



TEMA 3. DISEÑO DE LA INTERFAZ DE USUARIO

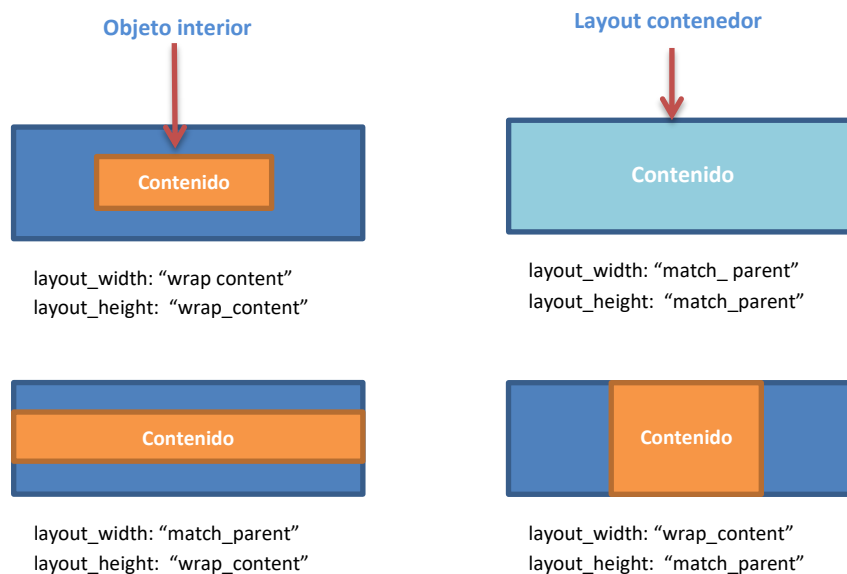
1. INTERFACES DE USUARIO. CLASES ASOCIADAS	2
2. LAYOUTS	4
2.1. LinearLayout	4
2.2. TableLayout	5
2.3. GridLayout	5
2.4. FrameLayout	6
2.5. AbsoluteLayout	6
2.6. RelativeLayout	7
2.7. ConstraintLayout	7
2.8. Elección de los Layout	8
3. CONTROLES BÁSICOS	8
3.1. TextView (Etiquetas)	8
3.2. Botones	9
3.2.1. Button	9
3.2.2. ToggleButton	10
3.2.3. ImageButton	10
3.3. EditText (Cuadro de Texto)	11
3.4. ImageView (Imagen)	12
3.5. AutoCompleteTextView	12
3.6. MultiAutoCompleteTextView	13
3.7. Spinner	13
3.8. CheckBox (Caja de selección)	14
3.9. RadioButton (Botón de radio)	15
3.10. Switch	16
3.11. RatingBar (Barra de puntuación)	17
3.12. SeekBar	17
3.13. ProgressBar (Barra de progreso)	18
4. EVENTOS DEL USUARIO	19
4.1. Uso de los Event Listeners	19
5. ANEXO	22
5.1. Problemas con clearCheck y eventos de RadioGroup	22
5.2. Obtener los elementos seleccionados de un grupo de CheckBox	23



1. INTERFACES DE USUARIO. CLASES ASOCIADAS

El interfaz de una aplicación Android es aquello que aparece en pantalla, que el usuario puede ver y, por tanto, interactuar con él. Android nos ofrece un gran número de componentes implementados y preparados para su uso en la interfaz, en los cuales la **clase View** sirve siempre como clase base.

La librería `android.view` nos proporciona los elementos de la interfaz para construir las vistas, lo que se puede hacer usando Java, en principio más complejo y, por tanto, menos eficiente, o bien definiéndolo en el archivo XML ubicado en el directorio `res/layout`.



La **clase View** como clase principal en la jerarquía de vistas tiene una serie de atributos que serán heredados por todos los elementos que de ella provienen. A cualquiera de ellos le puedes asignar valores medidos (entre comillas) en pulgadas (in), milímetros (mm), puntos (pt), píxeles (px), píxeles independientes de la densidad (dp) y píxeles independientes de la escala (sp). Más adelante se verá el significado y utilidad de cada uno; por el momento, es recomendable utilizar como unidad de medida los píxeles independientes de la densidad (dp).

- **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, sino 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.

Atributos de posicionamiento	layout_width	Ancho
	layout_height	Alto
Atributos para los márgenes	layout_margin	Cuatro márgenes
	layout_marginBottom	Margen inferior
	layout_marginLeft	Margen izquierdo
	layout_marginRight	Margen derecho
	layout_marginTop	Margen superior
Atributos para el espaciado	android:padding	Espaciado a los cuatro lados
	android:paddingtop	Espaciado superior
	android:paddingbottom	Espaciado inferior
	android:paddingleft	Espaciado izquierdo
	android:paddingleft	Espaciado derecho


Para posicionar los objetos que heredan de la clase View, además de poder tomar valores absolutos (en cualquiera de las unidades citadas), también pueden asignársele los valores relativos *wrap_content*, cuando se desea ajustar el tamaño al contenido del objeto, o *match_parent* (antes *fill_parent*) para tomar el máximo tamaño que le permite el contenedor.

