IES DR LLUÍS SIMARRO
ENSENYAMENT PÚBLIC
XATIVA

Programación multimedia y dispositivos móviles

UD05. Controles de selección

Desarrollo de Aplicaciones Multiplataforma

Anna Sanchis Perales
Enric Giménez Ribes

ÍNDICE

1. OBJETIVOS

Los objetivos a conocer y comprender son:

- Controlar la mayoría de las opciones que se disponen para crear una interfaz en Android utilizando las listas de selección.
- Conocer los adaptadores y su uso.
- Crear listas desplegables mediante el componente Spinners.
- Crear listas dinámicas mediante el nuevo componente, RecyclerView.
- Manejar eventos tanto en las listas desplegables como en listas dinámicas.

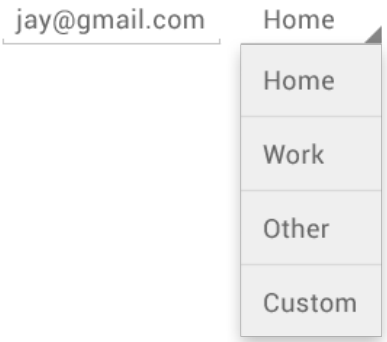
2. INTRODUCCIÓN

Las **listas** en Android son **contenedores** muy útiles para **organizar información** en forma **vertical** y con la capacidad de usar **scrolling** (desplazamiento) para simplificar su visualización.

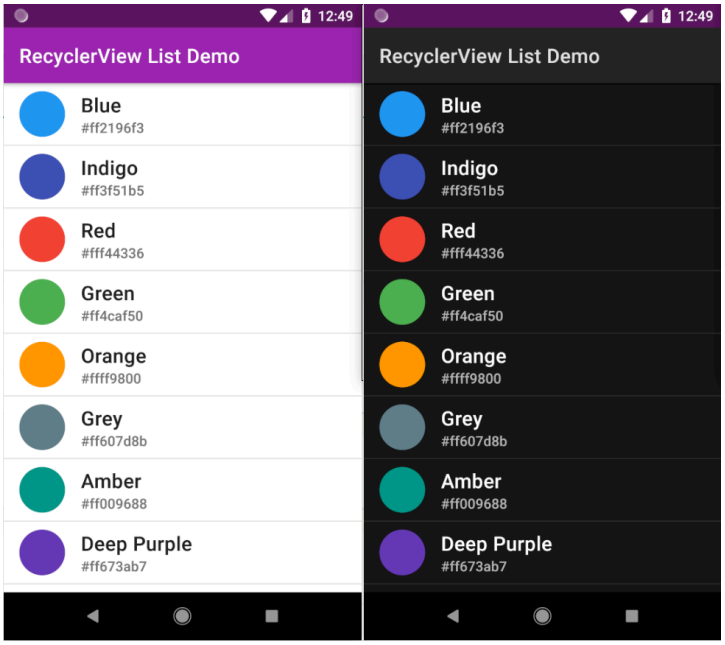
Esta técnica es muy popular en muchas aplicaciones, ya que permite mostrar al usuario un conjunto de datos de forma práctica y accesible.

Google liberó un nuevo componente alternativo llamado RecyclerView para crear listas y grid.

En la presente unidad hablaremos de las listas tipo desplegable, conocidas como **Spinner:**



Y del nuevo componente llamado **RecyclerView**:



3. ADAPTADORES

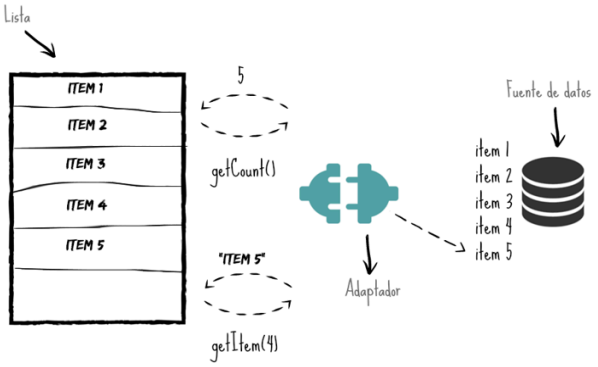
Un adaptador es un objeto que comunica a una lista los datos necesarios para crear las filas de la lista. Es decir, conecta la lista con una fuente de información como si se tratase de una adaptador de corriente que alimenta a un televisor. Además de proveer la información, **también genera los Views para cada elemento de la lista.** Mejor dicho, hace todo, solo que la lista es quién termina mostrando la información final.

Los adaptadores se representan programáticamente por la clase Adapter. Dependiendo de la naturaleza de la lista se elegirá el tipo de adaptador. Existen subclases de la clase Adapter proveídas por Android, que facilitan la mayoría de casos al poblar una lista. Pero si no satisfacen tus deseos, entonces puedes extenderlas para personalizar su comportamiento a tu gusto.

