test-en-port</string>
32   <string name="t1_1_en_port">t1_1_en_port</string>
33 </resources>
```

El listado siguiente muestra el archivo *R.java* para estos recursos:

```
public static final class string {  
    public static final int t1_1_en_port=0x7f070004;  
    public static final int t1_enport=0x7f070003;  
    public static final int t2=0x7f070005;  
    public static final int testport_port=0x7f070006;  
    public static final int teststring_all=0x7f070002;  
}
```

Como se aprecia, aunque hemos definido muchos strings, solo se han generado cinco IDs. A continuación se describe el comportamiento para cada string, que se ha probado con *en_US* y el modo *portrait*:

- *teststring_all*: Este ID aparece en las cinco variaciones, ya que aparece en todas. En nuestra configuración, se escogerá del directorio *values-en-rUS*. Basandonos en las reglas de precedencia, tener un directorio concreto para un idioma va por delante de las variaciones *en*, *port* y *en-port*
- *testport_port*: Este ID está en las variaciones *default* y *-port*. Ya que no hay ningún directorio que no se encuentra en ningún directorio que empiece por *-en*, la variación *-port* tendrá precedencia frente a *default*, por lo tanto se escogerá el valor desde la variación *-port*. Si estuviera en en una de las variaciones *-en*, sería esta última la que se escogería.
- *t1_enport*: Este ID está en tres variaciones, *default*, *-en* y *-port*. Debido a que se encuentra en *-en* y *-port* al mismo tiempo, se escogerá el valor de *-en*
- *t1_1_en_port*: Este ID se encuentra en cuatro variaciones, *default*, *-port*, *-en* y *-en-port*. El que toma precedencia frente a todos aquí es *-en-port*
- *t2*: Este ID se encuentra solo en *default*, por lo tanto se elige el valor de ahí.

El SDK de Android tiene algoritmos más detallados aún, sin embargo, con este ejemplo se ha mostrado lo esencial. La clave está en darse cuenta de la precedencia de cada variación sobre otra. A continuación, para aquellos que deseen ampliar información dejo unas URLs de referencia:



5 — StrictMode

5.1 Introducción

Android 2.3 introdujo una funcionalidad de depuración llamada *StrictMode*. Según Google, usaron esta característica para hacer cientos de mejoras a sus aplicaciones Android. Lo que hace el StrictMode es informar de las violaciones de políticas relacionadas con los hilos y la máquina virtual. Si se detecta dicha violación, obtenemos una alerta que nos lo indica. Junto a la alerta tendremos también una traza de la pila de ejecución (Stack Trace), donde podremos comprobar el lugar en el que se produjo la violación. En ese momento, podemos forzar el cierre de la aplicación o simplemente escribirla en el log y dejar que la aplicación continúe su ejecución.

Actualmente hay dos tipos de políticas disponibles para usar con StrictMode. La primera de ellas es referente a los hilos y está destinada principalmente a ejecutarse en el hilo principal, también conocido como el hilo de UI (User Interface).

No es una buena práctica hacer lecturas/escrituras a disco desde el hilo principal, como tampoco lo es realizar accesos a red. Google ha añadido al código del disco y de red hooks o ganchos, que son algoritmos abstractos que invocan a métodos abstractos. Por lo tanto, si activamos StrictMode para uno de nuestros hilos, y ese hilo realiza cualquiera de las dos acciones mencionadas anteriormente, seremos informados. Podemos elegir sobre qué aspectos de la política de hilos queremos ser informados, así como el método por el cual se nos informará.

Normalmente las que usaremos serán accesos a disco y red. En cuanto al método por el que seremos informados, podemos elegir: Escribirlo en el LogCat, mostrar un diálogo, hacer un destello en la pantalla, escribir en el archivo log de DropBox o forzar el cierre de la aplicación. Normalmente se usa el LogCat o forzar el cierre. A continuación vemos un ejemplo de como configurar StrictMode para políticas de hilos:

```
StrictMode.setThreadPolicy(new StrictMode.ThreadPolicy.Builder()
    .detectDiskReads()
    .detectDiskWrites()
    .detectNetwork()// or .detectAll() for all detectable problems
    .penaltyLog()
    .build());
```

Podemos ver que la clase Builder hace que configurar StrictMode sea bastante fácil. Todos los métodos que definen las políticas devuelven una referencia al objeto Builder. El último, *build()*, devuelve un objeto *ThreadPolicy*, que es el argumento que *setThreadPolicy()* espera recibir. *setThreadPolicy()* es un método estático, por ello no es necesario instanciar un objeto de tipo *StrictMode*.

En el ejemplo anterior, se ha configurado la política para que avise de lecturas-escrituras a disco, acceso a red y vamos a ser informados a través del logCat. Podemos usar *detectAll()* para ahorrarnos escribir los otros métodos de detección. También podemos usar el método *penaltyDeath()* para forzar el cierre de la aplicación una vez escrita la alerta StrictMode al LogCat.

Debido a que con el código de arriba hemos activado el StrictMode, no necesitamos seguir activándolo. Por lo tanto, podemos habilitar StrictMode al principio del método *onCreate()* de nuestra actividad principal, ya que se ejecuta en el hilo principal, avisándonos de todo lo que pase en ese hilo.

StrictMode también dispone de *VmPolicy*, que comprueba pérdidas de memoria si un objeto SQLite finaliza antes de que haya sido cerrado, o si cualquier objeto que pueda ser cerrado ha finalizado antes de ser cerrado. Las *VmPolicy* se crean de una forma similar como se muestra a continuación. Sin embargo, hay una diferencia entre éstas y *ThreadPolicy*, que no pueden usar alertas a través de diálogos.

```
StrictMode.setVmPolicy(new StrictMode.VmPolicy.Builder()
    .detectLeakedSqlLiteObjects()
    .penaltyLog()
    .penaltyDeath()
    .build());
```

Ya que la configuración ocurre en un hilo, quizás nos sorprenda que el código se está ejecutando en el hilo principal, pero la traza de la pila (Stack Trace) está ahí para ayudarnos a averiguar en qué punto se produjo la violación. Una vez detectada la violación, podemos resolverla moviendo el código que la produce a un hilo en segundo plano propio. O por otra parte podemos decidir que el código está bien en esa parte y no moverlo.

Algo importante que debemos hacer a la hora de distribuir nuestra aplicación es apagar el StrictMode. Hay varias formas de conseguir esto. La más sencilla es eliminar las llamadas a los métodos, pero hacer esto dificulta seguir desarrollando la aplicación. Podemos declarar un booleano al nivel de la aplicación y comprobar su valor antes de llamar a StrictMode. De este modo, al enviar la aplicación a producción, simplemente daríamos a esta variable el valor false y de se llamará nunca a StrictMode.

Un método más elegante para resolver este problema, es usar el atributo *debuggable* en nuestro *AndroidManifest*. Este atributo se coloca en el tag *<application>* de la forma *android:debuggable*. Una vez activado este atributo, puede fijarse como verdadero o falso dependiendo de si queremos depurar la aplicación o no. Podemos comprobar el estado de este atributo como se muestra más abajo. De modo que cuando esté activado, tendremos StrictMode activo, y cuando no lo esté, no.

```
//Devuelve si la aplicacion esta en modo debug o no
ApplicationInfo appInfo = context.getApplicationInfo();
int appFlags = appInfo.flags;
if ((appFlags & ApplicationInfo.FLAG_DEBUGGABLE) != 0) {
    //Aqui configurariamos el StrictMode
}
```

Si usamos eclipse como IDE, el plugin ADT configura el atributo *debuggable* automáticamente. Es decir, cuando usamos el emulador o un dispositivo real, eclipse fija el atributo *debuggable* a verdadero, lo que nos permite usar StrictMode y depurar nuestra aplicación. Cuando exportamos la aplicación para crear una versión de producción, eclipse lo fija a falso. Hay que tener cuidado con esto, ya que si nosotros añadimos el atributo a mano, eclipse no lo cambiará.

StrictMode no funciona en versiones Android anteriores a la 2.3. Si queremos usarlo con versiones anteriores, podemos usar técnicas espejo para llamar indirectamente a los métodos de StrictMode:

```
try{
    Class strictMode = Class.forName("android.os.StrictMode");
    Method enableDefaults = strictMode.getMethod("enableDefaults");
    enableDefaults.invoke(null);

} catch (Exception e){
    //Este dispositivo no soporta StrictMode
    Log.v("StrictMode", "no soportado, ignorando...");
}
```

El código de arriba determina si la clase *StrictMode* existe, y si existe, llama a *enableDefaults()*. En caso de no existir la aplicación no finalizará, puesto que hemos tratado la excepción y el bloque catch se invocará con una excepción del tipo *ClassNotFoundException*.

Si el *StrictMode* no está disponible para nuestra aplicación, se lanzará un error del tipo *VerifyError* al intentar acceder a él. Si envolvemos a *StrictMode* en una clase y capturamos el error, lo podremos ignorar si *StrictMode* no está habilitado. A continuación vamos a ver un ejemplo creando esta clase.

