Programación Multimedia y Dispositivos Móviles

UD 5. Elementos complejos en Android

Javier Carrasco

Curso 2024 / 2025

Elementos complejos en Android

5. Elementos complejos en Android

Este capítulo amplía el anterior, añadiendo elementos considerados complejos, ya que requieren de una parte importante de programación para poder hacerlos funcionar. Introduciendo el concepto de adaptador.

5.1. Adapter y AdapterView

Antes de seguir con elementos, o *widgets*, más complejos, se introducirán una serie de conceptos que ayudarán a entender mejor lo que está por venir.

- ***Adapter***[1], es un objeto que hará las funciones de puente entre un *AdapterView* y los datos de una vista. El *Adapter* se encargará del acceso a cada elemento y construir sus vistas. Entendiendo por vistas, aquello que se muestra al usuario.

  Existen diferentes tipos de *adapterse* según el elemento que vaya a utilizarse para representar la información, *ArrayAdapter*, *BaseAdapter*, *CursorAdapter*, *SimpleAdapter*, *SimpleCursorAdapter*, etc.

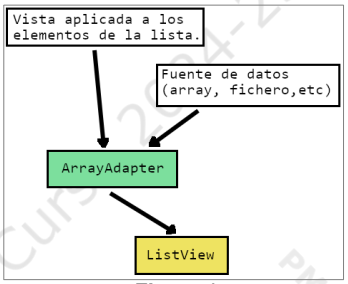

Figura 1

- ***AdapterView***[2], es la vista, cuyos hijos vienen determinados por el *Adapter*. Muestra los elementos cargados en un adaptador, los elementos que se deben cargar suelen venir de una fuente de datos basada en un *array*.

Cuando se utilice la clase *AdapterView*, se dispondrá de una serie de métodos que se deben utilizar, estos dispondrán de unos *callbacks* que habrá que sobrecargar, en función del elemento y acción a gestionar. Algunos de los más comunes son:

- AdapterView.OnItemClickListener
  - **onItemClick(AdapterView<?> parent, View view, int position, long id)**, este *callback* será invocado cuando un elemento sea pulsado dentro de este *AdapterView*. Sus parámetros son:
    - **parent**, indica el *AdapterView* donde se ha producido el clic, también se puede encontrar como `p0`.
    - **view**, es la vista dentro del *AdapterView* en la que se hizo el clic, también se puede encontrar como `p1`.
    - **position**, posición de la vista, del elemento, sobre el que se ha hecho clic, también puedes encontrarlo como `p2`.
    - **id**, identifica la fila que contiene el elemento pulsado, también se puede encontrar como `p3`.

1 *Adapter* (https://developer.android.com/reference/kotlin/android/widget/Adapter.html)
2 *AdapterView* (https://developer.android.com/reference/kotlin/android/widget/AdapterView.html)

- AdapterView.OnItemSelectedListener
  - **onItemSelected(AdapterView<?> parent, View view, int position, long id)**, funciona exactamente igual que `onItemClick()`.
  - **onNothingSelected(AdapterView<?> parent)**, será invocado cuando la selección desaparezca de la vista. El parámetro `parent` contiene el contenedor que ya tiene el elemento no seleccionado.

En el siguiente punto se verá como utilizar estos conceptos con los elementos, o *widgets*, que hacen uso de ellos y se pondrá en práctica los métodos vistos.

A continuación se mostrarán una serie de elementos que hacen uso de los *Adapters* y *AdapterViews*. Para los ejemplos de cada uno de los elementos se partirá de un proyecto nuevo con una actividad vacía y API mínima 23.

5.2. ListView simple

Un ***ListView***[3] es un *widget* que permite mostrar elementos de un *array* como una lista *scrollable* (desplazable) al usuario.

Como se muestra en el esquema, se necesitan dos elementos clave para montar el *Adapter*, en primer lugar la fuente de datos (un *array*, un fichero, una base de datos, etc), y en segundo lugar, un *layout* en el que montar cada uno de los *items* de la lista.

El primer paso consistirá en preparar el contenedor, añadiendo un componete *ListView* a la actividad que contendrá el listado, la actividad principal por ejemplo. Lo encontrarás en la biblioteca, en la sección *Legacy*.

```
<androidx.constraintlayout.widget.ConstraintLayout
    ...>

    <ListView
        android:id="@+id/listView"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

Para este ejemplo, la fuente de datos será un *array* simple creado como una propiedad de la *MainActivity*.

```
private val datos = arrayOf(
    "uno", "dos", "tres", "cuatro", "cinco",  "seis", "siete", "ocho", "nueve", ...)
```

3 *ListView* (https://developer.android.com/reference/android/widget/ListView)

El siguiente paso consistirá en crear el *Adapter*, donde se producirá la unión entre datos y vistas. Para el primer ListView se utilizará una vista simple, pre-diseñada de Android (*android.R.layout.simple_list_item_1*), fíjate que para hacer uso de los recursos de Android, se llama directamente al objeto ***android***. Tras crear el *Adapter*, se asigna este al contenedor, al *ListView* creado en la actividad.

```
val adaptador = ArrayAdapter(
    this,
    android.R.layout.simple_list_item_1, // lista pre-diseñada
    datos
)

binding.listView.adapter = adaptador
```

Hasta aquí, si lanzas la aplicación, podrás ver que ya dispones de un listado con los datos obtenidos del *array*. La lista incorpora su propio *scroll*, pero como podrás comprobar, todavía no tiene ningún tipo de interacción a la hora de seleccionar cualquier elemento.

Figura 2

Para añadir un evento *onClick* sobre cada ítem, habrá que crear un evento *onItemClickListener* para el *AdapterView* del adaptador. Se puede añadir a continuación de la asignación del adaptador.

```
binding.listView.onItemClickListener =
    object : AdapterView.OnItemClickListener {}
```

Al añadir estas líneas, el error que se produce se solucionará sobrecargando los métodos que pide la clase *AdapterView*. Para implementar los métodos que necesitas puedes utilizar *Alt+Intro* para añadirlos, o utilizar la opción de menú *Code* > *Implement Methods*. El código para el evento quedará como se muestra a continuación.

```
binding.listView.onItemClickListener =
    object : AdapterView.OnItemClickListener {
        override fun onItemClick(
            parent: AdapterView<*>?,
            view: View?,
            position: Int,
            id: Long
        ) {
            TODO("Not yet implemented")
        }
    }
```

Podría mostrarse un *Toast* con el nombre del ítem pulsado.

```
Toast.makeText(
    this@MainActivity, "Pulsado ${datos[position]}",
    Toast.LENGTH_SHORT
).show()
```

5.3. ListView personalizado

En el siguiente ejemplo se creará un *ListView* con vista personalizada, esto permite adaptar la forma de ver los elementos dentro del listado.

En primer lugar se creará la vista de los *items*, para ello se necesita un fichero XML. Se creará un nuevo `res/layout`, utilizando la opción **File > New > Android resource file**. Tal cual aparece en la siguiente imagen.

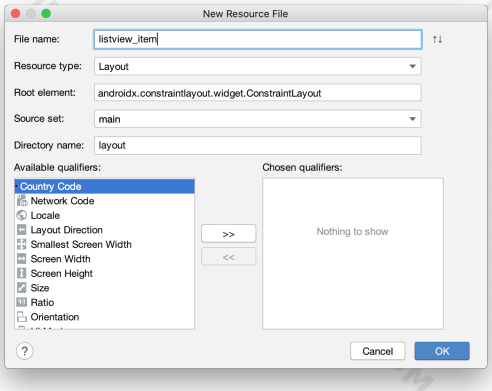

Figura 3

Una vez creado el fichero `listview_item.xml`, edita el fichero para que contenga el siguiente código, fíjate que se deja como elemento raíz el propio *TextView*.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Cada elemento de la lista se mostrará con el siguiente TextView -->
<TextView xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/titulo"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:padding="20dp"
    android:textSize="30sp"
    android:textStyle="bold"
    tools:text="Nombre persona" />
```

Este será el formato que tendrá cada uno de los elementos de lista. A continuación, se añade un componente *ListView* al *layout* `activity_main.xml`.

```
<ListView
    android:id="@+id/myListView"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:listitem="@layout/listview_item" />
```

