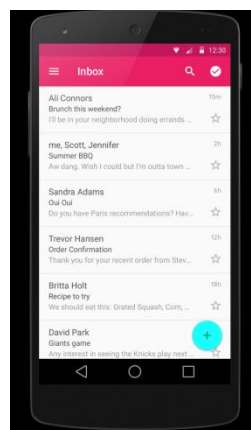




# RecyclerView





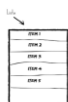
## Contenido

Patrón de diseño ViewHolder .....	3
RecyclerView .....	5
P_49_Recycler_01: Lista básica.....	6
Layouts .....	6
Modificando content_main.....	6
Layout ítem.....	6
Código clases .....	7
Clase para el modelo de datos: Item.java .....	7
Clase MainActivity.java .....	7
LayoutManager .....	8
ItemDecoration .....	11
Responder a eventos.....	12
Recycler_02: Cambiando los ítems (ItemAnimator) .....	13
Cambios en layout.....	13
Responder a eventos desde la actividad.....	13
Práctica propuesta .....	14



## Patrón de diseño ViewHolder

Uno de los elementos básicos de cualquier aplicación son las listas. Si nos fijamos, están prácticamente en cada pantalla y realmente son la mejor, si no la única, manera de mostrar una gran cantidad de datos de forma ordenada y clara.



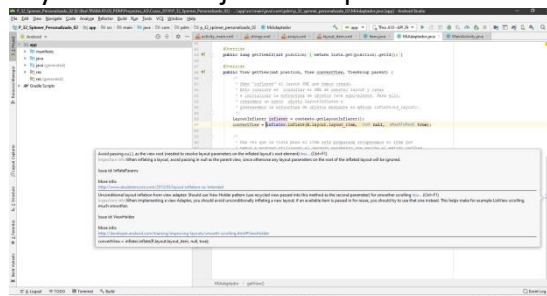
En Android las listas se suelen crear mediante Adapters a los que añadimos la información y, si queremos personalizar la lista mínimamente, también el diseño con el que se mostrarán.

Recuerda el Proyecto P\_32\_Spinner\_Personalizado\_02 del fichero Ejercicios\_08\_Controles\_de\_selección.

En nuestro Adapter personalizado teníamos entre otros el método `getView()` encargado de generar los layouts de cada ítem de la siguiente manera:

```
public class MiAdaptador extends BaseAdapter {
    ...
    @Override
    public View getView(int position, View convertView, ViewGroup parent) {
        LayoutInflater inflater = contexto.getLayoutInflater();
        convertView = inflater.inflate(R.layout.layout_item, null, true);
        Item item = lista.get(position);
        TextView textView = convertView.findViewById(R.id.textView2);
        textView.setText(item.getApi());
        ImageView imageView = convertView.findViewById(R.id.imageView);
        imageView.setImageDrawable(item.getImagen());
        return convertView;
    }
}
```

Para el ejercicio vale perfectamente, porque tenemos una lista con pocos elementos aunque en sus avisos AS ya nos aconseja usar el patrón de diseño View Holder:



Este diseño se complica cuando queremos cargar listas muy largas, ya que nuestros dispositivos pueden alcanzar fácilmente ciertos límites de procesamiento y de memoria si no utilizamos ciertos patrones y buenas prácticas.

Por qué?

Centrándonos en la definición del método `getView()` vimos que la forma normal de proceder consistía en primer lugar en “inflar” nuestro layout XML personalizado para crear todos los objetos correspondientes (con la estructura descrita en el XML) y posteriormente acceder a dichos objetos para modificar sus propiedades. Sin embargo, hay que tener en cuenta que esto se hace todas y cada una de las veces que se necesita mostrar un elemento de la lista en pantalla, se haya mostrado ya o no con anterioridad, ya que Android no “guarda” los elementos de la lista que desaparecen de pantalla (por ejemplo al hacer scroll sobre la lista). El efecto de esto es obvio, dependiendo del tamaño de la lista y sobre todo de la complejidad del layout que hayamos definido esto puede suponer la creación y destrucción de cantidades ingentes de objetos (que puede que ni siquiera nos sean necesarios), es decir, que la acción de inflar un layout XML puede ser bastante costosa, lo que podría aumentar mucho, y sin necesidad, el uso de CPU, de memoria y de batería.



