

Apuntes

[Descarregar aquests apunts](#)

Tema 8. Interfície d'Usuari III

Índex

1. [Spinner](#)
2. [RecyclerView](#)
 1. [Clic sobre un element de la llista](#)
 2. [Clic en qualsevol lloc de la vista](#)
 3. [Clic en qualsevol lloc de la vista passant informació a l'Activitat Principal](#)
 4. [Altres efectes sobre el Recycler](#)
 5. [Selecció Múltiple en el Recycler](#)

Spinner

Les llistes desplegable en Android es diuen **Spinners** Spinner. És una llista que mostra els elements una vegada polsada la fletxa de desplegament, l'usuari pot seleccionar de la llista emergent un element. El codi de la vista d'un Spinner serà el següent:

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

Els elements que mostrarà el nostre Spinner no es poden indicar directament, sinó que és necessari crear una estructura d'elements que puguem associar a aquest en el nostre codi. Una de les maneres seria usant un Array de Strings per a després assignar-ho al spinner:

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

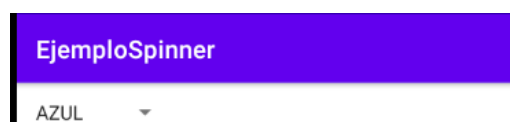
A continuació hem de definir l'adaptador, en el nostre cas un objecte ArrayAdapter, que serà l'objecte que lla llançarem amb el nostre Spinner.

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

Creem l'adaptador en si, al qual passem 3 paràmetres:

1. El context, que normalment serà simplement una referència a l'activitat on es crea l'adaptador.
2. L'ID del layout sobre el qual es mostraran les dades del control. En aquest cas li passem l'ID d'un layout predefinit en Android (android.R.layout.simple_spinner_item), format únicament per un control textView, però podríem passar l'ID de qualsevol layout del nostre projecte amb qualsevol estructura i conjunt de controls.
3. L'element que conté les dades a mostrar.

Amb aquestes simples línies tindrem un control similar al de la següent imatge:



📁 Anem a crea un projecte d'exemple anomenat Spinner al qual afegirem el següent codi en la **MainActivity.kt**, i no oblidem afegir un spinner amb `id=spinner*` en el layout **activity_main.xml**:

Amb aquestes simples línies tindrem un control similar al de la següent imatge:

```
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)
        8 val adaptador = ArrayAdapter(this, android.R.layout.
            simple_list_item_1, colores)
        listaColores = findViewById(R.id.spinner)
        11 listaColores.adapter=adaptador
        12 listaColores.setOnItemClickListener(this)
    }
```

🔗 **Línia 8** creem l'adaptador, com hem explicat anteriorment, per a assignar-lo al Spinner en la **Línia 11**.

Quant als esdeveniments llançats pel control Spinner, el més utilitzat serà el generat en seleccionar-se una opció de la llista desplegable, `onItemSelected`. Per a capturar aquest esdeveniment es procedirà de manera similar al ja vist per a altres controls anteriorment, assignant-li el seu controlador mitjançant el mètode `setOnItemSelectedListener()`, en aquest cas el fem mitjançant el mètode de derivar de la interfície, **Línia 12**.