Si se desea centrar o justificar la vista, se utilizará el atributo *layout_gravity*, y para distribuir el espacio disponible entre los diferentes objetos, *layout_weight*. Por último, para poder identificar el objeto debe utilizarse **@id/+ nombre** y para acceder a este identificador, **@id/nombre**. También pueden utilizarse identificadores definidos por el sistema **@android:id/nombre**.

Dentro de esta gran clase se encuentra la clase *android.view.ViewGroup*. Ella proporciona, como su propio nombre indica, los objetos *ViewGroup*, cuya función es contener y controlar colecciones de *View ()* de otros *ViewGroups*.



Los objetos *Views* han de ser hijos de objetos *ViewGroup*, por lo tanto, se ha de comenzar el diseño de la interfaz con un objeto *ViewGroup*, a partir del cual se puede realizar un árbol con tantas bifurcaciones como se desee.

- 
- ✓ Los **atributos de posicionamiento** deberán usarse siempre que se coloque un objeto en la vista de la aplicación.
 - ✓ Los **atributos de espaciado y márgenes** tan solo se usarán cuando se desee mejorar la estética de la aplicación.
 - ✓ Siempre es recomendable **identificar los objetos** en la vista aunque luego no se vaya a acceder a ellos desde el Java de la aplicación.

2. LAYOUTS

Los layouts son elementos no visuales destinados a controlar la distribución, la posición y las dimensiones de los controles que se insertan en su interior. Dentro de cada una pueden ubicarse todos los elementos que sean necesarios en el interfaz de la actividad, incluidos otros layouts, lo que permitirá estructurar la pantalla de la manera deseada. En función de la forma y el posicionamiento en pantalla, existe gran variedad de layouts.

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 en relación a otro o al padre.
- **ConstraintLayout:** Versión mejorada de RelativeLayout, que permite una edición visual desde el editor.
- **FrameLayout:** Permite el cambio dinámico de los elementos que contiene.
- **AbsoluteLayout:** Posiciona los elementos de forma absoluta.

2.1. LinearLayout

Es uno de los Layout más utilizado en la práctica. Distribuye los elementos uno detrás de otro, bien de forma horizontal o vertical. Para ello, hace uso del atributo `android:orientation`.



2.2. TableLayout

El panel TableLayout permite distribuir todos sus componentes hijos como si se tratara de una tabla mediante filas y columnas. La estructura de la tabla se define de manera similar a una tabla en formato HTML, es decir, indicando las filas que compondrán la tabla (objetos TableRow) y las columnas de cada una de ellas.

Por norma general, el ancho de cada columna corresponde al ancho del mayor componente de dicha columna, pero existen una serie de propiedades pueden modificar este comportamiento:

- *android:stretchColumns*: indica el número de columna que se expande para ocupar el espacio libre que dejan el resto de columnas a la derecha de la pantalla.
- *android:shrinkColumns*: indica las columnas que se pueden contraer para dejar espacio al resto de columnas de lado derecho de la pantalla.
- *android:collapseColumns*: indica las columnas de la tabla que se pueden ocultar completamente.

Todas estas propiedades del TableLayout pueden establecerse con una lista de índices de las columnas separados por comas, por ejemplo: *android:stretchColumns="1,2,3"* o un asterisco para indicar que se debe aplicar a todas las columnas, de esta forma: *android:stretchColumns="*"*.

- *android:layout_span*: una celda determinada puede ocupar el espacio de varias columnas de la tabla (análogo al atributo colspan de HTML) del componente concreto que ocupa dicho espacio.

También están disponibles las propiedades *android:layout_rowSpan* y *android:layout_columnSpan* para conseguir que una celda ocupe el lugar de varias filas o columnas.

2.3. GridLayout

El panel GridLayout está disponible a partir de la API 14 (Android 4.0) y sus características son similares al TableLayout, ya que se usa para distribuir los diferentes elementos de la interfaz de forma tabular mediante filas y columnas. La diferencia entre estos paneles radica en la forma que tiene el GridLayout de situar y distribuir sus elementos hijos en el espacio disponible.

En este caso, es necesario indicar el número de filas y columnas estableciendo sus propiedades *android:rowCount* y *android:columnCount*. Así, ya no es necesario incluir ningún elemento adicional del tipo TableRow para indicar las filas, como sí se



hace con el panel `TableLayout`. Es decir, los elementos hijos se irán colocando ordenadamente en las celdas de las filas y columnas (dependiendo de la orientación del dispositivo: propiedad `android:orientation`) hasta completar todas las celdas indicadas en los atributos anteriores. Adicionalmente, como ocurre con el caso anterior, también están disponibles las propiedades `android:layout_rowSpan` y `android:layout_columnSpan` para conseguir que una celda ocupe el lugar de varias filas o columnas.


