RecyclerView

El Form Widget "RecyclerView" es uno de los más utilizados, permite mostrar información en forma de lista "scroleable" y manejar eventos de clicks sobre los ítems.

La información que mostraremos en cada ítem, estará contenida en un objeto (Persona, en nuestro ejemplo). Utilizaremos un adapter propio, el cual hereda de RecyclerView.Adapter, para decirle al objeto RecyclerView, qué debe mostrar, este adapter recibirá una List de objetos Persona, y generará un objeto View por cada fila de la lista que se verá en pantalla.

NOTA: Antes de la versión de Android 5.0, el Form Widget utilizado para esta tarea era el "ListView", el cual requería mucho trabajo en la implementación del adapter para lograr una buena performance en el scrolling. Debido a que RecyclerView se encuentra en la biblioteca de compatibilidad, utilizaremos siempre esta opción.

Paso número uno: Incluyendo dependencias

En nuestro archivo build.gradle del módulo "app", incluiremos dentro de las dependencias:

```
dependencies {
    compile 'com.android.support:design:23.1.1'
}
```

Paso número dos: Definición de la entidad a representar

Definimos una entidad que contendrá la información que se quiere mostrar en una lista, en una clase "Persona".

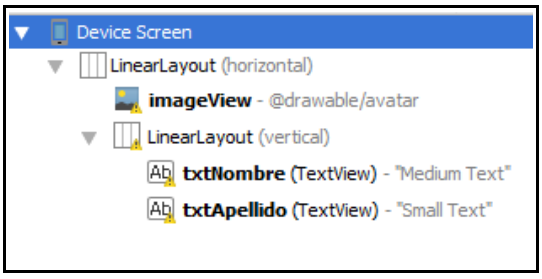
```
public class Persona {

        private  String nombre;
        private  String apellido;

        public void setNombre(String texto1) {
                this.nombre = texto1;
        }
        public String getNombre() {
                return nombre;
        }
        public void setApellido(String texto2) {
                this.apellido = texto2;
        }
        public String getApellido() {
                return apellido;
        }

}
```

Paso número tres: Definición del layout del item

En nuestro adapter personalizado, crearemos objetos View por cada item de la lista, estos objetos View se cargaran con la información correspondiente al item y se devolveran al Recyclerview para que este muestre el item en pantalla.
Para poder crear un objeto View con lo que nosotros queramos que tenga el item (por ejemplo una foto a la izquierda y dos textos a la derecha) deberemos crear un archivo XML de layout, que contendra solamente los elementos de un item, luego mediante un proceso llamado "inflate" podremos construir objetos View tomando como "molde" este archivo XML de layout.

Creamos el layout "item_layout.xml":

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <ImageView
        android:layout_width="60dp"
        android:layout_height="60dp"
        android:src="@drawable/avatar"
        android:id="@+id/imageView" />

    <LinearLayout
        android:layout_marginLeft="10dp"
        android:orientation="vertical"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent">

        <TextView
            android:layout_marginTop="5dp"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textAppearance="?android:attr/textAppearanceMedium"
            android:text="Medium Text"
            android:id="@+id/txtNombre" />

        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textAppearance="?android:attr/textAppearanceSmall"
            android:text="Small Text"
            android:id="@+id/txtApellido" />
    </LinearLayout>
</LinearLayout>
```



Paso número cuatro: Definición del adapter

Definimos nuestro propio Adapter heredando de RecyclerView.Adapter:

```
import android.support.v7.widget.RecyclerView.Adapter;
import android.view.ViewGroup;

public class MyAdapter extends Adapter<MyViewHolder> {

    @Override
    public MyViewHolder onCreateViewHolder(ViewGroup parent, int viewType) {
        return null;
    }

