2ºDAM - Programación multimedia y dispositivos móviles

Tema 1 - **Características específicas y entornos de desarrollo para la programación en dispositivos móviles**

1. Limitaciones que plantea la ejecución de aplicaciones en los dispositivos móviles

Piensa en tres limitaciones que tienen los dispositivos móviles de cara a la programación

1. Limitaciones que plantea la ejecución de aplicaciones en los dispositivos móviles

Algunas de estas restricciones a la que nos podemos enfrentar en estos dispositivos son:

- Suministro de energía limitado (normalmente dependiente de baterías).
- Procesadores con capacidad de cómputo reducida. Suelen tener una baja frecuencia de reloj por la necesidad de ahorrar energía. En algunos casos, por ejemplo, podrían no disponer de la capacidad de cálculos en punto flotante.
- Poca memoria principal (RAM).
- Almacenamiento de datos persistente reducido (pequeña memoria flash interna, tarjetas SD, etc.).
- Conexión a algún tipo de red intermitente y con ancho de banda limitado.
- Pantallas de reducidas dimensiones.
- Teclados con funcionalidad muy básica y muy pequeños.

1. Limitaciones que plantea la ejecución de aplicaciones en los dispositivos móviles

Este tipo de restricciones, y algunas otras que dependerán de cada dispositivo en concreto, habrán de ser tenidas muy en cuenta a la hora del análisis y diseño de una aplicación "**móvil**", pues no podemos pretender, que esa aplicación pueda contener la misma funcionalidad, que la que podemos encontrar habitualmente en un programa que es ejecutado en un ordenador de sobremesa o un portátil.

Por otro lado, no todo va a ser restricciones. También habrá que tener en consideración que esta tecnología va a aportar una serie de **ventajas** muy importantes: movilidad, poco peso, pequeño tamaño, facilidad para el transporte, conectividad a diversos tipos de redes de comunicaciones (mensajería SMS y MMS; voz; Internet; Bluetooth; infrarrojos; radiofrecuencia, etc.). Ésas serán las ventajas que podrás explotar en tus aplicaciones.

2. Tecnologías disponibles.

Cuando vas a desarrollar una aplicación para un dispositivo móvil, algunas de las primeras preguntas que te puedes hacer son:

- ¿Sobre qué tipos de dispositivos móviles se pueden hacer programas? ¿Sobre qué tipo de hardware se puede programar?
- ¿Qué sistema operativo puede llevar ese hardware?
- ¿Qué plataformas de desarrollo existen para desarrollar sobre ese hardware y ese sistema operativo? ¿con qué lenguajes puedo programar? ¿qué herramientas (compiladores, bibliotecas, entornos, etc.) hay disponibles?

Piensa en una respuesta para cada pregunta planteada arriba

2. Tecnologías disponibles.

Las respuestas a este tipo de preguntas pueden ser múltiples y muy variadas:

Respecto al hardware, te puedes encontrar con teléfonos móviles (smartPhones), PDA y con otros dispositivos. Entre los principales fabricantes de teléfonos móviles se encuentran **Nokia**, **Samsung**, **Apple**, **Motorola**, **LG**, **Sony Ericsson**,etc. Entre los principales fabricantes de PDA se encuentran HP, **HTC**, **Motorola**, **Samsung** o **LG**.

En cuanto a los sistemas operativos, dependiendo del hardware habrá sistemas diseñados para unos u otros dispositivos. Los hay basados en **Microsoft Windows**, en **Linux**, y en **MAC OS X**, así como otros totalmente originales y desarrollados específicamente para estos nuevos tipos de dispositivos. Entre los más populares se encuentran **Symbian OS**, **Android**, **iOS**, **Blackberry OS** y **Windows Phone**.

2. Tecnologías disponibles.

Si lo que deseas es conocer algo acerca de las plataformas de desarrollo disponibles para cada entorno (hardware y/o sistema operativo), podemos hablar de **Java ME**, Windows Mobile SDK, Maemo SDK, o bien de **IDE** como Microsoft Visual Studio, CodeWarrior, Eclipse, Netbeans.

Si te refieres a lenguajes de programación, normalmente te encontrarás con lenguajes que son ya viejos conocidos para otras plataformas, como pueden ser las aplicaciones de escritorio para los PCs o las aplicaciones web (**Java, C#, C**, etc.).

3. Entornos integrados de trabajo.

Entre las posibilidades para desarrollar aplicaciones **Java ME** puedes encontrar:

- **Netbeans**. Entorno utilizado para desarrollar aplicaciones **Java** (**Java SE** y **Java EE**) y lenguajes (C, C++, PHP) y que también permite el trabajo con **Java ME**.

- **Light Weight User Interface Toolkit** (**LWUIT**). Desarrollada por **Sun** (hoy día ya Oracle) para el desarrollo de aplicaciones con **Java ME**.

- Eclipse. Requiere la instalación del plugin **EclipseME**.

- Android studio, creada por google.

- Otros entornos que permitan el desarrollo de proyectos **Java ME** como **BlueJ**, **IntelliJ**, etc.

4.Módulos para el desarrollo de aplicaciones móviles.

Características de Android

- Plataforma abierta: está basada en Linux (de código abierto) y puede ser personalizado de forma gratuita por los programadores.
- Adaptable: puede adaptarse a diversos tipos de dispositivos como relojes, teléfonos, tabletas, etc.
- Portable: puede instalarse en máquinas de distinto hardware.
- Interface de usuario está basado en xml (tecnología web) y permite adaptarse a distintos tamaños de pantalla.
- Siempre conectado: permite estar conectado a internet para aplicaciones tipo mensajerías, de mapas, etc

4.Módulos para el desarrollo de aplicaciones móviles.

Características de Android

- Incorpora muchos servicios: GPS, bases de datos, reconocimientos de voz, etc.
- Tiene buena seguridad pues está basado en Linux y al igual que el SO es seguro.
- Optimizado, pues necesita poco procesador y poca memoria RAM. Esto es una ventaja para pequeños dispositivos. Tiene que ver con la máquina virtual dalvik, se usa para ejecutar las aplicaciones sobre esa pequeña máquina virtual. Cada aplicación de Android se ejecuta dentro de una instancia de la Dalvik VM, que a su vez permanece dentro de un proceso gestionado por el kernel Linux.
- Incorpora los codec más utilizados para reproducir audio y video. Tiene soporte para gráficos 3D.

4.Módulos para el desarrollo de aplicaciones móviles.

- ¿Qué software será necesario instalar?

Descarga JDK:
https://www.oracle.com/es/java/technologies/javase/javase-jdk8-downloads.html

Descarga Android Studio:
https://developer.android.com/studio

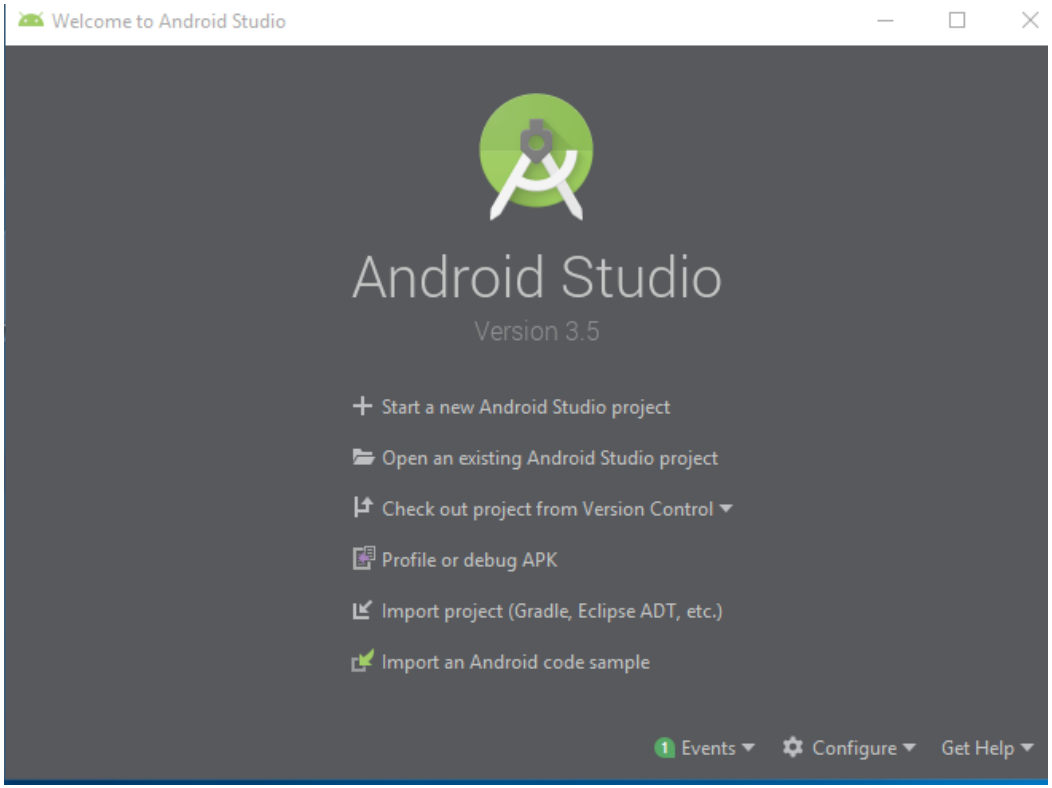
Vamos a instalar

5. Integración en el entorno de desarrollo.

Creación de un proyecto

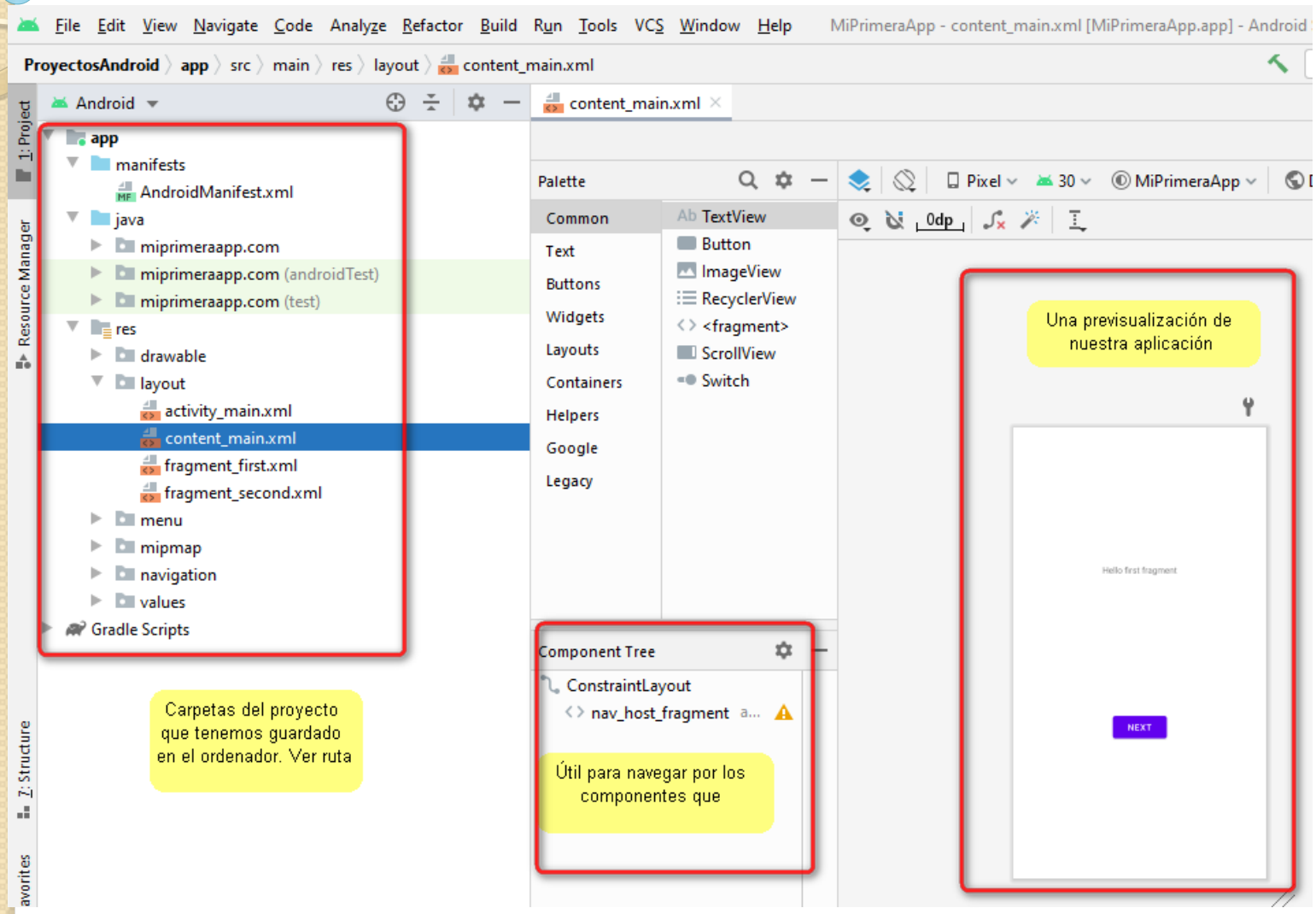
La primera vez que ejecutemos Android Studio, o cuando cerremos el proyecto activo, nos aparecerá la siguiente pantalla, de la que elegiremos la opción **Start a new Android Studio Project.** Si queremos abrir un nuevo proyecto teniendo uno activo ejecutaremos: **File → New → New Proyect…**

Vamos a crearnos un proyecto utilizando la versión de API de Android recomendada para que se pueda utilizar en el mayor número de dispositivos del mercado que hay actualmente. El nombre del proyecto será *PrimeraApp*, en este proyecto incluiremos una actividad vacía por defecto.



