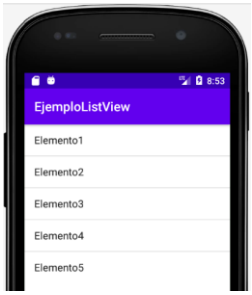
TEMA 5. LISTAS Y MENÚS

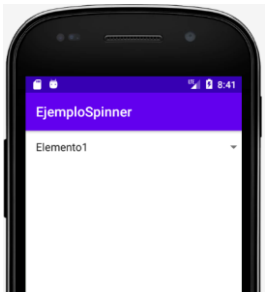
1. LOS LISTADOS

En este tema vamos a aprender a crear listas en Android y a manejar sus diferentes elementos, trabajando con tres de los objetos ViewGroup más frecuentes: ListView, GridView y Spinner. En la siguiente figura podemos ver la imagen que ofrecerían cada uno de ellos.



ListView

GridView



Spinner

1.1. ListView

Este objeto visualiza una lista deslizable verticalmente de varios elementos. Cada uno de ellos puede ser seleccionado sobre el propio control. En caso de disponer de más elementos de lo que es posible mostrar en pantalla, aparecerá una barra de scroll que permitirá acceder a los elementos no visibles. El código XML del objeto sobre el que construir el listado sería:

```
<ListView
    android:id="@+id/miLista"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"/>
```

Una vez creado el objeto en el layout, tenemos que manejar la lista desde código Java, primero para insertar los elementos y después para manipularla.

Los pasos que vamos a seguir para **insertar elementos en la lista** son:

1. Instanciar el objeto,
2. Declarar un array de valores (string), que serán los elementos de la lista,
3. Crear un adaptador que nos permita rellenar la lista y
4. Asociar el adaptador a la lista.

```
// Instancio el objeto
ListView listado = (ListView) findViewById(R.id.miLista);

//Declaro el array de valores
final String datos[] =]{"Elemento1","Elemento2","Elemento3",
     "Elemento4","Elemento5"};

// Creo el adaptador para poder rellenar la lista, pasándole los datos
ArrayAdapter<String> adaptador = new
     ArrayAdapter<String>(this,android.R.layout.simple_list_item_1,datos);

// Asocio el adaptador a la lista
listado.setAdapter(adaptador);
```

Un adaptador suele ser un objeto de la subclase BaseAdapter, que posibilita el convertir los datos en diferentes elementos de la lista. Android proporciona algunos adaptadores estándar, como ArrayAdapter y CursorAdapter. El primero permite manejar datos cuyo origen es un array, mientas que en el segundo caso los datos provendrán de una base de datos.

Para **obtener la información del elemento pulsado** se utiliza el manejador de eventos *setOnItemClickListener*, que podrá controlar el evento onItemClick, al que se le pasará como parámetro el adaptador y devolverá la posición del elemento tocado.

Obtener la posición del elemento pulsado

```
listado.setOnItemClickListener(new AdapterView.OnItemClickListener() {

    @Override
    public void onItemClick(AdapterView<?> adapterView, View view, int
posicion, long id) {

        // La tenemos en el parámetro posición
    }
});
```

Obtener el contenido del elemento pulsado

Una vez que se obtenga la posición, se puede obtener el contenido del elemento en sí mediante dos formas:

Obtener el contenido: Forma 1

Pedir al listado (adapterView en el ejemplo) que devuelva el elemento que se encuentra en la posición tocada.

```
String elemento = (String) adapterView.getItemAtPosition(posicion);
```

Insertado en el código anterior quedaría:

```
listado.setOnItemClickListener(new AdapterView.OnItemClickListener() {

    @Override
    public void onItemClick(AdapterView<?> listado, View view, int posicion,
long id) {

        String elemento = (String) listado.getItemAtPosition(posicion);

    }
});
```

Obtener el contenido: Forma 2

Obtenemos el adaptador del listado y se lo pedimos a este.

```
String elemento = (String) listado.getAdapter().getItem(posicion);
```

Insertado en el código quedaría:

```
listado.setOnItemClickListener(new AdapterView.OnItemClickListener() {

    @Override
    public void onItemClick(AdapterView<?> listado, View view, int posicion, long id)
{

        String elemento = (String) listado.getAdapter().getItem(posicion);

    }
});
```

Un método importante de los adaptadores es el getView, ya que lo más frecuente es que todos los elementos de un listado no cojan en pantalla (en la vista), por lo que el adaptador se ocupará de reutilizar filas y rellenar sus vistas asociadas (cada fila de la lista será una vista) con los nuevo datos para mostrar.

1.2. GridView

Muestra los datos en una rejilla bidimensional e incluye automáticamente un scroll para cuando los datos ocupen más tamaño que las posibilidades de la pantalla. Al igual que en el ListView, los datos provienen de un ListAdapter o incluso de un adaptador personalizado. Algunos atributos importantes son:

- *android:numColumns*: indica el número de columnas que deseamos establecer en el GridView.
- *android:columnWidth*: define el ancho de cada columna de una cuadrícula.
- *android:verticalSpacing*: separación vertical entre las filas del GridView.
- *android:horizontalSpacing*: separación horizontal entre las columnas del GridView.
- *android:strechMode*: define el modo en que se extenderán las columnas.
- *android:gravity*: define el posicionamiento del contenido en cada celda.

El código XML del objeto sobre el que construir el listado es similar al anterior, pero en este caso se añadirá el atributo del número de columnas que se desea que tenga la rejilla.

```
<GridView
    android:id="@+id/miGrid"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:numColumns="2"/>
```

Para manejar el GridView desde Java tras instanciar el objeto y declarar el array de valores (String), se debe asociar un adaptador a la lista.

```
GridView listado = (GridView) findViewById(R.id.miGrid);
final String datos[] = {"Elemento 1","Elemento 2","Elemento 3","Elemento
   4","Elemento 5"};
ArrayAdapter<String> adaptador = new ArrayAdapter<String>(this,
   android.R.layout.simple_list_item_1,datos);
listado.setAdapter(adaptador);
```

Para detectar el elemento tocado será exactamente igual que en el ListView:

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

    }
});
```

1.3. Spinner

Este objeto es una evolución del estudiado entre los objetos básicos del interfaz, tan solo que su aspecto puede ser mejorado mediante adaptadores personalizados. Tal y como el anterior, este muestra los datos en un listado desplegable de elementos, de los cuales el usuario puede seleccionar uno, que se mostraría al frente del Spinner.

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

El código Java para manipularlo es similar al de los dos casos anteriores.

