

CAPÍTULO 2.

Diseño de la interfaz de usuario: vistas y *layouts*

El diseño de la interfaz de usuario cobra cada día más importancia en el desarrollo de una aplicación. La calidad de la interfaz de usuario puede ser uno de los factores que conduzca al éxito o al fracaso de todo el proyecto.

Si has realizado alguna aplicación utilizando otras plataformas, advertirás que el diseño de la interfaz de usuario en Android sigue una filosofía muy diferente. En Android la interfaz de usuario no se diseña en código, si no utilizando un lenguaje de marcado en XML similar al HTML.

A lo largo de este capítulo mostraremos una serie de ejemplos que te permitirán entender el diseño de la interfaz de usuario en Android. Aunque no será la forma habitual de trabajar, comenzaremos creando la interfaz de usuario mediante código. De esta forma comprobaremos que cada uno de los elementos de la interfaz de usuario (las vistas) realmente son objetos Java. Continuaremos mostrando cómo se define la interfaz de usuario utilizando código XML. Pasaremos luego a ver las herramientas de diseño integradas en Android Studio. Se describirá el uso de *layouts*, que nos permitirá una correcta organización de las vistas, y el uso de recursos alternativos nos permitirá adaptar nuestra interfaz a diferentes circunstancias y tipos de dispositivos.

En este capítulo también comenzaremos creando la aplicación de ejemplo desarrollada a lo largo del curso, Asteroides. Crearemos la actividad principal, donde simplemente mostraremos cuatro botones, con los que se podrán arrancar diferentes actividades. A continuación aprenderemos a crear estilos y temas y los aplicaremos a estas actividades. Para terminar el capítulo propondremos varias prácticas para aprender a utilizar diferentes tipos de vistas y *layouts*.



Objetivos:

- Entender cómo se realiza el diseño de la interfaz de usuario en una aplicación Android.
- Aprender a trabajar con vistas y mostrar sus atributos más importantes.
- Enumerar los tipos de *layouts* que nos permitirán organizar las vistas.
- Mostrar cómo se utilizan los recursos alternativos.
- Aprender a crear estilos y temas para personalizar nuestras aplicaciones.
- Mostrar cómo interactuar con las vistas desde el código Java o Kotlin.
- Describir el uso de *layouts* basados en pestañas (*tabs*).

2.1. Creación de una interfaz de usuario por código

Veamos un primer ejemplo de cómo crear una interfaz de usuario utilizando exclusivamente código Java. Aunque esta no es la forma recomendable de trabajar con Android, resulta interesante para resaltar algunos conceptos.



Ejercicio: Creación de la interfaz de usuario por código

1. Abre el proyecto creado en el capítulo anterior y visualiza *MainActivity.java*.
2. Comenta la última sentencia del método `onCreate()` y añade las subrayadas:

```
@Override public void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    //setContentView(R.layout.activity_main);  
    TextView texto = new TextView(this);  
    texto.setText("Hola, Android");  
    setContentView(texto);  
}
```

```
override fun onCreate(savedInstanceState: Bundle?) {  
    super.onCreate(savedInstanceState)  
    //setContentView(R.layout.activity_main)  
    val texto = TextView(this)  
    texto.text="Hola, Android"  
    setContentView(texto)  
}
```



Nota sobre Java/Kotlin: Para poder utilizar el objeto `TextView` has de importar un nuevo paquete. Para ello añade al principio `import android.widget.TextView;`. Otra alternativa es pulsar **Alt-Intro** para que se añadan automáticamente los paquetes que faltan.

La interfaz de usuario de Android está basada en una jerarquía de clases descendientes de la clase `View` (vista). Una vista es un objeto que se puede dibujar y se utiliza como un elemento en el diseño de la interfaz de usuario (un botón, una imagen, una etiqueta de texto como la que se ha utilizado en el ejemplo, etc.). Cada uno de estos elementos se define como una subclase de la clase `View`; la subclase para representar un texto es `TextView`.

El ejemplo comienza creando un objeto de la clase `TextView`. El constructor de la clase acepta como parámetro una instancia de la clase `Context` (contexto). Un contexto es un manejador del sistema que proporciona servicios como la resolución de recursos, la obtención de acceso a bases de datos o las preferencias. La clase `Activity` es una subclase de `Context`, y como la clase `MainActivity` es una subclase de `Activity`, también es de tipo `Context`. Por ello, puedes pasar `this` (el objeto actual de la clase `MainActivity`) como contexto del `TextView`.

3. Después se define el texto que se visualizará en el `TextView` mediante `setText()`. Finalmente, mediante `setContentView()` se indica la vista que visualizará la actividad.
4. Ejecuta el proyecto para verificar que funciona.

2.2. Creación de una interfaz de usuario usando XML

En el ejemplo anterior hemos creado la interfaz de usuario directamente en el código. A veces puede ser muy complicado programar interfaces de usuario, ya que pequeños cambios en el diseño pueden corresponder a complicadas modificaciones en el código. Un principio importante en el diseño de *software* es que conviene separar todo lo posible el diseño, los datos y la lógica de la aplicación.

Android proporciona una alternativa para el diseño de interfaces de usuario: los ficheros de diseño basados en XML. Veamos uno de estos ficheros. Para ello accede al fichero `res/layout/activity_main.xml` de nuestro proyecto. Se muestra a continuación. Este `layout` o fichero de diseño proporciona un resultado similar al del ejemplo de diseño por código anterior:

```
<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">
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

NOTA: Cuando haces doble clic en el explorador del proyecto sobre `activity_main.xml`, probablemente lo abra en modo gráfico. Para verlo en modo texto, selecciona la pestaña Text.

Resulta sencillo interpretar su significado. Se introduce un elemento de tipo `ConstraintLayout`, cuya función, como se estudiará más adelante, es contener otros elementos de tipo `View`. Este `ConstraintLayout` tiene seis atributos. Los tres primeros, `xmlns:android`, `xmlns:app` y `xmlns:tools`, son declaraciones de espacios de nombres de XML que utilizaremos en este fichero (este tipo de parámetro solo es necesario especificarlo en el primer elemento). Los dos siguientes permiten definir la anchura y altura de la vista. En el ejemplo se ocupará todo el espacio disponible. El último atributo indica la actividad asociada a este `layout`.

Dentro del `ConstraintLayout` solo tenemos un elemento de tipo `TextView`. Este dispone de varios atributos. Los dos primeros definen el alto y el ancho (se ajustarán al texto contenido). El siguiente indica el texto a mostrar. Los cuatro siguientes indican la posición de la vista dentro del `ConstraintLayout`.



Ejercicio: Creación de la interfaz de usuario con XML

1. Para utilizar el diseño en XML regresa al fichero `MainActivity.java` y deshaz los cambios que hicimos antes (elimina las tres últimas líneas y quita el comentario).
2. Ejecuta la aplicación y verifica el resultado. Ha de ser muy similar al anterior.
3. Modifica el valor de `hello_world` en el fichero `res/values/strings.xml`.
4. Vuelve a ejecutar la aplicación y visualiza el resultado.

Analicemos ahora la línea en la que acabas de quitar el comentario:

```
setContentView(R.layout.activity_main)
```

Aquí, `R.layout.main` corresponde a un objeto `View` que se creará en tiempo de ejecución a partir del recurso `activity_main.xml`. Trabajar de esta forma, en comparación con el diseño basado en código, no quita velocidad y requiere menos memoria. Este identificador es creado automáticamente en la clase `R` del proyecto a partir de los elementos de la carpeta `res`. La definición de la clase `R` puede ser similar a:

```
public final class R {
    public static final class attr {
    }
    public static final class drawable {
        public static final int ic_launcher=0x7f020000;
    }
}
```

```
public static final class id {
    public static final int action_settings=0x7f070000;
}
public static final class layout {
    public static final int activity_main=0x7f030000;
}
public static final class menu {
    public static final int main=0x7f060000;
}
public static final class string {
    public static final int app_name=0x7f040000;
}
...
```

NOTA: Este fichero se genera automáticamente. Nunca debes editarlo.

Has de tener claro que los identificadores de la clase **R** son meros números que informan al gestor de recursos sobre qué datos ha de cargar. Por lo tanto, no se trata de verdaderos objetos; estos se crearán en tiempo de ejecución solo cuando sea necesario usarlos.



Ejercicio: El fichero R.java

1. En **Android Studio**, el fichero R.java no es accesible desde el explorador del proyecto. No obstante, puedes acceder a él si pulsas con el botón derecho sobre app y seleccionas *Show in Explorer* (o *Show in Dolphin*). Desde esta carpeta abre el fichero:

```
app\build\generated\not_namespaced_r_class_sources\debug\r
\nombre\del\paquete\R.java
```

Donde nombre\del\paquete has de reemplazarlo por el que corresponda al paquete de tu aplicación.

2. Compáralo con el fichero mostrado previamente. ¿Qué diferencias encuentras? (**RESPUESTA:** cambian los valores numéricos en hexadecimal y contiene muchos más identificadores.)
3. Abre el fichero *MainActivity.java* y reemplaza `R.layout.activity_main` por el valor numérico al que corresponde en *R.java*.
4. Ejecuta de nuevo el proyecto. ¿Funciona? ¿Crees que sería adecuado dejar este valor numérico?
5. Aunque haya funcionado, este valor puede cambiar en un futuro. Por lo tanto, para evitar problemas futuros vuelve a reemplazarlo por `R.layout.activity_main`.



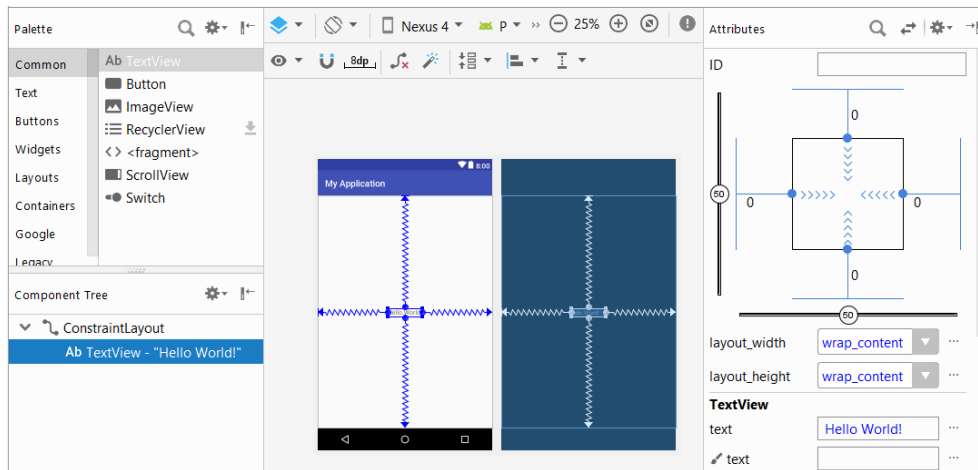
Preguntas de repaso: Interfaz de usuario en XML y en código



Preguntas de repaso: El fichero R.java







2.2.1. Edición visual de las vistas

Veamos ahora como editar los *layouts* o ficheros de diseño en XML. En el explorador del proyecto abre el fichero *res/layout/activity_main.xml*. Verás que en la parte inferior de la ventana central aparecen dos lengüetas: *Design* y *Text*. Podrás usar dos tipos de diseño: editar directamente el código XML (lengüeta *Text*) o realizar este diseño de forma visual (lengüeta *Design*). Veamos cómo se realizaría el diseño visual. La herramienta de edición de layouts se muestra a continuación:



NOTA: Si aparece un error con problemas de renderizado prueba otros niveles de API, en el desplegable que aparece junto al pequeño robot verde, o con otro tema, en el botón con forma de círculo.

En la parte inferior izquierda encontramos el marco **Component Tree** con una lista con todos los elementos del **layout**. Este **layout** tiene solo dos vistas: un **ConstraintLayout** que contiene un **TextView**. En el marco central aparece una representación de cómo se verá el resultado y a su derecha, con fondo azul, una representación con los nombres de cada vista y su tamaño. En la parte superior aparecen varios controles para representar este **layout** en diferentes configuraciones. Cuando diseñamos una vista en Android, hay que tener en cuenta que desconocemos el dispositivo final donde se visualizará y la configuración específica elegida por el usuario. Por esta razón, resulta importante que verifiques que el **layout** se ve de forma adecuada en cualquier configuración.

En la parte superior, de izquierda a derecha, encontramos los siguientes botones: El primero  permite mostrar solo la visualización de diseño, solo la visualización esquemática de vistas o ambas. El botón  muestra la orientación horizontal (*landscape*), vertical (*portrait*) y también podemos escoger el tipo de interfaz de usuario (coche, TV, reloj...), con  **Nexus 4** escogemos el tipo de dispositivo (tamaño y resolución de la pantalla), con  **24** la versión de Android, con  **AppTheme** cómo se verá nuestra vista tras aplicar un tema y con  **Default (en-us)** editar las traducciones.

Para editar un elemento, selecciónalo en el marco **Component Tree** o pincha directamente sobre él en la ventana de previsualización. Al seleccionarlo, puedes modificar alguna de sus propiedades en el marco **Properties**, situado a la derecha. Echa un vistazo a las propiedades disponibles para **TextView** y modifica alguna de ellas. En muchos casos te aparecerá un desplegable con las opciones disponibles. Aquí solo se muestra una pequeña parte de las propiedades disponibles. Pulsa en **View all properties** para mostrarlas todas.

El marco de la izquierda, **Palette**, te permite insertar de forma rápida nuevas vistas al **layout**. Puedes arrastrar cualquier elemento a la ventana de previsualización o al marco **Component Tree**. En el anexo D se ha incluido una lista con las vistas disponibles.

NOTA: El siguiente vídeo corresponde a una versión anterior de la herramienta. Aunque cambian algunos iconos el funcionamiento continúa siendo similar. Para crear un nuevo **layout** pulsa con el botón derecho en el explorador de proyecto sobre **app** y selecciona la opción: **New > Android resource file**



Vídeo[tutorial]: **Diseño visual de layouts: visión general**



Ejercicio: Creación visual de vistas

1. Crea un nuevo proyecto con los siguientes datos:

Phone and Tablet / Empty Activity


Nombre: Primeras Vistas

Minimum API level: API 19 Android 4.4 (KitKat)

☒ Use androidx.* artifact

Deja el resto de los parámetros con los valores por defecto.

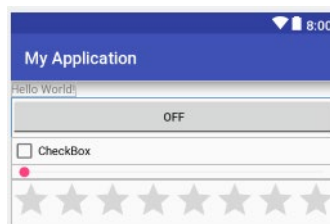
2. Abre el fichero `res/layout/activity_main.xml`.
3. Vamos a hacer que la raíz del `layout` se base en un `LinearLayout` vertical. Este tipo de `layout` es uno de los más sencillos de utilizar. Te permite representar las vistas una debajo de la otra. En el marco *Component Tree* pulsa con el botón derecho sobre `ConstraintLayout` y selecciona *Convert View...* Indica que quieres usar el `layout`, `LinearLayout`. El `layout` que ha añadido es de tipo horizontal. En nuestro caso lo queremos de tipo vertical, para cambiarlo pulsa con el botón derecho sobre `LinearLayout` y selecciona *LinearLayout/Convert orientation to vertical*.






Todas las operaciones que hacemos en modo diseño visual (lengüeta *Design*) también las podemos hacer con el editor de texto. Para probarlo, deshaz el trabajo anterior, pulsando el botón Undo  (Ctrl+Z). Selecciona la lengüeta *Text* y cambia la etiqueta `ConstraintLayout` por `LinearLayout`. Añade el atributo `orientation` a `LinearLayout` para que la orientación sea vertical. Elimina los atributos innecesarios del `TextView` que utiliza con `ConstraintLayout`. El resultado ha de ser similar a:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context=".MainActivity">
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        app:layout_constraintBottom_toBottomOf="parent" ... />
</LinearLayout>
```

Regresa a la lengüeta *Design*.