5. Integración en el entorno de desarrollo.

Carpetas proyecto



5. Integración en el entorno de desarrollo.

Componentes principales que pueden formar parte de una aplicación Android

- **Activity.** Las activitys son clases Java que heredan (extends) de la superclase Activity o de otras derivadas de ella. Contiene toda la lógica de la aplicación.
- **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, permitiéndonos también crear nuestros propios controles.
- **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, si la aplicación necesita en algún momento interactuar con el usuario.

5. Integración en el entorno de desarrollo.

Componentes principales que pueden formar parte de una aplicación Android

•**Intent.** Los intents (intenciones) son los encargados de las comunicaciones internas y externas de la aplicación. Pueden lanzar activitys, o pedir a otras aplicaciones externas que ejecuten una acción (ej. abrir galería de fotos).

- 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 ejecutar una activity desde cualquier otra,* iniciar un servicio, enviar un mensaje *broadcast, iniciar otra aplicación,* etc.
- Debemos tener en cuenta que un proyecto Android no tiene un único punto de entrada, sino que sus aplicaciones (activitys) reaccionan como resultado de *intents específicos.*
- Cuando se crea un nuevo proyecto, el sistema incluye en el fichero AndroidManifest.xml, una referencia a la clase MainActivity que contiene un intent (<intent-filter>) indicando que esa es la clase principal (clase Main) y que debe ejecutarse cuando se inicie el proyecto.

5. Integración en el entorno de desarrollo.

Componentes principales que pueden formar parte de una aplicación Android

•**Broadcast Receiber.** Los Broadcast Receiber son componentes que reaccionan a *intents específicos y pueden ejecutar acciones. Por ejemplo,* lanzar una determinada activity o devolver otro intent al sistema para que siga el proceso.

- También pueden detectar y reaccionar ante determinados mensajes o eventos globales generados por el sistema (por ejemplo: “Batería baja”, “SMS recibido”, “Tarjeta SD insertada”) o por otras aplicaciones.

•**Content Provider.** Un "proveedor de contenidos" 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.

5. Integración en el entorno de desarrollo.

Partes importantes de un proyecto

Al crear nuestro proyecto consta de dos ficheros importantes, vamos a verlos:

• **MainActivity.java:** contiene la lógica de nuestra aplicación, es decir, el funcionamiento. Por ejemplo, si al pulsar un botón tiene que suceder algo. Contiene una clase Java que hereda de la superclase AppCompatActivity.

- Una aplicación Android contiene múltiples clases, que se comunican entre sí por medio de eventos llamados **intents.**

- El proceso interno de la aplicación es llevado a cabo por estas clases, denominadas Activitys**.**

- En la clase **MainActivity.java** incluiremos las instrucciones necesarias para interactuar con los elementos contenidos en **activity_main.xml**

- Para activar un layout dentro de una activity se utiliza el método ***setContentView()***

5. Integración en el entorno de desarrollo.

Partes importantes de un proyecto

•**activity_main.xml:** Contiene un documento XML, denominado layout, con información sobre lo que se va a mostrar en la pantalla del dispositivo. Por ejemplo, botones, colores, etc.

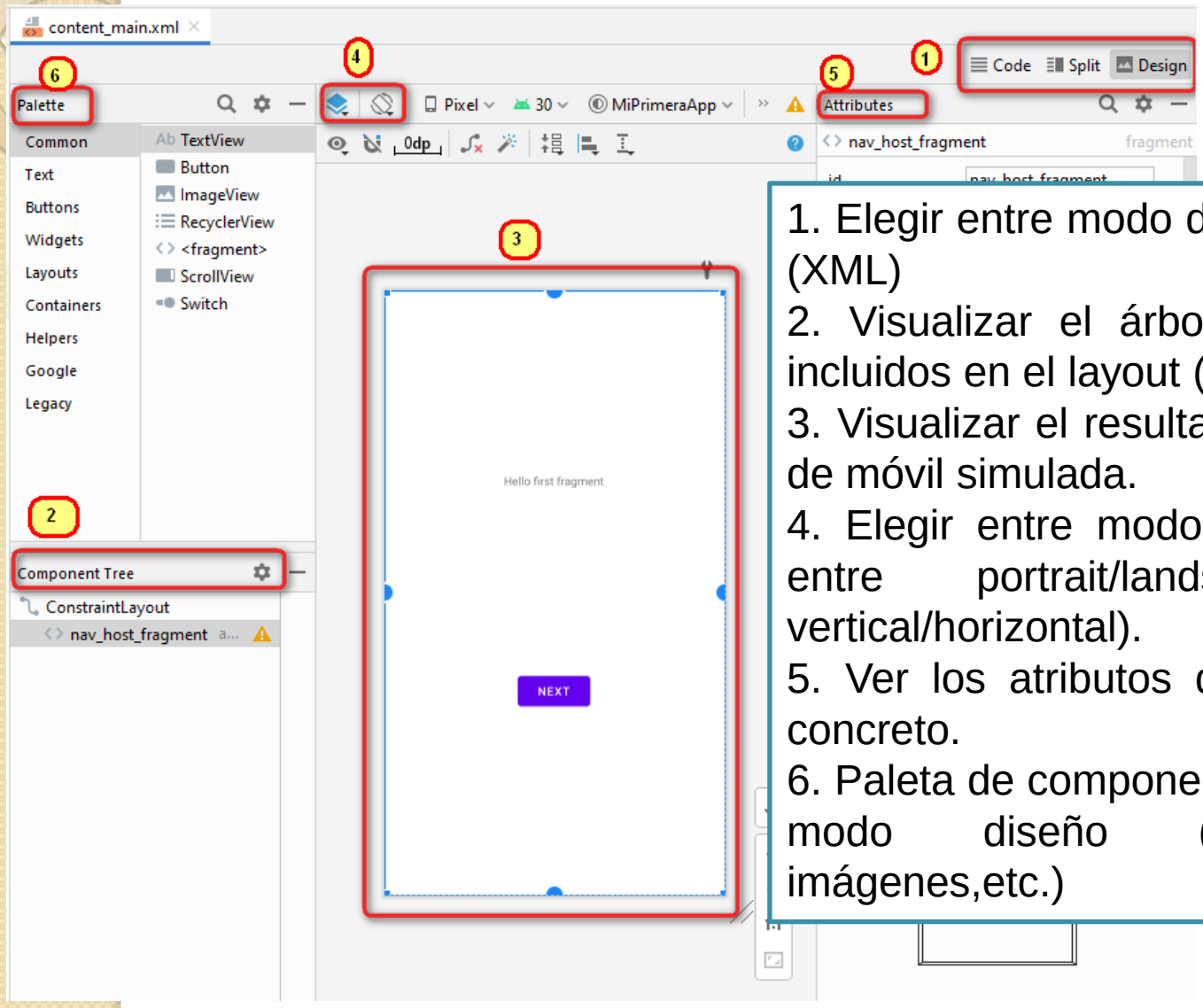
- Cada elemento del documento XML especifica un control, denominado widget, que parecerá en la pantalla.

- Las características de estos controles se especifican mediante atributos.

- El elemento raíz se denomina **layout (o un derivado)** y es un control contenedor, es decir, que puede contener otros controles o incluso otros layouts.

- **activity_main** puede verse y modificarse también en **modo diseño**, según nos muestra la siguiente pantalla.

5. Integración en el entorno de desarrollo.



1. Elegir entre modo diseño y modo texto (XML)
2. Visualizar el árbol de componentes incluidos en el layout (activity_main)
3. Visualizar el resultado en una pantalla de móvil simulada.
4. Elegir entre modo diseño/blueprint y entre portrait/landscape (pantalla vertical/horizontal).
5. Ver los atributos de un componente concreto.
6. Paleta de componentes para añadir en modo diseño (botones, texto, imágenes,etc.)

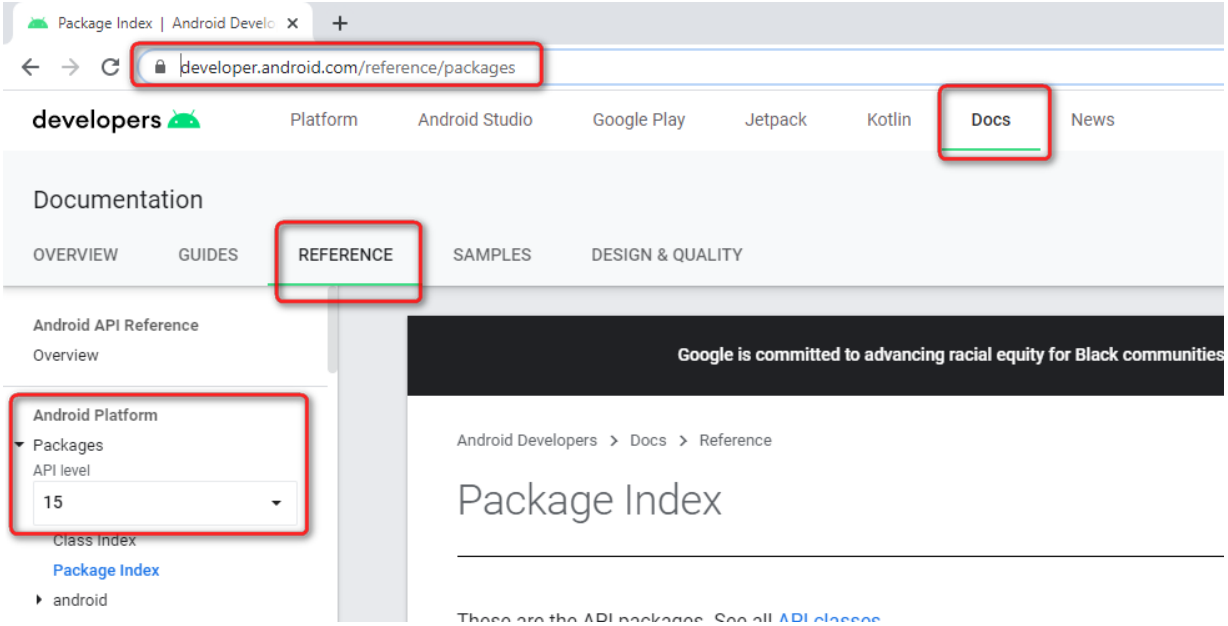
5. Integración en el entorno de desarrollo.

Ejecución

Para ejecutar la aplicación: **Run → Run 'app'...**

En la pantalla que aparece seleccionamos el emulador en el que queremos que se muestre. También permite crear un emulador nuevo, que es lo que haremos a continuación.

API Android



https://developer.android.com/reference/packages

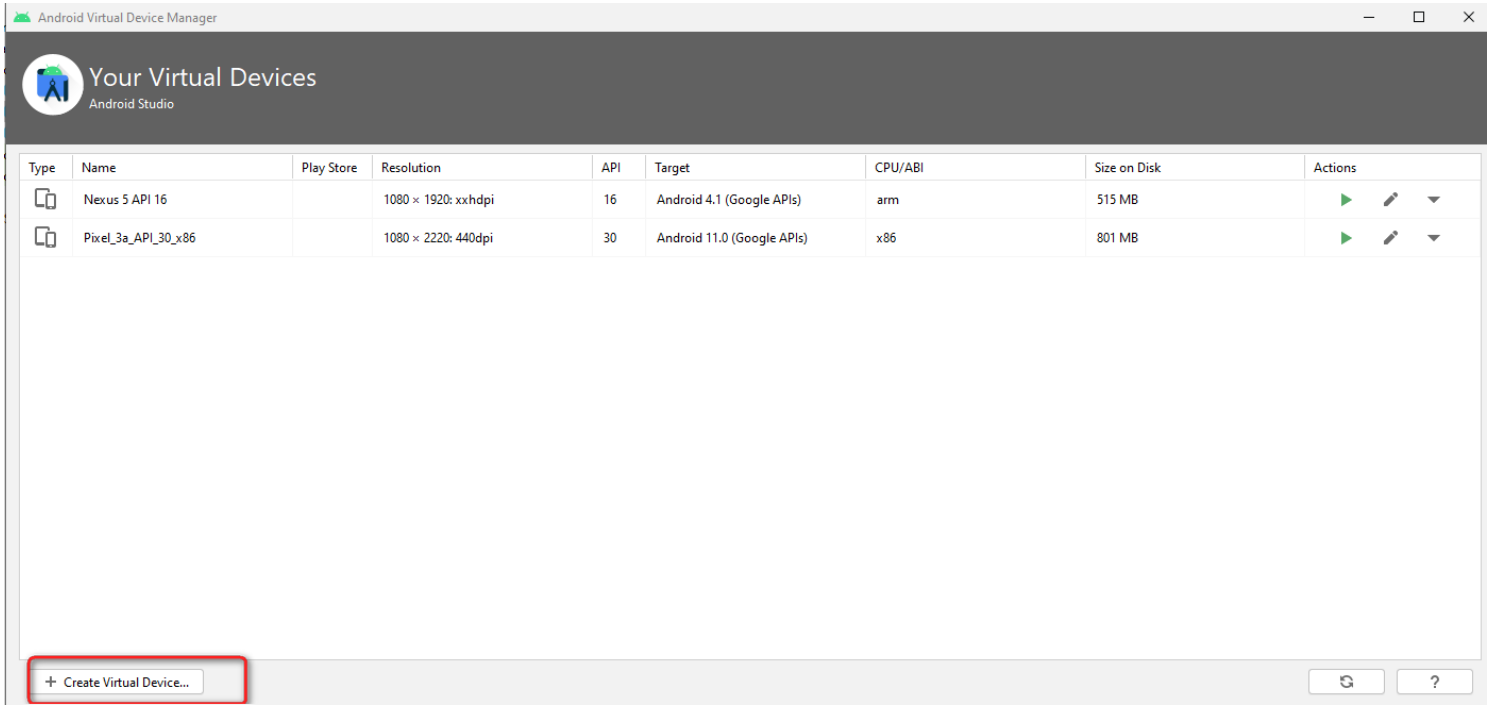
5. Integración en el entorno de desarrollo.

