Apunte Android
Laboratorio de Computación V
UTN-FRA

Autores:
Matías Ramos
Ernesto Gigliotti

Año: 2016

Revisión: v11

Clase 1

1. Qué es Android, conceptos generales
2. Arquitectura del sistema operativo
3. Versiones de Android.
4. Instalación de Android Studio
5. Instalación SDK de Android.
6. Crear proyecto Android
7. Archivos de un proyecto Android, estructura del proyecto.

Clase 2

1. Activities.
2. Android Manifest, permisos.
3. Archivos xml de layout.
4. Layout horizontal y vertical
5. RelativeLayout
6. TextView, Edittext
7. FindviewById
8. Botones , events y listeners

Clase 3

1. Componentes de una aplicación.
2. Concepto Model View Controller.
3. Activities: Ciclo de vida de una aplicación.
4. Internacionalización y Localización
5. Resources: Strings, Colors y Pictures.

Clase 4

1. RecycleView: lista de items
2. Adapter: Inflate desde archivo xml de layout
3. Eventos

Clase 5

1. Intents
2. Threads
3. HttpURLConnection: obteniendo una imagen y un texto desde internet.

Clase 6

1. XML. Sintaxis
2. XML Parsers

Clase 7

1. Menú
2. Toolbar
3. WebView

Clase 8

1. Diálogos
2. Shared Preferences
3. Selectors y Shapes

Clase 9

1. Sockets y Streams
2. Conexiones TCP

Clase 10

1. Navigation Drawer

Fragments

2.

Clase 1 Introducción

1.1 Qué es Android, conceptos generales

Android es un sistema operativo basado en en el kernel de Linux diseñado originalmente para dispositivos móviles, tales como teléfonos y tablets, pero que actualmente se encuentra en desarrollo para usarse en relojes de pulsera, electrodomésticos y autos. Fue desarrollado inicialmente por Android Inc., una firma comprada por Google en 2005.

Es el principal producto de la Open Handset Alliance, un conglomerado de fabricantes y desarrolladores de hardware, software y operadores de servicio.

Android tiene una gran comunidad de desarrolladores escribiendo aplicaciones para extender la funcionalidad de los dispositivos. Para 2014 existían más de 1.5 millones aplicaciones disponibles para Android.

Play Store es la tienda de aplicaciones en línea administrada por Google, aunque existe la posibilidad de obtener software de otras páginas no oficiales. Los programas están escritos en el lenguaje de programación Java.

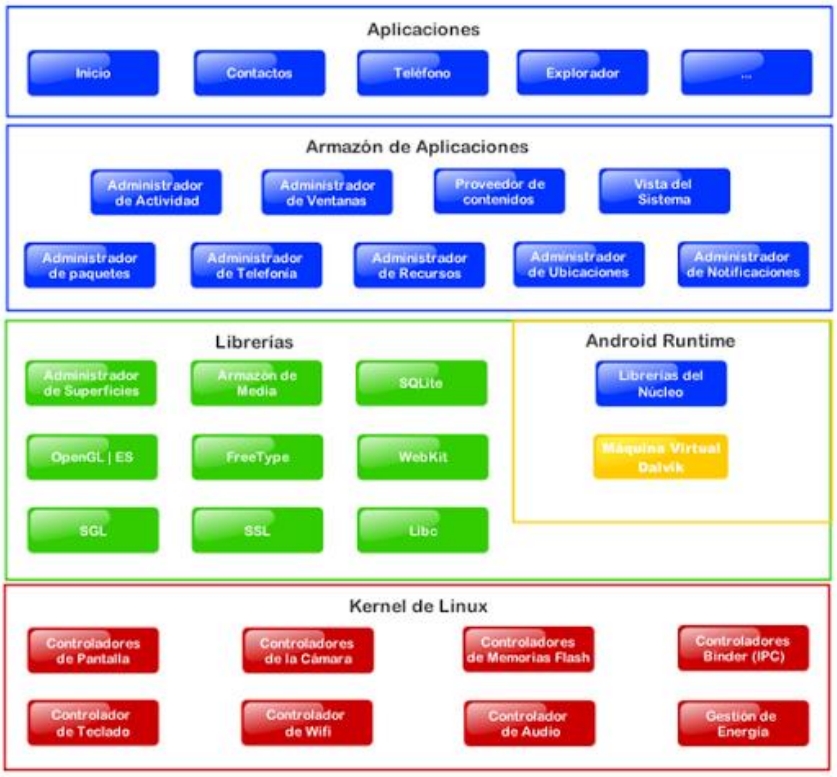
La estructura del sistema operativo Android se compone de aplicaciones que se ejecutan en un framework Java, sobre un núcleo de las bibliotecas Java en una máquina virtual ART (Android RunTime, a partir de la versión L) con una pre-compilación al instalar la aplicación. Las bibliotecas escritas en lenguaje C incluyen un administrador de interfaz gráfica (surface manager), un framework OpenCore, una base de datos relacional SQLite, una API gráfica OpenGL ES, un motor de renderizado WebKit, un motor gráfico SGL, SSL y una biblioteca estándar de C.

1.2 Arquitectura del sistema operativo

Como ya se ha mencionado, Android es una plataforma para dispositivos móviles que contiene una gran cantidad de software donde se incluye un sistema operativo, middleware y aplicaciones básicas para el usuario.

En las siguientes líneas se dará una visión global por capas de cuál es la arquitectura empleada en Android.

Cada una de estas capas utiliza servicios ofrecidos por las anteriores, y ofrece a su vez los suyos propios a las capas de niveles superiores, tal como muestra la siguiente figura ((c) Google):



- Aplicaciones: Este nivel contiene, tanto las incluidas por defecto de Android como aquellas que el usuario vaya añadiendo posteriormente, ya sean de terceras empresas o de su propio desarrollo. Todas estas aplicaciones utilizan los servicios, las API y bibliotecas de los niveles anteriores.

- Framework de Aplicaciones: Representa fundamentalmente el conjunto de herramientas de desarrollo de cualquier aplicación. Toda aplicación que se desarrolle para Android, ya sean las propias del dispositivo, las desarrolladas por Google o terceras compañías, o incluso las que el propio usuario cree, utilizan el mismo conjunto de API y el mismo "framework", representado por este nivel.

  Entre las API más importantes ubicadas aquí, se pueden encontrar las siguientes:

  - Activity Manager: Conjunto de API que gestiona el ciclo de vida de las aplicaciones en Android.
  - Window Manager: Gestiona las ventanas de las aplicaciones y utiliza la librería Surface Manager.
  - Telephone Manager: Incluye todas las API vinculadas a las funcionalidades propias del teléfono (llamadas, mensajes, etc.).

- Content Provider: Permite a cualquier aplicación compartir sus datos con las demás aplicaciones de Android. Por ejemplo, gracias a esta API la información de contactos, agenda, mensajes, etc. será accesible para otras aplicaciones.
- View System: Proporciona un gran número de elementos para poder construir interfaces de usuario (GUI), como listas, mosaicos, botones, "check-boxes", tamaño de ventanas, control de las interfaces mediante teclado, etc. Incluye también algunas vistas estándar para las funcionalidades más frecuentes.
- Location Manager: Posibilita a las aplicaciones la obtención de información de localización y posicionamiento.
- Notification Manager: Mediante el cual las aplicaciones, usando un mismo formato, comunican al usuario eventos que ocurran durante su ejecución: una llamada entrante, un mensaje recibido, conexión Wi-Fi disponible, ubicación en un punto determinado, etc. Si llevan asociada alguna acción, en Android denominada Intent, (por ejemplo, atender una llamada recibida) ésta se activa mediante un simple clic.
- XMPP Service: Colección de API para utilizar este protocolo de intercambio de mensajes basado en XML.

- Bibliotecas: La siguiente capa se corresponde con las bibliotecas utilizadas por Android. Éstas han sido escritas utilizando C/C++ y proporcionan a Android la mayor parte de sus capacidades más características. Junto al núcleo basado en Linux, estas bibliotecas constituyen el corazón de Android. Entre las más importantes ubicadas aquí, se pueden encontrar las siguientes:

    - Biblioteca libc: Incluye todas las cabeceras y funciones según el estándar del lenguaje C. Todas las demás bibliotecas se definen en este lenguaje.
    - Surface Manager: Es la encargada de componer los diferentes elementos de navegación de pantalla. Gestiona también las ventanas pertenecientes a las distintas aplicaciones activas en cada momento.
    - OpenGL/SL y SGL: Representan las librerías gráficas y, por tanto, sustentan la capacidad gráfica de Android. OpenGL/SL maneja gráficos en 3D y permite utilizar, en caso de que esté disponible en el propio dispositivo móvil, el hardware encargado de proporcionar gráficos 3D. Por otro lado, SGL proporciona gráficos en 2D, por lo que será la librería más habitualmente utilizada por la mayoría de las aplicaciones. Una característica importante de la capacidad gráfica de Android es que es posible desarrollar aplicaciones que combinen gráficos en 3D y 2D.
    - Media Libraries: Proporciona todos los códecs necesarios para el contenido multimedia soportado en Android (vídeo, audio, imágenes estáticas y animadas, etc.)
    - FreeType: Permite trabajar de forma rápida y sencilla con distintos tipos de fuentes.
    - Biblioteca SSL: Posibilita la utilización de dicho protocolo para establecer comunicaciones seguras.

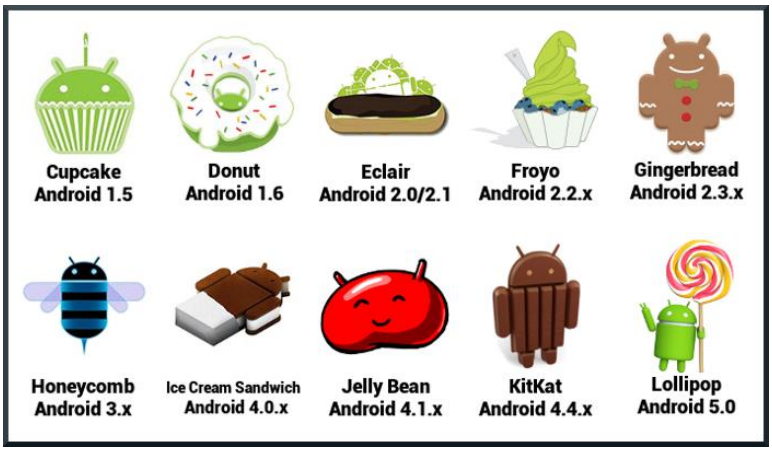
    - SQLite: Creación y gestión de bases de datos relacionales.
    - WebKit: Proporciona un motor para las aplicaciones de tipo navegador y formaba el núcleo del actual navegador incluido por defecto en la plataforma Android, en las nuevas versiones se incluye Chrome.

- Tiempo de ejecución de Android: Al mismo nivel que las bibliotecas de Android se sitúa el entorno de ejecución. Éste lo constituyen las Core Libraries, que son bibliotecas con una gran cantidad de clases Java y la máquina virtual ART.
  ART es el nombre de la máquina virtual que utiliza Android (desde la versión L, antes se utilizaba DalvikVM, la cual está basada en registros, es un intérprete que sólo ejecuta los archivos ejecutables con formato .dex (Dalvik Executable)).
  La principal diferencia entre la antigua Dalvik y la nueva ART, reside en que la antigua ejecuta una máquina virtual interpretando el código al tiempo que se inicia la aplicación. En cambio, ART es AOT (Ahead-Of-Time), es decir, se lanza una pre-compilación al instalar la aplicación y, por tanto, al ejecutarla esta ya no requiere tanta carga de datos como antes y hace que iniciar una aplicación se produzca en menos tiempo, y en algunos casos se reduce el tiempo de inicio y ejecución de una aplicación a la mitad.
  Tenemos algunas desventajas como el tiempo de instalación que se verá aumentado y las aplicaciones ocuparán más espacio.

- Kernel Linux: Android utiliza un núcleo de Linux como una capa de abstracción para el hardware disponible en los dispositivos móviles. Esta capa contiene los drivers necesarios para que cualquier componente hardware pueda ser utilizado mediante las llamadas correspondientes. Siempre que un fabricante incluye un nuevo elemento de hardware, lo primero que se debe realizar para que pueda ser utilizado desde Android es crear las bibliotecas de control o drivers necesarios dentro de este kernel de Linux embebido en el propio dispositivo.

1.3 Versiones Android

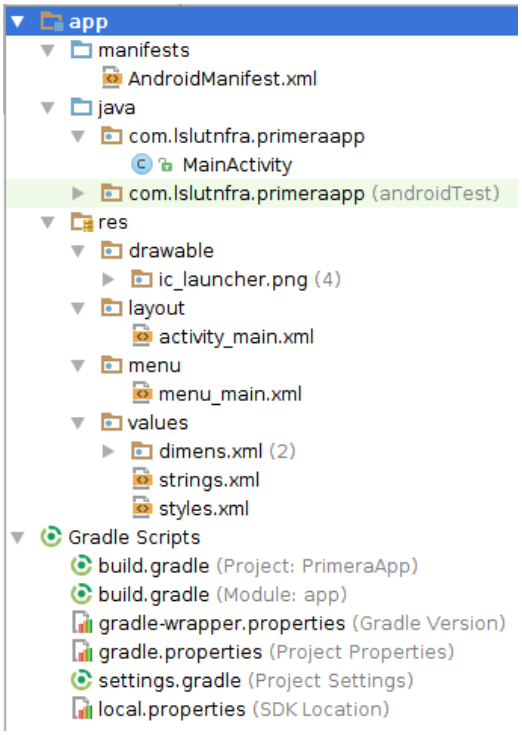
Las versiones de Android reciben, en inglés, el nombre de diferentes dulces. En cada versión el dulce elegido empieza por una letra distinta, conforme a un orden alfabético:



El 87,9% de los dispositivos Android corren 4.0+ (datos de la GDC 2015).

1.7 Estructura del proyecto.

Si nos ubicamos en el panel de la parte izquierda llamado "Project", el cual representa la estructura de tu proyecto. Vemos que están las carpetas .idea, app, build, gradle y otros archivos de configuración.
La carpeta app es la que contiene todo lo relacionado con tu proyecto, es la carpeta que nos interesa por el momento y donde incluiremos los archivos necesarios para que nuestra aplicación sea empaquetada.



Dentro de la carpeta **java** encontraremos los packages de nuestro código Java.

Dentro de la carpeta **res** encontraremos todos aquellos recursos para nuestra aplicación.

Uno de los recursos más relevantes es el archivo **strings.xml** que se encuentra dentro de la subcarpeta **values**. Este fichero almacena todas los textos que se muestran en los form widgets (controles, formas, botones, vistas, etc) de nuestras actividades.

En la carpeta **layout** se encuentran los archivos XML que describen las pantallas de nuestra aplicación

En **drawable** se encuentran los recursos de imágenes, mientras que en **menú** se encuentran archivos XML que representan los ítems del menú que capa pantalla puede tener.

En la carpeta **manifests**, se encuentra el archivo **AndroidManifest**, el cual es muy importante ya que posee metadata de la aplicación, como qué versiones soporta y los permisos que nuestra aplicación requiere.

A partir de Android 4.3 se incorpora al proyecto una segunda carpeta de recursos para almacenar imágenes llamada **mipmap**, en la cual se almacenará el ícono de nuestra aplicación. La diferencia con drawable es que dentro de mipmap podemos proveer al sistema una imagen con mucha más resolución de la requerida para el resto de las imágenes, ideal para una mayor calidad durante animaciones que cambian el tamaño de la imagen, o para lograr mostrar un ícono en el menú de aplicaciones, más grande que el predeterminado, por esta razón el ícono de nuestra aplicación debe ir en esta carpeta y no en drawable como se ubicaba antiguamente.

2.1 Activity

Las aplicaciones que tengan interfaces gráficas deberán tener al menos una clase del tipo Activity, ya que ésta actúa como lo que comúnmente se conoce como "form". En una Activity se colocan los elementos de la interfaz gráfica.

En nuestra subclase de Activity necesitamos implementar algunos métodos

- onCreate( ): Se ejecuta cuando la actividad es creada por primera vez.
- onStart( ): Se ejecuta cuando la actividad es visible al usuario.
- onResume( ): Se ejecuta cuando la actividad empieza a interactuar con el usuario.
- onPause( ): Se ejecuta cuando la actividad actual es pausada y otra actividad previa es desplegada en la pantalla.
- onStop( ): Se ejecuta cuando la actividad ya no es visible para el usuario.
- onRestart( ): Se ejecuta cuando la actividad ha sido detenida y esta reiniciando.

2.2 Android Manifest, permisos.

Este XML se genera automáticamente al crear un proyecto y en él se declaran todas las especificaciones de nuestra aplicación. Cuando hablamos de especificaciones hacemos mención a las Activities utilizadas, los Intents, bibliotecas, el nombre de la aplicación, el hardware que se necesitará, los permisos de la aplicación, etcétera.

- <manifest>: Dentro de este tag encontramos los siguientes atributos.
    - android:versionCode: Hace referencia al número de versión de desarrollo de nuestro programa, cada versión final que se desea publicar debe tener un versionCode distinto.
    - android:versionName: Este es el número de versión de nuestro programa.
    - package: Es el paquete de nuestro programa y con el cual se referencia nuestra aplicación en el Android Market y el teléfono.
- <application>: Aquí adentro van todas las Activities, Services, Providers, Receivers y las bibliotecas que se usan en nuestra aplicación.
- <activity> Se tendrá un activity por cada pantalla de nuestra aplicación y dentro de él van sus atributos y los distintos Intents para comunicarse.
    - android:name = ".UnActivity": Este es el nombre de la clase de nuestra Activity.
    - android:label = "@string/app_name": Es el texto que aparecerá en la barra superior de nuestra Activity.
    - android:theme = "@style/Theme.NoBackground": En Android uno puede crear Themes (conjuntos de estilos) para reutilizar en una aplicación y de esa manera poder mantener la coherencia en el diseño de la aplicación.
    - android:configChanges = "orientation|keyboardHidden": Este atributo establece que la Activity no detectará los cambios de orientación y por ocultación del teclado físico.

    - android:screenOrientation = "portrait": Determinando esta propiedad le decimos a nuestra Activity que sólo tendrá disposición en modo portrait (el teléfono puesto verticalmente y recordemos que "landscape" significa el teléfono puesto horizontalmente).

- <supports-screens>: Este tag es utilizado para describir las pantallas soportadas por nuestra aplicación. Entre los atributos podemos encontrar:
    - android:largeScreens = "false": Especifica si se van a soportar pantallas del tipo large(tabs, netbooks, etc.) con nuestra aplicación.
    - android:normalScreens = "true": Realiza lo mismo que el largeScreens pero haciendo referencia a las pantallas normal, prácticamente todos los celulares del tamaño similar al T-Mobile G1.
    - android:smallScreens = "false": Lo mismo que los anteriores pero para dispositivos con pantalla pequeña.
    - android:anyDensity = "true": Este atributo sirve para determinar que nosotros nos encargaremos de escalar las pantallas para las distintas densidades y que el teléfono no tendrá que hacerse cargo de adaptarlas automáticamente.