Cuando referenciamos un adaptador con una lista inmediatamente comienza un proceso de comunicación interno para poblar la lista con los datos correspondientes. Dicha comunicación se basa principalmente en los siguientes métodos del adaptador:

- **getCount():** retorna en la cantidad de elementos que tiene un adaptador. Con este valor la lista ya puede establecer un límite para añadir filas.
- **getItem():** obtiene un elemento de la fuente de datos asignada al adaptador en una posición establecida. Normalmente la fuente de datos es un arreglo o lista de objetos.
- **getView():** retorna el View elaborado e inflado de un elemento en una posición específica.

Aunque estos tres métodos no son los únicos que existen para establecer la relación, a mi parecer son los más significativos para entender el concepto de un adaptador.



Android proporciona, de serie, varios tipos de adaptadores sencillos, aunque podemos extender su funcionalidad fácilmente para adaptarlos a nuestras necesidades. Los más comunes son los siguientes:

- **ArrayAdapter:** Es el más sencillo de todos los adaptadores, y provee de datos a un control de selección a partir de un array de objetos de cualquier tipo.
- **SimpleAdapter:** Se utiliza para mapear datos sobre los diferentes controles definidos en un fichero XML de layout.
- **SimpleCursorAdapter:** Se utiliza para mapear las columnas de un cursor abierto sobre una base de datos sobre los diferentes elementos visuales contenidos en el control de selección.

4. SPINNER

Las listas desplegables en Android se llaman Spinner. Su funcionamiento es el siguiente, el usuario selecciona la lista, se muestra una especie de lista emergente con todas las opciones disponibles y al seleccionarse una de ellas ésta queda fijada en el control. Para añadir una lista de este tipo a nuestra aplicación vamos a comenzar creando una:

1. Creamos una nueva aplicación denominada **Tema5App1**.
2. Modificamos el layout activity_main.xml para añadir un elemento spinner. El código del layout quedará de la siguiente forma:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
   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="match_parent"
   android:layout_height="match_parent"
   tools:context=".MainActivity">

   <TextView
       android:id="@+id/textView"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content"
       android:layout_marginTop="16dp"
       android:text="Ejemplo Spinner"
       android:textSize="24sp"
       android:textStyle="bold"
       app:layout_constraintEnd_toEndOf="parent"
       app:layout_constraintHorizontal_bias="0.5"
       app:layout_constraintStart_toStartOf="parent"
       app:layout_constraintTop_toTopOf="parent" />

   <Spinner
       android:id="@+id/cmbOpciones"
       android:layout_width="match_parent"
       android:layout_height="wrap_content"
       android:layout_marginStart="16dp"
       android:layout_marginTop="16dp"
       android:layout_marginEnd="16dp"
       app:layout_constraintEnd_toEndOf="parent"
       app:layout_constraintHorizontal_bias="0.5"
       app:layout_constraintStart_toStartOf="parent"
       app:layout_constraintTop_toBottomOf="@+id/textView" />

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

3. Poco vamos a comentar de aquí ya que lo que nos interesan realmente son los datos a mostrar. Las opciones para personalizar el aspecto visual del control (fondo, color y tamaño de fuente…) son las mismas ya comentadas para los controles básicos.
4. Ahora vamos a enlazar nuestro adaptador (recomendamos leer el punto anterior donde se explica que es una adaptador) a este control. Comenzaremos definiendo un adaptador con los datos.

```
// Datos para el Spinner
val datos = arrayOf("Opción 1", "Opción 2", "Opción 3")

// Crear un ArrayAdapter usando los datos y un diseño predeterminado
val adapter = ArrayAdapter(this, android.R.layout.simple_spinner_item, datos)
```

Comentamos un poco el código. Sobre la primera línea no hay nada que decir, es tan sólo la definición del array Kotlin que contendrá los datos a mostrar en el control, en este caso un array sencillo con tres cadenas de caracteres.

En la segunda línea creamos el adaptador en sí, al que pasamos 3 parámetros:

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

Con esto ya tendríamos creado nuestro adaptador para los datos a mostrar y ya tan sólo nos quedaría asignar este adaptador a nuestro control de selección para que éste mostrase los datos en la aplicación.