Emuladores

Para gestionar los AVD en cualquier momento podemos ir a *Tools>AVD Manager*

• Hacemos clic en *Create Virtual Device*

• En las siguientes pantallas se puede seleccionar el tipo de dispositivo móvil y la imagen de la versión de Android que corre. Nosotros crearemos un dispositivo Nexus 5 con una imagen API 16



Tema 2 - INTRODUCCIÓN A LA PROGRAMACIÓN EN DISPOSITIVOS MÓVILES

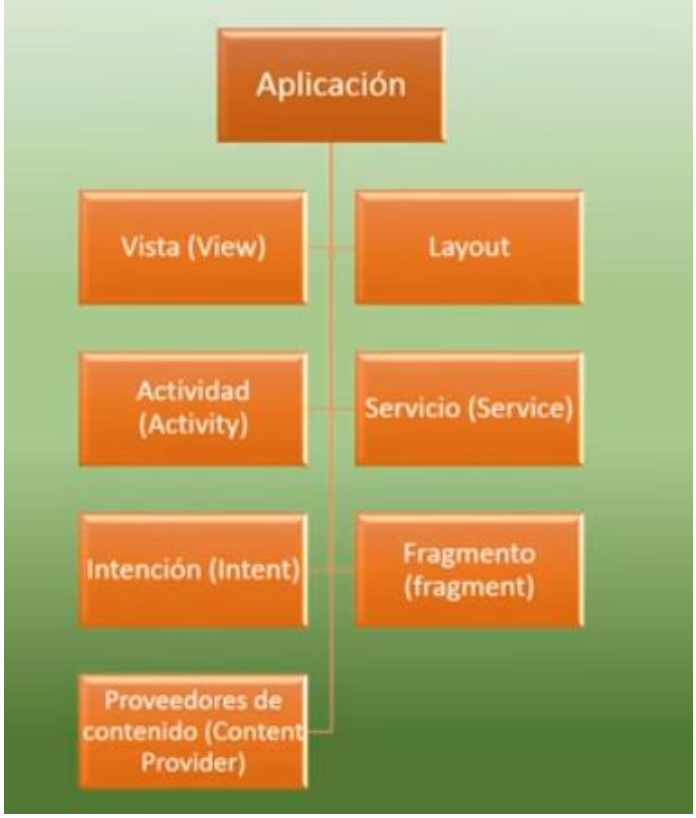
Parte I

Índice

Componentes de una aplicación 3
View (Vista) 3
Layouts 4
Activity (actividad) 4
Service (Servicio) 5
Intent 5
Fragment (fragmentos) 5
Content Provider (proveedores de contenido) 5
Cómo crear una interfaz de usuario 6
Controles de distribución 7
Introducción 7
Dimensiones 7
Posicionamiento 7
Root element y conversión 8
ConstraintLayout 8
LinearLayout 9
FrameLayout 10
TableLayout 11
Mejora del rendimiento de la interfaz 11
Controles básicos en Android 12
Control TextView 12
Control EditText 13
Controles tipo Button 14
Control ToggleButton 15
Control Switch 15
Control CheckBox 15
Control ImageButton 16
Controles RadioButton y RadioGroup 16
Control ImageView 16

Componentes de una aplicación

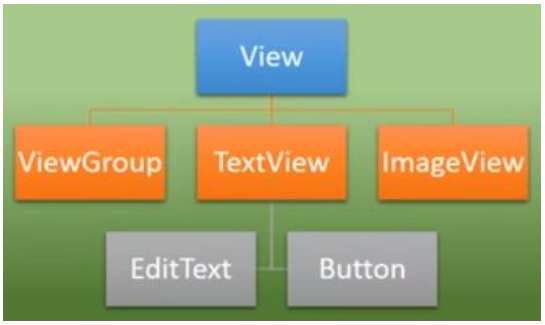
Recordemos que una aplicación Android se puede componer de los siguientes componentes:



Estos componentes se relacionan entre sí. No tiene por qué tener todos los componentes una aplicación. Recordemos brevemente cada uno.

View (Vista)

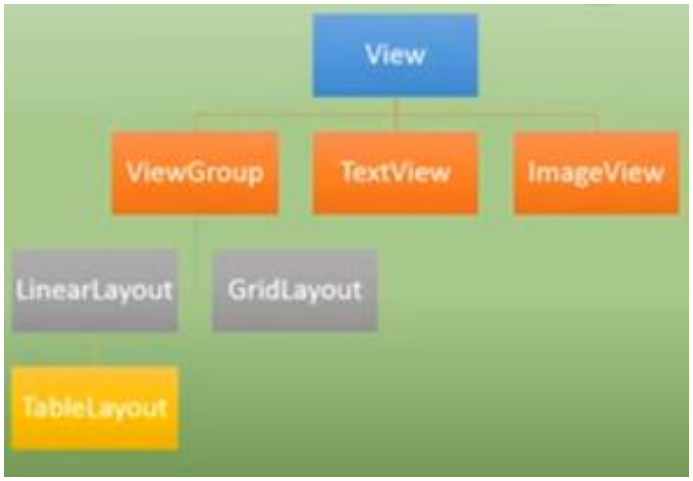
Las vistas son los componentes que forman la interfaz de usuario: textos, imágenes, botones, desplegables, etc. Son descendientes de la clase *View*. Aquí se pueden ver algunas subclases que heredan de la superclase *View*.



Layouts

Sirven para distribuir los componentes que forman parte de la interfaz. También heredan de la clase View. Son un conjunto de vistas agrupadas de una determinada forma. Por ejemplo:

- LinearLayout (Horizontal)
- LinearLayout (Vertical)
- GridLayout
- TableLayout



Activity (actividad)

Las activities son clases Java que heredan (extends) de la superclase *Activity* o de otras derivadas de ella. Contiene toda la lógica de la aplicación. Una aplicación Android va a estar formada por una o varias actividades. Por entendernos, una actividad se asemeja a cada pantallazo que utilicemos en la aplicación. Algunos ejemplos:

Service (Servicio)

Los servicios son componentes sin interfaz gráfica que se ejecutan en segundo plano. En concepto son como los servicios que tenemos en el sistema operativo. Pueden realizar cualquier tipo de acciones, por ejemplo actualizar datos, lanzar notificaciones, etc.

Intent

Podríamos decir que cuando un usuario hace algo sobre la aplicación, desencadena una serie de acciones. Por lo que podemos traducirlo en la voluntad de querer (intención) de realizar una acción.
Los intents (intenciones) son los encargados de las comunicaciones internas y externas de la aplicación. Pueden lanzar activites, o pedir a otras aplicaciones externas que ejecuten una acción. Por ejemplo, la voluntad de ir de una actividad a otra cuando el usuario pulse un icono o la acción de abrir galería de fotos.

Fragment (fragmentos)

Sirven para que una aplicación que se ve en el móvil se vea bien también en las tabletas.

Content Provider (proveedores de contenido)

Es el mecanismo que se ha definido en Android para compartir datos entre aplicaciones. Por ejemplo acceso a la agenda, a las fotografías, posicionamiento, etc.

Cómo crear una interfaz de usuario

Esta es la guía de referencia oficial de Android donde podemos encontrar información similar a cuando buscamos en el API de Java: https://developer.android.com/reference/packages

Nosotros crearemos una interfaz de usuario programando en xml y apoyándonos de los editores visuales que nos ofrece Android Studio. También se puede programar utilizando java pero no es aconsejable por organización del proyecto por lo que no es lo habitual.

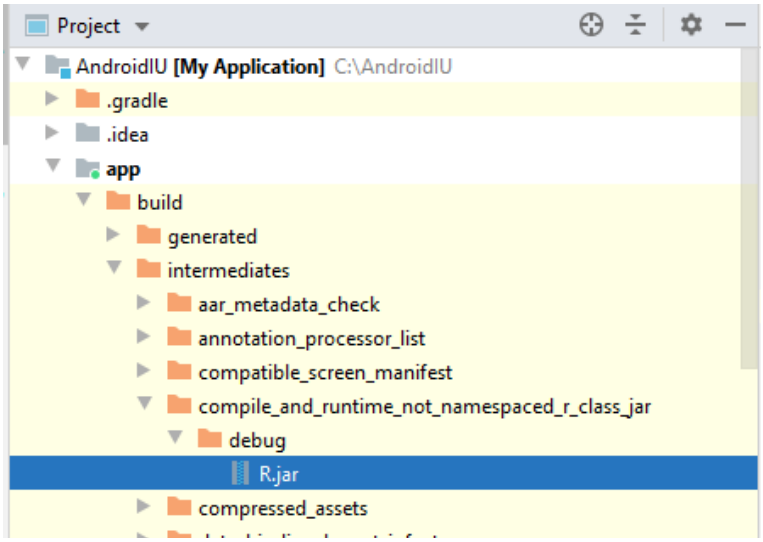
Si nos fijamos en *MainActivity.java* vemos que por defecto nos crea una clase MainActivity que hereda de AppCompactActivity. Esta clase, AppCompactActivity, hereda de Activity y ésta a su vez de Context. Esto podemos comprobarlo revisando la guía de referencia oficial que se indica al principio de este apartado.

Otro punto que puede llamarnos la atención en este fichero *MainActivity.java* cuando hemos creado una aplicación básica por defecto es que hace referencia a R. Se trata de un archivo que se va generando en tiempo de ejecución, por lo que este archivo se va modificando automáticamente. Contiene los recursos que carga nuestra app cuando se tiene que ejecutar. Por tanto, este archivo no debe ser editado NUNCA. Cuando queremos referenciar a algún componente lo mejor es dejarlo con la ruta, como en el ejemplo cuando creamos una aplicación vacía.

```
public class MainActivity extends AppCompatActivity {

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

Si ponemos en Android Studio la vista de proyecto, podemos ver el R.jar que se va creando en la siguiente ruta:



Para cambiar la interfaz de forma visual lo hacemos en los ficheros xml. Normalmente se nos genera por defecto de manera automática los ficheros *activity_main.xml* y *content_main.xml*. Se puede decir que se suele hacer esta separación en dos ficheros para dejar en *content_main.xml* la parte visual que quieres mostrar al usuario con los elementos principales

y así poder reutilizar la interfaz y por otro lado en *activity_main.xml* se suelen poner las acciones especiales como por ejemplo la paginación, botones flotantes, etc. Es decir, es por una cuestión de organización en el proyecto.

Controles de distribución

Introducción

- Los **layouts** son elementos no visibles destinados a controlar la distribución, posición y dimensiones de los controles que aparecen en la pantalla.
- Los *layouts* extienden a la clase base *ViewGroup*.
- A esta clase pertenecen otros elementos contenedores, es decir, capaces de contener a otros controles de la clase *View*.
- Android proporciona varias clases de *Layouts*.

Dimensiones

- Propiedades *layout_width* y *layout_height*: establecen el ancho y el alto tanto de un layout dentro de la pantalla o de otro layout, como de un control dentro del *layout*.
- Podrán tomar los siguientes valores:
    - "match_parent" Ajusta el tamaño a las dimensiones máximas que el padre le permita.
    - "wrap_content " Ajusta el tamaño al espacio mínimo que requiere el control o layout.
    - Un entero en unidades dp.
        - **dp: píxeles dependientes de la densidad**: unidad que se basa en la densidad de píxeles de la pantalla y sirve para que los controles muestren las mismas medidas físicas (no es exacto) en cualquier pantalla.
        - 63 dp equivalen a un centímetro.
        - No confundir con ***sp*,** que mide el tamaño de un texto.

Posicionamiento

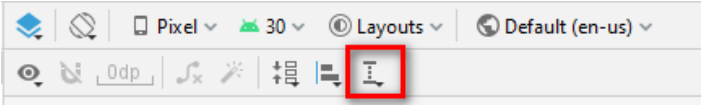
- Propiedad *layout-gravity*: sirve para definir el posicionamiento tanto de un layout en la pantalla o dentro de otro layout, como de un control en el layout que lo contiene.
- Propiedad *gravity*: sirve para especificar el posicionamiento del contenido dentro de un control (View).
- Los valores de gravity y layout-gravity pueden combinarse mediante el operador **|** (o utilizando la interfaz de diseño).

Root element y conversión

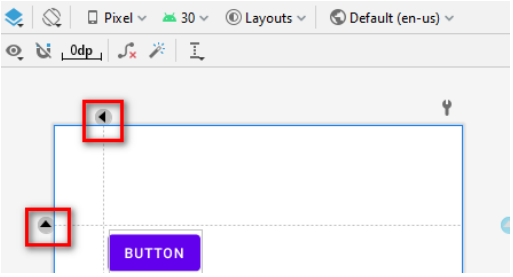
- Un *layout* puede crearse mediante ***New>Android Resource File*** con tipo *Layout.*
- Al crear un layout, se debe definir un *Root element*, que será el elemento raíz del layout (FrameLayout, LinearLayout, ConstraintLayout, …). Esto puede modificarse en cualquier momento de forma sencilla desde el *Component Tree*, mediante clic derecho> *Convert View…*
- Para lanzar un layout desde Java, se utiliza la función *setContentView(R.layout.<nombre>)*
- Por defecto, se crea un layout *activity_main.xml*, que se lanza desde *onCreate* mediante la función anterior.

ConstraintLayout

- *ConstraintLayout* reemplaza a *RelativeLayout* como layout principal para desarrollo de interfaces de usuario.
- Un ***ConstraintLayout*** permite especificar la posición de cada control tanto con respecto al layout que lo contiene como con respecto a cualquier otro control incluido en el mismo layout, todo mediante la interfaz gráfica.
    - De esta forma, al incluir un nuevo elemento *Y* podremos indicar por ejemplo que debe colocarse *debajo del elemento* X y *alineado a la derecha del layout* .
- Es posible utilizar guías y barreras para establecer límites y distancias en la interfaz. La forma más sencilla de hacerlo es mediante el botón de la interfaz.



- Es posible establecer las guías mediante distancia hacia uno de los lados del *ConstraintLayout* pulsando sobre el botón pequeño circular que sale encima o a la izquierda de las guías (dependiendo de si es una guía horizontal o vertical).



- Las barreras (*barrier*) se suelen utilizar para establecer distancias entre una vista respecto a un grupo de vistas.

Ejemplo: En código podemos crear 3 líneas de división horizontal, una al 20% otra al 50% y otra al 75% de la pantalla:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
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"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">
(…)
   <androidx.constraintlayout.widget.Guideline
       android:id="@+id/guideline"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content"
       android:orientation="horizontal"
       app:layout_constraintGuide_percent="0.75" />

   <androidx.constraintlayout.widget.Guideline
       android:id="@+id/guideline2"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content"
       android:orientation="horizontal"
       app:layout_constraintGuide_percent="0.2" />

   <androidx.constraintlayout.widget.Guideline
       android:id="@+id/guideline3"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content"
       android:orientation="horizontal"
       app:layout_constraintGuide_percent="0.5" />

(…)

(Podemos incluir componentes en cualquier parte del layout y poner
restricciones para que ocupe el espacio entre las guías que se definan)

</androidx.constraintlayout.widget.ConstraintLayout>
```