- <uses-permissions>: Mediante este Tag especificamos los permisos que va a necesitar nuestra aplicación para poder ejecutarse, además son los que deberá aceptar el usuario antes de instalarla. Por ejemplo, si se desea utilizar funcionalidades con Internet o el vibrador del teléfono, hay que indicar que nuestra aplicación requiere esos permisos.
    - android.permission.INTERNET: En caso de que nuestra aplicación desee accesar a Internet
    - android.permission.ACCESS_WIFI_STATE: Le permite a nuestras aplicaciones accesar a la información de las conexiones WI-FI.
    - android.permission.READ_CALENDAR, READ_CONTACTS: Y todos los permisos con el prefijo READ_ le permiten a nuestras aplicaciones leer la información de los content providers incorporados en Android.
    - android.permission.WRITE_CALENDAR: y todos los permisos con el prefijo WRITE_ le permiten a nuestras aplicaciones modificar la información de los content-providers incorporados en Android.

- <uses-sdk>: En este tag determinamos las distintas versiones Android que va a utilizar nuestra aplicación, tanto sobre qué versiones va a correr como qué versión fue utilizada para realizar nuestras pruebas. Mediante el atributo android:minSdkVersion establecemos a partir de qué versión de Android nuestra aplicación podrá correr.

Antes, una vez declarados los permiso en el Manifest, se aceptaban a la hora de instalar la aplicación. No existía la posibilidad de rechazar algun permiso y no otros. Desde la API 23, Android 6 M. Se pueden aceptar o rechazar los permisos individualmente.

2.3 Layout

Los layouts son elementos no visuales destinados a controlar la distribución, posición y dimensiones de los controles que se insertan en su interior. Estos componentes extienden a la clase base ViewGroup, como muchos otros componentes contenedores, es decir, capaces de contener a otros controles.

2.4 Layouts horizontal y vertical

Organiza sus objetos hijos (views) horizontal o verticalmente, según el valor de la propiedad "orientation".

Definicion en archivo xml de layout:

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

Las Views se apilan una debajo de la otra:

Parámetros de las Views contenidas

Los siguientes parámetros permiten ubicar con más precisión las views y contenedores en la pantalla:

- layout_width
- layout_height
- gravity
- layout_gravity

layout_width: Determina el ancho de la view (o contenedor), puede tomar dos valores posibles:

- **match_parent:** La view se extiende hasta llenar el espacio que ofrece el contenedor de la cual es hija.
- **wrap_content:** La view solo ocupa el espacio mínimo que necesita dentro del contenedor de la cual es hija.
- También puede setearse el ancho en px o dp.

layout_height : Determina el alto de la view (o contenedor), puede tomar los mismos valores que layout_width

Ejemplo:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >
    <Button
        android:id="@+id/button1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Button" />
</LinearLayout>
```

gravity: Es un parámetro usado por la View para alinear algo de su contenido (por ejemplo texto en un TextView o EditText). Los valores posibles son:

- **left** : a la izquierda (moviéndose en forma horizontal)
- **rigth**: a la derecha (moviéndose en forma horizontal)
- **center**: centrado (horizontal y vertical)
- **top**: arriba (moviéndose en forma vertical)
- **botton**: abajo (moviéndose en forma vertical)
- **center_vertical**: centrado (moviéndose en forma vertical)
- **center_horizontal**: centrado (moviéndose en forma horizontal)

Ejemplo con gravity seteado en alguno de los valores que se detallaron anteriormente:

```
<EditText
    android:id="@+id/editText1"
    android:layout_width="match_parent"
    android:layout_weight="0.75"
    android:text="hola"
    android:gravity="bottom"
    android:layout_height="wrap_content" />

<EditText
    android:id="@+id/editText2"
    android:layout_width="match_parent"
           android:layout_weight="0.25"
           android:text="hola"
           android:gravity="center"
    android:layout_height="wrap_content" />

<EditText
    android:id="@+id/editText3"
    android:layout_width="match_parent"
    android:layout_weight="0.0"
    android:text="hola"
    android:gravity="right"
    android:layout_height="wrap_content" />
```

layout_gravity: Es igual que el parámetro gravity, pero este parámetro es utilizado por el contenedor de la view, para ubicarla dentro de sí con las condiciones que aclara el parámetro.

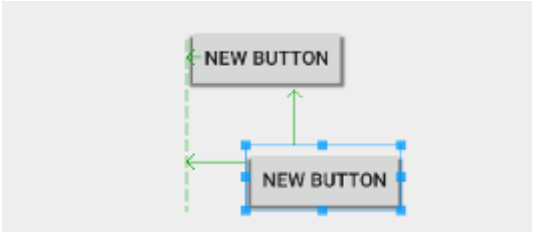
Ejemplo:

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

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

```
    <Button
        android:id="@+id/button1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="right"
        android:text="Button" />

</LinearLayout>
```

2.5 RelativeLayout

En este contenedor, la posición de cada View, se describe relativa a alguna otra View o contenedor. Es muy fácil posicionar los objetos mediante el editor gráfico del IDE, el cual generará automáticamente el archivo XML.



2.6 TextView y Edittext

TextView
El TextView Permite colocar un texto, seteando color, tamaño, posición, tipo de letra,etc.

Definición en XML:

```
<TextView
android:id="@+id/textView1"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:text="TextView" />
```

Propiedades de los TextView

- **android:typeface** que sirve para indicar el tipo de letra con que se mostrará el texto de la etiqueta (ejemplo: normal, sans, serif, monospace, etc.).
- **android:textStyle** para indicar si el tipo de letra seleccionado debe mostrarse en un formato en negritas (bold), cursiva (italic) o ambas (bold|italic).
- **android:textColor** para indicar el color que tendrá el texto mostrado en la etiqueta, este debe estar expresado en formato RGB Hexa (ejemplo: #A4C639 para el verde característico de Android).
- **android:textSize** Configura el tamaño del texto. La definición del valor para este atributo se conforma de un número de punto flotante seguido de una unidad de medida, que pueden ser: sp (scaled pixels), px (pixels), dp (density-independent pixels), in (inches), mm (millimeters).
- **android:shadowColor:** el color de la sombra del texto en formato RGB.
- **android:shadowRadius:** especifica el radio de la sombra con un número de punto flotante.

Utilizando TextView desde código java.

En la Activity obtenemos el objeto TextView mediante el método "findViewById()" el cual recibe como parámetro el id que se genera automáticamente en la clase R y que corresponde al id del objeto TextView definido en el xml:

```
TextView tv = (TextView) findViewById(R.id.textView1);
tv.setText("Texto desde Codigo");
```

EditText

Permite al usuario ingresar texto por pantalla.

Definición en XML:

```
<EditText
android:id="@+id/editText1"
android:layout_width="match_parent"
android:layout_height="wrap_content" >
```

Atributos del EditText:

- **android:inputType** indica el tipo de texto que se puede ingresar en el campo, algunos de los valores posibles son Phone, Number, Text, TextPassword.
- **android:hint** muestra un texto en gris, antes de que el usuario ingrese algo en el campo, es una ayuda para saber que ingresar en dicho campo.
- **android:capitalize** indica que texto se capitaliza, Algunos valores posibles son: sentences,none,characters.

- **android:digits** limita que caracteres pueden ser ingresados, pro ejemplo al poner "01" solo pueden ingresarse los numeros "0" y "1".

- **android:cursorVisible** indica si se muestra el cursor o no, valores posibles : True-false.
- **android:autoText :** Si es True, encuentra y corrige errores en el texto.
- **android:editable :** Si es True, deja editar el texto (False para TextView y True para EditText)

  Para obtener el texto escrito en este objeto desde el código Java, se utiliza el método "getText()" el cual devuelve un objeto "Editable", mediante el método "toString()" obtenemos el Texto.

2.7 FindVIewById

Cada vez que nosotros escribimos un nuevo tag dentro de nuestro archivo XML de layout, se creará e instanciará un objeto de este tipo en tiempo de ejecución al cargarse nuestra Activity. Por eso cada elemento dentro de nuestra activity.xml tiene una clase java asociada, por ejemplo cuando nosotros arrastramos y tiramos un elemento del tipo EditText, se escribe un nuevo tag dentro de nuestro xml, que en tiempo de ejecución va estar representado por una instancia de la clase EditText.

Nosotros vamos a tener accesos a esa instancia generada por la API de Android (al ejecutarse el método setContentView) mediante un método que está definido dentro de la clase java Activity, la cual es nuestra clase padre. FindViewById, como su nombre lo indica, nos va a devolver "la view" por el id que le pasemos. Como se observa el valor id es un valor del tipo int, lo cual puede resultar un poco extraño ya que nosotros le seteamos esta propiedad en nuestro xml con un nombre, no con un valor entero.
La relación entre los ids asignados en el XML y el código Java, se lleva a cabo mediante una clase llamada "R" (resurces), la cual contiene dentro una clase estática denominada "id" la cual posee todos los ids que definimos nosotros como constantes enteras, así que si nosotros definimos en el xml un botón con esta propiedad android:id="@+id/btn" dentro de la clase estática id vamos a encontrar public static final int btn=0x7f080040, este es el verdadero id de nuestro botón, así que es el que vamos a utilizar cuando ejecutamos el método findViewById. La invocación sería:

```
Button boton = (Button) findViewById(R.id.btn);
```

Debemos castear, ya que el método nos devuelve una instancia de View, la clase padre de todos los objetos que pueden colocarse en pantalla por medio del archivo XML de layout.

2.8 Botones y eventos

Button

La clase "Button" como todas las clases que representan un objeto que va en pantalla, hereda de la clase View. Este tipo de objeto nos permitirá colocar en pantalla un botón que contiene un texto y que el usuario puede presionar, el programador podrá capturar el evento de "botón presionado" para realizar alguna acción.

Para su definición en un archivo xml de layout, se utiliza el tag "Button", con los atributos que ya conocemos del TextView : id,layout_width,layout_height y text.

```
<Button
android:id="@+id/button1"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:text="Button" />
```

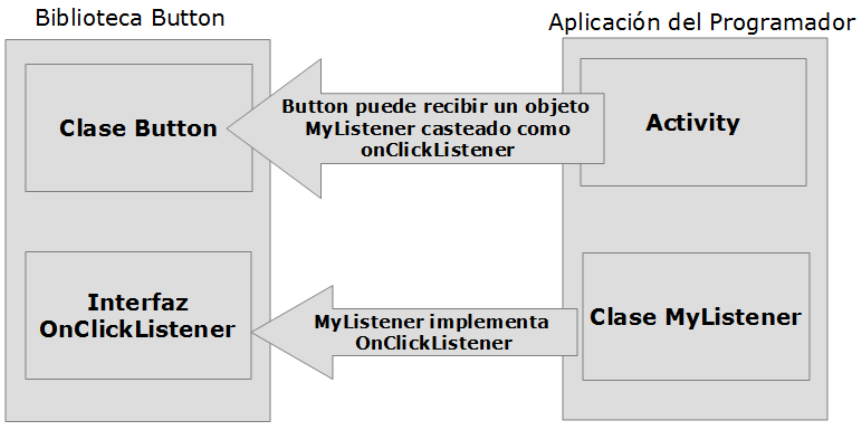
Para obtener en nuestra clase Activity una referencia de este objeto, lo haremos del mismo modo que se explicó anteriormente, mediante el método "findViewById()"

```
Button boton = (Button) findViewById(R.id.button1);
```

A continuación veremos como asociar una clase que "escuche" los eventos de este botón.

Listeners

Un "Listener" es un objeto que "escucha" los eventos que otro objeto produce. En este caso, ese otro objeto será nuestro botón.
El Listener es un objeto de una clase que implementa una interfaz que el objeto Button conoce; esto es así debido a que de este modo podremos pasar una referencia de nuestro objeto Listener (que es de una clase que la clase Button no conoce) a el objeto Button, ya que esta clase sí conoce la interfaz que el Listener implementa:



De este modo, la Clase Button, que internamente controla los eventos de Touch que se producen sobre la pantalla, y detecta que se hizo "click" sobre el objeto botón, ejecutará un método del objeto Listener del cual recibió una referencia.

¿Cómo sabe la clase Button qué métodos tiene nuestra clase MyListener? la respuesta es que no lo sabe; pero como la clase MyListener implementa una interfaz que la clase Button sí conoce: onClickListener, y esta interfaz define el método "onClick()", cuando se produzca un evento sobre el botón, la clase Button ejecutará el método "onClick()" del objeto MyListener pasado como referencia tratado como del tipo OnClickListener.

Veamos ahora el código Java que representa el diagrama:

La clase "MyListener", implementando la interfaz de la biblioteca del botón "OnClickListener":

```
public class MyListener implements OnClickListener{
	@Override
	public void onClick(View v) {
		//...
	}
}
```

En la activity, creamos un objeto de nuestra clase MyListener, y obtenemos una referencia del boton:

```
MyListener listener = new MyListener();
Button boton = (Button) findViewById(R.id.button1);
```

Luego pasamos una referencia de nuestro objeto "listener" al objeto "boton" para que éste le pueda ejecutar el método "onClick()" cuando se produce un evento sobre el mismo:

```
boton.setOnClickListener((OnClickListener)listener);
```

Como puede observarse esto se hace mediante el método "setOnClickListener()" de la clase Button, el cual recibe nuestro objeto "listener" casteado al tipo "OnClickListener" (que es el único tipo de dato que la clase Button conoce).

A partir de ese momento, cuando se produzca un click sobre el objeto Button que está en pantalla, se ejecutará el método "onClick()" de nuestra clase "MyListener".

> **NOTA:** Es muy común implementar la interfaz "OnClickListener" directamente en la Activity, de modo que cuando seteamos el listener al boton, le pasamos "this":
>
> ```
> boton.setOnClickListener(this);
> ```
>
> De este modo deberemos implementar en la clase que hereda de activity, el método onClick.

3.1 Componentes de una aplicación Android

Veremos los distintos tipos de componentes de software con los que podremos construir una aplicación Android.

En aplicaciones desktop estamos acostumbrados a manejar conceptos como ventana, control, eventos o servicios como los elementos básicos en la construcción de una aplicación. En Android vamos a disponer de esos mismos elementos básicos aunque con un pequeño cambio en la terminología y el enfoque. Repasemos los componentes principales que pueden formar parte de una aplicación Android.

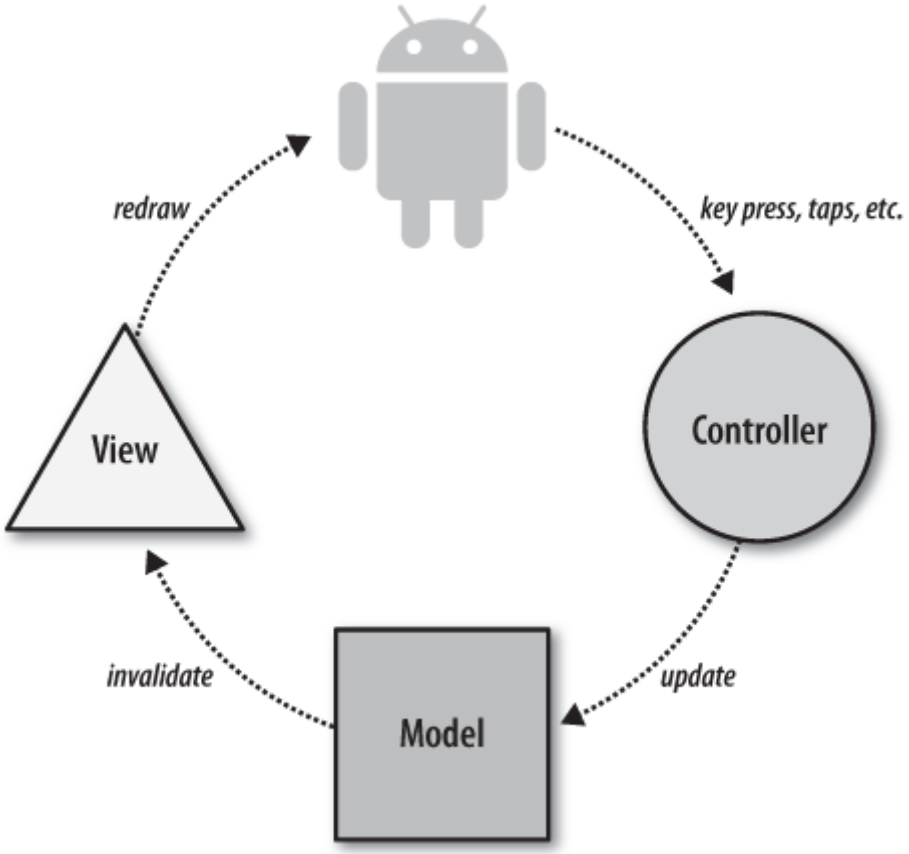
- Activity: Representan el componente principal de la interfaz gráfica de una aplicación Android. Se puede pensar en una activity como el elemento análogo a una ventana en cualquier otro lenguaje visual.
- View: Los objetos view son los componentes básicos con los que se construye la interfaz gráfica de la aplicación. Android pone a nuestra disposición una gran cantidad de controles básicos, como cuadros de texto, botones, listas desplegables o imágenes, aunque también existe la posibilidad de extender la funcionalidad de estos controles básicos o crear nuestros propios controles personalizados.
- Service: Los servicios son componentes sin interfaz gráfica que se ejecutan en segundo plano. En concepto, son exactamente iguales a los servicios presentes en cualquier otro sistema operativo. Los servicios pueden realizar cualquier tipo de acciones, por ejemplo actualizar datos, lanzar notificaciones, o incluso mostrar elementos visuales (p.ej. activities) si se necesita en algún momento la interacción con del usuario.
- Content Provider: Un content provider es el mecanismo que se ha definido en Android para compartir datos entre aplicaciones. Mediante estos componentes es posible compartir determinados datos de nuestra aplicación sin mostrar detalles sobre su almacenamiento interno, su estructura, o su implementación. De la misma forma, nuestra aplicación podrá acceder a los datos de otra a través de los content provider que se hayan definido.
- Broadcast Receiver: Un broadcast receiver es un componente destinado a detectar y reaccionar ante determinados mensajes o eventos globales generados por el sistema operativo (por ejemplo: "Batería baja", "SMS recibido", "Tarjeta SD insertada", …) o por otras aplicaciones (cualquier aplicación puede generar "mensajes" no dirigidos a una aplicación concreta sino a cualquiera que quiera escucharlo).
- Widgets: Se llama widget o form-widget a todo elemento que podemos colocar en la pantalla de nuestra aplicación.Estos elementos heredan de View. También se conoce como Widget a los elementos que van en el escritorio del sistema operativo y que permiten interacción con el usuario.

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

-

3.2 Concepto Model View Controller.

MVC: Patrón de diseño utilizado para aislar la lógica de negocio de la interfaz de usuario. Es muy común en android utilizar un patrón MVC por cada pantalla de nuestra aplicación:

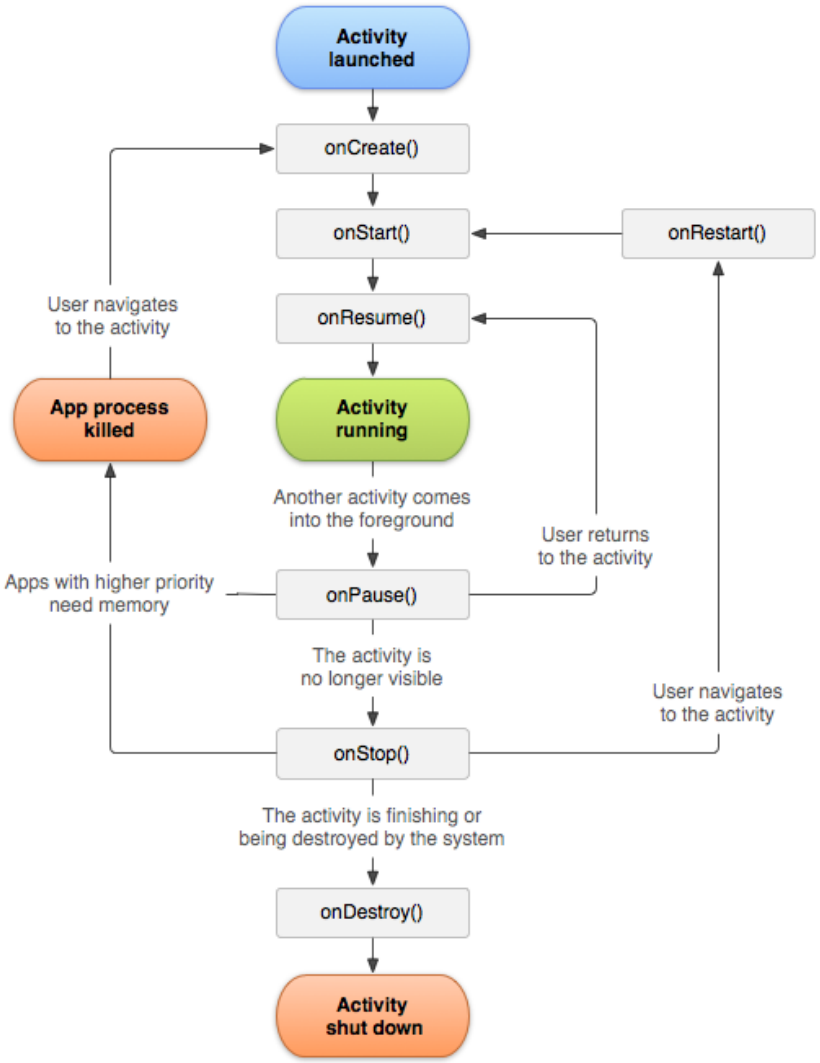


Modelo: Representa la información (data) de la aplicación y las reglas para usarla y manipularla.
Vista: Porción de la aplicación encargada de mostrar en pantalla.
Controlador: Porción de la aplicación que responde a acciones externas.

Comúnmente se utiliza a la Activity como el controller y el modelo (en el caso que sea simple), construyendo una clase aparte para el manejo de la vista.

3.3 Activities: Ciclo de vida de una aplicación.

Existen una gran cantidad de métodos en nuestra activity que podemos sobreescribir, y que se irán ejecutando a medida que la Activity cambie de estado.



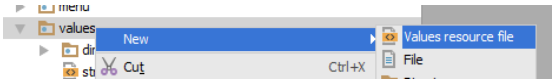
Hasta ahora solo reescribimos el método onCreate, existe otro método igual de importante que es el onStop, el cual será ejecutado antes de que la aplicación se cierre o pase a segundo plano.
Si la Activity pasa a segundo plano pero todavia es visible, solo se ejecutará onPause (por ejemplo con un dialogo)
Si la Activity pasa por completo a segundo plano se ejecutará onPause y onStop.
Si la Activity se cierra, se ejecutará también onDestroy.

Es importante aclarar que cuando el OS termina con la aplicación por requerimientos de RAM, el último método que se va a ejecutar con seguridad no es onDestroy sino onStop.

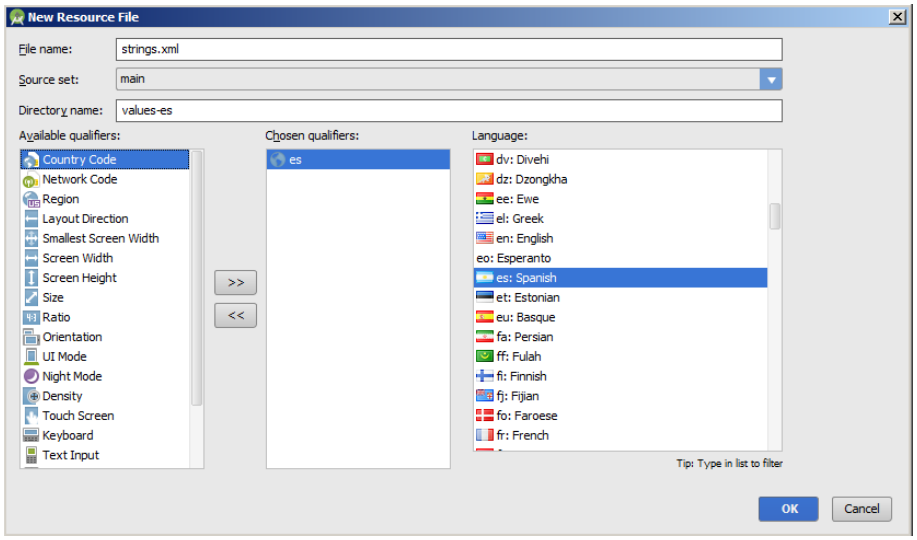
3.4 Internacionalización y Localización

Es importante tener la interfaz gráfica de la aplicación en el lenguaje correcto. Para ello, se nombran los directorios values con el idioma correspondiente. Por ej. values-es (español) o values-en para (inglés). Dentro de estos directorios colocaremos un archivos strings.xml en donde para las mismas "claves" definiremos textos en diferentes idiomas.

Para crear un nuevo directorio de values junto con un archivo strings.xml, hacemos click derecho sobre la carpeta value y seleccionamos new Value Resource File



Luego cargamos como nombre strings.xml y seleccionamos como modificador "Language". Veremos que el directorio pasará a llamarse values-es



Ponemos como ejemplo un archivo strings.xml para inglés y para español:

Inglés:

```
<resources>
    <string name="app_name">My Application</string>
    <string name="hello_world">Hello world!</string>
    <string name="action_settings">Settings</string>
</resources>
```

Español:

```
<resources>
    <string name="app_name">Mi Aplicación</string>
    <string name="hello_world">Hola mundo</string>
    <string name="action_settings">Opciones</string>
</resources>
```

Para obtener un string desde el código java:

```
String str = getString(R.string.hello_world);
```

Según el idioma del SO, el string tomará como valor lo que esta en un archivo o el otro.

Para utilizar estos strings desde un archivo xml, por ejemplo en un textview:

```
android:text:@string/hello_world
```

Además del idioma, podemos colocar modificadores por región, ya que a pesar de tener el mismo idioma diferentes países utilizan diferentes palabras.

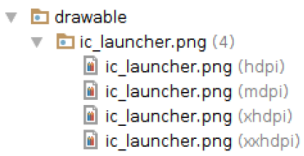
Por ej. values-es-rAR para Argentina o values-es-rCO para Colombia.

El values sin modificadores será el por defecto. En el caso que no se encuentre ningún values para una región o idioma específico, se utilizará este xml por defecto.

3.5 Resources: Strings, Colors y Pictures

De la misma forma que manejamos los textos por medio de strings.xml, podemos definir un archivo de colores colors.xml.

Con respecto a las imágenes, deberán colocarse en la carpeta drawable, las cuales tienen los modificadores de formato según la densidad de la pantalla:



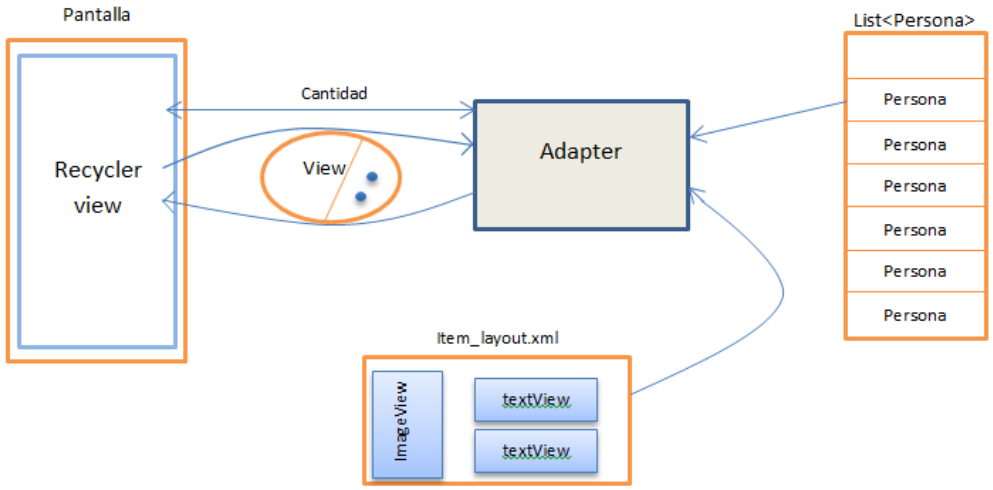
Esto le permite al sistema operativo tomar la imagen del tamaño más parecido al tamaño que debe dibujar en pantalla, para evitar que se pixele frente a un gran rango de densidades de pantalla (variación de los dpi)

El tamaño de las imágenes debe especificarse en la unidad dp y no en pixeles, esta unidad es independiente de la densidad de la pantalla, logrando que la imagen se va del mismo tamaño a pesar de la variación en la densidad:

4 RecyclerView

El Form Widget "RecyclerView" es uno de los más utilizados, permite mostrar en pantalla colecciones grandes de datos en forma de lista "scroleable" y manejar eventos de click sobre los ítems. Pero lo va a hacer de una forma algo distinta a la que estábamos habituados con los controles anteriores. Y es que RecyclerView no va a hacer "casi nada" por sí mismo, sino que se va a sustentar sobre otros componentes complementarios para determinar cómo acceder a los datos y cómo mostrarlos.

La información que mostraremos en cada ítem, estará contenida en un objeto (Persona, en nuestro ejemplo). Utilizaremos un adapter propio, el cual hereda de RecyclerView.Adapter, para decirle al objeto RecyclerView, qué debe mostrar. Este adapter recibirá una List de objetos Persona y generará un objeto View por cada fila de la lista que se verá en pantalla



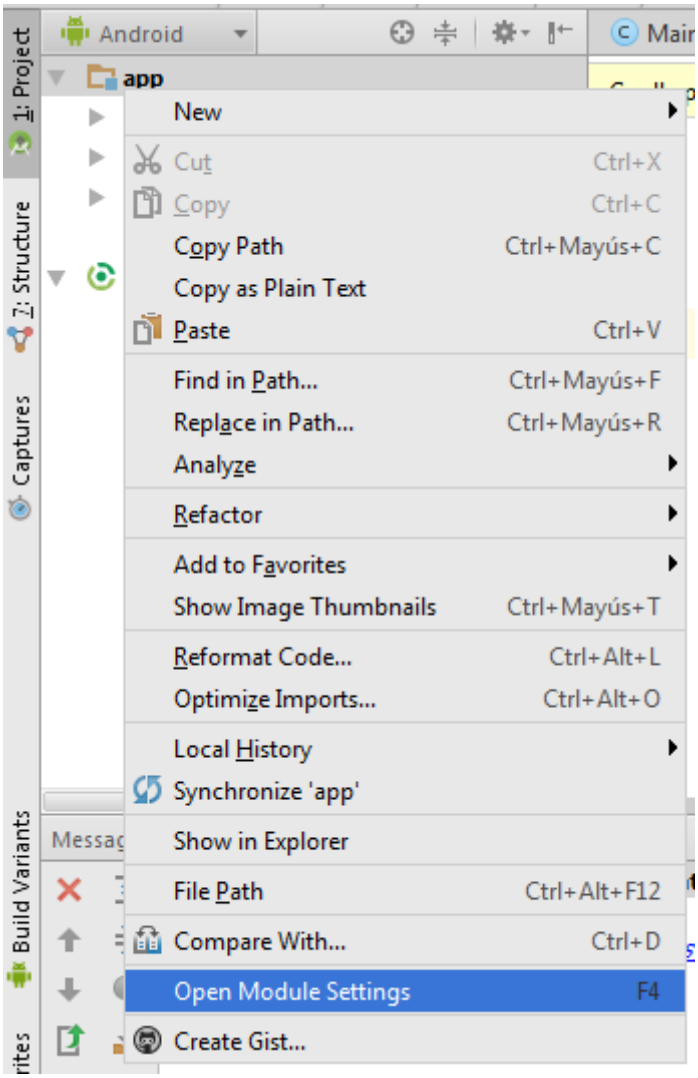
NOTA: Antes de la versión de Android 5.0, el Form Widget utilizado para esta tarea era el "ListView", el cual requería mucho trabajo en la implementación del adapter para lograr una buena performance en el scrolling. Debido a que RecyclerView se encuentra en la biblioteca de compatibilidad, utilizaremos siempre esta opción.

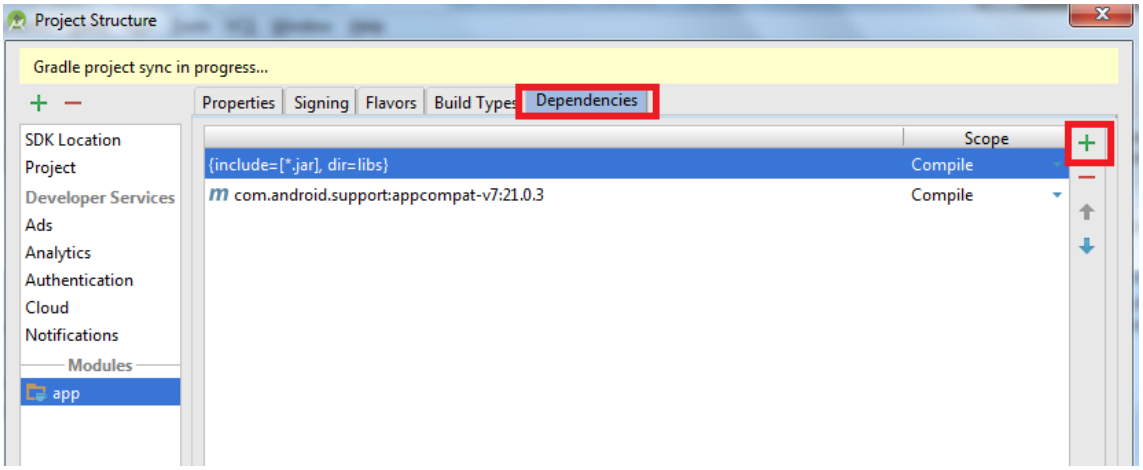
Paso número uno: Incluyendo dependencias

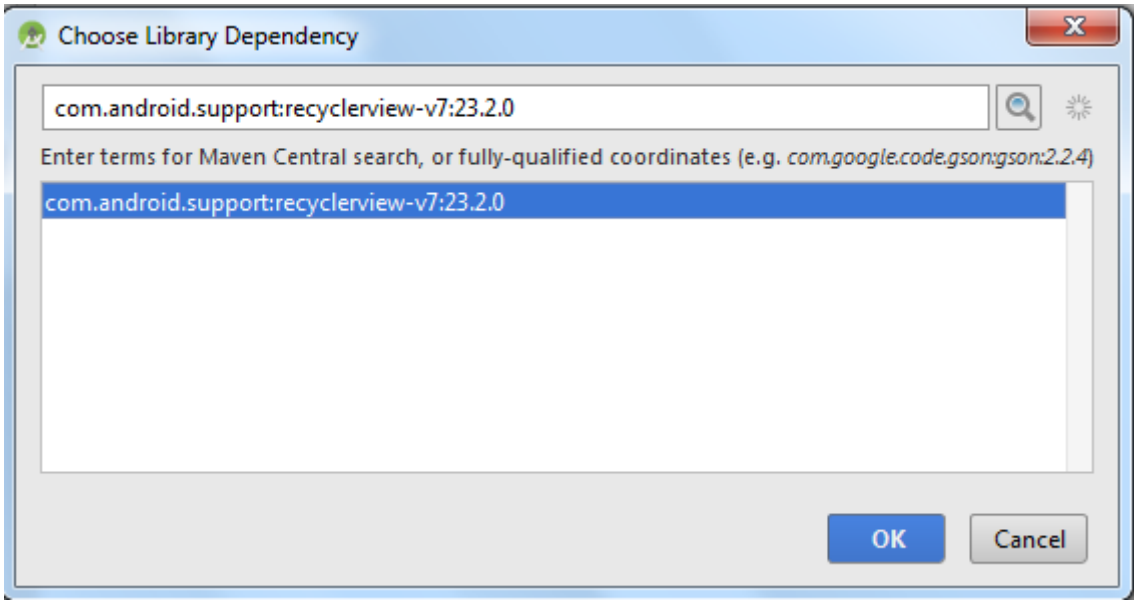
En nuestro archivo build.gradle del módulo "app", incluiremos dentro de las dependencias:

```
dependencies {
  compile 'com.android.support:recyclerview-v7:+'
}
```

O bien, pulsando botón derecho sobre la carpeta **app** > **Open module Settings**







Paso número dos: Definición de la entidad a representar

Definimos una entidad que contendrá la información que se quiere mostrar en una lista en una clase "Persona".