Uno de los patrones que mejoran este comportamiento es el patrón ViewHolder, con el que podemos conseguir un procesamiento un 15% más rápido de las listas mejorando así la fluidez y, por consiguiente, la experiencia de usuario.

Copia el citado proyecto, renombra la copia como P\_46\_Spinner\_Personalizado\_ViewHolder y trabaja sobre ella.

Para aliviar el citado problema, Android nos propone un método que permite reutilizar algún layout que ya hayamos inflado con anterioridad y que ya no nos haga falta por algún motivo, por ejemplo porque el elemento correspondiente de la lista ha desaparecido de la pantalla al hacer scroll. De esta forma evitamos todo el trabajo de crear y estructurar todos los objetos asociados al layout, por lo que tan sólo nos quedaría obtener la referencia a ellos mediante `findViewById()` y modificar sus propiedades.

¿Pero cómo podemos reutilizar estos layouts "ya generados con anterioridad"? Pues es sencillo, siempre que exista algún layout que pueda ser reutilizado éste se va a recibir a través del parámetro view del método `getView()`. De esta forma, en los casos en que éste no sea null podremos obviar el trabajo de inflar el layout. Veamos cómo quedaría el método `getView()` tras esta optimización:

```
@Override
public View getView(int position, View convertView, ViewGroup parent) {
    Item item = lista.get(position);
    /* Solo "inflamos" la vista si esta no está ya creada*/
    if (convertView == null) {
        LayoutInflater inflater = contexto.getLayoutInflater();
        view = inflater.inflate(R.layout.layout_item, null, true);
        TextView textView = (TextView) view.findViewById(R.id.textView2);
        textView.setText(item.getApi());
        ImageView imageView = (ImageView) view.findViewById(R.id.imageView);
        imageView.setImageDrawable(item.getImagen());
    }
    // Finalmente devolvemos la vista terminada de configurar.
    return convertView;
}
```

Con la optimización que acabamos de implementar conseguimos ahorrarnos el trabajo de inflar el layout definido cada vez que se muestra un nuevo elemento. Pero aún hay otras llamadas relativamente costosas que se siguen ejecutando en todas las llamadas. Me refiero a la obtención de la referencia a cada uno de los objetos a modificar mediante el método `findViewById()`. La búsqueda por ID de un control determinado dentro del árbol de objetos de un layout también puede ser una tarea costosa dependiendo de la complejidad del propio layout. ¿Por qué no aprovechamos que estamos "guardando" un layout anterior para guardar también la referencia a los controles que lo forman de forma que no tengamos que volver a buscarlos? Pues eso es exactamente lo que vamos a hacer mediante lo que suelen llamar patrón de diseño ViewHolder.

Nuestra clase ViewHolder tan sólo va a contener una referencia a cada uno de los controles que tengamos que manipular de nuestro layout, en nuestro caso un TextView y un ImageView. Definamos por tanto dentro de la MiAdaptador esta nueva clase de la siguiente forma:

```
private static class ViewHolder {
    TextView nombreVersion;
    ImageView imagenVersion;
}
```



La idea será por tanto crear e inicializar el objeto ViewHolder la primera vez que inflemos un elemento de la lista y asociarlo a dicho elemento de forma que posteriormente podamos recuperarlo fácilmente. ¿Pero dónde lo guardamos? Fácil, en Android todos los controles tienen una propiedad llamada Tag (podemos asignarla y recuperarla mediante los métodos setTag() y getTag() respectivamente) que puede contener cualquier tipo de objeto, por lo que resulta ideal para guardar nuestro objeto ViewHolder. De esta forma, cuando el parámetro view llegue informado sabremos que también tendremos disponibles las referencias a sus controles hijos a través de la propiedad Tag. Veamos el código modificado de getView() para aprovechar esta nueva optimización:

```
@Override
public View getView(int position, View view, ViewGroup parent) {
    Item item = lista.get(position);
    ViewHolder viewHolder;
    /* Solo "inflamos" la vista si esta no está ya creada*/
    if (convertView == null) {
        LayoutInflater inflater = contexto.getLayoutInflater();
        view = inflater.inflate(R.layout.layout_item, null, true);
        viewHolder = new ViewHolder();
        viewHolder.nombreVersion = convertView.findViewById(R.id.textView2);
        viewHolder.imagenVersion = convertView.findViewById(R.id.imageView);
        convertView.setTag(viewHolder);
    }
    else{
        viewHolder = (ViewHolder) convertView.getTag();
    }
    viewHolder.nombreVersion.setText(item.getApi());
    viewHolder.imagenVersion.setImageDrawable(item.getImagen());
    return convertView;
}
```

Con estas dos optimizaciones hemos conseguido que la aplicación sea mucho más respetuosa con los recursos del dispositivo.