Existe también una manera de indicar de forma explícita la fila y la columna donde un determinado elemento hijo debe incluirse mediante los atributos `android:layout_row` y `android:layout_column` (el primer valor sería el 0). En general, no suele ser necesario aplicar estas propiedades, salvo en diseños complejos.

2.4. `FrameLayout`

Éste es el panel más sencillo de todos los Layouts de Android. Un panel `FrameLayout` coloca todos sus componentes hijos alineados pegados a su esquina superior izquierda de forma que cada componente nuevo añadido oculta el componente anterior. Por esto, se suele utilizar para mostrar un único control en su interior, a modo de contenedor (placeholder) sencillo para un único elemento, por ejemplo, una imagen.

Este contenedor ofrece la posibilidad de modificar la visibilidad del elemento deseado dentro de su contenido, para lo cual será necesario utilizar la propiedad `android:visibility`, lo que es muchas veces útil para determinados efectos de animación.

Para colocar los objetos hijos en la posición deseada, podemos utilizar la propiedad `android:gravity`.



No debe confundirse `android:gravity` con `android:layout_gravity`: el primero establece la posición del contenido y el segundo posiciona el objeto con respecto a su elemento padre.

2.5. `AbsoluteLayout`

Permite indicar las coordenadas absolutas de pantalla donde se quiere visualizar cada elemento:

`android:layout_x="posición horizontal"` `android:layout_y="posición vertical"`

Los fabricantes crean dispositivos de muy variada resolución, por lo que Android nos permite un rango de resoluciones de pantalla muy variado, lo que hace que el contenedor distribuya los elementos que acoge según sus propias reglas, y que el resultado no pueda ser el deseado. Por ello, no es recomendable utilizar este tipo de contenedor, que hoy día ha sido marcado como obsoleto.



2.6. RelativeLayout

El panel RelativeLayout permite especificar la posición de cada componente de forma relativa a su elemento padre o a cualquier otro elemento incluido en el propio layout.

Un panel RelativeLayout dispone de múltiples propiedades para colocar cada componente. Las principales son:

Posición relativa a otro control:

- *android:layout_above*: arriba.
- *android:layout_below*: debajo.
- *android:layout_toLeftOf*: a la izquierda de.
- *android:layout_toRightOf*: a la derecha de.

Alineación con respecto al control

- *android:layout_alignLeft*: alinear a la izquierda.
- *android:layout_alignRight*: alinear a la derecha.
- *android:layout_alignTop*: alinear arriba.
- *android:layout_alignBottom*: alinear abajo.
- *android:layout_alignBaseline*: alinear en la base.

Posición relativa al layout padre:

- *android:layout_alignParentLeft*: alinear a la izquierda.
- *android:layout_alignParentRight*: alinear a la derecha.
- *android:layout_alignParentTop*: alinear arriba.
- *android:layout_alignParentBottom*: alinear abajo.
- *android:layout_centerHorizontal*: alinear horizontalmente al centro.
- *android:layout_centerVertical*: alinear verticalmente al centro.
- *android:layout_centerInParent*: centrar.

2.7. ConstraintLayout

Con la llegada de Android Studio 2.2, Google presentó este nuevo Layout con el que cambiar radicalmente la forma de diseñar interfaces gráficas. ConstraintLayout permite simplificar el anidamiento de interfaces con un diseño visual para el que utiliza herramientas *drag and drop*.

Siguiendo el estilo del RelativeLayout, los objetos se posicionan en relación a objetos hermanos y padres, pero es más flexible y más sencillo de usar mediante la vista diseño de Android Studio.



2.8. Elección de los Layout

Algunas orientaciones sobre cuando usar cada layout podrían ser:

- **LinearLayout:** Diseños muy sencillos.
- **RelativeLayout:** Hay una nueva alternativa, por lo que es mejor usarla.
- **ConstraintLayout:** Usar por defecto.
- **FrameLayout:** Varias vistas superpuestas.
- **AbsoluteLayout:** Nunca. Aunque está bien conocerlo por si acaso.

