Controles de selección (V): RecyclerView

Con la llegada de Android 5.0, Google ha incorporado al SDK de Android un nuevo componente que viene a mejorar a los clásicos `ListView` y `GridView` existentes desde hace tiempo. Probablemente no sólo venga a mejorarlos sino también a sustituirlos en la mayoría de ocasiones ya que aporta flexibilidad de sobra para suplir la funcionalidad de ambos controles e ir incluso más allá.

Este nuevo control se llama `RecyclerView` y, al igual que conseguimos con `ListView` y`GridView`, nos va a permitir mostrar en pantalla colecciones grandes de datos. Pero lo va a hacer de una forma algo distinta a la que estábamos habituados con los controles anteriores. Y es que `RecyclerView` no va a hacer "casi nada" por sí mismo, sino que se va a sustentar sobre otros componentes complementarios para determinar cómo acceder a los datos y cómo mostrarlos. Los más importantes serán los siguientes:

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

Los dos primeros componentes deberíais ya reconocerlos si habéis revisado los capítulos anteriores del curso dedicados a los controles de selección. De igual forma que hemos hecho con los componentes anteriores, un `RecyclerView` se apoyará también en un*adaptador* para trabajar con nuestros datos, en este caso un adaptador que herede de la clase `RecyclerView.Adapter`. La peculiaridad en esta ocasión es que este tipo de adaptador nos "obligará" en cierta medida

a utilizar el patrón *View Holder* que ya describimos en el segundo artículo sobre ListView, y de ahí la necesidad del segundo componente de la lista anterior, `RecyclerView.ViewHolder`.

Los tres siguientes sí son una novedad, siendo el más importante el primero de ellos, el `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. Por suerte, el SDK incorpora de serie tres `LayoutManager` para las tres representaciones más habituales: 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 del nuevo componente: su flexibilidad.

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.

Como hemos comentado, 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.

En el resto del artículo voy a intentar describir de forma más o menos detallada cómo crear una aplicación de ejemplo análoga a la creada en el capítulo sobre ListView, pero utilizando en esta ocasión el nuevo componente `RecyclerView`.

Vamos a empezar por la parte más sencilla, crearemos un nuevo proyecto en Android Studio y añadiremos a la sección de dependencias del fichero *build.gradle* del módulo principal la referencia a la librería de soporte *recyclerview-v7*, lo que nos permitirá el uso del componente `RecyclerView` en la aplicación:

```
1
2    dependencies {
3        compile 'com.android.support:appcompat-v7:21.0.3'
4        compile 'com.android.support:recyclerview-v7:+'
5    }
```

Tras esto ya podremos añadir un nuevo `RecyclerView` al layout de nuestra actividad principal:

```
1    <android.support.v7.widget.RecyclerView
2        android:id="@+id/RecView"
3        android:layout_width="match_parent"
4        android:layout_height="match_parent" />
```

Como siguiente paso 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.

Un par de anotaciones antes de seguir. En primer lugar, para nuestra aplicación de ejemplo utilizaré como fuente de datos una lista (`ArrayList`) de objetos de tipo `Titular`, la misma clase que ya utilizamos en el capítulo sobre el control ListView. En segundo lugar, la representación de los datos que queremos lograr será también la misma que conseguimos en aquel ejemplo, es decir, cada elemento de la lista consistirá en dos líneas sencillas de texto donde mostraremos el título y subtítulo de cada objeto `Titular`. Por tanto, tanto la clase `Titular`, como el layout asociado a los elementos de la lista (fichero*listitem_titular.xml*) podéis copiarlos directamente de dicho proyecto.

Como ya dijimos, la clase `RecyclerView.Adapter` nos obligará a hacer uso del patrón *ViewHolder* (recomiendo leer el artículo donde explicamos la utilidad de este patrón). Por tanto, para poder seguir con la implementación del adaptador debemos definir primero 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`) e inicializarlas en el constructor utilizando como siempre el método `findViewById()` sobre la vista recibida como parámetro. Por comodidad añadiremos también un método auxiliar, que llamaremos