En el codi inferior veiem implementat el mètode, que mostra un Toast amb el color triat.

```
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 tindre en compte, si les dades a mostrar en el control són estàtics, seria definir la llista de possibles valors com un recurs de tipus string-array. Per a això, primer crearíem un nou fitxer XML en la carpeta *res/values* anomenat per exemple *spinner.xml* i inclouríem en ell els valors seleccionables de la següent 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 aquest cas, a l'hora de crear l'adaptador utilitzaríem el mètode

`createFromResource()` per a fer referència al array XML que acabem de crear:

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

Ejercicio PropuestoSpinner

RecyclerView

Els [RecyclerView](#) són els elements que proporciona Android per a mostrar quantitats de dades, normalment en forma de llistes. Aquesta llibreria permet adaptar, de manera senzilla, l'eixida al gust del dissenyat, a més de millorar el rendiment i la resposta de l'aplicació respecte a les anteriors solucions. Això és pel fet que, amb aquest disseny, quan un element es desplaça fora de la pantalla no es destrueix la seua vista, sinó que es reutilitza per als elements nous que ara es mostren en ella.

RecyclerView se sustentarà sobre altres components complementaris per a determinar com accedir a les dades i com mostrar-los. Els més importants seran els següents:

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

D'igual forma que hem fet amb el `Spinner`, un `RecyclerView` es recolzarà també en un adaptador per a treballar amb les nostres dades, en aquest cas un adaptador que herete de la classe `RecyclerView.Adapter`. La peculiaritat és que aquest tipus d'adaptador **obliga*** a utilitzar un `RecyclerView.ViewHolder` (element que permet optimitzar l'accés a les dades de l'adaptador).

Una vista de tipus `RecyclerView` no determina, per si sola, la forma en què es mostraran en pantalla els elements de la nostra col·lecció, sinó que delegarà aqueixa tasca a un altre component anomenat `LayoutManager`, que també haurem de crear i associar al `RecyclerView` per al seu correcte funcionament. Per sort, el SDK incorpora de sèrie tres `LayoutManager` per a les tres representacions més habituals:

- `LinearLayoutManager` per a la visualització com a llista vertical o horitzontal,
- `GridLayoutManager` per a la visualització com a taula tradicional ()
- `StaggeredGridLayoutManager` que visualitza els elements com una taula apilada o de cel·les no alineades.

Per tant, sempre que optem per alguna d'aquestes distribucions d'elements no haurem de crear el nostre propi `LayoutManager` personalitzat, encara que per descomptat res ens impedeix fer-ho.

Els dos últims components de la llista `ItemDecoration` i `ItemAnimator`, s'encarregaran de definir com es representaran alguns aspectes visuals de la nostra col·lecció de dades (més enllà de la distribució definida pel `LayoutManager`), per exemple: *marcadors o separadors d'elements*, i de com s'animaran els elements en realitzar-se determinades accions sobre la col·lecció, per exemple: *en afegir o eliminar elements*.

De totes maneres, no sempre serà obligatori implementar tots aquests components per a fer ús d'un `RecyclerView`. **El més habitual serà implementar el Adapter i el ViewHolder, utilitzar algun dels LayoutManager predefinitos**, i només en cas de necessitat crear els `Item Decoration` i `Item Animator` necessaris per a donar un toc de personalització especial a la nostra aplicació.

📦 Crearem un exemple que mostre un recycler amb la llista d'una sèrie d'usuaris. Per a això haurem de crear un nou projecte **EjemploRecycler**. El primer que haurem de fer és crear una classe *pojo* que ens servisca per a les dades dels usuaris. Per exemple **Usuari.kt** de la següent manera:

```

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

```

El següent pas seria afegir el recycler al lloc on volem que siga mostrat, per exemple a l'activitat principal. En aquest cas podria quedar la **main_activity.xml** de la següent 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/recyclerView"
        android:background="@color/azul"
        android:layout_width="match_parent"
        android:layout_height="match_parent"/>
</androidx.constraintlayout.widget.ConstraintLayout>