```
Spinner listado = (Spinner) findViewById(R.id.miSpinner);
final String[] datos = {"Elemento1","Elemento2", "Elemento3",
        "Elemento4", "Elemento5"};
ArrayAdapter<String> adaptador = new ArrayAdapter<>(this,
        android.R.layout.simple_list_item_1, datos);
listado.setAdapter(adaptador);
```

Para detectar el elemento pulsado, usamos el escuchador ***setOnItemSelectedListener*** (fíjate que en los anteriores era *setOnItemClickListener*).

```
listado.setOnItemSelectedListener(new AdapterView.OnItemSelectedListener() {
    @Override
    public void onItemSelected(AdapterView<?> adapterView, View view, int i, long l)
{
        // Obtener el texto del elemento pulsado
    }

    @Override
    public void onNothingSelected(AdapterView<?> adapterView) {

    }
});
```

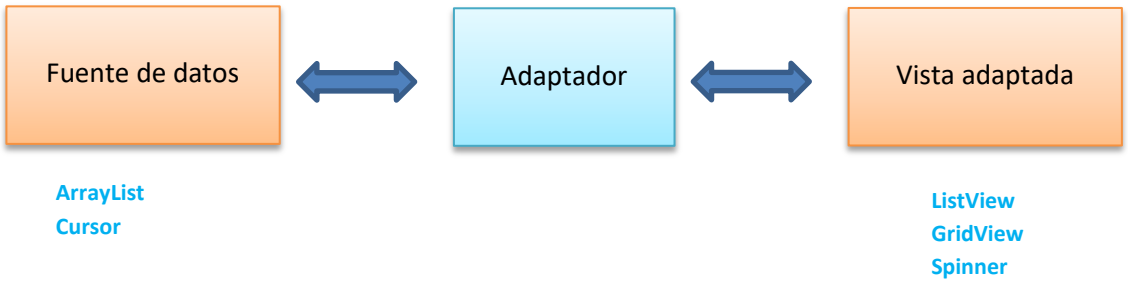
1.4. Adaptadores personalizados

Si bien la utilización del simple_list_item1 como adaptador es bastante sencilla, si se necesita incluir más de un elemento en el listado o se quisiera mejorar el aspecto visual del mismo (personalizando colores, fuentes o mostrando imágenes, enlaces web, etc.), se tendría que elaborar un adaptador personalizado.

1.4.1 Función de los adaptadores

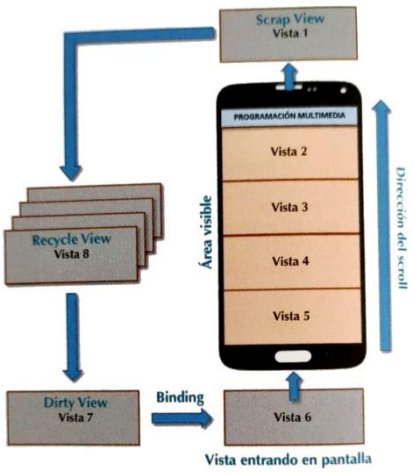
El entender la función de los adaptadores hará más fácil su programación e implementación en las aplicaciones. Para ello, se va a ver cómo funciona el reciclaje de vistas.

Cuando se conecta un ListView a un adaptador, este se ocupa de crear las instancias de las filas necesarias hasta que ListView tenga suficientes elementos para completar la altura que ocupa el listado en la pantalla, no siendo necesario preparar elementos adicionales en memoria.



Si se interacciona con el listado y se desplazan los elementos hacia arriba o hacia abajo, aquellos elementos que salen de la pantalla quedarán en memoria para un uso posterior, incorporando tantas nuevas filas como las que han sido guardadas en memoria.

Las vistas que salen, denominadas *ScrapViews*, pueden volver a utilizarse posteriormente sin tener que inflarse, solo deben actualizarse (reciclarse). Por tanto, el objeto de lista solo necesita mantener suficientes vistas en la memoria para completar el espacio asignado en el diseño, y algunas vistas reciclables adicionales.



1.4.2 Ejemplo de adaptador personalizado

Vistos estos conceptos básicos sobre adaptadores, se va a crear uno que gestione un ListView (útil también para GridView y Spinner), que mostrará dos líneas de texto con formato personalizado.

Primer paso: crear una clase Java que gestione y suministre los datos

El primer paso será realizar una clase Java que gestione y suministre los datos. Esta deberá tener tantos atributos como se vayan a poner en el listado. Para este ejemplo, podría servir la siguiente:

```
public class Datos {

    private String texto1;
    private String texto2;

    public Datos(String text1, String text2){
        texto1 = text1;
        texto2 = text2;
    }

    public String getTexto1(){
        return texto1;
    }

    public String getTexto2(){
        return texto2;
    }
}
```

Segundo paso: crear el layout para la nueva vista

A continuación, deberá crearse en la carpeta *res/layout* un fichero XML con el diseño que se quiere que tenga la vista de datos (cada elemento del listado). Por ejemplo, podríamos crear el siguiente layout (llamaremos al layout *elemento.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/miTexto1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textStyle="bold"
        android:textSize="18sp"/>

    <TextView
        android:id="@+id/miTexto2"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:textStyle="normal"
        android:textSize="12sp"/>

</LinearLayout>
```

Tercer paso: crear el adaptador

Por último, se creará el adaptador, para lo cual pueden ser utilizadas dos variantes: ArrayAdapter y BaseAdapter.

1.4.2.1 ArrayAdapter

Es el adaptador más sencillo. Convierte un ArrayList de objetos en vistas cargadas en un contenedor, que en este caso es un ListView. Este adaptador se ocupa de localizar los datos para la lista y convertirlos en un objeto para la vista.

Para ello es conveniente que se cree una clase independiente, que extenderá a ArrayAdapter, a cuyo constructor se le pasará el contexto (que suele ser la actividad principal) y la clase que se ha creado para gestionar el array de objetos con los datos que se van a mostrar.

Dentro del constructor se llamará a la superclase, pasándole como parámetros el contexto, el layout que da formato a cada elemento de la lista y los datos. Además, se pasará la referencia del objeto datos como parámetro para otros métodos (previamente los tendremos que haber declarado). El aspecto que tendría hasta ahora esta clase, a la que se ha llamado Adaptador, sería el siguiente:

```
public class Adaptador extends ArrayAdapter<Datos> {

    private Datos[] datos;

    public Adaptador(Context context, Datos[] datos){
        super(context,R.layout.elemento,datos);
        this.datos = datos;
    }
}
```

A continuación, y dentro de esta clase, se debe sobrescribir el método *getView* que se ocupará de generar y rellenar con los datos cada uno de los elementos del listado mostrado en la interfaz gráfica de cada elemento de la lista.