```
StrictModeWrapper.java
1 import android.content.Context;
2 import android.content.pm.ApplicationInfo;
3 import android.os.StrictMode;
4
5 public class StrictModeWrapper{
6     public static void init(Context context){
7         ApplicationInfo appInfo = context.getApplicationInfo();
8         int appFlags = appInfo.flags;
9         if ((appFlags & ApplicationInfo.FLAG_DEBUGGABLE) != 0){
10             StrictMode.setThreadPolicy(new StrictMode.ThreadPolicy.Builder()
11                 .detectDiskReads()
12                 .detectDiskWrites()
13                 .detectNetwork()
14                 .penaltyLog()
15                 .build());
16             StrictMode.setVmPolicy(new StrictMode.VmPolicy.Builder()
17                 .detectLeakedSqlLiteObjects()
18                 .penaltyLog()
19                 .penaltyDeath()
20                 .build());
21         }
22     }
23 }
```

Como se puede apreciar, simplemente hemos metido todos los ejemplos que vimos anteriormente en una clase. Ahora para configurar *StrictMode* tenemos que hacer lo siguiente:

```
try{
    StrictModeWrapper.init(this);

} catch (Throwable throwable){
    Log.v("StrictMode", "no soportado, ignorando...");
}
```

this es el contexto local del objeto desde el que llamemos al método *init*, por ejemplo podría ser desde el método *onCreate()* de nuestra actividad principal.

5.1.1 Ejemplo de uso

Time	pid	tag	Message
11-16 12:36:35 D 454	StrictMo		at android.os.Looper.loop(Looper.java:123)
11-16 12:36:35 D 454	StrictMo		at android.app.ActivityThread.main(ActivityThread.java:3683)
11-16 12:36:35 D 454	StrictMo		at java.lang.reflect.Method.invokeNative(Native Method)
11-16 12:36:35 D 454	StrictMo		at java.lang.reflect.Method.invoke(Method.java:507)
11-16 12:36:35 D 454	StrictMo		at com.android.internal.os.ZygoteInit\$MethodAndArgsCaller.run(ZygoteInit.java:839)
11-16 12:36:35 D 454	StrictMo		at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:597)
11-16 12:36:35 D 454	StrictMo		at dalvik.system.NativeStart.main(Native Method)
11-16 12:37:21 D 454	StrictMo		StrictMode policy violation; ~duration=1106 ms: android.os.StrictMode\$StrictModeDiskReadViolation: policy=55 violat
11-16 12:37:21 D 454	StrictMo		at android.os.StrictMode\$AndroidBlockGuardPolicy.onReadFromDisk(StrictMode.java:745)
11-16 12:37:21 D 454	StrictMo		at dalvik.system.BlockGuard\$WrappedFilesystem.open(BlockGuard.java:228)
11-16 12:37:21 D 454	StrictMo		at java.io.FileInputStream.<init>(FileInputStream.java:80)
11-16 12:37:21 D 454	StrictMo		at android.app.ContextImpl.getSharedPreferences(ContextImpl.java:375)
11-16 12:37:21 D 454	StrictMo		at android.content.ContextWrapper.getSharedPreferences(ContextWrapper.java:146)
11-16 12:37:21 D 454	StrictMo		at android.app.Activity.getSharedPreferences(Activity.java:3522)
11-16 12:37:21 D 454	StrictMo		at es.masterd.lector_rss.activities.TitularesActivity.cargaNoticias(TitularesActivity.java:180)
11-16 12:37:21 D 454	StrictMo		at es.masterd.lector_rss.activities.TitularesActivity.onResume(TitularesActivity.java:64)
11-16 12:37:21 D 454	StrictMo		at android.app.Instrumentation.callActivityOnResume(Instrumentation.java:1150)
11-16 12:37:21 D 454	StrictMo		at android.app.Activity.performResume(Activity.java:3832)
11-16 12:37:21 D 454	StrictMo		at android.app.ActivityThread.performResumeActivity(ActivityThread.java:2110)
11-16 12:37:21 D 454	StrictMo		at android.app.ActivityThread.handleResumeActivity(ActivityThread.java:2135)
11-16 12:37:21 D 454	StrictMo		at android.app.ActivityThread.handleLaunchActivity(ActivityThread.java:1668)