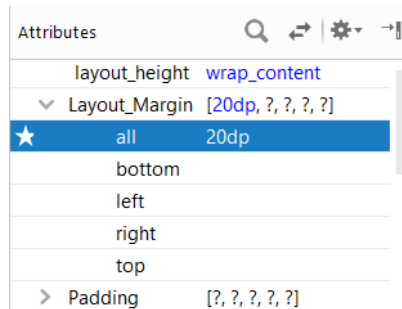
4. Desde la paleta de la izquierda arrastra, al área de diseño, los siguientes elementos: `ToggleButton`, `CheckBox`, `SeekBar` y `RatingBar`.



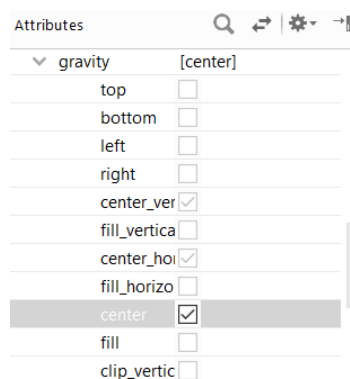
5. Selecciona la primera vista que estaba ya creada (`TextView`) y pulsa `<Supr>` para eliminarla.
6. Selecciona la vista `ToggleButton`. Pulsa ahora  (Set layout width to wrap_content). Conseguirás que la anchura del botón se ajuste a la anchura de su texto. Pulsa el botón  (Set layout width to match_parent) para que el ancho del botón se ajuste a su contenedor. Observa en el marco *Attributes* como cambia la propiedad `layout_width`. Si el botón anterior no funciona cámbialo desde este marco. Deja el valor `wrap_content`.
7. Pulsa el botón  (Convert orientation to horizontal), para conseguir que el `LinearLayout` donde están las diferentes vistas tenga una orientación horizontal. Comprobarás que no caben todos los elementos. Pulsa el botón  (Convert orientation to vertical), para volver a una orientación vertical.
8. Con la vista `ToggleButton` seleccionada, Pulsa el botón  (Set layout height to match_parent). Conseguirás que la altura del botón se ajuste a la altura de su contenedor. El problema es que el

resto de los elementos dejan de verse. Vuelve a pulsar este botón para regresar a la configuración anterior (también puedes pulsar *Ctrl-Z*).

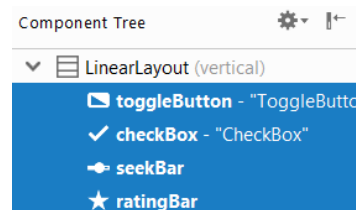
9. Selecciona la vista **CheckBox**. Ve al marco **Attributes** y en la parte inferior pulsa en **All attributes**. Busca la propiedad **layout_margin** en el campo **all** introduce "20dp". Se añadirá un margen alrededor de la vista.





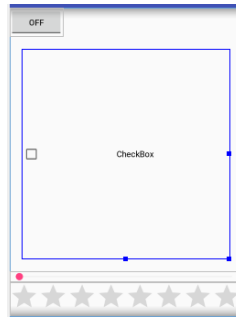
10. Busca la propiedad **gravity** y selecciona **center**.






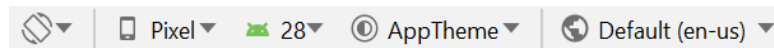
11. Observa que hay un espacio sin usar en la parte inferior del **layout**. Vamos a distribuir este espacio entre las vistas. Desde el marco **Commonnet Tree** selecciona las cuatro vistas que has introducido dentro del **LinearLayout**. Para una selección múltiple mantén pulsada la tecla **Ctrl**.




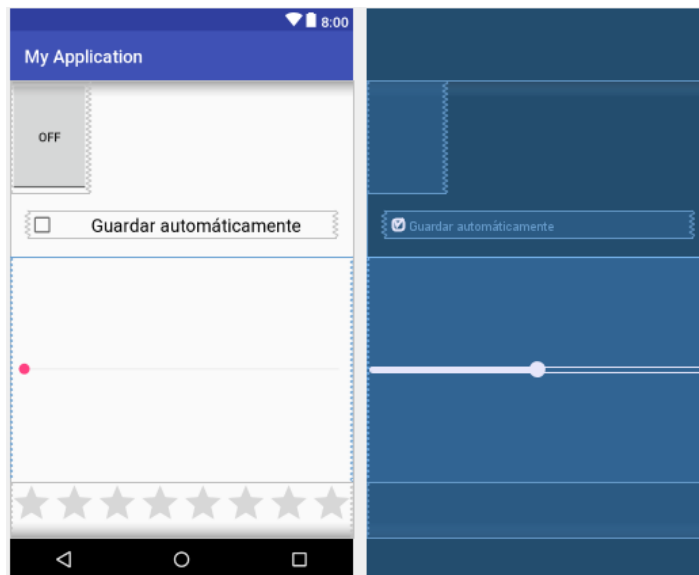
12. Aparecerá un nuevo botón:  (**Distribute Weights Evenly**). Púlsalo y la altura de las vistas se ajustará para que ocupen la totalidad del **layout**. Realmente, lo que hace es dividir el espacio sin usar de forma proporcional entre las vistas. Es equivalente a poner **layout_weight** = 1 y **layout_height** = 0dp para todas las vistas de este **layout**. Esta propiedad se modificará en un siguiente punto.
13. Selecciona las cuatro vistas y pulsa el botón  (**Clear All Weight**) para eliminar los pesos introducidos.
14. Selecciona la vista **CheckBox**. Asigna, en el marco **Attributes**, **layout_height** = 1 y **layout_weight** = 0dp en esta vista. Observa como toda la altura restante es asignada a la vista seleccionada.



15. Para asignar un peso diferente a cada vista, repite los pasos anteriores donde asignábamos peso 1 a todas las vistas (**botón**). Pula la lengüeta *Text* y modifica manualmente el atributo *layout_weight* para que el *ToggleButton* tenga valor 2; *CheckBox* tenga valor 0.5; *SeekBar* valor 4 y *RatingBar* valor 1. Pula la lengüeta *Design*. Como puedes observar, estos pesos permiten repartir la altura sobrante entre las vistas.
16. Utiliza los siguientes botones:  33%  , para ajustar el zum.
17. Utiliza los botones de la barra superior para observar cómo se representará el *layout* en diferentes situaciones y tipos de dispositivos:



18. Selecciona la vista *CheckBox* y observa las diferentes propiedades que podemos definir en el marco *Attributes*. Algunas ya han sido definidas por medio de la barra de botones. En concreto, y siguiendo el mismo orden que en los botones, hemos modificado: *Layout margin = 20dp*, *gravity = center* y *Layout weight = 0.5*.
19. Busca la propiedad *Text* y sustituye el valor *CheckBox* por “Guardar automáticamente” y *Text size* por “9pt”.
20. Pula el botón  para mostrar la visualización de diseño, la esquemática (Blueprint) o ambas. A continuación, se muestra el resultado final obtenido:



21. Pula sobre la lengüeta *Text*. Pula las teclas *Ctrl-Alt-L* para que formatee adecuadamente el código XML. A continuación, se muestra este código:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/activity_main"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
```



```
tools:context="com.example.primerasvistas.MainActivity">
<ToggleButton
    android:id="@+id/toggleButton"
    android:layout_width="wrap_content"
    android:layout_height="0dp"
    android:layout_weight="2"
    android:text="ToggleButton" />
<CheckBox
    android:id="@+id/checkBox"
    android:layout_width="match_parent"
    android:layout_height="0dp"
    android:layout_margin="20dp"
    android:layout_weight="0.5"
    android:gravity="center"
    android:text="Guardar automàticament"
    android:textSize="9pt" />
<SeekBar
    android:id="@+id/seekBar"
    android:layout_width="match_parent"
    android:layout_height="0dp"
    android:layout_weight="4" />
<RatingBar
    android:id="@+id/ratingBar"
    android:layout_width="match_parent"
    android:layout_height="0dp"
    android:layout_weight="1" />
</LinearLayout>
```

22. Ejecuta el proyecto para ver el resultado en el dispositivo.



Ejercicio: Vistas de entrada de texto

1. Añade en la parte superior del *layout* anterior una vista de tipo entrada de texto `EditText`, de tipo normal (*Plain Text*). Lo encontrarás dentro del grupo *Text*. Debajo de esta, una de tipo correo electrónico (*E-mail*) seguida de una de tipo palabra secreta (*Password*). Continúa así con otros tipos de entradas de texto.
2. Ejecuta la aplicación.
3. Observa como al introducir el texto de una entrada se muestra un tipo de teclado diferente.

2.2.2. Los atributos de las vistas



Vídeo[tutorial]: Atributos de la clase *View* en Android



Vídeo[tutorial]: Atributos de la clase *TextView* en Android



Recursos adicionales: Atributo de dimensión

En muchas ocasiones tenemos que indicar la anchura o altura de una vista, un margen, el tamaño de un texto o unas coordenadas. Este tipo de atributos se conocen como atributos de dimensión. Dado que nuestra aplicación podrá ejecutarse en una gran variedad de dispositivos con resoluciones muy

diversas, Android nos permite indicar estas dimensiones de varias formas. En la siguiente tabla se muestran las diferentes posibilidades:

- px** (píxeles): Estas dimensiones representan los píxeles en la pantalla.
- mm** (milímetros): Distancia real medida sobre la pantalla.
- in** (pulgadas): Distancia real medida sobre la pantalla.
- pt** (puntos): Equivale a 1/72 pulgadas.
- dp** (píxeles independientes de la densidad): Presupone un dispositivo de 160 píxeles por pulgada. Si luego el dispositivo tiene otra densidad, se realizará el correspondiente ajuste. A diferencia de otras medidas como mm, in y pt este ajuste se hace de forma aproximada dado que no se utiliza la verdadera densidad gráfica, si no el grupo de densidad en que se ha clasificado el dispositivo (ldpi, mdpi, hdpi...). Esta medida presenta varias ventajas cuando se utilizan recursos gráficos en diferentes densidades. Por esta razón, Google insiste en que se utilice siempre esta medida. Desde un punto de vista práctico un dp equivale aproximadamente a 1/160 pulgadas. Y en dispositivos con densidad gráfica mdpi un dp es siempre un pixel.
- sp** (píxeles escalados): Similar a dp, pero también se escala en función del tamaño de fuente que el usuario ha escogido en las preferencias. Indicado cuando se trabaja con fuentes.



Recursos adicionales: *Tipos de vista y sus atributos*

Consulta el anexo D para conocer una lista con todos los descendientes de la clase `View` y sus atributos.



Preguntas de repaso: *Las vistas y sus atributos*

2.3. Layouts

Si queremos combinar varios elementos de tipo vista, tendremos que utilizar un objeto de tipo *layout*. Un *layout* es un contenedor de una o más vistas y controla su comportamiento y posición. Hay que destacar que un *layout* puede contener otro *layout* y que es un descendiente de la clase `View`.

La siguiente lista describe los *layout* más utilizados en Android:

LinearLayout: Dispone los elementos en una fila o en una columna.

TableLayout: Distribuye los elementos de forma tabular.

RelativeLayout: Dispone los elementos con relación a otro o al padre.

ConstraintLayout: Versión mejorada de `RelativeLayout`, que permite una edición visual desde el editor **y trabajar con porcentajes**.

FrameLayout: Permite el cambio dinámico de los elementos que contiene.

AbsoluteLayout: Posiciona los elementos de forma absoluta.

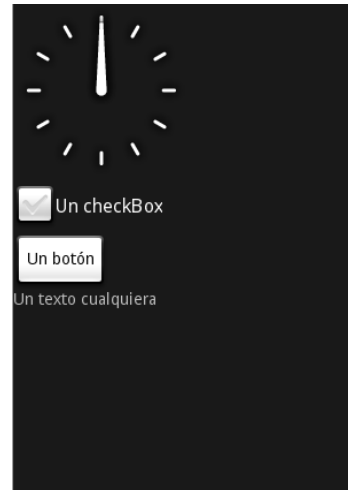
Dado que un ejemplo vale más que mil palabras, pasemos a mostrar cada uno de estos *layouts* en acción:

LinearLayout es uno de los *layout* más utilizado en la práctica. Distribuye los elementos uno a continuación de otro, bien de forma horizontal o vertical.

```

<LinearLayout xmlns:android="http://...
    android:layout_height="match_parent"
    android:layout_width="match_parent"
    android:orientation="vertical">
    <AnalogClock
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
    <CheckBox
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Un checkBox"/>
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Un botón"/>
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Un texto cualquiera"/>
</LinearLayout>

```



TableLayout distribuye los elementos de forma tabular. Se utiliza la etiqueta `<TableRow>` cada vez que queremos insertar una nueva línea.

```

<TableLayout xmlns:android="http://...
    android:layout_height="match_parent"
    android:layout_width="match_parent">
    <TableRow>
        <AnalogClock
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"/>
        <CheckBox
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Un checkBox"/>
    </TableRow>
    <TableRow>
        <Button
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Un botón"/>
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Un texto cualquiera"/>
    </TableRow>
</TableLayout>

```

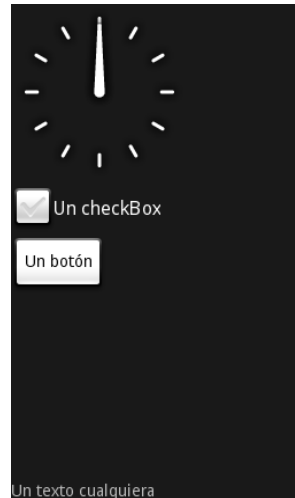


RelativeLayout permite comenzar a situar los elementos en cualquiera de los cuatro lados del contenedor e ir añadiendo nuevos elementos pegados a estos.

```

<RelativeLayout
    xmlns:android="http://schemas...
    android:layout_height="match_parent"
    android:layout_width="match_parent">
    <AnalogClock
        android:id="@+id/AnalogClock01"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentTop="true"/>
    <CheckBox
        android:id="@+id/CheckBox01"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@id/AnalogClock01"
        android:text="Un checkBox"/>
    <Button
        android:id="@+id/Button01"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Un botón"
        android:layout_below="@id/CheckBox01"/>
    <TextView
        android:id="@+id/TextView01"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentBottom="true"
        android:text="Un texto cualquiera"/>
</RelativeLayout>

```

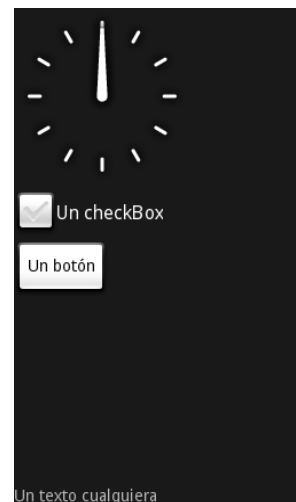


ConstraintLayout versión más flexible y eficiente de RelativeLayout.