```
public class Persona {

	private String nombre;
	private String apellido;

	public void setNombre(String texto1) {
		this.nombre = texto1;
	}
	public String getNombre() {
		return nombre;
	}
	public void setApellido(String texto2) {
		this.apellido = texto2;
	}
	public String getApellido() {
		return apellido;
	}
}
```

Paso número tres: Definición del layout del item

Creamos el layout "item_layout.xml":

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

<ImageView
android:layout_width="60dp"
android:layout_height="60dp"
android:src="@drawable/avatar"
android:id="@+id/imageView" />

<LinearLayout
android:layout_marginLeft="10dp"
android:orientation="vertical"
android:layout_width="fill_parent"
android:layout_height="fill_parent">

<TextView
android:layout_marginTop="5dp"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:textAppearance="?android:attr/textAppearanceMedium"
android:text="Medium Text"
android:id="@+id/txtNombre" />

<TextView
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:textAppearance="?android:attr/textAppearanceSmall"
android:text="Small Text"
android:id="@+id/txtApellido" />
</LinearLayout>
</LinearLayout>
```

Paso número cuatro: Definición del adapter

Definimos nuestro propio Adapter heredando de RecyclerView.Adapter:

```
import android.support.v7.widget.RecyclerView.Adapter;
import android.view.ViewGroup;

public class MyAdapter extends Adapter<MyViewHolder> {

    @Override
    public MyViewHolder onCreateViewHolder(ViewGroup parent, int viewType) {
        return null;
    }

    @Override
    public void onBindViewHolder(MyViewHolder holder, int position) {

    }

    @Override
    public int getItemCount() {
         return 0;
    }
}
```

También deberemos definir la clase MyViewHolder, la cual deberá heredar de RecyclerView.ViewHolder

```
import android.support.v7.widget.RecyclerView;
import android.view.View;

public class MyViewHolder extends RecyclerView.ViewHolder {

    TextView txtNombre;
    TextView txtApellido;

    public MyViewHolder(View itemView) {
        super(itemView);
    }
}
```

Dentro de esta clase ViewHolder, definimos como atributos, las Views que contiene el ítem que definimos en el archivo de layout (los dos TextViews). La idea de esta clase es contener las referencias de las Views que contiene el ítem para no tener que ejecutar el método findViewById cada vez que se quiera obtener la referencia de los TextViews contenidos dentro de la View del item, de modo que lo ejecutamos en el constructor de esta clase el cual recibe como argumento la View que representa a un item, que más adelante veremos como crearla.

```
public MyViewHolder(View itemView)
{
    super(itemView);
    txtNombre = (TextView) itemView.findViewById(R.id.txtNombre);
    txtApellido = (TextView) itemView.findViewById(R.id.txtApellido);
}
```

Como se observa en la figura, un objeto MyViewHolder es un objeto que contendrá como atributos, dos TextViews obtenidos de una View pasada como argumento en su constructor, así como también dicha View (como atributo de la clase ViewHolder de la que hereda).

La ventaja de tener los dos TextViews como atributos, es no tener que llamar a *findViewById* cada vez que se necesitan.

Nuestro adapter, deberá recibir la lista de objetos persona que queremos representar en el RecyclerView, de modo que escribiremos un constructor que reciba la lista de objetos Persona:

```
public class MyAdapter extends Adapter<MyViewHolder> {

    private List<Persona> lista;

    public MyAdapter(List<Persona> lista)
    {
        this.lista=lista;
    }
```

El adapter debe devolver en el método getItemCount, la cantidad de ítems de la lista, de modo que reemplazamos el "return 0" por el size de la lista:

```
@Override
public int getItemCount() {
    return lista.size();
}
```

Ahora nos resta hablar sobre los métodos "onCreateViewHolder" y "onBindViewHolder".

El primero de ellos, es donde deberemos construir un objeto View a partir de nuestro archivo item_layout.xml y una vez obtenido dicho objeto View (que dentro posee los dos TextViews, y el ImageView que definimos en el archivo de layout) deberemos construir un objeto MyViewHolder que lo "envuelva", por último devolvemos este objeto MyViewHolder.

El RecyclerView le ejecutará este método al adapter cada vez que tenga que colocar un ítem en pantalla y no tenga ninguno para reutilizar.

```
@Override
public MyViewHolder onCreateViewHolder(ViewGroup parent, int viewType) {
    View v = LayoutInflater.from(parent.getContext()).inflate(R.layout.item_layout,
                                                              parent, false);
    MyViewHolder myViewHolder = new MyViewHolder(v);
    return myViewHolder;
}
```

En la primera línea creamos un objeto View a partir del archivo xml de layout que definimos para el ítem utilizando el método estático *from* de la clase LayoutInflater el cual devuelve un objeto Inflater, al cual le ejecutamos el método inflate.

Luego creamos nuestro objeto MyViewHolder pasándole la View creada y lo devolvemos.

El método onBindViewHolder, se ejecutará por cada ítem, antes de que éstos se muestren en pantalla, para que aquí los carguemos con la información proveniente de nuestro modelo de datos (nuestra lista de personas) en este método se recibe como argumento el objeto MyViewHolder que dentro tiene la View y la referencia de los TextViews del ítem, y también la posición del ítem que se quiere representar, de forma que podamos acceder al objeto Persona correspondiente en la lista y cargar los datos en los TextViews:

```
@Override
public void onBindViewHolder(MyViewHolder holder, int position) {

    Persona p = lista.get(position);

    holder.txtNombre.setText(p.getNombre());
    holder.txtApellido.setText(p.getApellido());
}
```

Paso número cinco: colocar el RecyclerView en el layout de la Activity.

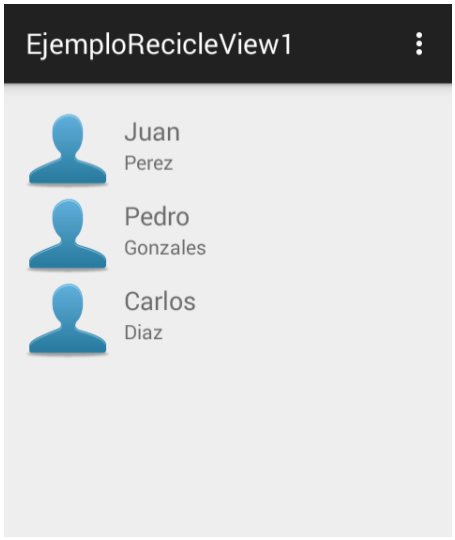
Por último, colocamos en el archivo de layout que carga la Activity, el objeto RecyclerView y le asignamos el id “list”:

```
<android.support.v7.widget.RecyclerView
android:id="@+id/list"
android:layout_width="match_parent"
android:layout_height="match_parent"
android:scrollbars="vertical" />
```

Ahora en la Activity, obtenemos la referencia del RecyclerView, creamos el modelo de datos, el adapter, y los relacionamos.

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

    List<Persona> personas = new ArrayList<Persona>();
    personas.add(new Persona("Juan","Perez"));
    personas.add(new Persona("Pedro","Gonzales"));
    personas.add(new Persona("Carlos","Diaz"));

    RecyclerView list = (RecyclerView)findViewById(R.id.list);

    LinearLayoutManager layoutManager = new LinearLayoutManager(this);
    list.setLayoutManager(layoutManager);

    MyAdapter adapter = new MyAdapter(personas);
    list.setAdapter(adapter);
}
```

Se debe notar que además del Adapter, le estamos asignando al RecyclerView, un LinearLayoutManager, de esta forma le indicamos que coloque un item debajo del otro. El SDK incorpora de serie tres LayoutManager para las tres representaciones más habituales: lista vertical u horizontal (LinearLayoutManager), tabla tradicional (GridLayoutManager) y tabla apilada o de celdas no alineadas (StaggeredGridLayoutManager). Siempre que optemos por alguna de estas distribuciones de elementos no tendremos que crear nuestro propio LayoutManager personalizado, aunque por supuesto nada nos impide hacerlo, y ahí uno de los puntos fuertes de este componente: su flexibilidad.



Modificación del modelo de datos

Supongamos que, obtenemos más objetos Persona, cuando esto ocurra, agregaremos los objetos a la lista “personas” pero este cambio no se reflejará en

pantalla hasta que no ejecutemos el método *notifyDataSetChanged* perteneciente al adapter. Al ejecutar este método se redibujará la lista y se verán los nuevos ítems. Esto es también necesario si modificamos cualquier atributo de cualquier objeto Persona de la lista.

```
adapter.notifyDataSetChanged();
```

Eventos

Es muy común querer escuchar el evento de click sobre un ítem de la lista, para ello, deberemos setear el listener de click sobre la view de cada ítem al momento de su creación.

Crearemos una interface donde definiremos el método que se ejecutará cuando se produzca un click sobre un ítem:

```
public interface MyOnItemClick {
    void onItemClick(int position);
}
```

Luego haremos que nuestra Activity implemente la interface:

```
public class MainActivity extends ActionBarActivity implements MyOnItemClick {
//…
@Override
public void onItemClick(int position) {

}
```

De esta manera, nuestra Activity será el listener de los clicks sobre los ítems, solo nos resta trabajar en nuestro adapter para notificar a la activity cuando una view de un ítem es presionada.

Agregamos en el constructor del adapter una referencia de la activity del tipo MyItemClick.

```
private List<Persona> lista;
private MyOnItemClick listener;

public MyAdapter(List<Persona> lista, MyOnItemClick listener)
{
    this.lista=lista;
    this.listener = listener;
}
```

Y en la activity, al crear el adapter, le pasamos “this”, ya que la activity implementa la interface.

```
MyAdapter adapter = new MyAdapter(personas,this);
```

Dentro del adapter, modificaremos el método onCreateViewHolder, para que este listener se pase como argumento en el constructor de nuestra clase MyViewHolder:

```
@Override
public MyViewHolder onCreateViewHolder(ViewGroup parent, int viewType) {
```

```
    View v = LayoutInflater.from(parent.getContext())
                                    .inflate(R.layout.item_layout, parent, false);
    MyViewHolder myViewHolder = new MyViewHolder(v,listener);
    return myViewHolder;
}
```

Por último, modificaremos la clase MyViewHolder para que reciba nuestro listener. Esta clase será la que escuche el evento de click sobre la View, y cuando se produzcan, ejecutará nuestro método onItemClick reportando el evento. Implementamos la interfaz OnClickListener en la clase MyViewHolder, y escribimos el método onClick

```
public class MyViewHolder extends RecyclerView.ViewHolder implements
                                                            View.OnClickListener {
//...
@Override
public void onClick(View v) {


}
```

Modificamos el constructor para que reciba MyOnItemClick, y también asignamos a la View itemView, como listener, a esta clase (osea *this*) mediante setOnClickListener:

```
private MyOnItemClick listener;

public MyViewHolder(View itemView,MyOnItemClick listener) {
    super(itemView);
    txtNombre = (TextView) itemView.findViewById(R.id.txtNombre);
    txtApellido = (TextView) itemView.findViewById(R.id.txtApellido);
    itemView.setOnClickListener(this);
    this.listener = listener;
}
```

Al producirse un click sobre la view (que es el item) se ejecutará el método onClick de la clase MyViewHolder, allí deberemos ejecutar el método onItemClick de nuestro listener.

```
@Override
public void onClick(View v) {
    listener.onItemClick(position);
}
```

Como este método requiere la posición del ítem, y el objeto View no conoce en qué posición se encuentra, creamos un atributo más en la clase MyViewHolder del tipo *int* llamado *position* y un método para poder cargarlo desde el adapter:

```
private int position;

public void setPosition(int position)
{
    this.position = position;
}
```

Ahora solo nos resta llamar a este método que actualiza la posición del objeto MyViewHolder en la lista en el método onBindViewHolder en el adapter:

```
@Override
public void onBindViewHolder(MyViewHolder holder, int position) {
    Persona p = lista.get(position);
    holder.txtNombre.setText(p.getNombre());
    holder.txtApellido.setText(p.getApellido());
    holder.setPosition(position);
```

}

5.1 Intents

Un Intent, es una descripción de una acción que se quiere realizar, por sí solo, éste no hace nada, sino que describe algo que tiene que ocurrir. Esta entidad se representa mediante un objeto.

Un IntentFilter, es la definición de una acción que una aplicación es capaz de llevar a cabo. Los IntentFilters de una aplicación, declaran qué tipo de acciones puede realizar dicha aplicación. Estos filtros se declaran como información de la aplicación en el Manifest.

Funcionamiento

Cuando una aplicación quiere realizar una acción, ésta declara su "intención" de realizarla mediante un Intent, y lo envía al OS. El sistema, al recibir este Intent, busca qué otra u otras aplicaciones pueden llevar a cabo dicha acción, analizando los IntentFilters de todas las aplicaciones disponibles.

En el caso de existir más de una aplicación que pueda llevar a cabo la tarea, el sistema le preguntará al usuario con cuál de todas ellas desea realizar la acción.

Tipos de Intents

- **Explícitos:** Se declara el nombre de una clase en particular que se quiere ejecutar.
- **Implícitos:** Se declara una acción a ejecutarse, sin saber qué aplicación la llevará a cabo.

Intents explícitos, ejecutando Activities

La forma de disparar una Activity desde otra Activity es mediante un Intent, en el cual se pasará como parámetro el nombre de la clase que se quiere ejecutar.

Dentro de la primera Activity:

```
Intent i = new Intent(this,Pantalla2Activity.class);
startActivity(i);
```

El primer parámetro del constructor del Intent es el Contexto, en este caso se pasa la Activity, y el segundo parámetro, es el objeto Class de la clase que queremos ejecutar, el cual lo obtenemos agregando ".class" el nombre de la clase.

Una vez creado el objeto Intent, se ejecuta el método “startActivity()”, perteneciente a la clase Activity, el cual recibe como parámetro el Intent anterior. En ese momento se disparará la segunda Activity.

Pasando información entre Activities

Si la información es de tipos de datos primitivos, se pasa directamente por medio de putExtra():

```
Intent i = new Intent(this,Pantalla2Activity.class);
i.putExtra("clave1","valor String");
i.putExtra("clave2",53);
startActivity(i);
```

El cual recibe 2 parámetros, "clave" y "valor", la clave siempre es un String, y el valor, puede ser cualquier tipo de dato primitivo.

¿Cómo llegan estos datos a la Activity que se dispara? Esta Activity podrá recuperar el objeto Intent mediante el método "getIntent()"

Código en la segunda Activity:

```
Intent intent = getIntent();
```

Luego se pide al Intent los extras, los cuales están contenidos en un objeto tipo "Bundle"

```
Bundle extras = intent.getExtras();
```

Este objeto Bundle, es el que contiene la información pasada a la Activity, y la cual puede obtenerse mediante el nombre de las claves:

```
String strActivity1 = extras.getString("clave1");
int intActivity1 = extras.getInt("clave2");
```

Intents implisitos

Para poder realizar una acción de este tipo, debemos previamente tener definido en el Manifest el permiso correspondiente. Luego antes de lanzar el intent debemos evaluar si el permiso fue aceptado o no, ya que a partir de la api 23 existe esta posibilidad.

```
Intent callIntent = new Intent
(Intent.ACTION_CALL,Uri.parse("tel:44444444"));

if (ContextCompat.checkSelfPermission(this,
Manifest.permission.CALL_PHONE) != PackageManager.PERMISSION_GRANTED){

ActivityCompat.requestPermissions(this,
       new String[]{Manifest.permission.ACTION_CALL},
      0);
```

```
}else{
startActivity(callIntet);
}
```

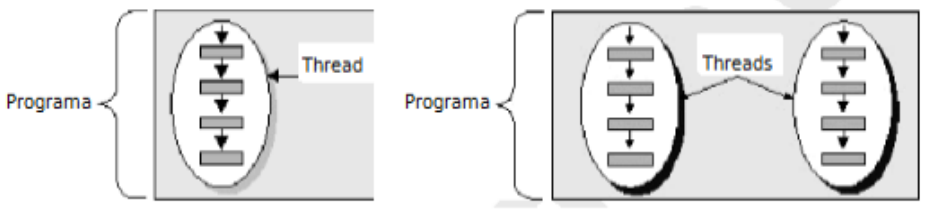
El primer if `ContextCompat.checkSelfPermission(this, Manifest.permission.CALL_PHONE)!= PackageManager.PERMISSION_GRANTED` lo que hace es validar si el usuario NO le dio permiso a la aplicación para realizar llamadas, si esto es así utilizamos el `ActivityCompat.requestPermissions` el cual le va a solicitar al usuario que acepte los permisos para que la aplicación pueda realizar llamadas. Si la condición del if no se cumple quiere decir que la aplicación tiene permisos para realizar llamadas, así que en el else podemos lanzar el intent. En este caso lanzamos el intent con `startActivity`, pero tenemos otra manera de lanzarlo. En caso de que esperemos un resultado debemos utilizar el método `startActivityForResult` pasandole el intent y un valor entero. Cuando se termine de ejecutar el intent nos va a llegar el resultado de la acción que intentamos realizar. Para poder interactuar con el resultado debemos sobrescribir el siguiente método en la activity que utilizamos para lanzar el intent

```
@Override
 protected void onActivityResult(int requestCode, int resultCode,
Intent data)
```

El primer parámetro que nos llega es el mismo valor entero que le pasamos al lanzar el intent, de esta manera podemos identificar de cual de todos los intents que lanzamos es la resupuesta. El segundo parámetro es para saber si el resultado fue positivo o negativo, debemos utilizar las constantes de la clase Activity `RESULT_OK` o `RESULT_CANCELED`. El último parámetro va a depender del intent, vamos a tener la información extra.

5.2 Threads

Una regla importante de las interfaces con el usuario, es que **siempre** deben responder a las acciones que el usuario realice, la pantalla jamás debe quedar congelada, de ser así se lanzará una excepción. Esto obliga al programador a realizar ciertas tareas que requieren tiempo (conexiones a internet, procesamiento de datos, etc.) en hilos de programa que se ejecuten en paralelo con la interfaz de usuario.



Si una Activity se bloquea por más de unos segundos, se lanzará una excepción y la aplicación se cerrará.

Para evitar este problema, y darle al usuario una sensación de que la aplicación está respondiendo todo el tiempo, utilizaremos Threads, es decir, lanzaremos ejecuciones de programa en paralelo con el hilo principal.

Utilizando la clase Thread de Java

Hay dos formas de crear Threads, se puede definir una clase que herede de Thread, y hacer Override del método "run()" (en donde se ejecutará en paralelo el código que coloquemos allí dentro) o puede definirse una clase que implemente la interfaz Runnable, e implementar el método "run()" de la misma forma que antes. La diferencia entre las dos maneras, está en la forma de disparar el Thread:

Ejemplo heredando de Thread:

Definimos una clase que hereda de Thread