Y dando un paso más cambia el modo de inflar por:

```
view = inflater.inflate(R.layout.layout_item, parent, false);
```

y conseguimos ajustar la altura necesaria para el ítem del Spinner.

Y (si quieres) acepta los avisos de AS (hacer private, etc.)

Ejecuta la aplicación para comprobar su funcionamiento.

## RecyclerView

Con la llegada de Android 5 (y Material Design) se incorporó un nuevo componente de control de selección llamado RecyclerView.

[RecyclerView](#) es una clase que hereda de [ViewGroup](#), que al igual que [ListView](#) y [GridView](#), nos va permitir mostrar grandes colecciones o conjuntos de datos con la única diferencia que solo se dedica a *reciclar* vistas delegando la tarea de renderizado a otros componentes que serán los que determinen la forma de acceder a los datos y cómo mostrarlos.

Sustituye a los clásicos ListView y GridView existentes desde hace tiempo ya que la ventaja de RecyclerView es que no necesita cargar toda la colección de elementos sino que los irá mostrando conforme el usuario vaya desplazándose por la lista, evitando así el colapso del terminal por tener que cargar un gran número de datos. RecyclerView se basa en un sistema de reutilización de vistas (ViewHolder) para poder conseguir dicho funcionamiento.



## P\_47\_Recycler\_01: Lista básica

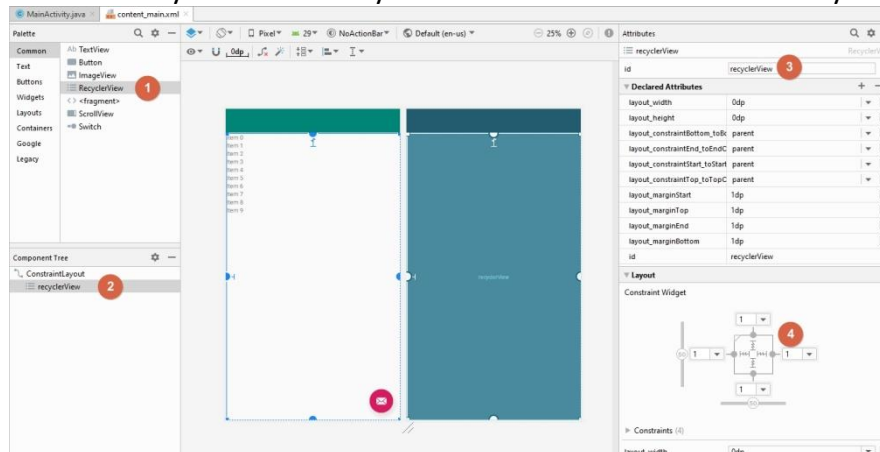
Deseamos una lista de palabras (en nuestro ejemplo, "Texto1", "Texto2",... "Texto50"). Crea un proyecto con la actividad principal basada en la plantilla Basic Activity y llamado P\_47\_Recycler\_01.

### Layouts

Necesitamos añadir el RecyclerView al layout de nuestra actividad y construir el layout que tendrá cada ítem de nuestra lista:

### Modificando content\_main

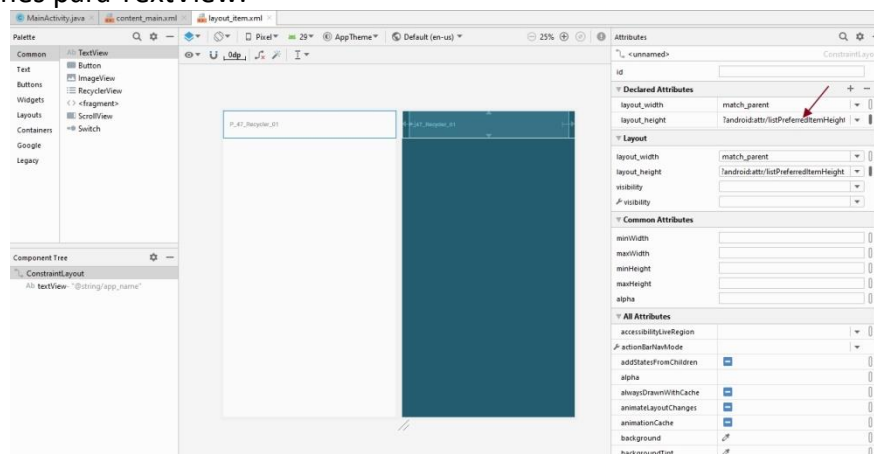
Eliminamos TextView y añadimos RecyclerView identificándolo como recyclerView:



Lógicamente podremos añadirle otros atributos: color de fondo, padding, visualización de scroll, etc.

### Layout ítem

Igual que hacíamos con los Spinner, debemos crear un layout con la forma en la que queremos que se represente cada elemento de la lista. Lo llamaremos, por ejemplo, layout\_item y en este ejemplo es tan simple que solo contiene un TextView. RECUERDA fijar bien la altura preferida del ítem (?android:attr/listPreferredItemHeight) en ConstraintLayout y construir las restricciones para TextView:





## Código clases

Tendremos que modificar nuestra clase MainActivity y crear otras clases (la del modelo de datos, la del adapter, etc.).

### Clase para el modelo de datos: Item.java

Necesitamos una clase POJO con el modelo de datos

```

public class Item {
    private String texto;

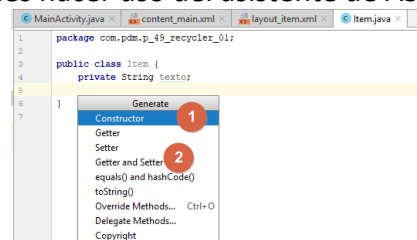
    public Item(String texto) {
        this.texto = texto;
    }

    public String getTexto() {
        return texto;
    }

    public void setTexto(String texto) {
        this.texto = texto;
    }
}

```

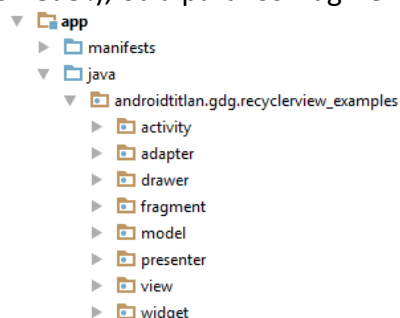
Recuerda que si no quieres teclear mucho, puedes hacer uso del asistente de AS:



¿Qué es un POJO en programación?

**POJO** son las iniciales de "Plain Old **Java** Object", que puede interpretarse como "Un objeto **Java** Plano Antiguo". Un **POJO** es una instancia de una clase que no extiende ni implementa nada en especial. Para los programadores **Java** sirve para enfatizar el uso de clases simples y que no dependen de un framework en especial.

(Hablando de organización, hay quien separa las clases por tipo y dentro del paquete construye una carpeta para las clases que son actividades, otra para las clases que son modelo de datos (las clases POJO!), otra para los fragmentos, etc. Ejemplo:



Como nuestros ejercicios son básicos, no vamos a seguir esta organización de momento; pero, si una aplicación empieza a ser "seria" siempre es conveniente para ayudarnos en nuestra tarea de programar.)

### Clase MainActivity.java

Lo primero que debemos establecer es el conjunto de datos que queremos ver, lo haremos cuando se crea la actividad llamando al método leerDatos() que nos devuelve una lista (ArrayList) de objetos de tipo Item:

```
ArrayList<Item> datos = leerDatos();
```

En el citado método, normalmente leeremos los datos de una BD del dispositivo o de un servidor, o de recursos de la aplicación, etc. En nuestro caso, textos que, por simplificar, serán: "Texto 0 ", "Texto 1", "Texto 2",...

```

private ArrayList<Item> leerDatos() {
    ArrayList<Item> datos = new ArrayList<>();
    for(int i=0; i<50; i++)
        datos.add(new Item("Texto " + i));
    return datos;
}