5. Ahora declararemos la variable que recoge el Spinner.

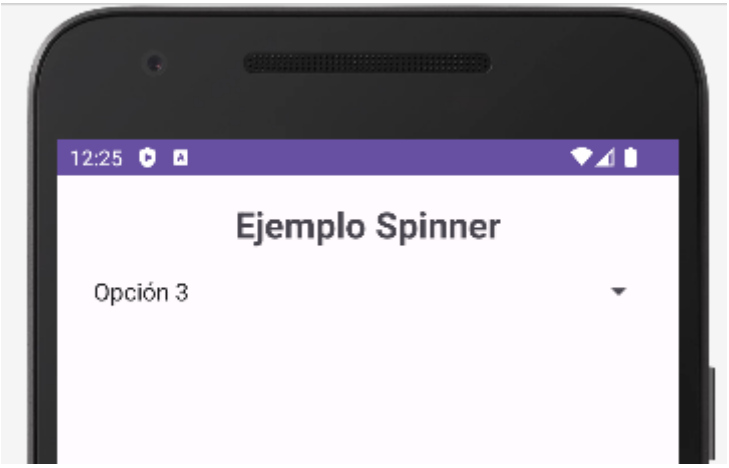
```
val spinner = binding.cmbOpciones
```

```
// Especificar el diseño del menú desplegable
adapter.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item)

// Asignar el ArrayAdapter al Spinner
spinner.adapter = adapter
```

Comenzamos como siempre por obtener una referencia al control a través de binding. Y en la última línea asignamos el adaptador al control spinner. ¿Y la segunda línea para qué es? Cuando indicamos en el apartado posterior cómo construir un adaptador vimos cómo uno de los parámetros que le pasábamos era el ID del layout que utilizamos para visualizar los elementos del control. Sin embargo, en el caso del control Spinner, este layout tan sólo se aplicará al elemento seleccionado en la lista, es decir, al que se muestra directamente sobre el propio control cuando no está desplegado. Sin embargo, antes indicamos que el funcionamiento normal del control Spinner incluye entre otras cosas mostrar una lista emergente con todas las opciones disponibles. Pues bien, para personalizar también el aspecto de cada elemento en dicha **lista emergente** tenemos el método **setDropDownViewResource**(ID_layout), al que podemos pasar otro ID de layout distinto al primero sobre el que se mostrarán los elementos de la lista emergente. En este caso hemos utilizado otro layout predefinido en Android para las listas desplegables (android.R.layout.simple_spinner_dropdown_item), formado por una etiqueta de texto con la descripción de la opción.

6. Ejecutamos la aplicación y obtendremos, con estas simples líneas de código, mostrar un control como el que vemos en la siguiente imagen:



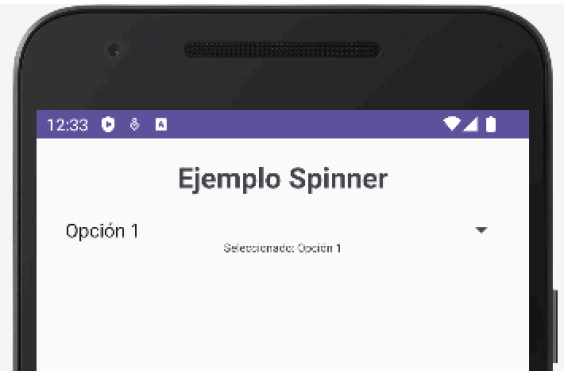
7. Ahora vamos a añadir más cosas, situamos por debajo del Spinner un elemento TextView, el cual tendrá de id=lblMensaje.
8. Capturamos en el onCreate el elemento lblMensaje en una variable de tipo TextView.
9. En cuanto a los eventos lanzados por el control Spinner, el más comúnmente utilizado será el generado al seleccionarse una opción de la lista desplegable, **onItemSelected**. Para capturar este evento se procederá de forma similar a lo ya visto para otros controles anteriormente, asignado el controlador mediante el método setOnItemSelectedListener(). Por ello asignamos el siguiente código:

```
// Manejar la selección de elementos del Spinner
spinner.setOnItemSelectedListener(object : AdapterView.OnItemSelectedListener {
   override fun onItemSelected(parent: AdapterView<*>?, view: View?, position: Int,
id: Long) {
       val seleccion = datos[position]
       binding.lblMensaje.text = "Seleccionado: $seleccion"
   }

   override fun onNothingSelected(parent: AdapterView<*>?) {
       // Acciones a realizar si no se selecciona nada
       binding.lblMensaje.text = ""
   }
})
```

Para este evento definimos dos métodos, el primero de ellos (**onItemSelected**) que será llamado cada vez que se selecciona una opción en la lista desplegable, y el segundo (**onNothingSelected**) que se llamará cuando no haya ninguna opción seleccionada (esto puede ocurrir por ejemplo si el adaptador no tiene datos). Además, vemos cómo para recuperar el dato seleccionado utilizamos el método getItemAtPosition() del parámetro AdapterView que recibimos en el evento.