```
public class MyThread extends Thread{

	@Override
	public void run() {
		for(int i=0 ; i<10 ; i++){
			Log.d("1","Ejecucion desde Thread");
		}
	}
}
```

En la Activity, creamos un objeto de esta clase y ejecutamos el método "start()" de la clase padre, de este modo, el método "run()" comenzará a ejecutarse en un Thread aparte.

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

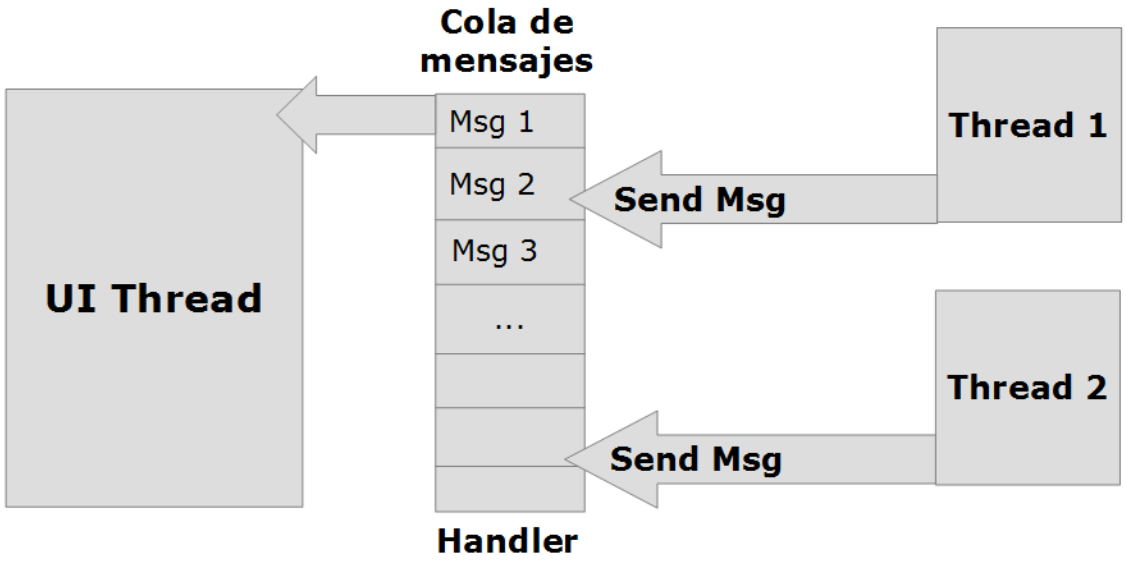
Comunicando la Activity con el Thread

No es posible refrescar los elementos de la interfaz gráfica del usuario (GUI) desde un thread que no sea el de la interfaz gráfica del usuario. Si el programador hace esto, el OS lanzará una excepción y se cerrará la aplicación.

Esto es así, debido a que si se tienen más de dos Threads, es necesario sincronizar los métodos para evitar que modifiquen la interfaz de usuario al mismo tiempo, esto agrega una complejidad extra a la Activity, para evitarla, se diseñó el sistema para que no sea posible actualizar la GUI desde un Thread que no sea el de la GUI.

La forma que Android ofrece para comunicar Threads, es a través de las clases "Handler" y "Message"

El Thread de la GUI, creará un objeto Handler, y pasará una referencia de este objeto a todos los Threads con los que quiera comunicarse. Este objeto se encargará de "encolar" los mensajes de todos los Threads y de hacerlos llegar a la GUI de a uno, es decir, la sincronización queda encapsulada dentro de la clase Handler.



Ejemplo:

```
public class Pantalla1Activity extends Activity implements Callback{
	private Handler handler;

	@Override
	public void onCreate(Bundle savedInstanceState) {
		super.onCreate(savedInstanceState);
		setContentView(R.layout.main);
		handler = new Handler(this);
		Worker w = new Worker(handler);
		Thread t = new Thread(w);
		t.start();
	}

	@Override
	public boolean handleMessage(Message msg) {
		// Aca se reciben los mensajes
		return true;
	}
}
```

En el ejemplo, puede observarse que la Activity (GUI) crea un objeto del tipo Handler, este objeto, para ser creado, necesita un objeto del tipo Callback, que es el objeto en donde se depositarán los mensajes que se envían a través del Handler, en este caso, la propia Activity implementa la interfaz Callback (por eso pasa this en el constructor), y hace Override del método ”handleMessage()” el cual será llamado por objeto Handler para descargar los mensajes de su cola.

En este caso, creamos una clase llamada Worker que utiliza Threads, el objeto Handler es pasado como parámetro en el constructor, esta es la definición de la clase:

```
public class Worker implements Runnable{
	private Handler h;

	public Worker (Handler h)
	{
		this.h = h;
	}

	@Override
	public void run() {

		for(int i=0 ; i<10 ; i++)
		{
			sendMsg("Worker:"+i);

			try {
				Thread.sleep(1000);
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
		}

		sendMsg("Fin de thread worker");
	}

	private void sendMsg(String msg)
	{
		Message message = new Message();
		message.obj = msg;
		h.sendMessage(message);
	}
}
```

En esta clase, se recibe el objeto Handler en el constructor, y se guarda una referencia del mismo, ya que con él, se enviarán los mensajes a la Activity.

La clase Worker, tiene un método "run()" que itera 10 veces en un bucle con un delay de 1 segundo, es delay, se consigue mediante la llamada de un método estático de la clase Thread, llamado "sleep()"

Por cada iteración, se llama al método "sengMsg()" definido en la propia clase. En este método, se utiliza el objeto Handler perteneciente a la GUI, para enviar un mensaje a la misma por medio del método "sendMessage()" el cual recibe como parámetro, un objeto del tipo Message.

```
Message message = new Message();
message.obj = msg;
h.sendMessage(message);
```

El objeto Message, posee 3 atributos que pueden ser cargados con información:

- **obj** **:** atributo tipo Object.
- **arg1** **:** atributo tipo int.
- **arg2** **:** atributo tipo int.

En el ejemplo, utilizamos el atributo "obj" para transmitir un objeto String desde el Thread Worker hacia la Activity.

Para recibir esta información en la Activity, dentro del método "handleMessage()" el cual recibe como parámetro el objeto Message:

```
@Override
public boolean handleMessage(Message msg) {
	String text = (String) msg.obj;
	TextView tv = (TextView) findViewById(R.id.lbl1);
	tv.setText(text);
	return true;
}
```

En este caso, casteamos el objeto del mensaje a String y se lo asignamos a un TextView, esto es posible porque la asignación se lleva a cabo en la GUI.

El resultado de la ejecución es la siguiente, el TextView se actualizará cada 1 segundo

5.5 Conexiones HTTP

Conexiones HTTP

Existen dos bibliotecas que permiten la conexión HTTP contra un servidor web:

- **HttpURLConnection :** Biblioteca standard de java, se recomienda en versiones de Android mayores a Gingerbread por ser liviana.
- **Apache HTTP Components :** Las bibliotecas de Android incluyen las "Apache HTTP Components Libraries" las cuales son de código abierto y pueden usarse en java standard si se incluyen por separado, pero en las bibliotecas de Android fueron incluidas como parte de las standard por lo que no hay que incluirlas para usarlas.

Centraremos este material en el uso de la biblioteca de HttpURLConnection.

HttpURLConnection

Para realizar un request HTTP, deberemos crear un objeto del tipo HttpURLConnection el cual recibe como argumento un objeto del tipo URL, el cual es creado especificando la URL sobre la que se quiere realizar un request HTTP. Una vez obtenido este objeto, podemos indicarle el protocolo HTTP a utilizar (GET, POST, PUT, PATCH, DELETE) y si vamos a obtener información o a leer información de la respuesta. También podemos configurar los timeouts de conexión, y luego realizar el request, del cual obtendremos un objeto InputStream, que nos permitirá leer la información obtenida desde el servidor.

Creación de una clase HttpManager

Crearemos una clase HttpManager que nos simplificará la configuración y la obtención de datos permitiéndonos hacer requests http.

```
	public class HttpManager {

		private String url;
		private HttpURLConnection conn;

		public HttpManager(String url)
		{
			conn = crearHttpUrlConn(url);
		}
```

En el constructor, recibimos la URL y creamos el objeto HttpURLConnection mediante el método *crearHttpUrlConn* el cual tendrá el siguiente contenido:

```
private HttpURLConnection crearHttpUrlConn(String strUrl)
{
	URL url = null;
	try {
		url = new URL(strUrl);
		HttpURLConnection urlConnection = (HttpURLConnection)

url.openConnection();
		urlConnection.setReadTimeout(10000 /* milliseconds */);
		urlConnection.setConnectTimeout(15000 /* milliseconds */);
		return urlConnection;
	} catch (MalformedURLException e) {
		e.printStackTrace();
	} catch (IOException e) {
		e.printStackTrace();
	}
	return null;
}
```

Como se observa en el método, creamos un objeto URL mediante el cual podemos crear el objeto HttpURLConnection el cual devolvemos. En caso de error devolveremos null.

Escribiremos ahora un método que nos indique si se pudo crear el objeto correctamente:

```
public boolean isReady(){
    return conn!=null;
}
```

Estamos en condiciones de escribir un método que realice un request del tipo GET y que nos devuelva un array de bytes con la respuesta. Si el request devolvía texto, luego convertiremos dicho array de bytes en un objeto String, si el request era sobre un archivo binario (un archivo de imagen por ejemplo) nos quedaremos con dicho array de bytes.

```
public byte[] getBytesDataByGET() throws IOException {
        conn.setRequestMethod("GET");
        conn.connect();
        int response = conn.getResponseCode();
        if(response==200) {
                InputStream is = conn.getInputStream();
                return readFully(is);
        }
        else
                throw new IOException();
}
```

En este método, configuramos al objeto HttpURLConnection para realizar un request del tipo GET y luego ejecutamos el método connect el cual es bloqueante y realiza la conexión.

NOTA: Es importante recordar que como el método es bloqueante y puede tardar varios segundos, deberemos ejecutar este método en un Thread.

Una vez que el método connect termina, podemos obtener el código de respuesta (200 OK, 404 página no encontrada, etc.) En el caso de obtener un 200, estamos en condiciones de leer la respuesta.

Leyendo la respuesta del request mediante InputStream

Para leer la respuesta del servidor, deberemos obtener un objeto InputStream del objeto HttpURLConnection, mediante el método getInputStream

```
InputStream is = conn.getInputStream();
```

Este stream tiene métodos que nos permitirá obtener los bytes del mensaje que respondió el servidor. Para ello creamos un método privado en nuestra clase HttpManager, que se encargue de ir leyendo los bytes desde el stream y devolver un array de bytes, llamamos a este método readFully.

```
private byte[] readFully(InputStream inputStream) throws
                                                        IOException
{
            ByteArrayOutputStream baos = new ByteArrayOutputStream();
            byte[] buffer = new byte[1024];
            int length = 0;
            while ((length = inputStream.read(buffer)) != -1) {
                  baos.write(buffer, 0, length);
            }
            inputStream.close();
            return baos.toByteArray();
      }
```

En el método readFully, creamos un objeto ByteArrayOutputStream, en el cual podemos ir concatenando bytes de forma eficiente, en este caso, creamos un buffer auxiliar de 1Kbyte en donde vamos leyendo los datos del InputStream, luego escribimos dicho buffer en el objeto ByteArrayOutputStream concatenándolos con los existentes.
Al terminar de leer el stream, simplemente retornamos un objeto array de bytes obtenido desde el ByteArrayOutputStream.

Por último agregamos un método que haga un request por GET y nos devuelva un String, para ello utilizaremos el método escrito previamente, y luego convertiremos el array de bytes a String:

```
      public String getStrDataByGET() throws IOException {
            byte[] bytes = getBytesDataByGET();
            return new String(bytes,"UTF-8");
      }
```

Subiendo datos por POST

Escribiremos un pequeño script en nuestro servidor, para que nos devuelva lo mismo que enviamos por POST, con la clave "data".

```
<?php
echo($_POST["data"]);
?>
```

http://www.lslutnfra.com/alumnos/practicas/postEcho.php

De esta manera, si realizamos un request a dicha URL, pasándo un texto con la clave "data" por POST, obtendremos como respuesta del request, el mismo texto.

Agregaremos a nuestra clase HttpManager, un método para hacer un request POST

```
public byte[] getBytesDataByPOST(Uri.Builder postParams) throws

IOException
{
	conn.setRequestMethod("POST");
	conn.setDoOutput(true);
	String query = postParams.build().getEncodedQuery();
	OutputStream os = conn.getOutputStream();
	BufferedWriter writer = new BufferedWriter(new
					OutputStreamWriter(os,"UTF-
8"));
	writer.write(query);
	writer.flush();
	writer.close();
	os.close();

	int response = conn.getResponseCode();

	if(response==200) {
		InputStream is = conn.getInputStream();
		return readFully(is);
	}
	else
		throw new IOException();
}
```

La primera diferencia que encontramos es que pasamos un String con la palabra "POST" y ejecutamos el método setDoOutput, esto nos permitirá obtener un OutputStream para escribir los datos que queremos enviar al servidor.
Los datos clave-valor que queremos enviar por POST, estará dentro del objeto Uri.Builder, el cual recibiremos como argumento. Para crear este objeto y cargarle los pares clave-valor podemos utilizar el siguiente código:

```
	Uri.Builder params = new Uri.Builder();
	params.appendQueryParameter("data","Hola mundo");
	params.appendQueryParameter("data2", "LSL UTN-FRA");
```

Dentro de nuestro método, transformamos este objeto Uri.Builder en un String con el formato correcto para el protocolo HTTP, mediante:

```
String query = postParams.build().getEncodedQuery();
```

Luego escribimos este String mediante el objeto OutputStream:

```
BufferedWriter writer = new BufferedWriter(new OutputStreamWriter(os,
"UTF-8"));
writer.write(query);
writer.flush();
writer.close();
os.close();
```

Es importante destacar que en este caso no ejecutamos el método connect del objeto "conn" ya que la conexión se establecerá mediante el uso del OutputStream. Luego obtenemos la respuesta de forma idéntica que en el método getBytesDataByGET.

NOTA: Es importante recordar que para que la aplicación pueda conectarse a Internet, es necesario aclarar el permiso de conexión en el archivo manifest.xml.

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

6 XML Parsers

6.1 Sintaxis

El formato XML (Extensible Markup Languaje) permite crear documentos en un formato que puede ser leído fácilmente tanto para una persona como para una computadora.Se utiliza para representar estructuras de datos.

Ejemplo:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE Edit_Mensaje SYSTEM "Edit_Mensaje.dtd">

<Edit_Mensaje>
     <Mensaje>
          <Remitente>
               <Nombre>Nombre del remitente</Nombre>
               <Mail> Correo del remitente </Mail>
          </Remitente>
          <Destinatario>
               <Nombre>Nombre del destinatario</Nombre>
               <Mail>Correo del destinatario</Mail>
          </Destinatario>
          <Texto>
               <Asunto>
                    Este es mi documento con una estructura muy sencilla
                    no contiene atributos ni entidades...
               </Asunto>
               <Parrafo>
                    Este es mi documento con una estructura muy sencilla
                    no contiene atributos ni entidades...
               </Parrafo>
          </Texto>
     </Mensaje>
</Edit_Mensaje>
```

Como se observa, el formato se trata de tags anidados, en donde dentro de cada tag puede haber un texto, u otro/s par/es de tags.

Un tag se abre y se cierra con la siguiente sintaxis:

<tag> contenido **</tag>**

El contenido puede ser uno o más pares de tags.

Cada tag puede tener atributos los cuales están compuestos por una clave y un valor:

<tag **nombre=”juan”** > Contenido </tag>

Cuando el tag no posee contenido, puede abrirse y cerrarse con el mismo tag:

<tag nombre=”juan” />

6.2 Parsers

Existen 3 tipos principales de parsers:

- DOM (Document Object Model)
- SAX
- XmlPull

Los parsers tipo DOM cargan en memoria todo el documento. Esto permite acceder a cualquier parte del documento en forma aleatoria.

Los parsers tipo SAX no necesitan cargar en memoria todo el documento. Son "stream-based", parsean el documento a medida que lo leen. No puede accederse a cualquier parte del documento aleatoriamente. El parser comienza a recorrer el XML y genera eventos al encontrar los tags.

Los parsers tipo Pull, son más parecidos a iteradores. El programador generará la petición para "avanzar" y parsear la porción siguiente del archivo.

Parseando con XmlPull

Creamos el objeto parser:

```
XmlPullParser parser = Xml.newPullParser();
```

Le pasamos un InputStream o un Reader (fuente de datos):

```
parser.setInput(new StringReader(xmlTxt));
```

Iteramos al parser y leemos los eventos:

```
event = parser.getEventType();
while (event != XmlPullParser.END_DOCUMENT)
{
    switch (event)
    {
      case XmlPullParser.START_DOCUMENT:
             //...
      case XmlPullParser.START_TAG:
             //...
      case XmlPullParser.END_TAG:
            //...
    }
    event = parser.next();
}
```

Obtenemos el texto dentro del tag, por ejemplo, si el formato es:

<titulo>Texto dentro del tag</titulo>

```
case XmlPullParser.START_TAG:
{
    String tag = parser.getName();
    if (tag.equals(“titulo”))
    {
         // inicio del tag <titulo>
         String texto = parser.nextText()
    }
    // Preguntamos por el resto de los tags
    // ...
    break;
}
```

Lectura de atributos. Por ejemplo si el formato es:

<titulo atributo='5' >Texto dentro del tag</titulo>

```
case XmlPullParser.START_TAG:
{
    String tag = parser.getName();
    if (tag.equals(“titulo”))
    {
       // inicio del tag <titulo>
       String att = parser.getAttributeValue(null,"atributo");
        String texto = parser.nextText()
    }
    break;
}
```

JSON

JSON (JavaScript Object Notation) es un formato para el intercambios de datos, básicamente JSON describe los datos con una sintaxis dedicada que se usa para identificar y gestionar los datos. Nació como una alternativa a XML y, una de las mayores ventajas que tiene su uso es que puede ser leído por cualquier lenguaje de programación. Por lo tanto, puede ser usado para el intercambio de información entre distintas tecnologías. Además debido a su naturaleza y al ser más compacto suele ser mucho más rápido trabajar con JSON antes que con XML.

Pongamos un ejemplo, imagínese que tenemos una lista de frutas con su nombre y cantidad:

Frutas:

- Manzana - 10 unidades
- Pera - 20 unidades
- Naranja - 30 unidades

Esto se traduciría en JSON de la siguiente manera:

```
{
   "frutas": [
       { "nombre_fruta":"Manzana" , "cantidad":10 },
       { "nombre_fruta":"Pera" , "cantidad":20 },
       { "nombre_fruta":"Naranja" , "cantidad":30 }
    ]
}
```

Supongamos que tenemos la siguiente clase con la cual queremos trabajar en Java:

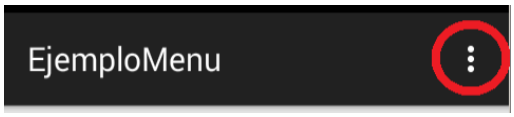
```
public class Fruta {
     public String nombre;
     public Integer cantidad;
}
```

Nuestro objetivo es pasar de una cadena de JSON a una Lista de Objetos Fruta con los que poder trabajar cómodamente en Java, para lo cual deberemos hacer lo siguiente.

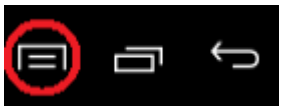
```
try {
   JSONObject jsonObject = new JSONObject(jsonFrutas);
   JSONArray frutas = jsonObject.getJSONArray("frutas");
   for(int i=0;i<frutas.length();i++){
       JSONObject fruta = frutas.getJSONObject(i);
       String nombre = fruta.getString("nombre_fruta");
       Integer cantidad = fruta.getInt("cantidad");
   }
}catch (Exception e){
   e.printStackTrace();
}
```

7.1 Menú

El sistema operativo provee un botón llamado "Menú", al presionarse, la Activity que esté activa en ese momento, tendrá la posibilidad de capturar ese evento y generar un menú en pantalla.
Cuando el dispositivo no posee botón menú, aparecerá un botón en la ToolBar para desplegar las opciones, sin embargo, el mecanismo explicado a continuación es el mismo.