```



Como con cualquier otra vista, para poder usar el RecyclerView, debemos instanciar un objeto de su clase:

```
RecyclerView recyclerView = findViewById(R.id.recyclerView);
```

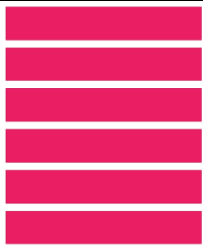
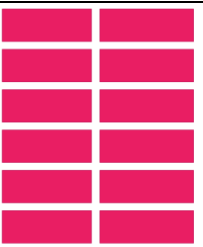
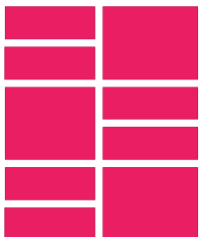
Tras obtener la referencia incluimos también una llamada al método `setHasFixedSize()`. Aunque esto no es obligatorio, es conveniente hacerlo cuando tengamos certeza de que el tamaño de nuestro RecyclerView no va a variar (por ejemplo debido a cambios en el contenido del adaptador), ya que esto permitirá aplicar determinadas optimizaciones sobre el control.

```
recyclerView.setHasFixedSize(true);
```

El siguiente paso obligatorio será determinar la forma en la que se distribuirán los datos en pantalla; para ello es necesario configurar un objeto `LayoutManager` (que va a ser el encargado de manejar el posicionamiento de los elementos) y asociarlo al `RecyclerView`.

## LayoutManager

Cuando decidíamos utilizar un `Spinner` o en 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` (obsoleto) 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`. El SDK incorpora de serie tipos de `LayoutManager` para las representaciones más habituales:

LinearLayoutManager	
	<pre>LinearLayoutManager layoutManager = new LinearLayoutManager (     this,     LinearLayoutManager.VERTICAL,     false);</pre> <p><small>public LinearLayoutManager (Context context, int orientation, boolean reverseLayout)</small></p> <p><small>Parameters</small></p> <p><small>context</small> Current context, will be used to access resources.</p> <p><small>orientation</small> Layout orientation. Should be <code>HORIZONTAL</code> or <code>VERTICAL</code>.</p> <p><small>reverseLayout</small> When set to true, layouts from end to start.</p>
GridLayoutManager	
	<pre>GridLayoutManager gridLayoutManager = new GridLayoutManager (     this,     2, //number of grid columns     GridLayoutManager.VERTICAL,     false);</pre>
StaggeredGridLayoutManager	
	<pre>StaggeredGridLayoutManager staggeredGridLayoutManager = new StaggeredGridLayoutManager (     2, //number of grid columns     GridLayoutManager.VERTICAL);</pre> <p><small>//Sets the gap handling strategy for StaggeredGridLayoutManager</small></p> <pre>staggeredGridLayoutManager.setGapStrategy (StaggeredGridLa youtManager.GAP_HANDLING_NONE);</pre>





## GridLayoutManager + SpanSizeLookup

Usando el método `setSpanSizeLookup(SpanSizeLookup spanSizeLookup)` establecemos la cantidad de spans que debe ocupar cada elemento en el adaptador.



```
GridLayoutManager gridLayoutManager = new
GridLayoutManager(
    this,
    2, //number of grid columns
    GridLayoutManager.VERTICAL,
    false);

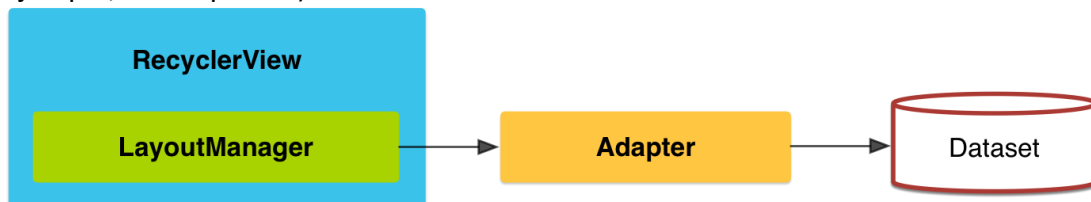
gridLayoutManager.setSpanSizeLookup(new
GridLayoutManager.SpanSizeLookup() {
    @Override
    public int getSpanSize(int position) {
        //stagger rows custom
        // las celdas de posición múltiplo de 3 (3,6,9,...)
        return (position % 3 == 0 ? 2 : 1);
    }
});
```

El potencial que brindan estos layouts por defecto es más que suficiente la mayoría de las veces para ser usados de forma directa en un RecyclerView, aunque también puedes implementar un estilo de renderizado diferente a los que el framework te provee [creando tu propio LayoutManager](#) que se adapte a tu necesidad. De ahí uno de los puntos fuertes del nuevo componente: su flexibilidad.