Fíjate en la propiedad `tools:listitem`, ésta te permitirá establecer el *layout* que vayas a utilizar como listado como vista previa y ver como quedaría. Ya se puede crear el código fuente para establecer la relación entre ambos elementos y cargar la información. Para este ejemplo, los datos se cargarán desde un *array* estático.

```
class MainActivity : AppCompatActivity() {
    // Fuente de datos para el ListView.
    private val nombres = arrayOf(
        "Javier", "Nacho", "Patricia", "Miguel", "Susana", "Rosa", "Juan", "Pedro",
        "Asunción", "Antonio", "Lorena", "Verónica", "Paola", "Esteban", "Andrea", "María"
    )

    private lateinit var binding: ActivityMainBinding

    override fun onCreate(savedInstanceState: Bundle?) {
      super.onCreate(savedInstanceState)
      binding = ActivityMainBinding.inflate(layoutInflater)
      setContentView(binding.root)

      // Se crea el Adapter uniendo la vista y los datos.
      val adapter = ArrayAdapter(this, R.layout.listview_item, nombres)

      // Se asigna el Adapter al Listview.
      binding.myListView.adapter = adapter

      // Se utiliza un AdapterView para conocer el elemento pulsado.
      binding.myListView.onItemClickListener = object: AdapterView.OnItemClickListener {
          override fun onItemClick(p0: AdapterView<*>?, p1: View?, p2: Int, p3: Long) {
                Toast.makeText(
                    applicationContext,
                    "${binding.myListView.getItemAtPosition(p2)}",
                    Toast.LENGTH_SHORT
                ).show()
          }
        }
    }
}
```

Fíjate que los pasos están bastante claros y son mecánicos, la línea `val adapter = ArrayAdapter(this, R.layout.listview_item, nombres)` es dónde se realiza la unión entre la vista y los datos, para en la línea siguiente asignarla al componente *ListView* que mostrará la información. Observa que la diferencia radica en el tipo de vista que se utilizará para mostrar los elementos.

No te preocupes por la cantidad de código, parte de éste lo auto-completa el propio IDE según vayas escribiendo o utilizando la implementación automática de miembros.

Una vez se lance la aplicación podrás ver el resultado, y debe ser muy similar al que se muestra en la imagen que se muestra en la figura. Cada vez que se haga clic sobre uno de los elementos de la lista deberá mostrarse un *toast* con el nombre pulsado.

Es posible que, según las circunstancias que se den, se necesiten obtener los datos desde otra fuente diferentes, o varias, por ejemplo, un *array* de datos almacenado en el fichero `strings.xml` dentro de `res/values`. A continuación, puedes ver el *array* utilizado en el ejemplo.

```xml
<resources>
    ...
    <string-array name="array_nombres">
        <item>Javier</item>
        <item>Nacho</item>
        <item>Patricia</item>
        <item>Miguel</item>
        <item>Susana</item>
        <item>Rosa</item>
        <item>Juan</item>
        <item>Pedro</item>
        <item>Asunción</item>
        <item>Antonio</item>
        <item>Lorena</item>
        <item>Verónica</item>
        <item>Paola</item>
        <item>Esteban</item>
        <item>Andrea</item>
        <item>María</item>
    </string-array>
</resources>
```

Figura 4

En tal caso, únicamente se deberá modificar la unión entre vista y datos para que adopte la siguiente forma, obteniéndose la información de la variable de tipo *array* creada en *strings.xml*.

```
// Se establece la fuente de datos.
val nombres2 = resources.getStringArray(R.array.array_nombres)
// Se crea el Adapter uniendo la vista y los datos.
val adapter = ArrayAdapter(this, R.layout.listview_item, nombres2)
```

5.4. GridView

El funcionamiento básico de un ***GridView***[4] es exactamente igual al mostrado para el *ListView*, la principal diferencia entre ambos radica en que el *GridView* permite mostrar la información en formato de tabla. A continuación, se muestra el código que debes utilizar si quieres un *GridView* simple.

Tras comenzar un nuevo proyecto, crea el fichero `gridview_item.xml` y edita el fichero para que contenga el siguiente código.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Cada elemento de la lista se mostrará con el siguiente TextView -->
<TextView xmlns:android="http://schemas.android.com/apk/res/android"
```

4 *GridView* (https://developer.android.com/reference/android/widget/GridView)

```
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/titulo"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:padding="20dp"
    android:textSize="20sp"
    android:textStyle="bold"
    tools:text="Nombre persona" />
```

Como puedes ver, para este ejemplo, el código es exactamente igual al utilizado para el *ListView*. A continuación, añade un *GridView* al fichero `activity_main.xml`, donde encontrarás una ligera diferencia con respecto al ejemplo anterior.

```
<GridView
    android:id="@+id/myGridView"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:numColumns="auto_fit"
    tools:listitem="@layout/gridview_item" />
```

En los *GridView*, una propiedad importante es `android:numColumns`, esta permitirá ajustar automáticamente el número de columnas utilizando `auto_fit` o especificar un valor fijo si fuese necesario.

El código **Kotlin** que se utilizará es exactamente el mismo que el empleado en el ejemplo anterior, cambiando *ListView* por *GridView* a la hora de crear el *adapter* y el nuevo componente añadido.

El resultado que se obtenga diferirá con respecto al *ListView*, pero el funcionamiento será exactamente igual al que se ha visto.

A continuación, se ampliará el uso de *GridView*, también valdría para el *ListView*, añadiendo algo más de información a cada uno de los elementos que forman la lista. Para ello se utilizará la clase abstracta ***BaseAdapter***, hasta ahora se había utilizado *ArrayAdapter*, que viene bien para listas o *arrays*. *ArrayAdapter* hereda de la clase *BaseAdapter*, pero al tratarse de una clase abstracta, será necesario escribir algo más de código.

Para este ejemplo, comienza por crear un nuevo proyecto, *My GridView 2*, por ejemplo, y crea un nuevo *layuout* llamado `gridview_item.xml`, pero esta vez, su contenido será como el que se muestra a continuación.

Figura 5

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="150dp"
    android:layout_height="wrap_content"
    android:background="#eee"
```

```
    android:gravity="center"
    android:orientation="vertical"
    android:padding="15dp">

    <ImageView
        android:id="@+id/image"
        android:layout_width="150dp"
        android:layout_height="150dp"
        android:contentDescription="@string/desc_imagen"
        app:srcCompat="@mipmap/ic_launcher" />

    <TextView
        android:id="@+id/tvName"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:gravity="center"
        android:textSize="20sp"
        tools:text="Descripción" />
</LinearLayout>
```

La idea de este *layout* es conseguir algo parecido a lo que puedes ver en la imagen que se muestra en la siguiente figura.

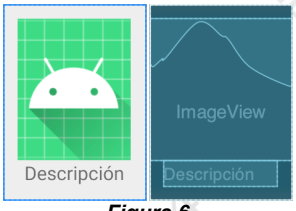

Figura 6

El siguiente paso será crear un *GridView*, en este caso, en la `activity_main.xml`, además, se ajustará su anchura para que case con la que se le ha dado a la imagen.

```
<GridView
    android:id="@+id/myGridView"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:columnWidth="150dp"
    android:horizontalSpacing="15dp"
    android:numColumns="auto_fit"
    android:verticalSpacing="15dp" />
```

Ahora viene la parte de programación, lo primero que se necesitará es crear una clase para representar la información, para este ejemplo, `MyItems.kt` (**File > New > Kotlin Class/File**). Concretamente se utilizará una *data class* para este caso. El contenido de la *data class* será como se muestra a continuación.