    @Override
    public void onBindViewHolder(MyViewHolder holder, int position) {

    }

    @Override
    public int getItemCount() {
        return 0;
    }
}
```

Nótese import RecyclerView.Adapter

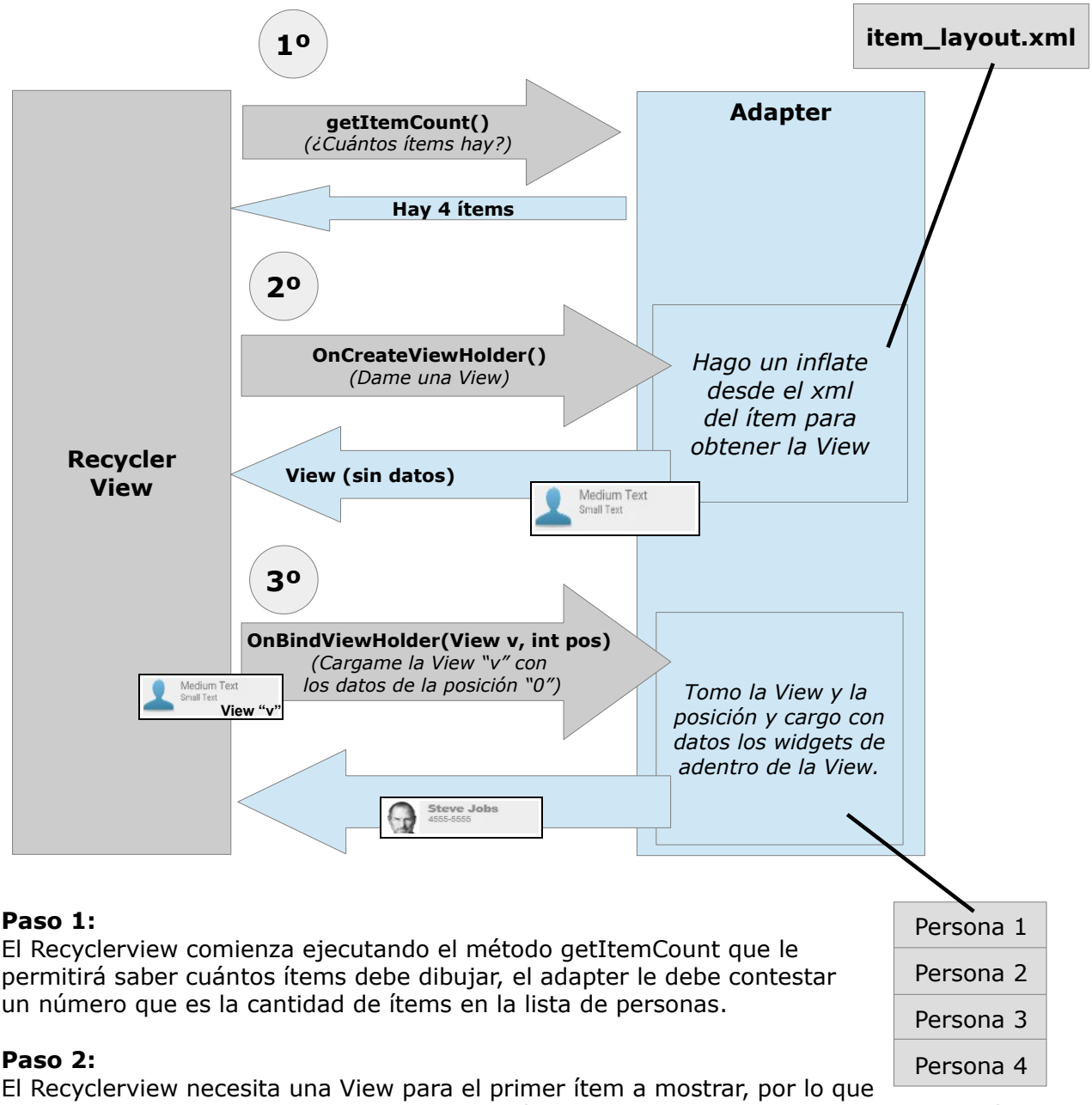
Al heredar, deberemos sobreescribir tres métodos definidos en la clase Adapter. Estos métodos no serán invocados por nosotros, sino por el objeto RecyclerView, recordemos que el objeto RecyclerView tendrá la referencia de un objeto MyAdapter, por lo que internamente ejecutará estos métodos para poder saber qué debe dibujar en pantalla. Los métodos son los siguientes:

- **onCreateViewHolder:** El recyclerview ejecutará este método cada vez que necesite de un objeto View nuevo para poder representar un ítem de la lista en pantalla. Aquí dentro deberemos realizar el antes mencionado proceso de "inflate" para poder crear un objeto View a partir de nuestro archivo XML que representa a un ítem visualmente.

- **onBindViewHolder:** El recyclerview ejecutará este método cada vez que tenga un objeto View disponible y lo necesite cargar con los datos del ítem que quiere representar (recordemos que el objeto View creado no tiene cargado datos asociados a nuestra lista de personas) el ítem a mostrar esta asociado a una posición de nuestra lista de personas, en este método se obtendrá el objeto Persona que corresponde y se cargarán los datos del mismo en el objeto View.

- **getItemCount:** El recyclerview ejecutará este método para saber cuantos ítems debe representar, se devuelve el tamaño de la lista de personas.

A continuación mostramos un diagrama que pretende explicar el funcionamiento paso a paso del diálogo entre en Recyclerview y el Adapter.



Paso 1:
El Recyclerview comienza ejecutando el método getItemCount que le permitirá saber cuántos ítems debe dibujar, el adapter le debe contestar un número que es la cantidad de ítems en la lista de personas.

Paso 2:
El Recyclerview necesita una View para el primer ítem a mostrar, por lo que ejecuta a onCreateViewHolder para que el método le devuelva una. Dentro de este método el adapter creará una View tomando como modelo el archivo XML de layout definido para tal fin. Es importante destacar que los widgets dentro de esta view no tienen cargados datos (los textviews no tienen textos cargados y el ImageView no tiene una imagen en particular)