LinearLayout

- Un *LinearLayout* visualiza los controles que contiene en una sola dimensión, o todos en fila o todos en columna, dependiendo del valor de la propiedad *android:orientation*.
- Pueden utilizarse las propiedades *gravity* y *layout-gravity***.**
- Propiedad *layout_weight*: Permite dar a los controles contenidos en el *Layout* unas dimensiones proporcionales entre ellos. Para ello se incluye esta propiedad en cada uno de los controles y se le asigna distintos valores enteros.
    - Por ejemplo, si asignamos a un control un valor 1 y a otro un valor 2 el segundo ocupará el doble de espacio que el primero.
    - Es conveniente asignar 0dp al ancho (o alto) de los controles para que el peso sea exacto.
- Se pueden anidar y combinar con el *layout Space* (y la propiedad *padding*) para obtener interfaces más complejas.

Ejemplo de un LinearLayout en horizontal:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:gravity="center_horizontal"
    android:weightSum="6">


    <Button
        android:id="@+id/button5"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:text="Este es el texto del botón 1"
        android:layout_weight="3" />


    <androidx.appcompat.widget.AppCompatEditText
        android:layout_width="10dp"
        android:layout_height="wrap_content"/>

    <Button
        android:id="@+id/button6"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_weight="2"
        android:text="Este es el botón2" />

    <Button
        android:id="@+id/button7"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:text="3" />
</LinearLayout>
```

FrameLayout

- Es un *Layout* pensado para reservar un área de la pantalla para mostrar un único *item*. Pretende evitar que los hijos de cada ***FrameLayout*** se superpongan al cambiar de pantalla u orientación.
- Se puede, sin embargo, asignar varios hijos a un ***FrameLayout*** y utilizar propiedades para distribuirlos en el mismo.
- Un ***FrameLayout*** coloca todos sus controles hijos alineados, en principio, en su esquina superior izquierda, de forma que cada control quedará parcial o totalmente oculto por el control siguiente, dependiendo de su tamaño relativo.
    - Esa posición puede cambiarse mediante la propiedad *layout-gravity*.
- Si se añaden mediante Java, los hijos se muestran utilizando una cola, donde los más recientes se muestran por encima (LIFO).

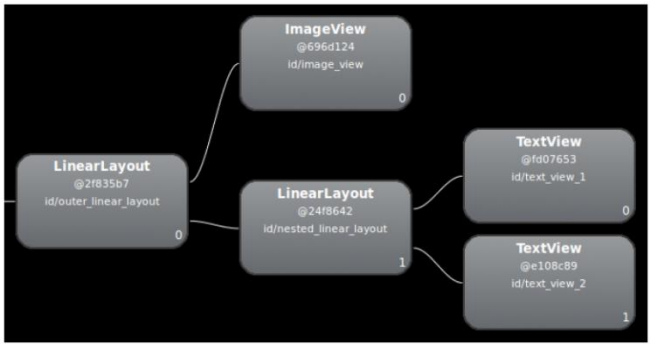
TableLayout

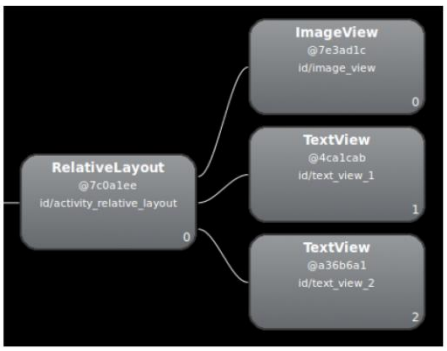
- Reemplaza a GridLayout (legacy) y tiene un propósito muy similar.
- Un **TableLayout** permite distribuir los controles que contiene de forma tabular, definiendo las filas necesarias mediante etiquetas ***TableRow***.
- En cada fila (***TableRow***) se añaden celdas, que serán los elementos View que contenga la fila (imágenes, botones, etc.)
- El número de columnas será el máximo número de celdas de entre las filas de la tabla.
- Si queremos que un control ocupe el espacio destinado a varias celdas, incluiremos en su definición la propiedad ***layout_span***. No es posible que un control ocupe el espacio destinado a celdas de distintas filas (sí se puede con *GridLayout*).
- El ancho de columna lo determina la columna más ancha de la tabla.
- Es posible utilizar los pesos de forma similar a cómo se hace con otras vistas (0dp + layout_weight="x").

Mejora del rendimiento de la interfaz

- Para mejorar el rendimiento de una interfaz se proponen dos técnicas basadas en las recomendaciones de desarrollo de Android:

  a. **Optimizar las jerarquías reduciendo el número de niveles**. Es mejor utilizar ViewGroup complejos (como por ejemplo, TableLayout), que utilizar varios niveles de ViewGroup más sencillos (como por ejemplo, TableLayout). Se ha de intentar reducir la profundidad del árbol que se visualiza en el *Component Tree*.
  Hay que tener especial cuidado con los layout que se utilizan muchas veces en la interfaz. Por ejemplo, al crear listas.





  Una mejora en una interfaz de 3 niveles a 2 niveles, como se muestra en las imágenes, puede ser muy notable si esa interfaz se ejecuta muchas veces para generar una lista.

  b. **Reutilización de layouts.** En ocasiones se utiliza un layout en varias actividades (pantallas) de una aplicación. Para organizar los archivos XML y generar un código más limpio y eficiente, se reutilizan los layout mediante la etiqueta
  include: <include layout="@layout/<nombre>"/>
  Puede darse el caso de que queramos crear un layout sin elemento raíz para evitar anidamiento innecesario. Para esto, podemos utilizar la etiqueta *merge*:

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

  …

  </merge>

Controles básicos en Android

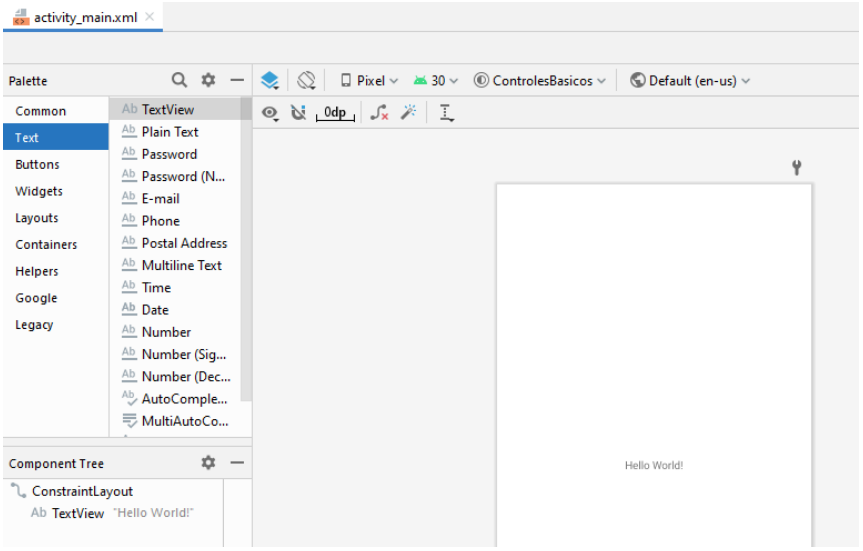
Control TextView

- Un control de la clase ***TextView*** se utiliza para mostrar un texto en la pantalla.
- Método *setText*() : permite mostrar un texto en el control asociado al objeto sobre el que se aplica.

Ejemplo

Para crear un control *TextView* y manipularlo desde una activity debemos hacer lo siguiente:

- Borramos el *TextView* que aparece por defecto.
- Seleccionamos la pestaña **Design** de **activity_main,** elegimos el tipo de control *TextView* que queremos mostrar (dede *Common* o *Text*) y lo arrastramos a la pantalla.



- Modificamos sus propiedades para obtener el siguiente documento (bien en modo texto o bien en la ventana **Component tree** ⇒ probar de ambas formas) intentando deducir el efecto de cada propiedad.

```
<TextView
    android:id="@+id/tv1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_centerHorizontal="true"
    android:layout_centerVertical="true"
    android:layout_marginBottom="100dp"
    android:background="?attr/colorPrimaryDark"
    android:text="@string/ejemplo1"
    android:textColor="@android:color/holo_red_light"
    android:textSize="40sp"
    app:layout_constraintBottom_toTopOf="@+id/textView2"
 />
```

- Incluye en el fichero *res -> values -> strings.xml* la traducción del texto que queremos que se muestre por pantalla en ejemplo1. Debe quedar similar a esta información:

```
<resources>
    <string name="app_name">ControlesBasicos</string>
    <string name="ejemplo1">Ejemplo traducir</string>
</resources>
```

Este fichero se utiliza para las traducciones de cadenas de caracteres.

Ejecutamos con el emulador para ver resultados y vemos que aparece la traducción que hemos puesto en *res -> values -> strings.xml*. Cambia la traducción y vuelve a ejecutar la aplicación.

- Realizar cambios desde dentro de MainActivity :

```
public class MainActivity extends AppCompatActivity {

    //Creamos un objeto de la clase TextView
    private TextView tv1 ;

//Este método es el primero que se ejecuta cuando se inicia la activity
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        //Asociamos esta activity al layout activity_main
        setContentView(R.layout.activity_main);
        //Asociamos el objeto con el control
        tv1 =(TextView)findViewById(R.id.tv1);
        //Se mostrará en la pantalla dentro del control asociado el texto
        tv1.setText( "Esto es un control TextView" );
    }
}
```

Ejecutamos con el emulador para ver resultados.

Control EditText

- Un control de la clase ***EditText*** es un editor de texto. Permite pues la introducción de texto por parte del usuario.
- Si queremos que el control muestre un texto por defecto podemos utilizar las propiedades text o hint. En este segundo caso el texto se mostrará atenuado.
- Propiedad *inputType*: permite indicar el tipo de contenido que se va a introducir en el control.
    - Dependiendo del valor que se le de a esta propiedad, Android mostrará un teclado de un determinado tipo.
- Método *setText()*: Permite mostrar un texto.
- Método *getText()*: Almacena en el objeto *EditText* el contenido de su control asociado.
- El valor almacenado es de la clase *Editable*, por lo que, si queremos almacenar ese valor en un objeto *String* debemos aplicarle el método *toString*() .
    - La razón de lo anterior es que los controles *EditText* no sólo permiten editar texto plano, sino también texto enriquezido, i.e. texto con formato.
- Pueden elegirse controles *EditText* de diferentes tipos desde la pestaña *Text* de la ventana *Palette* (Plain Text, Password, E-mail, Phone, …).

Controles tipo Button

- Un control de tipo ***Button*** es un control que, mediante la propiedad *onClick*, puede asociarse a un método de su clase Java correspondiente.
    - Este método se ejecutará cuando se pulse el control.
- Puede mostrar un texto mediante la propiedad text.
- También puede formatearse mediante multitud de propiedades: background, typeface, textColor, textSize , etc.
- Los controles Button se encuentran en la carpeta Buttons de la ventana Palette (Button, ImageButton, Checkbox, …).

Ejemplo

Aplicación que permite introducir un texto en un control *EditText* y, al pulsar un botón, el texto se muestra en un *TextView*.

activity_main.xml

```
...