```

També hem de pensar com és el disseny que desitgem per a cadascun dels elements del recycler, això es fa en un recurs layout i de la forma que ja sabem. Per exemple podria ser l'arxiu **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>

```

✋ Per a aconseguir una separació entre els elements de la llista, es pot usar el contenidor **CardView** que ja coneixem de temes anteriors. RecyclerView no té la separació per elements creada per defecte, s'hauria de simular de diferents maneres, una és usant targetes com en aquest cas.

El següent pas seria escriure el nostre adaptador. Aquest adaptador haurà d'estendre de la classe **RecyclerView.Adapter**, de la qual haurem de sobreescriure principalment tres mètodes:

- *onCreateViewHolder()* . Encarregat de crear els nous objectes ViewHolder necessaris per als elements de la col·lecció.
- *onBindViewHolder()* . Encarregat d'actualitzar les dades d'un ViewHolder ja existent.
- *onItemCount()* . Indica el nombre d'elements de la col·lecció de dades.

El mètode **onCreateViewHolder** retorna un objecte de tipus **Holder**, per la qual cosa primer haurem de crear una classe que herete de **RecyclerView.ViewHolder**, que podria ser com la següent, **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 patró **ViewHolder** té un únic propòsit en la seua implementació, millorar el rendiment. I és que quan parlem d'aplicacions mòbils, el rendiment és un tema de gran importància. El ViewHolder manté una referència als elements del RecyclerView mentre l'usuari realitza scrolling en l'aplicació. Així s'eviten les freqüents anomenades a `findViewById`, realitzant-la la primera vegada i la resta usant la referència en el ViewHolder.

🔗 **Línia 0** es definirà com una altra classe estesa de la classe **RecyclerView.ViewHolder**, i serà bastant senzilla, tan sols haurem d'incloure com a atributs les referències als controls del layout d'un element de la llista (en el nostre cas els dos `TextView`) i unflar-les en el constructor sobre la vista rebuda com a paràmetre **Línies 10 i 11**. Afegirem també un mètode, normalment anomenat **bind()**, que s'encarregue d'assignar els continguts als dos textos a partir de l'objecte `Usuari`, **Línies 6 i 7**.

Ara ja podem crear l'adaptador, que en el nostre projecte podria ser **Adaptador.kt** que herete de **RecyclerView.Adapter** i que ens obligarà a sobreescriure els mètodes que siguin necessaris, quedant el codi com veiem en la imatge:


```

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
    }
}

```

✂ Com es pot veure en la imatge, en el mètode **onCreateViewHolder** ens limitarem a unflar una vista a partir del layout corresponent als elements de la llista (recyclerlayout.xml), i crear i retornar un nou ViewHolder cridant al constructor de la nostra classe Holder passant-li aquesta vista com a paràmetre.

Línia 6 i 8.

Els dos mètodes restants són encara més senzills. En **onBindViewHolder()** tan sols haurem de recuperar l'objecte corresponent a la posició rebuda com a paràmetre i cridar al mètode bind del nostre Holder, però sempre des de l'objecte ViewHolder rebut com a paràmetre. Per part seua, **getItemCount()** retornarà la grandària de la llista de dades.

Amb això tindríem finalitzat l'adaptador, per la qual cosa ja podríem assignar-ho al RecyclerView en la nostra activitat principal. Ho farem amb el següent codi