`bindTitular()`, que se encargue de asignar los contenidos de los dos cuadros de texto a partir de un objeto `Titular` cuando nos haga falta.

```
public class AdaptadorTitulares
        extends RecyclerView.Adapter<AdaptadorTitulares.TitularesViewHolder> {

    //...

    public static class TitularesViewHolder
            extends RecyclerView.ViewHolder {

        private TextView txtTitulo;
        private TextView txtSubtitulo;

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

            txtTitulo = (TextView)itemView.findViewById(R.id.LblTitulo);
            txtSubtitulo = (TextView)itemView.findViewById(R.id.LblSubTitulo);
        }

        public void bindTitular(Titular t) {
            txtTitulo.setText(t.getTitulo());
            txtSubtitulo.setText(t.getSubtitulo());
        }
    }

    //...
}
```

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

Los dos métodos restantes son aún más sencillos. En `onBindViewHolder()` tan sólo tendremos que recuperar el objeto `Titular` 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 de la lista de datos.

```
1
2
3     public class AdaptadorTitulares
4             extends RecyclerView.Adapter<AdaptadorTitulares.TitularesViewHolder> {
5
6         private ArrayList<Titular> datos;
7
8         //...
9
10        public AdaptadorTitulares(ArrayList<Titular> datos) {
11            this.datos = datos;
12        }
13
14        @Override
15        public TitularesViewHolder onCreateViewHolder(ViewGroup viewGroup, int viewT
16
17            View itemView = LayoutInflater.from(viewGroup.getContext())
18                    .inflate(R.layout.listitem_titular, viewGroup, false);
19
20            TitularesViewHolder tvh = new TitularesViewHolder(itemView);
21
22            return tvh;
23        }
24
25        @Override
26        public void onBindViewHolder(TitularesViewHolder viewHolder, int pos) {
27            Titular item = datos.get(pos);
28
29            viewHolder.bindTitular(item);
30        }
31
32        @Override
33        public int getItemCount() {
34            return datos.size();
35        }
36
37        //...
38    }
```

Con esto tendríamos finalizado el adaptador, por lo que ya podríamos asignarlo al `RecyclerView` en nuestra actividad principal. Esto es tan sencillo como lo era en el caso de `ListView`/`GridView`, tan sólo tendremos que crear nuestro adaptador personalizado

`AdaptadorTitulares` pasándole como parámetro la lista de datos y asignarlo al control `RecyclerView` mediante `setAdapter()`.

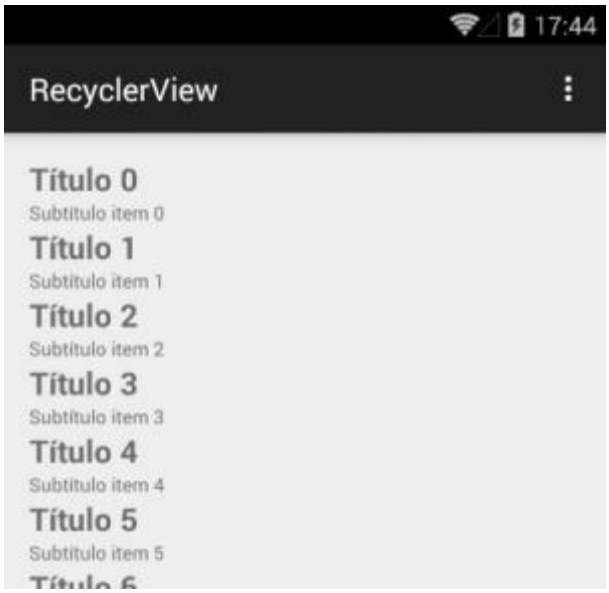
```
public class MainActivity extends ActionBarActivity {

    private RecyclerView recView;

    private ArrayList<Titular> datos;

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

        //inicialización de la lista de datos de ejemplo
        datos = new ArrayList<Titular>();
        for(int i=0; i<50; i++)
            datos.add(new Titular("Título " + i, "Subtítulo item " + i));

        //Inicialización RecyclerView
        recView = (RecyclerView) findViewById(R.id.RecView);
        recView.setHasFixedSize(true);

        final AdaptadorTitulares adaptador = new AdaptadorTitulares(datos);

        recView.setAdapter(adaptador);

        //...
    }

    //...
}
```

Algunos comentarios más sobre el código anterior. He aprovechado el método `onCreate()` para inicializar la lista de datos de ejemplo (atributo `datos`) con 50 titulares con un texto tipo. Tras obtener la referencia al `RecyclerView` he incluido también una llamada a `setHasFixedSize()`. Aunque esto no es obligatorio, sí es conveniente hacerlo cuando tengamos certeza de que el tamaño de nuestro `RecyclerView` no va a variar (por ejemplo debido a cambios en el contenido del adaptador), ya que esto permitirá aplicar determinadas optimizaciones sobre el control.