Si queremos mostrar los datos en forma de lista con desplazamiento vertical 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()`:

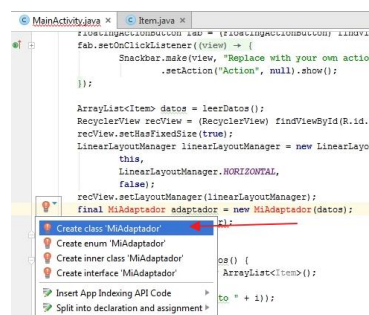
```
RecyclerView.LayoutManager layoutManager=new LinearLayoutManager(this,
LinearLayoutManager.VERTICAL, false);
recView.setLayoutManager(layoutmanager);
```

Para cargar los datos en el RecyclerView necesitamos un adaptador (de nombre, por ejemplo, `MiAdaptador`).



```
MiAdaptador miAdaptador = new MiAdaptador(datos);
recView.setAdapter(miAdaptador);
```

Podemos usar el asistente de AS para que nos cree la clase corrigiendo los errores que aparecen en la MainActivity:



El código de la nueva clase propuesta por el asistente de AS es:

```
public class MiAdaptador {
    public MiAdaptador(ArrayList<Item> datos) {
    }
}
```

Empecemos nuestras modificaciones, en el constructor de la clase añadimos:

```
this.datos=datos;
```

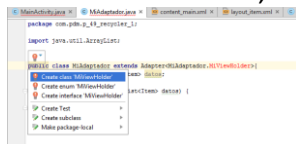


El adaptador a crear hereda de la clase RecyclerView.Adapter, dicha clase obliga a seguir el patrón de diseño ViewHolder.

Esa obligatoriedad del uso del patrón ViewHolder se plasma indicándose en la declaración de la clase:

```
public class MiAdaptador extends RecyclerView.Adapter<MiAdaptador.MiViewHolder> {
```

haciendo uso del asistente de corrección de errores,



se crea la clase MiViewHolder dentro de la propia clase MiAdaptador (esto no es obligatorio, podría ser una clase externa si no hubiéramos puesto MiAdaptador).

```
public class MiViewHolder {
}
```

Lo importante de esta clase es que extienda de RecyclerView.ViewHolder

```
public class MiViewHolder extends RecyclerView.ViewHolder{
```

volvemos a hacer uso del asistente de corrección para que la clase quede:

```
public class MiViewHolder extends RecyclerView.ViewHolder{
    public MiViewHolder(View itemView) {
        super(itemView);
    }
}
```

Dentro de esta clase es donde tenemos que incluir como atributos las referencias a los controles del layout de un elemento de la lista (en nuestro caso el TextView) e inicializarlas en el constructor utilizando como siempre el método findViewById() sobre la vista recibida como parámetro.

```
public class MiViewHolder extends RecyclerView.ViewHolder{
    public TextView textView;
    public MiViewHolder(View itemView) {
        super(itemView);
        textView=itemView.findViewById(R.id.textview);
    }
}
```

Volviendo a usar el asistente, añadimos los métodos que faltan y empezamos a completarlos:

- onCreateViewHolder(): Encargado de crear e inflar las vistas que va a mostrar el RecyclerView.
- onBindViewHolder(): Reemplaza el contenido de la vista con nuestra colección de datos.
- getItemCount(): Devuelve el número de elementos de la colección de datos.

```
public class MiAdaptador extends RecyclerView.Adapter<MiAdaptador.MiViewHolder> {
    <Item> datos;
    List<Item> datos; {
```

```
@Override
public MiViewHolder onCreateViewHolder(ViewGroup parent, int viewType) {
    View view=LayoutInflater.from(parent.getContext()).inflate(R.layout.layout_item,
parent, false);
    return new MiViewHolder(view);
}
```

Nos limitamos a inflar una vista a partir del layout correspondiente a los elementos de la lista y crear y devolver un nuevo MiViewHolder llamando al constructor de nuestra clase MiViewHolder pasándole dicha vista como parámetro.

```
@Override
public void onBindViewHolder(MiViewHolder holder, int position) {
    String texto=datos.get(position).getTexto();
    holder.textView.setText(texto);
}
```

Tan sólo tendremos que recuperar el objeto Item correspondiente a la posición recibida como parámetro y asignar sus datos sobre el MiViewHolder también recibido como parámetro

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

Devolverá el tamaño de la lista de datos



Ejecuta la aplicación para comprobar su funcionamiento.