3. CONTROLES BÁSICOS

3.1. TextView (Etiquetas)

Permite mostrar un texto al usuario. Sus parámetros más importantes son:

- *android:text*: define cuál será el texto que se mostrará en pantalla.
- *android:background*: establece el color de fondo.
- *android:textColor*: asigna el color de texto.
- *android:textSize*: determina el tamaño del texto (es aconsejable utilizar unidades de medida sp).
- *android:textStyle*: elige el estilo del texto: normal, negrita (bold) o cursiva (italic).
- *android:typeface*: fija el tipo de letra por el que podemos optar: sans, monospace o serif.

```
<TextView
    android:id="@+id/texto1"
    android:text="Texto Plano"
    android:textSize="30sp"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"/>
```

Para poder manipular este objeto desde Java, primero debe asociarse a la vista, para lo que es necesario importar la librería widget (esto suele hacerlo de manera automática Android Studio) que da acceso a los objetos incluidos en el interfaz. En este caso, se necesitará *import Android.widget.TextView*;

Si Android Studio deja librerías sin importar, puede hacerse de forma manual, posicionando el cursor sobre los objetos no reconocidos y pulsando Alt+Intro.





A continuación se deberá instanciar el objeto y acceder al recurso:

```
final TextView lblEtiqueta = (TextView)findViewById(R.id.texto1);
```

Donde *lblEtiqueta* es el nombre que adquirirá esta instancia en Java y *R.id.texto* es el nombre con el que lo identificamos en el fichero XML de la vista asociada a este objeto.

Una vez hecho esto, se pueden obtener (*get*) o modificar (*set*) los atributos de este objeto, y así, si se quisiera cambiar el contenido de la etiqueta de texto, tan solo se tendría que escribir:

```
lblEtiqueta.setText("Esto es un texto");
```

3.2. Botones

3.2.1. Button

Permite crear botones en la interfaz gráfica. Esta clase hereda de *TextView*, por lo que tiene toda su funcionalidad.

Podemos crearlo desde XML con el siguiente código:

```
<Button  
    android:id="@+id/btn1"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:text="Botón"/>
```

Para [manejar los eventos de este objeto](#), tenemos que crear un *manejador de eventos* desde código Java. Para ello, se debe poner un identificador al botón en la vista (en este caso, *miBoton*), y luego instanciarlo y asociarle un manejador de eventos. Sería de la siguiente forma:

```
miBoton.setOnClickListener(new View.OnClickListener() {  
    @Override  
    public void onClick(View view) {  
        // Código que se ejecutará  
    }  
});
```

Para cambiar el color de fondo de un botón, tengo que utilizar la propiedad *android:backgroundTint*.



3.2.2. *ToggleButton*

Es un tipo de botón que puede encontrarse en dos estados: pulsado y no pulsado. En este caso, en lugar de definir un solo texto, podemos definir dos: uno para cuando está pulsado y otro para cuando no. Esto se hace asignando las propiedades *android:textOn* y *android:textOff*, respectivamente.

```
<ToggleButton
    android:id="@+id/toggleBtn"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textOn="Pulsado"
    android:textOff="No pulsado"/>
```

El *listener* de este objeto podrá ser del tipo *OnCheckedChangeListener()*, creando para ello un *CompoundButton*.

```
final ToggleButton miToggleBtn = (ToggleButton)
findViewById(R.id.toggleBtn);
miToggleBtn.setOnCheckedChangeListener(new
CompoundButton.OnCheckedChangeListener() {
    @Override
    public void onCheckedChanged(CompoundButton compoundButton,
boolean isChecked) {
        if (isChecked) {
            // El botón está activado
        }
        else {
            // El botón no está activado
        }
    }
});
```

Otra forma de hacerlo sería con el listener *OnClickListener* y utilizar el método *isChecked()* para comprobar su estado.

3.2.3. *ImageButton*

Es un botón que muestra una imagen en lugar de un texto asignando la propiedad *android:src*. Normalmente, indicamos esta propiedad usando el descriptor de alguna imagen que hayamos copiado en la carpeta */res/drawable*.

La imagen la podemos establecer desde el diseño del layout o desde Java.



Diseño del layout

```
<ImageButton  
    android:id="@+id/imgBtn"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:src="@drawable/stop"/>
```

Java

```
final ImageButton img = (ImageButton) findViewById(R.id.imgBtn);  
img.setImageResource(R.drawable.stop);
```

3.3. EditText (Cuadro de Texto)

Permite la introducción y edición de texto por parte del usuario. Este objeto hereda todas las propiedades de TextView.

```
<EditText  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"/>
```

Por defecto, EditText lee texto plano y asocia el teclado textual del dispositivo móvil. Dispone de numerosas opciones para mostrar el contenido, gracias al modificador *inputType*, según la conveniencia de la aplicación desarrollada:

- *textUri*: texto que se usará como URL.
- *textEmailAddress*: texto que se usará como dirección de correo.
- *textPersonName*: nombre de una persona.
- *textPassword*: contraseña.
- *textVisiblePassword*: contraseña que se mostrará.
- *number*: entrada numérica
- *date*: texto tratado como fecha.
- *time*: texto tratado como hora.
- *textMultiLine*: permite multilínea.
- *textCapSentences*: pone en mayúsculas la primera letra de cada frase.
- *textCapWords*: pone en mayúsculas la primera letra de cada palabra.
- *textNoSuggestions*: desactiva la escritura predictiva.