```

<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas...
    android:layout_height="match_parent"
    android:layout_width="match_parent">
    <AnalogClock
        android:id="@+id/AnalogClock01"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintTop_toTopOf="parent"/>
    <CheckBox
        android:id="@+id/CheckBox01"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Un checkBox"
        app:layout_constraintTop_toBottomOf=
            "@+id/AnalogClock01"
        app:layout_constraintTop_toTopOf="parent"/>
    <Button
        android:id="@+id/Button01"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Un botón"
        app:layout_constraintTop_toBottomOf=
            "@+id/CheckBox01"
        app:layout_constraintLeft_toLeftOf=
            "@+id/CheckBox01"/>
    <TextView
        android:id="@+id/TextView01"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Un texto cualquiera"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"/>
</androidx.constraintlayout.widget.ConstraintLayout>

```



FrameLayout posiciona las vistas usando todo el contenedor, sin distribuir las espacialmente. Este *layout* suele utilizarse cuando queremos que varias vistas ocupen un mismo lugar. Podemos hacer que solo una sea visible, o superponerlas. Para modificar la visibilidad de un elemento utilizaremos la propiedad *visibility*.

```

<FrameLayout xmlns:android="http://schemas...
    android:layout_height="match_parent"
    android:layout_width="match_parent">
    <AnalogClock
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
    <CheckBox
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Un checkBox"/>
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Un botón"
        android:visibility="invisible"/>
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Un texto cualquiera"
        android:visibility="invisible"/>
</FrameLayout>

```



RelativeLayout permite indicar las coordenadas (x, y) donde queremos que se visualice cada elemento. No es recomendable utilizar este tipo de *layout*. La aplicación que estamos diseñando tiene que visualizarse correctamente en dispositivos con cualquier tamaño de pantalla. Para conseguirlo, no es una buena idea trabajar con coordenadas absolutas. De hecho, este tipo de *layout* ha sido marcado como obsoleto.

```

<RelativeLayout xmlns:android="http://schemas.
    android:layout_height="match_parent"
    android:layout_width="match_parent">
    <AnalogClock
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_x="50px"
        android:layout_y="50px"/>
    <CheckBox
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Un checkBox"
        android:layout_x="150px"
        android:layout_y="50px"/>
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Un botón"
        android:layout_x="50px"
        android:layout_y="250px"/>
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Un texto cualquiera"
        android:layout_x="150px"
        android:layout_y="200px"/>
</RelativeLayout>

```



Si tienes dudas sobre cuando usar cada layout usa la siguiente tabla:

LinearLayout: Diseños muy sencillos.

RelativeLayout: Nunca, hay una nueva alternativa

ConstraintLayout: Usar por defecto.

FrameLayout: Varias vistas superpuestas.

AbsoluteLayout: Nunca. Aunque está bien conocerlo por si acaso.



Vídeo[tutorial]: *Los layouts en Android*



Preguntas de repaso: *Tipos de layouts*



Preguntas de repaso: *Atributos de los layouts*

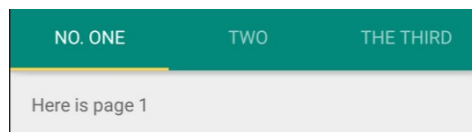
También podemos utilizar otros *layouts*, que se describen a continuación:

ScrollView: Visualiza una vista en su interior; cuando esta no cabe en pantalla, se permite un deslizamiento vertical.

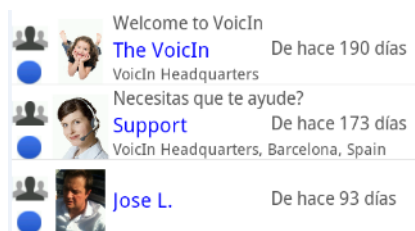
HorizontalScrollView: Visualiza una vista en su interior; cuando esta no cabe en pantalla, se permite un deslizamiento horizontal.



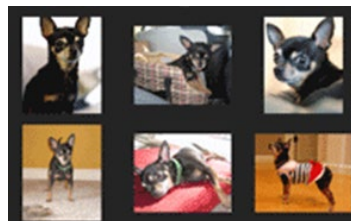
TabLayout, FragmentTabHost, TabLayout o TabHost: Proporciona una lista de pestañas que pueden ser pulsadas por el usuario para seleccionar el contenido a visualizar. Se estudia al final del capítulo.



ListView: Visualiza una lista deslizable verticalmente de varios elementos. Su utilización es algo compleja. Se verá un ejemplo en el capítulo siguiente.



GridView: Visualiza una cuadrícula deslizable de varias filas y varias columnas.



RecyclerView: Versión actualizada que realiza las mismas funciones que **ListView** o **GridView**. Se verá en el siguiente capítulo.

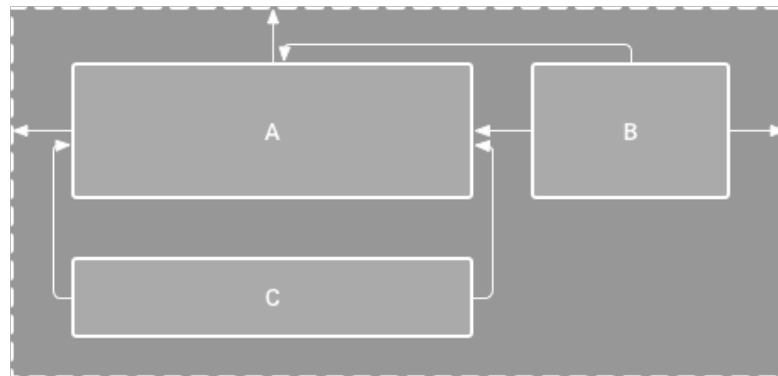
ViewPager: Permite visualizar una serie de páginas, donde el usuario puede navegar arrastrando a derecha o izquierda. Cada página ha de ser almacenada en un fragment.

2.3.1. Uso de ConstraintLayout

Este nuevo layout ha sido añadido en una librería de compatibilidad, por lo que se nos anima a usarlo de forma predeterminada. Nos permite crear complejos diseños sin la necesidad de usar *layouts* anidados. El hecho de realizar diseños donde un *layout* se introduce dentro de otro y así repetidas veces, ocasionaba problemas de memoria y eficiencia en dispositivos de pocas prestaciones.

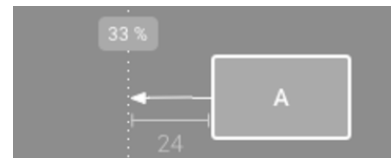
Es muy parecido a *RelativeLayout*, pero más flexible y fácil de usar desde el editor visual de Android Studio (disponible desde la versión 2.3). De hecho, se recomienda crear tu *layout* con las herramientas *drag-and-drop*, en lugar de editar el fichero XML. El resto de *layouts* son más fáciles de crear desde XML.

Las posiciones de las diferentes vistas dentro de este *layout* se definen usando *constraint* (en castellano restricciones). Un *constraint* puede definirse en relación al contenedor (*parent*), a otra vista o respecto a una línea de guía (*guideline*). Es necesario definir para cada vista al menos un *constraint* horizontal y uno vertical. No obstante, también podemos definir más de un *constraint* en el mismo eje. Veamos un ejemplo:




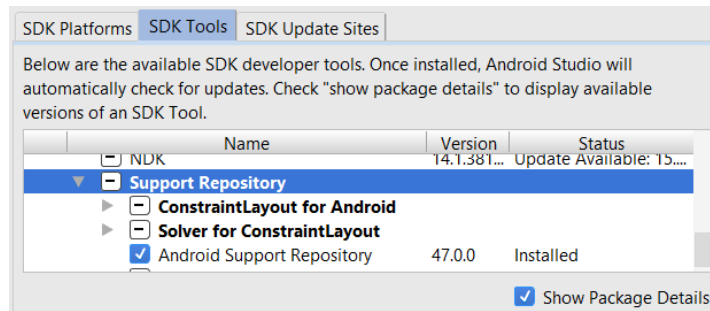
Observa cómo la vista A está posicionada con respecto al contenedor. La vista B está posicionada verticalmente con respecto a la vista A, aunque se ha definido un segundo *constraint* con respecto al lado derecho del contenedor. Veremos más adelante cómo se maneja esta circunstancia.

También podemos posicionar las vistas usando alineamientos. Observa cómo el borde superior de B está alineado con el borde superior de A. La vista C define dos alineamientos horizontales simultáneos, pero **ninguno** vertical, lo que ocasionará un error de compilación. También podemos realizar alineamientos usando la línea base de texto y líneas de guía, como se muestra a continuación:



Ejercicio: Creación de un layout con ConstraintLayout

1. Abre el proyecto PrimerasVistas o crea uno nuevo.
2. Abre *SDK Manager* (pulsas  en la barra de herramientas). Asegúrate que tienes instalados, en la pestaña *SDK Tools*, en *Support Repository*, los componentes *ConstraintLayout for Android* y *Solver for ConstraintLayout*.



3. En Gradle *Scripts/Bulid.gradle (Module:app)* ha de estar la dependencia:

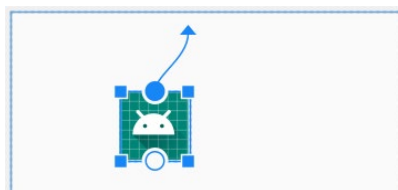
```
dependencies {
    ...
    implementation 'androidx.constraintlayout:constraintlayout:1.1.2'
}
```

- Abre el *layout activity_main.xml* creado en el ejercicio anterior. Pulsa con el botón derecho en *Component Tree* y selecciona la opción *Convert LinearLayout to ConstraintLayout*. Esta herramienta nos permite convertir nuestros viejos diseños que se basaban en *LinearLayout* o *RelativeLayout* en este nuevo tipo de *layouts*. Por desgracia no siempre funciona todo lo bien que desearíamos.
- Crea un nuevo *layout*. Para ello, pulsa con el botón derecho sobre *app/res/layout* y selecciona *New/Layout resource file*. Como nombre introduce "constraint" y en *Root element*: *androidx.constraintlayout.widget.ConstraintLayout*.
- Vamos a desactivar la opción de *Autoconnect* de la barra de acciones del *ConstraintLayout*. Es el segundo icono con forma de imán:

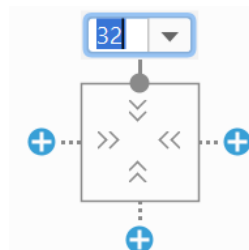


Tener activa esta opción es útil para diseñar más rápido los *layouts*. No obstante, a la hora de aprender a usar los *constraint*, es mejor ir haciéndolos de uno en uno.

- Dentro del área *Palette*, selecciona *Common* y arrastra una vista de tipo *ImageView* al área de diseño. Se abrirá una ventana con diferentes recursos *Drawable*. Selecciona en *Project*, *ic_launcher*.
- Para definir el primer *constraint*, pulsa sobre el punto de anclaje que aparece en la parte superior del *ImageView* y arrástralo hasta el borde superior del contenedor:

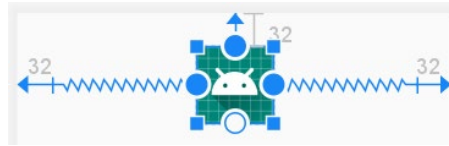


- En la parte superior derecha nos aparece un editor visual para los *constraint*. Por defecto la distancia seleccionada ha sido 8dp. Pulsa sobre este número y cámbialo a 32dp:



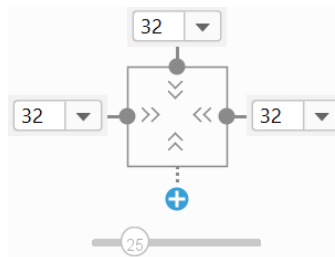
NOTA: Según las recomendaciones de *Material Design*, los márgenes y tamaños han de ser un múltiplo de 8dp.

10. Realiza la misma operación con el punto de anclaje izquierdo, arrastrándolo al borde izquierdo. Ya tenemos la restricción horizontal y vertical, por lo que la vista está perfectamente ubicada en el *layout*.
11. Arrastra el punto de anclaje derecho al borde derecho, introduciendo una distancia de 32dp:



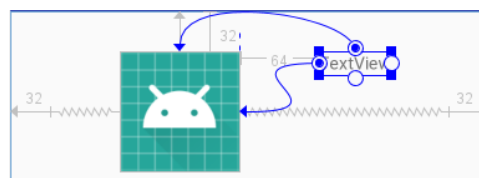
Observa como en este caso, al tener que cumplir simultáneamente dos *constraint* horizontales la imagen se centra horizontalmente. Esto se representa con la línea en zigzag, representando un muelle, que estira de la vista desde los dos lados. El pequeño botón con una cruz roja que aparece nos permite borrar todos los *constraint* de la vista.

12. Si en lugar de querer la imagen centrada la queremos en otra posición, podemos ir al editor de *constraint* y usar la barra deslizante (*Horizontal Bias*). Desplázala a la posición 25.

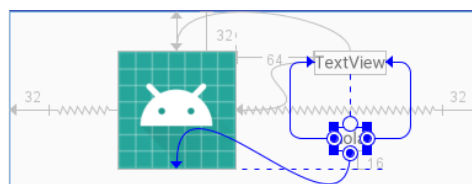


Observa en el área de trabajo como la longitud del muelle de la izquierda es un 25 %, frente al 75 % del muelle de la derecha.

13. Seleccionando el *ImageView* en el área de trabajo, arrastra el cuadrado azul de la esquina inferior derecha hasta aumentar su tamaño a 96x96 dp (múltiplos de 8). Otra alternativa es modificar los valores *layout_width* y *layout_height* en el área *Properties*.
14. Desde el marco *Palette / Common*, añade un *TextView* a la derecha del *ImageView*. Arrastra el punto de anclaje de la izquierda hasta el punto de la derecha de la imagen y establece un margen de 64 dp. Arrastra el punto de anclaje superior del *TextView* hasta el punto superior de la imagen:

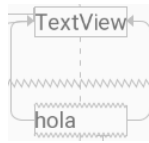


15. Añade un nuevo *TextView* bajo el anterior, con texto "hola". Introduce tres *constraint*, usando los puntos de anclaje inferior, izquierdo y derecho, tal y como se muestra en la siguiente figura:



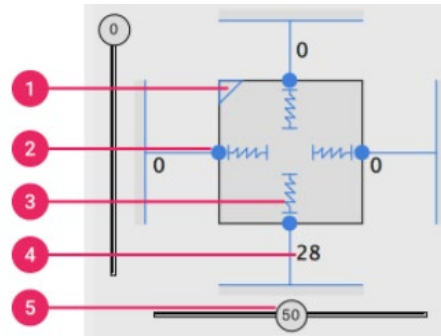
El margen inferior ha de ser 16dp y el izquierdo y derecho 0. De esta forma hemos centrado horizontalmente los dos *TextView*.




16. También podemos conseguir que el ancho del nuevo *TextView* coincida con el superior. Para ello selecciona la vista y en el campo *layout_width* e introduce *match_constraint* o *0dp*. Con esto, hacemos que el ancho se calcule según las restricciones de los *constraint*. Quita los márgenes laterales para que los anchos de las dos vistas coincidan.



17. Haz que desde `MainActivity` se visualice este *layout* y ejecuta el proyecto en un dispositivo.


Una vez familiarizados con los conceptos básicos de los *constraint* vamos a ver con más detalle las herramientas disponibles. Veamos en editor de *constraint*:




- (1) **relación de tamaño:** Puede establecer el tamaño de la vista en una proporción, por ejemplo a 16:9, si al menos una de las dimensiones de la vista está configurada como "ajustar a *constraint*" (0dp). Para activar la relación de tamaño, haz clic donde señala el número 1.
- (2) **eliminar *constraint*:** Se elimina la restricción para este punto de anclaje.
- (3) **establecer alto/ancho:** Para cambiar la forma en la que se calcula las dimensiones de la vista, pulsa en este elemento. Existen tres posibilidades:
 -  **ajustar a contenido:** equivale al valor `warp_content`. (Ej. 1^{er} `TextView`)
 -  **ajustar a *constraint*:** equivale a poner 0dp. (Ej. 2^o `TextView`)
 -  **tamaño fijo:** equivale a poner un valor concreto de dp. (Ej. `ImageView`)
 Aunque se representan 4 segmentos, realmente podemos cambiar 2, los horizontales para el ancho y los verticales para el alto.
- (4) **establecer margen:** Podemos cambiar los márgenes de la vista.
- (5) **sesgo del *constraint*:** Ajustamos cómo se reparte la dimensión sobrante.

También es importante repasar las acciones disponibles cuando trabajamos con `ConstraintLayout`:





 **Ocultar *constraint*:** Elimina las marcas que muestran las restricciones y los márgenes existentes.

 **Autoconectar:** Al añadir una nueva vista se establecen unos *constraint* con elementos cercanos de forma automática.


 **Definir márgenes:** Se configura los márgenes por defecto.

 **Borrar todos los *constraint*:** Se eliminan todas las restricciones del *layout*.

 **Crear automáticamente *constraint*:** Dada una vista seleccionada, se establecen unos *constraint* con elementos cercanos de forma automática.


 **Empaquetar / expandir:** Se agrupan o se separan los elementos.

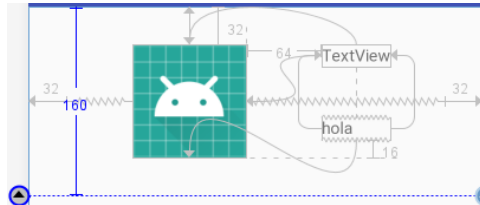
 **Alinear:** Centra o justifica los elementos seleccionados.

 **Añadir línea de guía:** Se crea una nueva línea de referencia.

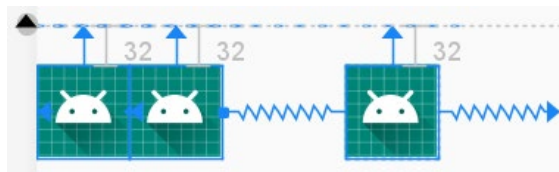



Ejercicio: Líneas guía y cadenas en ConstraintLayout

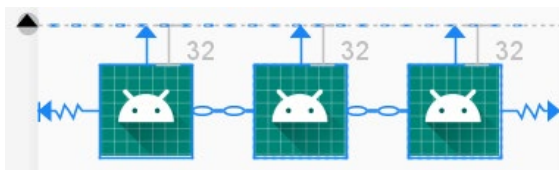
1. Siguiendo con el *layout* del ejercicio anterior. Pulsa en la acción *Guideline*  y selecciona *Add Horizontal GuideLine*. Aparecerá un círculo gris con un pequeño triángulo pegado al borde izquierdo, desde donde sale una línea guía horizontal. Arrástrala hacia abajo hasta separarla una distancia de 160dp.



2. Esta guía nos permite dividir el *layout* en dos áreas: la superior, donde ya hemos realizado una especie de cabecera, y la inferior. A partir de ahora los elementos de la parte inferior los colocaremos en relación a esta línea guía.
3. Selecciona el *ImageView*, cópialo (Ctrl+C) y pégalo tres veces (Ctrl+V). Acabamos de hacer tres copias de la imagen, pero no son visibles al estar en la misma posición. En el área *Component Tree*, selecciona *imageView2* y arrástralo bajo la línea guía, pegado a la izquierda. Coloca *imageView3* bajo la línea guía, en el centro, e *imageView4*, a la derecha de este.
4. Selecciona *imageView2* con el botón derecho y borra todos sus *constraint* seleccionando *Clear Constraint of Selección*. Repite esta operación para *imageView3* e *imageView4*.
5. Las tres imágenes han de tener un *constraint* desde el punto de anclaje superior a la línea guía con un margen de 32dp. Desde el punto de anclaje izquierdo de *imageView2*, establece un *constraint* con el borde izquierdo y en las otras dos imágenes, al punto de anclaje derecho de la vista de la derecha. El punto de anclaje derecho de la tercera vista únelo al borde derecho. El resultado ha de ser el siguiente:




6. Para conseguir que estas tres vistas formen una cadena, selecciónalas y utiliza la acción *Align*  / *Horizontaly* o el botón derecho *Chains* / *Create Horizontal Chain*:



Observa cómo las vistas ahora están unidas por medio de un conector de cadena y aparece un cono con dos eslabones en la parte inferior.

Si abres la lengüeta *Text* para estudiar el XML, puedes comprobar que para establecer la cadena se han añadido dos *constraint*, desde el punto de anclaje derecho de las dos primeras vista hacia la vista de su izquierda. Es decir, una restricción mutua: de A -> B y de B -> A.

NOTA: En la versión actual del editor visual, parece que establecer la cadena indicando estos *constraint* individualmente no parece posible. Hay que usar la acción *Align*  / *Horizontaly* o *Chains* / *Create Horizontal Chain*.

7. También es posible otras distribuciones de cadena, pulsando sobre cualquier botón con dos eslabones (se muestra en la figura anterior). Podemos hacer que los márgenes se distribuyan solo entre las vistas o solo en los extremos izquierdo y derecho:





Si abres la lengüeta **Text** **para estudiar el XML** puedes comprobar que estas dos nuevas configuraciones de cadena se consiguen añadiendo en la primera vista el atributo:

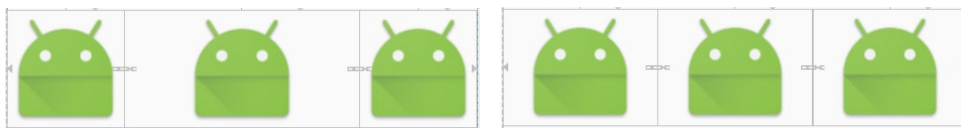
```
app:layout_constraintHorizontal_chainStyle="spread_inside"
```

Para la primera distribución y para la segunda:

```
app:layout_constraintHorizontal_chainStyle="packed"
```

Para la distribución del punto anterior el valor es **"spread"** o no indicar nada.

- Existe otra distribución en la que los márgenes desaparecen y se ajusta el ancho de las vistas hasta cubrir todo el espacio disponible. Selecciona la vista central y en el editor de *constraint* pulsa sobre el icono  hasta que aparezca . Recuerda que esta acción es equivalente a poner 0 en **layout_width**. El resultado se muestra a la izquierda:



Para conseguir el resultado de la derecha, hemos repetido la operación con las otras dos imágenes.

Si en lugar de repartir los anchos por igual, quieres otra configuración, puedes usar el atributo **layout_constraintHorizontal_weight** (funciona igual que **layout_weight** en un **LinearLayout**).

- Ejecuta el proyecto en un dispositivo.

Al crear *constraint*, recuerde las siguientes reglas:

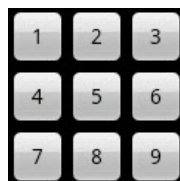
- Cada vista debe tener al menos dos restricciones: una horizontal y otra vertical.
- Solo puede crear restricciones que compartan el mismo plano. Así, el plano vertical (los lados izquierdo y derecho) de una vista puede anclarse solo a otro plano vertical.
- Cada manejador de restricción puede utilizarse para una sola restricción, pero puede crear varias restricciones (desde vistas diferentes) al mismo punto de anclaje.



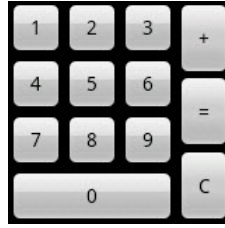
Práctica: Uso de layouts

- Utiliza un **ConstraintLayout** para realizar un diseño similar al siguiente:

- Utiliza un **TableLayout** para realizar un diseño similar al siguiente:



- Utiliza un **LinearLayout** horizontal que contenga en su interior otros **LinearLayout** para realizar un diseño similar al siguiente. Ha de ocupar toda la pantalla:



4. Visualiza el resultado obtenido en diferentes tamaños de pantalla. ¿Se visualiza correctamente?
5. Realiza el ejercicio de la calculadora usando un `ConstraintLayout`.



Preguntas de repaso: `ConstraintLayout`

2.4. Una aplicación de ejemplo: Asteroides

A lo largo de este libro vamos a ir creando una aplicación de ejemplo que toque los aspectos más significativos de Android. Comenzamos en este capítulo creando una serie de vistas que nos permitirán diseñar una sencilla interfaz de usuario. Si quieres ver cómo quedará la aplicación una vez termines el libro, puedes ver el siguiente vídeo:



Vídeo[tutorial]: Asteroides



Enlaces de interés:

- *Asteroides*: Puedes descargarla la aplicación de Google Play:
<https://play.google.com/store/apps/details?id=es.upv.mmoviles.asteroides&hl>



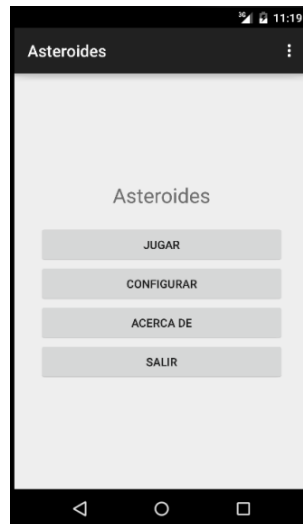
Práctica: Creación de la aplicación Asteroides

1. Crea un nuevo proyecto con los siguientes datos:

```
Phone and Tablet / Empty Activity
Name: Asteroides
Package name: org.example.asteroides
Language: Java
Minimum API level: API 19 Android 4.4 (KitKat)
☒ Use AndroidX.* artifacts
```

NOTA: Deja el resto de los parámetros con su valor por defecto.

2. Abre el fichero `res/layout/activity_main.xml` y trata de crear una vista similar a la que ves a continuación. Ha de estar formada por un `LinearLayout` que contiene un `TextView` y cuatro `Button`. Trata de utilizar recursos para introducir los cinco textos que aparecen.

**Solución:**

1. El fichero *activity_main.xml* ha de ser similar al siguiente:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:gravity="center"
    android:padding="30dp"
    tools:context=".MainActivity" >
    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/tituloAplicacion"
        android:gravity="center"
        android:textSize="25sp"
        android:layout_marginBottom="20dp"/>
    <Button android:id="@+id/button01"
        android:layout_height="wrap_content"
        android:layout_width="match_parent"
        android:text="@string/Arrancar"/>
    <Button android:id="@+id/button02"
        android:layout_height="wrap_content"
        android:layout_width="match_parent"
        android:text="@string/Configurar"/>
    <Button android:id="@+id/button03"
        android:layout_height="wrap_content"
        android:layout_width="match_parent"
        android:text="@string/Acercade"/>

    <Button android:id="@+id/button04"
        android:layout_height="wrap_content"
        android:layout_width="match_parent"
        android:text="@string/Salir"/>
</LinearLayout>
```

2. El fichero *res/values/strings.xml* ha de tener el siguiente contenido:

```
<resources>
    <string name="Arrancar">Jugar</string>
    <string name="Configurar">Configurar</string>
    <string name="Acercade">Acerca de </string>
    <string name="Salir">Salir</string>
```



```
<string name="tituloAplicacion">Asteroides</string>
<string name="app_name">Asteroides</string>
<string name="action_settings">Configuración</string>
</resources>
```



Práctica: Uso de `ConstraintLayout` en Asteroides

Repite la práctica anterior pero esta vez usando un `ConstraintLayout`:

2.5. La aplicación Mis Lugares

En este libro vamos a crear una segunda aplicación con características muy diferentes de Asteroides. Tendrá por nombre Mis Lugares y permitirá que los usuarios guarden información relevante sobre los sitios que suelen visitar (restaurantes, tiendas, etc.). En el capítulo 1 se implementaron algunas clases de esta aplicación y se presentó un vídeo que describía su funcionamiento. A diferencia de Asteroides, esta aplicación utilizará un diseño y varios elementos de Material Design. Una explicación más extensa de estos conceptos se abordará en *El Gran Libro de Android Avanzado*. Para más información sobre Material Design y el siguiente ejercicio te recomendamos un tutorial ¹.



Vídeo[tutorial]: Mis Lugares



Ejercicio: Creación de la aplicación Mis Lugares

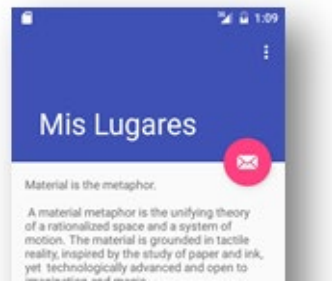
1. Crea un nuevo proyecto con los siguientes datos:

```
Phone and Tablet / Scrolling Activity
Name: Mis Lugares
Package name: com.example.mislugares
Language: Java ó Kotlin
Minimum API level: API 19 Android 4.4 (KitKat)
☒ Use AndroidX.* artifact
```

De esta forma, se creará una aplicación con una actividad basada en Material Design. Dispondrá de una barra de acciones y un botón flotante. El contenido principal podrá desplazarse, a la vez que la barra de acciones cambia de tamaño.

2. En el navegador de proyecto pulsa con el botón derecho sobre la clase `ScrollingActivity` y selecciona `Refactor > Rename`. Introduce como nuevo nombre `MainActivity`. En la carpeta `res/layout` renombra el fichero `activity_scrolling.xml` por `activity_main.xml`. En la misma carpeta renombra `content_scrolling.xml` por `content_main.xml`. Finalmente, en la carpeta `res/menu` renombra `menu_scrolling.xml` por `menu_main.xml`.
3. Ejecuta el proyecto y verifica su comportamiento:

¹ <http://www.androidcurso.com/index.php/688>



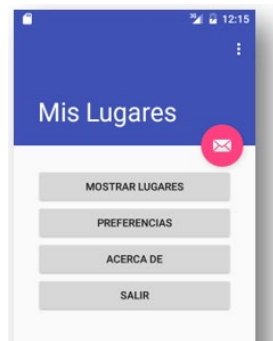
NOTA: Además del tutorial indicado puedes hacer el siguiente ².



Práctica: Creación de una primera actividad en Mis Lugares

En esta práctica crearemos una primera actividad que contendrá simplemente cuatro botones.

1. Edita el fichero `res / layout / content_main.xml` y trata de crear una vista similar a la que ves a continuación. Has de dejar el `NestedScrollView` que ya tenías y reemplazar el `TextView` por un `ConstraintLayout` o `LinearLayout` que contenga cuatro `Button`. Un `NestedScrollView` solo puede contener dentro un elemento, por lo que no puedes introducir directamente los cuatro botones. Usando un `layout` que los contenga se resuelve el problema.



Solución: <http://www.androidcurso.com/index.php/115> (con `LinearLayout`)

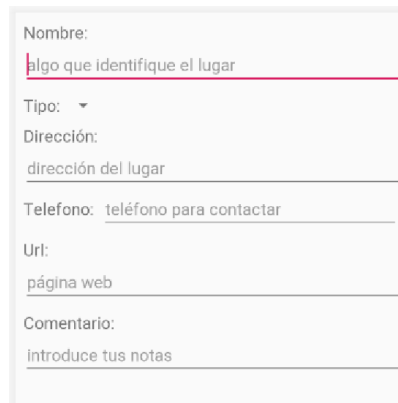


Práctica: Un formulario para introducir nuevos lugares

El objetivo de esta práctica es crear un `layout` que permita introducir y editar lugares en la aplicación Mis Lugares.

1. Crea un nuevo `layout` con nombre `edicion_lugar.xml`.
2. Ha de parecerse al siguiente formulario. Puedes basarte en un `LinearLayout` o un `ConstraintLayout` para distribuir los elementos. Pero es importante que este `layout`, se encuentre dentro de un `NestedScrollView` para que cuando el formulario no quepa en pantalla se pueda desplazar verticalmente.

