Clase 2

Curso Desarrollo en ANDROID V2.0

Emiliano Gonzalez

Pedro Coronel

Asunción – Paraguay

2013

Componentes de una aplicación Android

En el apartado anterior vimos la estructura de un proyecto Android y aprendimos dónde colocar cada uno de los elementos que componen una aplicación, tanto elementos de software como recursos gráficos o de datos. En éste nuevo post vamos a centrarnos específicamente en los primeros, es decir, veremos los distintos tipos de componentes de software con los que podremos construir una aplicación Android.

En Java o .NET estamos acostumbrados a manejar conceptos como ventana, control, eventos o servicios como los elementos básicos en la construcción de una aplicación. Pues bien, en Android vamos a disponer de esos mismos elementos básicos aunque con un pequeño cambio en la terminología y el enfoque. Repasemos los componentes principales que pueden formar parte de una aplicación Android [Por claridad, y para evitar confusiones al consultar documentación en inglés, intentaré traducir lo menos posible los nombres originales de los componentes].

Activity

Las actividades (activities) representan el componente principal de la interfaz gráfica de una aplicación Android. Se puede pensar en una actividad como el elemento análogo a una ventana en cualquier otro lenguaje visual.

View

Los objetos view son los componentes básicos con los que se construye la interfaz gráfica de la aplicación, análogo por ejemplo a los controles de Java o .NET. De inicio, Android pone a nuestra disposición una gran cantidad de controles básicos, como cuadros de texto, botones, listas desplegables o imágenes, aunque también existe la posibilidad de extender la funcionalidad de estos controles básicos o crear nuestros propios controles personalizados.

Service

Los servicios son componentes sin interfaz gráfica que se ejecutan en segundo plano. En concepto, son exactamente iguales a los servicios presentes en cualquier otro sistema operativo. Los servicios pueden realizar cualquier tipo de acciones, por ejemplo actualizar datos, lanzar notificaciones, o incluso mostrar elementos visuales (p.ej. activities) si se necesita en algún momento la interacción con del usuario.

Content Provider

Un content provider es el mecanismo que se ha definido en Android para compartir datos entre aplicaciones. Mediante estos componentes es posible compartir determinados datos de nuestra aplicación sin mostrar detalles sobre su almacenamiento interno, su estructura, o su implementación. De la misma forma, nuestra aplicación podrá acceder a los datos de otra a través de los content provider que se hayan definido.

Broadcast Receiver

Un broadcast receiver es un componente destinado a detectar y reaccionar ante determinados mensajes o eventos globales generados por el sistema (por ejemplo: “Batería baja”, “SMS recibido”, “Tarjeta SD insertada”, …) o por otras aplicaciones (cualquier aplicación

puede generar mensajes (intents, en terminología Android) broadcast, es decir, no dirigidos a una aplicación concreta sino a cualquiera que quiera escucharlo).

Widget

Los widgets son elementos visuales, normalmente interactivos, que pueden mostrarse en la pantalla principal (home screen) del dispositivo Android y recibir actualizaciones periódicas. Permiten mostrar información de la aplicación al usuario directamente sobre la pantalla principal.

Intent

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

En el siguiente apartado empezaremos ya a ver algo de código, analizando al detalle una aplicación sencilla.

Controles Básicos en Android

En los próximos apartados vamos a hacer un repaso de los diferentes controles que pone a nuestra disposición la plataforma de desarrollo de este sistema operativo. Empezaremos con los controles más básicos y seguiremos posteriormente con algunos algo más elaborados.

Tenemos 3 formas de utilizar controles y views en android

- Drag and Drop en el visor gráfico y manipularlos mediante el cuadro de propiedades
- Declararlos en el fichero xml del layout
- Declararlos en el archivo Java

Control Button [API]

Un control de tipo Button es el botón más básico que podemos utilizar. En el ejemplo siguiente definimos un botón con el texto “Púlsame” asignando su propiedad android:text. Además de esta propiedad podríamos utilizar muchas otras como el color de fondo (android:background), estilo de fuente (android:typeface), color de fuente (android:textcolor), tamaño de fuente (android:textSize), etc.

```
<Button android:id="@+id/BtnBoton1"
    android:text="Púlsame"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />
```

Control ToggleButton [API]

Un control de tipo ToggleButton es un tipo de botón que puede permanecer en dos estados, pulsado/no_pulsado. En este caso, en vez de definir un sólo texto para el control definiremos dos, dependiendo de su estado. Así, podremos asignar las propiedades android:textOn y android:textoOff para definir ambos textos.