El siguiente paso obligatorio será asociar al `RecyclerView` un `LayoutManager` determinado, para determinar la forma en la que se distribuirán los datos en pantalla. Como ya dijimos, si nuestra intención es mostrar los datos en forma de lista o tabla (al estilo de los antiguos `ListView` o `GridView`) no tendremos que implementar nuestro propio `LayoutManager` (una tarea nada sencilla), ya que el SDK proporciona varias clases predefinidas para los tipos más habituales. En nuestro caso particular queremos mostrar los datos en forma de lista con desplazamiento vertical. Para ello tenemos disponible la clase `LinearLayoutManager`, por lo que tan sólo tendremos que instanciar un objeto de dicha clase indicando en el constructor la orientación del desplazamiento (`LinearLayoutManager.VERTICAL` o`LinearLayoutManager.HORIZONTAL`) y lo asignaremos al `RecyclerView` mediante el método `setLayoutManager()`. Esto lo haremos justo después del código anterior, tras la asignación del adaptador:

```
1    //...
2
3    recView.setAdapter(adaptador);
4
5    recView.setLayoutManager(
6        new LinearLayoutManager(this,LinearLayoutManager.VERTICAL,false));
7
8    //...
```

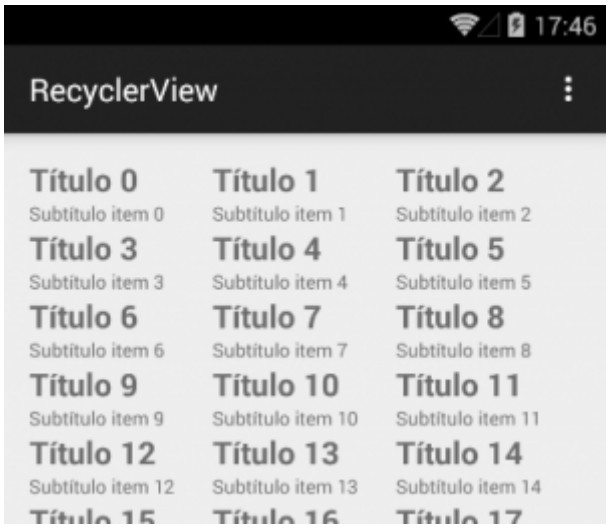
Llegados aquí ya sería posible ejecutar la aplicación de ejemplo para ver cómo quedan nuestros datos en pantalla, ya que los dos componentes que nos faltan por comentar (`ItemDecoration` e `ItemAnimator`) no son obligatorios para el funcionamiento básico del control `RecyclerView`.



Lo bueno de todo lo que llevamos hasta el momento, es que si cambiáramos de idea y quisiéramos mostrar los datos de forma tabular tan sólo tendríamos que cambiar la asignación del `LayoutManager` anterior y utilizar un `GridLayoutManager`, al que pasaremos como parámetro el número de columnas a mostrar.

```
1    //...
2
3    recView.setAdapter(adaptador);
4
5    recView.setLayoutManager(new GridLayoutManager(this,3));
6
7    //...
```

Con este simple cambio la aplicación quedaría con el siguiente aspecto:



El siguiente paso que nos podemos plantear es cómo responder a los eventos que se produzcan sobre el `RecyclerView`, como opción más habitual el evento *click* sobre un elemento de la lista. Para sorpresa de todos la clase `RecyclerView` no tiene incluye un evento `onItemClick()` como ocurría en el caso de `ListView`. Una vez más, `RecyclerView` delegará también esta tarea a otro componente, en este caso a la propia vista que conforma cada elemento de la colección.