Este método será llamado cada vez que haya que mostrar un nuevo elemento de la lista y realizará el siguiente proceso. Lo primero que hará es "inflar" el layout que da formato a cada elemento (en este ejemplo, elemento.xml). Para esto, se instancia un objeto LayoutInflater, se localiza su contexto (con getContext) y se asocia la vista al elemento a "inflar". Por último, queda tan solo referenciar los objetos que aparecen en el XML (texto1 y texto2), que se rellenarán con los datos que se encuentran en la posición que se va a mostrar en el array.

```
@Override
public View getView(int position, View convertView, ViewGroup parent){

    LayoutInflater mostrado = LayoutInflater.from(getContext());
    View elemento = mostrado.inflate(R.layout.elemento,parent,false);

    TextView texto1 = (TextView) elemento.findViewById(R.id.miTexto1);
    texto1.setText(datos[position].getTexto1());

    TextView texto2 = (TextView) elemento.findViewById(R.id.miTexto2);
    texto2.setText(datos[position].getTexto2());

    return elemento;
}
```

Ya solo queda preparar la llamada desde el Java principal. Primero se prepara el array con la información para mostrar.

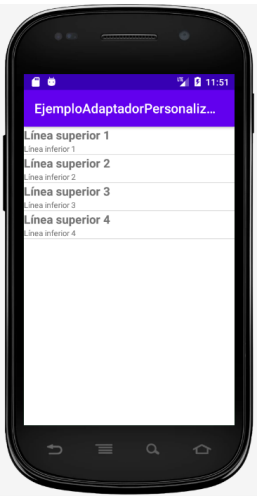
```
Datos[] datos = new Datos[]{
        new Datos("Línea superior 1", "Línea inferior 1"),
        new Datos("Línea superior 2", "Línea inferior 2"),
        new Datos("Línea superior 3", "Línea inferior 3"),
        new Datos("Línea superior 4", "Línea inferior 4")};
}
```

Como es fácil imaginar, no es operativo incluir en la clase principal fuentes de datos con mucha información, sino que se debe recurrir a otras alternativas, como pueden ser una clase independiente, fichero en memoria, base de datos.

A continuación, se incorpora el objeto ListView en el XML principal de la aplicación, se instancia y se crea el adaptador (que será una instancia del adaptador personalizado) y se asocia al ListView.

```
ListView listado = (ListView) findViewById(R.id.miLista);
Adaptador miAdaptador = new Adaptador(this,datos);
listado.setAdapter(miAdaptador);
```

El resultado sería el siguiente:



Si queremos añadir una cabecera al listado, haríamos lo siguiente:

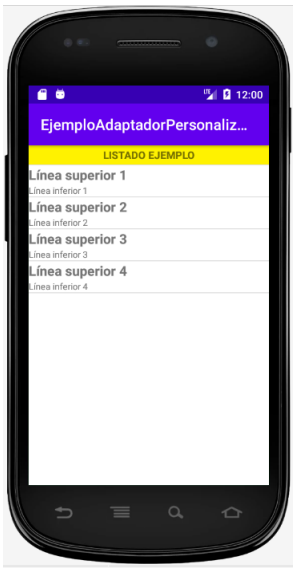
Crear un fichero XML con el aspecto que queremos que tenga la cabecera (por ejemplo cabecera.xml)

```
<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:layout_width="match_parent"
        android:layout_height="25dp"
        android:text="LISTADO EJEMPLO"
        android:textStyle="bold"
        android:background="#fff200"
        android:gravity="center"/>
</LinearLayout>
```

Añadir la cabecera a la lista

Para ello, se insertará en la cabecera utilizando dos líneas de código (antes de aplicar el adaptador al listado): una que crea la vista de la cabecera y gestiona su inflado, y otra que añade la cabecera a la lista.

El nuevo aspecto de la aplicación sería el siguiente:



Por último, nos queda detectar el elemento pulsado. Para ello, se utiliza un proceso similar al visto anteriormente. Se crea un escuchador al listado (*setOnItemClickListener*) para ese evento (*onItemClick*), que debe recibir cuatro parámetros, el control que contiene la lista (*AdapterView*), la vista del objeto pulsado (View), la posición del elemento pulsado (*position*) y el id del elemento pulsado.

```
listado.setOnItemClickListener(new AdapterView.OnItemClickListener() {
    @Override
    public void onItemClick(AdapterView<?> adaptador, View view, int position, long
id) {
        String seleccionado = ((Datos)
adaptador.getItemAtPosition(position)).getTexto1();
    }
});
```

1.4.2.2 Esquema de funcionamiento

1) Crear la clase Java que gestione y suministre los datos a mostrar
2) Crear el layout para la nueva vista
3) Crear el adaptador

   a. Crear una clase independiente que extenderá a *ArrayAdapter*. A su constructor se le pasará el contexto y la clase creada para gestionar el array.
   b. En su constructor se llamará a la superclase, pasándole el contexto, el layout creado en el paso 2, y los datos.
   c. Sobreescribimos el método getView
      i. Inflar el layout que da formato a los datos
      ii. Referenciamos los objetos del XML.
4) Preparar la llamada desde el Java principal
   a. Se prepara el array con la información para mostrar
   b. Se incorpora el objeto ListView en el XML principal de la aplicación, se instancia y se crea el adaptador (que será una instancia del adaptador personalizado) y se asocia al ListView.
5) Si se quiere añadir una cabecera al listado
   a. Creamos el layout con el aspecto que queremos en la cabecera.
   b. Insertamos la cabecera antes de aplicar el adaptador al listado
      i. Creamos la vista de la cabecera y la inflamos
      ii. Añadimos la cabecera a la lista
6) Detectar el elemento pulsado añadiendo el *listener* al listado.

1.4.2.3 Personalización con otros elementos

En las listas personalizadas no hay por qué añadir siempre texto, sino que se pueden incorporar otros elementos como direcciones Web, CheckBox, RadioButton, imágenes…Para ello, hay que realizar un diseño del XML que gestiona su vista con el formato adecuado a las necesidades, y luego incorporar estos objetos al Java que gestiona los datos.

Lógicamente, la fuente de datos (en el caso del ejemplo anterior el array) debe tener todos los elementos para mostrar. En caso de tratarse de imágenes, en el array debe aparecer la URL o recurso de memoria donde se encuentran.

1.4.2.4 BaseAdapter

Esta clase es muy utilizada en Android para crear adaptadores. La construcción es muy similar al de ArrayAdapter, ya que previamente se ha de diseñar el aspecto que se quiere que tenga cada elemento de la vista de datos, así como una clase Java que gestione los datos (pueden servir los utilizados en ArrayAdapter).