Paso 3:
El Recyclerview necesita cargar a esa View con datos antes de mostrarla en pantalla (es decir, necesita que los dos Textviews y el ImageView dentro de la View sean cargados con datos). Para ello ejecuta el método onBindViewHoder, al cual le pasa como argumento la View a ser cargada y la posición de la lista de Personas de donde se debe tomar la información para cargar la View.

El paso 2 y 3 se repiten hasta que se llena la pantalla de ítems, es decir que el proceso se repite para cada ítem hasta que los ítems hayan cubierto la pantalla, si la lista tiene 100 ítems, pero en pantalla entran solo 6, entonces el proceso de los pasos 2 y 3 se ejecutará para las posiciones de la 0 a la 5 (es decir para los primeros 6 ítems)

Reutilización de Views

En el estado mencionado anteriormente, si el usuario comienza a hacer "scroll" hacia abajo, para visualizar los ítems de la lista que están por debajo de los primeros 6 visibles, el Recyclerview comenzará solo a ejecutar el método onBindViewHolder, y nunca más ejecutará el método onCreateViewHolder, en otras palabras, nunca más creará objetos View, solo cargará los que ya existen con diferentes datos, aprovechando que cuando la view de un ítem se deja de ver, se puede utilizar como otro ítem de los que sí se ven.

Esto hace que la sensación de "scroll" sea fluida, ya que el proceso de creación de Views es costoso y de crearse Views todo el tiempo la animación no podría ser tan fuida.

Diferencia entre View y MyViewHolder

Como se observa en la definición del método onCreateViewHolder, no se devuelve un objeto View como se dijo en la explicación anterior, sino un objeto ViewHolder.

Esta diferencia se debe a tratar de mejorar la performance en la ejecución del método onBindViewHolder. Debido a que dentro de este método debemos acceder a los widgets dentro de la View (los dos textviews por ejemplo) para ello debemos ejecutar el método findViewById, que nos permitirá encontrar los dos textviews dentro de la View, pero como este proceso es también costoso, se utiliza un objeto MyViewHolder que va a "contener" a la View y a las referencias de todos los widgets que posee dentro.
De esta forma nos evitamos ejecutar findViewById cada vez que se ejecuta el método onBindViewHolder.

La clase MyViewHolder, la cual deberá heredar de RecyclerView.ViewHolder:

```
import android.support.v7.widget.RecyclerView;
import android.view.View;

public class MyViewHolder extends RecyclerView.ViewHolder {

    TextView txtNombre;
    TextView txtApellido;

    public MyViewHolder(View itemView) {
        super(itemView);
    }
}
```

Dentro de esta clase MyViewHolder, definimos como atributos, las Views que contiene el ítem que definimos en el archivo de layout (los dos TextViews). La idea de esta clase es contener las referencias de las Views que contiene el ítem para no tener que ejecutar el método findViewById cada vez que se quiera obtener la referencia de los TextViews contenidos dentro de la View del item, de modo que lo ejecutamos en el constructor de esta clase el cual recibe como argumento la View que representa a un ítem, que más adelante veremos como crearla.

```
    public MyViewHolder(View itemView)
    {
        super(itemView);
        txtNombre = (TextView) itemView.findViewById(R.id.txtNombre);
        txtApellido = (TextView) itemView.findViewById(R.id.txtApellido);
    }
```

Como se observa en la figura, un objeto MyViewHolder es un objeto que contendrá como atributos, dos TextViews obtenidos de una View pasada como argumento en su constructor, así como también dicha View (como atributo de la clase ViewHolder de la que hereda).

La ventaja de tener los dos TextViews como atributos, es no tener que llamar a *findViewById* cada vez que se necesitan.

Creación del adapter

Nuestro adapter, deberá recibir la lista de objetos persona que queremos representar en el RecyclerView, de modo que escribiremos un constructor que reciba la lista de objetos Persona:

```
public class MyAdapter extends Adapter<MyViewHolder> {

    private List<Persona> lista;

    public MyAdapter(List<Persona> lista)
    {
        this.lista=lista;
    }
```

El adapter debe devolver en el método getItemCount, la cantidad de ítems de la lista, de modo que reemplazamos el "return 0" por el size de la lista:

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

Ahora nos resta hablar sobre los métodos "onCreateViewHolder" y "onBindViewHolder".

El primero de ellos, es donde deberemos construir un objeto View a partir de nuestro archivo item_layout.xml y una vez obtenido dicho objeto View (que dentro posee los dos TextViews, y el ImageView que definimos en el archivo de layout) lo retornamos.

El RecyclerView le ejecutará este método al adapter cada vez que tenga que colocar un ítem en pantalla y no tenga ninguno para reutilizar.

```
@Override
public MyViewHolder onCreateViewHolder(ViewGroup parent, int viewType) {
    View v = LayoutInflater.from(parent.getContext()).inflate(R.layout.item_layout, parent, false);
    MyViewHolder myViewHolder = new MyViewHolder(v);
    return myViewHolder;
}
```

En la primera línea creamos un objeto View a partir del archivo xml de layout que definimos para el ítem utilizando el método estático *from* de la clase LayoutInflater el cual devuelve un objeto Inflater, al cual le ejecutamos el método inflate.
Luego creamos nuestro objeto MyViewHolder pasándole la View creada y lo devolvemos.

El método onBindViewHolder, se ejecutará por cada ítem, antes de que éstos se muestren en pantalla, para que aquí los carguemos con la información proveniente de nuestro modelo de datos (nuestra lista de personas) en este método se recibe como argumento el objeto

MyViewHolder que dentro tiene la View y la referencia de los TextViews del ítem, y también la posición del ítem que se quiere representar, de forma que podamos acceder al objeto Persona correspondiente en la lista y cargar los datos en los TextViews:

```
@Override
public void onBindViewHolder(MyViewHolder holder, int position) {
    Persona p = lista.get(position);
    holder.txtNombre.setText(p.getNombre());
    holder.txtApellido.setText(p.getApellido());
}
```

Paso número cinco: colocar el RecyclerView en el layout de la Activity.

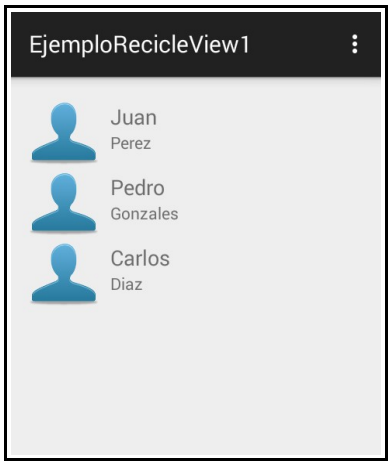
Por último, colocamos en el archivo de layout que carga la Activity, el objeto RecyclerView y le asignamos el id "list":

```
    <android.support.v7.widget.RecyclerView
        android:id="@+id/list"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:scrollbars="vertical" />
```

Ahora en la Activity, obtenemos la referencia del RecyclerView, creamos el modelo de datos, el adapter, y los relacionamos.

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

    List<Persona> personas = new ArrayList<Persona>();
    personas.add(new Persona("Juan","Perez"));
    personas.add(new Persona("Pedro","Gonzales"));
    personas.add(new Persona("Carlos","Diaz"));

    RecyclerView list = (RecyclerView)findViewById(R.id.list);

    LinearLayoutManager layoutManager = new LinearLayoutManager(this);
    list.setLayoutManager(layoutManager);

    MyAdapter adapter = new MyAdapter(personas);
    list.setAdapter(adapter);
}
```

Se debe notar que además del Adapter, le estamos asignando al RecyclerView, un LinearLayoutManager, de esta forma le indicamos que coloque un item debajo del otro, al cambiar este objeto o escribir uno propio, podemos lograr que los ítems se coloquen en diferentes posiciones.