Botón de menú en ToolBar



Botón de menú Físico

Para activar el menú, la Activity posee dos métodos a los que debe hacerse Override:

- **onCreateOptionsMenu():** Se llamará al presionar el botón menú, aquí se debe construir el menú en pantalla.

- **onOptionsItemSelected():** Se llamará al presionar una opción en el menú.

Creando el menú

Existen dos formas de crear el menú, por código, o mediante un archivo xml de resource, explicaremos esta última opción por ser la recomendada.

Dentro de la carpeta menú que se encuentra dentro de los recursos (carpeta res) se encontrarán los archivos xml que definen los ítems de menú.

Luego, se debe utilizar la clase MenuInflater para generar el objeto "Menu" a partir del archivo xml.

Ejemplo de archivo **menu_main.xml**:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
xmlns:app="http://schemas.android.com/apk/res-auto"
xmlns:tools="http://schemas.android.com/tools"
tools:context=".MainActivity">

<item android:id="@+id/action_settings"
android:title="Settings"
android:orderInCategory="100"
app:showAsAction="never" />

<item android:id="@+id/action_opcion1"
android:title="Opcion 1"
android:orderInCategory="101"
```

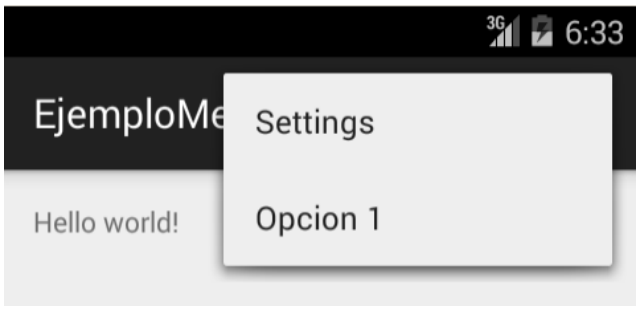
```
app:showAsAction="never" />

</menu>
```

Para crear el menú a partir de este archivo, dentro del método onCreateOptionsMenu():

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

El método "getMenuInflater()" es propio de la clase Activity, y nos devuelve un objeto MenuInflater, el cual a partir del método "inflate()" podrá cargar al objeto "menu" con los ítems definidos en el archivo xml.

Al ejecutar la aplicación, veremos que ahora el icono de menú, que se encuentra en la Toolbar, tendrá nuestras dos opciones:



Recibiendo eventos del menú

Para recibir los eventos del menú, solo basta con hacer Override del método "onOptionsItemSelected()" el cual tendrá como parámetro un objeto del tipo MenuItem, este objeto, corresponderá con el ítem seleccionado en el menú.

El objeto MenuItem posee el método "getItemId()" el cual devolverá el Id del ítem, (correspondiente al id pasado como parámetro en el método "add()" o el definido en el xml al crear el menú.)

Ejemplo:

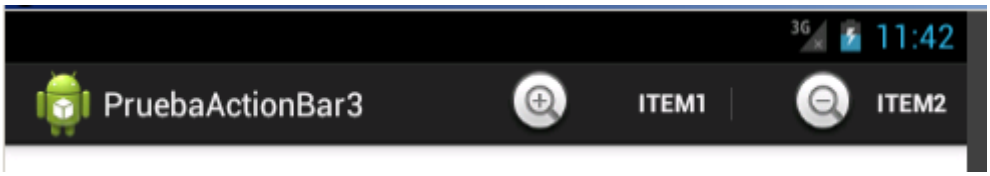
```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
 int id = item.getItemId();

 if (id == R.id.action_settings) {
        Log.d("menu","Click en settings");
       return true;
 }
 if (id == R.id.action_opcion1) {
       Log.d("menu","Click en opcion 1");
```

```
            return true;
        }
        return super.onOptionsItemSelected(item);
    }
```

En el caso de que el ítem seleccionado no concuerde con ninguno de los creados, probablemente sea porque se eligió una de las opciones de sistema, por lo que en ese caso, llamamos a el método "onOptionsItemSelected()" de la clase padre, mediante "super".

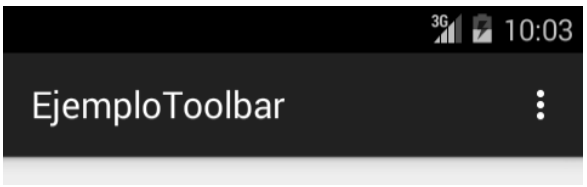
7.2 Toolbar

La Toolbar es un Widget en la parte superior de la pantalla que identifica a la aplicación y permite al usuario un acceso rápido y standard a los controles de navegación. Antes de la versión de Android 5.0, El widget que cumplía esta función era llamado ActionBar :



ActionBar utilizada antes de Android 5.0

A partir de Android 5.0 apareció la ToolBar, la cual nos da una mayor flexibilidad, permitiéndonos modificar su tamaño, color, agregarle widgets, etc.



Nueva Toobar de Android 5.0

Compatibilidad

La ToolBar esta disponible a partir de la API 21 (Android 5.0 Lolipop). Para utilizarla en versiones anteriores, es necesario incluir la biblioteca de compatibilidad "appcompat" la cual se agrega automáticamente a nuestro proyecto cuando lo creamos a partir de Android 4.0. Para verificar esta condición, podemos chequear el archivo build.gradle de nuestra app, en donde deberá aparecer la dependencia:

```
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    compile 'com.android.support:appcompat-v7:21.0.3'
}
```

Agregando "Action Items"

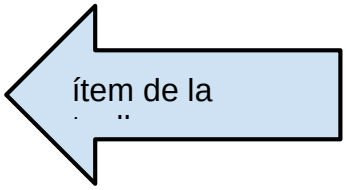
Para agregar ítems a la derecha de la Toolbar (botones de compartir, buscar, etc) se deben definir en el mismo archivo xml en donde se definen los ítems del menú.

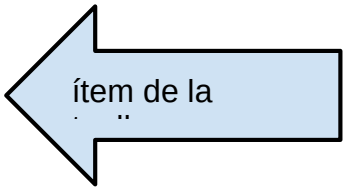
Ejemplo:

En el archivo menu_main.xml definiremos el ítems que queremos ver en la Toolbar, junto con sus íconos:

```
<menu
```


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

<item
android:id="@+id/action_settings"
android:orderInCategory="100"
app:showAsAction="never"
android:title="@string/action_settings"/>
<item
android:id="@+id/menu_item1"
android:orderInCategory="100"
app:showAsAction="always"
android:icon="@android:drawable/btn_plus"
android:title="item1"/>
<item
android:id="@+id/menu_item2"
android:orderInCategory="100"
app:showAsAction="always"
```

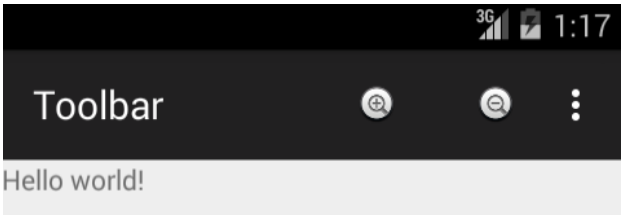





```
android:icon="@android:drawable/btn_minus"
android:title="item2"/>

</menu>
```

Si el atributo "showAsAction" toma el valor "never", el ítem no es parte de la Toolbar, en cambio si toma el valor "always" aparecerá siempre, si toma el valor "ifRoom" el ítem se mostrará si hay lugar para hacerlo. El ícono se define en el atributo "icon"

Ahora los ítems se verán en la Toolbar:

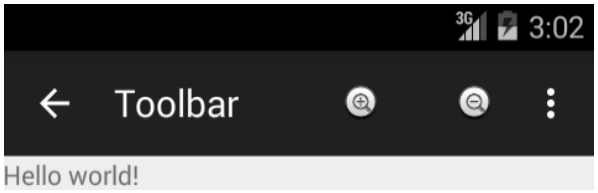


Capturando los eventos

Los eventos de los ítems de la Action Bar se capturan igual que los ítems del menú; sobreescribiendo el método "onOptionsItemSelected", y preguntando por los ids de los ítems.

Botón de back

Al configurar el botón como "Back", aparecerá a la izquierda del mismo una flecha indicando que al presionarse se volverá a la pantalla anterior.



Para que esta flecha aparezca, debemos ejecutar el método "setDisplayHomeAsUpEnabled" en el onCreate de la Activity:

```
ActionBar ab = getSupportActionBar();
ab.setDisplayHomeAsUpEnabled(true);
```

Para capturar el evento sobre este botón, se debe preguntar por el id definido en la clase R de Android llamado "home":

```
switch(item.getItemId()){
	case R.id.menu_item1:
	{
		Log.d("main","Click en action item 1");
		break;
	}
	case R.id.menu_item2:
	{
		Log.d("main","Click en action item 2");
		break;
	}
	case android.R.id.home:{
		finish(); // Simulo back
		break;
	}
}
```

Agregando un SearchView

Es posible agregar en la Toolbar un Widget que tenga un botón de búsqueda y un campo para escribir un texto. Para agregar este elemento, debemos definir un item de menú con las siguientes características:

```
<item android:id="@+id/campo_buscar"
    android:orderInCategory="100"
    app:showAsAction="ifRoom"
    android:title="Buscar"
    app:actionViewClass="android.support.v7.widget.SearchView"
/>
```

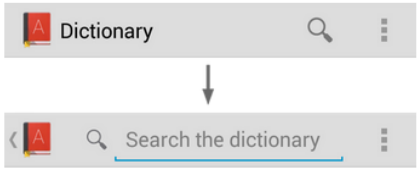


Figure 5. An action bar with a collapsible `SearchView`.

Para escuchar los eventos de este widget (tendremos un evento cada vez que el usuario modifica un caracter de lo que va escribiendo, y uno cuando realiza la búsqueda) deberemos implementar la interface SearchView.OnQueryTextListener, en el siguiente ejemplo, implementamos la interface en una Activity:

```
public class MainActivity extends ActionBarActivity implements
                                                    SearchView.OnQueryTextListener
{

	@Override
	public boolean onQueryTextSubmit(String s) {
		Log.d("activity", "Hago una busqueda con:" + s);
		return false;
	}

	@Override
	public boolean onQueryTextChange(String s) {
		Log.d("activity", "cambio texto:" + s);
		return false;
	}
```

Para setear el listener (es decir la Activity en nuestro caso) al objeto SearchView que se encuentra en la Toolbar, deberemos hacerlo en el método onCreateOptionsMenu (donde creamos los ítems de menú) obteniendo primero el objeto MenuItem que corresponde a este widget definido en el archivo XML de menú, y luego a partir del *MenuItem*, mediante el método estático *getActionView* de la clase *MenuItemCompat*, podemos obtener la referencia del SearchView.
Una vez obtenido el objeto SearchView, le seteamos el listener mediante el método *setOnQueryTextListener*.

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {

	getMenuInflater().inflate(R.menu.menu_main, menu);

	MenuItem searchItem = menu.findItem(R.id.campo_buscar);
```

```
        SearchView searchView = (SearchView)
MenuItemCompat.getActionView(searchItem);

        // seteamos listener con “this”
        searchView.setOnQueryTextListener(this);

        return true;
}
```

7.3 WebView

El widget WebView nos permite visualizar contenido HTML y ejecutar JS dentro del contenedor, siendo ideal para representar páginas web ya sea de forma local, u obtenidas desde Internet.

Existen tres formas de cargar un contenido en un WebView:

- Desde una URL publicada en Internet.
- Desde una página web local de nuestra aplicación, alojada en assets.
- Desde un String con contenido HTML.

Para colocar este objeto en nuestro Layout, procedemos de igual forma que para colocar un Botón o un campo de texto:, asignándole un id, para luego poder obtener la referencia de dicho objeto desde nuestra Activity, utilizando el método findviewById.

```
WebView webview = (WebView) findViewById(R.id.webView);
```

Cargando una página web desde Internet

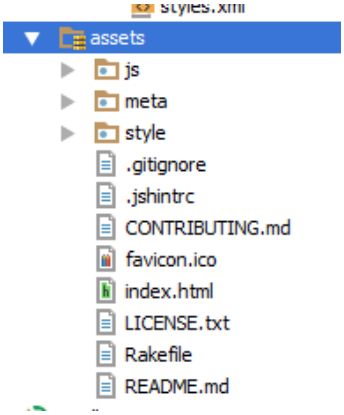
```
// Habilitamos JS
WebSettings webSettings = webview.getSettings();
webSettings.setJavaScriptEnabled(true);

// cargamos pagina
webview.setWebViewClient(new WebViewClient());
webview.loadUrl("http://www.lslutnfra.com");
```

Antes de cargar la página, debemos habilitar JS y asignar un WebViewClient al objeto WebView, de no hacerlo, la URL se abrirá en el navegador del dispositivo.

Cargando una página local en assets

Bajaremos el juego 2048 el cual está escrito en JS y copiaremos los archivos del mismo en la carpeta assets de nuestro proyecto (para crear esta carpeta hacemos click derecho sobre nuestro proyecto->new->Folder->Assets Folder)

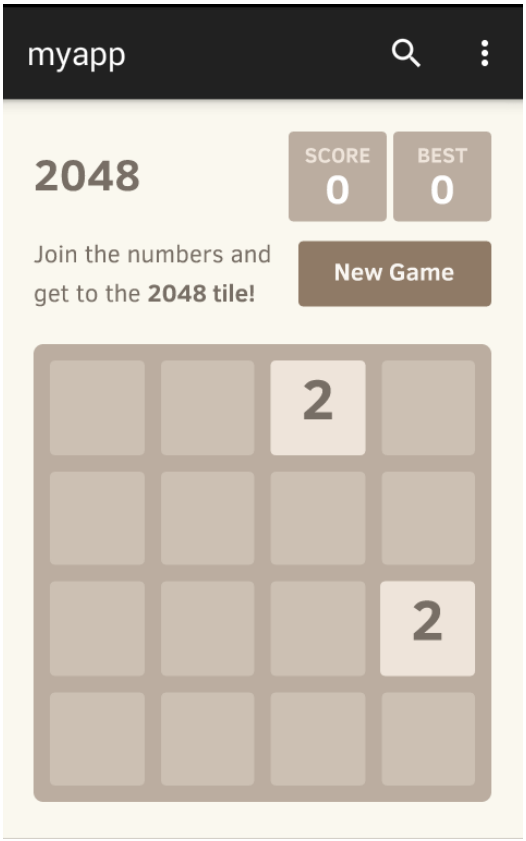
https://github.com/gabrielecirulli/2048



Luego cambiamos la línea en donde indicábamos la URL externa, por una que haga referencia a una dirección de contenido interno:

```
webview.loadUrl("file:///android_asset/index.html");
```

De esta manera se cargará la página local:

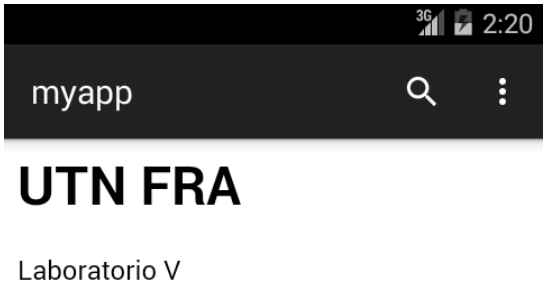


Cargando contenido HTML desde un String

Para cargar un String con contenido HTML, utilizamos el método loadData, que además del string recibe otro string con parámetros de configuración del contenido HTML a representar, entre ellos, el tipo de contenido y el encoding:

```
webview.loadData("<h1>UTN FRA</h1> <p>Laboratorio V</p>",
```

```
                                                    "text/html charset=utf-
8",null);
```



8.1 Diálogos

Los diálogos son ventanas que le aparecen al usuario sobre la pantalla activa, (igual que los menúes contextuales) pero que permiten mostrar cualquier tipo de contenido. La analogía con un sistema operativo de escritorio, sería las ventanas de alertas o información, donde el usuario puede elegir "Aceptar" ,"Cancelar" , "Si", "No", etc.

Un aspecto importante a tener en cuenta con los diálogos en Android, es que son asincrónicos, es decir, cuando se dispara el diálogo, el código no queda bloqueado, como pasa en la mayoría de los lenguajes con programas para escritorio, sino que el programa que dispara la ventana sigue corriendo, y éste debe encargarse de capturar el evento cuando el diálogo es cerrado.

Los pasos para construir un diálogo son los siguientes:

- Crear un objeto Builder
- Setear cantidad de botones, y el contenido de la ventana.
- Setear los listeners para los botones.
- Decirle al Builder que cree el diálogo
- Crear una clase que herede de FragmentDialog
- Sobreescribir el método onCreateDialog y devolver el objeto AlertDialog creado con el builder.

Ejemplo:

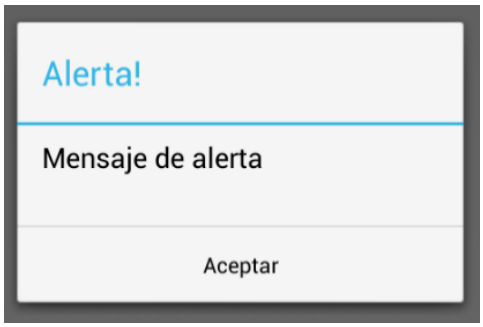
Primero, definimos una clase que funcionará como listener de los botones que aparezcan en el diálogo. Para ello, creamos una clase llamada "ListenerAlert" e implementamos la interfaz "onClickListener" correspondiente al Package "android.content.DialogInterface"

```
import android.content.DialogInterface;
import android.content.DialogInterface.OnClickListener;
import android.util.Log;

public class ListenerAlert implements OnClickListener{

	@Override
	public void onClick(DialogInterface dialog, int which) {
		Log.d("dialog","Click!");
	}
}
```

Luego, creamos una clase "MiDialogo" y sobreescribimos el método "onCreateDialog"

```
public class MiDialogo extends DialogFragment {

	@Override
	public Dialog onCreateDialog(Bundle savedInstanceState) {
		AlertDialog.Builder builder = new

AlertDialog.Builder(getActivity());
		builder.setTitle("Alerta!");
		builder.setMessage("Mensaje de alerta");

		ListenerAlert l = new ListenerAlert();
		builder.setPositiveButton("Aceptar", l);

		// Creamos el dialogo
		AlertDialog ad = builder.create();
		return ad;
	}
```

Dentro del método, creamos un objeto del tipo Builder (del package "android.app.AlertDialog") en el cual seteamos las características del diálogo, como el título, el mensaje, y los botones que contendrá. También podemos observar que creamos un objeto de nuestra clase ListenerAlert, y se lo pasamos al botón, para poder recibir el evento de click del mismo.



Además de "setPositiveButton()", existen dos botones más que pueden agregarse:

- setNegativeButton()
- setNeutralButton()

Para identificar qué botón se presionó, en el listener, existe el parámetro "which":

```
public void onClick(DialogInterface dialog, int which)
```

El cual tiene un valor distinto según cual de los tres botones se presionó, los valores están definidos en la clase AlertDialog:

- `AlertDialog.BUTTON_NEGATIVE`
- `AlertDialog.BUTTON_POSITIVE`

- AlertDialog.BUTTON_NEUTRAL

Para lanzar el dialogo desde la Activity, creamos nuestro objeto "MiDialogo" y le ejecutamos el método "show" el cual recibe como parámetros una instancia de "FragmentManager" y un nombre.

