

Apuntes

[Descargar estos apuntes](./Resumen Interfaz de Usuario RecyclerView.pdf)

Resumen Interfaz de Usuario RecyclerView

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 Multiple en el Recycler](#)

Spinner

Spinners.

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

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

```
class MainActivity : AppCompatActivity(),
    AdapterView.OnItemClickListener {
    var colores = arrayOf("Rojo", "Verde", "Azul")
    lateinit var listaColores: Spinner
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        //Si tuviera que cargar colores desde un xml
        //colores = resources.getStringArray(R.array.colores)
        val adaptador = ArrayAdapter(this, android.R.layout.
            simple_list_item_1, colores)
        listaColores = findViewById(R.id.spinner)
        listaColores.adapter=adaptador
        listaColores.setOnItemClickListener(this)
    }
}
```

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

Podríamos definir los valores en un xml:

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

RecyclerView

[RecyclerView]

Lo más habitual será implementar el **Adapter** y el **ViewHolder**, utilizar alguno de los **LayoutManager** predefinidos

Añadir el recycler al lugar donde queremos que sea mostrado:

```
<androidx.recyclerview.widget.RecyclerView
    android:id="@+id/recyclerList"
    android:background="@color/azul"
    android:layout_width="match_parent"
    android:layout_height="match_parent"/>
```

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, por ejemplo **recyclerlayout.xml**:

👉 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**:

```

class Holder(v: View) : RecyclerView.ViewHolder(v) {
    val textNombre: TextView
    val textApellido: TextView
    private lateinit var binding: RecyclerViewlayoutBinding

    init {
        binding = RecyclerViewlayoutBinding.bind(v)
        textNombre = binding.textView
        textApellido = binding.textView2
    }

    fun bind(entity: Usuario) {
        textNombre.setText(entity.nombre)
        textApellido.setText(entity.apellidos)
    }
}

```

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(val datos: ArrayList<Usuario>) :
    RecyclerView.Adapter<Holder>()
{
    override fun onCreateViewHolder(viewGroup: ViewGroup,
                                     i: Int): Holder
    {
        val itemView: View =
            LayoutInflater.from(viewGroup.context)
                .inflate(R.layout.recyclerviewlayout, viewGroup, false)
        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
    }
}

```

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() {
    private lateinit var binding: ActivityMainBinding
    override fun onCreate(savedInstanceState: Bundle?)
    {
        super.onCreate(savedInstanceState)
        binding = ActivityMainBinding.inflate(layoutInflater)
        val view = binding.root
        setContentView(view)

        val datos = anadirDatos()
        val recyclerView = binding.mirecycler
        val adaptador = Adaptador(datos)
        recyclerView.adapter = adaptador
        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
    }
}

```

Click sobre un elemento de la lista

Para sorpresa de todos, la clase RecyclerView no tiene incluye un evento `onItemClickListener()`. RecyclerView delegará esta tarea 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()`.

```

class Adaptador internal constructor(val datos: ArrayList<Usuario>) :
    RecyclerView.Adapter<Holder>(), View.OnClickListener
{
    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)
        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
    }
    fun onClick(listener: View.OnClickListener) {
        this.listenerClick = listener
    }
    override fun onClick(p0: View?) {
        listenerClick?.onClick(p0)
    }
}

```

🔗 El método `onClick` será al que tendremos que llamar desde donde queramos detectar el evento. Esto lo podemos ver en el código de la **MainActivity**.

```

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

```

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:

```

class Holder(v: View, context: Context) : RecyclerView.ViewHolder(v),
    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 {
        this.context=context
        textNombre = v.findViewById(R.id.textview)
        textApellido = v.findViewById(R.id.textview2)
        imagen=v.findViewById(R.id.imagen)
        imagen.setOnClickListener(this)
    }
    override fun onClick(p0: View?) {
        val i = Intent(Intent.ACTION_DIAL)
        startActivity(context,i,null)
    }
}

```


✍ Para lanzar el intent necesitaremos el contexto, por lo que se lo pasaremos mediante el constructor del Holder. 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 **Línea5**. 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 **Línea10**. No olvidar el método al que nos llega la interface para que no sea nula **Línea17**. 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