

Apuntes

[Descargar estos apuntes](#)

Tema 8. Interfaz de Usuario III

Índice

1. [Spinner](#)
2. [RecyclerView](#)
 1. [Click sobre un elemento de la lista](#)
 2. [Click en cualquier lugar de la vista](#)
 3. [Click en cualquier lugar de la vista pasando información a la Actividad Principal](#)
 4. [Otros efectos sobre el Recycler](#)
 5. [Selección Múltiple en el Recycler](#)

Spinner

Las listas desplegables en Android se llaman **Spinners**. Es una lista que muestras los elementos una vez pulsada la flecha de despliegue, el usuario puede seleccionar de la lista emergente uno elemento. El código de la vista de un Spinner será el siguiente:

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

Los elementos que mostrará nuestro Spinner no se pueden indicar directamente, sino que es necesario crear una estructura de elementos que podamos asociar a éste en nuestro código. Una de las maneras sería usando un Array de Strings para luego asignarlo al spinner:

```
var colores = arrayOf("Rojo", "Verde", "Azul")
```

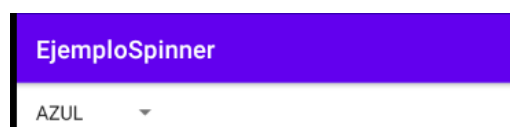
A continuación debemos definir el adaptador, en nuestro caso un objeto ArrayAdapter, que será el objeto que lanzaremos con nuestro Spinner.

```
val adaptador = ArrayAdapter(this,
    android.R.layout.simple_list_item_1, colores)
```

Creamos el adaptador en sí, al que pasamos 3 parámetros:

1. El contexto, que normalmente será simplemente una referencia a la actividad donde se crea el adaptador.
2. El ID del layout sobre el que se mostrarán los datos del control. En este caso le pasamos el ID de un layout predefinido en Android (android.R.layout.simple_spinner_item), formado únicamente por un control textView, pero podríamos pasar el ID de cualquier layout de nuestro proyecto con cualquier estructura y conjunto de controles.
3. El elemento que contiene los datos a mostrar.

Con estas simples líneas tendremos un control similar al de la siguiente imagen:



📁 Vamos a crear un proyecto de ejemplo llamado Spinner al que añadiremos el siguiente código en la **MainActivity.kt**, y no olvides añadir un spinner con *id=spinner* en el layout **activity_main.xml**:

```
class MainActivity : AppCompatActivity(),
    AdapterView.OnItemSelectedListener {
    var colores = arrayOf("Rojo", "Verde", "Azul")
    lateinit var listaColores: Spinner
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        8 val adaptador = ArrayAdapter(this, android.R.layout.
            simple_list_item_1, colores)
        listaColores = findViewById(R.id.spinner)
        11 listaColores.adapter=adaptador
        12 listaColores.setOnItemSelectedListener(this)
    }
```

📌 **Línea 8** creamos el adaptador, como hemos explicado anteriormente, para asignarlo al Spinner en la **Línea 11**.

En cuanto a los eventos lanzados por el control Spinner, el más utilizado será el generado al seleccionarse una opción de la lista desplegable, `onItemSelected`. Para capturar este evento se procederá de forma similar a lo ya visto para otros controles anteriormente, asignándole su controlador mediante el método **`setOnItemSelectedListener()`**, en este caso lo hacemos mediante el método de derivar de la interfaz, **Línea 12**.

En el código inferior vemos implementado el método, que muestra un Toast con el color elegido.

```
override fun onItemSelected(adapterView: AdapterView<*>?,
    view: View?, i: Int, l: Long){
    val x = Toast.makeText(this, "El color elegido es: " +
        colores[i], Toast.LENGTH_LONG)
    x.show()
}
```

Nota ampliativa

Una alternativa a tener en cuenta, si los datos a mostrar en el control son estáticos, sería definir la lista de posibles valores como un recurso de tipo string-array. Para ello, primero crearíamos un nuevo fichero XML en la carpeta **res/values** llamado por

ejemplo spinner.xml e incluiríamos en él los valores seleccionables de la siguiente forma:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string-array name="colores">
        <item>ROJO</item>
        <item>VERDE</item>
        <item>AZUL</item>
    </string-array>
</resources>
```

En este caso, a la hora de crear el adaptador utilizaríamos el método `createFromResource()` para hacer referencia al array XML que acabamos de crear:

```
val adaptador=ArrayAdapter.createFromResource(this,R.array.colores,
                                              android.R.layout.simple_list_item_1)
```

Ejercicio Propuesto Spinner

RecyclerView

Los **RecyclerView** son los elementos que proporciona Android para mostrar cantidades de datos, normalmente en forma de listas. Esta librería permite adaptar, de forma sencilla, la salida al gusto del diseñado, además de mejorar el rendimiento y la respuesta de la aplicación con respecto a las anteriores soluciones. Esto es debido a que, con este diseño, cuando un elemento se desplaza fuera de la pantalla no se destruye su vista, sino que se reutiliza para los elementos nuevos que ahora se muestran en ella.

RecyclerView 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

De igual forma que hemos hecho con el **Spinner**, 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 es que este

tipo de adaptador **obliga** a utilizar un `RecyclerView.ViewHolder` (elemento que permite optimizar el acceso a los datos del adaptador).

Una vista de tipo RecyclerView 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:

- `LinearLayoutManager` para la visualización como lista vertical u horizontal,
- `GridLayoutManager` para la visualización como tabla tradicional ()
- `StaggeredGridLayoutManager` que visualiza los elementos como una tabla apilada o de celdas no alineadas.

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.

Los dos últimos componentes de la lista `ItemDecoration` e `ItemAnimator`, se encargarán de definir cómo se representarán algunos aspectos visuales 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: *añadir o eliminar elementos*.

De todas formas, 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 Item Decoration e Item Animator necesarios para dar un toque de personalización especial a nuestra aplicación.

📦 Vamos a crear un ejemplo que muestre un recycler con la lista de una serie de usuarios. Para ello deberemos crear un nuevo proyecto **EjemploRecycler**. Lo primero que tendremos que hacer es crear una clase *pojo* que nos sirva para los datos de los usuarios. Por ejemplo **Usuario.kt** de la siguiente manera:

```
class Usuario(nombre:String, apellidos:String) {
    var nombre: String
    var apellidos: String
    init {
        this.nombre = nombre
        this.apellidos = apellidos
    }
}
```

El siguiente paso sería añadir el recycler al lugar donde queremos que sea mostrado, por ejemplo a la actividad principal. En este caso podría quedar la **main_activity.xml** de la siguiente manera:

```
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">
    <androidx.recyclerview.widget.RecyclerView
        android:id="@+id/recyclerList"
        android:background="@color/azul"
        android:layout_width="match_parent"
        android:layout_height="match_parent"/>
</androidx.constraintlayout.widget.ConstraintLayout>
```

También tenemos que pensar cual es el diseño que deseamos para cada uno de los elementos del recycler, esto se hace en un recurso layout y de la forma que ya sabemos. Por ejemplo podría ser el archivo **recyclerlayout.xml**:

```
0 <?xml version="1.0" encoding="utf-8"?>
<androidx.cardview.widget.CardView
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:card_view="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    card_view:cardCornerRadius="4dp"
    card_view:cardUseCompatPadding="true"
    card_view:cardElevation="2dp">
    <LinearLayout
        android:padding="8dp"
        android:background="#493DEC"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:gravity="center"
        android:orientation="horizontal">
```

```

<LinearLayout
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_weight="0.75"
    android:orientation="vertical">
    <TextView
        android:id="@+id/textView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Large Text"
        android:textColor="@android:color/white"
        android:textSize="20sp" />
    <TextView
        android:id="@+id/textView2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Medium Text"
        android:textColor="@android:color/white"
        android:textSize="15sp" />
</LinearLayout>
</LinearLayout>
</androidx.cardview.widget.CardView>

```

✋ Para conseguir una separación entre los elementos de la lista, se puede usar el contenedor **CardView** que ya conocemos de temas anteriores. **RecyclerView** no tiene la separación por elementos creada por defecto, se tendría que simular de distintas maneras, una es usando tarjetas como en este caso.

El siguiente paso sería escribir nuestro adaptador. Este adaptador deberá extender de 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.

El método **onCreateViewHolder** devuelve un objeto de tipo **Holder**, por lo que primero deberemos crear una clase que herede de **RecyclerView.ViewHolder**, que podría ser como la siguiente, **Holder.kt**:

```

0  class Holder(v: View) : RecyclerView.ViewHolder(v) {
    val textNombre: TextView
    val textApellido: TextView