Veamos un ejemplo a continuación.

```
<ToggleButton android:id="@+id/BtnBoton2"
    android:textOn="ON"
    android:textOff="OFF"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />
```

Control ImageButton [API]

En un control de tipo ImageButton podremos definir una imagen a mostrar en vez de un texto, para lo que deberemos asignar la propiedad android:src. Normalmente asignaremos esta propiedad con el descriptor de algún recurso que hayamos incluido en la carpeta /res/drawable. Así, por ejemplo, en nuestro caso hemos incluido una imagen llamada "ok.png" por lo que haremos referencia al recurso "@drawable/ok".

```
<ImageButton android:id="@+id/BtnBoton3"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:src="@drawable/ok" />
```

Eventos de un botón

Como podéis imaginar, aunque estos controles pueden lanzar muchos otros eventos, el más común de todos ellos y el que querremos capturar en la mayoría de las ocasiones es el evento onClick. Para definir la lógica de este evento tendremos que implementarla definiendo un nuevo objeto View.OnClickListener() y asociándolo al botón mediante el método setOnClickListener(). La forma más habitual de hacer esto es la siguiente:

```
final Button btnBoton1 = (Button)findViewById(R.id.BtnBoton1);
btnBoton1.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View arg0)
    {
        lblMensaje.setText("Botón 1 pulsado!");
    }
});
```

En el caso de un botón de tipo ToggleButton suele ser de utilidad conocer en qué estado ha quedado el botón tras ser pulsado, para lo que podemos utilizar su método isChecked(). En el siguiente ejemplo se comprueba el estado del botón tras ser pulsado y se realizan acciones distintas según el resultado.

```
final ToggleButton btnBoton2 =
(ToggleButton)findViewById(R.id.BtnBoton2);
btnBoton2.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View arg0)
    {
        if(btnBoton2.isChecked())
            lblMensaje.setText("Botón 2: ON");
        else
            lblMensaje.setText("Botón 2: OFF");
    }
});
```

Personalizar el aspecto un botón [y otros controles]

En la imagen anterior vimos el aspecto que presentan por defecto los tres tipos de botones disponibles. Pero, ¿y si quisiéramos personalizar su aspecto más allá de cambiar un poco el tipo o el color de la letra o el fondo?

Para cambiar la forma de un botón podríamos simplemente asignar una imagen a la propiedad android:background, pero esta solución no nos serviría de mucho porque siempre se mostraría la misma imagen incluso con el botón pulsado, dando poca sensación de elemento "clickable".

La solución perfecta pasaría por tanto por definir diferentes imágenes de fondo dependiendo del estado del botón. Pues bien, Android nos da total libertad para hacer esto mediante el uso de selectores. Un selector se define mediante un fichero XML localizado en la carpeta /res/drawable, y en él se pueden establecer los diferentes valores de una propiedad determinada de un control dependiendo de su estado.

Por ejemplo, si quisiéramos dar un aspecto plano a un botón ToggleButton, podríamos diseñar las imágenes necesarias para los estados "pulsado" (en el ejemplo toggle_on.png) y "no pulsado" (en el ejemplo toggle_off.png) y crear un selector como el siguiente:

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

    <item android:state_checked="false"
          android:drawable="@drawable/toggle_off" />
    <item android:state_checked="true"
          android:drawable="@drawable/toggle_on" />

</selector>
```

Este selector lo guardamos por ejemplo en un fichero llamado toggle_style.xml y lo colocamos como un recurso más en nuestra carpeta de recursos /res/drawable. Hecho esto, tan sólo bastaría hacer referencia a este nuevo recurso que hemos creado en la propiedad android:background del botón:

```
<ToggleButton android:id="@+id/BtnBoton4"
    android:textOn="ON"
    android:textOff="OFF"
    android:padding="10dip"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:background="@drawable/toggle_style"/>
```

En la siguiente imagen vemos el aspecto por defecto de un ToggleButton y cómo ha quedado nuestro ToggleButton personalizado.

Imágenes, etiquetas y cuadros de texto

En este apartado nos vamos a centrar en otros tres componentes básicos imprescindibles en nuestras aplicaciones: las imágenes (ImageView), las etiquetas (TextView) y por último los cuadros de texto (EditText).

Control ImageView [API]

El control ImageView permite mostrar imágenes en la aplicación. La propiedad más interesante es android:src, que permite indicar la imagen a mostrar. Nuevamente, lo normal será indicar como origen de la imagen el identificador de un recurso de nuestra carpeta /res/drawable, por ejemplo android:src="@drawable/unaimagen".

Además de esta propiedad, existen algunas otras útiles en algunas ocasiones como las destinadas a establecer el tamaño máximo que puede ocupar la imagen, android:maxWidth y android:maxHeight.

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

En la lógica de la aplicación, podríamos establecer la imagen mediante el método setImageResorce(…), pasándole el ID del recurso a utilizar como contenido de la imagen.

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

Control TextView [API]

El control TextView es otro de los clásicos en la programación de GUIs, las etiquetas de texto, y se utiliza para mostrar un determinado texto al usuario. Al igual que en el caso de los botones, el texto del control se establece mediante la propiedad android:text. A parte de esta propiedad, la naturaleza del control hace que las más interesantes sean las que establecen el formato del texto mostrado, que al igual que en el caso de los botones son las siguientes: android:background (color de fondo), android:textColor (color del texto),

android:textSize (tamaño de la fuente) y android:typeface (estilo del texto: negrita, cursiva, …).

```
<TextView android:id="@+id/LblEtiqueta"
android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:text="Escribe algo:"
android:background="#AA44FF"
android:typeface="monospace" />
```

De igual forma, también podemos manipular estas propiedades desde nuestro código. Como ejemplo, en el siguiente fragmento recuperamos el texto de una etiqueta con getText(), y posteriormente le concatenamos unos números, actualizamos su contenido mediante setText() y le cambiamos su color de fondo con setBackgroundColor().

```
final TextView lblEtiqueta = (TextView)findViewById(R.id.LblEtiqueta);
String texto = lblEtiqueta.getText().toString();
texto += "123";
lblEtiqueta.setText(texto);
```

Control EditText [API]

El control EditText es el componente de edición de texto que proporciona la plataforma Android. Permite la introducción y edición de texto por parte del usuario, por lo que en tiempo de diseño la propiedad más interesante a establecer, además de su posición/tamaño y formato, es el texto a mostrar, atributo android:text.

```
<EditText android:id="@+id/TxtTexto"
android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:layout_below="@id/LblEtiqueta" />
```

De igual forma, desde nuestro código podremos recuperar y establecer este texto mediante los métodos getText() y setText(nuevoTexto) respectivamente:

```
final EditText txtTexto = (EditText)findViewById(R.id.TxtTexto);
String texto = txtTexto.getText().toString();
txtTexto.setText("Hola mundo!");
```

Un detalle que puede haber pasado desapercibido. ¿Os habéis fijado en que hemos tenido que hacer un toString() sobre el resultado de getText()? La explicación para esto es que el método getText() no devuelve un String sino un objeto de tipo Editable, que a su vez implementa la interfaz Spannable. Y esto nos lleva a la característica más interesante el control EditText, y es que no sólo nos permite editar texto plano sino también texto enriquecido o con formato.

Control RadioButton [API]

Al igual que los controles checkbox, un radio button puede estar marcado o desmarcado, pero en este caso suelen utilizarse dentro de un grupo de opciones donde una, y sólo una, de ellas debe estar marcada obligatoriamente, es decir, que si se marca una de ellas se desmarcará

automáticamente la que estuviera activa anteriormente. En Android, un grupo de botones RadioButton se define mediante un elemento RadioGroup, que a su vez contendrá todos los elementos RadioButton necesarios. Veamos un ejemplo de cómo definir un grupo de dos controles RadioButton en nuestra interfaz:

```
<RadioGroup android:id="@+id/gruporb"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent" >

    <RadioButton android:id="@+id/radio1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Opción 1" />