Este adaptador extenderá de BaseAdapter, que tiene una serie de métodos que es aconsejable conocer:

- int getCount()
- Object getItem(int position)

- long getItemId(int position)
- View getView(int position, View convertView, ViewGroup parent)

De ellos, *getCount*() devuelve el número total de elementos que se mostrarán en la lista, para lo que tiene en cuenta el tamaño del array. Cuando un elemento de la lista está preparado para mostrarse, es llamado el método *getView*, colaborando con la clase *LayoutInflater* para su mostrado en pantalla.

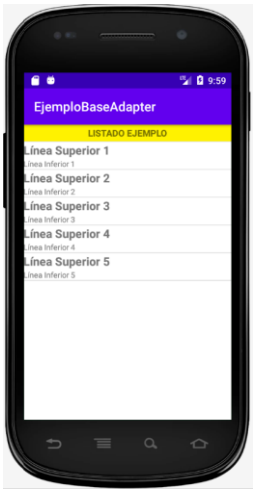
Para conocer el elemento que se encuentra en una posición específica dentro de la colección d datos, será necesario recurrir a *getItem*, mientras que *getItemId* devuelve el identificador de fila de una determinada posición de la lista. En casos simples, coincidirá con la posición.

Ahora el constructor del adaptador recibirá un *ArrayList* generado a partir del Java que gestione el suministro de datos (POJO). Y, al igual que con *ArrayAdapter*, será el método *getView* quien gestione el inflado del *ListView* con los datos aportados, obteniendo en cada caso los datos del array según la posición en la que se encuentre. El código final del adaptador quedaría de la siguiente forma:

```
public class Adaptador extends BaseAdapter {

    private ArrayList<Datos> datos;
    private Context contexto;

    public Adaptador(Context contexto, ArrayList<Datos> datos){
        super();
        this.contexto = contexto;
        this.datos = datos;
    }

    @Override
    public View getView(int posicion, View view, ViewGroup parent){

        LayoutInflater mostrado = LayoutInflater.from(contexto);
        View elemento = mostrado.inflate(R.layout.elemento, parent, false);
        TextView texto1 = (TextView) elemento.findViewById(R.id.miTexto1);
        texto1.setText(datos.get(posicion).getTexto1());
        TextView texto2 = (TextView) elemento.findViewById(R.id.miTexto2);
        texto1.setText(datos.get(posicion).getTexto2());

        return elemento;
    }

    @Override
    public int getCount(){
        return datos.size();
    }

    @Override
    public Object getItem(int posicion){
        return datos.get(posicion);
    }

    @Override
    public long getItemId(int posicion){
        return posicion;
    }
}
```

Además, se han de hacer modificaciones en el Java principal, ya que ahora se suministrarán los datos en un ArrayList, quedando esa parte del código del siguiente modo:

```
ArrayList<Datos> datos = new ArrayList<Datos>();
datos.add(new Datos("Línea Superior 1", "Línea Inferior 1"));
datos.add(new Datos("Línea Superior 2", "Línea Inferior 2"));
datos.add(new Datos("Línea Superior 3", "Línea Inferior 3"));
datos.add(new Datos("Línea Superior 4", "Línea Inferior 4"));
datos.add(new Datos("Línea Superior 5", "Línea Inferior 5"));
```



2. MENÚS

En Android me puedo encontrar con varios tipos de menús: de opciones, contextuales, etc.

En cualquiera de los casos, existen dos formas de crearlos: definiendo el menú en un fichero XML e "inflándolo" después, o creando el menú directamente mediante código Java.

Los menús de opciones se sitúan en la barra de acciones, que surgió con Android 3.0. Existen dos tipos de barras de acciones: la *ActionBar*, que surgió precisamente con Android 3.0 y que, hoy en día, está obsoleta, y la *ToolBar*, que apareció con Android 5.0 y, hoy en día, sigue vigente.

La ActionBar era un elemento que se integraba directamente, por lo que para crear un menú no teníamos que hacer ningún paso precio. A diferencia de esta, la *ToolBar* no se incrusta automáticamente, sino que tenemos que integrarla en el diseño insertando un elemento *ToolBar*. Si no hacemos este paso previo, no podremos visualizar el menú creado.

2.1. Barra de Acciones (*ToolBar*)

Para incluir una barra de acciones en nuestra app, lo primero que tenemos que hacer es insertar un widget ToolBar en nuestro layout.

```
<androidx.appcompat.widget.Toolbar
    android:id="@+id/toolbar"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:background="?attr/colorPrimary"
    android:elevation="4dp"
    android:theme="@style/ThemeOverlay.AppCompat.ActionBar"
    app:popupTheme="@style/ThemeOverlay.AppCompat.Light"/>
```

Con los dos últimos atributos podemos controlar el tema aplicado al *ToolBar* y el menú de *overflow*.

Además, tenemos que tener en cuenta que si estamos usando un *ConstraintLayout*, tenemos que indicar las siguientes restricciones:

```
app:layout_constraintBaseline_toTopOf="parent"
app:layout_constraintLeft_toLeftOf="parent"
```

Por último, en Java tendremos que añadir el siguiente código en el método *onCreate()*.

```
Toolbar toolbar = findViewById(R.id.toolbar);
setSupportActionBar(toolbar);
```

2.2. OptionsMenu

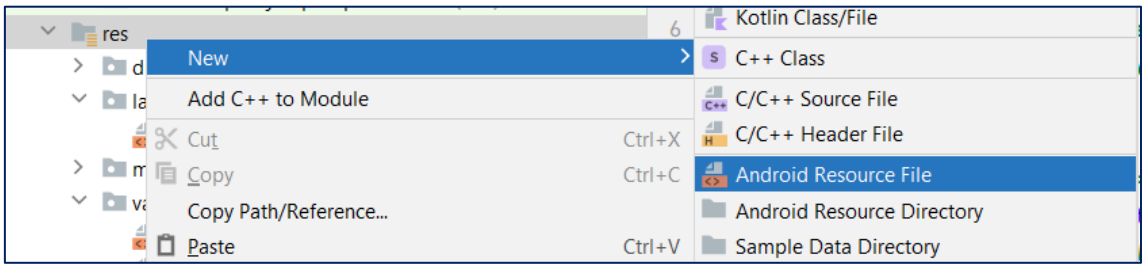
Se trata del menú más simple y clásico, que se despliega al pulsar una tecla u opción de pantalla. Podemos crearlo incorporando de dos formas: a partir de un fichero XML, o directamente desde Java.