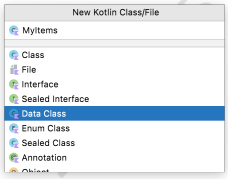

Figura 7

```kotlin
// Clase encargada de almacenar la información de un ítem.
data class MyItems(val name: String, val image: Int)
```

Ahora, crea otra nueva clase llamada `ItemAdapter` que extenderá de **BaseAdapter** y se encargará de "inflar" los ítems, básicamente de poner cada dato en su sitio.

```kotlin
// Clase ItemAdapter que hereda de la clase abstracta BaseAdapter.
class ItemAdapter(val context: Context, val itemsList: ArrayList<MyItems>): BaseAdapter() {
    override fun getCount(): Int {
        return itemsList.size
    }

    override fun getItem(p0: Int): Any {
        return itemsList[p0]
    }

    override fun getItemId(p0: Int): Long {
        return p0.toLong()
    }

    // Devuelve la vista cargada de cada elemento al adaptador.
    override fun getView(p0: Int, p1: View?, p2: ViewGroup?): View {
        return if (p1 == null) { // Se comprueba si la vista es nula.
            val item = this.itemsList[p0]
            val layoutInflater = LayoutInflater.from(context)
            val binding = GridviewItemBinding.inflate(layoutInflater, p2, false)

            binding.image.setImageResource(item.image)
            binding.tvName.text = item.name

            binding.root
        } else p1.rootView  // La vista que llena no es nula y se devuelve.
    }
}
```

En este fragmento de código aparece una nueva clase conocida como `ViewGroup`, ésta representa un objeto invisible en el que se organizarán las vistas individuales de cada elemento (*View*), las contendrá.

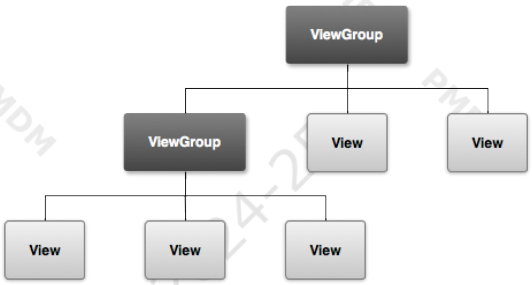

Figura 8

El método `getView()`, dónde se utiliza el *LayoutInflater*, también se puede escribir así,

```
var inflator = context!!.getSystemService(
    Context.LAYOUT_INFLATER_SERVICE
) as LayoutInflater
```

en vez de la línea utilizada,

```
val layoutInflater = LayoutInflater.from(context)
```

¿Qué significa "inflar"?, básicamente es añadir a un elemento contenedor, en este caso el *GridView* a través del adaptador, elementos de otra vista, que viene a ser `gridview_item.xml`.

Vuelve a la clase `MainActivity.kt` para terminar de escribir el código que falta.

```
class MainActivity : AppCompatActivity() {
    private lateinit var binding: ActivityMainBinding

    var adapter: ItemAdapter? = null
    var itemsList = ArrayList<MyItems>()

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)

        // Se crea la fuente de datos con imágenes de muestra.
        itemsList.add(MyItems("Estrella", android.R.drawable.btn_star))
        itemsList.add(MyItems(Botón Check", android.R.drawable.checkbox_off_background))
        itemsList.add(MyItems("Lupa", android.R.drawable.ic_menu_search))
        itemsList.add(MyItems("Candado", android.R.drawable.ic_lock_lock))
        itemsList.add(MyItems("Launcher Back", R.drawable.ic_launcher_background))
        itemsList.add(MyItems("Launcher Icon", R.drawable.ic_launcher_foreground))
        itemsList.add(MyItems("Launcher", R.mipmap.ic_launcher))

        // Se generamos el adaptador.
        adapter = ItemAdapter(this, itemsList)
        // Asignamos el adapter
        binding.myGridView.adapter = adapter
    }
}
```

Para saber que vista ha pulsado el usuario puede hacerse de varias formas, la primera, se podría añadir un evento `setOnClickListener()` sobre cada una de las vistas de la siguiente forma, en la misma función `getView()` de `ItemAdapter()` se añadirán las siguientes líneas.

```
// Pulsación sobre la vista.
binding.root.setOnClickListener {
    Toast.makeText(context, "${binding.tvName.text}", Toast.LENGTH_LONG).show()
}
```

La otra opción es la que se ha estado utilizando hasta ahora, con una pequeña variación, ya que la vista individual de cada elemento ahora es más compleja. Para este método se necesitará añadir lo siguiente `import androidx.core.view.get` o pulsar `Alt+Intro` para añadir las dependencias. En este caso se implementará la funcionalidad en el método `onStart()`.

```
override fun onStart() {
    super.onStart()

    binding.myGridView.onItemClickListener = object : AdapterView.OnItemClickListener {
        override fun onItemClick(p0: AdapterView<*>?, p1: View?, p2: Int, p3: Long) {
            val bindingItem = GridviewItemBinding.bind(p1!!)
            Toast.makeText(
                applicationContext,
                "Pulsado ${bindingItem.tvName.text}",
                Toast.LENGTH_SHORT
            ).show()
        }
    }
}
```

Fíjate como se hace la referencia, en este caso, al texto del *TextView*. También se pueden utilizar *lambdas* para generar el mismo código, algo de uso muy común en **Kotlin**.

```
override fun onStart() {
    super.onStart()

    binding.myGridView.onItemClickListener =
        AdapterView.OnItemClickListener { p0, p1, p2, p3 ->
            val bindingItem = GridviewItemBinding.bind(p1!!)
            Toast.makeText(
                applicationContext,
                "Pulsado ${bindingItem.tvName.text}",
                Toast.LENGTH_SHORT
            ).show()
        }
}
```

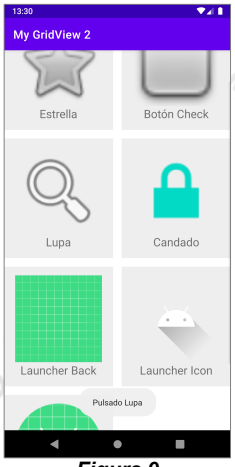

Figura 9

El resultado que se debería obtener tras estas líneas de código debe ser algo similar a lo que se ve en la figura.

La principal diferencia entre estos dos sistemas para controlar que *item* se ha pulsado, ya bien sea un *ListView* o un *GridView*, es la siguiente; el primero, está implementado dentro del método `getView()`, únicamente funcionará si se hace clic sobre la propia imagen, el segundo, se encuentra implementado en el método `onStart()`, y funcionará si se pulsa sobre el *item* completo, no solo sobre la imagen.

5.5. Spinner

Los ***spinner***[5] son elementos desplegables que permiten seleccionar un elemento de una lista múltiple. Tiene una menor personalización que los anteriores, pero será necesario utilizar la clase *Adapter*.

Se partirá del *string-array* de nombres utilizado para el *ListView* de ejemplo (se podría añadir un primer elemento al *string-array* del tipo "Selecciona un elemento") y, se añade al fichero `activity_main.xml` un *spinner*.

```
<Spinner
    android:id="@+id/mySpinner"
    style="@android:style/Widget.Material.Spinner.Underlined"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_marginEnd="8dp"
    android:minHeight="48dp"
    android:spinnerMode="dropdown" />
```

La propiedad `style` te permite seleccionar el estilo del *Spinner* y, la propiedad `spinnerMode` te da la opción de elegir su comportamiento, modo desplegable (*dropdown*) o modo diálogo (*dialog*).

Una vez esté añadido, ve a la clase `MainActivity.kt` para añadir los elementos a la lista y el código necesario para saber que elemento es seleccionado por el usuario.

```
class MainActivity : AppCompatActivity() {
    private lateinit var binding: ActivityMainBinding

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)

        // Se crea el Adapter, uniendo la fuente de datos con una vista
        // por defecto para el spinner de Android.
        val adapter = ArrayAdapter.createFromResource(
            this,
            R.array.array_nombres,
            android.R.layout.simple_spinner_item
        )

        // Se especifica el diseño que debe utilizarse para mostrar la lista.
        adapter.setDropDownViewResource(
            android.R.layout.simple_spinner_dropdown_item
        )

        // Se carga el Adapter en el Spinner.
```

5 *Spinner* (https://developer.android.com/guide/topics/ui/controls/spinner)

```
        binding.mySpinner.adapter = adapter
    }

    override fun onStart() {
       super.onStart()

       binding.mySpinner.onItemSelectedListener =
         object : AdapterView.OnItemSelectedListener {
           override fun onItemSelected(p0: AdapterView<*>?, p1: View?, p2: Int, p3: Long) {
                 if (p2 > 0) {
                     Toast.makeText(
                         applicationContext,
                         "${binding.mySpinner.getItemAtPosition(p2)}!",
                         Toast.LENGTH_LONG
                     ).show()
                 }
                 Log.d(
                     "MySpinner",
                     "${binding.mySpinner.getItemAtPosition(p2)}!"
                 )
             }
             override fun onNothingSelected(p0: AdapterView<*>?) {
                 // No se hace nada.
             }
         }
    }
}
```

El resultado que se debería obtener con todo lo visto hasta aquí sería algo similar a lo que se puede ver en la figura mostrada.