```

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
    }
}

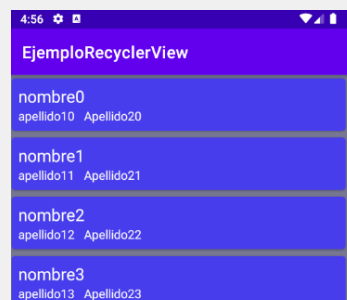
```

✎ Instanciam un objecte de la classe **Adaptador** creada per nosaltres **Línia 8**. Li ho afegirem al recyclerView unflat en l'activitat principal **Línia 9** i li associarem un LayoutManager determinat, per a obtindre la forma en la qual es distribuiran les dades en pantalla.

Com ja vam dir, si la nostra intenció és mostrar les dades en forma de llista o taula (a l'estil dels antics ListView o GridView) no haurem d'implementar el nostre propi LayoutManager. En el nostre cas particular volem mostrar les dades en forma de llista amb desplaçament vertical **Línia 10**. Per a això tenim disponible la classe LinearLayoutManager, per la qual cosa tan sols tindrem que instanciar un objecte d'aquesta classe indicant en el constructor l'orientació del desplaçament

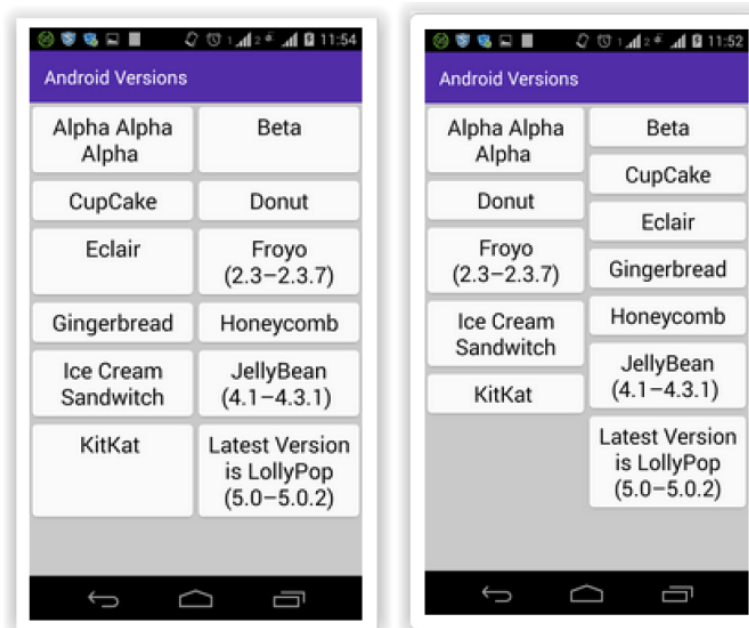
(**LinearLayoutManager.VERTICAL** o **LinearLayoutManager.HORITZONTAL**) i ho assignarem al RecyclerView mitjançant el mètode **setLayoutManager()** .

Al final aconseguirem un efecte semblant al següent, que mostra la llista de dades i sobre el que es pot fer scrolling:



GridLayoutManager y StaggeredGridLayoutManager

Una altra part positiva de la classe RecyclerView, és que si canviàrem d'idea i volguérem mostrar les dades de forma tabular tan sols hauríem de canviar l'assignació del LayoutManager anterior i utilitzar un **GridLayoutManager**, al qual passarem com a paràmetre el nombre de columnes a mostrar. Si el que volem és mostrar-la de manera tabular però sense igualar les cel·les usarem el **StaggeredGridLayoutManager**, el tercer argument disposa si ha de dimensionar les cel·les al seu contingut o no.



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

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

ItemDecoration e ItemAnimation

Els **ItemDecoration** ens serviran per a personalitzar l'aspecte d'un RecyclerView més enllà de la distribució dels elements en pantalla. L'exemple típic d'això són els separadors o divisors d'una llista. RecyclerView no té cap propietat dividir com en el cas del ListView, per la qual cosa aquesta funcionalitat hem de suplir-la amb un **ItemDecoration**.

Finalment parlem molt breument de **ItemAnimator**, permetrà definir les animacions que mostrarà el nostre RecyclerView en realitzar les accions més comunes sobre un element (afegir, eliminar, moure, modificar). Aquest component tampoc és senzill d'implementar, però per sort el SDK també proporciona una implementació per defecte que pot servir-nos en la majoria d'ocasions, encara que per descomptat podrem personalitzar creant el nostre propi ItemAnimator. Aquesta implementació per defecte es diu DefaultItemAnimator.

Clic sobre un element de la llista

El següent pas que ens podem plantejar és com respondre als esdeveniments que es produeixen sobre el RecyclerView, com a opció més habitual l'esdeveniment clic sobre un element de la llista. Per a sorpresa de tots, la classe RecyclerView no té inclou un esdeveniment onItemClick() com ocorre amb **ListFragment**. Una vegada més, RecyclerView delegarà també aquesta tasca a un altre component, en aquest cas a la pròpia vista que conforma cada element de la col·lecció, és a dir a l'adaptador.

Aprofitarem la creació de cada nou ViewHolder per a assignar a la seua vista associada l'esdeveniment onClick. Addicionalment, per a poder fer això des de fora de l'adaptador, inclourem el listener corresponent com a atribut de l'adaptador, i dins d'aquest ens limitarem a assignar l'esdeveniment a la vista del nou ViewHolder i a llançar-ho quan siga necessari des del mètode onClick(). És més fàcil veure-ho sobre el codi:

```
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)  
    }  
}
```

✎ Per a implementar l'esdeveniment onClick sobre un element de la llista, primer fem que l'adaptador creat per nosaltres herete de la interface **OnClickListener** **Línia 2**. Per a aconseguir que es detecte la pulsació sobre la

vista de l'element, posem el listener sobre la vista **Línia 8**. En heretar de la interface, haurem d'anul·lar el onClick **Línia 21**, en aquest mètode llançarem l'esdeveniment onClick d'una propietat d'aquest tipus que ens haurem declarat abans **Línia 4**. Perquè aquesta propietat no siga nul·la, haurem de crear un mètode al qual li arribi una variable d'aquest tipus i li siga assignada **Línia 18**. Aquest últim mètode serà al qual haurem de cridar des d'on vulguem detectar l'esdeveniment. Això el podem veure en el codi de la **MainActivity**.

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

✎ Amb l'objecte adaptador assignat al recycler podem cridar a la funció onClick i passar un anònim de tipus OnClickListener, que serà invocat en prémer sobre un element de la llista. Amb la vista que entra podem saber que posició a sigut premuda a través del mètode `getChildAdapterPosition()` de la classe RecyclerView.

✎ **Reconstrueix l'exemple i afeg la funcionalitat del Clic llarg per a eliminar l'element premut de la llista. Per a actualitzar l'adaptador, una vegada eliminat pots usar, `adaptador.notifyItemRemoved(posicioneliminada)`**

Clic en qualsevol lloc de la vista

Si volguérem detectar la pulsació de qualsevol element de la línia del recycler, haurem d'actuar sobre aqueixa vista en el Holder (és on podem fer referència a cadascun dels view del layout). La manera de fer-ho és igual com s'ha fet anteriorment, usant un listener de la interfície que ens faci falta i manant la informació a través d'aquesta.

Per a veure el funcionament, primer inclourem una imatge en el layout **recyclerlayout.xml**, perquè en prémer sobre aqueixa imatge s'obriga el dial del telèfon. Podríem afegir la imatge de la següent 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>

```

Ara afegirem en el codi de la classe Holder (ací és on se sap que vista s'ha premut d'entres totes), l'anomenada al intel que obri el díal:

```

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 aquest cas, com no es vol passar informació des del Holder fins a l'Adaptador o l'activitat principal, l'única cosa que hem d'anul·lar (haurem heretat de OnClickListener) és el mètode onClick **Fila 2 i 18**, on llançarem el intent. Per a llançar el intent necessitarem el context, per la qual cosa li ho passarem mitjançant el constructor del Holder **Línies 1 i 12**. No oblidis posar el listener sobre la vista que t'interesse **Línia 16**. Al constructor de l'adaptador també li haurà d'arribar, de la mateixa manera, el context de l'Activitat.

Clic en qualsevol lloc de la vista passant informació a l'Activitat Principal

En aquest cas a més de detectar la pulsació sobre un element, haurem de retornar la informació que ens interesse cap endarrere. Es podria usar un ViewModel, com ja hem explicat i usat amb anterioritat, però utilitzarem una Interface per a 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
    }
}
```

✧ **Línia 5** creem un objecte de la mena de Interface creada, en aquest cas seria així:

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

Línia 13 i 14 posem listeners de l'esdeveniment Clic sobre les vistes que necessitem. En anul·lar el OnClickListener, cridem al mètode de la interface amb el text que desitgem **Línia 20**. Perquè la interface no siga nul·la, seguim l'estratègia que ja coneixem de l'anterior punt, creguem un mètode al qual li arriba la

interface **Línia 24.**

En el **Adaptador** haurem de fer els següents canvis:

```
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
}
}
```

✂ Creem la propietat de tipus PasarCadenaInterface **Línea5**. amb el **holder** cridem al mètode creat per a passar la instància al holder. Quan aquesta s'execute, ens arribarà la cadena i cridarem al seu torn al mètode perquè d'aqueixa forma arriben les dades a la MainActivity **Línea10**. No oblidar el mètode al qual ens arriba la interface perquè no siga nul·la **Línea17**. En la **Main** haurem de cridar a aquest últim mètode de la mateixa forma que en l'adaptador.