10. Ejecutamos la aplicación y obtenemos el siguiente resultado:

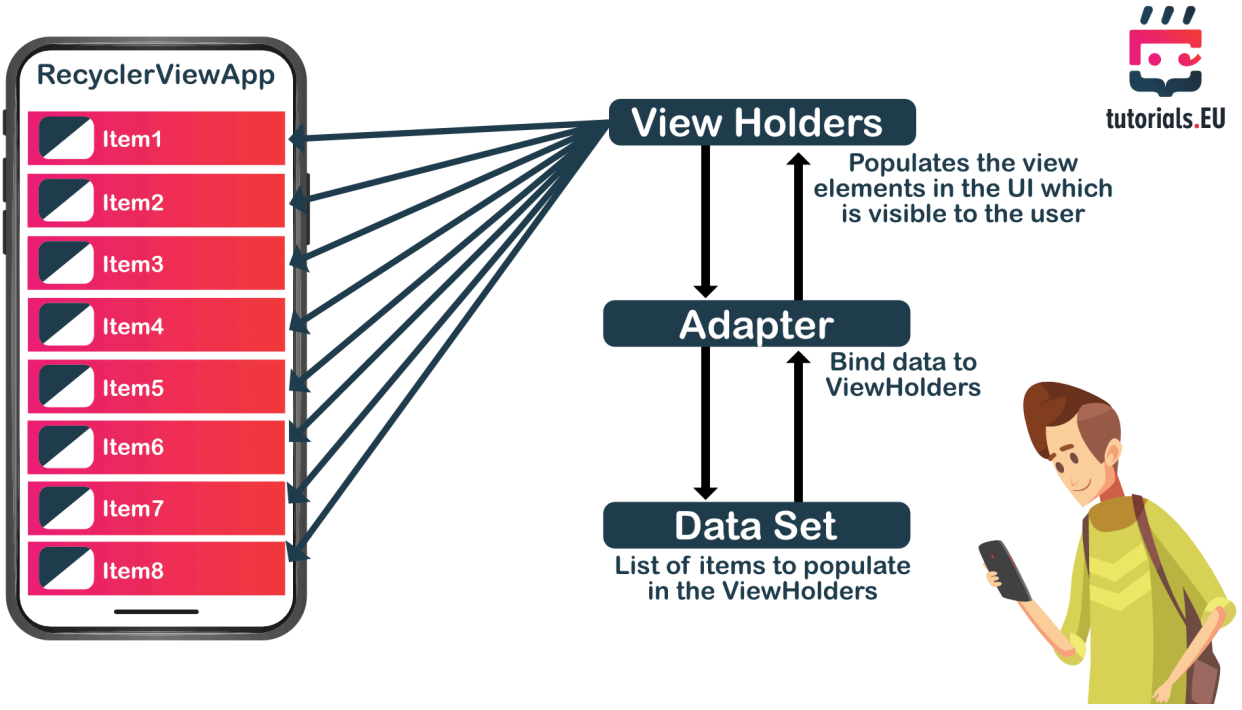


5. RECYCLERVIEW

El control **RecyclerView** nos ayudará a crear controles de tipo lista y tabla. Este control llegó con Android 5.0 como una mejor alternativa a los antiguos controles ListView (lista) y GridView (tabla) de versiones anteriores, aportando en la mayoría de ocasiones flexibilidad de sobra para suplir la funcionalidad de ambos, e ir incluso más allá.

RecyclerView nos va a permitir mostrar en pantalla colecciones de datos. Y es que RecyclerView no va a hacer «casi nada» por sí mismo, sino que se va a sustentar sobre otros componentes complementarios para determinar cómo acceder a los datos y cómo mostrarlos. Los más importantes serán los siguientes:

- **RecyclerView.ViewHolder** → Hace referencia a los objetos visuales.
- **RecyclerView.Adapter** → Configura cada una de las vistas a partir de los datos.
- **LayoutManager** → Distribución de la vista principal.
- **ItemDecoration** → Separadores de elementos.
- **ItemAnimator** → Animaciones automáticas.



5.1 RecyclerView.Adapter / RecyclerView.ViewHolder

Un RecyclerView se apoyará en un adaptador para trabajar con nuestros datos, en este caso un adaptador que herede de la clase **RecyclerView.Adapter**. La peculiaridad en esta ocasión es que este tipo de adaptador utilizará internamente el patrón view holder que dotará de una mayor eficiencia al control, y de ahí la necesidad del segundo componente de la lista anterior, **RecyclerView.ViewHolder**.

Por resumirlo de alguna forma, un view holder se encargará de contener y gestionar las vistas o controles asociados a cada elemento individual de la lista. El control RecyclerView se encargará de crear tantos view holder como sea necesario para mostrar los elementos de la lista que se ven en pantalla y los gestionará eficientemente de forma que no se tengan que crear nuevos objetos para mostrar más elementos de la lista al hacer scroll, sino que tratará de «reciclar» aquellos que ya no sirven por estar asociados a otros elementos de la lista que ya han salido de la pantalla.

5.2 LayoutManager

Anteriormente, cuando decidíamos utilizar un ListView ya sabíamos que nuestros datos se representarían de forma lineal con la posibilidad de hacer scroll en un sentido u otro, y en el caso de elegir un GridView la representación sería tabular. En cambio, una vista de tipo RecyclerView no determina por sí sola la forma en que se van a mostrar en pantalla los elementos de nuestra colección, sino que va a delegar esa tarea a otro componente llamado **LayoutManager**, que también tendremos que crear y asociar al RecyclerView para su correcto funcionamiento. Por suerte, el SDK incorpora de serie algunos LayoutManager para las representaciones más habituales de los datos:

- Lista vertical y horizontal (LinearLayoutManager).
- Tabla tradicional (GridLayoutManager).
- Tabla apilada o de celdas no alineadas (StaggeredGridLayoutManager).