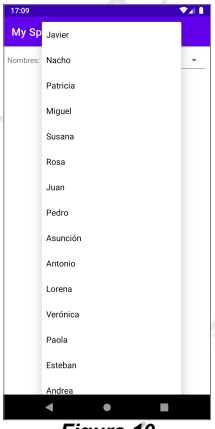

Figura 10

También se puede utilizar la clase ***Log***[6] para mostrar información en la consola de Android Studio en lugar de utilizar siempre *Toasts* o *SnackBars* por ejemplo.

Gracias a esta clase, es posible ver información de la aplicación mediante el *Logcat* del IDE, lo que permite realizar trazas del código. Puedes utilizar el método `d` para *debug*, `e` para mostrar errores, `w` para los *warnings*, etc.

```
Log.d("MySpinner", "${binding.mySpinner.getItemAtPosition(p2)}!")
```

Otra propiedad que puede resultar interesante de los *Spinner* es `popupBackground`, puedes asignarle un recurso *drawable* para establecer un diseño a lista desplegable, por ejemplo, cambiarle el color de fondo y establecer un borde alrededor de la lista.

6 *Log* (https://developer.android.com/reference/android/util/Log)

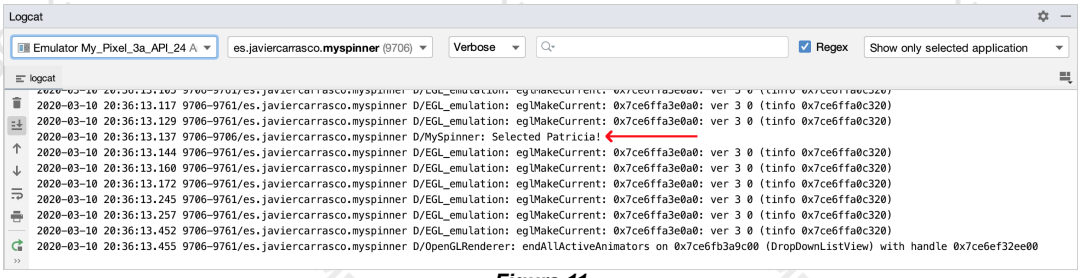

Figura 11

5.6. AutoCompleteTextView

Este elemento, que ya conocerás, permite auto-completar aquello que el usuario va escribiendo, pero para ello, debe conocer la fuente con la que auto-completar. Su funcionamiento es muy parecido al visto con el *Spinner*. Lo primero será añadir el elemento a la vista.

```
<AutoCompleteTextView
    android:id="@+id/autoCompleteTextView"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:hint="@string/hint_autoComplete" />
```

Puedes dejarlo así, sencillo, o añadirle un *TextInputLayout* para hacerlo más vistoso, dejando el código de la siguiente forma.

```
<com.google.android.material.textfield.TextInputLayout
    style="@style/Widget.MaterialComponents.TextInputLayout.OutlinedBox.ExposedDropdownMenu"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:hint="@string/hint_autoComplete"
    android:labelFor="@id/autoCompleteTextView" >

    <AutoCompleteTextView
        android:id="@+id/autoCompleteTextView"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />
</com.google.android.material.textfield.TextInputLayout>
```

Observa que la propiedad *hint* del *AutoCompleteTextView* se ha pasado al *TextInputLayout*, de esta forma se mantiene en el borde dibujado.

Ahora, desde la clase se montará el *adapter* necesario, se utilizará el *string-array* utilizado para el *Spinner*, pero esta vez, sin la opción “Selecciona un elemento”, con este tipo de elemento ya no es necesario.

```
// Se crea un adapter de tipo Array.
val adapter = ArrayAdapter.createFromResource(
    this,
```

```
        R.array.array_nombres,
        android.R.layout.simple_list_item_1
    )
    // Se asigna el adapter al AutoComplete.
    binding.autoCompleteTextView.setAdapter(adapter)
```

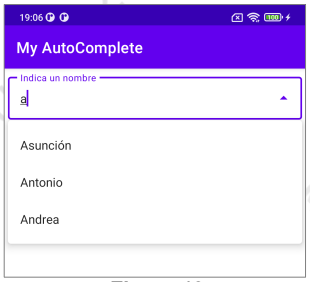

Figura 12

Para saber que elemento se seleccionado de la lista, de una forma sencilla, bastará con añadir un *listener* al *AutoCompleteTextView*. Observa que, como otros elementos de texto, se puede acceder a su propiedad *text* para acceder al dato.

```
binding.autoCompleteTextView.setOnItemClickListener { adapterView, view, i, l ->
    Toast.makeText(this, binding.autoCompleteTextView.text, Toast.LENGTH_SHORT).show()
}
```

5.7. RecyclerView y CardView

Los *widgets* ***RecyclerView***[7] *y* ***CardView***[8] son dos componentes que suelen trabajar juntos, pero no necesariamente. Aparecen en **Android** a partir de la versión *Lollipop* (**API 21** Android 5.0) con la intención de hacer la vida del programador algo más fácil. Debes saber, que haciendo uso de estos dos elementos se cumplirá con la idea de *Material Design* de Google.

Empieza añadiendo un **RecyclerView** a la vista principal, en versiones anteriores en el *Layout Editor* aparecía con un símbolo de descarga junto al componente, esto indica que se añadirán dependencias al *build.gradle (Module: app)*. Su código será algo así.

```
<androidx.recyclerview.widget.RecyclerView
    android:id="@+id/recyclerView"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

5.7.1. RecyclerView simple

Para este ejemplo se establecerá una fuente de datos sencilla, añadiendo la siguiente propiedad a la clase principal.

```
private val datos = mutableListOf(
    "Uno", "Dos", "Tres", "Cuatro", "Cinco", "Seis", "Siete", "Ocho", ... )
```

Dentro de la misma clase se creará una clase interna llamada `ViewHolder`, esta heredará de *RecyclerView.ViewHolder*, y será la encargada de crear el elemento visible para cada ítem.

```
inner class ViewHolder(view: View) : RecyclerView.ViewHolder(view) {
    val textField: TextView = view.findViewById(android.R.id.text1)
}
```

7 RecyclerView (https://developer.android.com/reference/androidx/recyclerview/widget/RecyclerView)
8 CardView (https://developer.android.com/reference/kotlin/androidx/cardview/widget/CardView)

Como verás a continuación, `android.R.id.text1` hace referencia al elemento de la vista predefinida que se utilizará al crear el adaptador. Antes de crear el adaptador en el método *onCreate()*, se establecerá el tipo de listado para el *RecyclerView*.

```
binding.recyclerView.layoutManager = LinearLayoutManager(this)
```

Aunque esta propiedad también puedes dejarla establecida desde el propio XML utilizando la siguiente línea.

```
app:layoutManager="androidx.recyclerview.widget.LinearLayoutManager"
```

La creación del adaptador dentro del método *onCreate()* quedará como se muestra a continuación.

```
binding.recyclerView.adapter =
        object : RecyclerView.Adapter<ViewHolder>() {
            // Devuelve el número de elementos.
            override fun getItemCount(): Int {
                return datos.size
            }

            // Se encarga de "inflar" la vista.
            override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder {
                return ViewHolder(
                    LayoutInflater.from(parent.context)
                        .inflate(android.R.layout.simple_list_item_1, parent, false)
                )
            }

            // Se une la vista con el dato.
            override fun onBindViewHolder(holder: ViewHolder, position: Int) {
                holder.textField.text = datos[position]
            }
        }
```

Ya dispondrías de un listado básico, pero sin interacción con el usuario. A continuación, se añadirá un *listener* para detectar la selección del usuario. Para este tipo básico puede hacerse de dos maneras, dentro del método *onBindViewHolder*, añadiendo el *listener* al *TextView*.