² <http://www.androidcurso.com/index.php/689>



- Introduce a la derecha del `TextView` con texto "Tipo:" un `Spinner` con `id` `tipo`. Más adelante configuraremos esta vista para que muestre un desplegable con los tipos de lugares.
- Las vistas `EditText` han de definir el atributo `id` con los valores: `nombre`, `direccion`, `telefono`, `url` y `comentario`. Utiliza también el atributo `hint` para dar indicaciones sobre el valor a introducir. Utiliza el atributo `inputType` para indicar qué tipo de entrada esperamos. De esta manera se mostrará un teclado adecuado (por ejemplo, si introducimos un correo electrónico aparecerá la tecla @).

NOTA: El atributo `inputType` admite los siguientes valores (en negrita los que has de utilizar en este ejercicio): `none`, `text`, `textCapCharacters`, `textCapWords`, `textCapSentences`, `textAutoCorrect`, `textAutoComplete`, **`textMultiLine`**, `textImeMultiLine`, `textNoSuggestions`, **`textUri`**, `textEmailAddress`, `textEmailSubject`, `textShortMessage`, `textLongMessage`, `textPersonName`, **`textPostalAddress`**, `textPassword`, `textVisiblePassword`, `textWebEditText`, `textFilter`, `textPhonetic`, `textWebEmailAddress`, `textWebPassword`, `number`, `numberSigned`, `numberDecimal`, `numberPassword`, **`phone`**, `datetime`, `date` y `time`.

- Abre la clase `MainActivity` y en el método `onCreate()` reemplaza el layout:

```
setContentView(R.layout.activity_main.edicion Lugar)
```

- Comenta todas las líneas de este método que hay debajo usando `/* ... */`. Como ya no se crea el layout `activity_main` los id de vista a los que se accede ya no existen.
- Ejecuta la aplicación y verifica cómo cambia el tipo de teclado en cada `EditText`.
- Deshaz el cambio realizado en el punto 5 y 6.



Solución: <http://www.androidcurso.com/index.php/115>

2.6. Recursos alternativos

Una aplicación Android va a poder ser ejecutada en una gran variedad de dispositivos. El tamaño de pantalla, la resolución o el tipo de entradas puede variar mucho de un dispositivo a otro. Por otra parte, nuestra aplicación ha de estar preparada para diferentes modos de funcionamiento, como el modo "automóvil" o el modo "noche", y para poder ejecutarse en diferentes idiomas.


A la hora de crear la interfaz de usuario, hemos de tener en cuenta todas estas circunstancias. Afortunadamente, la plataforma Android nos proporciona una herramienta de gran potencia para resolver este problema: el uso de los recursos alternativos.

NOTA: Las prácticas de este apartado se proponen para la aplicación *Asteroides*, pero también pueden realizarse con *Mis Lugares*.



Práctica: Recursos alternativos en Asteroides

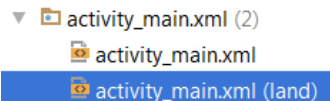
- Ejecuta la aplicación *Asteroides* (o *Mis Lugares*).

2. Los teléfonos móviles basados en Android permiten cambiar la configuración en apaisado y en vertical. Para conseguir este efecto con el emulador, pulsa el botón . Si usas un dispositivo de pantalla pequeña, observas como el resultado de la vista que acabas de diseñar en vertical no queda todo lo bien que desearíamos.

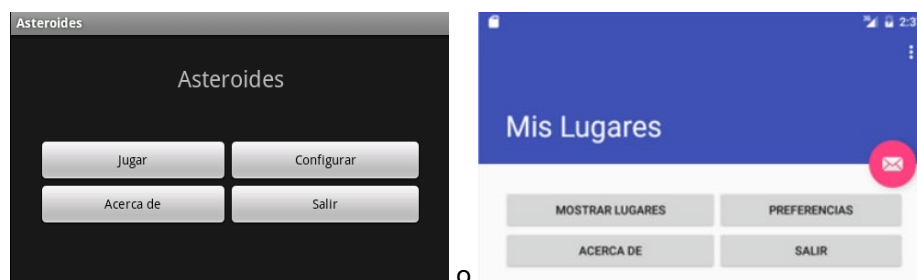


Para resolver este problema, Android te permite diseñar una vista diferente para la configuración horizontal y otra para la vertical.

3. Pulsa con el botón derecho sobre la carpeta *res/layout* y selecciona *New > Layout resource file*. Aparecerá una ventana donde has de rellenar en *File name*: *activity_main*, en *Available qualifiers*: selecciona *Orientation* y pulsa en el botón *>>*. En el desplegable *Screen orientation*: selecciona *Landscape*. Pulsa en *OK*. Observa como ahora hay dos recursos para el fichero *activity_main.xml*. El primero es el recurso por defecto, mientras que el segundo es el que se usará cuando el dispositivo esté en orientación *Landscape*.



4. Crea en el nuevo layout (*land*) una vista similar a la que ves a continuación: formada por un *TableLayout* con dos *Button* por columna.



5. Ejecuta de nuevo la aplicación y observa como la vista se ve correctamente en las dos orientaciones.



Solución:

Has de obtener un código XML similar al siguiente:

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:gravity="center"
    android:padding="30dp"
    tools:context=".MainActivity" >
    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/tituloAplicacion"
        android:gravity="center"
```

```

        android:textSize="25sp"
        android:layout_marginBottom="20dp"/>
    <TableLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:gravity="center"
        android:stretchColumns="*">
        <TableRow>
            <Button android:id="@+id/button01"
                android:layout_height="wrap_content"
                android:layout_width="match_parent"
                android:text="@string/Arrancar"/>
            <Button android:id="@+id/button02"
                android:layout_height="wrap_content"
                android:layout_width="match_parent"
                android:text="@string/Configurar"/>
        </TableRow>
        <TableRow>
            <Button android:id="@+id/button03"
                android:layout_height="wrap_content"
                android:layout_width="match_parent"
                android:text="@string/Acercade"/>
            <Button android:id="@+id/button04"
                android:layout_height="wrap_content"
                android:layout_width="match_parent"
                android:text="@string/Salir"/>
        </TableRow>
    </TableLayout>
</LinearLayout>

```

NOTA: Para conseguir que en un `TableLayout` las columnas se ajusten a todo el ancho de la tabla, poner `stretchColumns="*"`. `stretchColumns="0"` significa que se asigne la anchura sobrante a la primera columna. `stretchColumns="1"` significa que se asigne la anchura sobrante a la segunda columna. `stretchColumns="*"` significa que se asigne la anchura sobrante entre todas las columnas.

Android utiliza una lista de sufijos para expresar recursos alternativos. Estos sufijos pueden hacer referencia a la orientación del dispositivo, el lenguaje, la región, la densidad de píxeles, la resolución, el método de entrada, etc.

Por ejemplo, si queremos traducir nuestra aplicación al inglés, español y francés, siendo el primer idioma el usado por defecto, crearíamos tres versiones del fichero `strings.xml` y lo guardaríamos en los tres directorios siguientes:

```

res/values/strings.xml
res/values-es/strings.xml
res/values-fr/strings.xml

```

Aunque internamente el SDK de Android utiliza la estructura de carpetas anterior, en Android Studio el explorador del proyecto muestra los recursos alternativos de la siguiente manera:

```

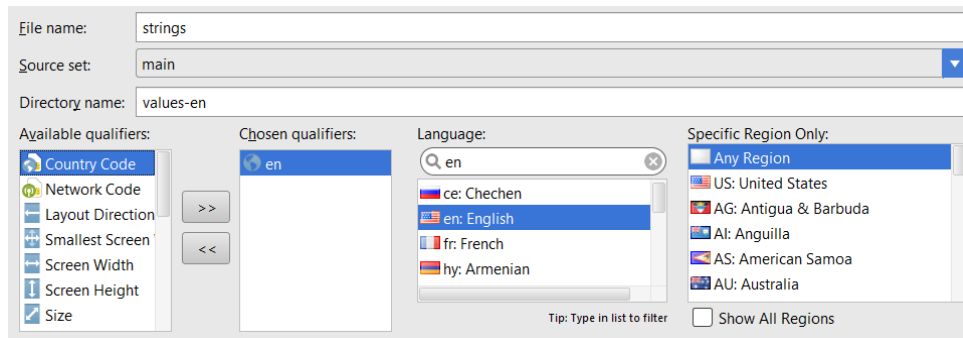
res
  values
    strings.xml (3)
    strings.xml
    strings.xml (es)
    strings.xml (fr)

```



Ejercicio: Traducción de Asteroides

1. Crea un nuevo recurso alternativo para `strings.xml (en)`: Pulsa con el botón derecho en `res/values`, selecciona `New > Values resource file` e introduce `strings` como nombre de fichero. Mueve el cualificador `Locale` (o `Language`) desde el marco de la izquierda a la derecha, pulsando el botón `>>`. En `Language` selecciona `English`, en `Specific Region Only` deja el valor `Any Region` y pulsa `OK`.



NOTA: Observa cómo además del idioma también permite seleccionar la región. De esta forma podremos diferenciar entre inglés americano, británico, australiano, ...

2. Copia el contenido del recurso por defecto, *strings.xml*, al recurso para inglés, *strings.xml (en)*. Traduce los textos al inglés. No has de traducir los nombres de los identificadores de recursos (*accion_mostrar*, *app_name*, ...) estos han de ser igual en todos los idiomas.
3. Ejecuta la aplicación.
4. Vamos a cambiar la configuración de idioma en un dispositivo Android. Para ello accede a *Ajustes* del dispositivo (*Settings*) y selecciona la opción *Personal > Idioma y entrada*. Dentro de esta opción selecciona como idioma *Español* o *Inglés*. **NOTA:** Observa que en otros idiomas permite seleccionar tanto el idioma como la región. Por desgracia, para el español solo permite dos regiones: *España* y *Estados Unidos*.
5. Observa como el texto aparece traducido al idioma seleccionado.

Otro ejemplo de utilización de recursos diferenciados lo podemos ver con el icono que se utiliza para lanzar la aplicación. Observa cómo, al crear una aplicación, este icono se crea en cinco carpetas *mipmap* diferentes, para utilizar un icono distinto según la densidad de píxeles del dispositivo:

```
res/mipmap-mdpi/ic_launcher.png
res/mipmap-hdpi/ic_launcher.png
res/mipmap-xhdpi/ic_launcher.png
res/mipmap-xxhdpi/ic_launcher.png
res/mipmap-xxxhdpi/ic_launcher.png
```

NOTA: En el siguiente capítulo se describe por qué se actúa de esta manera.

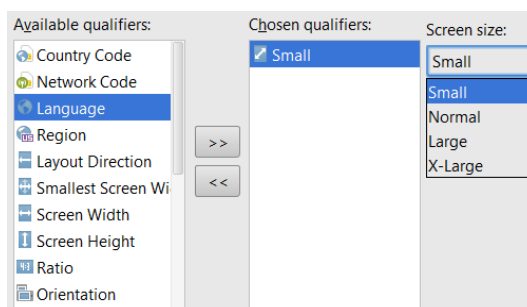
Resulta posible indicar varios sufijos concatenados; por ejemplo:

```
res/values-en-rUS/strings.xml
res/values-en-rUK/strings.xml
```

Pero cuidado, Android establece un orden a la hora de encadenar sufijos. Puedes encontrar una lista de estos sufijos en el apéndice C y en este enlace:

<http://developer.android.com/guide/topics/resources/providing-resources.html>

Para ver los sufijos disponibles, también puedes pulsar con el botón derecho sobre una carpeta de recursos y seleccionar *New > Android resource file*. Esta opción te permite crear un nuevo recurso y poner el sufijo deseado de forma y orden correctos.





Vídeo[tutorial]: *Uso de recursos alternativos en Android*



Ejercicio: *Creando un layout para tabletas en Asteroides*

Si ejecutas la aplicación Asteroides (o Mis Lugares) en una tableta, observarás que los botones son demasiado alargados (izquierda). Queremos que en este caso la apariencia sea similar a la mostrada a la derecha:



1. Crea un recurso alternativo a `res/values/dimens.xml`, que sea utilizado en pantallas de tamaño `xlarge` (7-10,5 pulgadas) en orientación `land` (apaisado). En este fichero define el siguiente valor:

```
<resources>
  <dimen name="margen_botones">150dp</dimen>
</resources>
```

2. Añade el mismo valor al recurso por defecto, pero esta vez con `30dp`.
3. Modifica los ficheros `res/layout/content_main.xml` y `content_main.xml (lan)`, reemplazando `android:padding="30dp"` por `android:padding="@dimen/margen_botones"`.
4. Verifica que la aplicación se visualiza correctamente en todos los tipos de pantalla, tanto en horizontal como en vertical.



Preguntas de repaso: *Recursos alternativos*



Enlaces de interés: *Recursos alternativos*

http://www.androidcurso.com/images/dcomg/ficheros/recursos_alternativos.pdf

2.7. Tipos de recursos y recursos del sistema

La definición de los recursos en Android es un aspecto muy importante en el diseño de una aplicación. Una de sus principales ventajas es que facilita a los diseñadores gráficos e introductores de contenido trabajar en paralelo con los programadores.

Añadir un recurso a nuestra aplicación es muy sencillo, no tenemos más que añadir un fichero dentro de una carpeta determinada de nuestro proyecto. Para cada uno de los recursos que añadamos el sistema crea, de forma automática, un `id` de recurso dentro de la clase `R`.

2.7.1. Tipos de recursos

Según la carpeta que utilicemos, el recurso creado será de un tipo específico. Pasamos a enumerar las carpetas y los tipos posibles:

Carpeta**identificador****Descripción**

res/drawable/ R.drawable	Ficheros en <i>bitmap</i> (.png, .jpg o .gif). Ficheros PNG en formato Nine-patch (.9.png). Ficheros XML con descriptores gráficos (véase clase Drawable).
res/mipmap/ R.mipmap	Ficheros en <i>bitmap</i> (.png, .jpg o .gif). Estos gráficos no son rescalados para adaptarlos a la densidad gráfica del dispositivo, si no que se buscará en las subcarpetas el gráfico con la densidad más parecida y se utilizará directamente.
res/layout/ R.layout	Ficheros XML con los <i>layouts</i> usados en la aplicación.
res/menu/ R.menu	Ficheros XML con la definición de menús, que podemos asignar a una actividad o a una vista.
res/anim/ R.anim	Ficheros XML que permiten definir animaciones Tween, también conocidas como animaciones de vista.
res/animator R.animator	Ficheros XML que permiten modificar las propiedades de un objeto a lo largo del tiempo (véase apartado “Animación de propiedades”). Solo desde la versión 3.0.
res/xml/ R.xml	Otros ficheros XML, como los ficheros de preferencias.
res/raw/ R.raw	Ficheros que se encuentran en formato binario. Por ejemplo, ficheros de audio o vídeo.
res/values/	Ficheros XML que definen un determinado valor para definir un color, un estilo, una cadena de caracteres, etc. Se describen en la siguiente tabla.

Tabla 1: Tipos de recursos según carpeta en Android.

Veamos los tipos de recursos que encontramos dentro de la carpeta *values*:

Fichero por defecto**identificador****Descripción**

strings.xml R.string	Identifica cadenas de caracteres. <pre><string name="saludo">¡Hola Mundo!</string></pre>
colors.xml R.color	Un color definido en formato ARGB (alfa, rojo, verde y azul). Los valores se indican en hexadecimal en uno de los formatos: #RGB, #ARGB, #RRGGBB ó #AARRGGBB. <pre><color name="verde_opaco">#0f0</color> <color name="red_translucido">#80ff0000</color></pre>
dimensions.xml R.dimen	Un número seguido de una unidad de medida. px – píxeles; mm – milímetros; in – pulgadas; pt – puntos (= 1/72 pulgadas); dp – píxeles independientes de la densidad (= 1/160 pulgadas); sp – igual que dp, pero cambia según las preferencias de tamaño de fuente. <pre><dimen name="alto">2.2mm</dimen> <dimen name="tamano_fuente">16sp</dimen></pre>
styles.xml R.style	Definen una serie de atributos que pueden ser aplicados a una vista o a una actividad. Si se aplican a una actividad, se conocen como temas. <pre><style name="TextoGrande" parent="@style/Text"> <item name="android:textSize">20pt</item> <item name="android:textColor">#000080</item></pre>

Fichero por defecto
identificador**Descripción**

	<code></style></code>
<code>R.int</code>	Define un valor entero. <code><integer name="max_asteroides">5</integer></code>
<code>R.bool</code>	Define un valor booleano. <code><bool name="misiles_ilimitados">true</bool></code>
<code>R.id</code>	Define un recurso de <i>id</i> único. La forma habitual de asignar un <i>id</i> a los recursos es con el atributo <code>id="@+id/nombre"</code> . Aunque en algunos casos puede ser interesante disponer de un <i>id</i> previamente creado, para que los elementos así nombrados tengan una determinada función. Este tipo de <i>id</i> se utiliza en las vistas <i>TabHost</i> y <i>ListView</i> . <code><item type="id" name="button_ok" /></code> <code><item type="id" name="dialog_exit" /></code>
<code>R.array</code>	Una serie ordenada de elementos. Pueden ser de <i>strings</i> , de enteros o de recursos (<i>TypedArray</i>). <code><string-array name="dias_semana"></code> <code><item>lunes</item></code> <code><item>martes</item></code> <code></string-array></code> <code><integer-array name="primos"></code> <code><item>2</item><item>3</item><item>5</item></code> <code></integer-array></code> <code><array name="asteroides"></code> <code><item>@drawable/asteroide1</item></code> <code><item>@drawable/asteroide2</item></code> <code></array></code>

Tabla 2: Tipos de recursos en carpeta *values*.

Aunque el sistema crea ficheros que aparecen en la columna de la izquierda de la tabla anterior y se recomienda definir los recursos de cadena dentro de *strings.xml*, hay que resaltar que no es más que una sugerencia de organización. Sería posible mezclar cualquier tipo de recurso de esta tabla dentro de un mismo fichero y poner a este fichero cualquier nombre.



Vídeo[tutorial]: Tipos de recursos en Android

2.7.2. Acceso a los recursos

Una vez definido un recurso, este puede ser utilizado desde un fichero XML o desde Java. A continuación se muestra un ejemplo desde XML:

```
<ImageView
    android:layout_height="@dimen/alto"
    android:layout_width="match_parent"
    android:background="@drawable/asteroide"
    android:text="@string/saludo"
    android:text_color="@color/verde_opaco"/>
```

Para acceder a un recurso definido en los ejemplos anteriores, usaremos el siguiente código:

```
Resources res = getResources();
Drawable drawable = ContextCompat.getDrawable(R.drawable.asteroide);
String saludo = res.getString(R.string.saludo);
int color = ContextCompat.getColor(R.color.verde_opaco);
float tamanoFuente = res.getDimension(R.dimen.tamano_fuente);
int maxAsteroides = res.getInteger(R.integer.max_asteroides);
```

```
boolean ilimitados = res.getBoolean(R.bool.misiles_ilimitados);
String[] diasSemana = res.getStringArray(R.array.dias_semana);
int[] primos = res.getIntArray(R.array.primos);
TypedArray asteroides = res.obtainTypedArray(R.array.asteroides);
Drawable asteroide1 = asteroides.getDrawable(0);
```


```
val drawable = ContextCompat.getDrawable( R.drawable.asteroide)
val saludo = resources.getString(R.string.saludo)
val color = ContextCompat.getColor(R.color.verde_opaco)
val tamanoFuente = resources.getDimension(R.dimen.tamano_fuente)
val maxAsteroides = resources.getInteger(R.integer.max_asteroides)
val ilimitados = resources.getBoolean(R.bool.misiles_ilimitados)
val diasSemana = resources.getStringArray(R.array.dias_semana)
val primos = resources.getIntArray(R.array.primos)
val asteroides = resources.obtainTypedArray(R.array.asteroides)
val asteroide1 = asteroides.getDrawable(0)
```

2.7.3. Recursos del sistema

Además de los recursos que podamos añadir a nuestra aplicación, también podemos utilizar una serie de recursos que han sido incluidos en el sistema.



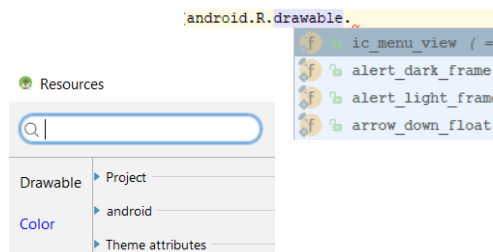
Vídeo[tutorial]: *Recursos del sistema en Android*

Usar recursos del sistema tiene muchas ventajas. No consumen memoria en nuestra aplicación, al estar ya incorporados al sistema. Además, los usuarios están familiarizados con ellos. Por ejemplo, si utilizamos el recurso `android.R.drawable.ic_menu_edit`, se mostrará al usuario el icono: . Muy posiblemente, el usuario ya está familiarizado con este icono y lo asocia a la acción de editar. Otra ventaja es que los recursos del sistema se adaptan a las diferentes versiones de Android. Si se utiliza el tema `android.R.style.Theme_Panel`, este es bastante diferente en cada una de las versiones, pero seguro que estará en consonancia con el resto de estilos para esta versión. Lo mismo ocurre con el icono anterior. Este icono es diferente en algunas versiones, pero al usar un recurso del sistema nos aseguramos de que se mostrará el adecuado a la versión del usuario. Finalmente, estos recursos se adaptan siempre a las configuraciones locales. Si yo utilizo el recurso `android.R.string.cancel`, este será “Cancelar”, “Cancel”, “取消”, etc., según el idioma escogido por el usuario.

Para acceder a los recursos del sistema desde código, usaremos la clase `android.R`. Se usa la misma estructura jerárquica de clases. Por ejemplo, `android.R.drawable.ic_menu_edit`. Para acceder desde XML, utilizamos la sintaxis habitual pero comenzando con `@android:`. Por ejemplo, `@android:drawable/ic_menu_edit`.

Para buscar recursos del sistema tienes varias alternativas:

- Usa la opción de autocompletar Studio.
- Emplea el buscador de recursos que en el editor de *layouts*.
- Usa la aplicación `android.R` para explorar los recursos del sistema.



de Android

se incluye

2.8. Estilos y temas

Si tienes experiencia con el diseño de páginas web, habrás advertido grandes similitudes entre HTML y el diseño de *layouts*. En los dos casos se utiliza un lenguaje de marcado y se trata de crear diseños independientes del tamaño de la pantalla donde se visualizarán. En el diseño web resultan clave las

hojas de estilo en cascada (CSS), que permiten crear un patrón de diseño y aplicarlo a varias páginas. Cuando diseñas los *layouts* de tu aplicación, vas a poder utilizar unas herramientas similares conocidas como estilos y temas. Te permitirán crear patrones de estilo que podrán ser utilizados en cualquier parte de la aplicación. Estas herramientas te ahorrarán mucho trabajo y te permitirán conseguir un diseño homogéneo en toda tu aplicación.



Vídeo[tutorial]: *Estilos y temas en Android*

2.8.1. Los estilos

Un estilo es una colección de propiedades que definen el formato y la apariencia que tendrá una vista. Podemos especificar cosas como tamaño, márgenes, color, fuentes, etc. Un estilo se define en ficheros XML, diferente del fichero XML Layout que lo utiliza.

Veamos un ejemplo. El siguiente código:

```
<TextView
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:textColor="#00FF00"
    android:typeface="monospace"
    android:text="Un texto" />
```

Es equivalente a escribir:

```
<TextView
    style="@style/MiEstilo"
    android:text="Un texto" />
```

Habiendo creado en el fichero *res/values/styles.xml* con el siguiente código:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <style name="MiEstilo"
        parent="@android:style/TextAppearance.Medium">
        <item name="android:layout_width">match_parent</item>
        <item name="android:layout_height">wrap_content</item>
        <item name="android:textColor">#00FF00</item>
        <item name="android:typeface">monospace</item>
    </style>
</resources>
```

Observa como un estilo puede heredar todas las propiedades de un padre (parámetro *parent*) y a partir de estas propiedades realizar modificaciones.

Heredar de un estilo propio

Si vas a heredar de un estilo definido por ti, no es necesario utilizar el atributo *parent*. Por el contrario, puedes utilizar el mismo nombre de un estilo ya creado y completar el nombre con un punto más un sufijo. Por ejemplo:

```
<style name="MiEstilo.grande">
    <item name="android:textSize">18pt</item>
</style>
```

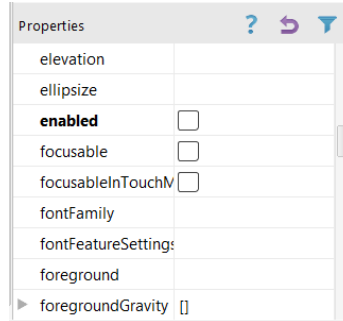
Crearía un nuevo estilo que sería igual a *MiEstilo* más la nueva propiedad indicada. A su vez, puedes definir otro estilo a partir de este:

```
<style name="MiEstilo.grande.negrita">
    <item name="android:textStyle">bold</item>
</style>
```



Práctica: *Creando un estilo*

1. Abre el proyecto Asteroides o Mis Lugares.
2. Crea un nuevo estilo y llámalo `EstiloBoton`. Para ver las propiedades que puedes modificar te recomendamos que consultes la "Referencia de la clase View" en el anexo D. Para un botón puedes definir los atributos de `View`, `TextView` y `Button`. Otra alternativa consiste en seleccionar un botón en el editor visual de vistas y en la ventana *Properties* buscar las propiedades disponibles:



3. Aplícalo al primer botón del *layout*.
4. Crea un nuevo estilo y llámalo `EstiloBoton.Alternativo`. Este ha de modificar alguno de los atributos anteriores y añadir otros, como `padding`.
5. Aplícalo al segundo botón del *layout*.
6. Visualiza el resultado.

2.8.2. Los temas

Un tema es un estilo aplicado a toda una actividad o aplicación, en lugar de a una vista individual. Cada elemento del estilo solo se aplicará a aquellos elementos donde sea posible. Por ejemplo, `CodeFont` solo afectará al texto.

Para aplicar un tema a toda una aplicación, edita el fichero *AndroidManifest.xml* y añade el parámetro `android:theme` en la etiqueta `<application>`:

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

También puedes aplicar un tema a una actividad en concreto:

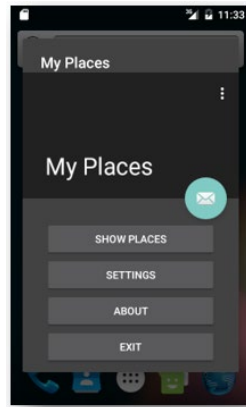
```
<activity android:theme="@style/MiTema">
```

Además de crear tus propios temas, vas a poder utilizar algunos disponibles en el sistema. Puedes encontrar una lista de todos los estilos y temas disponibles en Android en: <http://developer.android.com/reference/android/R.style.html>



Ejercicio: Aplicando un tema del sistema

1. Abre el proyecto Asteroides o Mis Lugares.
2. Aplica a la actividad principal el tema `@style/Theme.AppCompat.Dialog` tal y como se acaba de mostrar.
3. Visualiza el resultado. Este tema es utilizado en cuadros de diálogo. No parece muy adecuado para nuestra actividad.



4. Deshaz el cambio realizado en este ejercicio.



Práctica: Modificando el tema por defecto de la aplicación

1. Abre el proyecto Asteroides o Mis Lugares.
2. Abre el fichero `res/values/styles.xml` (recurso por defecto).
3. Observa cómo se define el estilo `AppTheme` que será usado como estilo por defecto en la aplicación. Hereda de `Theme.AppCompat.Light.DarkActionBar` y solo define los colores principales usados en la aplicación. Añade las líneas subrayadas para personalizar un par de aspectos:

```
<style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar">
    <!-- Customize your theme here. -->
    <item name="android:typeface">serif</item>
    <item name="android:textColor">#0000FF</item>
    <item name="colorPrimary">@color/colorPrimary</item>
    <item name="colorPrimaryDark">@color/colorPrimaryDark</item>
    <item name="colorAccent">@color/colorAccent</item>
</style>
```

4. Ejecuta la aplicación para visualizar el resultado.
5. Modifica otros atributos y comprueba el resultado.



Preguntas de repaso: Estilos y temas

2.9. Uso práctico de vistas y layouts

En este apartado vamos a aprender a usar varios tipos de vistas y *layouts* desde un punto de vista práctico. También empezaremos a escribir código que se ejecutará cuando ocurran ciertos eventos:



Ejercicio: Un botón con gráficos personalizados

1. Crea un nuevo proyecto con nombre *Mas Vistas* y tipo de actividad *Empty Activity*. Puedes dejar el resto de parámetros con los valores por defecto.
2. Crea el fichero `boton.xml` en la carpeta `res/drawable/`. Para ello puedes utilizar el menú *File > New > Drawable Resource File*. Introduce en el campo *File name*: «boton». Reemplaza el código por el siguiente:

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

```
<selector
  xmlns:android="http://schemas.android.com/apk/res/android">
  <item android:drawable="@drawable/boton_pulsado"
        android:state_pressed="true" />
  <item android:drawable="@drawable/boton_con_foco"
        android:state_focused="true" />
  <item android:drawable="@drawable/boton_normal" />
</selector>
```

Este XML define un recurso único gráfico (*drawable*) que cambiará en función del estado del botón. El primer ítem define la imagen usada cuando se pulsa el botón, el segundo ítem define la imagen usada cuando el botón tiene el foco (cuando el botón está seleccionado con la rueda de desplazamiento o las teclas de dirección) y el tercero, la imagen en estado normal. Los gráficos, y en concreto los *drawables*, se estudiarán en el capítulo 4.

NOTA: El orden de los elementos `<item>` es importante. Cuando se va a dibujar se recorren los ítems en orden hasta que se cumpla una condición. Debido a que "boton_normal" es el último, solo se aplica cuando las condiciones `state_pressed` y `state_focused` no se cumplen.