En caso de querer utilizar más de una opción, se utilizará el separador |





Otras opciones interesantes serían:

- *android:hint*: pone un texto predefinido (que aparecerá en gris).
- *android:lines*: limita el número de líneas.
- *android:digits*: limita el número de dígitos (numérico).
- *android:maxLength*: limita el número de caracteres.

3.4. ImageView (Imagen)

Permite mostrar imágenes en la aplicación. La propiedad más útil es *android:src*, que permite establecer la imagen que se muestra. Lo usual es indicar como origen de la imagen el identificador de un recurso de la carpeta */res/drawable*.

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

Si en lugar de hacer el diseño en XML lo queremos hacer desde la lógica de la aplicación, haríamos lo siguiente:

```
ImageView img = (ImageView) findViewById(R.id.img);
img.setImageResource(R.drawable.ic_launcher);
```

3.5. AutoCompleteTextView

Variante del TextView al que se añaden sugerencias de autocompletado. Para ello usa la propiedad *android:completionThreshold*, que define el número de caracteres mínimo que se han de escribir para que la aplicación ofrezca una sugerencia.

```
<AutoCompleteTextView
    android:id="@+id/acText"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:hint="Elige una opción"
    android:completionThreshold="4"/>
```

Los elementos que se sugerirán deben ser definidos en el fichero Java en un array de String, al que se le debe crear un adaptador de tipo *ArrayAdapter* (más adelante se estudiarán los adaptadores en más profundidad), cuya misión es adaptar los datos del



array al *AutoCompleteTextView*. Este adaptador se pasará como argumento en el constructor, junto al contexto de la actividad y la forma de mostrarse (en este caso, en forma de lista desplegable).

Por último, se pasa el adaptador a la instancia del texto definido en el XML. Como en casos anteriores, deberán importarse las librerías necesarias, que aquí serán `android.widget.ArrayAdapter` y `android.widget.AutoCompleteTextView`. El código final quedaría así:

```
String[] opciones = {"Opción 1", "Opción 2", "Opción 3", "Opción 4", "Opción 5"};  
AutoCompleteTextView textoLeido = (AutoCompleteTextView) findViewById(R.id.acText);  
ArrayAdapter<String> adaptador = new ArrayAdapter<String>(this,  
    android.R.layout.simple_dropdown_item_1line, opciones);  
textoLeido.setAdapter(adaptador);
```

3.6. MultiAutoCompleteTextView

La diferencia entre este objeto y el anterior es que este permite ofrecer sugerencias para cada bloque de texto introducido, marcando mediante un delimitador (token) el elemento a partir del cual volverá a hacernos la sugerencia. En el siguiente ejemplo, se usa la coma como elemento delimitador.

```
MultiAutoCompleteTextView textoLeido2 = (MultiAutoCompleteTextView)  
    this.findViewById(R.id.macText);  
ArrayAdapter<String> adaptador2 = new ArrayAdapter<String>(this,  
    android.R.layout.simple_dropdown_item_1line, opciones);  
textoLeido2.setTokenizer(new MultiAutoCompleteTextView.CommaTokenizer());  
textoLeido2.setAdapter(adaptador2);
```

3.7. Spinner

Muestra una lista desplegable para elegir un único elemento.

```
<Spinner  
    android:id="@+id/miSpinner"  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"/>
```

En Java este objeto depende de la librería *android.widget.Spinner*, y para su uso deberá instanciarse, asignarle un array de valores y un adaptador (del tipo *simple_spinner_item*).

```
Spinner spinner = (Spinner) findViewById(R.id.miSpinner);  
String[] valores = {"Valor 1", "Valor 2", "Valor 3", "Valor 4", "Valor 5"};  
spinner.setAdapter(new ArrayAdapter<String>(this,  
    android.R.layout.simple_spinner_item, valores));
```



También puede añadirse la lista de valores directamente al XML mediante el comando `android:entries="@array/valores"`, debiéndose crear previamente el fichero `res/values/valores.xml`.

Para leer el elemento que en ese momento está seleccionado escribiremos el siguiente código:

```
Spinner spinner = (Spinner) findViewById(R.id.miSpinner);  
String valor = spinner.getSelectedItem().toString();
```

Para crear un listener que responda a los elementos de cambio, primero tendremos que crear un adaptador, y posteriormente utilizaremos `setOnItemSelectedListener`, que puede responder a dos eventos: `onItemSelected` y `onNothingSelected`. Este devolverá el contenido del array que está en el adaptador.