```
holder.textField.setOnClickListener {
    Toast.makeText(
        this@MainActivity,
        "Pulsado '${datos[position]}'",
        Toast.LENGTH_SHORT
    ).show()
}
```

O añadir el *listener* a la vista completa del ítem, en tal caso, deberá crearse dentro de la clase *ViewHolder*.

```
inner class ViewHolder(view: View) : RecyclerView.ViewHolder(view) {
    val textField: TextView = view.findViewById(android.R.id.text1)

    init {
        // Evento onClick sobre la vista completa.
        itemView.setOnClickListener {
            Toast.makeText(
                this@MainActivity, "Pulsado '${textField.text}'", Toast.LENGTH_SHORT
            ).show()
        }
    }
}
```

5.7.2. RecyclerView personalizado

En ocasiones, con un listado simple no es suficiente, y se necesita tener listados más complejos, adaptados a las necesidades del proyecto. En tal caso, será necesario personalizar la vista que mostrará cada elemento.

Tras añadir un *RecyclerView* a la actividad principal, se creará la vista individual para cada uno de los elementos, en este caso se mostrará la imagen de un animal, su nombre común y su nombre científico. Para tal caso, se creará un nuevo *layout* llamado `item_animal_list.xml`.

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical">

    <ImageView
        android:id="@+id/iv_animalImage"
        android:layout_width="150dp"
        android:layout_height="150dp"
        android:contentDescription="@string/desc_image"
        app:srcCompat="@mipmap/ic_launcher_round" />

    <TextView
        android:id="@+id/tv_nameAnimal"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_alignParentTop="true"
        android:layout_marginStart="5dp"
        android:layout_marginTop="10dp"
        android:layout_toEndOf="@+id/iv_animalImage"
        android:text="@string/name_text"
        android:textSize="30sp" />

```

```
    <TextView
        android:id="@+id/tv_latinName"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@+id/tv_nameAnimal"
        android:layout_alignBottom="@+id/iv_animalImage"
        android:layout_alignParentEnd="true"
        android:layout_marginStart="5dp"
        android:layout_marginTop="10dp"
        android:layout_toEndOf="@+id/iv_animalImage"
        android:text="@string/latin_text"
        android:textSize="18sp" />
</RelativeLayout>
```

Figura 13

Ahora, se creará el modelo de datos que contendrá la información sobre los animales, para eso se creará la *data class* `MyAnimal.kt` con el siguiente código.

```
data class MyAnimal(val animalName: String, val latinName: String, val imageAnimal: Int)
```

Seguidamente, se creará el *adapter* para el *RecyclerView*, hasta ahora se ha estado colocando todo el código en la clase *MainActivity.kt*, pero hay que empezar a modularizar, crea una nueva clase llamada `MyAnimalAdapter.kt`, que heredará de `RecyclerView.Adapter`, y deberás sobrecargar sus métodos.

Esta clase se encargará de recorrer la lista de animales que se le pase, para que, utilizando una clase *ViewHolder* propia se rellenarán los campos. Observa el código siguiente.

```
class MyAnimalAdapter(val animalsList: MutableList<MyAnimal>) :
    RecyclerView.Adapter<MyAnimalAdapter.MyAnimalViewHolder>() {

    // Es el encargado de devolver el ViewHolder ya configurado.
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): MyAnimalViewHolder {
        return MyAnimalViewHolder(
            ItemAnimalListBinding.inflate(
                LayoutInflater.from(parent.context),
                parent,
                false
            ).root
        )
    }
```

```
    // Método encargado de pasar los objetos, uno a uno, al ViewHolder personalizado.
    override fun onBindViewHolder(holder: MyAnimalViewHolder, position: Int) {
        holder.bind(animalsList[position])
    }

    // Devuelve el tamaño de la fuente de datos.
    override fun getItemCount() = animalsList.size

    // Esta clase interna se encarga de rellenar cada una de las vistas que
    // se inflarán para cada uno de los elementos del RecyclerView.
    class MyAnimalViewHolder(view: View) : RecyclerView.ViewHolder(view) {
        // Se usa View Binding para localizar los elementos en la vista.
        private val binding = ItemAnimalListBinding.bind(view)

        fun bind(animal: MyAnimal) {
            binding.tvNameAnimal.text = animal.animalName
            binding.tvLatinName.text = animal.latinName

            Glide.with(binding.root)
                .load(animal.imageAnimal)
                .override(150, 150)
                // Centra la imagen y redondea las esquinas.
                .transform(CenterCrop(), RoundedCorners(10))
                .into(binding.ivAnimalImage)

            // Listener sobre la vista.
            itemView.setOnClickListener {
                Toast.makeText(binding.root.context, animal.animalName,
                    Toast.LENGTH_SHORT).show()
            }
        }
    }
}
```

Observa que se utiliza *Glide* para establecer la imagen el *ImageView*, modificando su comportamiento mediante la cláusula *transform*. Se podría haber hecho con la siguiente línea, pero no se podría ajustar como se ha hecho con *Glide*.

```
binding.ivAnimalImage.setImageResource(animal.imageAnimal)
```

Si te fijas en el *listener* que se ha añadido en la función *bind*, se ha hecho sobre el objeto `itemView`, este objeto hace referencia a la vista del elemento que se está representando, de esta forma, se detectará el click sobre cualquier parte del elemento mostrado.

A continuación, fíjate como quedaría la clase principal `MainActivity.kt`, donde se han añadido dos nuevos métodos, uno para configurar el *RecyclerView*, y otro que se encargará de generar la fuente de datos.

```
class MainActivity : AppCompatActivity() {
```

```
    private lateinit var binding: ActivityMainBinding
    private lateinit var myAdapter: MyAnimalAdapter

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)
        setUpRecyclerView()
    }

    private  fun setUpRecyclerView() {
        // Esta opción a TRUE significa que el RV tendrá hijos del mismo tamaño,
        // optimiza su creación.
        binding.recyclerView.setHasFixedSize(true)

        // Se indica el contexto para RV en forma de lista.
        binding.recyclerView.layoutManager = LinearLayoutManager(this)

        // Se genera el adapter.
        myAdapter = MyAnimalAdapter(getAnimals())

        // Se asigna el adapter al RV.
        binding.recyclerView.adapter = myAdapter
    }

    // Método encargado de generar la fuente de datos.
    private fun getAnimals(): MutableList<MyAnimal> {
        val animals: MutableList<MyAnimal> = arrayListOf()

        animals.add(MyAnimal("Cisne", "Cygnus olor", R.drawable.cisne))
        animals.add(MyAnimal("Erizo", "Erinaceinae", R.drawable.erizo))
        animals.add(MyAnimal("Gato", "Felis catus", R.drawable.gato))
        animals.add(MyAnimal("Gorrión", "Passer domesticus", R.drawable.gorrion))
        animals.add(MyAnimal("Mapache", "Procyon", R.drawable.mapache))
        animals.add(MyAnimal("Oveja", "Ovis aries", R.drawable.oveja))
        animals.add(MyAnimal("Perro", "Canis lupus familiaris", R.drawable.perro))
        animals.add(MyAnimal("Tigre", "Panthera tigris", R.drawable.tigre))
        animals.add(MyAnimal("Zorro", "Vulpes vulpes", R.drawable.zorro))

        return animals
    }
}
```

El resultado que se debería obtener sería algo muy parecido a la imagen que muestra en la figura, fíjate que se muestra el *toast* con el nombre del elemento que ha sido pulsado.

Observa que, al hacer clic sobre cualquier elemento de la lista resultante, sobre cualquier parte del ítem, se lanzará un *toast* indicando el nombre del animal. Esto se consigue mediante el código utilizado en el método `bind()` de la clase `ViewHolder` dentro de la clase `RecyclerAdapter`.

Si en vez de un *ListView*, prefieres utilizar un G*ridView*, vista en modo cuadrícula, únicamente deberás modificar la línea en la que se indica el contexto con forma de lista por esta otra que se muestra continuación, en la que se establece el contexto en forma de rejilla.

```
// Se indica el contexto para RV en forma de grid.
binding.myRVAnimals.layoutManager = GridLayoutManager(this, 2)
```

El segundo parámetro de la clase `GridLayoutManager()` se utilizará para indicar el número de columnas que se quieras mostrar en el *RecyclerView*.