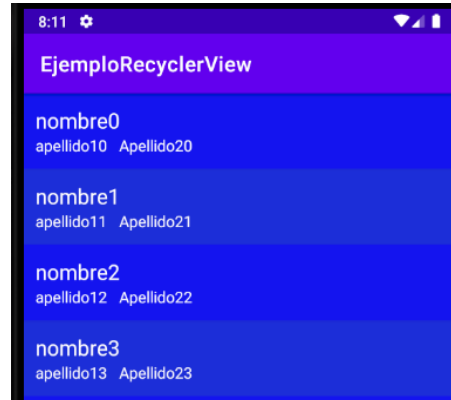
```
adaptador.pasarCadena(object : PasarCadenaInterface {
    override fun pasarCadena(cadena: String) {
        Toast.makeText(
            applicationContext,
            "Has pulsado " +cadena,
            Toast.LENGTH_SHORT ).show() }
    })
```

✍ **Reconstrueix els exemples i comprova el seu funcionament, amb control de Clic en la imatge i en els dos textos nomene i cognom. En aquests últims que arribe la informació a la Main i que es mostre amb un Toast**

Altres efectes sobre el Recycler

Canviar el color de les línies

Actuant sobre el Holder podem aconseguir un altre tipus d'efectes, com per exemple canviar el color de les línies pareixes i de les imparells de la llista.

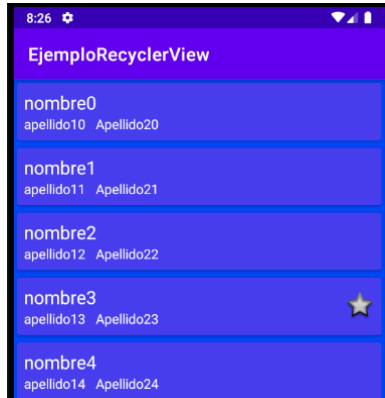


```
class Holder(v: View, context: Context) : RecyclerView.ViewHolder(v) {  
    ...  
4    fun bind(entity: Usuario, pos:Int) {  
        textNombre.setText(entity.nombre)  
        textApellido.setText(entity.apellidos)  
7        if (pos % 2 == 0) v.setBackgroundColor(  
            ContextCompat.getColor(context,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

✎ Per a aconseguir aqueix efecte només s'ha hagut de controlar que posició s'està carregant i depenent de si és parell o no, s'acolorix d'un color o un altre la vista del CardView **Línia de 7 a 9**. Al constructor li hem hagut de passar el context per a poder usar el mètode `*getColor` i al mètode **bind** li hem passat la posició de l'element que es carrega **Línia 4**.

Fer visible algun element

D'altra banda podríem aconseguir l'efecte d'afegir algun element si es compleix determinada condició per exemple, en aquest cas el que fem és mostrar una imatge, que en principi està oculta en el layout de la línia del recycler, en el cas que el nom continga un 3. El codi també hauria d'anar 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 element del recycler

Una opció molt utilitzada en les llistes, és la de controlar el lliscament a l'esquerra o cap a la dreta sobre un dels seus elements. Per a això haurem d'usar una classe que ja està definida i que haurem de copiar en el nostre projecte, anomenada

SwipeDetector (es passa en els recursos). En l'Adaptador hauríem d'implementar la interfície onTouch, perquè ens detecte el desplaçament, això ho farem com ho vam fer en punts anteriors. **Línies marcades en l'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, haurem de cridar al mètode **onTouch** de l'adaptador passant-li un objecte de la classe **SwipeDetector** **Línia 2**, però a l'hora de detectar el moviment haurem d'usar la interfície **onClickListener** **Línia 3**, com veiem en el codi següent. Si ens fixem, podrem veure que l'objecte **swipeDetector** detectar si el moviment s'ha produït a la dreta o a l'esquerra.