Por tanto, siempre que optemos por alguna de estas distribuciones de elementos no tendremos que crear nuestro propio LayoutManager personalizado, aunque por supuesto nada nos impide hacerlo, y ahí uno de los puntos fuertes de este componente: su flexibilidad.

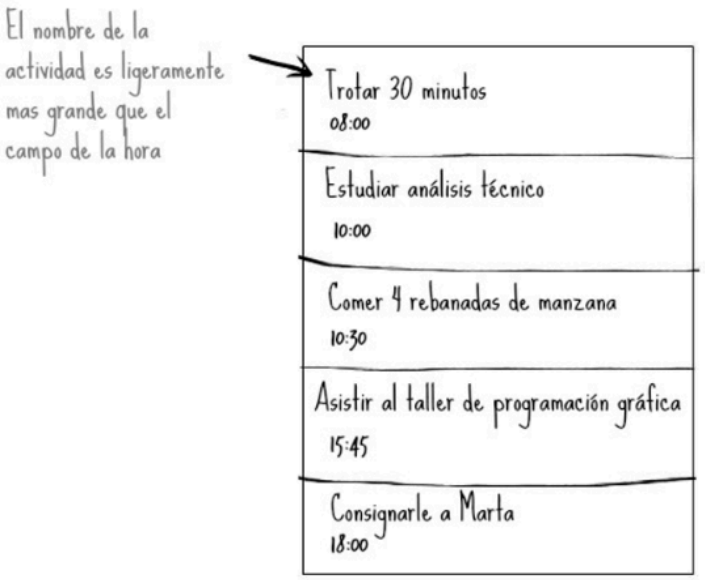
5.3 ItemDecoration e ItemAnimator

Los dos últimos componentes de la lista se encargan de definir cómo se representarán algunos aspectos visuales concretos de nuestra colección de datos (más allá de la distribución definida por el LayoutManager), por ejemplo marcadores o separadores de elementos, y de cómo se animarán los elementos al realizar determinadas acciones sobre la colección, por ejemplo al añadir o eliminar elementos.

Como hemos comentado, no siempre será obligatorio implementar todos estos componentes para hacer uso de un RecyclerView. Lo más habitual será implementar el Adapter y el ViewHolder, utilizar alguno de los LayoutManager predefinidos, y sólo en caso de necesidad crear los **ItemDecoration** e **ItemAnimator** necesarios para dar un toque de personalización especial a nuestra aplicación.

5.4 Ejemplo

Para describir el uso del control RecyclerView de la forma más completa posible, aunque sin complicarnos de forma innecesaria, vamos a crear una aplicación de ejemplo que mostrará una **lista/tabla de elementos formados cada uno de ellos por dos líneas de texto que llamaremos título y subtítulo** (por supuesto se podrían añadir muchos más elementos, por ejemplo imágenes, checkboxes… en definitiva cualquier control disponible).



Los pasos que siempre tenemos que hacer para implementar este tipo de lista son los siguientes:

1. En nuestro caso, crearemos una fuente de datos de tipo tarea (**Tarea.kt**) y crearemos de forma estática esos datos (**TareaDatos.kt**).
2. Diseñar un layout que contenga el RecyclerView (**activity_main.xml**).
3. Diseñar un layout individual de cada ítem de la lista que se repetirá (**listitem_tarea.xml**).
4. Crear un adaptador para recoger los datos y hacer la asignación a cada elemento de la lista (**DatosAdapter.kt**).
5. Implementar una pantalla o actividad (**MainActivity.kt**) que visualice el RecyclerView.
    a. Definir un manejador de layout (**LayoutManager**) que posicione la vista.

Vamos a empezar por la parte más sencilla, crearemos un nuevo proyecto en Android Studio lo guardaremos como **Tema5App2**.

Tarea.kt

Crearemos una clase que encapsula los datos que compondrán cada elemento a mostrar en la lista.

```
class Tarea(private var nombre: String, private var hora: String) {
    // Getter para obtener el nombre
    fun getNombre(): String {
        return nombre
    }

    // Setter para establecer el nombre
    fun setNombre(nuevoNombre: String) {
        nombre = nuevoNombre
    }

    // Getter para obtener la hora
    fun getHora(): String {
        return hora
    }

    // Setter para establecer la hora
    fun setHora(nuevaHora: String) {
        hora = nuevaHora
    }

}
```

TareaDatos

Ahora generamos las tareas que añadiremos a nuestro adaptador. Para ello, dentro de la clase Tarea crearemos una segunda clase TareaDatos que representa el ambiente de intercambio de datos entre el adaptador y los objetos tipo Tarea. Lo hemos generado como un companion object.