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` y utilizar un `GridLayoutManager`:

```
RecyclerView.LayoutManager layoutManager=new  
GridLayoutManager(this,3,GridLayoutManager.VERTICAL,false);
```

### 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.

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

```
RecyclerView.ItemDecoration itemDecoration = new DividerItemDecoration(this,  
DividerItemDecoration.VERTICAL);  
recView.addItemDecoration(itemDecoration);
```

Ejecuta la aplicación para comprobar su funcionamiento.



## Responder a eventos

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 incluye un evento `onItemClick()`. 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, veamos una de ellas.

Supongamos que al hacer click en un elemento deseamos que empiece una nueva actividad (créala con el nombre SegundaActivity, con plantilla Basic Activity y en el layout añade un TextView con identificador `textView2`) que servirá para visualizar el nº de ítem pulsado.

Añade al método `onCreate()` de la clase SegundaActivity:

```
Bundle bundle=getIntent().getExtras();
int recibido = bundle.getInt("enviado");
TextView textView=findViewById(R.id.textView2);
textView.setText(String.valueOf(recibido));
```

Modifica la clase MiViewHolder del adaptador para que haga uso del método `getLayoutPosition()` que como su nombre indica devuelve la posición del layout del ítem:

```
public class MiViewHolder extends RecyclerView.ViewHolder {
    public TextView textView;
    public MiViewHolder(View itemView) {
        super(itemView);
        itemView.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Intent intent = new Intent(applicationContext, SegundaActivity.class);
                Bundle bundle = new Bundle();
                bundle.putInt("enviado", getLayoutPosition());
                intent.putExtras(bundle);
                intent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
                applicationContext.startActivity(intent);
            }
        });
        textView = (TextView) itemView.findViewById(R.id.textView);
    }
}
```

Observa que para llamar al método `startActivity()`, necesitamos el contexto de la aplicación, eso nos hace tener que cambiar el constructor en el adapter:

```
public MiAdaptador(ArrayList<Item> datos, Context applicationContext) {
    this.datos = datos;
    this.applicationContext = applicationContext;
}
```

Y en la MainActivity necesitaremos cambiar:

```
MiAdaptador adaptador = new MiAdaptador(datos, getApplicationContext());
```

Prueba la aplicación para comprobar su funcionamiento.

Funciona, pero deberíamos mejorar el código porque estamos aplicando la respuesta al evento en el Adapter. Es decir, estamos mezclando lógica (respuesta al evento) con diseño (construir el recycler en el adapter). En el siguiente ejemplo, lo veremos.

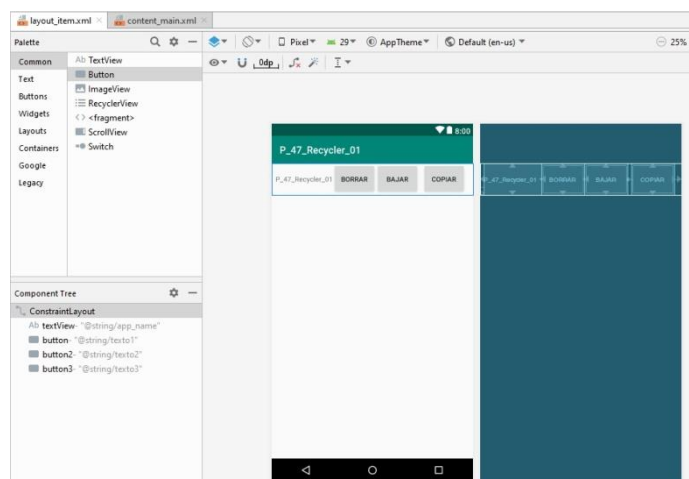


## Recycler\_02: Cambiando los ítems (ItemAnimator)

Para no perder el trabajo hecho hasta ahora, haz una copia del proyecto llamada P\_48\_Recycler\_02. Borra la segunda actividad (clase java, layout y referencia en AndroidManifest).