```

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()
    })

```

Selecció Multiple en el Recycler

Android aporta la biblioteca de **recyclerview-selection** per a [seleccionar més d'un element d'una llista](#), creada amb un recycler. És a dir l'acció que ocorre en la majoria de llistes, quan realitzem un clic llarg i després ens deixa seleccionar elements. Per a poder implementar aquesta selecció, haurem de tindre creat un RecyclerView en perfecte funcionament, a partir d'ací haurem de seguir una sèrie de passos:

1. Implementar **ItemDetailsLookup**. Aquest element permet que la biblioteca de selecció accedisca a la informació sobre els elements de RecyclerView als quals s'atorga un **MotionEvent** (alguna acció de moviment: pulsació, arrossegament, etc.). Per a construir-ho necessitarà la informació de l'element premut, que es pot extraure del **Holder**. Per a això el primer que farem serà crear en el **Holder.Kt**, un mètode al qual es puga cridar per a identificar de manera exclusiva l'element de la llista premut. Aquest mètode ha de retornar una instància de la classe **ItemDetailsLookup.ItemDetails**.

```

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

```

I sobreescriure els dos mètodes de la classe abstracta per a identificar de manera única l'element sobre el qual s'ha realitzat l'acció. Per a això podem usar les propietats `adapterPosition` i `itemId`, que posseeix la classe `ViewHolder`. Com es pot veure en el codi, hem fet de tipus `Long` la classe genèrica `ItemDetails`. El long determina el tipus de clau de selecció que usarem. Hi ha tres tipus de claus que es poden usar per a identificar elements seleccionats: *Parcelable*, *String* i *Long*. Ara passarem a implementar la classe, per exemple **LookUp.kt**, que herete de `ItemDetailsLookup` de la mena de la clau seleccionada. En el mètode que anul·lem, es captura la vista seleccionada i amb ella accedim al mètode creat amb anterioritat 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. Actualitza les Views dels elements de RecyclerView perquè reflectisca si l'usuari va realitzar o no una selecció. Aquests canvis es poden fer en el Holder, com hem vist anteriorment, encara que en aquest cas haurem de tindre en compte si l'element ha sigut seleccionat. Per a això usem un tipus `SelectionTracker` que passarem com a argument des de l'Adaptador. S'aconsella que es mostre els elements seleccionats mitjançant un canvi 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 els anteriors passos mitjançant un objecte de tipus `SelectionTracker.Builder` **Línia 6 a 14**, creat en l'activitat principal i passat a

l'adaptador, que al seu torn el manarà al Holder. Aquest element inicialitza el rastrejador de selecció. Al seu constructor se li passa: l'ID de selecció (una cadena única), la instància del RecyclerView, el proveïdor de claus (es pot triar entre varis de la BCL), la classe creada en el pas un per al control de l'element premut i l'estratègia d'emmagatzematge en aquest cas de tipus Long com la clau (aquesta instància permet assegurar que no es perden els elements seleccionats en els canvis d'estat del dispositiu).

```
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
```

A més s'haurà de fixar la grandària del recycler **Línia 1**, assignar a l'adaptador la propietat de HasStableIds a true **Línia 3**. i passar la instància de Tracker a l'adaptador mitjançant un mètode que crearem en aquest **línia 15**.

Per a controlar amb més seguretat l'estabilitat de l'app, s'hauria d'anul·lar el mètode `onSaveInstanceState` per a passar la informació capturada, a la instància del `tracker`.

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

I en iniciar l'aplicació afegir la següent línia per a recuperar el tracker guardat:

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

Per la seua part en l'adaptador només tindrem aquests dos canvis.

```

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

```

Després de realitzar aquests passos, podem provar el funcionament de l'aplicació.


Reconstrueix l'exemple i prova el funcionament

4. Només ens queda controlar la pulsació i realitzar alguna cosa quan això ocórrega. Haurem d'afegir un observador al Tracker i per a això s'utilitza un objecte **SelectionTracker.SelectionObserver**, que s'encarrega de registrar els canvis. Dins d'aquest objecte posarem el codi que creguem convenient, en aquest cas solament vam mostrar un Toast amb l'id dels elements que estan en la selecció.

```

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 comportament esperat quan es fa una selecció multiple, és el que se superpose una toolbar sobre la ActionBar amb un menú específic per a les accions que puguin ocórrer sobre la selecció. Per a això s'haurà d'implementar la classe **ActionMode** que serà activada quan l'observador detecte que té selecció. Aquesta implementació es veurà en el tema de menús