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

</RadioGroup>
```

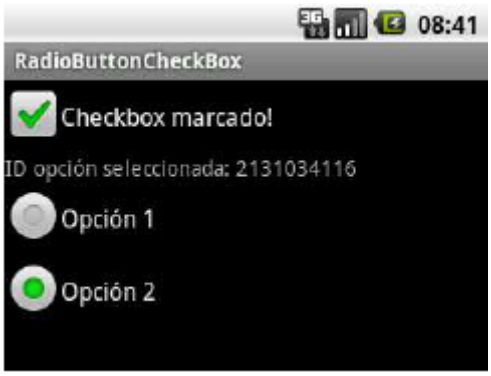
En primer lugar vemos cómo podemos definir el grupo de controles indicando su orientación (vertical u horizontal) al igual que ocurría por ejemplo con un LinearLayout. Tras esto, se añaden todos los objetos RadioButton necesarios indicando su ID mediante la propiedad android:id y su texto mediante android:text. Una vez definida la interfaz podremos manipular el control desde nuestro código java haciendo uso de los diferentes métodos del control RadioGroup, los más importantes: check(id) para marcar una opción determinada mediante su ID, clearCheck() para desmarcar todas las opciones, y getCheckedRadioButtonId() que como su nombre indica devolverá el ID de la opción marcada (o el valor -1 si no hay ninguna marcada). Veamos un ejemplo:

```
final RadioGroup rg = (RadioGroup)findViewById(R.id.gruporb);
rg.clearCheck();
rg.check(R.id.radio1);
int idSeleccionado = rg.getCheckedRadioButtonId();
```

En cuanto a los eventos lanzados, al igual que en el caso de los checkboxes, el más importante será el que informa de los cambios en el elemento seleccionado, llamado también en este caso onCheckedChange. Vemos cómo tratar este evento del objeto RadioGroup:

```
final RadioGroup rg = (RadioGroup)findViewById(R.id.gruporb);
  rg.setOnCheckedChangeListener(
    new RadioGroup.OnCheckedChangeListener() {
        public void onCheckedChanged(RadioGroup group, int checkedId)
{
            lblMensaje.setText("ID opcion seleccionada: " +
checkedid);
        }
});
```

Veamos finalmente una imagen del aspecto de estos dos nuevos tipos de controles básicos que hemos comentado en este apartado:



Interfaz de usuario en Android: Layouts

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

FrameLayout

Éste es el más simple de todos los layouts de Android. Un `FrameLayout` coloca todos sus controles hijos alineados con su esquina superior izquierda, de forma que cada control quedará oculto por el control siguiente (a menos que éste último tenga transparencia). Por ello, suele utilizarse para mostrar un único control en su interior, a modo de contenedor (*placeholder*) sencillo para un sólo elemento sustituible, por ejemplo una imagen.
Los componentes incluidos en un `FrameLayout` podrán establecer sus propiedades `android:layout_width` y `android:layout_height`, que podrán tomar los valores "`match_parent`" (para que el control hijo tome la dimensión de su layout contenedor) o "`wrap_content`" (para que el control hijo tome la dimensión de su contenido). **NOTA:** Si estás utilizando una versión de la API de Android inferior a la 8 (Android 2.2), en vez de "`match_parent`" deberás utilizar su equivalente "`fill_parent`".
Ejemplo:

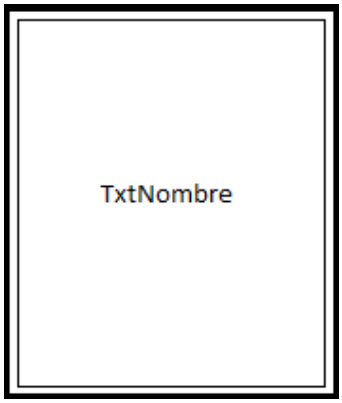
```
1    <FrameLayout
```

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

        <EditText android:id="@+id/TxtNombre"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:inputType="text" />

    </FrameLayout>
```

Con el código anterior conseguimos un layout tan sencillo como el siguiente:



LinearLayout

Este layout apila uno tras otro todos sus elementos hijos de forma horizontal o vertical según se establezca su propiedad android:orientation.

Al igual que en un `FrameLayout`, los elementos contenidos en un `LinearLayout` pueden establecer sus propiedades `android:layout_width` y `android:layout_height` para determinar sus dimensiones dentro del layout. Pero en el caso de un `LinearLayout`, tendremos otro parámetro con el que jugar, la propiedad `android:layout_weight`.