    fun bind(entity: Usuario) {
6      textNombre.setText(entity.nombre)
7      textApellido.setText(entity.apellidos)
    }
    init {
10     textNombre = v.findViewById(R.id.textView)
11     textApellido = v.findViewById(R.id.textView2)
    }
}

```

👉 El patrón **ViewHolder** tiene un único propósito en su implementación, mejorar el rendimiento. Y es que cuando hablamos de aplicaciones móviles, el rendimiento es un tema de gran importancia. El ViewHolder mantiene una referencia a los elementos del RecyclerView mientras el usuario realiza scrolling en la aplicación. Así se evitan las frecuentes llamadas a `findViewById`, realizándola la primera vez y el resto usando la referencia en el ViewHolder.

✎ **Línea 0** se definirá como otra clase extendida de la clase **RecyclerView.ViewHolder**, y será bastante sencilla, 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 inflarlas en el constructor sobre la vista recibida como parámetro **Líneas 10 y 11**. Añadiremos también un método, normalmente llamado **bind()**, que se encargue de asignar los contenidos a los dos textos a partir del objeto `Usuario`, **Líneas 6 y 7**.

Ahora ya podemos crear el adaptador, que en nuestro proyecto podría ser **Adaptador.kt** que herede de **RecyclerView.Adapter** y que nos obligará a sobrescribir los métodos que sean necesarios, quedando el código como vemos en la imagen:


```

class Adaptador internal constructor(val datos: ArrayList<Usuario>) :
    RecyclerView.Adapter<Holder>()
{
    override fun onCreateViewHolder(viewGroup: ViewGroup, i: Int):Holder
    {
        6      val itemView: View = LayoutInflater.from(viewGroup.context)
              .inflate(R.layout.recyclerlayout, viewGroup, false)
        8      return Holder(itemView)
    }
    override fun onBindViewHolder(holder: Holder, position: Int) {
        val item: Usuario = datos[position]
        holder.bind(item)
    }
    override fun getItemCount(): Int {
        return datos.size
    }
}

```

✂ Como se puede ver en la imagen, en el método **onCreateViewHolder** nos limitaremos a inflar una vista a partir del layout correspondiente a los elementos de la lista (recyclerlayout.xml), y crear y devolver un nuevo ViewHolder llamando al constructor de nuestra clase Holder pasándole dicha vista como parámetro.

Línea 6 y 8.

Los dos métodos restantes son aún más sencillos. En **onBindViewHolder()** tan sólo tendremos que recuperar el objeto correspondiente a la posición recibida como parámetro y llamar al método bind de nuestro Holder, pero siempre desde el objeto ViewHolder recibido como parámetro. Por su parte, **getItemCount()** devolverá el tamaño de la lista de datos.

Con esto tendríamos finalizado el adaptador, por lo que ya podríamos asignarlo al RecyclerView en nuestra actividad principal. Lo haremos con el siguiente código

```

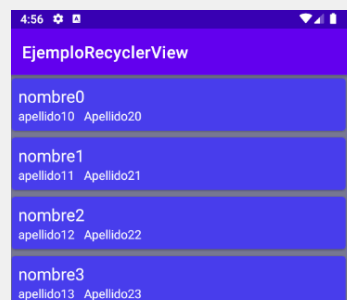
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?)
    {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        val datos = anadirDatos()
        val recyclerView = findViewById<RecyclerView>(R.id.recyclerList)
        8 val adaptador = Adaptador(datos)
        9 recyclerView.adapter = adaptador
        10 recyclerView.layoutManager =
            LinearLayoutManager(this, LinearLayoutManager.VERTICAL, false)
    }
    private fun anadirDatos():ArrayList<Usuario>
    {
        var datos = ArrayList<Usuario>()
        for (i in 0..19)
            datos.add(Usuario("nombre$i", "apellido1$i", "apellido2$i"))
        return datos
    }
}

```

✎ Instanciamos un objeto de la clase **Adatador** creada por nosotros **Línea 8**. Se lo añadiremos al recyclerView inflado en la actividad principal **Línea 9** y le asociaremos un LayoutManager determinado, para obtener la forma en la que se distribuirán los datos en pantalla.

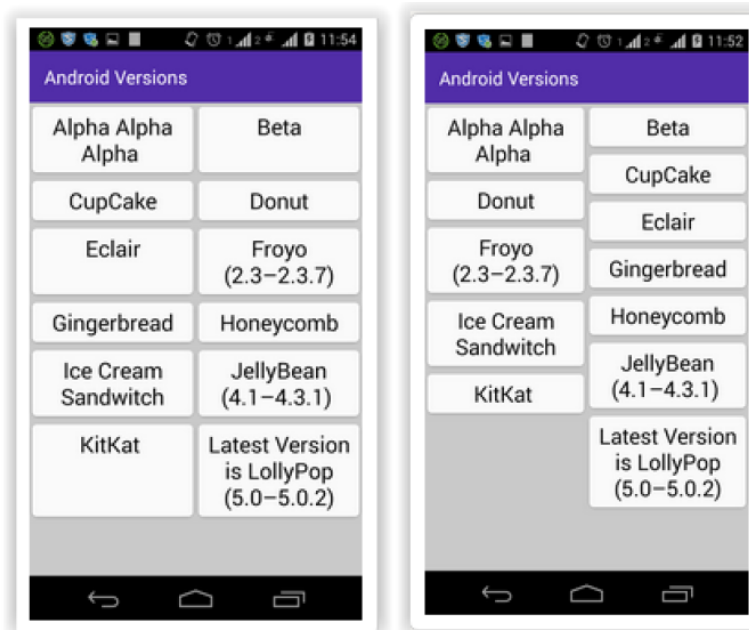
Como ya dijimos, si nuestra intención es mostrar los datos en forma de lista o tabla (al estilo de los antiguos ListView o GridView) no tendremos que implementar nuestro propio LayoutManager. En nuestro caso particular queremos mostrar los datos en forma de lista con desplazamiento vertical **Línea 10**. 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()** .

Al final conseguiremos un efecto parecido al siguiente, que muestra la lista de datos y sobre el que se puede hacer scrolling:



GridLayoutManager y StaggeredGridLayoutManager

Otra parte positiva de la clase RecyclerView, 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. Si lo que queremos es mostrarla de forma tabular pero sin igualar las celdas usaremos el **StaggeredGridLayoutManager**, el tercer argumento dispone si tiene que dimensionar las celdas a su contenido o no.