### Cambios en layout

Modifiquemos nuestro layout\_item para añadirle tres botones que nos permitan borrar, bajar una posición o copiar un determinado ítem de la lista:



### Responder a eventos desde la actividad

Para responder a los clicks de los botones necesitaremos hacer uso de los métodos `notifyItemRemoved()`, `notifyItemMoved()` y `notifyItemInserted()` de la clase [RecyclerView.Adapter](#).

Vamos a ver otra manera de responder a eventos, respondiendo desde la MainActivity a partir de interfaz propia (siempre es mejor responder a los eventos en actividades que en adaptadores, se supone que el adaptador puede ser reusado por otras actividades que podrían tener respuestas distintas, el famoso MVC – modelo-vista-controlador).

Observa las modificaciones de la clase MiAdaptador:

```
public class MiAdaptador extends Adapter<MiAdaptador.MiViewHolder> {

    public ArrayList<Item> datos;
    private MiOnItemClickListener listener;

    public interface MiOnItemClickListener {
        void miOnItemClick(int botonClickado, int posicionItem);
    }

    public MiAdaptador(ArrayList<Item> datos, MiOnItemClickListener listener) {
        this.datos = datos;
        this.listener = listener;
    }

    @Override
    public MiViewHolder onCreateViewHolder(ViewGroup parent, int position) {
        View view =
        LayoutInflater.from(viewGroup.getContext()).inflate(R.layout.layout_item, viewGroup,
        false);
        return new MiViewHolder(view);
    }

    @Override
    public void onBindViewHolder(MiViewHolder holder, int position) {
        String texto = datos.get(position).getTexto();
```



```

        holder.textView.setText(texto);
    }

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

    public class MiViewHolder extends RecyclerView.ViewHolder implements
    View.OnClickListener {
        public Button borrar, bajar, copiar;
        public TextView textView;

        public MiViewHolder(View itemView) {
            super(itemView);
            borrar = itemView.findViewById(R.id.button);
            bajar = itemView.findViewById(R.id.button2);
            copiar = itemView.findViewById(R.id.button3);
            borrar.setOnClickListener(this);
            bajar.setOnClickListener(this);
            copiar.setOnClickListener(this);
            textView = itemView.findViewById(R.id.textView);
        }

        @Override
        public void onClick(View v) {
            listener.miOnClick(v.getId(), getLayoutPosition());
        }
    }
}

```

**Modifica en MainActivity** la construcción del adapter (haz caso a las sugerencias de AS para crear variable global miAdaptador y para que la variable datos sea de tipo final):

```

miAdaptador = new MiAdaptador(datos, new MiAdaptador.MiOnClickListener() {
    @Override
    public void miOnClick(int botonClickado, int posicionItem) {
        switch (botonClickado) {
            case R.id.button:
                datos.remove(posicionItem);
                miAdaptador.notifyItemRemoved(posicionItem);
                break;
            case R.id.button2:
                // si el ítem no es el último
                if (posicionItem < recyclerView.getChildCount() - 1) {
                    Item aux = datos.get(posicionItem);
                    datos.set(posicionItem, datos.get(posicionItem + 1));
                    datos.set(posicionItem + 1, aux);
                    miAdaptador.notifyItemMoved(posicionItem, posicionItem + 1);
                }
                break;
            case R.id.button3:
                String texto = datos.get(posicionItem).getTexto();
                datos.add(posicionItem + 1, new Item("Copia de " + texto));
                miAdaptador.notifyItemInserted(posicionItem + 1);
                break;
        }
    }
});
recyclerView.setAdapter(miAdaptador);
RecyclerView.ItemAnimator itemAnimator = new DefaultItemAnimator();
recyclerView.setItemAnimator(itemAnimator);

```

Observa el uso de los métodos `notifyItemRemoved()`, `notifyItemMoved()` y `notifyItemInserted()`.

La clase `ItemAnimator` nos permite muchas otras animaciones.

## Práctica propuesta

Realiza los ejercicios propuestos en Ejercicios\_11\_ RecyclerView