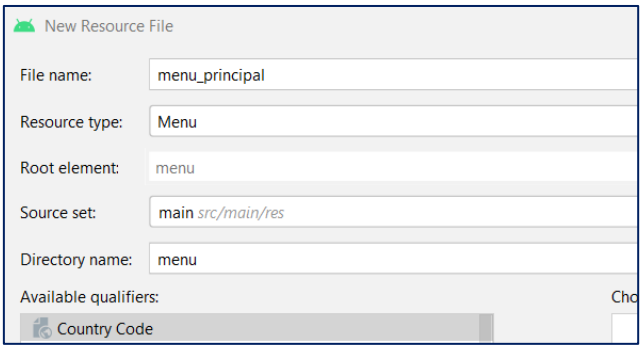
Método 1: fichero XML

Para ello, definiremos las opciones en un fichero XML, con el nombre deseado, ubicado en **res/menu**, con la siguiente estructura:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/MnOp1" android:title="Opción de menú A"></item>
    <item android:id="@+id/MnOp2" android:title="Opción de menú B"></item>
    <item android:id="@+id/MnOp3" android:title="Opción de menú C"></item>
</menu>
```

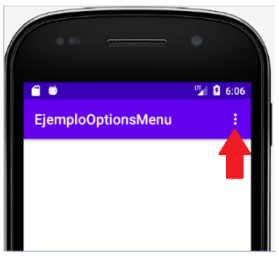
Para crear el fichero del menú, pulsamos con el botón derecho del ratón la carpeta *res* y elegimos *New>Android Resource File*, ponemos el nombre deseado al menú y en *Resource Type* elegimos *Menú*.

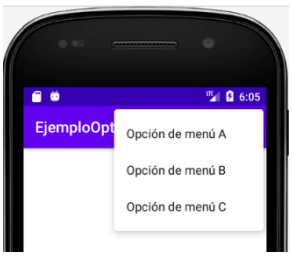




Una vez creada la estructura del menú, para mostrarlo ("inflarlo") se ha de utilizar en Java el siguiente método:

```
@Override
public boolean onCreateOptionsMenu(Menu menu){
    getMenuInflater().inflate(R.menu.menu_principal,menu);
    return true;
}
```





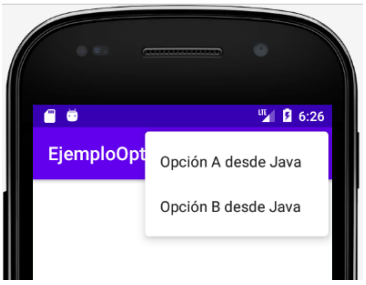
Por último, le asociaremos un escuchador que responda a alguno de los identificadores que se han asignado a cada opción de menú.

```
@Override
public boolean onOptionsItemSelected(MenuItem item){

    switch (item.getItemId()){
        case R.id.MnOp1:
            // Código para realizar
            return true;
        case R.id.MnOp2:
            // Código para realizar
            return true;
        case R.id.MnOp3:
            // Código para realizar
            return true;
        default:
            return super.onOptionsItemSelected(item);
    }
}
```

Método 2: crearlo desde Java

Para crearlo en su totalidad desde Java, se han de introducir los identificadores y luego añadir cada una de las opciones del menú con el método *onCreateOptionsMenu*.

```
private static final int MnOp1 = 1;
private static final int MnOp2 = 2;

---------------

@Override
public boolean onCreateOptionsMenu(Menu menu){
    menu.add(Menu.NONE, MnOp1, Menu.NONE, "Opción A desde Java");
    menu.add(Menu.NONE, MnOp2, Menu.NONE, "Opción B desde Java");
    return true;

}
```



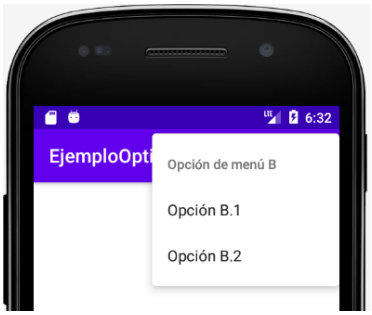
En su escuchador, los casos del switch serán 1 y 2, los valores que se le han dado a los identificadores de las opciones.

2.3. Submenú

Método 1: desde XML

Para crear subopciones dentro de un menú, se puede proceder igual que en el caso anterior, definiéndolo desde XML, para lo cual debe crearse un nuevo menú dentro de las etiquetas del ítem al que se quieren asociar opciones secundarias, como en el siguiente ejemplo:

```
<item android:id="@+id/MnOp2" android:title="Opción de menú B">
    <menu>
        <item android:id="@+id/MnOp2_1" android:title="Opción B.1"/>
        <item android:id="@+id/MnOp2_2" android:title="Opción B.2"/>
    </menu>
</item>
```



Método 2: desde Java

Para hacerlo desde Java, se crea una instancia de la clase subMenu y se añade a la opción de menú de la que dependa. Luego se incorpora cada una de las opciones de submenú.

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

    SubMenu smnu = menu.addSubMenu(Menu.NONE, MnOp1, Menu.NONE, "Opción A desde
Java");
    smnu.add(Menu.NONE, MnOp1_1, Menu.NONE, "Opción A.1");
    smnu.add(Menu.NONE, MnOp1_2, Menu.NONE, "Opción A.2");
    menu.add(Menu.NONE, MnOp2, Menu.NONE, "Opción B desde Java");
    return true;
}
```

Los escuchadores que responden a cada uno de los eventos de las opciones del submenú se construyen de igual forma que las de las opciones principales.

2.4. Menú Contextual

Son opciones de menú que aparecen en la interfaz de usuario cuando se realiza una pulsación larga sobre algún elemento de la pantalla. Para construirlo se debe asociar el menú contextual a algún objeto de la pantalla, frecuentemente a una etiqueta de texto.

Para identificar a los elementos afectados por dicha pulsación y asociarle su menú contextual, deberemos utilizar el método *registerForContextMenu*().

```
TextView etiqueta = (TextView) findViewById(R.id.lbl1);
registerForContextMenu(etiqueta);
```

El menú, y en su caso el submenú contextual, podremos definirlo de cualquiera de las dos formas vistas anteriormente: desde XML o desde Java.

El evento encargado de la construcción del menú es *onCreateContextMenu*, en el que se debe construir e inflar el menú contextual.

```
@Override
public void onCreateContextMenu(ContextMenu menu, View v, ContextMenu.ContextMenuInfo
        menuInfo){

    super.onCreateContextMenu(menu,v,menuInfo);
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.menu_principal, menu);
}
```

El escuchador para este menú se construye de manera similar a los anteriores, pero se sustituye *onOptionsItemSelected* por *onContextItemSelected*.