```
recyclerView.setLayoutManager(new GridLayoutManager(this, 2));
```

```
recyclerView.setLayoutManager(new StaggeredGridLayoutManager(2,1 ))
```

ItemDecoration e ItemAnimation

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**.

Por último hablemos muy brevemente de **ItemAnimator**, 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.

Click sobre un elemento de la lista

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 ocurre con `ListFragment`. 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, es decir al adaptador.

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(). Es más fácil verlo sobre el código:

```
class Adaptador internal constructor(val datos: ArrayList<Usuario>) :  
2    RecyclerView.Adapter<Holder>(), View.OnClickListener  
{  
4    lateinit var listenerClick: View.OnClickListener;  
    override fun onCreateViewHolder(viewGroup: ViewGroup, i: Int): Holder {  
        val itemView: View = LayoutInflater.from(viewGroup.context)  
            .inflate(R.layout.recyclerlayout, viewGroup, false)  
8        itemView.setOnClickListener(this)  
        return Holder(itemView)  
    }  
    override fun onBindViewHolder(holder: Holder, position: Int) {  
        val item: Usuario = datos[position]  
        holder.bind(item)  
    }  
    override fun getItemCount(): Int {  
        return datos.size  
    }  
18    fun onClick(listener: View.OnClickListener) {  
        this.listenerClick = listener  
    }  
21    override fun onClick(p0: View?) {  
        listenerClick?.onClick(p0)  
    }  
}
```

✍ Para implementar el evento onClick sobre un elemento de la lista, primero hacemos que el adaptador creado por nosotros herede de la interface **OnClickListener** **Línea 2**. Para conseguir que se detecte la pulsación sobre la vista del elemento, ponemos el escuchador sobre la vista **Línea 8**. Al heredar de

la interface, deberemos anular el onClick **Línea 21**, en este método lanzaremos el evento onClick de una propiedad de este tipo que nos habremos declarado antes **Línea 4**. Para que esta propiedad no sea nula, tendremos que crear un método al que le llegue una variable de este tipo y le sea asignada **Línea 18**. Este último método será al que tendremos que llamar desde donde queramos detectar el evento. Esto lo podemos ver en el código de la **MainActivity**.

```
0 adaptador.onClick(View.OnClickListener { v ->
    Toast.makeText(
        this@MainActivity,
        "Has pulsado" + recyclerView.getChildAdapterPosition(v),
        Toast.LENGTH_SHORT
    ).show()
})
```

✎ Con el objeto adaptador asignado al recycler podemos llamar a la función onClick y pasar un anónimo de tipo OnClickListener, que será invocado al pulsar sobre un elemento de la lista. Con la vista que entra podemos saber que posición a sido pulsada a través del método `getChildAdapterPosition()` de la clase RecyclerView.

✎ **Reconstruye el ejemplo y añade la funcionalidad del Click largo para eliminar el elemento pulsado de la lista. Para actualizar el adaptador. una vez eliminado puedes usar, `adaptador.notifyItemRemoved(posicioneliminada)`**

Click en cualquier lugar de la vista

Si quisiéramos detectar la pulsación de cualquier elemento de la línea del recycler, deberemos actuar sobre esa vista en el Holder (es donde podemos hacer referencia a cada uno de los view del layout). La forma de hacerlo es igual como se ha hecho anteriormente, usando un listener de la interfaz que nos haga falta y mandando la información a través de esta.

Para ver el funcionamiento, primero vamos a incluir una imagen en el layout **recyclerlayout.xml**, para que al pulsar sobre esa imagen se abra el dial del teléfono. Podríamos añadir la imagen de la siguiente manera:

```

<androidx.cardview.widget.CardView>
    ...
    <ImageView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/imagen"
        android:src="@android:drawable/ic_menu_call"/>
    </LinearLayout>
</androidx.cardview.widget.CardView>

```

Ahora añadiremos en el código de la clase Holder (aquí es donde se sabe que vista se ha pulsado de entre todas), la llamada al intent que abre el diálogo:

```

1  class Holder(v: View, context: Context) : RecyclerView.ViewHolder(v),
2                                     View.OnClickListener {
    val textNombre: TextView
    val textApellido: TextView
    val context: Context
    val imagen: ImageView
    fun bind(entity: Usuario) {
        textNombre.setText(entity.nombre)
        textApellido.setText(entity.apellidos)
    }
    init {
12         this.context=context
        textNombre = v.findViewById(R.id.textview)
        textApellido = v.findViewById(R.id.textview2)
        imagen=v.findViewById(R.id.imagen)
16         imagen.setOnClickListener(this)
    }
18     override fun onClick(p0: View?) {
        val i = Intent(Intent.ACTION_DIAL)
        startActivity(context,i,null)
    }
}

```

✎ En este caso, como no se quiere pasar información desde el Holder hasta el Adaptador o la actividad principal, lo único que tenemos que anular (habremos heredado de OnClickListener) es el método onClick **Fila 2 y 18**, donde lanzaremos el intent. Para lanzar el intent necesitaremos el contexto, por lo que se lo pasaremos mediante el constructor del Holder **Líneas 1 y 12**. No olvides poner el escuchador sobre la vista que te interese **Línea 16**. Al constructor del adaptador también le tendrá que llegar, de la misma manera, el contexto de la Actividad.

Click en cualquier lugar de la vista pasando información a la Actividad Principal

En este caso además de detectar la pulsación sobre un elemento, tendremos que devolver la información que nos interese hacia atrás. Se podría usar un ViewModel, como ya hemos explicado y usado con anterioridad, pero vamos a utilizar una Interface para recordar el sistema.

```
class Holder(v: View) : RecyclerView.ViewHolder(v),
    View.OnClickListener {

    val textNombre: TextView
    val textApellido: TextView
5    lateinit var pasarCadenaInterface: PasarCadenaInterface
    fun bind(entity: Usuario) {
        textNombre.setText(entity.nombre)
        textApellido.setText(entity.apellidos)
    }
    init {
        textNombre = v.findViewById(R.id.textView)
        textApellido = v.findViewById(R.id.textView2)
13        textNombre.setOnClickListener(this)
14        textApellido.setOnClickListener(this)
    }
    override fun onClick(p0: View?) {
        var cadena:String
        if(p0?.id==R.id.textView) cadena=textNombre.text.toString()
        else cadena=textApellido.text.toString()
20        pasarCadenaInterface.pasarCadena(cadena)
    }
    fun pasarCadena(pasarCadenaInterface: PasarCadenaInterface)
    {
24        this.pasarCadenaInterface=pasarCadenaInterface
    }
}
```

✧ **Línea 5** creamos un objeto del tipo de Interface creada, en este caso sería así:

```
interface PasarCadenaInterface{
    fun pasarCadena(cadena:String)
}
```

Línea 13 y 14 ponemos escuchadores del evento Click sobre las vistas que necesitemos. Al anular el OnClick, llamamos al método de la interface con el texto que deseamos **Línea 20**. Para que la interface no sea nula, seguimos la estrategia que ya conocemos del anterior punto, creamos un método al que le

llega la interface **Línea 24**.

En el **Adaptador** tendremos que hacer los siguientes cambios:

```
class Adaptador internal constructor(val datos: ArrayList<Usuario>) :
    RecyclerView.Adapter<Holder>(), View.OnClickListener,
    View.OnLongClickListener{
    ...
5    lateinit var pasarCadenaInterface: PasarCadenaInterface
    override fun onCreateViewHolder(viewGroup: ViewGroup, i: Int): Holder{
        ...
        val holder = Holder(itemView)
        holder.pasarCadena(object : PasarCadenaInterface{
10            override fun pasarCadena(cadena: String) {
                pasarCadenaInterface.pasarCadena(cadena)
            }
        })
        return holder
    }
    ...
17 fun pasarCadena(pasarCadenaInterface: PasarCadenaInterface)
    {
        this.pasarCadenaInterface = pasarCadenaInterface
    }
}
```

✂ Creamos la propiedad de tipo PasarCadenaInterface **Línea 5** con el **holder** llamamos al método creado para pasar la instancia al holder. Cuando esta se ejecute, nos llegará la cadena y llamaremos a su vez al método para que de esa forma lleguen los datos a la MainActivity **Línea 10**. No olvidar el método al que nos llega la interface para que no sea nula **Línea 17**. En la **Main** tendremos que llamar a este último método de la misma forma que en el adaptador.

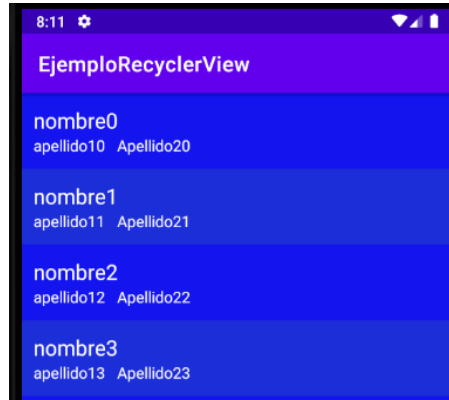
```
adaptador.pasarCadena(object : PasarCadenaInterface {
    override fun pasarCadena(cadena: String) {
        Toast.makeText(
            applicationContext,
            "Has pulsado " + cadena,
            Toast.LENGTH_SHORT ).show() }
    })
```

✍ **Reconstruye los ejemplos y comprueba su funcionamiento, con control de Click en la imagen y en los dos textos nombre y apellido. En estos últimos que llegue la información a la Main y que se muestre con un Toast**

Otros efectos sobre el Recycler

Cambiar el color de las líneas

Actuando sobre el Holder podemos conseguir otro tipo de efectos, como por ejemplo cambiar el color de las líneas pares y de las impares de la lista.

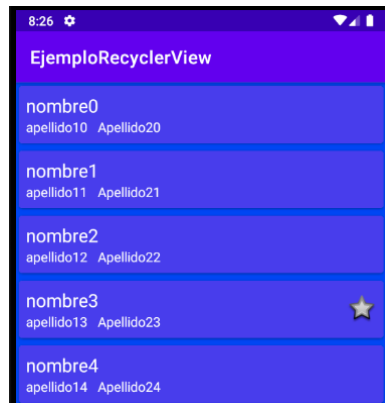


```
class Holder(v: View, context: Context) : RecyclerView.ViewHolder(v) {  
    ...  
4    fun bind(entity: Usuario, pos:Int) {  
        textNombre.setText(entity.nombre)  
        textApellido.setText(entity.apellidos)  
7        if (pos % 2 == 0) v.setBackgroundColor(  
            ContextCompat.getColor(context, R.color.oscuro))  
9        else v.setBackgroundColor(  
            ContextCompat.getColor(context, R.color.claro))  
    }  
    ...  
}
```

✎ Para conseguir ese efecto solo se ha tenido que controlar que posición se está cargando y dependiendo de si es par o no, se colorea de un color u otro la vista del CardView **Línea de 7 a 9**. Al constructor le hemos tenido que pasar el contexto para poder usar el método `getColor` y al método **bind** le hemos pasado la posición del elemento que se carga **Línea 4**.

Hacer visible algún elemento

Por otro lado podríamos conseguir el efecto de añadir algún elemento si se cumple determinada condición por ejemplo, en este caso lo que hacemos es mostrar una imagen, que en principio está oculta en el layout de la línea del recycler, en el caso de que el nombre contenga un 3. El código también tendría que ir en el bind del Holder.



```
class Holder(v: View, context: Context) : RecyclerView.ViewHolder(v) {  
    ...  
4    fun bind(entity: Usuario, pos: Int) {  
        textNombre.setText(entity.nombre)  
        textApellido.setText(entity.apellidos)  
7        if (entity.nombre?.contains("3")==true)  
            imagen.setVisibility(View.VISIBLE)  
9        else imagen.setVisibility(View.INVISIBLE)  
    }  
    ...  
}
```

Detectar swipe izq/der sobre un elemento del recycler

Una opción muy utilizada en las listas, es la de controlar el deslizamiento a la izquierda o hacia la derecha sobre uno de sus elementos. Para ello tendremos que usar una clase que ya está definida y que tendremos que copiar en nuestro proyecto, llamada **SwipeDetector** (se pasa en los recursos). En el Adaptador tendríamos que implementar la interfaz onTouch, para que nos detecte el desplazamiento, esto lo haremos como lo hicimos en puntos anteriores. **Líneas marcadas en el Adaptador.**