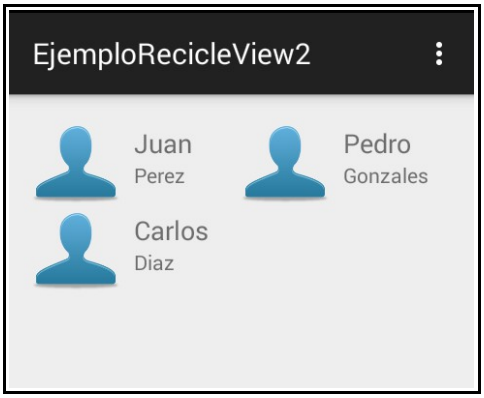


Transformando la lista en una grilla

Como se mencionó anteriormente, si cambiamos el objeto LinearLayoutManager por GridLayoutManager, logramos que los ítems se representen en forma de grilla:

```
//LinearLayoutManager layoutManager = new LinearLayoutManager(this);
GridLayoutManager layoutManager = new GridLayoutManager(this,2);
```

El segundo parámetro que se le pasa al constructor, es la cantidad de columnas que deseamos que posea la grilla.



Paginación de ítems en la lista

Muchas veces, la información mostrada en la lista es mucha, y se obtiene de un medio que no es rápido, y no se puede entregar toda la información de una sola vez (por ejemplo si se obtiene por HTTP), por ello, la información se pagina, por ejemplo, en bloques de 10 ítems, para ello, se necesita conocer el momento que el adapter mostró el ultimo ítem, porque en ese momento, debe irse a buscar la página siguiente y agregarla a la lista.

Detectando que se mostró el último ítem de la lista:

Primero, agregamos en el Constructor de nuestro adapter un parámetro en donde pasaremos una referencia de la Activity que utilizará el adapter:

```
private MainActivity a;
public MyAdapter(List<Persona> lista, MainActivity a)
{
    this.lista=lista;
    this.a=a;
}
```

En la Activity, pasaremos “this” al segundo parámetro, para que el adapter tenga la referencia de la Activity y pueda llamar a los métodos de la misma.

```
MyAdapter adapter = new MyAdapter(personas,this);
```

En el adapter, en el método onBindViewHolder, agregamos:

```
// si se mostro la ultima posicion, envio un mensaje a la activity
if((position+1)==getItemCount())
{
        a.seMostroFinalLista();
}
//_____________________________________________________________
```

El método getItemCount devuelve la cantidad de elementos que tiene la lista, y "position" es el argumento pasado a onBindViewHolder para saber que elemento de la lista se debe cargar con datos, si coincide con el último, llamamos a un método de la Activity, para avisarle que se mostró el ultimo ítem.

Puede observarse que "a" es la referencia a MainActivity, por lo que puede llamarse al método "seMostroFinalLista()" que agregaremos en nuestra MainActivity, o cualquier otro.

```
public void seMostroFinalLista()
{
    if(flagNuevaPagina==false)
    {
        flagNuevaPagina=true;

        // pido el siguiente bloque de datos a algun servicio, y cuando lleguen los datos, ponemos
       // nuevamente en false el flag
        // ....
    }
}
```

¿Por qué se utiliza el "flagNuevaPagina"?

Debido a que no esta garantizado la cantidad de veces que se llama al método onBindViewHolder (puede ser más de una vez para el último ítem), no estaría bien que cada vez que se llama, se traiga una pagina más, por otro lado, el servicio que trae los datos de la nueva página, tarda una considerable cantidad de tiempo, por lo que si el usuario hace scroll hacia arriba y luego hacia abajo más de una vez, no estaría bien que por cada vez que llegue abajo, se traiga una página más, de modo que con el flag, solo se activará el pedido de una nueva página, si ya se completó la última pedida.