3. Descarga de <http://www.androidcurso.com/index.php/119> las tres imágenes que aparecen a continuación. Para bajar cada imagen, pulsa sobre los nombres. Guárdalos con el nombre de fichero que se indica a continuación:

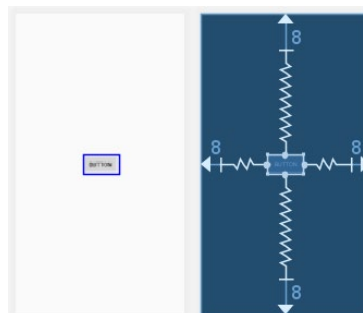


boton_normal.jpg

boton_con_foco.jpg

boton_pulsado.jpg

4. Selecciona los tres ficheros y cópialos en el portapapeles (*Ctrl-C*), selecciona la carpeta `res/drawable/` del proyecto y pega los ficheros (*Ctrl-V*). Te preguntará si quieres copiarlos a la carpeta de recursos por defecto a alguna de recursos alternativos. Selecciona la primera opción.
5. Abre el fichero `res/layout/activity_main.xml`.
6. En la ventana *Component Tree*, elimina el `TextView` que encontrarás dentro del `ConstraintLayout`.
7. Selecciona el `ConstraintLayout`. En la ventana *Attributes* busca el atributo `background`. Pulsa en el icono  y selecciona el recurso `Color/android/white`.
8. Arrastra una vista de tipo `Button` dentro del `ConstraintLayout`.
9. Sitúa el botón en el centro. Para ello, selecciona cada uno de sus puntos de anclaje arrastrando hasta el borde al que mira. El resultado ha de ser:



10. Selecciona el atributo `background` y pulsa el botón selector de recurso (con puntos suspensivos). Selecciona `Drawable/boton`.
11. Modifica el atributo `Text` para que no tenga ningún valor.

12. Introduce en el atributo `onClick` el valor `sePulsa`.

A continuación, se muestra el código resultante para `activity_main.xml`:

```
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity"
    android:background="@android:color/white">
    <Button android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/button"
        android:background="@drawable/boton"
        android:onClick="sePulsa"
        app:layout_constraintStart_toStartOf="parent"
        android:layout_marginStart="8dp"
        android:layout_marginTop="8dp"
        app:layout_constraintTop_toTopOf="parent"
        android:layout_marginBottom="8dp"
        app:layout_constraintBottom_toBottomOf="parent"
        android:layout_marginEnd="8dp"
        app:layout_constraintEnd_toEndOf="parent"/>
</androidx.constraintlayout.widget.ConstraintLayout>
```

13. Abre el fichero `MainActivity` e introduce antes de la última llave, el código:

```
public void sePulsa(View view){
    Toast.makeText(this, "Pulsado", Toast.LENGTH_SHORT).show();
}
```

```
fun sePulsa(view: View) {
    Toast.makeText(this, "Pulsado", Toast.LENGTH_SHORT).show()
}
```

NOTA: Pulsa **Alt-Intro** para que se añadan automáticamente los paquetes que faltan en la sección `import`.

El método anterior se ejecutará cuando se pulse el botón. A este tipo de métodos se los conoce como escuchadores de eventos (*listeners*). Este método se limita a lanzar un *toast*, es decir, un aviso que permanece cierto tiempo sobre la pantalla y luego desaparece. Los tres parámetros son: el contexto utilizado, que coincide con la actividad, el texto a mostrar y el tiempo que permanecerá este texto. Los conceptos de *actividad* y *contexto* se desarrollarán en el siguiente capítulo.

14. Ejecuta el proyecto y verifica el resultado.

2.9.1. Acceder y modificar propiedades de las vistas por código



Ejercicio: Acceder y modificar las propiedades de las vistas por código

1. Abre el `layout activity_main.xml` creado en el ejercicio anterior.
2. En la paleta de vistas, dentro de `Text`, busca *Number (Decimal)* y arrástralo arriba del botón rojo.
3. Modifica algunos atributos de esta vista: `hint` = "Introduce un número", `ID` = "entrada".
4. Desde el marco *Palette/Common*, arrastra *Button* arriba del botón rojo.
5. Modifica algunos atributos de esta vista: Haz que su anchura ocupe toda la pantalla, que su texto sea "0" y que su `id` sea "boton0".
6. Busca ahora `TextView` y arrástralo debajo del botón rojo.

7. Modifica algunos atributos de esta vista: `TextColor = #0000FF`, `Text = ""`, `Hint = "Resultado"`, `id = "salida"`.
8. Ajusta las restricciones de las vistas introducidas.
9. Abre el fichero `MainActivity`. **En Java**, vamos a añadir dos nuevas propiedades a la clase. Para ello copia el siguiente código al principio de la clase (antes del método `onCreate()`):

```
private EditText entrada;
private TextView salida;
```

NOTA: Recuerda pulsar **Alt-Intro** para que se añadan los paquetes de las dos nuevas clases utilizadas.

10. **En Java**, copia al final del método `onCreate()` las siguientes líneas:

```
entrada = findViewById(R.id.entrada);
salida = findViewById(R.id.salida);
```

Como se explicó al principio del capítulo, las diferentes vistas definidas en `activity_main.xml` son creadas como objetos Java cuando se ejecuta `setContentView(R.layout.activity_main)`. Si queremos manipular algunos de estos objetos hemos de declarar las variables (paso 9) y asignarles la referencia al objeto correspondiente (paso 10). Para ello, hay que introducir el atributo `id` en XML y utilizar el método `findViewById(R.id.valor_en_atributo_id)`. Este método devuelve un objeto de la clase `View`.

En Kotlin este proceso se realiza automáticamente. Lo único que tienes que hacer es asegurarte que se ha importado el paquete `kotlinx.android.synthetic.main.activity_main.*`.

11. Introduce en el atributo `onClick` del botón con `id boton0` el valor `"sePulsa0"`. De esta manera, cuando se pulse sobre el botón se ejecutará el método `sePulsa0()`. Según la jerga de Java, diremos que este método es un escuchador del evento `click` que puede generar el objeto `boton0`.
12. Añade el siguiente método al final de la clase `MainActivity`.

```
public void sePulsa0(View view){
    entrada.setText(entrada.getText()+"0");
}
```

```
fun sePulsa0(view: View) {
    entrada?.setText(entrada?.text.toString() + "0")
}
```

Lo que hace es asignar como textos de `entrada` el resultado de concatenar al texto de `entrada` el carácter `"0"`. **NOTA:** Si eres nuevo en Kotlin, te habrá extrañado el uso de `?`. Lee la sección Tratamiento de null en Kotlin ³.

13. Añade al botón con texto `"0"` el atributo `tag = "0"`.
14. Modifica el método `sePulsa0()` de la siguiente forma:

```
entrada.setText(entrada.getText()+(String)view.getTag());
```

```
entrada?.setText(entrada?.getText().toString() + view?.tag as String)
```

El resultado obtenido es equivalente al anterior. En algunos casos será interesante utilizar un mismo método como escuchador de eventos de varias vistas. Podrás averiguar la vista que causó el evento, dado que esta se pasa como parámetro del método. En el ejemplo sabemos que en el atributo `tag` guardamos el carácter a insertar. El atributo `tag` puede ser usado libremente por el programador para almacenar un objeto de la clase `Object` (en la práctica podemos usar cualquier tipo de clase, dado que `Object` es la clase raíz de la que heredan todas las clases en Java). En nuestro caso hemos almacenado un objeto `String`, por lo que necesitamos una conversión de tipo. **NOTA:** Utiliza esta forma de trabajar en la práctica para no tener que crear un método `onClick` para cada botón.

15. Modifica el código de `sePulsa0()` con el siguiente código:

³ <http://www.androidcurso.com/index.php/922>

```
salida.setText(String.valueOf(Float.parseFloat(
    entrada.getText().toString()) * 2.0));
```

```
salida?.setText((java.lang.Float.parseFloat(
    entrada?.getText().toString()) * 2.0).toString())
```

En este código el valor de `entrada` es convertido en `Float`, multiplicado por dos y convertido en `String` para ser asignado a `salida`.

16. Ejecuta el proyecto y verifica el resultado. *NOTA: En este ejercicio no se ha realizado la verificación de que los datos introducidos por el usuario. Has de tener introducir datos válidos y en el orden adecuado.*



Preguntas de repaso: Uso práctico de Vistas

2.10. Uso de *tabs* (pestañas)

Los *tabs* nos van a permitir crear una interfaz de usuario basada en pestañas, donde, de una forma muy intuitiva, podemos ofrecer al usuario diferentes contenidos, que son seleccionados al pulsar una de las pestañas que se muestran en la parte superior:



En este apartado usaremos la clase `FragmentTabHost` para crear pestañas, aunque existen otras alternativas:

- `TabHost`: Usado en las primeras versiones de Android. A partir del nivel de API 13, `TabHost` ha sido declarado obsoleto. Google reorientó su jerarquía de clases para introducir el concepto de fragment.
- `TabLayout`: Nueva alternativa propuesta en Material Design. Disponible en la librería de compatibilidad Design Support Library. Su uso se describe en *El Gran Libro de Android Avanzado*.

NOTA: Hasta la versión 3.0 (API 11) no aparece `FragmentTabHost`. Entonces, no podría usarse en niveles de API anteriores. Para resolver este problema, y más generalmente para poder usar fragments en versiones anteriores a la 3.0, Google ha creado la librería de compatibilidad⁴ (`android.support`). Se añade por defecto al crear un nuevo proyecto. Por lo tanto, podemos usar fragments y clases relacionadas desde cualquier versión.

Para crear en XML una interfaz de usuario basada en pestañas, puedes usar `FragmentTabHost` en el nodo raíz. Dentro un `LinearLayout`, que contendrá tanto el `TabWidget` para la visualización de las pestañas como un `FrameLayout` para mostrar el contenido. A continuación se muestra el esquema a utilizar:

```
<androidx.fragment.app.FragmentTabHost
    android:id="@android:id/tabhost" .../>
    <LinearLayout ...>
        <TabWidget    android:id="@android:id/tabs" .../>
        <FrameLayout  android:id="@android:id/tabcontent" .../>
    </LinearLayout>
</androidx.fragment.app.FragmentTabHost>
```

NOTA: El siguiente video utiliza `TabHost` en lugar `FragmentTabHost`. No obstante, muchos de los conceptos que se explican siguen siendo válidos.

⁴ <http://developer.android.com/tools/extras/support-library.html>



Vídeo[tutorial]: La vista TabHost en Android



Ejercicio: Uso de `FragmentTabHost` (Mejor saltar y hacer el siguiente ejercicio)

1. Crea un nuevo proyecto con nombre *Tabs* y tipo *Empty Activity*.
2. Reemplaza el código de `activity_main.xml` por el siguiente:

```
<androidx.fragment.app.FragmentTabHost
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@android:id/tabhost"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >
    <LinearLayout android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical" >
        <TabWidget android:id="@android:id/tabs"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:layout_weight="0"
            android:orientation="horizontal" />
        <FrameLayout android:id="@android:id/tabcontent"
            android:layout_width="match_parent"
            android:layout_height="0dp"
            android:layout_weight="1" />
    </LinearLayout>
</androidx.fragment.app.FragmentTabHost>
```

La primera cosa extraña de este código es el nombre de la primera etiqueta. En lugar de indicar simplemente `FragmentTabHost`, se ha indicado el nombre completo de dominio. Como estudiaremos en el capítulo 4, cada vez que usemos una vista que no sea del sistema (por ejemplo creada por nosotros o creada en una librería como en este caso) tendremos que indicar la clase donde se crea con su nombre completamente cualificado. Es decir, el nombre de la clase precedida de su paquete.

Como puedes observar un `FragmentTabHost` es el nodo raíz del diseño, que contiene dos elementos combinados por medio de un `LinearLayout`. El primero es un `TabWidget` para la visualización de las pestañas y el segundo es un `FrameLayout` para mostrar el contenido asociado de cada lengüeta. En número de lengüetas y su contenido se indicará por código.

3. Abre el fichero `MainActivity.java` y reemplaza el código por el siguiente:

```
import androidx.appcompat.app.AppCompatActivity;
import androidx.fragment.app.FragmentActivity;
import androidx.fragment.app.FragmentTabHost;

public class MainActivity extends FragmentActivity {
    private FragmentTabHost tabHost;
    @Override protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        tabHost = findViewById(android.R.id.tabhost);
        tabHost.setup(this,
            getSupportFragmentManager(), android.R.id.tabcontent);
        tabHost.addTab(tabHost.newTabSpec("tab1").setIndicator("Lengüeta 1"),
            Tab1.class, null);
        tabHost.addTab(tabHost.newTabSpec("tab2").setIndicator("Lengüeta 2"),
            Tab2.class, null);
        tabHost.addTab(tabHost.newTabSpec("tab3").setIndicator("Lengüeta 3"),
            Tab3.class, null);
    }
}
```

}

```

class MainActivity : FragmentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        if (tabhost!=null) {
            tabhost.setup(this, supportFragmentManager,
                android.R.id.tabcontent)
            tabhost.addTab(
                tabhost.newTabSpec("tab1").setIndicator("Lengüeta 1"),
                Tab1::class.java, null)
            tabhost.addTab(
                tabhost.newTabSpec("tab2").setIndicator("Lengüeta 2"),
                Tab2::class.java, null)
            tabhost.addTab(
                tabhost.newTabSpec("tab3").setIndicator("Lengüeta 3"),
                Tab3::class.java, null)
        }
    }
}

```

Observa como la clase creada extiende de `FragmentActivity` en lugar de `Activity`. Esto permitirá que la actividad trabaje con *fragments*; en concreto, vamos a crear un *fragment* para cada lengüeta. Se han añadido varias líneas en el método `onCreate()`. Empezamos inicializando la variable `tabHost`, luego se utiliza el método `setup()` para configurarla. Para ello indicamos el contexto, manejador de *fragments* y donde se mostrarán los *fragments*.

Cada una de las siguientes tres líneas introduce una nueva lengüeta usando el método `addTab()`. Se indican tres parámetros: un objeto `TabSpec`, una clase con el *fragment* a visualizar en la lengüeta y un `Bundle` por si queremos pasar información a la lengüeta. El método `newTabSpec()` crea una nueva lengüeta en un `TabHost`. Se le pasa como parámetro un `String`, que se utiliza como identificador y devuelve el objeto de tipo `TabSpec` creado.



Nota sobre Java: Dado que el método `newTabSpec()` devuelve un objeto de tipo `TabSpec`, tras la llamada, podemos llamar al método `setIndicator()`, que nos permitirá introducir un descriptor en la pestaña recién creada.

NOTA: También podremos asignar iconos a las lengüetas con el método `setIndicator()`. En el capítulo siguiente se estudiarán los iconos disponibles en el sistema y cómo crear nuevos iconos. En las últimas versiones de Android solo podemos visualizar un texto o un icono. Para ver el icono introduce un texto vacío.

4. Crea un nuevo layout y llámalo `tab1.xml`:

```

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >
    <TextView android:id="@+id/text"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:gravity="center_vertical|center_horizontal"
        android:text="Lengüeta 1"
        android:textAppearance=
            "?android:attr/textAppearanceMedium" />
</LinearLayout>

```

5. Crea una nueva clase con `Tab1.java`:

```

public class Tab1 extends Fragment {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
    }
}

```

```
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container,
    Bundle savedInstanceState) {
    return inflater.inflate(R.layout.tab1, container, false);
}
}
```

```
class Tab1 : Fragment() {
    override fun onCreateView(inflater: LayoutInflater,
        container: ViewGroup?, savedInstanceState: Bundle?): View? {
        return inflater?.inflate(R.layout.tab1, container, false)
    }
}
```



Nota sobre Java/Kotlin: Pulsa **Alt-Intro** para que automáticamente se añadan los paquetes que faltan. Si la clase `Fragment` se encuentra en más de un paquete, selecciona `android.support.v4.app.FragmentTabHost`.

Un *fragment* se crea de forma muy parecida a una actividad. También dispone del método `onCreate()`. En este ejemplo se llama al mismo método del antecesor, sin introducir nuevo código. Un *fragment* también tiene asociada una vista, aunque a diferencia de una actividad, no se asocia en el método `onCreate()`, si no que dispone de un método especial para esta tarea: `onCreateView()`. Dentro de este método vamos a utilizar un `LayoutInflater` para, a partir de un layout en XML, crear un objeto de la clase `View`. Para ello usaremos su método `inflate()`, que dispone de tres parámetros. El recurso de layout, el contenedor donde se tiene previsto insertar el layout y cuándo queremos insertarlo: ya mismo (`true`), o si de momento no queremos insertarlo (`false`).