```

class Adaptador internal constructor(val datos: ArrayList<Usuario>) :
    RecyclerView.Adapter<Holder>(), View.OnClickListener,
3    View.OnLongClickListener, View.OnTouchListener
{
    lateinit var listenerClick: View.OnClickListener;
    lateinit var listenerLong: View.OnLongClickListener
7    lateinit var listenerOnTouch: View.OnTouchListener

    override fun onCreateViewHolder(viewGroup: ViewGroup, i: Int): Holder {
        val itemView: View = LayoutInflater.from(viewGroup.context)
            .inflate(R.layout.recyclerlayout, viewGroup, false)
        itemView.setOnClickListener(this)
        itemView.setOnLongClickListener(this)
14        itemView.setOnTouchListener(this)
        val holder=Holder(itemView)
        return holder
    }
18    fun onTouch(listenerOnTouch: View.OnTouchListener)
    {
        this.listenerOnTouch=listenerOnTouch
    }
    override fun onTouch(p0: View?, p1: MotionEvent?): Boolean {
        listenerOnTouch.onTouch(p0,p1)
        return false
25    }
}

```

En la ActivityMain, tendremos que llamar al método **OnTouch** del adaptador pasándole un objeto de la clase **SwipeDetector** **Línea 2**, pero a la hora de detectar el movimiento tendremos que usar la interfaz **onClikListener** **Línea 3**, como vemos en el código siguiente. Si nos fijamos, podremos ver que el objeto **swipeDetector** detectar si el movimiento se ha producido a la derecha o a la izquierda.

```

val swipeDetector = SwipeDetector()
2 adaptador.onTouch(swipeDetector)
3 adaptador.onClick(View.OnClickListener { v ->
    if (swipeDetector.swipeDetected()) {
        when (swipeDetector.action) {
            SwipeDetector.Action.LR -> {
                Toast.makeText(
                    applicationContext,
                    "Has pulsado Izquierda",
                    Toast.LENGTH_SHORT
                ).show()
            }
            SwipeDetector.Action.RL -> {
                Toast.makeText(
                    applicationContext,
                    "Has pulsado Derecha",
                    Toast.LENGTH_SHORT
                ).show()
            }
        }
    } else
        Toast.makeText(
            applicationContext,
            "Has pulsado" + recyclerView.getChildAdapterPosition(v),
            Toast.LENGTH_SHORT
        ).show()
    })

```

Selección Múltiple en el RecyclerView

Android aporta la biblioteca de `recyclerview-selection` para [seleccionar más de un elemento de una lista](#), creada con un `recycler`. Es decir la acción que ocurre en la mayoría de listas, cuando realizamos un click largo y después nos deja seleccionar elementos. Para poder implementar esta selección, tendremos que tener creado un `RecyclerView` en perfecto funcionamiento, a partir de aquí deberemos de seguir una serie de pasos:

1. Implementar **ItemDetailsLookup**. Este elemento permite que la biblioteca de selección acceda a la información sobre los elementos de `RecyclerView` a los que se otorga un `MotionEvent` (alguna acción de movimiento: pulsación, arrastre, etc.). Para construirlo necesitará la información del elemento pulsado, que se puede extraer del `Holder`. Para ello lo primero que haremos será crear en el **Holder.Kt**, un método al que se pueda llamar para identificar de forma exclusiva el elemento de la lista pulsado. Este método tiene que devolver una instancia de la clase `ItemDetailsLookup.ItemDetails`.

```

fun getItemDetails(): ItemDetailsLookup.ItemDetails<Long> =
    object: ItemDetailsLookup.ItemDetails<Long>() {
        override fun getPosition(): Int=adapterPosition
        override fun getSelectionKey(): Long? =itemId
    }

```

Y sobrescribir los dos métodos de la clase abstracta para identificar de forma única el elemento sobre el que se ha realizado la acción. Para ello podemos usar las propiedades `adapterPosition` e `itemId`, que posee la clase `ViewHolder`. Como se puede ver en el código, hemos hecho de tipo `Long` la clase genérica `ItemDetails`. El long determina el tipo de clave de selección que usaremos. Hay tres tipos de claves que se pueden usar para identificar elementos seleccionados: *Parcelable*, *String* y *Long*. Ahora pasaremos a implementar la clase, por ejemplo **LookUp.kt**, que herede de `ItemDetailsLookup` del tipo de la clave seleccionada. En el método que anulamos, se captura la vista seleccionada y con ella accedemos al método creado con anterioridad en el Holder.