Será por tanto dicha vista a la que tendremos que asociar el código a ejecutar como respuesta al evento click. Esto podemos hacerlo de diferentes formas, yo tan sólo mostraré una de ellas. En nuestro caso aprovecharemos la creación de cada nuevo `ViewHolder` para asignar a su vista asociada el evento `onClick`. Adicionalmente, para poder hacer esto desde fuera del adaptador, incluiremos el listener correspondiente como atributo del adaptador, y dentro de éste nos limitaremos a asignar el evento a la vista del nuevo `ViewHolder` y a lanzarlo cuando sea necesario desde el método `onClick()`. Creo que es más fácil verlo sobre el código:

```
1     public class AdaptadorTitulares
2             extends RecyclerView.Adapter<AdaptadorTitulares.TitularesViewHolder>
3             implements View.OnClickListener {
```

```
4
5           //...
6           private View.OnClickListener listener;
7
8           @Override
9           public TitularesViewHolder onCreateViewHolder(ViewGroup viewGroup, int viewT
10              View itemView = LayoutInflater.from(viewGroup.getContext())
11                      .inflate(R.layout.listitem_titular, viewGroup, false);
12
13              itemView.setOnClickListener(this);
14
15              TitularesViewHolder tvh = new TitularesViewHolder(itemView);
16
17              return tvh;
18          }
19
20          //...
21
22          public void setOnClickListener(View.OnClickListener listener) {
23              this.listener = listener;
24          }
25
26          @Override
27          public void onClick(View view) {
28              if(listener != null)
29                  listener.onClick(view);
30          }
31      }
```

Como vemos, nuestro adaptador implementará la interfaz `OnClickListener`, declarará un listener (el que podremos asignar posteriormente desde fuera del adaptador) como atributo de la clase, en el momento de crear el nuevo `ViewHolder` asociará el evento a la vista, y por último implementará el evento `onClick`, que se limitará a lanzar el mismo evento sobre el listener externo. El método adicional `setOnClickListener()` nos serivirá para asociar el listener *real* a nuestro adaptador en el momento de crearlo. Veamos cómo quedaría nuestra actividad principal con este cambio:

```
1       public class MainActivity extends ActionBarActivity {
2
3           //...
4
5           @Override
6           protected void onCreate(Bundle savedInstanceState) {
```

```
            //...

            recView = (RecyclerView) findViewById(R.id.RecView);
            recView.setHasFixedSize(true);

            final AdaptadorTitulares adaptador = new AdaptadorTitulares(datos);

            adaptador.setOnClickListener(new View.OnClickListener() {
                @Override
                public void onClick(View v) {
                    Log.i("DemoRecView", "Pulsado el elemento " +
                            recView.getChildAdapterPosition(v));
                }
            });

            recView.setAdapter(adaptador);

            recView.setLayoutManager(new LinearLayoutManager(this,
                                    LinearLayoutManager.VERTICAL,false));

            //...
        }
    }
```

En este caso al pulsar sobre un elemento de la lista tan sólo escribiremos un mensaje de log indicando la posición en la lista del elemento pulsado, lo que conseguimos llamando al método `getChildPosition()` del `RecyclerView`.

Con esto ya podríamos volver a ejecutar la aplicación de ejemplo y comprobar si todo funciona correctamente según lo definido al pulsar sobre los elementos de la lista.

Por último vamos a describir brevemente los dos componentes restantes relacionados con `RecyclerView`.

En primer lugar nos ocuparemos de `ItemDecoration`. Los `ItemDecoration` nos servirán para personalizar el aspecto de un `RecyclerView` más allá de la distribución de los elementos en pantalla. El ejemplo típico de esto son los separadores o divisores de una lista. `RecyclerView` no tiene ninguna propiedad `divider` como

en el caso del `ListView`, por lo que dicha funcionalidad debemos suplirla con un `ItemDecoration`.

Crear un `ItemDecoration` personalizado no es una tarea demasiado fácil, o al menos no inmediata, por lo que por ahora no nos detendremos mucho en ello. Para el caso de los separadores de lista podemos encontrar en un ejemplo del propio SDK de Android un ejemplo de `ItemDecoration` que lo implementa. La clase en cuestión se llama `DividerItemDecoration` y la incluyo en el proyecto de ejemplo de este apartado (tenéis el enlace a github al final del artículo). Si estudiamos un poco el código veremos que tendremos que extender de la clase `RecyclerView.ItemDecoration` e implementar sus métodos `getItemOffsets()` y `onDraw()`/`onDrawOver()`. El primero de ellos se encargará de definir los límites del elemento de la lista y el segundo de dibujar el elemento de personalización en sí. En el fondo no es complicado, aunque sí lleva su trabajo construirlo desde cero.