2.5. Menú contextual en listas

Puede ser útil hacer que aparezcan menús contextuales en alguno de los elementos de un listado para ofrecer, por ejemplo, opciones alternativas. Vamos a ver cómo se haría utilizando como ejemplo un elemento ListView.

```
<ListView
    android:id="@+id/listado"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"/>
```

Como ya hemos visto, el listado lo podemos crear desde XML o desde Java. Una vez creado el adaptador y asociado, se registrará la vista para que tenga un menú contextual.

```
private ListView listado;
-----------

listado = (ListView) findViewById(R.id.listado);
final String datos[] = new String[]{"OPCIÓN DE MENÚ A", "OPCIÓN DE MENÚ B","OPCIÓN DE
        MENÚ C", "OPCIÓN DE MENÚ D" };
ArrayAdapter<String> adaptador = new ArrayAdapter<String>(this,
        android.R.layout.simple_list_item_1,datos);
listado.setAdapter(adaptador);
registerForContextMenu(listado);
```

A continuación, se han de crear tantos ficheros XML de menú como menús contextuales distintos se desea que aparezcan.

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/ListaOpA1" android:title="OPCIÓN 1 DE MENÚ DE LISTA A"/>
    <item android:id="@+id/ListaOpA2" android:title="OPCIÓN 2 DE MENÚ DE LISTA A"/>
</menu>
```

Por último, inflaremos el menú contextual. Para saber cuál de ellos debemos inflar, obtendremos del adaptador el elemento que se ha pulsado.

```
@Override
public void onCreateContextMenu(ContextMenu menu, View v, ContextMenu.ContextMenuInfo
menuInfo){
    super.onCreateContextMenu(menu, v, menuInfo);
    MenuInflater inflater = getMenuInflater();
    AdapterView.AdapterContextMenuInfo info = (AdapterView.AdapterContextMenuInfo)
menuInfo;
    menu.setHeaderTitle(listado.getAdapter().getItem(info.position).toString());
    switch (info.position){
        case 0:
            inflater.inflate(R.menu.menu_c1,menu);
            return;
        case 1:
            inflater.inflate(R.menu.menu_c2,menu);
            return;
    }
}
```

Al igual que en el menú contextual anterior, el escuchador que se va a utilizar se construye a partir de *onContextItemSelected*.

```
@Override
public boolean onContextItemSelected(MenuItem item){

    switch (item.getItemId()){
        case R.id.ListaOpA1:
            // Código
            return true;
        case R.id.ListaOpA2:
            // Código
            return true;
        default:
            return super.onContextItemSelected(item);
    }
}
```

Si quisiéramos obtener información del elemento de la lista sobre el que hemos desplegado el menú contextual, lo haríamos con el objeto *info* obtenido con *getMenuInfo()*. El atributo *position* de este elemento, nos devolverá la posición del elemento del ListView seleccionado.

```
@Override
public boolean onContextItemSelected(MenuItem item){
    AdapterView.AdapterContextMenuInfo info = (AdapterView.AdapterContextMenuInfo)
item.getMenuInfo();

    // Obtengo la posición del elemento del ListView
    int pulsado = info.position;

    // Obtengo el elemento del ListView
    String texto = listado.getItemAtPosition(pulsado).toString();

    …

}
```

3. ANEXO: RECYCLERVIEW

Aparte de las clases vistas anteriormente para ver listados, disponemos de otra clase más que también hereda de ViewGroup: RecyclerView. Esta clase, que permite mostrar grandes colecciones o conjuntos de datos, solo se dedica a reciclar vistas, así que otros componentes serán los responsables de acceder a los datos y mostrarlos. Los más importantes en los que se va a apoyar son los siguientes:

- RecyclerView.Adapter
- RecyclerView.ViewHolder
- LayoutManager
- ItemDecoration
- ItemAnimator

Vamos a ir viendo qué hacen estos elementos:

RecyclerView.Adapter

Al igual que pasaba en los listados vistos anteriormente, necesitamos un adaptador para suministrar los datos. En este caso, RecyclerView.Adapter. La peculiaridad en esta ocasión es que este tipo de adaptador utilizará internamente el patrón ViewHolder que dotará de una mayor eficiencia al control, y de ahí la necesidad del segundo componente de la lista anterior, RecyclerView.ViewHolder

RecyclerView.ViewHolder

Por resumirlo de alguna forma, un view holder se encargará de contener y gestionar las vistas o controles asociados a cada elemento individual de la lista. El control RecyclerView se encargará de crear tantos view holder como sea necesario para mostrar los elementos de la lista que se ven en pantalla y los gestionará eficientemente de forma que no tenga que crear nuevos objetos para mostrar más elementos de la lista al hacer scroll, sino que tratará de "reciclar" aquellos que ya no sirven por estar asociados a otros elementos de la lista que ya han salido de la pantalla.

LayoutManager

Anteriormente, cuando decidíamos utilizar un ListView ya sabíamos que nuestros datos se representarían de forma lineal con la posibilidad de hacer scroll en un sentido u otro, y en el caso de elegir un GridView la representación sería tabular. Una vista de tipo RecyclerView, por el contrario, no determina por sí sola la forma en que se van a mostrar en pantalla los elementos de nuestra colección, sino que va a delegar esa tarea a

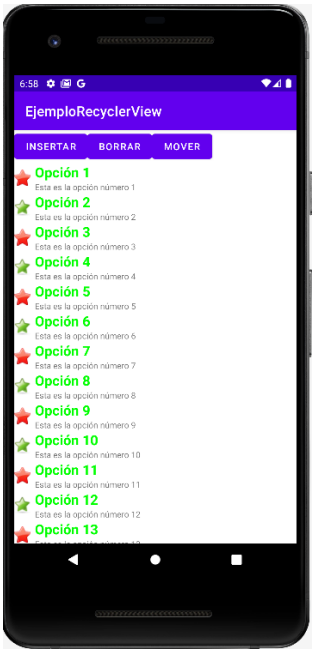
otro componente llamado LayoutManager, que también tendremos que crear y asociar al RecyclerView para su correcto funcionamiento. El SDK incorpora de serie algunos LayoutManager para las representaciones más habituales de los datos: lista vertical u horizontal (LinearLayoutManager), tabla tradicional (GridLayoutManager) y tabla apilada o de celdas no alineadas (StaggeredGridLayoutManager). Por tanto, siempre que optemos por alguna de estas distribuciones de elementos no tendremos que crear nuestro propio LayoutManager personalizado, aunque por supuesto nada nos impide hacerlo, y ahí uno de los puntos fuertes de este componente: su flexibilidad.

ItemDecoration e ItemAnimator

Los dos últimos componentes de la lista se encargarán de definir cómo se representarán algunos aspectos visuales concretos de nuestra colección de datos (más allá de la distribución definida por el LayoutManager), por ejemplo marcadores o separadores de elementos, y de cómo se animarán los elementos al realizarse determinadas acciones sobre la colección, por ejemplo al añadir o eliminar elementos.

No siempre será obligatorio implementar todos estos componentes para hacer uso de un RecyclerView. Lo más habitual será implementar el Adapter y el ViewHolder, utilizar alguno de los LayoutManager predefinidos, y sólo en caso de necesidad crear los ItemDecoration e ItemAnimator necesarios para dar un toque de personalización especial a nuestra aplicación.

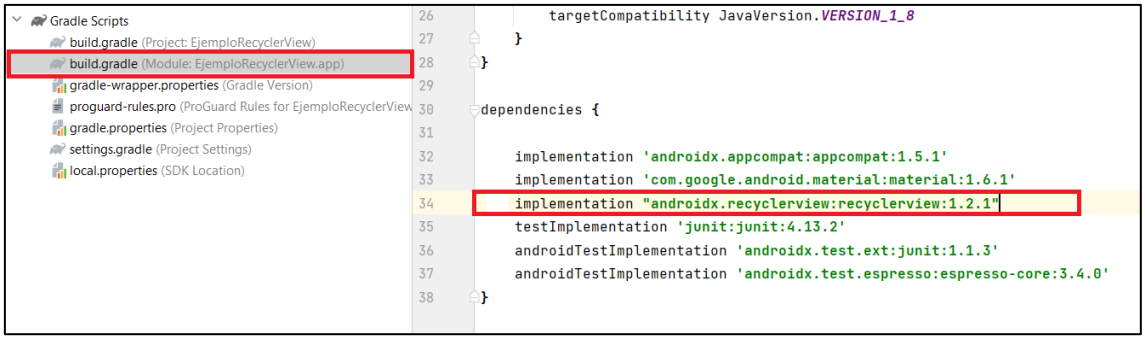
A continuación, vamos a describir los **pasos para el uso de RecyclerView.** Para que sea más didáctico, vamos a hacerlo mientras creamos la siguiente aplicación, donde mostramos un listado donde cada opción contiene dos textos y un icono.



Paso 1. Agregar dependencia al gradle

Añadiremos a la sección de dependencias del fichero build.gradle del módulo principal la referencia a la librería de soporte androidx.recyclerview, lo que nos permitirá el uso del componente RecyclerView en la aplicación:

```
implementation "androidx.recyclerview:recyclerview:1.2.1"
```



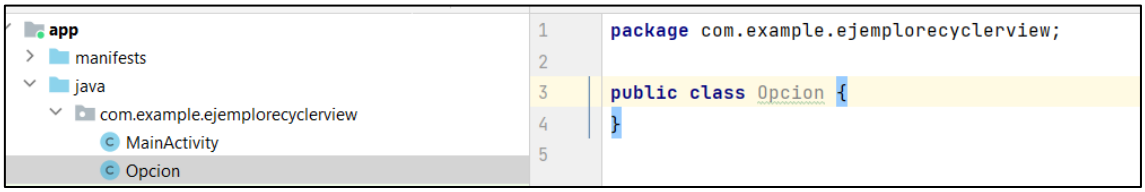
Paso 2. Codificación de la vista (XML)

Para ello, tenemos que incluir el siguiente widget:

```
<androidx.recyclerview.widget.RecyclerView
    android:id="@+id/recView"
    android:layout_width="match_parent"
    android:layout_height="match_parent"/>
```

Paso 3. Creamos una clase que encapsulará los datos

Esta clase la usaremos para obtener cada uno de los elementos que compongan el listado. Por lo tanto, en ella codificaré los getters y los setters necesarios.



En nuestro ejemplo, los elementos del listado van a tener dos textos y un icono.

```
public class Opcion {

    private String titulo;
    private String subtitulo;
    private int icono;

    public Opcion(String titulo, String subtitulo, int icono){
        this.setTitulo(titulo);
        this.setSubtitulo(subtitulo);
        this.setIcono(icono);
    }

    // Definimos los getters y los setters
    public String getTitulo(){
        return titulo;
    }

    public void setTitulo(String titulo){
        this.titulo = titulo;
    }

    public String getSubtitulo(){
        return subtitulo;
    }

    public void setSubtitulo(String subtitulo){
        this.subtitulo = subtitulo;
    }

    public int getIcono(){
        return icono;
    }

    public void setIcono(int icono){
        this.icono = icono;
    }
}
```

Paso 4. Creamos el layout con la estructura visual que queramos

En nuestro caso, tenemos que añadir dos etiquetas de texto y una imagen para el icono.

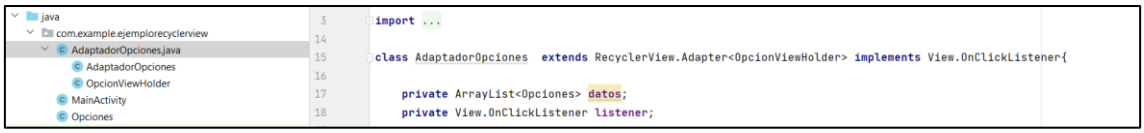
```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal">

    <ImageView
        android:layout_width="wrap_content"
        android:layout_height="match_parent"
        android:id="@+id/icono"
        android:layout_marginRight="6dip"/>

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="vertical">

        <TextView
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:id="@+id/LblTitulo"
            android:textColor="#00FF00"
            android:textSize="20dp"
            android:textStyle="bold"/>

        <TextView
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:id="@+id/LblSubtitulo"
            android:textSize="12dp"
            android:textStyle="normal"/>

    </LinearLayout>

</LinearLayout>
```

Paso 5. Creamos el adaptador

A continuación escribiremos nuestro adaptador. Este adaptador deberá extender a la clase RecyclerView.Adapter, de la cual tendremos que sobrescribir principalmente tres métodos:

- **onCreateViewHolder()**: encargado de crear los nuevos objetos ViewHolder necesarios para los elementos de la colección.
- **onBindViewHolder()**: encargado de actualizar los datos de un ViewHolder ya existente.
- **onItemCount()**: indica el número de elementos de la colección de datos.

Para nuestra aplicación de ejemplo utilizaré como fuente de datos un simple array de objetos de tipo Opcion.



Como ya hemos comentado, la clase *RecyclerView.Adapter* nos obligará de cierta forma a hacer uso del patrón view holder. Por tanto, para poder seguir con la implementación del adaptador lo primero que **definiremos será el ViewHolder necesario** para nuestro caso de ejemplo. Lo definiremos como clase interna a nuestro adaptador, extendiendo de la clase *RecyclerView.ViewHolder*, y será bastante sencillo, tan sólo tendremos que incluir como atributos las referencias a los controles del layout de un elemento de la lista (en nuestro caso los dos *TextView* y el icono) e inicializarlas en el constructor utilizando como siempre el método *findViewById()* sobre la vista recibida como parámetro.