< EditText
android :layout_width= "wrap_content"
android :layout_height= "wrap_content"
android :id= "@+id/et1"
android :layout_alignParentTop= "true"
android :layout_centerHorizontal= "true"
android :layout_marginTop= "71dp"
android :textSize= "30dp"
android :hint= "Teclee un texto..." />

< Button
android :layout_width= "wrap_content"
android :layout_height= "wrap_content"
android :text= "Aceptar"
android :id= "@+id/button"
android :layout_centerVertical= "true"
android :layout_centerHorizontal= "true"
android :onClick= "aceptar"
android :textSize= "30dp" />

< TextView
android :layout_width= "wrap_content"
android :layout_height= "wrap_content"
android :id= "@+id/tv1"
android :layout_alignParentBottom= "true"
android :layout_centerHorizontal= "true"
android :layout_marginBottom= "55dp"
android :hint= "Mostrar..."
android :textSize= "30dp" />
…
```

MainActivity.java

```
package com.example.usuario.ejemplo2;
import...
public class MainActivity extends AppCompatActivity {
@Override
protected void onCreate(Bundle savedInstanceState) {
super .onCreate(savedInstanceState);
setContentView(R.layout. activity_main );
EditText et1 =(EditText)findViewById(R.id. et1 );
TextView tv1 =(TextView)findViewById(R.id. tv1 );
}
public void aceptar(View v){
tv1 .setText( et1 .getText().toString());
}
}
```

Control ToggleButton

- Un control de tipo ***ToggleButton*** puede permanecer en 2 estados: seleccionado o no seleccionado.
- Propiedades *textOn* y *textOff*: Permiten especificar el texto que debe aparecer dentro del control dependiendo de que éste se encuentre seleccionado o no.
- Propiedad *checked* : Booleana. Establece que el control se muestre por defecto seleccionado o no.
- Método *isChecked*() : Booleano. Devuelve **true** si el control se encuentra seleccionado y **false** en caso contrario.
- Los controles *ToggleButton* se encuentran en la carpeta **Buttons** de la ventana **Palette.**

Control Switch

- Análogo a ToggleButton , sólo cambia su aspecto exterior.
- Los controles Switch se encuentran en la carpeta **Buttons** de la ventana **Palette.**

Control CheckBox

- Análogo a ToggleButton, cambia su aspecto exterior.
    - En este caso representa una casilla de verificación.
    - En este caso solo hay un atributo *text* , en lugar de *textOn/Off*
    - Se recomienda usar *CheckBox* cuando se quiera mostrar una lista de opciones a elegir y las otras cuando sea sólo una opción.
- Los controles CheckBox se encuentran en la carpeta **Buttons** de la ventana **Palette**.

Control ImageButton

- Un control de tipo *ImageButton* permite mostrar una imagen dentro de
- un botón.
- Propiedad ***src***: Debe contener la ruta de la imagen.
  - Para añadir un recurso de imagen basta con arrastrar las imágenes a la carpeta *drawable* (vista de proyecto)

  - Si tenemos almacenada en dicha carpeta una imagen denominada *imagen.jpg*, le asignaremos a la propiedad *src* el valor *@drawable/imagen* (sin extensión)
  - Se puede utilizar la propiedad scaleType="centerInside" para reescalar la imagen al tamaño elegido (por ej., 100dp)
- Los controles *ImageButton* se encuentran en la carpeta **Buttons** de la ventana **Palette**

Controles RadioButton y RadioGroup

- Un control *RadioButton* es análogo a un CheckBox , sólo cambia su aspecto.
- Se puede utilizar un control *RadioButton* de manera análoga a como se utilizaría cualquier control que pueda permanecer en 2 estados (activado/desactivado).
- Un control RadioGroup es un control contenedor.
  - Dentro de él se definen varios controles *RadioButton*, con la característica de que sólo uno de ellos puede estar activado.
  - Al activar un *RadioButton* se desactiva el que en ese momento estuviera activado dentro de su mismo *RadioGroup*.
  - Se definen ids individuales para cada botón y para el grupo.
  - Mediante la función *getCheckedRadioButtonId()* se puede obtener el ID del botón pulsado en cada momento.
- Los controles *RadioButton* /*RadioGroup* se encuentran en la carpeta **Buttons** de la ventana **Palette**

Control ImageView

- Un control de tipo *ImageView* permite mostrar una imagen.
- Propiedad **src**: Muestra una imagen por defecto.
- Método *setImageResource*(). Recibe como parámetro el **ID** de una imagen y la muestra en el control asociado al objeto al que se aplica.
  - El **ID** es un identificador que Android asocia a cada control o recurso definido en la aplicación. Se almacenan en la clase **R**.
  - El ID del recurso podemos especificarlo a partir de la clase R: R.drawable.imagen (siendo *imagen.jpg* el nombre del recurso)
  - El problema surge cuando no podemos especificar el nombre del recurso por estar éste almacenado en un objeto o en una variable, entonces debemos obtenerlo haciendo uso del método *getResources()*:
    int *resId =* ***this*** *.getResources().getIdentifier(* ***"nombreRecurso"*** *,* ***"drawable"****,* ***this.*** *getPackageName());*
- Los controles *ImageView* se encuentran en la carpeta **Widgets** de la ventana **Palette.**

TEMA 2 – INTRODUCCIÓN A LA PROGRAMACIÓN EN DISPOSITIVOS MÓVILES

Parte II

Índice

Controles de selección 3
Adaptadores (adapters) 3
Control Spinner 4
Control ListView (legacy) 4
Action bar y Fragment 5
1. La Barra de Acciones (Action bar) 5
2. Menús 6
2.1.- Menús principales 6
2.2.- Submenús 7
3. Action Bar personalizada 8
4. Fragments 8
4.1.- Creación de un fragment 9
Activity (actividad) 10
1.- El Fichero AndroidManifest 10
1.1.- Introducción 10
1.2.- Elementos Principales de un AndroidManifest.xml 10
2.- La clase Intent 11
3.- Ciclo de vida de una Activity 11
3.1.- Estados de una Activity 12
3.2.- Métodos protegidos 12
3.3.- Aspectos destacados del ciclo de vida 13
4.- Paso de parámetros entre Activitys 13
4.1.- Métodos putExtra() y get...Extra() 13

Controles de selección

Adaptadores (adapters)

- Un control de selección permite al usuario elegir una opción de entre varias.
- Un ***adaptador*** es una interfaz entre los controles de selección y el modelo de datos.
    - Es el responsable de almacenar los datos y mostrarlos en el control de selección con el que se le asocie en el formato especificado.
- De los diferentes adaptadores que Android proporciona el más sencillo viene definido por la clase ***ArrayAdapter*** , que provee de datos a un control de selección a partir de objetos de cualquier tipo.
- Un objeto ***ArrayAdapter*** se crea de la forma siguiente:
    - Supongamos definido el array String datos[]={"A","B","C"}
    - La instrucción:

```
ArrayAdapter<String>adapter=new
ArrayAdapter<String>(this,android.R.layout.<formato_predefinido>,datos);
```

    - Crea el objeto adapter, almacena en él los datos del array *datos[]* y los va a mostrar en el formato predefinido que hayamos elegido, dentro del control de selección al que se asocie.
    - <formato_predefinido> puede variar entre versiones, actualmente sería *support_simple_spinner_dropdown_item*
    - Podemos pasarle el nombre de un layout personalizado por nosotros. Para ello, creamos un layout que contenga solamente un *TextView* como *Root Element* (crear con clic derecho en *res*) con el formato que queremos que muestren los items.
        - Si el nombre del layout es *formato.xml*, la instrucción anterior quedaría:
          ```
          ArrayAdapter<String> adapter=new
          ArrayAdapter<String>(this,R.layout.formato, datos);
          ```
    - Podemos también crear un fichero xml en *res>>values* para indicar el array de valores que queremos usar. Por ejemplo un fichero *opciones.xml*

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string-array name="combo_dias" >
        <item>Lunes</item>
        <item>Martes</item>
        <item>Miércoles</item>
        <item>Jueves</item>
        <item>Viernes</item>
        <item>Sabado</item>
        <item>Domingo</item>
    </string-array>
</resources>
```

    - Cuando creamos el adaptador, podemos indicar el layout y el array de datos por parámetro. Por ejemplo el layout formato.xml y el fichero de datos opciones.xml que habíamos indicado que se llamaba combo_dias:

```
ArrayAdapter<CharSequence>
adapter=ArrayAdapter.createFromResource(this,R.array.combo_dias,R.l
ayout.formato);
```

Control Spinner

- Un control ***Spinner*** muestra una lista de items cuando se pulsa y permite seleccionar uno de ellos.
    - Esta lista de items deberá estar almacenada en un objeto *ArrayAdapter*.
- Método *setAdapter*() . Aplicado a un objeto *Spinner* , le asocia el objeto *ArrayAdapter* que recibe como parámetro.
- Método *getSelectedItemPosition*(). Aplicado a un objeto *Spinner*, devuelve la posición del item seleccionado.
    - La posición del primer item es 0.
- Los controles *Spinner* se encuentran en la carpeta **Containers** de la ventana **Palette** .

Control ListView (legacy)

- Un control ***ListView*** muestra una lista de items seleccionables directamente por el usuario, permitiendo hacer scroll sobre ellos si no pueden mostrarse todos.
- En las últimas versiones, se ha sustituído *ListView* por *RecyclerView*, que es más avanzado y flexible. De todas formas, *ListView* se sigue utilizando y es interesante comprender su funcionamiento.
- Lo que pretendemos con este control de selección es que se realice una acción en el momento en el que el usuario pulse alguno de sus items.
- Método *setAdapter*() . Al igual que ocurría con la clase *Spinner*, le asocia al objeto *ListView* sobre el que se aplica el objeto *ArrayAdapter* que recibe como parámetro.
- Evento *setOnItemClickListener*() : Aplicado a un objeto *ListView* , permite indicar las acciones que debe llevar a cabo la aplicación cuando el usuario pulse en uno de sus items.
    - Mediante el siguiente código:

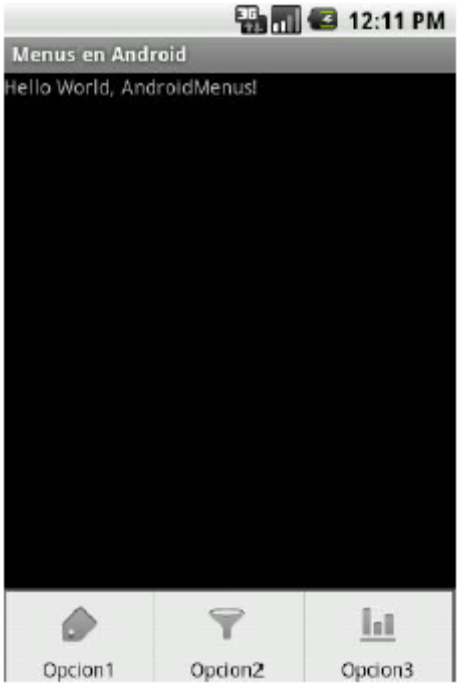
```
lv1.setOnItemClickListener(new AdapterView.OnItemClickListener() {
@Override public void onItemClick(AdapterView<?> parent, View view, int
posicion , long id) {
................
}
});
```

    - *lv1* es un objeto *ListView* al que previamente se le ha asociado un objeto *ArrayAdapter*.
    - Dentro del método *onItemClick*() indicaremos las acciones a llevar a cabo.
    - ***posicion*** contiene la posición del item pulsado, comenzando por 0.
    - Dado que dentro de un evento sólo se pueden incluir un limitado número de instrucciones, es aconsejable incluir únicamente una llamada a un método con ***posicion*** como parámetro de entrada.
        - Dentro del método indicaremos las acciones a realizar por nuestra aplicación.
- Los controles *ListView* se encuentran en la carpeta **Containers** de la ventana **Palette.**

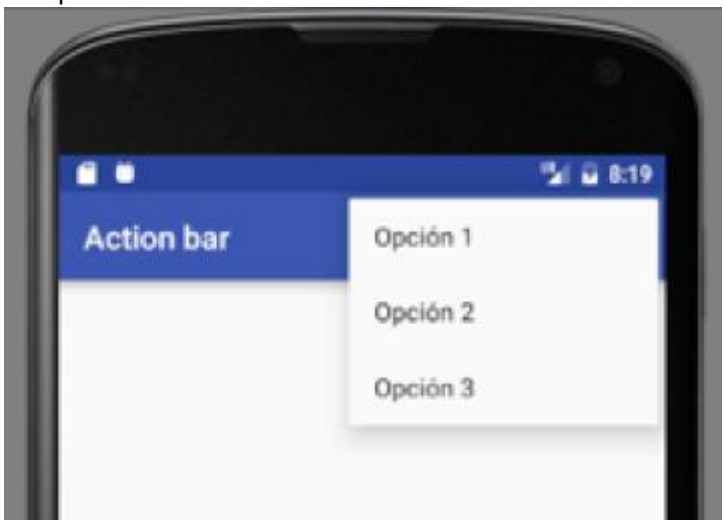
Action bar y Fragment

1. La Barra de Acciones (Action bar)

- La ***Action bar***, o *App bar*, es la barra de título y herramientas que aparece en la parte superior de la gran mayoría de aplicaciones actuales de la plataforma Android.
- Hasta la versión 3 (API 11) las aplicaciones podían ofrecer al usuario un conjunto de opciones en forma de menú que aparecían por defecto en la parte inferior de la pantalla:



- A partir de la versión 3 éstas se sitúan en el menú de opciones de la Action bar, también llamado menú de overflow, que se sitúa por defecto en la parte superior derecha de la pantalla:



- Con el objeto de asegurar la compatibilidad entre versiones, la forma de incluir opciones en una *Action bar* es la misma que la utilizada para crear un menú.

- Si el móvil implementa una versión anterior a la API 11 las opciones definidas en el menú aparecerán en la parte inferior de la pantalla y si implementa la API 11 o posterior, aparecerán en el menú de opciones de la *Action bar*.
- **Nota** : El título que aparece en la *Action bar*, que por defecto es el nombre de la aplicación, se establece, dentro del fichero ***AndroidManifest.xml*** , en el atributo ***android:label*** del elemento ***<application>***

2. Menús

- En Android podemos encontrar 3 tipos diferentes de menús:
    - *Menús Principales*. Los más habituales, aparecen en la zona inferior de la pantalla o en la *ActionBar.*
    - *Submenús*. Son menús secundarios que se pueden mostrar al pulsar sobre una opción de un menú principal.
    - *Menús Contextuales*. Aparecen al realizar una pulsación larga sobre algún elemento de la pantalla. No los veremos en este curso.

2.1.- Menús principales

- Existen 2 formas de crear un menú principal: definiéndolo dentro de un fichero XML o mediante código java:

1. **Crear menú Desde XML**. En el primer caso debemos crear un fichero XML, es decir, crear un nuevo recurso de tipo menú (botón derecho en ***res***).
    - El fichero consistirá en un elemento raíz <menu> que contenga un elemento <item> por cada opción que queramos que se muestre.
    - Cada elemento <item> deberá contener al menos 2 propiedades, una con su identificador y otra con el título de la opción.
    - Ejemplo (**/res/menu/menu_main.xml**):

```
<?xml version="1.0" encoding="utf-8"?>
<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/MnuOpc1"
        android:title="Opcion1"/>
    <item android:id="@+id/MnuOpc2"
        android:title="Opcion2"/>
    <item android:id="@+id/MnuOpc3"
        android:title="Opcion3"/>
</menu>
```

NOTA: En una aplicación compleja deberían utilizarse string (@string/id-string) en lugar de los literales en los títulos a hardcode.