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

        <EditText android:id="@+id/TxtNombre"
            android:layout_width="match_parent"
            android:layout_height="match_parent" />

        <Button android:id="@+id/BtnAceptar"
            android:layout_width="wrap_content"
            android:layout_height="match_parent" />
```

```
</LinearLayout>
```

Esta propiedad nos va a permitir dar a los elementos contenidos en el layout unas dimensiones proporcionales entre ellas. Esto es más dificil de explicar que de comprender con un ejemplo. Si incluimos en un `LinearLayout` vertical dos cuadros de texto (`EditText`) y a uno de ellos le establecemos un `layout_weight="1"` y al otro un `layout_weight="2"` conseguiremos como efecto que toda la superficie del layout quede ocupada por los dos cuadros de texto y que además el segundo sea el doble (relación entre sus propiedades `weight`) de alto que el primero.

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

    <EditText android:id="@+id/TxtDato1"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:inputType="text"
        android:layout_weight="1" />

    <EditText android:id="@+id/TxtDato2"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:inputType="text"
        android:layout_weight="2" />

</LinearLayout>
```


Con el código anterior conseguiríamos un layout como el siguiente:

Así pues, a pesar de la simplicidad aparente de este layout resulta ser lo suficiente versátil como para sernos de utilidad en muchas ocasiones.

TableLayout

Un `TableLayout` permite distribuir sus elementos hijos de forma tabular, definiendo las filas y columnas necesarias, y la posición de cada componente dentro de la tabla.
La estructura de la tabla se define de forma similar a como se hace en HTML, es decir, indicando las filas que compondrán la tabla (objetos `TableRow`), y dentro de cada fila las columnas necesarias, con la salvedad de que no existe ningún objeto especial para definir una columna (algo así como un *TableColumn*) sino que directamente insertaremos los controles necesarios dentro del `TableRow` y cada componente insertado (que puede ser un control sencillo o incluso otro `ViewGroup`) corresponderá a una columna de la tabla.
De esta forma, el número final de filas de la tabla se corresponderá con el número de elementos TableRow insertados, y el número total de columnas quedará determinado por el número de componentes de la fila que más componentes contenga.
Por norma general, el ancho de cada columna se corresponderá con el ancho del mayor componente de dicha columna, pero existen una serie de propiedades que nos ayudarán a modificar este comportamiento:

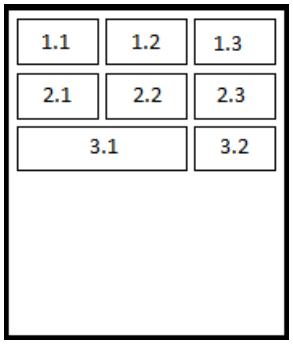
- `android:stretchColumns`. Indicará las columnas que pueden expandir para absorver el espacio libre dejado por las demás columnas a la derecha de la pantalla.
- `android:shrinkColumns`. Indicará las columnas que se pueden contraer para dejar espacio al resto de columnas que se puedan salir por la derecha de la pantalla.
- `android:collapseColumns`. Indicará las columnas de la tabla que se quieren ocultar completamente.

Todas estas propiedades del `TableLayout` pueden recibir una lista de índices de columnas separados por comas (ejemplo: `android:stretchColumns="1,2,3"`) o un asterisco para indicar que debe aplicar a todas las columnas (ejemplo: `android:stretchColumns="*"`).
Otra característica importante es la posibilidad de que una celda determinada pueda ocupar el espacio de varias columnas de la tabla (análogo al atributo `colspan` de HTML). Esto se indicará mediante la propiedad `android:layout_span` del componente concreto que deberá tomar dicho espacio.
Veamos un ejemplo con varios de estos elementos:

```
1  <TableLayout
2      xmlns:android="http://schemas.android.com/apk/res/android"
3      android:layout_width="match_parent"
4      android:layout_height="match_parent" >
5
6      <TableRow>
7          <TextView android:text="Celda 1.1" />
8          <TextView android:text="Celda 1.2" />
9          <TextView android:text="Celda 1.3" />
```

```
10        </TableRow>
11
12        <TableRow>
13            <TextView android:text="Celda 2.1" />
14            <TextView android:text="Celda 2.2" />
15            <TextView android:text="Celda 2.3" />
16        </TableRow>
17
18        <TableRow>
19            <TextView android:text="Celda 3.1"
20                    android:layout_span="2" />
21            <TextView android:text="Celda 3.2" />
22        </TableRow>
23    </TableLayout>
```

El layout resultante del código anterior sería el siguiente:



GridLayout

Este tipo de layout fue incluido a partir de la API 14 (Android 4.0) y sus características son similares al `TableLayout`, ya que se utiliza igualmente para distribuir los diferentes elementos de la interfaz de forma tabular, distribuidos en filas y columnas. La diferencia entre ellos estriba en la forma que tiene el `GridLayout` de colocar y distribuir sus elementos hijos en el espacio disponible. En este caso, a diferencia del `TableLayout` indicaremos el número de filas y columnas como propiedades del layout, mediante `android:rowCount` y`android:columnCount`. Con estos datos ya no es necesario ningún tipo de elemento para indicar las filas, como hacíamos con el elemento `TableRow` del `TableLayout`, sino que los diferentes elementos hijos se irán colocando ordenadamente por filas o columnas (dependiendo de la propiedad `android:orientation`) hasta completar el número de filas o columnas indicadas en los atributos anteriores. Adicionalmente, igual que en el caso anterior, también tendremos 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 forma de indicar de forma explícita la fila y columna que debe ocupar un determinado elemento hijo contenido en el `GridLayout`, y se consigue utilizando los

atributos`android:layout_row` y `android:layout_column`. De cualquier forma, salvo para configuraciones complejas del grid no suele ser necesario utilizar estas propiedades. Con todo esto en cuenta, para conseguir una distribución equivalente a la del ejemplo anterior del `TableLayout`, necesitaríamos escribir un código como el siguiente:

```
<GridLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:rowCount="2"
    android:columnCount="3"
    android:orientation="horizontal" >

        <TextView android:text="Celda 1.1" />
        <TextView android:text="Celda 1.2" />
        <TextView android:text="Celda 1.3" />

        <TextView android:text="Celda 2.1" />
        <TextView android:text="Celda 2.2" />
        <TextView android:text="Celda 2.3" />

        <TextView android:text="Celda 3.1"
               android:layout_columnSpan="2" />

        <TextView android:text="Celda 3.2" />

</GridLayout>
```

RelativeLayout

Este layout permite especificar la posición de cada elemento de forma relativa a su elemento padre o a cualquier otro elemento incluido en el propio layout. De esta forma, al incluir un nuevo elemento X podremos indicar por ejemplo que debe colocarse *debajo del elemento Y* y *alineado a la derecha del layout padre*. Veamos esto en el ejemplo siguiente:

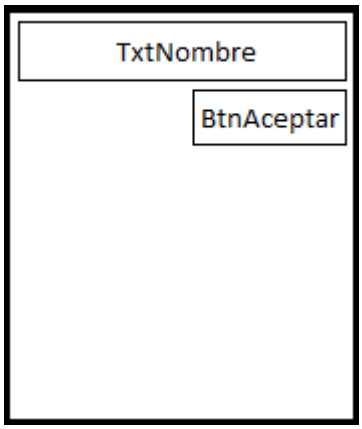
```
<RelativeLayout
        xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="match_parent"
        android:layout_height="match_parent" >

        <EditText android:id="@+id/TxtNombre"
              android:layout_width="match_parent"
              android:layout_height="wrap_content"
              android:inputType="text" />

        <Button android:id="@+id/BtnAceptar"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
```

```
14                  android:layout_below="@id/TxtNombre"
15                  android:layout_alignParentRight="true" />
16  </RelativeLayout>
```

En el ejemplo, el botón `BtnAceptar` se colocará debajo del cuadro de texto `TxtNombre`(`android:layout_below="@id/TxtNombre"`) y alineado a la derecha del layout padre (`android:layout_alignParentRight="true"`), Quedaría algo así:



Al igual que estas tres propiedades, en un RelativeLayout tendremos un sinfín de propiedades para colocar cada control justo donde queramos. Veamos las principales [creo que sus propios nombres explican perfectamente la función de cada una]:

Posición relativa a otro control:

- android:layout_above.
- android:layout_below.
- android:layout_toLeftOf.
- android:layout_toRightOf.
- android:layout_alignLeft.
- android:layout_alignRight.
- android:layout_alignTop.
- android:layout_alignBottom.
- android:layout_alignBaseline.

Posición relativa al layout padre:

- android:layout_alignParentLeft.
- android:layout_alignParentRight.
- android:layout_alignParentTop.
- android:layout_alignParentBottom.
- android:layout_centerHorizontal.
- android:layout_centerVertical.

- android:layout_centerInParent.

  Opciones de margen (también disponibles para el resto de layouts):

- android:layout_margin.
- android:layout_marginBottom.
- android:layout_marginTop.
- android:layout_marginLeft.
- android:layout_marginRight.

  Opciones de espaciado o padding (también disponibles para el resto de layouts):

- android:padding.
- android:paddingBottom.
- android:paddingTop.
- android:paddingLeft.
- android:paddingRight.