6. Repite los dos pasos anteriores para crear `tab2.xml` y `Tab2.java`.
7. Repite de nuevo para crear el layout `tab3.xml` y la clase `Tab3.java`.
8. Modifica estos ficheros para que cada *layout* sea diferente y para que cada *fragment* visualice el *layout* correspondiente.
9. Ejecuta el proyecto y verifica el resultado.



NOTA: Si en uno de los layouts asignados a un *fragment* has utilizado el atributo `onClick`, el método indicado ha de ser introducido dentro de la actividad. Si lo introduces dentro del *fragment* no será reconocido.



Ejercicio: Añadir pestañas con `TabLayout`

1. Crea un nuevo proyecto de tipo *Basic Activity* y con nombre *Tabs*. La versión mínima de API ha de ser 17.
2. Reemplaza el código de `activity_main.xml` por el siguiente:

```
<androidx.coordinatorlayout.widget.CoordinatorLayout... >
```



```
<com.google.android.material.appbar.AppBarLayout... >
<androidx.appcompat.widget.Toolbar... />
<com.google.android.material.tabs.TabLayout
    android:id="@+id/tabs"
    android:layout_width="match_parent"
    android:layout_height="wrap_content" />
</com.google.android.material.appbar.AppBarLayout>
<com.google.android.material.floatingactionbutton.FloatingActionBu... />
<include layout="@layout/content_main" />
</androidx.coordinatorlayout.widget.CoordinatorLayout>
```

Observa como el `TabLayout` ha sido insertado dentro del `AppBarLayout`, es decir estarán dentro de la barra de acciones.

- En el método `onCreate()` de `MainActivity` añade el siguiente código:

```
//Pestañas
TabLayout tabs = findViewById(R.id.tabs);
tabs.addTab(tabs.newTab().setText("Pestaña 1"));
tabs.addTab(tabs.newTab().setText("Pestaña 2"));
tabs.addTab(tabs.newTab().setText("Pestaña 3"));
```

Comenzamos buscando la vista `TabLayout` del `Layout`. Las siguientes tres sentencias permiten insertar tres pestañas. Aparecerán en el orden en que se añaden usando el método `addTab()`. El método `newTab()` crea la nueva pestaña, que es configurada a continuación, asignándole un texto.

Ejecuta el proyecto y verifica que aparecen las tres pestañas.

- Para darle funcionalidad a las pestañas añade a continuación este código :

```
final TextView texto = findViewById(R.id.texto);
tabs.addOnTabSelectedListener(new TabLayout.OnTabSelectedListener() {
    @Override public void onTabSelected(TabLayout.Tab tab) {
        switch (tab.getPosition()) {
            case 0:
                texto.setText("Pestaña 1");
                break;
            case 1:
                texto.setText("Pestaña 2");
                break;
            case 2:
                texto.setText("Pestaña 3");
                break;
        }
    }
    @Override public void onTabUnselected(TabLayout.Tab tab) {}
    @Override public void onTabReselected(TabLayout.Tab tab) {}
});
```

Comenzamos buscando el `TextView` que queremos modificar. Luego, asignado un escuchador de eventos al `TabLayout` formado por tres métodos. Los nombres de los métodos nos indican claramente cuándo van a ser llamados. Los tres tienen como parámetro la pestaña que produce el evento (`Tab`). De los tres métodos solo introducimos código en el primero: Según la posición de la etiqueta seleccionada, modificamos el texto del `TextView`.

- Abre el fichero `content_main.xml` y añade un id al `TextView`:

```
<TextView
    android:id="@+id/texto"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Hello World!" />
```

Ejecuta el proyecto y verifica el resultado.

- Aunque no es el caso de nuestro ejemplo, en la mayoría de las ocasiones las pestañas se usan para asignar a cada una de ellas un *fragment*, de forma que al ser pulsadas se cambia el *fragment* que visualiza la actividad. Veamos cómo hacerlo.

En `content_main.xml` reemplaza el `TextView` por el siguiente código:


```
<androidx.viewpager.widget.ViewPager
    android:id="@+id/viewpager"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:layout_behavior="@string/appbar_scrolling_view_behavior"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toTopOf="parent"/>
```

7. Dentro de `MainActivity` añade el siguiente código al final (justo antes de `}`):

```
public class MiPagerAdapter extends FragmentPagerAdapter {

    public MiPagerAdapter(FragmentManager fm) {
        super(fm);
    }

    @Override public int getCount() {
        return 3;
    }

    @Override public CharSequence getPageTitle(int position) {
        switch (position) {
            case 0: return "Fragment 1";
            case 1: return "Fragment 2";
            case 2: return "Fragment 3";
        }
        return null;
    }

    @Override public Fragment getItem(int position) {
        Fragment fragment = null;
        switch (position) {
            case 0: fragment = new Tab1();
                    break;
            case 1: fragment = new Tab2();
                    break;
            case 2: fragment = new Tab3();
                    break;
        }
        return fragment;
    }
}
```

Esta clase es un adaptador (`Adapter`), es un mecanismo muy utilizado en Android para hacer de puente entre nuestros datos y el interfaz de usuario. Más adelante lo utilizaremos para rellenar los datos de un `RecyclerView` o un `Spinner`. Para este ejemplo, lo que hace es indicar cuantos tabs queremos (`getCount()`), el título para cada tab (`getPageTitle(int)`) y que `Fragment` visualizamos en cada tab (`getItem(int)`).

8. En `onCreate()` reemplaza todo el código creado para los tabs por:

```
ViewPager viewPager = findViewById(R.id.viewpager);
viewPager.setAdapter(new MiPagerAdapter(getSupportFragmentManager()));
TabLayout tabs = findViewById(R.id.tabs);
tabs.setupWithViewPager(viewPager);
```

9. Crea un nuevo `layout` y llámalo `tab1.xml`:

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

```

        android:layout_height="match_parent"
        android:gravity="center_vertical|center_horizontal"
        android:text="Pestaña 1"
        android:textAppearance=
            "?android:attr/textAppearanceMedium" />
    </LinearLayout>

```

10. Crea una nueva clase con *Tab1.java*:

```

public class Tab1 extends Fragment {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
    }
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState) {
        return inflater.inflate(R.layout.tab1, container, false);
    }
}

```

```

class Tab1 : Fragment() {
    override fun onCreateView(inflater: LayoutInflater,
        container: ViewGroup?, savedInstanceState: Bundle?): View? {
        return inflater?.inflate(R.layout.tab1, container, false)
    }
}

```

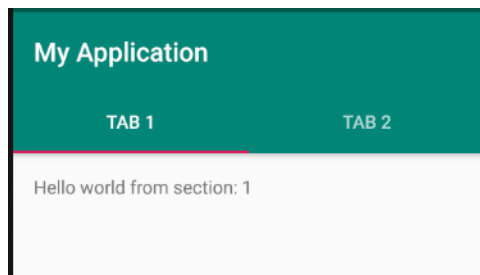
Un *fragment* se crea de forma muy parecida a una actividad. También dispone del método `onCreate()`. En este ejemplo se llama al mismo método del antecesor, sin introducir nuevo código. Un *fragment* también tiene asociada una vista, aunque a diferencia de una actividad, no se asocia en el método `onCreate()`, si no que dispone de un método especial para esta tarea: `onCreateView()`. Dentro de este método vamos a utilizar un `LayoutInflater` para, a partir de un layout en XML, crear un objeto de la clase `View`. Para ello usaremos su método `inflate()`, que dispone de tres parámetros. El recurso de layout, el contenedor donde se tiene previsto insertar el layout y cuándo queremos insertarlo: ya mismo (`true`), o si de momento no queremos insertarlo (`false`).

11. Repite los dos pasos anteriores para crear *tab2.xml* y *Tab2.java*.
12. Repite de nuevo para crear el layout *tab3.xml* y la clase *Tab3.java*.
13. Modifica estos ficheros para que cada *layout* sea diferente y para que cada *fragment* visualice el *layout* correspondiente.

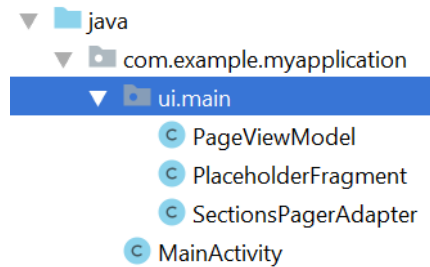


Ejercicio: Añadir pestañas con *TabLayout* de forma automática

1. Crear un nuevo proyecto y seleccionar como actividad inicial *Tabbed Activity*. Si ya dispones de un proyecto y quieres añadir una actividad con pestañas utiliza la opción *File / New / Activity / Tabbed Activity*.
2. Ejecuta el proyecto y verifica que el resultado es parecido al ejercicio anterior:



3. En el explorador del proyecto observa como además de MainActivity se han creado tres clase tres clases en el paquete *com.example.myapplication.ui.main*:



Se ha creado este subpaquete para dar a entender que estas tres clases son utilizadas por la actividad main.

SectionsPagerAdapter tendría una función similar a *MiPagerAdapter*. Es decir, indicar cuantas pestañas queremos, sus nombres y los *Fragments* para cada una.

PlaceholderFragment es el fragment que pondremos dentro de las pestañas. En lugar de crear tres fragments diferentes (*Tab1*, *Tab2* y *Tab3*) tendremos una única clase a la que pasaremos un valor entero según la pestaña a crear. Cuando vaya a crear la vista utilizará este valor entero para personalizarla. Realmente la personalización se hace a través de la siguiente clase.

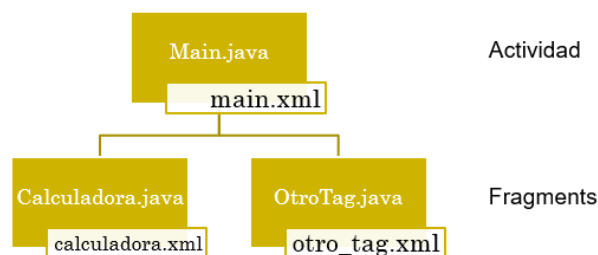
PageViewModel La clase *ViewModel* permite almacenar datos relacionados con la interfaz de usuario. Su principal función es que los datos sobrevivan a los cambios de configuración, como las rotaciones de pantalla.

4. Trata de modificar el proyecto para que se generen tres pestañas en lugar de dos.

2.10.1. Evento onClick en un Fragment

Trabajar con una actividad que contiene varios fragments, suele ocasionar ciertos errores en la programación que se repiten con mucha frecuencia. Para ayudarte a realizar este trabajo correctamente, hemos introducido este apartado.

Supongamos que tenemos la actividad *Main.java* que visualiza el layout *main.xml*. Dentro de esta actividad se pueden visualizar dos fragments, aunque no de forma simultánea. Por ejemplo, se podría visualizar uno u otro usando un *TabLayout*. El siguiente esquema muestra los nombres de los fragments y los layouts que estos visualizan.



Vamos a suponer que dentro del layout *calculadora* se ha añadido en uno de sus botones un atributo *onClick*:

```
<Button ... onClick="sePulsa">
```

Importante: el método *sePulsa(View v)* hay que declararlo en la actividad. Es en la actividad donde lo va a buscar, si lo pones en otro sitio no lo encontrará.

En este punto se nos plantea una cuestión: ¿Dónde poner el código que gestiona el comportamiento de un fragment? ¿En la clase de la actividad (*Main.java*)? ¿O en la del fragment (*Calculadora.java*)? La respuesta puede variar según el contexto, pudiéndose dar dos casos principales:

Los fragments son parte del mismo proceso

En el caso de que tanto la actividad como los fragments son parte del mismo proceso, podemos centralizar todo el código en la actividad. Dividir el código en varias clases, si se trata de un código con

una sola función, puede ser confuso. El código de cada fragment solo tendrá que mostrar el layout correspondiente.

En este supuesto puede darse el caso de que queramos acceder a una vista definida dentro del layout de un fragment. Resulta frecuente querer disponer de esta vista en un objeto, e inicializarlo en el método `onCreate()`. Pero cuidado, si lo intentamos hacer en el método `onCreate()` de la actividad nos dará un error. En este método tras llamar al `super`, podemos suponer que la actividad ya está creada y también su layout. Sin embargo, los diferentes fragments no están todavía creados, por lo que al tratar de acceder a una de sus vistas, va a dar un error. Para resolver el problema podemos usar el siguiente código:

```
public class Main extends Activity {
    EditText cantidad;

    @Override public void onCreate(...) {
        super.onCreate(...);
        cantidad = findViewById(R.id.editText1); //NO FUNCIONA
    }

    public void sePulsa(View view) {
        if (cantidad == null) {
            cantidad = findViewById(R.id.editText1);
        }
        cantidad.setText(cantidad.getText() + (String) view.getTag());
    }
}
```

Lo que hacemos es posponer la creación del objeto `cantidad` hasta que sepamos seguro que el fragment donde aparece ha sido creado. En el ejemplo, si el método `sePulsa()` está asociado a un botón que aparece en el mismo fragment, cuando se ejecute el método podemos estar seguros que el fragment está creado. Como solo hace falta crearlo una vez, comparamos si en la variable hay `null` para crearlo. Si no, es que ya está creado.

Los fragments son independientes

Se puede dar el caso de que cada fragment tenga una función diferente. Por ejemplo, en uno hay una calculadora y en otro un diccionario. En este caso es importante que el código de cada fragment se escriba en su clase. Si actuamos de esta forma, podremos reutilizar el fragment en otra aplicación. Por ejemplo, si queremos añadir una calculadora en otra aplicación, no tendremos más que incluir su clase `Calculadora.java` con su layout.

Aquí aparece un problema, si queremos controlar la pulsación de un botón de la calculadora y lo hacemos con el atributo `onClick`, el método llamado ha de estar en la actividad. Esto rompe el principio de que todo el código que controle el fragment ha de estar en su clase.

Para resolver el problema vamos a utilizar un método alternativo, que consiste en programar un escuchador de evento por código. Veamos un ejemplo:

```
public class Calculadora extends Fragment {
    EditText cantidad;
    ...
    @Override public View onCreateView(...) {
        View v = inflater.inflate(R.layout.calculadora, container, false);
        cantidad = v.findViewById(R.id.editText1);
        Button boton = v.findViewById(R.id.Button00);
        boton.setOnClickListener(new OnClickListener() {
            public void onClick(View view) {
                sePulsa(view);
            }
        });
    }

    public void sePulsa(View view) {
        cantidad.setText(cantidad.getText() + (String) view.getTag());
    }
}
```

Si lo comparamos con el caso anterior, la asignación del layout al fragment se realiza de forma diferente. Ahora usamos en método `onCreateView()`, en lugar de `onCreate()`. El layout es creado en la vista, `v`, usando un el inflador, `inflater`. Recuerda que un inflador nos convierte un fichero XML a su correspondiente objeto Java. Llamando a `v.findViewById()` podemos extraer vistas concretas del layout.

Tras extraer `boton`, le asociamos un escuchador de evento `onClick` llamando a `setOnClickListener()`. Para crear un escuchador de eventos en Java, hay que crear un objeto de la clase adecuada `OnClickListener` y escribir un método para cada uno de los eventos que el botón puede generar. En nuestro caso solo el método `onClick(View)`. El parámetro que recibe el método corresponde al objeto que lanzo el evento. Tal y como se ha escrito el código solo puede ser `botón`.