Companion object

Un companion object en Kotlin es un objeto que está vinculado a una clase en particular en lugar de a una instancia de esa clase. Puedes pensar en él como una especie de "objeto singleton" asociado a una clase específica. Los companion objects se utilizan para agrupar funciones y propiedades que deben ser compartidas entre todas las instancias de una clase, similar a cómo se utilizarían miembros estáticos en Java o en otros lenguajes de programación orientados a objetos.

Algunas características importantes de los companion objects en Kotlin son:

- Acceso a miembros estáticos: Los miembros de un companion object se pueden acceder directamente a través del nombre de la clase, sin necesidad de crear una instancia de la

clase.

- Inicialización perezosa: Los miembros del companion object se inicializan de forma perezosa, es decir, la primera vez que se accede a ellos.
- Pueden tener propiedades y funciones: Los companion objects pueden contener propiedades y funciones, lo que los hace muy versátiles.
- Se pueden heredar y sobrescribir: A diferencia de los miembros estáticos en Java, los companion objects pueden heredarse y, si es necesario, sobrescribirse en subclases.

El contenido de la clase será el siguiente:

```
class TareaDatos {
   companion object {
       val TAREAS = arrayListOf(
           Tarea("Correr 30 minutos", "08:00"),
           Tarea("Estudiar móviles", "10:00"),
           Tarea("Comer 4 rebanadas de manzana", "10:30"),
           Tarea("Asistir al taller de programación", "15:45"),
           Tarea("Quedar con Belén", "18:00")
       )
   }
}
```

activity_main.xml

Tras esto ya podremos añadir un nuevo RecyclerView al layout de nuestra actividad principal:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:tools="http://schemas.android.com/tools"
   android:layout_width="match_parent"
   android:layout_height="wrap_content"
   android:orientation="vertical"
   android:padding="16dp"
   tools:context=".MainActivity">

   <TextView
       android:layout_width="match_parent"
       android:layout_height="wrap_content"
       android:gravity="center"
       android:text="Ejemplo RecyclerView"
       android:textStyle="bold"
       android:textSize="25dp"
       android:layout_marginBottom="20dp" />
```

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

</LinearLayout>
```

item_tarea.xml

En cada elemento de la lista queremos mostrar ambos datos, por lo que el siguiente paso será crear un layout XML con la estructura visual que deseemos. En nuestro caso los mostraremos en dos etiquetas de texto (TextView), la primera de ellas en negrita y con un tamaño de letra un poco mayor. Colocaremos este layout en /app/res/layout:

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

    <TextView
        android:id="@+id/lblTitulo"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:textSize="24sp"
        android:textStyle="bold"
        tools:text="Título" />

    <TextView
        android:id="@+id/lblSubtitulo"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:textSize="16sp"
        android:textStyle="normal"
        tools:text="Subtitulo" />
</LinearLayout>
```

Adaptador

A continuación, escribiremos nuestro adaptador. Este adaptador deberá extender a la clase **RecyclerView.Adapter,** de la cual tendremos que sobrescribir principalmente tres métodos:

- **onCreateViewHolder**(). Encargado de crear los nuevos objetos ViewHolder necesarios para

los elementos de la colección.

- **onBindViewHolder**(). Encargado de actualizar los datos de un ViewHolder ya existente.
- **getItemCount**(). Indica el número de elementos de la colección de datos.

Para nuestra aplicación de ejemplo utilizaremos como fuente de datos un simple array de objetos de tipo Tarea, como hemos creado en la clase TareaDatos.

TareaAdapter

El primer paso será crearnos la clase TareaAdapter, ésta recibirá como parámetro la lista con las tareas que queramos mostrar.

```
class TareaAdapter (private val tareas: List<Tarea>)
```

Además, hereda de la clase ViewHolder que a continuación vamos a definir como una clase interna de TareaAdapter.

ViewHolder

Como ya hemos comentado, la clase RecyclerView.Adapter nos obligará en cierta forma a hacer uso del patrón view holder. Por tanto, para poder seguir con la implementación del adaptador lo primero que definiremos será el ViewHolder necesario para nuestro caso de ejemplo.

Lo definiremos como clase interna a nuestro adaptador, extendiendo la clase **RecyclerView.ViewHolder**. Recibirá como parámetro un objeto de tipo view, que será un ítem de la lista y utilizaremos el binding para referenciar los elementos que hayamos definido en el archivo item_tarea (en este ejemplo). Previamente, habremos activado el binding a nivel de gradle.