- En la actividad donde queremos que aparezca el menú incluiremos el evento *onCreateOptionsMenu*()

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

```
        super.onCreateOptionsMenu(menu);
        getMenuInflater().inflate(R.menu. menu_main , menu);
        return true;
    }
```

- Para indicar las acciones a realizar cada vez que el usuario pulse una opción deberemos utilizar el método *onOptionsItemSelected*()

```
@Override
public boolean onOptionsItemSelected( MenuItem item){
switch (item.getItemId()){
case R.id. MnuOpc1 :
...;
return true;
case R.id. MnuOpc2 :
...;
return true;
case R.id. MnuOpc3 :
...;
return true;
default:
return super.onOptionsItemSelected(item);
}
}
```

2. **<u>Crear menú desde Java</u>**. En el segundo caso, podemos crear directamente el menú dentro del método *onCreateOptionsMenu*().
   - Para ello, añadimos cada opción al objeto menú mediante el método ***add()*** , del modo siguiente:
     Ejemplo:

```
...
private static final int OPCION1=1;
private static final int OPCION2=2;
private static final int OPCION3=3;
...
@Override
public boolean onCreateOptionsMenu(Menu menu) {
super.onCreateOptionsMenu(menu);
menu.add(Menu.NONE,OPCION1,Menu.NONE,"Opcion 1");
menu.add(Menu.NONE,OPCION2,Menu.NONE,"Opcion 2");
menu.add(Menu.NONE,OPCION3,Menu.NONE,"Opcion 3");
return true;
}
```

   - Se incluyen las líneas *menu.add(...)* sustituyendo a *getMenuInflater().inflate(...)*
   - Para indicar las acciones a realizar utilizaremos el método *onOptionsItemSelected()* igual que en el caso anterior, refiriéndonos en este caso a las constantes *OPCIONX* en vez de al ID de los elementos <item>.

2.2.- Submenús

- Un submenú es un menú secundario que se muestra al pulsar una opción de un menú principal.
- Al igual que ocurría en el caso de los menús principales, Los submenús pueden declararse como *menu* dentro del fichero XML o directamente mediante código.

- En el primer caso añadiremos al fichero XML que contiene la declaración del menú principal un *<menu>* por cada opción que contiene submenus, y los *<item>* necesarios en su interior.
- En el segundo caso, las opciones que vayan a contener submenús habrán de añadirse utilizando el método *addSubMenu()* en vez de *add()* .

Ejemplo
Siguiendo con el ejemplo anterior, si queremos que la opción 3 contenga un submenú con 2 opciones debemos añadir el siguiente código:

```
…
private static final int OPCION31=31;
private static final int OPCION32=32;
...
//Sustituimos menu.add(Menu.NONE,OPCION3,Menu.NONE,"Opcion 3"); por:
SubMenu smnu=menu.addSubMenu(Menu.NONE,OPCION3,Menu.NONE,"Opcion3");
smnu.add(Menu.NONE,OPCION31,Menu.NONE,"Opcion3.1");
setIcon(android.R.drawable.ic_menu_preferences); //asigna icono predefinido
al submenú
smnu.add(Menu.NONE,OPCION32,Menu.NONE,"Opcion3.2");
setIcon(android.R.drawable.ic_menu_agenda); //asigna icono predefinido al
submenú
```

3. Action Bar personalizada

Los pasos para crear una *action bar* personalizada son los siguientes:

1. Por defecto, el tema de la aplicación y el action bar, se establece en AndroidManifest.xml como *android:theme="@style/Theme.Menus"*.Si queremos utilizar una *Action Bar* personalizada, tenemos que modificar el valor de dicho atributo por uno de los que incluye *NoActionBar* (por ejemplo: `@style/Theme.AppCompat.Light.NoActionBar`)

2. Agregar a la interfaz un elemento *androidx.appcompat.widget.Toolbar* (de forma textual o desde *Palette>Containers>Toolbar.* También se podría crear un layout propio con este elemento como raíz e incluir el layout en la interfaz principal.

3. En el *onCreate()* de la actividad Java, añadir la *Action Bar* mediante ***setSupportActionBar(myToolbar)*** donde *myToolbar* es el objeto de la clase *Toolbar* que se corresponde con el *Toolbar* de la interfaz (**debes importar el paquete con el mismo nombre que el elemento del paso 2**).

   Ahora se podrían realizar cambios sobre la barra de acciones y usar sus herramientas desde Java, mediante el objeto *myToolbar*. Si hemos creado un menú, este se acoplará a la nueva barra de acciones igual que ocurría con la barra por defecto.

4. Fragments

- Un ***fragment*** no puede considerarse ni un control ni un contenedor, aunque se parecería más a lo segundo.
- Un *fragment* podría definirse como una porción de la interfaz de usuario que puede añadirse o eliminarse de dicha interfaz de forma independiente al resto de elementos de la actividad, así como reutilizarse en otras actividades.

- La necesidad de usar fragmentos nace con la versión 3.0 (API 11) de Android debido a los múltiples tamaños de pantalla que estaban apareciendo en el mercado y a la capacidad de orientación de la interfaz (Landscape y Portrait).
    - Estas características necesitaban dotar a las aplicaciones Android de la capacidad para adaptarse y responder a la interfaz de usuario sin importar el dispositivo.
- Un *fragment* es una sección *modular* de la interfaz de usuario (UI) embebida dentro de una actividad anfitriona, lo que permite versatilidad y optimización de diseño.
    - Se trata de "miniactividades" contenidas dentro de una actividad anfitriona, con su propio diseño (un layout propio) y clase java para manejarlo.
- Estas nuevas entidades permiten reusar código y ahorrar tiempo de diseño a la hora de desarrollar una aplicación.

4.1.- Creación de un fragment

- Creamos en */res/layout* un fichero *layout* con el formato y controles que queremos que se muestren en la UI del fragment. Guardamos este layout con el nombre de *fragmento_ejemplo.xml*

- Creamos en la misma carpeta de *MainActivity* una clase java que extienda a la superclase *Fragment* (botón derecho *>New>Java Class*).
    - También podríamos usar subclases de fragments contenidas en la API de Android como por ejemplo ListFragment , DialogFragment , PreferenceFragment

```
public class FragmentoEjemplo extends Fragment {
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container,
Bundle savedInstanceState){
return inflater.inflate(R.layout. fragmento_ejemplo ,container,false);
}
}
```

- El método *onCreateView* () recibe como parámetro un objeto de la clase *LayoutInflater*, provisto por la clase *Fragment*.
- La clase que hemos creado es la encargada de *inflar* el fragment a través del método *inflate* (). Éste recibe los siguientes parámetros:
    - El fichero XML conteniendo el layout (*fragmento_ejemplo*).
    - El contenedor donde será insertado. En este caso el view de la actividad anfitriona.
    - Una bandera (objeto Bundle) indicando que el view que se producirá debe adherirse al contenedor.
- Una vez creada la UI del fragment y su clase correspondiente, debemos incluir en la UI de la actividad anfitriona (*activity_main* en este caso) un control contenedor de tipo *fragment*, que se encuentra en la carpeta *Containers* de la ventana *Palette* (también en *Common*).
    - Al pulsar sobre dicho control se nos muestra una ventana que nos permite elegir entre varios fragments proporcionados por la API de Android y el/los fragments creados por nosotros.
- Después en la clase *MainActivity.java* podemos utilizar los componentes del fragmento y definir los métodos que queramos utilizar con sus componentes.

Activity (actividad)

1.- El Fichero AndroidManifest

1.1.- Introducción

- El fichero *AndroidManifest.xml* es utilizado por Android para almacenar información sobre el contenido y estructura de una aplicación.
- Es un fichero XML generado por Android al crear un proyecto nuevo (*/manifests/AndroidManifest.xml*).
- En él se declaran todos los componentes de la aplicación, tales como las *Activitys* utilizadas, los *Intents*, el nombre de la aplicación, el hardware necesario para su ejecución, permisos, etc.
    - Algunas de estas componentes las incluye Android por defecto y otras debe incluirlas el programador.
- Hay que tener en cuenta que una aplicación Android no tiene un punto de entrada preestablecido, **en AndroidManifest debemos indicar cuál es la Activity que debe ejecutarse cuando se lance la aplicación**.

1.2.- Elementos Principales de un AndroidManifest.xml

- <manifest>: Es el elemento raíz. Debe contener, al menos, los siguientes atributos:
    - xmlns: define el namespace que será utilizado en el resto del documento.
      xmlns:android = " *http://schemas.android.com/apk/res/android*"
    - package: identifica el paquete raíz de nuestra aplicación.
      `package = "com.example.usuario.proyecto1"`
- <application>: En este elemento deben especificarse todas las Activities, Services, Providers, Receivers, etc. que vamos a utilizar en nuestra aplicación. Atributos:
    - *allowBackup*: Booleano, habilita a Android a que, al hacer un backup el usuario, se realice una copia de seguridad tanto de la aplicación como de los datos que en ese momento contenga.
        - Incorporado a partir de la API 17. `android:allowBackup = "true"`
    - *icon|label|theme* Permiten referenciar los recursos que se van a utilizar para el icono y título de la aplicación así como para el estilo.
      ```
      android:icon = "@mipmap/ic_launcher"
      android: label = "@string/app_name"
      android:theme = "@style/AppTheme"
      ```
    - *supportsRtl Booleano*. Permite, o no, la realización de diseños de derecha a izquierda.
- <activity> Dentro del elemento <application>. Debe especificarse un elemento <activity> por cada actividad creada en nuestra aplicación. Puede contener los siguientes atributos:
    - *name*: Indica el nombre de la clase Activity que se encuentra en el package indicado en el título del elemento <manifest>.
      `android:name = ".NombreActivity"`
    - *label*: Especifica el string que van a aparecer en la barra de título cuando sea lanzada esta activity. Si no se especifican, se utilizarán por defecto los que se hayan indicado en <application>
    - *screenOrientation*: Especifica la orientación con la que se va a mostrar la activity en el dispositivo (*landscape|portrait*).
      `android:screenOrientation = "landscape"`

    - *configChanges*: Permite especificar qué cambios de configuración queremos manejar nosotros mismos y que no lo haga por defecto el dispositivo. Por ejemplo, podemos especificar que los cambios debidos a la aparición del teclado, a su ocultación y al cambio de orientación de la pantalla los vamos a manejar nosotros.
      android:configChanges = "keyboard|keyboardHidden|orientation"
- <intent-filter>: Debe situarse dentro de un elemento <activity> e indica el modo en que ésta va a ser iniciada. Es decir, le indica a Android cómo debe reaccionar la Activity cuando se lleve a cabo algún intent específico sobre ella. Dentro de este elemento incluimos las diferentes acciones que deben ponerse en marcha.
    - <action>: especifica el punto de entrada principal de nuestra aplicación.
      <action android:name = " android.intent.action.MAIN" />
    - <category>: indica que la activity sea añadida al lanzador de la aplicación.
      <category android:name = "android.intent.category.LAUNCHER" />
- <uses-permission>: Mediante este elemento 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 la activity va a acceder a la memoria externa del teléfono, a internet, etc.
      <uses-permission android:name= "android.permission.WRITE_EXTERNAL_STORAGE" />

2.- La clase Intent

- La clase ***Intent*** posibilita la comunicación entre las diferentes partes de una aplicación Android.
- Existen dos tipos de intents:
    - *intents explícitos*. Indican exactamente el componente a lanzar. Se utilizan para ir ejecutando los componentes internos de la aplicación.
        - Por ejemplo, para lanzar una Activity desde otra.
    - *intents implícitos*. Se utilizan para solicitar tareas abstractas del tipo "hacer una foto" ó "enviar un mensaje".
- Cualquier activity definida en AndroidManifest puede ser iniciada programáticamente mediante un *Intent*.
    - La clase *Intent* tiene un **constructor** para hacer esto:
    - *Intent(Contexto , NombreActivity. class).* Ejemplo:
      Intent intent = new Intent (this , Actividad2.class);
      startActivity(intent);
      Donde *Actividad2* deberá estar declarada en *AndroidManifest*.