```
class OpcionViewHolder extends RecyclerView.ViewHolder{
    private TextView titulo;
    private TextView subtitulo;
    private ImageView icono;

    public OpcionViewHolder(View itemView){
        super(itemView);

        titulo = (TextView) itemView.findViewById(R.id.LblTitulo);
        subtitulo = (TextView) itemView.findViewById(R.id.LblSubtitulo);
        icono = (ImageView) itemView.findViewById(R.id.icono);
    }

    public void bindOpcion(Opciones t){
        titulo.setText(t.getTitulo());
        subtitulo.setText(t.getSubtitulo());
        icono.setImageResource(t.getIcono());
    }

}
```

Finalizado nuestro *ViewHolder* podemos ya seguir con la implementación del adaptador sobrescribiendo los métodos indicados. En el método *onCreateViewHolder()* nos limitaremos a inflar (construir) una vista a partir del layout correspondiente a los elementos de la lista, y crear y devolver un nuevo *ViewHolder* llamando al constructor de nuestra clase pasándole dicha vista como parámetro.

```
class AdaptadorOpciones  extends RecyclerView.Adapter<OpcionViewHolder>
implements View.OnClickListener{

    private ArrayList<Opciones> datos;
    private View.OnClickListener listener;

    AdaptadorOpciones(ArrayList<Opciones> datos){
        this.datos = datos;
    }

    @Override
    public OpcionViewHolder onCreateViewHolder(ViewGroup viewGroup, int
viewType){
        View itemView =
LayoutInflater.from(viewGroup.getContext()).inflate(R.layout.opcion,viewGrou
p,false);
        itemView.setOnClickListener(this);
        OpcionViewHolder ovh = new OpcionViewHolder(itemView);
        return ovh;
    }

    @Override
    public void onBindViewHolder(OpcionViewHolder viewHolder, int pos){
        Opciones item = datos.get(pos);
        viewHolder.bindOpcion(item);
    }

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

    public void setOnClickListener(View.OnClickListener listener){
        this.listener = listener;
    }

    @Override
    public void onClick(View view){
        if (listener != null)
            listener.onClick(view);
    }
```

Los dos métodos restantes son aún más sencillos. En *onBindViewHolder()* tan sólo tendremos que recuperar el objeto Opcion correspondiente a la posición recibida como parámetro y asignar sus datos sobre el ViewHolder también recibido como parámetro. Por su parte, getItemCount() tan sólo devolverá el tamaño del array de datos.

Paso 6. Asignamos el adaptador a la clase principal

Con esto tendríamos finalizado el adaptador, por lo que ya podríamos asignarlo al RecyclerView en nuestra actividad principal. Tan sólo tendremos que crear nuestro adaptador personalizado AdaptadorOpciones pasándole como parámetro la lista de datos y asignarlo al control RecyclerView mediante su propiedad adapter.

```
public class MainActivity extends AppCompatActivity {

    private ArrayList<Opciones> datos; // Guardamos las 32 opciones del listado
    private RecyclerView recView;  // Lista tipo RecyclerView
    private Button btnInsertar;
    private Button btnBorrar;
    private Button btnMover;

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

        // Buscamos el layout del RecyclerView
        recView = (RecyclerView) findViewById(R.id.RecView);

        // Indicamos que el tamaño del RecyclerView no cambia
        recView.setHasFixedSize(true);

        datos = new ArrayList<Opciones>();

        // Definimos las 32 opciones
        for (int i=1; i<=32; i++){
            if (i%2 == 0){
                datos.add(new Opciones("Opción " + i,"Esta es la opción número " + i,
R.drawable.star1));
            }
            else{
                datos.add(new Opciones("Opción " + i, "Esta es la opción número " + i,
R.drawable.star2));
            }
        }

        // Creamos el adaptador que se usa para mostrar las opciones del listado
        final AdaptadorOpciones adaptador = new AdaptadorOpciones(datos);

        // Definimos el evento onClick
        adaptador.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                Toast.makeText(getBaseContext(), "Has hecho clic en " +
datos.get(recView.getChildAdapterPosition(view)).getTitulo(),Toast.LENGTH_SHORT).show();
            }
        });

        recView.setAdapter(adaptador);
        recView.setLayoutManager(new
LinearLayoutManager(this,LinearLayoutManager.VERTICAL,false));
        recView.setItemAnimator(new DefaultItemAnimator());

        //Definimos los eventos onClic
        btnInsertar = (Button) findViewById(R.id.BtnInsertar);
        btnInsertar.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                datos.add(1,new Opciones("Nueva opción","Subtítulo nueva
opción",R.drawable.star1));
                adaptador.notifyItemInserted(1);
            }
        });

        btnBorrar = (Button) findViewById(R.id.BtnBorrar);
        btnBorrar.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                if (datos.size()<2) return;

                datos.remove(1);
                adaptador.notifyItemRemoved(1);
            }
        });

        btnMover = (Button) findViewById(R.id.BtnMover);
        btnMover.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
```

```
                Opciones aux = datos.get(1);
                datos.set(1, datos.get(2));
                datos.set(2,aux);

                adaptador.notifyItemMoved(1,2);
            }
        });
    }
}
```

4. ANEXO 2: SPINNER PERSONALIZADO

Como comentamos en el apartado 1.4, al igual que podemos crear adaptadores personalizados con *ListView* y *GridView*, también podemos hacerlo con el elemento *Spinner*. Sin embargo, si lo creamos siguiendo los mismos pasos que con los elementos anteriores, veremos que al desplegar el *Spinner* el programa fallará y se cerrará. Esto es debido a que el método *getView* que implementamos en el adaptador tan solo carga la vista del elemento que aparece seleccionado en el *Spinner*, pero no infla el resto de los elementos cuando desplegamos la lista. Para hacer esto, tenemos que implementar el método ***getDropDownView***, que es el que se encargará de rellenar los elementos de la lista (cosa que hace el método *getView* en los *ListView* y los *GridView*).

El código para ello sería el siguiente (el ejemplo se basa en una fuente de datos que solo incluye una imagen):

```
@Override
public View getDropDownView(int position, @Nullable View convertView, @NonNull
ViewGroup parent) {

    if (convertView==null){
        convertView =
LayoutInflater.from(getContext()).inflate(R.layout.elemento,parent,false);
    }

    ImageView icono = convertView.findViewById(R.id.icono);
    Datos elemento = getItem(position);

    if (elemento!=null){
        icono.setImageResource(elemento.getIcono());
    }

    return convertView;
}
```