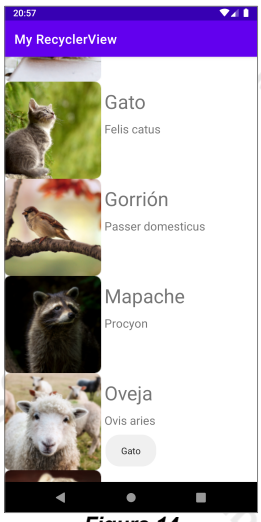

Figura 14

Los ***CardView***, como cualquier *ViewGroup*, se podrán añadir a los *layouts*, *activities* o *fragments* (que se verán más adelante). Para añadir un *CardView*, al igual que con *RecyclerView*, si fuese necesario, se importará la librería, pero ya se encarga de eso **Android Studio**.

La principal ventaja de utilizar los *CardView*, además de agrupar los elementos en modo tarjeta, es la posibilidad de manipular el sombreado y elevación del *layout*, así como la posibilidad de redondear las esquinas, destaca por su personalización estética.

Utilizando el código visto para el *RecyclerView*, para añadir un *CardView*, únicamente tendrás que modificar el fichero `item_animal_list.xml` de la siguiente forma.

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.cardview.widget.CardView
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_margin="5dp"
    android:clickable="true"
    android:focusable="true"
    android:foreground="?android:attr/selectableItemBackground"
    android:padding="15dp"
    app:cardCornerRadius="10dp"
    app:cardElevation="8dp">

    <RelativeLayout
        android:layout_width="match_parent"
        ...
    </RelativeLayout>
</androidx.cardview.widget.CardView>
```

A continuación, se comentarán las propiedades que pueden ayudar en el diseño de los *CardView* consiguiendo así el efecto de tarjeta.

- **app:cardCornerRadius**, permite especificar el redondeo de las esquinas.

- **app:cardElevation**, eleva la *CardView*, a mayor elevación, mayor proyección de la sombra.
- Por defecto, las *CardView* no tienen el efecto *Ripple*, para ello se deberán añadir los dos siguientes atributos,

```
android:clickable="true"
android:foreground="?android:attr/selectableItemBackground"
```

Estos dos atributos se pueden añadir a otro tipo de elementos de un *layout*, como un *TextView*, un *ImageView*, etc, y poder aplicar el efecto.

¿Qué es el efecto *Ripple*? Es un *feedback*, devuelve una respuesta, generada sobre el elemento para dar sensación de movimiento sobre dicho elemento al ser pulsado.

El resultado que se obtendría debería ser similar al que se muestra en la imagen de la figura.

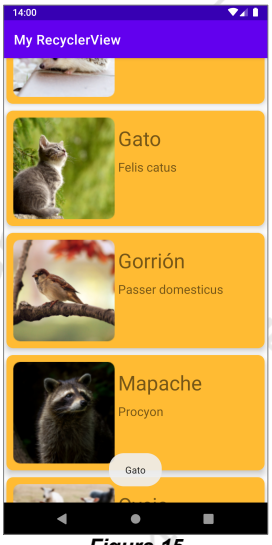

Figura 15

5.8. Actualizar el Adapter del RecyclerView

La interacción con un *RecyclerView* va más allá de un simple listado, en ocasiones, puede utilizarse para eliminar elementos de la lista, marcar favoritos, dar paso a detalles, etc.

A continuación, se mostrarán una serie de métodos mediante los cuales se puede actualizar el listado, actualizando los ítems mostrados y por ende, la fuente de datos, esto es, actualizando el adaptador. En primer lugar, si es un listado básico como el visto en el ejemplo *"My Basic RecyclerView"*, visto en el punto 5.7.1., donde todo está en una misma clase, puede hacerse añadiendo el siguiente código a la *inner class ViewHolder*, desde la cual, cuando se produzca una pulsación larga sobre un elemento, éste se eliminará de la lista. Para simplificar se eliminará directamente, pero lo ideal sería confirmar la eliminación o mostrar un *SnackBar* con posibilidad de deshacer los cambios.

```
inner class ViewHolder(view: View) : RecyclerView.ViewHolder(view) {
    val textField: TextView = view.findViewById(android.R.id.text1)

    init {
        ...
        itemView.setOnLongClickListener {
            datos.removeAt(adapterPosition) // Se elimina el dato de la fuente de datos.

            // Actualiza el Adapter.
            binding.recyclerView.adapter!!.notifyItemRemoved(adapterPosition)
            true
        }
    }
}
```

Como puedes observar, el *Adapter* dispone de los métodos *notify* que te permitirán actualizar el adaptador según sea la necesidad. En este ejemplo se ha utilizado `notifyItemRemoved()`, para indicar el elemento (la posición obtenida mediante *adapterPosition*) que se ha eliminado del listado.

Te habrás dado cuenta que, lo que realmente se actualiza es el *Adapter*, esto puede suponer un problema cuando se crea una clase independiente para gestionar el *RecyclerVew*, ya que desde esta, no hay acceso directo al *Adapter*. En esto casos, deberá optarse por otra estrategia.

Fíjate como podría resolverse esta situación para el ejemplo *"My RecyclerView"* desarrollado en el punto 5.7.2.

La idea, que no la única, es crear un *interface* que deberá implementar la clase encargada de gestionar el adaptador, manejando el evento desde ésta, accediendo así directamente al *adapter*. El primer paso será crear en la clase `RecyclerAdapter` el interface, un método (público) encargado de capturar el evento desde la clase *parent*, y se instanciará un objeto sobre la interface para gestionarlo en el adaptador.

```
class RecyclerAdapter(animalsList: MutableList<MyAnimal>) :
    RecyclerView.Adapter<RecyclerAdapter.ViewHolder>() {
    private var animals: MutableList<MyAnimal>

    // Se instancia la interface para el LongClick.
    private lateinit var mLongClickListener: ItemLongClickListener

    // Interface que debe implementar la clase que use el adaptador.
    interface ItemLongClickListener {
        // Este método recibe la vista sobre la que se ha pulsado y la posición.
        fun onItemLongClick(view: View, position: Int)
    }

    // Método encargado de capturar el evento, se instancia en el adaptar.
    fun setLongClickListener(itemLongClickListener: ItemLongClickListener?) {
        mLongClickListener = itemLongClickListener!!
    }
    ...
```

El siguiente paso será activar la pulsación larga sobre el `itemView`, en el método `bind()` de la clase `ViewHolder`.

```
itemView.setOnLongClickListener {
    binding.root.setBackgroundColor(Color.RED)
    mLongClickListener.onItemLongClick(it, adapterPosition)

    true
}
```

Recuerda que el objeto `adapterPosition` devuelve la posición del ítem en el listado. A continuación, en la clase `MainActivity`, que es la clase *parent* del adaptador, deberá extenderse de este nuevo *adapter*.

```
class MainActivity : AppCompatActivity(), RecyclerAdapter.ItemLongClickListener {
```

Esto producirá un error que se solucionará sobrecargando el *interface* que se acaba de crear.

```
override fun onItemLongClick(view: View, position: Int) {
    TODO("Not yet implemented")
}
```

También se cambiará la forma de obtener la fuente de datos, creando una *MutableList* sobre la que se trabajará.

```
companion object {
    private var listAnimals: MutableList<MyAnimal> = MainActivity().getAnimals()
}
```

Ahora habrá que añadir la siguiente línea en la creación del adaptador antes de asignárselo al *RecyclerView*.

```
myAdapter = RecyclerAdapter(listAnimals)
myAdapter.setLongClickListener(this)
```

El método sobrecargado deberá encargase de la operación, pudiendo quedar de la siguiente forma, actualizando la fuente de datos y refrescando la vista del *Recycler*.