```
MiDialogo dialog = new MiDialogo();
dialog.show(getSupportFragmentManager(), "dialogo");
```

Lista de items en dialogo

Es posible mostrar una lista de ítems en el diálogo ejecutando el método "setItems" al objeto builder, en vez de "setMessage".El segundo parámetro de setItems es un listener.

```
builder.setTitle("Elija una opcion");
String[] items = new String[]{"item1","item2","item3","item4"};
builder.setItems(items, this);
```

Diálogos con contenido personalizado

Es posible definir un archivo de layout, al igual que se hace para el contenido de una Activity, y setear dicho archivo, a la ventana de diálogo, de modo que es posible definir cualquier cantidad de objetos en la ventana, con cualquier disposición.

Ejemplo:
Primero, creamos un archivo de layout con el nombre *layout_dialogo*, en este archivo, tendremos una imagen y un campo de texto para ingresar un valor:

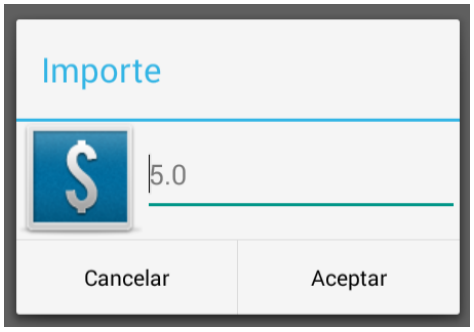
```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
android:layout_width="match_parent"
android:layout_height="match_parent"
android:gravity="center_vertical"
android:orientation="horizontal">
```

```
        <ImageView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:src="@drawable/pesos_icon"
        android:id="@+id/imageView" />

        <EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="5.0"
        android:id="@+id/editText" />
</LinearLayout>
```

Luego, en la Activity, creamos un objeto del tipo LayoutInflater, el cual nos permitirá crear un objeto View a partir del archivo xml. Una vez obtenido este objeto View, se lo seteamos al objeto builder mediante el método "setView()":

```
// cargamos la vista desde el xml
LayoutInflater li =LayoutInflater.from(getActivity());
View viewAlert = li.inflate(R.layout.layout_dialogo,null);

// cargamos view en el builder
AlertDialog.Builder builder = new
AlertDialog.Builder(getActivity());
builder.setTitle("Importe");
builder.setView(viewAlert);

// asignamos listeners
ListenerAlert l = new ListenerAlert();
builder.setPositiveButton("Aceptar", l);
builder.setNegativeButton("Cancelar", l);

// Creamos el dialogo
AlertDialog ad = builder.create();
return ad;
```

El resultado del diálogo es el siguiente:



8.2 Shared Preferences

SharedPreferences le permite al programador leer y escribir información a través de diferentes componentes y aplicaciones. Generalmente se utiliza para guardar la configuración de la aplicación que el usuario puede modificar. Cuando se utiliza este método, se genera un archivo xml el cual contiene la información cargada. Mediante SharedPreferences se podrán almacenar tipos de datos primitivos y Strings.

Este contenedor de información, puede estar asociado a un nombre arbitrario que asigna el programador (en este caso se puede acceder desde cualquier Activity), a una Activity en particular (se utilizará el nombre de la Activity para acceder), o a el contexto (se utilizará el nombre del contexto para acceder)

Escribiendo información

- Utilizando un nombre:

```
SharedPreferences prefs =
getSharedPreferences("miConfig",Context.MODE_PRIVATE);
      Editor editor = prefs.edit();
      editor.putString("key_1", "Hola mundo");
      editor.putInt("key_2", 5);
      editor.commit();
```

- Utilizando el nombre de una Activity en particular:

```
SharedPreferences prefs = this.getPreferences(Context.MODE_PRIVATE);
```

- Utilizando el nombre del Contexto:

```
SharedPreferences prefs =
PreferenceManager.getDefaultSharedPreferences(this);
```

Puede observarse que existen 3 maneras de obtener el objeto SharedPreferences el cual nos proporciona los métodos para leer y escribir información.

En el primer caso, accedemos a la información mediante un nombre, en el ejemplo "miConfig" el cual es pasado al método "getSharedPreferences()" perteneciente a la Activity junto con el parámetro que indica el tipo de acceso al archivo , en este caso "Privado", éste método nos devolverá el objeto SharedPreferences.

Los otros modos de acceder/crear el archivo SharedPreferences son los siguientes:

- **MODE_PRIVATE:** Solo esta aplicación puede acceder a la información.

- **MODE_WORLD_READABLE:** Cualquier aplicación puede leer la información.
- **MODE_WORLD_WRITEABLE:** Cualquier aplicación puede escribir información.

Una vez obtenido el objeto SharedPreferences, llamaremos al método "edit()" el cual nos devolverá un objeto del tipo Editor, el cual tendrá los métodos necesarios para escribir la información mediante pares clave-valor:

```
editor.putString("key_1", "Hola mundo");
editor.putInt("key_2", 5);
```

Por último, para que la información se guarde en el archivo, ejecutamos el método "commit()" del objeto Editor.

Leyendo información

Para recuperar la información guardada, obtenemos el objeto SharedPreferences como se vio anteriormente y luego llamaremos a los métodos "getString()", "getInt()",etc. a los cuales le pasaremos el nombre de la "clave" de la información que queremos leer:

```
String datoStr = prefs.getString("key_1", "default value");
int datoInt = prefs.getInt("key_2", 7);
```

Puede observarse que además de el nombre de la clave que corresponde con la información que se quiere leer, se le pasa un segundo valor, que es el valor que devolverá el método por defecto, si la información no se encuentra en el archivo.

8.3 Selectors y Shapes

Un Shape es una definición de una figura geométrica mediante un archivo XML ubicado en los recursos drawables, esto nos permite utilizar a estos archivos como cualquier archivo de imagen colocado en dicha carpeta. Esto quiere decir que cuando creemos un archivo XML para un Shape, se generará en la clase R.drawable el atributo correspondiente para poder ser referenciado desde nuestro código Java.

Los Shapes se utilizan comúnmente para definir el atributo background de cualquier View. A continuación mostraremos la definición de un rectángulo de color rojo:

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

       <corners android:radius="1dp"/>

       <solid android:color="#FFFF0000" />
```

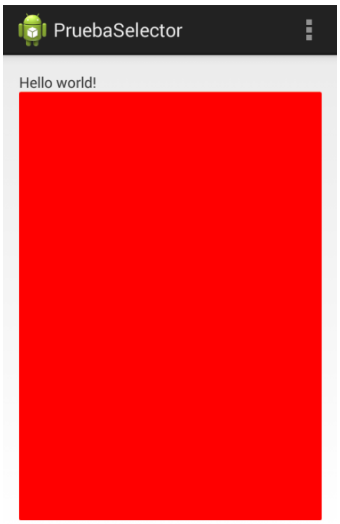
```
</shape>
```

En este caso definimos que la figura sea un rectángulo mediante la propiedad android:shape, le indicamos que posee puntas redondeadas, y un color de fondo Rojo. Para más información sobre todas las propiedades que pueden aplicarse a la definición de la figura, puede leerse la documentación de Android Developers:

http://developer.android.com/guide/topics/resources/drawable-resource.html
Si guardamos este archivo con el nombre "fondo_rojo.xml" en la carpeta "drawables", podremos referirnos a este archivo como si fuera una imagen, por ejemplo para colocar de fondo de un layout:

```
<LinearLayout
      android:orientation="horizontal"
      android:layout_width="fill_parent"
      android:layout_height="fill_parent"
      android:layout_below="@+id/textView"
      android:layout_alignParentLeft="true"
      android:background="@drawable/fondo_rojo"
      android:layout_alignParentStart="true"
      android:id="@+id/v">
</LinearLayout>
```

Aquí se observa que asigamos a la propiedad android:background en valor @drawable/fondo_rojo. De esta forma el layout tendrá asignado como background nuestro shape:



Asignación de diferentes imágenes según estado: Selectors

Si la View donde colocamos el fondo es "cliqueable", es decir que tiene la posibilidad de que el usuario haga un click sobre la misma (un Button por ejemplo, o un layout si le habilitamos la opción) probablemente querramos que al hacer click sobre la misma, el color de fondo cambie.

No es necesario escribir por código este comportamiento, ya que puede definirse un archivo XML también ubicado en drawables, el cual asignará la propiedad drawable según el estado en que se encuentra la View.

A continuación escribiremos el archivo fondo.xml el cual guardaremos junto con fondo_rojo.xml y también supondremos la existencia del archivo fondo_amarillo.xml, el cual es idéntico a fondo_rojo pero con otro color de fondo.

Archivo fondo.xml:

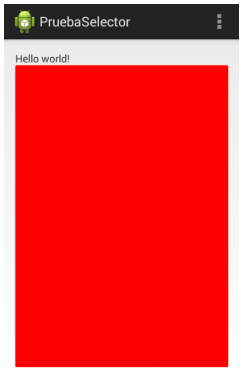
```
<?xml version="1.0" encoding="utf-8"?>
<selector xmlns:android="http://schemas.android.com/apk/res/android">

	<item  android:drawable="@drawable/fondo_rojo"
		android:state_selected="true">
	</item>
	<item  android:drawable="@drawable/fondo_amarillo"
		android:state_pressed="true">
	</item>
	<item  android:drawable="@drawable/fondo_rojo">
	</item>

</selector>
```

Como se observa, dentro del selector tenemos ítems los cuales definen un estado: state_selected, state_pressed, y normal (sin atributo)

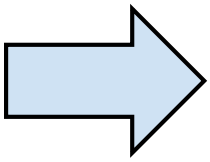
Cada ítem con su estado, tiene asignado una propiedad drawable diferente (puede ser un shape o un archivo de imagen normal). Este archivo fondo.xml, al igual que los shapes, serán tratados como si fueran un archivo de imagen en drawables, por esta razón, puede asignarse como background a cualquier view.

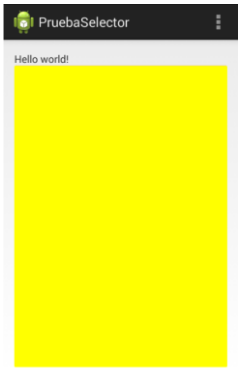
Modificaremos el ejemplo anterior del Layout, y ahora asignaremos este selector como fondo, también habilitaremos la posibilidad de hacer click sobre el layout, mediante el atributo android:clickeable:

```
<LinearLayout
	android:orientation="horizontal"
	android:layout_width="fill_parent"
	android:layout_height="fill_parent"
	android:layout_below="@+id/textView"
	android:layout_alignParentLeft="true"
	android:background="@drawable/fondo"
	android:clickable="true"
	android:layout_alignParentStart="true"
	android:id="@+id/v">
</LinearLayout>
```



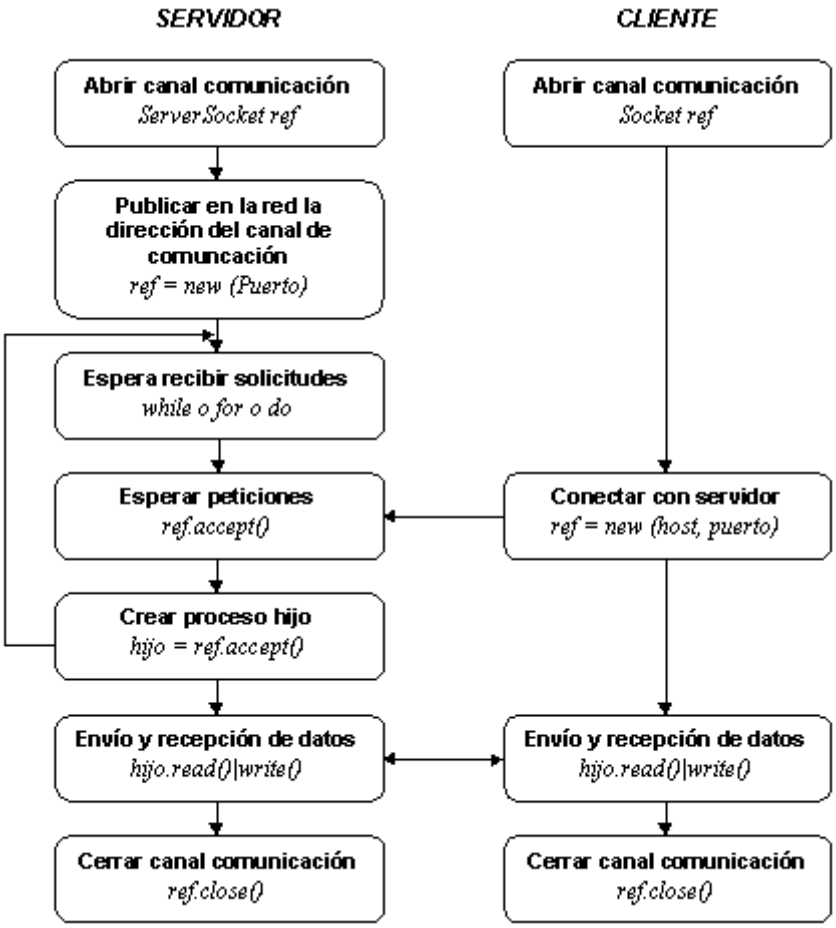
Ahora el hacer click sobre el layout, cambiará en color de fondo:



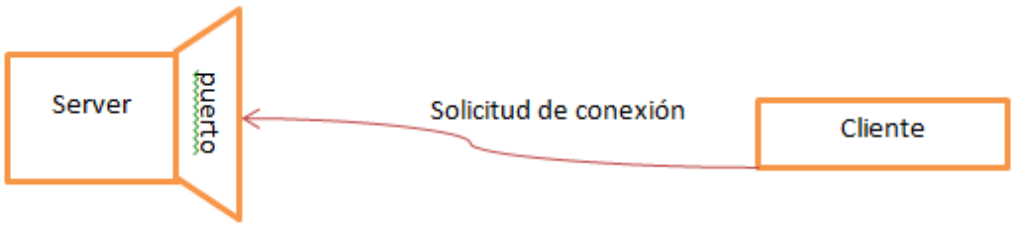


9.1 Sockets

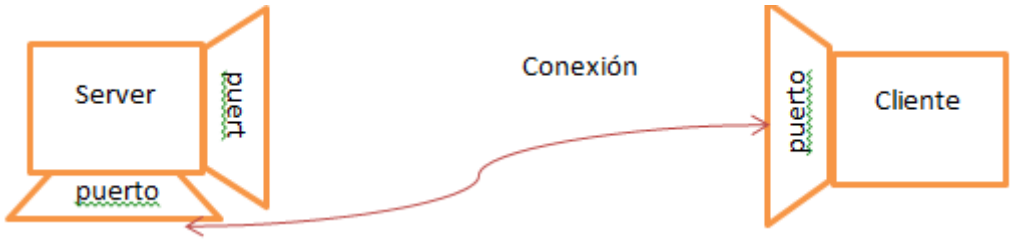
Los sockets son un sistema de comunicación entre procesos de diferentes máquinas en una red. Más exactamente, un socket es un punto de comunicación por el cual un proceso puede emitir y recibir información.
Los sockets son capaces de utilizar el protocolo de streams TCP (Transfer Control Protocol) y el datagramas UDP (User Datagram Protocol).



Normalmente, un servidor se ejecuta sobre una computadora específica y tiene un socket que responde a un puerto específico. El servidor únicamente espera, escuchando a través de un socket a que un cliente haga una petición.
En el lado del cliente, éste conoce el nombre del host de la máquina en la cual el servidor se está ejecutando y el número de puerto en el cual está conectado. Para realizar una petición de conexión, el cliente intenta encontrar al servidor en la máquina servidora en el puerto especificado.



Si todo va bien, el servidor acepta la conexión. Además de aceptar, el servidor obtiene un nuevo socket sobre un puerto diferente. Esto se debe a que necesita un nuevo socket (y, en consecuencia, un número de puerto diferente) para seguir escuchando peticiones de conexión, mientras atiende las necesidades del cliente que se conectó.

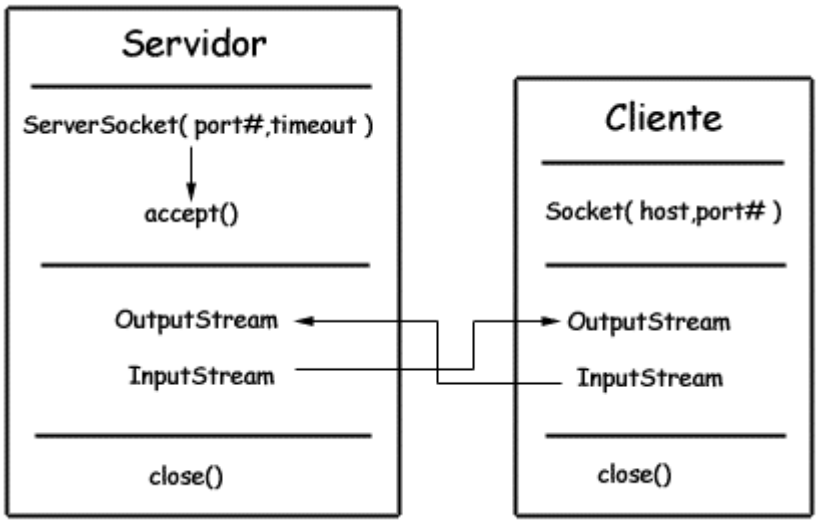


Por la parte del cliente, si la conexión es aceptada, un socket se crea de forma satisfactoria y puede usarse para comunicarse con el servidor. Es importante darse cuenta que el socket en el cliente no está utilizando el número de puerto usado para realizar la petición al servidor. En lugar de éste, el cliente asigna un número de puerto local en la máquina en la cual está siendo ejecutado. Ahora el cliente y el servidor pueden comunicarse escribiendo o leyendo en o desde sus respectivos sockets.

Sockets en JAVA

Dentro del paquete java.net encontramos la clase Socket y ServerSocket, las cuales vamos a utilizar para armar una conexión bidireccional entre dos programas dentro de una red.
El modelo más simple es:

- El servidor establece un puerto y espera durante un cierto tiempo (timeout en segundos), a que el cliente establezca la conexión. Cuando el cliente solicite una conexión, el servidor la autorizará con el método accept().
- El cliente establece una conexión con la máquina host a través del puerto que se designe.
- El cliente y el servidor se comunican mediante objetos del tipo InputStream y OutputStream



Una implementación de un servidor podría ser la siguiente:

```
try {
      ServerSocket serverSocket = new ServerSocket(4097);
      while(true)
      {
          System.out.println("Esperando conexion...");
          Socket clientSocket = serverSocket.accept();

          System.out.println("Se establecio conexion
                       con:"+clientSocket.getRemoteSocketAddress());

          ThreadParaCliente tc = new ThreadParaCliente(clientSocket);
          tc.start();
      }
 } catch (IOException e) {
      e.printStackTrace();
 }
```

El servidor habilita el puerto 4097 para recibir clientes y empieza a escuchar, nuestro código se va a detener en el método accept() hasta que un cliente establezca una conexión.
Cuando se acepta la conexión, se lanza un hilo para mantener el diálogo en otro proceso y poder seguir escuchando a nuevos clientes.