Adaptador

```
ArrayAdapter<CharSequence> adapter =  
    ArrayAdapter.createFromResource(this, R.array.valores,  
        android.R.layout.simple_spinner_item);  
spinner.setAdapter(adapter);
```

Manejador de eventos

```
spinner.setOnItemSelectedListener(new AdapterView.OnItemSelectedListener() {  
    @Override  
    public void onItemSelected(AdapterView<?> adaptador, View view, int  
        posicion, long id) {  
        adaptador.getItemAtPosition(posicion);  
    }  
  
    @Override  
    public void onNothingSelected(AdapterView<?> adapterView) {  
    }  
});
```

3.8. CheckBox (Caja de selección)

Permite marcar o desmarcar opciones en una aplicación.

```
<CheckBox  
    android:id="@+id/miCheck"  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:text="Pulsa opción"  
    android:checked="false"/>
```



En el código de la aplicación podemos utilizar los métodos *isChecked()* para conocer el estado del componente y *setChecked(boolean)* para establecer un estado en concreto.

Para capturar los cambios de estado, utilizaremos el manejador de eventos basado en *setOnCheckedChangeListener* y en *onCheckedChangeListener*.

```
CheckBox checkBox = (CheckBox) findViewById(R.id.miCheck);
checkBox.setOnCheckedChangeListener(new CompoundButton.OnCheckedChangeListener() {
    @Override
    public void onCheckedChanged(CompoundButton compoundButton, boolean pulsado) {
        if (pulsado){
            // Código para pulsado
        }
        else{
            // Código para no pulsado
        }
    }
});
```

3.9. RadioButton (Botón de radio)

Permite elegir una opción entre un conjunto. Pueden agruparse mediante *RadioGroup*, de tal manera que de los que estén incluidos en este grupo, solo uno de ellos puede y debe estar marcado.

```
<RadioGroup
    android:id="@+id/grupo"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical">

    <RadioButton
        android:id="@+id/radio1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Opción " />

    <RadioButton
        android:id="@+id/radio2"
        android:layout_width="wrap_content"
        android:layout_height="match_parent"
        android:text="Opción 2" />

</RadioGroup>
```

Podemos usar los métodos *check* para **marcar**, *clearCheck* para **desmarcar** o **localizar el elemento marcado** de la siguiente manera:



```
RadioGroup miGrupo = (RadioGroup) findViewById(R.id.grupo);  
miGrupo.clearCheck();  
miGrupo.check(R.id.radio1);  
int idMarcado = miGrupo.getCheckedRadioButtonId();
```

Para asociarle un manejador de eventos, procederíamos de la siguiente manera:

```
RadioGroup miGrupo = (RadioGroup) findViewById(R.id.grupo);  
miGrupo.setOnCheckedChangeListener(new RadioGroup.OnCheckedChangeListener() {  
    {  
        @Override  
        public void onCheckedChanged(RadioGroup radioGroup, int checkedId) {  
            // Código  
        }  
    }  
});
```

Para obtener el botón que ha sido seleccionado, podemos hacerlo desde el manejador con el siguiente código:

```
final RadioButton rb = (RadioButton) findViewById(checkedId);
```

3.10. Switch

Este elemento consta de dos estados, encendido (marcado) o apagado (desmarcado), bastante parecido al ToogleButton, pero simulando un control deslizante.

```
<Switch  
    android:id="@+id/miSwitch"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"/>
```

Para manejar los eventos debe procederse de manera similar que en anteriores ocasiones.



```
Switch pulsador = (Switch) findViewById(R.id.miSwitch);
pulsador.setOnCheckedChangeListener(new CompoundButton.OnCheckedChangeListener() {
    @Override
    public void onCheckedChanged(CompoundButton compoundButton, boolean pulsado) {
        if (pulsado){
            // Código para pulsado
        }
        else{
            // Código para no pulsado
        }
    }
});
```

3.11. RatingBar (Barra de puntuación)

Control de selección diseñado para dar una puntuación. Los atributos de este control son múltiples y sería conveniente consultar la página oficial de desarrolladores de Android. En el siguiente ejemplo, se fijan el número de estrella y el valor de los incrementos. En algunos dispositivos, el número de estrellas debe ajustarse a los márgenes establecidos para este control.



```
<RatingBar
    android:id="@+id/miRating"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:numStars="7"
    android:rating="1.0"
    android:stepSize="1.0"/>
```

El manejador de eventos será utilizado de la siguiente forma:

```
RatingBar miControl = (RatingBar) findViewById(R.id.miRating);
miControl.setOnRatingBarChangeListener(new RatingBar.OnRatingBarChangeListener() {
    @Override
    public void onRatingChanged(RatingBar ratingBar, float v, boolean b) {
        // Código para cambio de valor
    }
});
```

3.12. SeekBar

Sirve para elegir un valor numérico entre un rango de valores predefinidos en la aplicación. Algunos atributos serían:

- *android:max*: define el valor máximo del control.
- *android:min*: define el valor mínimo del control.
- *android:progress*: fija el punto de partida del control.
- *android:thumb*: personaliza la imagen del elemento deslizable.