```
inner class ViewHolder(view:View):RecyclerView.ViewHolder(view){
   val binding = ItemTareaBinding.bind(view) //Vinculamos la vista a nuestro adapter

}
```

Tras ello haremos explícita la herencia de la clase TareaAdapter, incorporando en la definición de la clase el pertinente código. Al hacerlo nos mostrará un error, pues es obligatorio sobreescribir ciertos métodos, haremos Ctrl + I y nos los añadirá a la definición de la clase. El código quedará como se muestra a continuación:

```
class TareaAdapter (private val tareas: List<Tarea>):
RecyclerView.Adapter<TareaAdapter.ViewHolder>(){
```

```
    inner class ViewHolder(view:View):RecyclerView.ViewHolder(view){
        val binding = ItemTareaBinding.bind(view) //Vinculamos la vista a
nuestro adapter

    }

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
ViewHolder {
        TODO("Not yet implemented")
    }

    override fun getItemCount(): Int {
        TODO("Not yet implemented")
    }

    override fun onBindViewHolder(holder: ViewHolder, position: Int) {
        TODO("Not yet implemented")
    }

}
```

Métodos del Adaptador: onCreateViewHolder(), onBindViewHolder() y getItemCount()

Finalizado nuestro ViewHolder podemos ya seguir con la implementación del adaptador sobreescribiendo los métodos indicados.

- El método **onCreateViewHolder**() nos limitaremos a inflar (construir) una vista a partir del layout correspondiente a los elementos de la lista (item_tarea), y crear y devolver un nuevo ViewHolder llamando al constructor de nuestra clase ViewHolder que hemos definido con el inner class.

  Además, definimos una variable global para tener acceso al context de la actividad.
- En **onBindViewHolder**() tan sólo tendremos que recuperar el objeto Tarea correspondiente a la posición recibida como parámetro y asignar sus datos sobre el ViewHolder también recibido como parámetro.
- En **getItemCount**() devolverá el tamaño del array de datos.

```
// El layout manager invoca este método para renderizar cada elemento del RecyclerView
override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder {
```

```
//Inflar la vista item_tarea en el Recycler
   context = parent.context
   val view = LayoutInflater.from(context).inflate(R.layout.item_tarea, parent, false)
   return ViewHolder(view)
}

// Este método devolverá el tamaño de la lista
override fun getItemCount(): Int = tareas.size

override fun onBindViewHolder(holder: ViewHolder, position: Int) { //Asignamos el
contenido a cada item
   val tarea = tareas.get(position)
   with(holder){
       binding.lblTitulo.text = tarea.getNombre()
       binding.lblSubtitulo.text = tarea.getHora()
   }
}
```

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

TareaAdapter.java

Mostramos a continuación como queda el código completo de ésta clase:

```
class TareaAdapter (private val tareas: List<Tarea>):
RecyclerView.Adapter<TareaAdapter.ViewHolder>(){

   private lateinit var context: Context

   inner class ViewHolder(view:View):RecyclerView.ViewHolder(view){
       val binding = ItemTareaBinding.bind(view) //Vinculamos la vista a nuestro
adapter

   }

   // El layout manager invoca este método para renderizar cada elemento del
RecyclerView
   override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder {
//Inflar la vista item_tarea en el Recycler
       context = parent.context
       val view = LayoutInflater.from(context).inflate(R.layout.item_tarea, parent,
false)
       return ViewHolder(view)
   }

   // Este método devolverá el tamaño de la lista
   override fun getItemCount(): Int = tareas.size

   override fun onBindViewHolder(holder: ViewHolder, position: Int) { //Asignamos el
```

```
contenido a cada item
        val tarea = tareas.get(position)
        with(holder){
            binding.lblTitulo.text = tarea.getNombre()
            binding.lblSubtitulo.text = tarea.getHora()
        }
    }

}
```

MainActivity.kt

Ahora ya podríamos asignar el adaptador al RecyclerView eso lo haremos en nuestra actividad principal. Nos crearemos objeto de nuestro adaptador personalizado TareaAdapter pasándole como parámetro la lista de datos (que serán un array de objetos Tarea), y asignarlo al control RecyclerView mediante su propiedad adapter.

```
class MainActivity : AppCompatActivity() {

    private lateinit var tareaAdapter: TareaAdapter
    private lateinit var linearLayoutManager: LinearLayoutManager

    private lateinit var binding: ActivityMainBinding


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

        tareaAdapter = TareaAdapter(TareaDatos.TAREAS)
        linearLayoutManager = LinearLayoutManager(this)