El código del método run del Thread podría ser el siguiente:

```
public ThreadParaCliente (Socket clientSocket)
{
      this.clientSocket = clientSocket;
}

public void run() {

      try {
          InputStream is = clientSocket.getInputStream();
          InputStreamReader isr = new InputStreamReader(is);
          BufferedReader br = new BufferedReader(isr);

          OutputStream os = clientSocket.getOutputStream();
          OutputStreamWriter osw = new OutputStreamWriter(os);
          BufferedWriter bw = new BufferedWriter(osw);
          bw.write("Conexion establecida\n");
          bw.flush();

          while(true)
          {
            //System.out.println("espero mensaje...");
            String msg = br.readLine();
            if(msg==null)
                  break;

            System.out.println(
                clientSocket.getRemoteSocketAddress()+">"+msg);
            bw.write("ECHO>"+msg+"\n");
            bw.flush();
         }

      } catch (IOException e) {
          e.printStackTrace();
      }
}
```

Obtenemos las instancias de los Stream del cliente y comenzamos a dialogar. Todo lo que escribamos en el Output lo leerá el cliente y todo lo que el cliente escriba en el Input lo leerá el servidor.

Del lado del cliente, lanzamos en un thread la conexión creando un objeto Socket con la IP y puerto del Server, luego obtenemos los Streams que nos permitirán leer y escribir contra el Server:

```
@Override
public void run() {

	try {
	clientSocket = new Socket("192.168.1.101",4097);

	OutputStream os = clientSocket.getOutputStream();
	OutputStreamWriter osw = new OutputStreamWriter(os);
	bw = new BufferedWriter(osw);



	InputStream is = clientSocket.getInputStream();
	InputStreamReader isr = new InputStreamReader(is);
	BufferedReader br = new BufferedReader(isr);

	while(true) {
		String msgStr = br.readLine();
		if(msgStr==null)
			break;
		Log.d("t", "Llego:" + msgStr);
		Message m = new Message();
		m.obj = msgStr;
		h.sendMessage(m);
	}

	} catch (IOException e) {
	e.printStackTrace();
	}

	try {
	clientSocket.close();
	} catch (IOException e) {
	e.printStackTrace();
	}
}

public void sendMsg(String s)
{
	try {
	if(bw!=null) {
		bw.write(s + "\n");
		bw.flush();
	}
	} catch (IOException e) {
	e.printStackTrace();
	}
```

```
}
```

10.1 Navigation Drawer

Este componente nos permitirá generar un contenido que aparece desde la izquierda de la pantalla mediante un gesto de arrastre desde el borde izquierdo hacia la derecha o presionando el botón Home en la ToolBar. Generalmente el contenido que se muestra es una lista de opciones creada con un archivo XML de menú.

Para utilizar este componente, deberemos agregar un “DrawerLayout” en el archivo xml de layout de la Activity donde queremos que exista el drawer. Ejemplo:

```
<android.support.v4.widget.DrawerLayout
       android:id="@+id/drawer_layout"
       android:background="#FF000000"
       android:layout_width="match_parent"
       android:layout_height="match_parent">
       <!-- The main content view -->
       <LinearLayout
              android:id="@+id/contenedor"
              android:layout_width="match_parent"
              android:layout_height="match_parent"
              android:orientation="vertical" >
              <ImageView
              android:layout_width="match_parent"
              android:src="@drawable/sistema_solar"
              android:layout_height="wrap_content" />
       </LinearLayout>
       <!-- Menú Deslizante -->
       <android.support.design.widget.NavigationView
       android:layout_width="match_parent"
       android:layout_height="match_parent"
       android:id="@+id/navigation_view"
       android:layout_gravity="start"
       android:fitsSystemWindows="true"
       app:headerLayout="@layout/nav_header"
       app:menu="@menu/nav_menu" />
</android.support.v4.widget.DrawerLayout>
```

En este caso, el contenido de la pantalla es un LinearLayout con un ImageView dentro, este Layout (con id "contenedor") y el DrawerLayout, deben tener las propiedades "match_parent" tanto en el alto como en el ancho.
Por otro lado, debajo, y siempre dentro del tag "DrawerLayout", tenemos la View que se mostrará a la izquierda de la pantalla cuando el usuario haga el gesto. En este ejemplo, se utilizó una NavigationView provista por la biblioteca design support.

NavigationView

Esta View está especialmente diseñada para componer el menú lateral del Navigation Drawer. Los dos atributos principales de esta View son:

- **app:headerLayout** Aquí indicamos un archivo XML de layout que se colocará en el header de la barra lateral.
- **app:menu** Aquí indicamos un archivo XML de menú mediante el cual se mostrarán los ítems en la barra lateral.

Para utilizar esta View, deberemos incluir la biblioteca design support en el archivo build.gradle de nuestro módulo app:

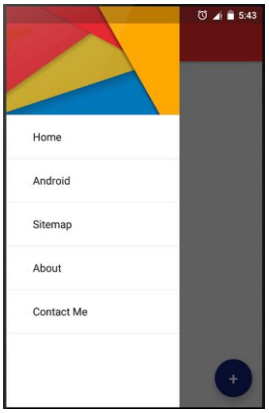
```
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    compile 'com.android.support:appcompat-v7:22.2.1'
    compile 'com.android.support:design:22.2.1'
}
```

Creando el Header

Crearemos un archivo XML de layout llamado nav_header.xml en el cual pondremos como header una imagen:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
android:orientation="vertical"
android:layout_width="match_parent"
android:layout_height="wrap_content">
       <ImageView
       android:layout_width="match_parent"
       android:scaleType="centerCrop"
       android:src="@drawable/sistema_solar"
       android:layout_height="100dp" />
</LinearLayout>
```

Este es un header típico para Navigation Drawer:



Creando los items

Para crear los ítems que formarán la lista, deberemos crear un archivo XML de menú. Cada ítem podrá contener un ícono si se desea, al igual que un ítem de menú.
Creamos el archivo nav_menu.xml con el siguiente contenido:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
<group
android:checkableBehavior="single">
<item
android:id="@+id/drawer_item_mercurio"
android:checked="true"
android:title="Mercurio"/>
<item
android:id="@+id/drawer_item_venus"
android:title="Venus"/>
<item
android:id="@+id/drawer_item_tierra"
android:title="Tierra"/>
<item
android:id="@+id/drawer_item_marte"
android:title="Marte"/>
<item
android:id="@+id/drawer_item_jupiter"
android:title="Jupiter"/>
</group>
</menu>
```

Agregando botón menú para lanzar el drawer
Si ejecutamos este ejemplo, notaremos que el drawer aparece al realizar el gesto, pero no presionando el botón en la ToolBar, ya que para ésto necesitaremos utilizar un objeto del tipo "ActionBarDrawerToogle"

Agreguemos en el onCreate de nuestra Activity, el siguiente código:

```
mDrawerLayout = (DrawerLayout) findViewById(R.id.drawer_layout);
mDrawerToggle = new ActionBarDrawerToggle(
                    this,
                    mDrawerLayout,
                    R.string.drawer_open,
                    R.string.drawer_close
                    );
```

```
mDrawerLayout.setDrawerListener(mDrawerToggle);
getSupportActionBar().setDisplayHomeAsUpEnabled(true);
getSupportActionBar().setHomeButtonEnabled(true);
```

En este programa, obtenemos el DrawerLayout que está en pantalla y le seteamos un "DrawerListener" mediante el método "setDrawerListener". También habilitamos el uso del botón Home de la Action Bar mediante "setDisplayHomeAsUpEnabled" y "setHomeButtonEnabled"

La porción más importante de este ejemplo es la creación del objeto ActionBarDrawerToogle, este objeto es el que le pasamos al DrawerLayout como listener.

El primer argumento en el contructor de este objeto es la Activity que contine al Drawer, luego el objeto DrawerLayout que está en pantalla, y por último el constructor recibe 2 recursos del tipo string, que es la descripción del drawer cuando está abierto y cerrado.

Ahora deberemos capturar el evento de click sobre el botón home, para ello en nuestra Activity sobreescribimos el método "onOptionsItemSelected" (el mismo evento que para los ítems de menú)

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
        if (mDrawerToggle.onOptionsItemSelected(item)) {
                return true;
        }
        return super.onOptionsItemSelected(item);
}
```

Al recibir el evento, llamamos al método "onOptionsItemSelected del objeto "ActionBarDrawerToogle" el cual devolverá true si el evento era el botón de home. De esta manera el drawer aparecerá mediante una animación, al presionarse el botón.

También deberemos sobreescribir dos métodos más en la Activity para notificar cambios de estado de la Activity, al objeto "ActionBarDrawerToogle" :

```
@Override
protected void onPostCreate(Bundle savedInstanceState) {
        super.onPostCreate(savedInstanceState);
        // Sync the toggle state after onRestoreInstanceState has occurred.
        mDrawerToggle.syncState();
}

@Override
public void onConfigurationChanged(Configuration newConfig) {
        super.onConfigurationChanged(newConfig);
        mDrawerToggle.onConfigurationChanged(newConfig);
}
```

De esta manera, ahora aparecerá habilitado el botón Home.


2:13
EjemploNavDrawer
2:14
EjemploNavDrawer

Selección de ítems

Para detectar el ítem de la lista que se seleccionó, utilizaremos un listener sobre el NavigationView. En este ejemplo, haremos que el listener sea nuestra Activity:

```
navigationView = (NavigationView) findViewById(R.id.navigation_view);
navigationView.setNavigationItemSelectedListener(this);
```

En la Activity, deberemos implementar la interface OnNavigationItemSelectedListener, la cual nos obligará a implementar el método onNavigationItemSelected en el cuál podremos detectar qué ítem fue seleccionado:

```
@Override
public boolean onNavigationItemSelected(MenuItem menuItem) {
      menuItem.setChecked(true);
      switch(menuItem.getItemId()){
          case R.id.drawer_item_mercurio:
            Toast.makeText(this,"Mercurio",Toast.LENGTH_SHORT).show();
            break;
            //...
      mDrawerLayout.closeDrawers();
```

Mediante setChecked, hacemos que el ítem quede selecionado en la lista.

10.2 Fragments

Muchas veces necesitamos cambiar completamente el contenido de la pantalla de nuestra Activity según la opción que el usuario presiona en nuestro menú generado con drawer.

Para ello, utilizaremos Fragments, definiremos una clase que herede de Fragment y un archivo xml de layout por cada contenido que queramos mostrar, cada Fragment hará un inflate su archivo xml de layout. Recordemos que un Fragment puede pensarse como una Activity, con la característica de que la Activity que lo contiene, lo puede mostrar y ocultar bajo cierta lógica.

Definimos 2 archivos de layout para cada **tipo de contenido** que queremos mostrar en la pantalla principal:

layout_contenido1 layout_contenido2

Como se observa en las figuras, en el archivo layout_contenido1.xml, colocamos un título, un texto, y luego una imagen:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
android:orientation="vertical" android:layout_width="match_parent"
android:background="#FF000000"
android:layout_height="match_parent">

<TextView
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:textAppearance="?android:attr/textAppearanceLarge"
android:text="@string/titulo_mercurio"
android:id="@+id/txtTitulo"
android:textColor="#FFFFFFFF"
android:layout_gravity="center_horizontal" />

<TextView
android:layout_margin="20dp"
android:layout_gravity="center_horizontal"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:textAppearance="?android:attr/textAppearanceSmall"
android:text="@string/info_mercurio"
android:textColor="#FFFFFFFF"
android:id="@+id/txtInfo" />

<ImageView
android:layout_width="match_parent"
android:layout_height="300dp"
android:id="@+id/imgPlaneta"
android:src="@drawable/mercurio"
android:layout_gravity="center_horizontal" />
</LinearLayout>
```

Mientras que en el archivo layout_contenido2.xml, colocamos primero el título, luego la imagen y por último el texto:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
android:orientation="vertical" android:layout_width="match_parent"
android:background="#FF000000"
android:layout_height="match_parent">

<TextView
```

```
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:textAppearance="?android:attr/textAppearanceLarge"
android:text="@string/titulo_mercurio"
android:id="@+id/txtTitulo"
android:textColor="#FFFFFFFF"
android:layout_gravity="center_horizontal" />

<ImageView
android:layout_width="match_parent"
android:layout_height="300dp"
android:id="@+id/imgPlaneta"
android:src="@drawable/mercurio"
android:layout_gravity="center_horizontal" />

<TextView
android:layout_margin="20dp"
android:layout_gravity="center_horizontal"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:textAppearance="?android:attr/textAppearanceSmall"
android:text="@string/info_mercurio"
android:textColor="#FFFFFFFF"
android:id="@+id/txtInfo" />
</LinearLayout>
```

Utilizaremos los dos layouts como “modelos” para cargarlos con la información de cada planeta según corresponda. Esta lógica la resolveremos en dos clases que serán los Fragments que muestren la información utilizando un layout y el otro respectivamente.

Definimos dos clases: “Contenido1” y “Contenido2” ambas deben heredar de fragment y reescribir el método “onCreateView” donde se hace un inflate del archivo xml de layout correspondiente y se devuelve la view obtenida. Puede pensarse en este método como el “onCreate” de la Activity.

Ejemplo de clase “Contenido1”

```
public class Contenido1 extends Fragment{

@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container,
                                                    Bundle savedInstanceState) {

    View v = inflater.inflate(R.layout.layout_contenido1, container,false);

    ImageView img = (ImageView)v.findViewById(R.id.imgPlaneta);
    TextView txtTitulo = (TextView)v.findViewById(R.id.txtTitulo);
    TextView txtInfo = (TextView)v.findViewById(R.id.txtInfo);

    Bundle args = getArguments();
    int idPlaneta = args.getInt("idPlaneta");
    switch(idPlaneta)
    {
        case 0:
        img.setImageResource(R.drawable.mercurio);
        txtTitulo.setText(getText(R.string.titulo_mercurio));
        txtInfo.setText(getText(R.string.info_mercurio));
        break;
```

```
        case 1:
        img.setImageResource(R.drawable.venus);
        txtTitulo.setText(getText(R.string.titulo_venus));
        txtInfo.setText(getText(R.string.info_venus));
        break;
        case 2:
        img.setImageResource(R.drawable.tierra);
        txtTitulo.setText(getText(R.string.titulo_tierra));
        txtInfo.setText(getText(R.string.info_tierra));
        break;
        case 3:
        img.setImageResource(R.drawable.marte);
        txtTitulo.setText(getText(R.string.titulo_marte));
        txtInfo.setText(getText(R.string.info_marte));
        break;
        case 4:
        img.setImageResource(R.drawable.jupiter);
        txtTitulo.setText(getText(R.string.titulo_jupiter));
        txtInfo.setText(getText(R.string.info_jupiter));
        break;
    }
    return v;
}
```

Es importante remarcar el uso del método getArguments el cual devuelve un objeto Bundle, que adentro contiene, con la clave "idPlaneta" un número que le indica al fragment qué información mostrar. Este Bundle lo generaremos en la Activity, en el evento de click sobre un item de la lista, al generar el Fragment y lanzarlo.

Para la clase "Contenido2" se hace lo mismo y se reemplaza "layout_contenido1" por "layout_contenido2".

En nuestra Activity, no debemos cambiar demasiado con respecto al ejemplo anterior, solamente debemos llamar al método setContenido cuando se presiona un ítem, en el lugar donde mostrabamos un Toast.

En nuestra Activity, no debemos cambiar demasiado con respecto al ejemplo anterior, solamente debemos llamar al método setContenido cuando se presiona un ítem, en el lugar donde mostrabamos un Toast.

Por último escribimos el método "setContenido" en nuestra Activity:

```
private void setContenido(int numeroContenido)
{
        Fragment fragment;

        Bundle bundle = new Bundle();
        bundle.putInt("idPlaneta",numeroContenido);

        if(numeroContenido%2==0)
                fragment = new Contenido1();
        else
                fragment = new Contenido2();

        fragment.setArguments(bundle);
```

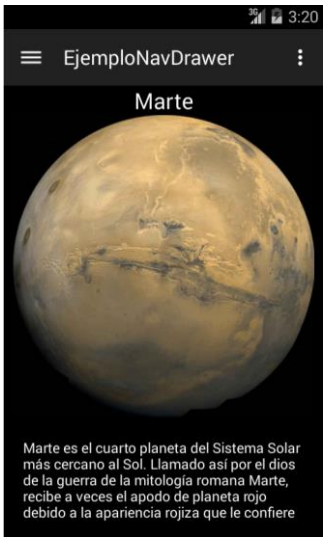
```
        FragmentManager fragmentManager = getSupportFragmentManager();
        fragmentManager.beginTransaction()
                                    .replace(R.id.contenedor, fragment)
                                    .commit();
}
```

Como se observa en el código, según el argumento se genera el Fragment Contenido1 o Contenido2 (según si es par o impar), y luego mediante el objeto FragmentManager, se hace un "replace" del fragment contenido en la view "R.id.contenedor" por el que creamos.

Si observamos el archivo xml de layout de la Activity, el contenedor es el LinearLayout utilizado como contenedor de lo que se muestra en la pantalla, el cual deberá estar vacío, de modo que debemos comentar el ImageView que habíamos colocado con anterioridad:

```
<!-- The main content view -->
<LinearLayout
android:id="@+id/contenedor"
android:layout_width="match_parent"
android:layout_height="match_parent"
android:orientation="vertical" >

<!-- <ImageView
android:layout_width="match_parent"
android:src="@drawable/sistema_solar"
android:layout_height="wrap_content" /> -->

</LinearLayout>
```

Se aclara que de esta forma, el fragment se esta creando nuevamente cada vez que el usuario elige una opción de la ListView, se podrían tratar dichos fragments



como singletons para reutilizar las mismas instancias ya generadas.

Conclusiones

En este ejemplo se pretende demostrar que cada contenido, que se muestra en pantalla al hacer click sobre un ítem de la lista del Navigation Drawer, puede ser un fragment diferente, o puede ser el mismo fragment, así como también que un mismo fragment puede trabajar con muchos archivos de layout, decidiendo cuál se carga, o cada fragment puede trabajar con un archivo de layout propio para ese fragment.

La anterior aclaración surge de la posibilidad de resolver el mismo ejemplo utilizando una sola clase "Contenido" que herede de fragment, y que seleccione qué layout cargar según el argumento idPlaneta.

La idea principal de un fragment es que tenga un solo archivo de layout (en la mayoría de los casos, los fragments serán completamente diferentes entre sí) y que maneje la lógica de la representación en dicho layout.