```
override fun onItemLongClick(view: View, position: Int) {
    Snackbar.make(
        binding.root,
        "¿Confirmas el borrado?",
        Snackbar.LENGTH_LONG
    ).setAction("Sí") {
        listAnimals.removeAt(position)
        myAdapter.notifyItemRemoved(position)
    }.show()
}
```

Con esto ya tendrías un *RecyclerView* del que poder eliminar ítems mediante una pulsación larga sobre el elemento.

Ahora bien, no siempre hace falta añadir una interface al adaptador del *RecyclerView*, podría hacerse esta misma operación desde el propio adaptador, quedando la clase `RecyclerAdapter` como se muestra a continuación.

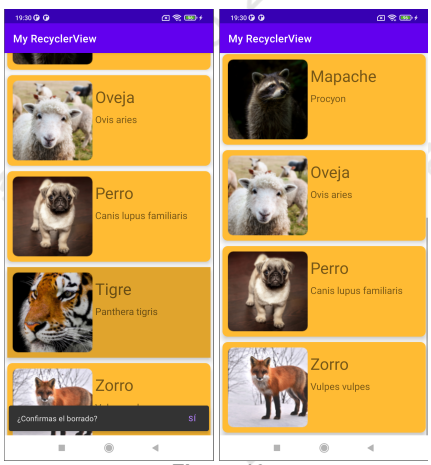

Figura 16

```
class RecyclerAdapter(val animalsList: MutableList<MyAnimal>) :
    RecyclerView.Adapter<RecyclerAdapter.ViewHolder>() {

    // Es el encargado de devolver el ViewHolder ya configurado.
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder {
        val layoutInflater = LayoutInflater.from(parent.context)
        return ViewHolder(
            ItemAnimalListBinding.inflate(layoutInflater, parent, false).root
        )
    }

    // Método encargado de pasar los objetos, uno a uno, al ViewHolder personalizado.
    override fun onBindViewHolder(holder: ViewHolder, position: Int) {
        holder.bind(animalsList.get(position))
    }

    // Devuelve el tamaño de la fuente de datos.
    override fun getItemCount() = animalsList.size

    // Esta clase interna se encarga de rellenar cada una de las vistas que
    // se inflarán para cada uno de los elementos del RecyclerView.
    inner class ViewHolder(view: View) : RecyclerView.ViewHolder(view) {
        // Se usa View Binding para localizar los elementos en la vista.
        private val binding = ItemAnimalListBinding.bind(view)

        fun bind(animal: MyAnimal) {
            binding.tvNameAnimal.text = animal.animalName
            binding.tvLatinName.text = animal.latinName
            Glide.with(binding.root)
                .load(animal.imageAnimal)
                .override(150, 150)
                // Centra la imagen y redondea las esquinas.
                .transform(CenterCrop(), RoundedCorners(10))
                .into(binding.ivAnimalImage)

            itemView.setOnClickListener {
                Toast.makeText(
                    binding.root.context,
                    animal.animalName,
                    Toast.LENGTH_SHORT
                ).show()
            }

            itemView.setOnLongClickListener {
                Snackbar.make(
                    binding.root,
                    "¿Confirmas el borrado?",
                    Snackbar.LENGTH_LONG
                ).setAction("Sí") {
```

```
                    animalsList.removeAt(adapterPosition)
                    notifyItemRemoved(adapterPosition)
                }.show()

                true
            }
        }
    }
}
```

Existen varias formas de actualizar un adaptador, bastante intuitivas según su nombre, se dispone de los métodos *notifyItemRemoved*, *notifyItemChanged*, *notifyItemInserted*, *notifyItemMoved* y *notifyDataSetChanged*, este último, no muy recomendado, actualiza el adaptador completo. También se dispone de sus variantes para rangos.

Se puede sustituir el uso de la interface y hacer uso de una de las propiedades de Kotlin, aprovechando así su potencia, la idea será utilizar una función como parámetro, lo que permitirá reducir la complejidad. En primer lugar, cambiará la declaración del adaptador del *RecyclerAdapter*, añadiendo un nuevo parámetro.

```
class RecyclerAdapter(
    animalsList: MutableList<MyAnimal>,
    private val onAnimalLongClick: (MyAnimal, pos: Int) -> Unit
) : RecyclerView.Adapter<RecyclerAdapter.ViewHolder>() {
    ...
```

Esta nueva variable, ***onAnimalLongClick***, contiene dos parámetros, el objeto pulsado y la posición que ocupa en la lista. En la clase ViewHolder se controlará el evento de la siguiente forma.

```
itemView.setOnLongClickListener {
    onAnimalLongClick(animal, adapterPosition)
    true
}
```

Ahora, desde la clase que carga el adaptador, el código para crearlo será algo parecido a lo que se muestra a continuación.

```
// Se genera el adapter.
myAdapter = RecyclerAdapter(listAnimals,
    onAnimalLongClick = { animal, pos ->
        Snackbar.make(
            binding.root,
            "¿Confirmas el borrado?",
            Snackbar.LENGTH_LONG
        ).setAction("Sí") {
            listAnimals.remove(animal)
            myAdapter.notifyItemRemoved(pos)
        }.show()
    })
```

No hay que registrar el evento como con el uso de la interface y permite mayor versatilidad para el adaptador creado.

5.8.1. Animaciones del RecyclerView

Como habrás comprobado, al interactuar con el *RecyclerView* se reproducen una serie de animaciones para que el efecto sea más agradable, pero en ocasiones puede ser que no interese tal efecto. Por ejemplo, si para marcar un elemento del listado como favorito, o que "me gusta", y no quieres que se vea el efecto que se produce al actualizar el elemento, puedes modificar las siguientes propiedades del ***ItemAnimator*** asociado al *RecyclerView*.

```
binding.mRecyclerView.itemAnimator?.apply {
    changeDuration = 0
    addDuration = 0
    moveDuration = 0 // 250ms por defecto
    removeDuration = 0 // 0ms para evitar animaciones
}
```

Con estas líneas puedes modificar los tiempos, o desactivar, la animación que te interese, también puedes directamente desactivarlo con:

```
binding.mRecyclerView.itemAnimator = null
```

Para volver a activar todas las animaciones, puedes volver a ponerlas todas a 250, su valor por defecto, o utilizar la siguiente línea para que vuelva a su estado normal.

```
binding.mRecyclerView.itemAnimator = DefaultItemAnimator()
```

5.9. Gestos del RecyclerView

Existen una serie de gestos, conocidos por todos, que pueden aplicarse a los *RecyclerView*, como son deslizar hacia abajo, generalmente para actualizar, desplazar lateralmente un elemento o simplemente conocer cómo se está moviendo el listado. Comenzaremos por éste último evento.

5.9.1. Detectar el movimiento del scroll

Esta operación puede resultar interesante, es común utilizarla para ocultar o mostrar elementos según el desplazamiento que se esté realizando, por ejemplo un botón flotante. En este caso, lo que se hará será añadir un *listener* al *scroll*.

```
binding.myRVAnimals.addOnScrollListener(object : RecyclerView.OnScrollListener() {

    override fun onScrolled(recyclerView: RecyclerView, dx: Int, dy: Int) {
        super.onScrolled(recyclerView, dx, dy)
        if (dy > 0)
            Log.d("SCROLL", "Down")
        else if (dy < 0)
```

```
            Log.d("SCROLL", "Up")
        }
})
```

El movimiento corresponde con el desplazamiento de la barra de desplazamiento, es decir, cuando salta el *scroll down* es porque la barra se desplaza hacia abajo, y viceversa.

5.9.2. Añadir Swipe Refresh

El *"Swipe to Refresh"* es el moviendo que se hace deslizando hacia abajo, cuando la lista está al inicio, para lanzar el refresco sobre el listado, sobre el adaptador. Para utilizar este elemento, en primer lugar habrá que añadir la siguiente dependencia al `build.gradle (Module:app)`.

```
implementation "androidx.swiperefreshlayout:swiperefreshlayout:1.1.0"
```

A continuación, en el *layout* que contenga el *RecyclerView*, habrá que "envolver" este elemento con el nuevo componente añadido.

```
<androidx.swiperefreshlayout.widget.SwipeRefreshLayout
    android:id="@+id/swipeRefresh"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

    <androidx.recyclerview.widget.RecyclerView
        android:id="@+id/myRVAnimals"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:scrollbars="vertical"
        tools:listitem="@layout/item_animal_list" />
</androidx.swiperefreshlayout.widget.SwipeRefreshLayout>
```

Este componente únicamente sirve para contener *RecyclerViews*, no se aplica en otros elementos. Si lanzas la aplicación y haces el gesto, verás que aparece una barra de progreso, pero ya no desaparece, esto habrá que gestionarse desde código.

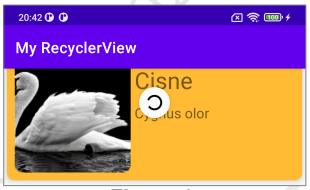

Figura 17

Siguiendo con el ejemplo de los animales, ya que se pueden eliminar elementos del listado, mediante el refresco se hará que vuelvan a aparecer todos. En la clase `MainActivity`, en el método *onStart* se añadirá el siguiente código para tal efecto.