- *android:rotation*: permite girar el control y ponerlo verticalmente.
- *android:progressDrawable*: personaliza el aspecto de la barra de desplazamiento.



```
<SeekBar
    android:id="@+id/miSeekBar"
    android:layout_width="match_parent"

    android:layout_height="wrap_content"
    android:max="10"
    android:progress="0"/>
```

Este objeto posee un *listener* que nos permite manejar tres eventos, uno para cuando empezamos a mover el control (*onStartTrackingTouch*), durante el movimiento (*onProgressChanged*) y al dejar de actuar sobre él (*onStopTrackingTouch*).

```
SeekBar miControl2 = (SeekBar) findViewById(R.id.miSeekBar);
miControl2.setOnSeekBarChangeListener(new SeekBar.OnSeekBarChangeListener()
{
    @Override
    public void onProgressChanged(SearchBar seekBar, int i, boolean b) {
        // Código para cambio de valor
    }

    @Override
    public void onStartTrackingTouch(SearchBar seekBar) {
        // Código para inicio de cambio
    }

    @Override
    public void onStopTrackingTouch(SearchBar seekBar) {
        // Código para final de cambio
    }
});
```

3.13. ProgressBar (Barra de progreso)

Sirve para informar al usuario de la evolución de un proceso, rellenando el tiempo de espera entre la acción del usuario y la respuesta de la aplicación.

Barra de progreso circular

```
<ProgressBar
    android:id="@+id/miProgreso"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    style="?android:attr/progressBarStyle"/>
```



Barra de progreso lineal

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

4. EVENTOS DEL USUARIO

En el apartado 3, cuando vimos los botones, hicimos una introducción a cómo se realizaría la gestión de los eventos de usuario. Ahora vamos a estudiarlo más en profundidad.

Como muchos otros entornos de desarrollo, **Android está basado en controladores de eventos** (*Events Handlers*). Es decir, se ejecuta un determinado código en respuesta a algún evento que ocurra en el sistema operativo. Normalmente, estos eventos se activan cuando el usuario interactúa con la aplicación.

En Android el programador puede capturar los eventos específicos del objeto Vista (View) con la que el usuario interactúa y ejecutar sentencias.

Por ejemplo, cuando el usuario toca con el dedo una Vista (podría ser un botón), el sistema operativo invoca el método *onTouchEvent()* de ese objeto. Para interceptar este método deberíamos extender la clase (creando una nueva clase heredada del botón) y reemplazar el código del método de la clase original. Sin embargo, extender cada objeto Vista para gestionar un evento no es práctico. Por esta razón, Android dispone en todas las clases View de una colección de interfaces con funciones callback que se pueden utilizar con mucha más facilidad. Estas interfaces, que se denominan **“Escuchadores de eventos”** (*Event listeners*), permiten controlar la interacción del usuario con la interfaz de usuario.

Esto no quiere decir que no podamos extender una clase Vista para crear una clase nueva que herede el comportamiento de la clase anterior y redefinir los eventos de la misma directamente.

4.1. Uso de los Event Listeners

Un *Event Listener* es una interfaz de la clase Vista (View) que contiene un único método de tipo callback. Android invoca estos métodos cuando la Vista detecta que el usuario está provocando un tipo concreto de interacción con este elemento de la interfaz de usuario.

Entre los más importantes, podemos destacar los siguientes métodos callback:



- ***onClick()***: de *View.OnClickListener*. Este método se invoca cuando el usuario toca un elemento con un dedo (modo contacto), hace clic con la bola de navegación (TrackBall) del dispositivo o presiona la tecla “Intro” estando en un objeto.
- ***onLongClick()***: de *View.OnLongClickListener*. Este método se invoca cuando el usuario toca y mantiene el dedo sobre un elemento (modo de contacto), hace clic sin soltar con la bola de navegación (TrackBall) del dispositivo o presiona la tecla “Intro” durante un segundo estando en un elemento.
- ***onFocusChange()***: de *View.OnFocusChangeListener*. Se invoca cuando el usuario mueve el cursor hacia una Vista o se aleja de ésta utilizando la bola de navegación (Trackball) o usando las teclas de navegación.
- ***onKey()***: de *View.OnKeyListener*. Se invoca cuando el usuario se centra en un elemento y presiona o libera una tecla del dispositivo.
- ***onTouch()***: de *View.OnTouchListener*. Se invoca cuando el usuario realiza una acción de tipo contacto con el dedo como presionar o soltar o cualquier gesto de movimiento en la pantalla dentro de la Vista.
- ***onCreateContextMenu()***: de *View.OnCreateContextMenuListener*. Se invoca cuando se crea un menú contextual como resultado de una “pulsación larga” sobre un elemento.