```

0 class LookUp (private val rv: RecyclerView)
  : ItemDetailsLookup<Long>() {
  override fun getItemDetails(event: MotionEvent)
    : ItemDetails<Long>? {
    val view = rv.findViewById(event.x, event.y)
    if(view != null) {
        return (rv.getChildViewHolder(view) as Holder).getItemDetails()
    }
    return null}
  }

```

2. Actualiza las Views de los elementos de RecyclerView para que refleje si el usuario realizó o no una selección. Estos cambios se pueden hacer en el Holder, como hemos visto anteriormente, aunque en este caso tendremos que tener en cuenta si el elemento ha sido seleccionado. Para ello usamos un tipo `SelectionTracker` que pasaremos como argumento desde el Adaptador. Se aconseja que se muestre los elementos seleccionados mediante un cambio de color.

```

fun bind(entity: Usuario, tracker: SelectionTracker<Long>?,) {
    textNombre.setText(entity.nombre)
    textApellido.setText(entity.apellidos)
    if(tracker!!.isSelected(adapterPosition.toLong()))
        cardView.background = ColorDrawable(Color.CYAN)
    else cardView.background= ColorDrawable(Color.LTGRAY)
}

```

3. Unir los anteriores pasos mediante un objeto de tipo `SelectionTracker.Builder`

Línea 6 a 14, creado en la actividad principal y pasado al adaptador, que a su vez lo mandará al Holder. Este elemento inicializa el rastreador de selección. A su constructor se le pasa: el ID de selección (una cadena única), la instancia del RecyclerView, el proveedor de claves (se puede elegir entre varios de la BCL), la clase creada en el paso uno para el control del elemento pulsado y la estrategia de almacenamiento en este caso de tipo Long como la clave (esta instancia permite asegurar que no se pierden los elementos seleccionados en los cambios de estado del dispositivo).

```
1  recyclerView.setHasFixedSize(true)
   val adaptador = Adaptador(datos, this)
3  adaptador.setHasStableIds(true) //Antes de asignar
                                   //el adaptador al recycler
   recyclerView.adapter = adaptador
6  tracker = SelectionTracker.Builder<Long>(
      "selecccion",
      recyclerView,
      StableIdKeyProvider(recyclerView),
      LookUp(recyclerView),
      StorageStrategy.createLongStorage()
    ).withSelectionPredicate(
      SelectionPredicates.createSelectAnything()
14     ).build()
15  adaptador.setTracker(tracker) //método que crearemos en el adaptador
```

Además se deberá fijar el tamaño del recycler **Línea 1**, asignar al adaptador la propiedad de HasStableIds a true **Línea 3**. y pasar la instancia de Tracker al adaptador mediante un método que crearemos en este **línea 15**.

Para controlar con más seguridad la estabilidad de la app, se debería de anular el método `onSaveInstanceState` para pasar la información capturada, a la instancia del `tracker`.

```
override fun onSaveInstanceState(outState: Bundle) {
    super.onSaveInstanceState(outState)
    if(outState != null)
        tracker?.onSaveInstanceState(outState)
}
```

Y al iniciar la aplicación añadir la siguiente línea para recuperar el tracker guardado:

```
if(savedInstanceState != null)
    tracker?.onRestoreInstanceState(savedInstanceState)
```

Por su parte en el adaptador solo tendremos estos dos cambios.

```

override fun onBindViewHolder(holder: Holder, position: Int) {
    val item: Usuario = datos[position]
    holder.bind(item, tracker) }
fun setTracker(tracker: SelectionTracker<Long>?) {
    this.tracker = tracker
}

```

Después de realizar estos pasos, podemos probar el funcionamiento de la aplicación.


Reconstruye el ejemplo y prueba el funcionamiento

4. Solo nos queda controlar la pulsación y realizar alguna cosa cuando esto ocurra. Tendremos que añadir un observador al Tracker y para ello se utiliza un objeto **SelectionTracker.SelectionObserver**, que se encarga de registrar los cambios. Dentro de este objeto pondremos el código que creamos conveniente, en este caso solamente mostramos un Toast con el id de los elementos que están en la selección.

```

tracker?.addObserver(
    object: SelectionTracker.SelectionObserver<Long>() {
        override fun onSelectionChanged() {
            if (tracker!!.hasSelection()) {
                var cadena=StringBuilder()
                tracker?.selection?.forEach { id->
                    cadena.append(id.toString() + "\n") }
                Toast.makeText(this@MainActivity, cadena,
                    Toast.LENGTH_SHORT).show()
            }
        }
    })

```

 El comportamiento esperado cuando se hace una selección multiple, es el de que se superponga una toolbar sobre la ActionBar con un menú específico para las acciones que puedan ocurrir sobre la selección. Para ello se deberá implementar la clase **ActionMode** que será activada cuando el observador detecte que tiene selección. Esta implementación se verá en el tema de menús