NOTA: Es más prolijo definir una interfaz con el método "seMostroFinalLista" y hacer que la Activity (u otra clase) implemente dicha interfaz, y que el adapter reciba un objeto del tipo de la interfaz en vez de una Activity.

Modificación del modelo de datos

Supongamos que, según el ejemplo anterior, obtenemos más objetos Persona al detectar el fin de scroll, cuando esto ocurra, agregaremos los objetos a la lista "personas" pero este cambio no se reflejará en pantalla hasta que no ejecutemos el método notifyDataSetChanged perteneciente al adapter. Al ejecutar este método se redibujará la lista y se verán los nuevos ítems. Esto es también necesario si modificamos cualquier atributo de cualquier objeto Persona de la lista.

```
adapter.notifyDataSetChanged();
```

Eventos

Es muy común querer escuchar el evento de click sobre un ítem de la lista, para ello, deberemos setear el listener de click sobre la view de cada ítem al momento de su creación.

Crearemos una interface donde definiremos el método que se ejecutará cuando se produzca un click sobre un ítem:

```
public interface MyOnItemClick {

    void onItemClick(int position);
}
```

Luego haremos que nuestra Activity implemente la interface:

```
public class MainActivity extends ActionBarActivity implements MyOnItemClick {
        //…
        @Override
        public void onItemClick(int position) {

        }
```

De esta manera, nuestra Activity será el listener de los clicks sobre los ítems, solo nos resta trabajar en nuestro adapter para notificar a la activity cuando una view de un ítem es presionada.

Agregamos en el constructor del adapter una referencia de la activity del tipo MyItemClick.

```
private List<Persona> lista;
private MyOnItemClick listener;

public MyAdapter(List<Persona> lista,  MyOnItemClick listener)
{
    this.lista=lista;
    this.listener = listener;
}
```

Y en la activity, al crear el adapter, le pasamos “this”, ya que la activity implementa la interface.

```
MyAdapter adapter = new MyAdapter(personas,this);
```

Dentro del adapter, modificaremos el método onCreateViewHolder, para que este listener se pase como argumento en el constructor de nuestra clase MyViewHolder:

```
@Override
public MyViewHolder onCreateViewHolder(ViewGroup parent, int viewType) {
    View v = LayoutInflater.from(parent.getContext()).inflate(R.layout.item_layout, parent, false);
    MyViewHolder myViewHolder = new MyViewHolder(v,listener);
    return myViewHolder;
}
```

Por último, modificaremos la clase MyViewHolder para que reciba nuestro listener. Esta clase será la que escuche los evento de click sobre la View, y cuando se produzcan, ejecutará nuestro método onItemClick reportando el evento.

Implementamos la interface OnItemClickListener en la clase MyViewHolder, y escribimos el método onClick

```
public class MyViewHolder extends RecyclerView.ViewHolder implements View.OnClickListener {
        //...
        @Override
        public void onClick(View v) {

        }
```

Modificamos el constructor para que reciba MyOnItemClick, y también asignamos a la View itemView, como listener, a esta clase (osea *this*) mediante setOnClickListener:

```
private MyOnItemClick listener;

public MyViewHolder(View itemView,MyOnItemClick listener) {
    super(itemView);
    txtNombre = (TextView) itemView.findViewById(R.id.txtNombre);
    txtApellido = (TextView) itemView.findViewById(R.id.txtApellido);
    itemView.setOnClickListener(this);
    this.listener = listener;
}
```

Al producirse un click sobre la view (que es el item) se ejecutará el método onClick de la clase MyViewHolder, allí deberemos ejecutar el método onItemClick de nuestro listener.

```
@Override
public void onClick(View v) {
    listener.onItemClick(position);
}
```

Como este método requiere la posición del ítem, y el objeto View no conoce en qué posición se encuentra, creamos un atributo más en la clase MyViewHolder del tipo *int* llamado *position* y un método para poder cargarlo desde el adapter:

```
private int position;

public void setPosition(int position)
{
    this.position = position;
}
```

Ahora solo nos resta llamar a este método que actualiza la posición del objeto MyViewHolder en la lista en el método onBindViewHolder en el adapter:

```
@Override
public void onBindViewHolder(MyViewHolder holder, int position) {
    Persona p = lista.get(position);
    holder.txtNombre.setText(p.getNombre());
    holder.txtApellido.setText(p.getApellido());
    holder.setPosition(position);
}
```