11-16 12:37:21 D 454	StrictMo		at android.app.ActivityThread.access\$1500(ActivityThread.java:117)
11-16 12:37:21 D 454	StrictMo		at android.app.ActivityThread\$H.handleMessage(ActivityThread.java:931)
11-16 12:37:21 D 454	StrictMo		at android.os.Handler.dispatchMessage(Handler.java:99)
11-16 12:37:21 D 454	StrictMo		at android.os.Looper.loop(Looper.java:123)
11-16 12:37:21 D 454	StrictMo		at android.app.ActivityThread.main(ActivityThread.java:3683)
11-16 12:37:21 D 454	StrictMo		at java.lang.reflect.Method.invokeNative(Native Method)
11-16 12:37:21 D 454	StrictMo		at java.lang.reflect.Method.invoke(Method.java:507)
11-16 12:37:21 D 454	StrictMo		at com.android.internal.os.ZygoteInit\$MethodAndArgsCaller.run(ZygoteInit.java:839)
11-16 12:37:21 D 454	StrictMo		at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:597)
11-16 12:37:21 D 454	StrictMo		at dalvik.system.NativeStart.main(Native Method)

Figura 5.1: StrictMode Logcat

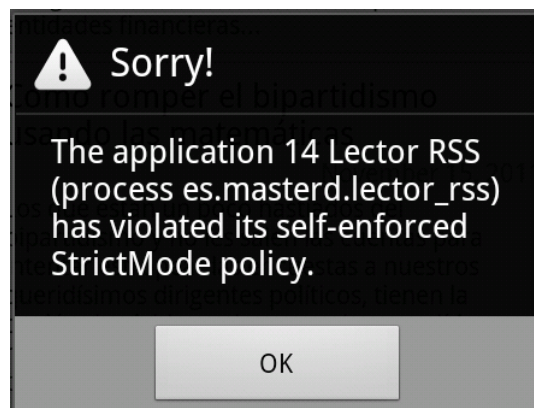


Figura 5.2: StrictMode Dialog



Bibliografía

Libros

[lbr11] Sayed Ibrahim Hashimi. *Pro Android 3*. 1.^a edición. Volumen 1. 1. City: APress, abr. de 2011, página 1200 (véase página 5).

Artículos

[Pro1] El Baúl del Programador. *Curso Android*. <http://elbauldelprogramador.com/p/guia-de-desarrollo-android.html>. Blog. 1 (véase página 5).



Índice alfabético

A

Actividades y tareas	11
Adaptadores	26
AdapterView	26
alwaysRetainTaskState	12
android:debuggable	62
android:orientation	21
android:theme	42
AndroidManifest	11
ArrayAdapter	26
Arrays de strings	51
AssetManager	56
Assets	56
AT_MOST	20

B

BaseAdapter	31
build()	61
Button	23, 49

C

centerInParent	23
CheckBox	25
ciclo de vida Actividades	12
Ciclo de vida de los componentes	12
ClassNotFoundException	63
clearTaskOnLaunch	12
Componentes	5
Componentes de las aplicaciones	9

Conceptos básicos	9
Context	20
Context Menu	35
Creando el proyecto	5

D

detectAll()	62
Diálogos	37, 39
dismissDialog(int)	40
draw()	19

E

EditText	24
Ejemplo 1: Pasar parámetros entre Actividades	15
enableDefaults()	63
Estilos	41
EXACTLY	20

F

findViewById()	49
finishOnTaskLaunch	12
FrameLayout	20
Fundamentos	9

G

getApplication()	20
getApplicationContext()	20, 37

getContext() 20
 getCount 32
 getEventsFromAnXMLFile 55
 getItem 32
 getItemId 32
 getQuantityString() 52
 getView() 32

I

ImageView 25
 Intents 11
 Introducción 5

J

java.util.AttributeSet 55

L

layout 20, 51
 layout() 19
 layout_below 23
 LayoutInflater 32
 LayoutParams 20
 LinearLayout 21, 49
 ListActivity 28
 Listeners 23
 ListView 27

M

MATCH_PARENT 20
 measure() 19
 Menús 33
 MenuInflater.inflate() 34

N

Notificaciones 37, 39

O

onCreate() 13, 63
 onCreateContextMenu() 35
 onCreateDialog(int) 39
 onCreateOptionsMenu() 34
 onDestroy() 14

onKeyListener 25
 onOptionsItemSelected() 34
 onPause() 13
 onPrepareDialog(int) 39
 onSaveInstanceState() 14
 onStop() 14, 15
 Options Menu 34

P

penaltyDeath() 62
 Plurales 52
 Procesos 15
 Procesos e Hilos 12

Q

qualifiers 57
 quantity 52

R

RAW 55
 receiver 12
 Recursos 45
 Recursos Layout 48
 RelativeLayout 22
 removeDialog(int) 40

S

setContentView() 19
 setOnClickListener() 23
 setOnItemClickListener() 23
 setOnKeyListener() 23
 setThreadPolicy() 61
 showDialog(int) 40
 simple_list_item_1 28
 StrictMode 61
 string 47, 51

T

Temas 41
 TextView 49
 TextView 23
 ThreadPolicy 61
 Tipos de Layouts 20
 Toast 37

Toast.LENGTH_LONG 37
Toast.LENGTH_SHORT 37

U

UNSPECIFIED 20

V

VerifyError 63
View 19, 20
View.MeasureSpec 20
ViewGroup 20
VmPolicy 62

W

WRAP_CONTENT 20

X

XmlPullParser 54, 55
XmlResourceParser 55