        binding.recyclerId.apply {
            layoutManager = linearLayoutManager
            adapter = tareaAdapter
        }
    }
```

En el método onCreate() será donde tras obtener la referencia al RecyclerView

Adicionalmente, le hemos creado y asignado un **LayoutManager** para determinar la forma en la que se distribuirán los datos en pantalla. Como ya dijimos, si nuestra intención es mostrar los datos en forma de lista o tabla (al estilo de los antiguos ListView o GridView) no tendremos que implementar

nuestro propio LayoutManager (una tarea nada sencilla), ya que el SDK proporciona varias clases predefinidas para los tipos más habituales. En nuestro caso particular queremos mostrar los datos en forma de lista con desplazamiento vertical. Para ello tenemos disponible la clase LinearLayoutManager, por lo que tan sólo tendremos que instanciar un objeto de dicha clase indicando en el constructor la orientación (LinearLayoutManager.VERTICAL o LinearLayoutManager.HORIZONTAL) y lo asignamos al RecyclerView mediante su propiedad layoutManager.

Llegados aquí ya sería posible ejecutar la aplicación de ejemplo para ver cómo quedan nuestros datos en pantalla, ya que los dos componentes que nos falta por comentar (ItemDecoration e ItemAnimator) no son obligatorios para el funcionamiento básico del control RecyclerView.

º5.4.1 Manejar eventos al seleccionar un ítem de la lista

El siguiente paso que nos podemos plantear es cómo responder a los eventos que se produzcan sobre el RecyclerView, como opción más habitual el **evento click** sobre un elemento de la lista. Para sorpresa de muchos la clase RecyclerView no incluye un evento onItemClick como ocurría en el caso de ListView. Una vez más, RecyclerView delega también esta tarea a otro componente, en este caso a la propia vista que conforma cada elemento de la colección.

Será por tanto dicha vista a la que tendremos que asociar el código a ejecutar como respuesta al evento click. Esto podemos hacerlo de diferentes formas, mostraremos una de ellas.

Necesitamos poder establecer una comunicación entre el adaptador y la main activity para así poder pasar el evento.

1. Crearemos una interfaz con nombre: OnClickListener y dentro definimos el método onClick que pasará el objeto Tarea sobre el cual se habrá hecho click.

```
interface OnClickListener {
   fun onClick(tarea: Tarea)
}
```

TareaAdapter

2. Modificamos el adaptador para agregarle esa propiedad, así que en el constructor le añadimos esta propiedad:

```
class TareaAdapter (private val tareas: List<Tarea>, private val
listener: OnClickListener):
RecyclerView.Adapter<TareaAdapter.ViewHolder>(){
...
```

3. En la clase interna ViewHolder, añadimos un nuevo método que recibe como parámetro la tarea, que es lo que posteriormente le enviaremos a la actividad.

```
inner class ViewHolder(view:View):RecyclerView.ViewHolder(view){
   val binding = ItemTareaBinding.bind(view) //Vinculamos la vista a
nuestro adapter

   fun setListener(tarea:Tarea){
       binding.root.setOnClickListener {listener.onClick(tarea)}
```

```
    }

}
```

4. Vinculamos cada elemento con el listener que terminamos de implementar, para ello modificaremos el método onBindViewHolder.

```
override fun onBindViewHolder(holder: ViewHolder, position: Int) {
//Asignamos el contenido a cada item
    val tarea = tareas.get(position)
    with(holder){
        setListener(tarea)
        binding.lblTitulo.text = tarea.getNombre()
        binding.lblSubtitulo.text = tarea.getHora()
    }
}
```

Con esto tenemos nuestro adaptador complementamente vinculado ahora nos falta adaptar nuestra actividad.

MainActivity

5. Antes que nada tenemos que hacer que implemente la interfaz, por ello modificamos la definición de la clase:

```
class MainActivity : AppCompatActivity(), OnClickListener{
…
```

6. Solucionamos el error haciendo que sobreescriba los métodos de la interfaz haciendo ctrl + i. Esto nos obligará a sobreescribir el método onClick, mostraremos un mensaje. Y también solucionamos el problema pasándole el segundo parámetro al constructor del TareaAdapter.

```
…
tareaAdapter = TareaAdapter(TareaDatos.TAREAS, this)
…
```

```
override fun onClick(tarea: Tarea) {
    Toast.makeText(this, tarea.getNombre(), Toast.LENGTH_SHORT).show()
}
```

5.4.2 Añadir decoración a la lista

Los **ItemDecoration** nos servirán para personalizar el aspecto de un RecyclerView más allá de la distribución de los elementos. El ejemplo típico de esto son los separadores o divisores de una lista.

Para el caso de los separadores de lista Android ya nos proporciona de serie la clase **DividerItemDecoration**. Para utilizar este componente deberemos simplemente crear el objeto y asociarlo a nuestro RecyclerView mediante addItemDecoration() en nuestra actividad principal:

Creamos la variable:

```
private lateinit var itemDecoration: DividerItemDecoration
...
```

Definimos el estilo:

```
itemDecoration = DividerItemDecoration(this,
DividerItemDecoration.VERTICAL)
```

Lo asignamos al recycler:

```
binding.recyclerId.apply {
    layoutManager = linearLayoutManager
    adapter = tareaAdapter
    addItemDecoration(itemDecoration)
}
```

Como nota adicional indicar que podremos añadir a un RecyclerView tantos ItemDecoration cómo queramos, que se aplicarán en el mismo orden que se hayan añadido con addItemDecoration().