```
override fun onStart() {
    super.onStart()

    binding.swipeRefresh.setOnRefreshListener {
        // Se limpia la lista.
        listAnimals.clear()

        // Se vuelven a añadir todos los elementos.
```

```
        listAnimals.addAll(getAnimals())

        // Se actualiza el adapter.
        myAdapter.notifyDataSetChanged()

        // Se desactiva el refresco.
        binding.swipeRefresh.isRefreshing = false
    }
}
```

Ahora podrás eliminar todos los ítems que quieras del listado y al refrescar, se volverá a recargar la lista de elementos.

5.9.3. Arrastrar y deslizar

Otro movimiento muy común es el desplazamiento lateral de los elementos de un *RecyclerView*. Conocido como *Drag&Swipe*, el *drag*, arrastre, que se produce manteniendo pulsado el elemento de la lista y una vez activado, llevarlo hacia el punto que esté configurado, el *swipe*, se producirá al deslizar en la dirección configurada el elemento de la lista.

Para poder aplicar estos gestos se hará uso de la clase `ItemTouchHelper` de la siguiente manera, pero no la única. Esta clase permite implementar un *callback* para recoger el producido al deslizar. Siguiendo con el ejemplo de los animales, al final del método *setUpRecyclerView* se añadirán las siguientes líneas.

```
ItemTouchHelper(object :
    ItemTouchHelper.SimpleCallback(ItemTouchHelper.UP, ItemTouchHelper.RIGHT) {

    override fun onMove(
        recyclerView: RecyclerView,
        viewHolder: RecyclerView.ViewHolder,
        target: RecyclerView.ViewHolder
    ): Boolean {
        // Se obtienen las posiciones de inicio y fin.
        val fromPos = viewHolder.adapterPosition
        val toPos = target.adapterPosition

        Log.d("onMove", "Movimiento from $fromPos to $toPos")
        myAdapter.notifyItemMoved(fromPos, toPos)

        return true
    }

    override fun onSwiped(viewHolder: RecyclerView.ViewHolder, direction: Int) {
        // Posición del elemento sobre el que se actúa.
        val position = viewHolder.adapterPosition

        // Se almacena el elemento a borrar temporalmente.
```

```
        val itemDeleted = listAnimals[position]

        // Se borra y actualiza el adapter.
        listAnimals.removeAt(position)
        myAdapter.notifyItemRemoved(position)

        Snackbar.make(
            binding.root,
            "${itemDeleted.animalName} eliminado.",
            Snackbar.LENGTH_LONG
        ).setAction("Deshacer") {
            listAnimals.add(position, itemDeleted)
            myAdapter.notifyItemInserted(position)
        }.show()
    }
}).attachToRecyclerView(binding.myRVAnimals)
```

En este ejemplo se hace uso del método ***SimpleCallback*** para hacerlo fácil, fíjate que se le pasan dos parámetros, que son de tipo entero y se utilizan las constantes de la clase *ItemTouchHelper*, en este caso UP y RIGHT. El primero indica el movimiento que se activará para el *drag*, el segundo, el movimiento para el *swipe*.

También es necesario sobrecargar los métodos ***onMove***, que se utilizará para el *drag*, y ***onSwiped***, que se utiliza para el deslizado lateral de ítem. En el primero simplemente se hace un cambio de posición actualizando el adaptador, en el segundo se elimina un elemento y haciendo uso de un *SnackBar* se de la opción de deshacer la acción.

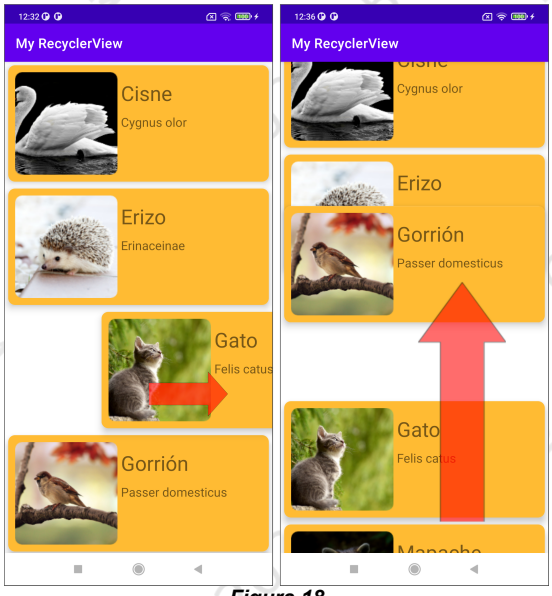

Figura 18

Como puedes ver en la imagen, al deslizar cualquiera de los ítems, el fondo bajo este queda en blanco. Cuando se realiza un *swipe* suele mostrarse el fondo de otro color y con una imagen que identifique la acción que se está llevando a cabo.

Figura 19

Para conseguir este resultado, dentro del *callback* creado para *ItemTouchHelper*, deberá sobrecargarse el método `onChildDraw` de la siguiente forma.

```
override fun onChildDraw(
    c: Canvas, recyclerView: RecyclerView, viewHolder: RecyclerView.ViewHolder,
    dX: Float, dY: Float, actionState: Int, isCurrentlyActive: Boolean
) {
    super.onChildDraw(c, recyclerView, viewHolder, dX, dY, actionState, isCurrentlyActive)

    if (dX > 0) { // Se comprueba que el itemView se está moviendo.
        // Se crea el icono a mostrar.
        val iconTrash: Drawable? =
            AppCompatResources.getDrawable(applicationContext, R.drawable.delete_sweep)
        // Se establece el margen izquierdo en base a la altura del itemView y la altura
        // máxima mediante la cual puede representarse el icono. Al dividirse entre 2, se
        // obtiene el valor para que se encuentre justo a la misma distancia del margen
        // izquierdo que del margen superior e inferior.
        val leftMargin = (viewHolder.itemView.height - iconTrash!!.intrinsicHeight) / 2

        // Se obtiene la posición de la esquina superior izquierda que tendrá el icono.
        val iconTopLeftCorner: Int =
            viewHolder.itemView.top + (viewHolder.itemView.height -
                    iconTrash.intrinsicHeight) / 2

        // Se obtiene la posición de la esquina inferior izquierda que tendrá el icono.
        val iconBottomLeftCorner: Int = iconTopLeftCorner + iconTrash.intrinsicHeight

        // Se obtien el margen izquierdo para el icono.
        val iconLeftMargin: Int = viewHolder.itemView.left + leftMargin

        // Se obtien el margen derecho para el icono.
        val iconRightMargin: Int =
            viewHolder.itemView.left + leftMargin + iconTrash.intrinsicWidth

        // Se asignan las medidas al icono en base a los datos obtenidos.
        iconTrash.setBounds(
            iconLeftMargin, iconTopLeftCorner,
```

```
            iconRightMargin, iconBottomLeftCorner
        )

        // Se crea un color para fondo que se pintará.
        val fondo = ColorDrawable(Color.RED)

        // Se establecen las medidas.
        fondo.setBounds(
            viewHolder.itemView.left,
            viewHolder.itemView.top,
            viewHolder.itemView.left + dX.toInt() + 20, // Pinta según se desplaza la X.
            viewHolder.itemView.bottom
        )

        // Se pintan en el canvas por este orden.
        fondo.draw(c)
        iconTrash.draw(c)
    }
}
```

Como puedes ver en este método se han realizado una serie de operaciones para conseguir la mejor posición posible, puedes evitarlo poniendo valores fijos a ojo, pero no quedaría tan bien frente a posibles cambios de tamaño.

Al terminar, te puedes encontrar un *callback* bastante grande, por lo que puede resultar interesante sacarlo y crear una clase independiente para su implementación.