3.- Ciclo de vida de una Activity

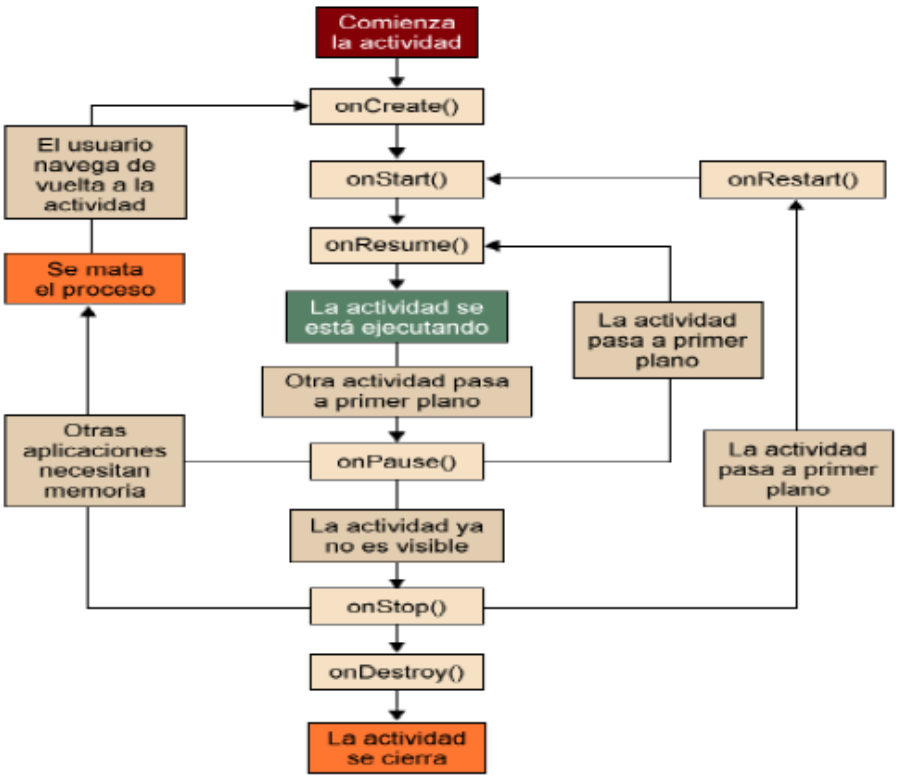
- Cuando se inicia una app compuesta por varias activitys se crea en la memoria del dispositivo una ***Activity stack*** (pila de activities).
    - Al ejecutar una *nueva activity* ⇒ se añade en la *cima de la pila* (**push**).
    - Al destruirla ⇒ se *quita de la pila* (**pop**) y se activa la siguiente.
- Las activitys de una app comparten una zona de memoria: *memory heap* .
    - Ojo con campos estáticos: no se destruyen con la Activity ⇒ ocupan memoria durante todo el tiempo de ejecución de la app.
- La relación entre Activitys-UIs no es siempre 1:1 (1:1, 1:N ó 1:0).

3.1.- Estados de una Activity

- El *Ciclo de vida de una Activity* describe los estados y las transiciones entre estados de una Activity durante el tiempo de ejecución de la app.
- Una activity puede encontrarse en 3 estados:
    - **Running** (ejecutándose). Está en la cima de la pila, ocupa toda la pantalla e interactúa con el usuario.
    - **Paused** (pausado). Activity todavía visible en la pantalla pero parcialmente oscurecida, porque:
        - Está en running otra actividad transparente
        - Aparece un diálogo
        - Pantalla del dispositivo está bloqueada.
    - **Stopped** (detenido). Activity completamente oscurecida por otra, deja de estar visible en la pantalla. Por ejemplo: se pulsa el botón *home* del dispositivo.
- Una Activity en estado *paused/stopped* puede ser destruida por Android en cualquier momento, por ejemplo por falta de memoria.
- Mientras una Activity está *paused/stopped* sigue ocupando memoria y está en espera de volver al estado running.
    - Si vuelve a *running*, sigue siendo el mismo objeto Java en la memoria ⇒ sus atributos serán los mismos que tenía cuando la Activity pasó a *paused/stopped*.

3.2.- Métodos protegidos

- Las ***Activitys*** cuentan con un conjunto de ***métodos protegidos*** que se ejecutan cuando tiene lugar un cambio de estado (pueden ser sobreescritos):
    - *onCreate*(): se inicia por primera vez la activity.
    - *onStart*(): después de onCreate() o al volver de un estado *stopped*
    - *onResume*(): después de onStart() o al volver de un estado *paused* .
    - *onPause*(): cuando otra actividad pasa a primer plano (modo *paused*).
        - Puede ser el último método de la Activity que se ejecute antes de ser destruida ⇒ guardar info que queramos que persista.
    - *onStop*(): cuando la Activity pasa a modo *stopped* .
        - siempre después de *onPause*() ⇒ antes de estar detenida (*stopped*) tiene que pasar por el estado *paused* .
        - Al igual que *onPause*() , puede ser el último método que se ejecute antes de que Android destruya la Activity.
    - *onDestroy*(): final del ciclo de vida de la Activity ⇒ último método que ejecutará Android antes de que la Activity sea destruida (se libera su espacio en la *memory heap* de la app).
        - Guardar información para próximas ejecuciones de la app.
        - Android puede haber destruido la Activity previamente.



3.3.- Aspectos destacados del ciclo de vida

1. Antes de entrar en estado *running* ⇒llamada a *onResume*()
2. Después de *onPause*() , no se puede asumir llamadas futuras a *onStop*() u *onDestroy*()⇒ No utilizar *onStop*()/*onDestroy*() para guardar información para futuras ejecuciones de la Activity.
5. No sabemos si una Activity en *paused* o *stopped* ha sido eliminada de la pila por Android, pero podemos utilizar el método *isFinishing*() :
   a. Puede ser llamado en cualquier momento y nos informa si la Activity va a ser eliminada.
   b. Si *isFinishing*() devuelve true entonces debemos salvar cualquier estado que queramos que persista para que no se pierda.

4.- Paso de parámetros entre Activitys

4.1.- Métodos putExtra() y get...Extra()

- Para enviar datos de tipo primitivo[1] de una *Activity A1* a otra *Activity A2* se utiliza el método *putExtra("nombre "," valor")* de la clase Intent :
  - *Parámetros*: identificador y dato, se envían en forma de par nombre/valor a la Activity llamada desde el Intent.
  - El tipo del dato vendrá indicado por su formato en el caso de ser una constante o por el tipo del atributo que lo contenga.
    Ejemplo:
    Intent intent = new Intent(this , Actividad2.class);
    intent. putExtra("entero",19) ;
    startActivity(intent);

[1] *Para enviar objetos o tipos complejos se puede implementar la clase Serializable, como en este ejemplo.*

- Para recuperar en *A2* los datos pasados por *A1* utilizaremos los métodos *get...Extra():*
    - Estos métodos pertenecen al método *getIntent*() y permiten recuperar la información enviada dependiendo del tipo de dato.
    - Si *get...Extra()* va a recibir un String admite como parámetro un identificador. Si va a recibir un dato numérico admite como parámetros un identificador y un valor por defecto. Ejemplo:
      int n um = getIntent().getIntExtra("entero",0) ;

Más información sobre las Activities en la web de referencia:

https://developer.android.com/guide/components/activities/intro-activities?hl=es

translated by Google
Se usó la API de Cloud Translation (//cloud.google.com/translate/?hl=es-419) para traducir esta página.

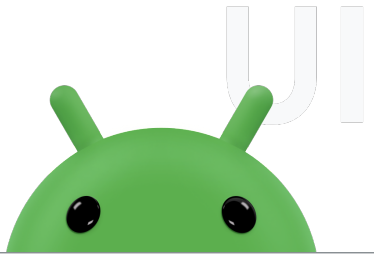
Switch to English

Cómo agregar botones de activación

Prueba la forma de Compose

Jetpack Compose es el kit de herramientas de IU recomendado para Android. Aprende a agregar componentes en Compose.

Interruptor → (https://developer.android.com/develop/ui/compose/components/switch?hl=es-419)



Si utilizas un diseño basado en `View`, hay tres opciones principales para implementar botones de activación. Te recomendamos que uses el componente `SwitchMaterial` (https://m3.material.io/components/switch/overview) de la biblioteca de Material Components (https://m3.material.io/develop/android/mdc-android):

```
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://scher
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="16dp">

    <com.google.android.material.switchmaterial.SwitchMaterial
        android:id="@+id/material_switch"
```

```
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/material_switch"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

Es posible que las apps heredadas sigan usando el componente `SwitchCompat` (https://developer.android.com/reference/androidx/appcompat/widget/SwitchCompat?hl=es-419) AppCompat más antiguo, como se muestra en el siguiente ejemplo:

```
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://scher
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="16dp">

    <androidx.appcompat.widget.SwitchCompat
        android:id="@+id/switchcompat"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/switchcompat"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

En el siguiente ejemplo, se muestra `AppCompatToggleButton` (https://developer.android.com/reference/androidx/appcompat/widget/AppCompatToggleButton?hl=es-419) , que es otro componente heredado que tiene una IU notablemente diferente:

```
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://scher
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="16dp">
```

```
    <TextView
        android:id="@+id/toggle_button_label"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintEnd_toStartOf="@id/toggle"
        app:layout_constraintHorizontal_chainStyle="packed"
        app:layout_constraintBaseline_toBaselineOf="@id/toggle"
        android:text="@string/toggle_button" />

    <androidx.appcompat.widget.AppCompatToggleButton
        android:id="@+id/toggle"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toEndOf="@id/toggle_button_label"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintBottom_toBottomOf="parent"/>

</androidx.constraintlayout.widget.ConstraintLayout>
```

Estos tres componentes ofrecen el mismo comportamiento, pero se ven diferentes. Las diferencias entre `SwitchMaterial` y `SwitchCompat` son sutiles, pero `AppCompatToggleButton` es muy diferente:

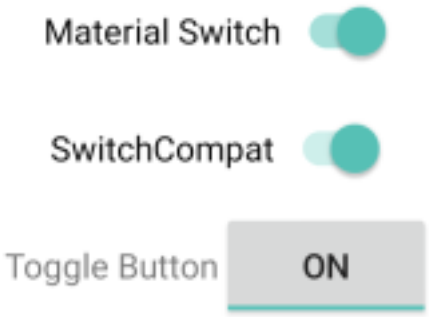


Figura 1: *Tres tipos de botones de activación.*

Cómo controlar los cambios de estado

`SwitchMaterial`, `SwitchCompat` y `AppCompatToggleButton` son subclases de `CompoundButton` (https://developer.android.com/reference/android/widget/CompoundButton?hl=es-419), lo que les

brinda un mecanismo común para controlar los cambios de estado verificados. Implementas una instancia de `CompoundButton.OnCheckedChangeListener` (https://developer.android.com/reference/android/widget/CompoundButton.OnCheckedChangeListener?hl=es-419) y la agregas al botón, como se muestra en el siguiente ejemplo:

Kotlin (#kotlin) Java (#java)

```
public class MainActivity extends AppCompatActivity {

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

        SwitchLayoutBinding binding = SwitchLayoutBinding.inflate(getLay
        setContentView(binding.getRoot());

        binding.materialSwitch.setOnCheckedChangeListener((buttonView, i
            if (isChecked) {
                // The switch is checked.
            } else {
                // The switch isn't checked.
            }
        });
    }
}
```

`CompoundButton.OnCheckedChangeListener` es una interfaz de método único (o *interfaz SAM*), por lo que puedes implementarla como una lambda. Se llama a la expresión lambda cada vez que cambia el estado de marcado, y el valor del booleano `isChecked` que se pasa a la expresión lambda indica el nuevo estado de marcado.

El contenido y las muestras de código que aparecen en esta página están sujetas a las licencias que se describen en la Licencia de Contenido (https://developer.android.com/license?hl=es-419). Java y OpenJDK son marcas registradas de Oracle o sus afiliados.

Última actualización: 2025-07-27 (UTC)

View Group.LayoutParams

Added in API level 1 (/guide/topics/manifest/uses-sdk-element#ApiLevels)

Kotlin (/reference/kotlin/android/view/ViewGroup.LayoutParams) | **Java**

```
public static class ViewGroup.LayoutParams
extends Object (/reference/java/lang/Object)
```

java.lang.Object (/reference/java/lang/Object)
↳ android.view.ViewGroup.LayoutParams

Known direct subclasses
AbsListView.LayoutParams (/reference/android/widget/AbsListView.LayoutParams), AbsoluteLayout.LayoutParams (/reference/android/widget/AbsoluteLayout.LayoutParams), Gallery.LayoutParams (/reference/android/widget/Gallery.LayoutParams), ViewGroup.MarginLayoutParams (/reference/android/view/ViewGroup.MarginLayoutParams), WindowManager.LayoutParams (/reference/android/view/WindowManager.LayoutParams)

AbsListView.LayoutParams (/reference/android/widget/AbsListView.LayoutParams)	AbsListView extends LayoutParams to provide a place to hold the view type.
AbsoluteLayout.LayoutParams (/reference/android/widget/AbsoluteLayout.LayoutParams)	Per-child layout information associated with AbsoluteLayout.
Gallery.LayoutParams (/reference/android/widget/Gallery.LayoutParams)	Gallery extends LayoutParams to provide a place to hold current Transformation information along with previous position/transformation info.
ViewGroup.MarginLayoutParams (/reference/android/view/ViewGroup.MarginLayoutParams)	Per-child layout information for layouts that support margins.
WindowManager.LayoutParams (/reference/android/view/WindowManager.LayoutParams)	

Known indirect subclasses
ActionBar.LayoutParams (/reference/android/app/ActionBar.LayoutParams), ActionMenuView.LayoutParams (/reference/android/widget/ActionMenuView.LayoutParams), FrameLayout.LayoutParams (/reference/android/widget/FrameLayout.LayoutParams), GridLayout.LayoutParams (/reference/android/widget/GridLayout.LayoutParams), LinearLayout.LayoutParams (/reference/android/widget/LinearLayout.LayoutParams),

RadioGroup.LayoutParams (/reference/android/widget/RadioGroup.LayoutParams), RelativeLayout.LayoutParams (/reference/android/widget/RelativeLayout.LayoutParams), TableLayout.LayoutParams (/reference/android/widget/TableLayout.LayoutParams), TableRow.LayoutParams (/reference/android/widget/TableRow.LayoutParams), Toolbar.LayoutParams (/reference/android/widget/Toolbar.LayoutParams)

ActionBar.LayoutParams (/reference/android/app/ActionBar.LayoutParams)	Per-child layout information associated with action bar custom views.
ActionMenuView.LayoutParams (/reference/android/widget/ActionMenuView.LayoutParams)	
FrameLayout.LayoutParams (/reference/android/widget/FrameLayout.LayoutParams)	Per-child layout information for layouts that support margins.
GridLayout.LayoutParams (/reference/android/widget/GridLayout.LayoutParams)	Layout information associated with each of the children of a GridLayout.
LinearLayout.LayoutParams (/reference/android/widget/LinearLayout.LayoutParams)	Per-child layout information associated with ViewLinearLayout.
RadioGroup.LayoutParams (/reference/android/widget/RadioGroup.LayoutParams)	This set of layout parameters defaults the width and the height of the children to `ViewGroup.LayoutParams.WRAP_CONTENT` (/reference/android/view/ViewGroup.LayoutParams#WRAP_CONTENT) when they are not specified in the XML file.
RelativeLayout.LayoutParams (/reference/android/widget/RelativeLayout.LayoutParams)	Specifies how a view is positioned within a `RelativeLayout` (/reference/android/widget/RelativeLayout).
TableLayout.LayoutParams (/reference/android/widget/TableLayout.LayoutParams)	This set of layout parameters enforces the width of each child to be `ViewGroup.LayoutParams.MATCH_PARENT` (/reference/android/view/ViewGroup.LayoutParams#MATCH_PARENT) and the height of each child to be `ViewGroup.LayoutParams.WRAP_CONTENT` (/reference/android/view/ViewGroup.LayoutParams#WRAP_CONTENT), but only if the height is not specified.
TableRow.LayoutParams (/reference/android/widget/TableRow.LayoutParams)	Set of layout parameters used in table rows.
Toolbar.LayoutParams (/reference/android/widget/Toolbar.LayoutParams)	Layout information for child views of Toolbars.