Los manejadores de eventos podemos gestionarlos de dos formas distintas:

Método 1: Definir el listener dentro de la Actividad

En el siguiente ejemplo, vemos cómo especificar los métodos de eventos sobre un EditText:



```
// Definimos el evento Change del EditText
texto.addTextChangedListener(new TextWatcher() {

    // Método que se lanza antes de cambiar el texto
    public void beforeTextChanged(CharSequence s, int start, int
count,
                                int after) {
        resultado1.setText("Texto antes de cambiar: "+s.toString());
    }

    // Método que se lanza cuando el texto cambia
    public void onTextChanged(CharSequence s, int start, int before,
int count) {
        resultado2.setText("Texto cambiado: "+s.toString());
    }

    // Método que se lanza cuando el texto cambia. La diferencia
    // con el método anterior es que la variable es modificable
    public void afterTextChanged(Editable) {
        // En este evento no hacemos nada
    }

}); // end onChange EditText
```

En el código anterior fíjate de qué manera se define el Listener de los cambios del texto de una Vista de tipo *EditText*. Dentro de este Listener establecemos los métodos que vamos a gestionar (escuchar): *beforeTextChanged*, *onTextChanged* y *afterTextChanged*.

Método 2: Trabajando con el layout

También es posible definir un método común en **toda la Actividad y asignarlo a las Vistas** en el fichero de diseño de la interfaz del usuario *res/layout/main.xml* de la siguiente manera:

Asigno el método en el layout (le pongo el nombre que quiera, en este caso se ha elegido el mismo que el evento, *onClick*).

```
<Button
    android:id="@+id/btnAceptar"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="@string/aceptar"
    android:onClick="onClick"/>
```



Defino el método en la actividad

```
public void onClick(View view){  
    // Comprobamos qué Vista (botón) ha invocado el método  
    switch (view.getId()){  
        case R.id.btnAceptar:  
            // Código  
            break;  
  
        case R.id.btnRechazar:  
            // Código  
            break;  
    }  
}
```

Fíjate en que la función *callback* **onclick()** no devuelve ningún resultado (void); sin embargo, otros métodos deben devolver un resultado lógico (boolean) para finalizar su ejecución. A continuación vemos en qué consiste cada evento:

- **onLongClick()**: este método devuelve “true” para indicar que se han llevado a cabo las operaciones necesarias para manejar el evento clic, por lo que ya no debe lanzarse cualquier otro método de tipo “clic”. En caso contrario, si el método devuelve el valor “false”, Android puede invocar a continuación otro método diferente de tipo “clic”.
- **onKey()**: este método devuelve “true” o “false” para avisar si se han llevado a cabo las operaciones necesarias para manejar el evento de teclado, por lo que ya no debe lanzarse cualquier otro método de tipo “teclado”.
- **onTouch()**: en este método ocurre como en los dos casos anteriores, según devuelva “true” o “false” para señalar si Android debe invocar los siguientes métodos.

5. ANEXO

5.1. Problemas con **clearCheck** y eventos de **RadioGroup**

Cuando queremos manipular los botones de radio de un *RadioGroup* con un manejador de tipo *onCheckedChange*, si hemos utilizado en otro manejador de eventos el método *clearCheck*, veremos que nuestro código falla. Esto es debido a que al ejecutar este evento se produce una llamada a *onCheckedChange*, y si dentro del manejador intentamos manipular el elemento llamado por este método, el programa fallará ya que no habrá ningún elemento que manipular (el id del método no devolverá ningún elemento). Para solucionar esto, tendré que hacer la siguiente comprobación antes de hacer ninguna manipulación.



```
final TextView etiqueta = findViewById(R.id.lbl);
final RadioGroup rg = findViewById(R.id.grupo1);
final Button reset = findViewById(R.id.btnreset);

rg.setOnCheckedChangeListener(new RadioGroup.OnCheckedChangeListener() {
    @Override
    public void onCheckedChanged(RadioGroup radioGroup, int id) {

        final RadioButton boton = findViewById(id);
        if (boton!=null) {
            etiqueta.setText(boton.getText());
        }

    }
});

reset.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        rg.clearCheck();
    }
});
```

5.2. Obtener los elementos seleccionados de un grupo de CheckBox

A diferencia de los botones de radio, los CheckBox no tienen un componente por defecto que los englobe y que nos permita saber cuáles están seleccionados. Para poder hacer esto de una manera cómoda, deberemos introducir todos los componentes checkbox que queramos evaluar dentro de una vista, y después recorrer los hijos de la vista comprobando si están seleccionados o no.

```
final GridLayout layoutCheck = findViewById(R.id.layoutCheck);
for (int i=0; i<layoutCheck.getChildCount(); i++){
    CheckBox micheck = (CheckBox) layoutCheck.getChildAt(i);
    if (micheck.isChecked()){
        // Código
    }
}
```