Para utilizar este componente deberemos simplemente crear el objeto y asociarlo a nuestro `RecyclerView` mediante `addItemDecoration()` en nuestra actividad principal:

```
public class MainActivity extends ActionBarActivity {

    //...

    @Override
    protected void onCreate(Bundle savedInstanceState) {

        //...

        recView = (RecyclerView) findViewById(R.id.RecView);
        recView.setHasFixedSize(true);

        final AdaptadorTitulares adaptador = new AdaptadorTitulares(datos);

        adaptador.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
  Log.i("DemoRecView", "Pulsado el elemento " + recView.getChildPosition(v));
```

```
            }
        });

        recView.setAdapter(adaptador);

        recView.setLayoutManager(
            new LinearLayoutManager(this,LinearLayoutManager.VERTICAL,false));

        recView.addItemDecoration(
                new DividerItemDecoration(this,DividerItemDecoration.VERTICAL_LI

        //...
    }
}
```

Como nota adicional indicar que podremos añadir a un `RecyclerView` tantos`ItemDecoration` como queramos, que se aplicarán en el mismo orden que se hayan añadido con `addItemDecoration()`.

Por último hablemos muy brevemente de `ItemAnimator`. Un componente `ItemAnimator` nos permitirá definir las animaciones que mostrará nuestro `RecyclerView` al realizar las acciones más comunes sobre un elemento (añadir, eliminar, mover, modificar). Este componente tampoco es sencillo de implementar, pero por suerte el SDK también proporciona una implementación por defecto que puede servirnos en la mayoría de ocasiones, aunque por supuesto podremos personalizar creando nuestro propio `ItemAnimator`. Esta implementación por defecto se llama `DefaultItemAnimator` y podemos asignarla al `RecyclerView` mediante el método `setItemAnimator()`.

```
public class MainActivity extends ActionBarActivity {

    //...

    @Override
    protected void onCreate(Bundle savedInstanceState) {

```

```
        //...

        recView = (RecyclerView) findViewById(R.id.RecView);
        recView.setHasFixedSize(true);

        final AdaptadorTitulares adaptador = new AdaptadorTitulares(datos);

        adaptador.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Log.i("DemoRecView", "Pulsado el elemento " +
 recView.getChildAdapterPosition(v));
            }
        });

        recView.setAdapter(adaptador);

        recView.setLayoutManager(
            new LinearLayoutManager(this,LinearLayoutManager.VERTICAL,false));

        recView.addItemDecoration(
                new DividerItemDecoration(this,DividerItemDecoration.VERTICAL_LI

        recView.setItemAnimator(new DefaultItemAnimator());

        //..
    }
}
```

Para probar su funcionamiento vamos a añadir a nuestro layout principal tres botones con las opciones de añadir, eliminar y mover un elemento de la lista (concretamente el segundo elemento de la lista). La implementación de estos botones es bastante sencilla y la incluiremos también en el `onCreate()` de la actividad principal:

```
btnInsertar = (Button)findViewById(R.id.BtnInsertar);

btnInsertar.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        datos.add(1, new Titular("Nuevo titular", "Subtitulo nuevo titular"));
        adaptador.notifyItemInserted(1);
    }
});

btnEliminar = (Button)findViewById(R.id.BtnEliminar);

btnEliminar.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
```

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

    btnMover = (Button)findViewById(R.id.BtnMover);

    btnMover.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {

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

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

Lo más interesante a destacar del código anterior son los métodos de notificación de cambios del adaptador. Tras realizar cada acción sobre los datos debemos llamar a su método de notificación correspondiente de forma que pueda refrescarse correctamente el control y se ejecute la animación correspondiente. Así, tras añadir un elemento llamaremos a `notifyItemInserted()` con la posición del nuevo elemento, al eliminar un dato llamaremos a `notifyItemRemoved()`, al actualizar un dato a `notifyItemChanged()` y al moverlo de lugar a `notifyItemMoved()`.

Ejecutemos por última vez la aplicación de ejemplo para revisar que las animaciones por defecto funcionan también según lo esperado.

Puedes consultar y/o descargar el código completo de los ejemplos desarrollados en este artículo accediendo a la página del curso en GitHub.