LayoutParams are used by views to tell their parents how they want to be laid out. See `ViewGroup_Layout_Attributes` (/reference/android/R.styleable#ViewGroup_Layout) for a list of all child view attributes that this class supports.

The base LayoutParams class just describes how big the view wants to be for both width and height. For each dimension, it can specify one of:

- FILL_PARENT (renamed MATCH_PARENT in API Level 8 and higher), which means that the view wants to be as big as its parent (minus padding)
- WRAP_CONTENT, which means that the view wants to be just big enough to enclose its content (plus padding)
- an exact number

There are subclasses of LayoutParams for different subclasses of ViewGroup. For example, AbsoluteLayout has its own subclass of LayoutParams which adds an X and Y value.

Developer Guides

For more information about creating user interface layouts, read the XML Layouts (/guide/topics/ui/declaring-layout) developer guide.

Summary

XML attributes

`android:layout_height` (/reference/android/view/ViewGroup.LayoutParams#attr_android:layout_he

`android:layout_width` (/reference/android/view/ViewGroup.LayoutParams#attr_android:layout_wid

Constants

`int`	`FILL_PARENT` (/reference/android/view/ViewGroup.LayoutParams#FILL_PARENT) Special value for the height or width requested by a View.
`int`	`MATCH_PARENT` (/reference/android/view/ViewGroup.LayoutParams#MATCH_PARENT) Special value for the height or width requested by a View.
`int`	`WRAP_CONTENT` (/reference/android/view/ViewGroup.LayoutParams#WRAP_CONTENT) Special value for the height or width requested by a View.

Fields

`public int`	`height` (/reference/android/view/ViewGroup.LayoutParams#height) Information about how tall the view wants t
`public LayoutAnimationController.AnimationParameters` (/reference/android/view/animation/LayoutAnimationController.AnimationParameters)	`layoutAnimationParameters` (/reference/android/view/ViewGroup.LayoutParams#layoutAnimation Used to animate layouts.
`public int`	`width` (/reference/android/view/ViewGroup.LayoutParams#width) Information about how wide the view wants

Public constructors

`LayoutParams` (/reference/android/view/ViewGroup.LayoutParams#LayoutParams(android.content.Context,%20android.util.AttributeSet))
`(Context` (/reference/android/content/Context) `c, AttributeSet` (/reference/android/util/AttributeSet) `attrs)`

Creates a new set of layout parameters.

`LayoutParams` (/reference/android/view/ViewGroup.LayoutParams#LayoutParams(android.view.ViewGroup.LayoutParams))
`(ViewGroup.LayoutParams` (/reference/android/view/ViewGroup.LayoutParams) `source)`

Copy constructor.

`LayoutParams` (/reference/android/view/ViewGroup.LayoutParams#LayoutParams(int,%20int))`(int width, int height)`

Creates a new set of layout parameters with the specified width and height.

Public methods

`void`	`resolveLayoutDirection` (/reference/android/view/ViewGroup.LayoutParams#resolveLayoutDirection(int)) `(int layoutDirection)` Resolve layout parameters depending on the layout direction.

Protected methods

`void` `setBaseAttributes` (/reference/android/view/ViewGroup.LayoutParams#setBaseAttributes(android.content.res.TypedArray,%20int,%20int))
`(TypedArray` (/reference/android/content/res/TypedArray) `a, int widthAttr, int heightAttr)`

Extracts the layout parameters from the supplied attributes.

Inherited methods

From class `java.lang.Object` (/reference/java/lang/Object)

`Object` (/reference/java/lang/Object)	`clone` (/reference/java/lang/Object#clone())`()` Creates and returns a copy of this object.
`boolean`	`equals` (/reference/java/lang/Object#equals(java.lang.Object))`(Object` (/reference/java/lang/Object) `obj)` Indicates whether some other object is "equal to" this one.
`void`	`finalize` (/reference/java/lang/Object#finalize())`()` Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
`final Class` (/reference/java/lang/Class)`<?>`	`getClass` (/reference/java/lang/Object#getClass())`()` Returns the runtime class of this `Object`.
`int`	`hashCode` (/reference/java/lang/Object#hashCode())`()` Returns a hash code value for the object.
`final void`	`notify` (/reference/java/lang/Object#notify())`()` Wakes up a single thread that is waiting on this object's monitor.
`final void`	`notifyAll` (/reference/java/lang/Object#notifyAll())`()` Wakes up all threads that are waiting on this object's monitor.
`String` (/reference/java/lang/String)	`toString` (/reference/java/lang/Object#toString())`()`

	Returns a string representation of the object.
`final void`	`wait` (/reference/java/lang/Object#wait(long,%20int)) `(long timeoutMillis, int nanos)` Causes the current thread to wait until it is awakened, typically by being *notified* or *interrupted*, or until a certain amount of real time has elapsed.
`final void`	`wait` (/reference/java/lang/Object#wait(long)) `(long timeoutMillis)` Causes the current thread to wait until it is awakened, typically by being *notified* or *interrupted*, or until a certain amount of real time has elapsed.
`final void`	`wait` (/reference/java/lang/Object#wait()) `()` Causes the current thread to wait until it is awakened, typically by being *notified* or *interrupted*.

XML attributes

android:layout_height

Specifies the basic height of the view. This is a required attribute for any view inside of a containing layout manager. Its value may be a dimension (such as "12dip") for a constant height or one of the special constants.

May be a dimension value, which is a floating point number appended with a unit such as "`14.5sp`". Available units are: px (pixels), dp (density-independent pixels), sp (scaled pixels based on preferred font size), in (inches), and mm (millimeters).

Must be one of the following constant values.

Constant	ValueDescription

fill_parent	ffffffff	The view should be as big as its parent (minus padding). This constant is deprecated starting from API Level 8 and is replaced by `match_parent`.
match_parent	ffffffff	The view should be as big as its parent (minus padding). Introduced in API Level 8.
wrap_content	fffffffe	The view should be only big enough to enclose its content (plus padding).

android:layout_width

Specifies the basic width of the view. This is a required attribute for any view inside of a containing layout manager. Its value may be a dimension (such as "12dip") for a constant width or one of the special constants.

May be a dimension value, which is a floating point number appended with a unit such as "`14.5sp`". Available units are: px (pixels), dp (density-independent pixels), sp (scaled pixels based on preferred font size), in (inches), and mm (millimeters).

Must be one of the following constant values.

Constant	Value	Description
fill_parent	ffffffff	The view should be as big as its parent (minus padding). This constant is deprecated starting from API Level 8 and is replaced by `match_parent`.
match_parent	ffffffff	The view should be as big as its parent (minus padding). Introduced in API Level 8.
wrap_content	fffffffe	The view should be only big enough to enclose its content (plus padding).

Constants

FILL_PARENT

Added in API level 1 (/guide/topics/manifest/uses-sdk-element#ApiLevels)
Deprecated in API level 15 (/guide/topics/manifest/uses-sdk-element#ApiLevels)

```
public static final int FILL_PARENT
```

Special value for the height or width requested by a View. FILL_PARENT means that the view wants to be as big as its parent, minus the parent's padding, if any. This value is deprecated starting in API Level 8 and replaced by `MATCH_PARENT` (/reference/android/view/ViewGroup.LayoutParams#MATCH_PARENT).

Constant Value: -1 (0xffffffff)

MATCH_PARENT

Added in API level 8 (/guide/topics/manifest/uses-sdk-element#ApiLevels)

```
public static final int MATCH_PARENT
```

Special value for the height or width requested by a View. MATCH_PARENT means that the view wants to be as big as its parent, minus the parent's padding, if any. Introduced in API Level 8.

Constant Value: -1 (0xffffffff)

WRAP_CONTENT

Added in API level 1 (/guide/topics/manifest/uses-sdk-element#ApiLevels)

```
public static final int WRAP_CONTENT
```

Special value for the height or width requested by a View. WRAP_CONTENT means that the view wants to be just large enough to fit its own internal content, taking its own padding into account.

Constant Value: -2 (0xfffffffe)

Fields

height

Added in API level 1 (/guide/topics/manifest/uses-sdk-element#ApiLevels)

```
public int height
```

Information about how tall the view wants to be. Can be one of the constants FILL_PARENT (replaced by MATCH_PARENT in API Level 8) or WRAP_CONTENT, or an exact size.

layoutAnimationParameters

Added in API level 1 (/guide/topics/manifest/uses-sdk-element#ApiLevels)

```
public LayoutAnimationController.AnimationParameters (/reference/android/view/animatic
```

Used to animate layouts.

width

Added in API level 1 (/guide/topics/manifest/uses-sdk-element#ApiLevels)

```
public int width
```

Information about how wide the view wants to be. Can be one of the constants FILL_PARENT (replaced by MATCH_PARENT in API Level 8) or WRAP_CONTENT, or an exact size.

Public constructors

LayoutParams

Added in API level 1 (/guide/topics/manifest/uses-sdk-element#ApiLevels)

```
public LayoutParams (Context (/reference/android/content/Context) c,
                AttributeSet (/reference/android/util/AttributeSet) attrs)
```

Creates a new set of layout parameters. The values are extracted from the supplied attributes set and context. The XML attributes mapped to this set of layout parameters are:

- `layout_width`: the width, either an exact value, `WRAP_CONTENT` (/reference/android/view/ViewGroup.LayoutParams#WRAP_CONTENT), or `FILL_PARENT` (/reference/android/view/ViewGroup.LayoutParams#FILL_PARENT) (replaced by `MATCH_PARENT` (/reference/android/view/ViewGroup.LayoutParams#MATCH_PARENT) in API Level 8)
- `layout_height`: the height, either an exact value, `WRAP_CONTENT` (/reference/android/view/ViewGroup.LayoutParams#WRAP_CONTENT), or `FILL_PARENT` (/reference/android/view/ViewGroup.LayoutParams#FILL_PARENT) (replaced by `MATCH_PARENT` (/reference/android/view/ViewGroup.LayoutParams#MATCH_PARENT) in API Level 8)

Parameters

`c`	`Context`: the application environment
`attrs`	`AttributeSet`: the set of attributes from which to extract the layout parameters' values

LayoutParams

Added in API level 1 (/guide/topics/manifest/uses-sdk-element#ApiLevels)

```
public LayoutParams (ViewGroup.LayoutParams (/reference/android/view/ViewGroup.LayoutP
```

Copy constructor. Clones the width and height values of the source.

Parameters

`source`	`ViewGroup.LayoutParams`: The layout params to copy from.

LayoutParams

Added in API level 1 (/guide/topics/manifest/uses-sdk-element#ApiLevels)

```
public LayoutParams (int width,
                int height)
```

Creates a new set of layout parameters with the specified width and height.

Parameters

`width`	`int`: the width, either `WRAP_CONTENT` (/reference/android/view/ViewGroup.LayoutParams#WRAP_CONTENT), `FILL_PARENT` (/reference/android/view/ViewGroup.LayoutParams#FILL_PARENT) (replaced by `MATCH_PARENT` (/reference/android/view/ViewGroup.LayoutParams#MATCH_PARENT) in API Level 8), or a fixed size in pixels

---

`height`	`int`: the height, either `WRAP_CONTENT` (/reference/android/view/ViewGroup.LayoutParams#WRAP_CONTENT), `FILL_PARENT` (/reference/android/view/ViewGroup.LayoutParams#FILL_PARENT) (replaced by `MATCH_PARENT` (/reference/android/view/ViewGroup.LayoutParams#MATCH_PARENT) in API Level 8), or a fixed size in pixels

Public methods

resolveLayoutDirection

Added in API level 17 (/guide/topics/manifest/uses-sdk-element#ApiLevels)

```
public void resolveLayoutDirection (int layoutDirection)
```

Resolve layout parameters depending on the layout direction. Subclasses that care about layoutDirection changes should override this method. The default implementation does nothing.

Parameters

`layoutDirection`	`int`: the direction of the layout `View.LAYOUT_DIRECTION_LTR` (/reference/android/view/View#LAYOUT_DIRECTION_LTR) `View.LAYOUT_DIRECTION_RTL` (/reference/android/view/View#LAYOUT_DIRECTION_RTL)

Protected methods

setBaseAttributes

```
protected void setBaseAttributes (TypedArray (/reference/android/content/res/TypedArray
                int widthAttr,
                int heightAttr)
```

Extracts the layout parameters from the supplied attributes.

Parameters

`a`	`TypedArray`: the style attributes to extract the parameters from
`widthAttr`	`int`: the identifier of the width attribute
`heightAttr`	`int`: the identifier of the height attribute

Content and code samples on this page are subject to the licenses described in the Content License (/license). Java and OpenJDK are trademarks or registered trademarks of Oracle and/or its affiliates.

Last updated 2025-02-10 UTC.