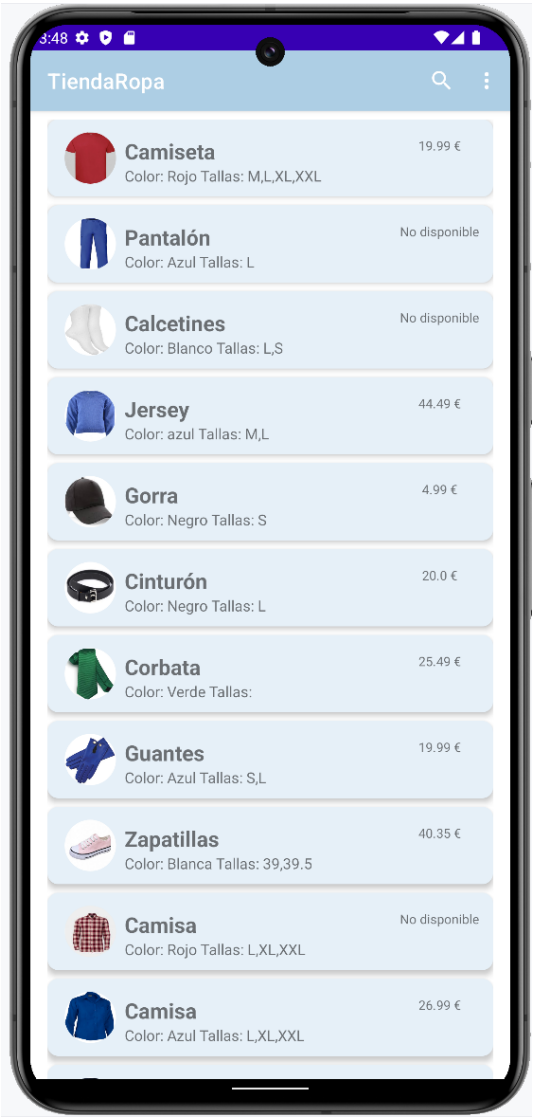
PROYECTO 5

TIENDA DE ROPA



En este proyecto haremos una app que mostrará una lista con los productos de una tienda de ropa, y también tendrá un buscador que servirá para filtrar los productos. Los datos de la tienda estarán en un archivo JSON incluido en la carpeta de recursos de la app.

Durante su desarrollo se tratarán estos conceptos:

- JSON
- Moshi
- Diseño de menús
- Menús en la barra de tareas
- RecyclerView
- RecyclerView.ViewHolder
- RecyclerView.Adapter
- CardView
- Configuración del tema
- ShapeableImageView
- SearchView

1 – JSON

El formato **JSON** es un formato de almacenamiento de información muy importante en la actualidad, que contiene los siguientes elementos:

- Objetos: Son parejas clave-valor encerradas entre llaves **{ }**.
- Cada pareja clave-valor tiene formato **clave:valor** y se separa de la siguiente mediante una coma **,**
- Las claves van encerradas entre comillas **“”**
- Los valores admitidos son: enteros, double, boolean, String y null
- Listas: Son una lista de datos (se admiten objetos y valores constantes) encerradas entre corchetes **[]**

- Abre Android Studio y crea un proyecto llamado **TiendaRopa,** usando la plantilla **Empty Views Activity**
- Copia los archivos con las imágenes de los productos que se venden en la tienda a la carpeta **res/drawable**
- Pulsa el botón derecho del ratón sobre la carpeta **res** y elige **new → directory**
- En la ventana que se abre llama al nuevo directorio **raw**
- Copia el archivo **datos.json** en la carpeta **res/raw**

El directorio **raw** de la carpeta de recursos es un directorio donde se ponen archivos con contenido arbitrario (pueden ser archivos de música, archivos json, archivos creados por nosotros…), que incluimos en el proyecto

- Abre el archivo **build.gradle.kts (Project)** y agrega el plugin **kapt**

```
plugins {
    alias(libs.plugins.android.application) apply false
    alias(libs.plugins.kotlin.android) apply false
    kotlin("kapt") version "2.0.21"
}
```

El plugin **kapt** es una herramienta para que **gradle** sepa leer el código fuente y generar, a partir de él, nuevo código fuente evitando a nosotros tener que escribir código que puede ser muy pesado.

- A continuación, abre el archivo **build.gradle.kts (Module)** y agrega las dependencias para el plugin **kapt** y para la librería **Moshi**

```
plugins {
     id("org.jetbrains.kotlin.kapt")
     // resto de la sección plugins omitido
}
// resto del archivo build.gradle.kts omitido
dependencies {
    implementation("com.squareup.moshi:moshi:1.15.1")
    kapt("com.squareup.moshi:moshi-kotlin-codegen:1.15.1")
    // resto de la sección dependencies omitido
}
```

2 – Moshi

Usaremos la librería **Moshi,** para convertir en objetos de Kotlin los contenidos de un archivo JSON. Tenemos que crear una clase[1] de Kotlin para cada objeto que veamos en el archivo JSON, y poner anotaciones para hacer corresponder (mapear) los datos del archivo JSON con las variables de instancia de la clase.

- Abre el archivo **datos.json** y comprueba que su estructura es la siguiente:

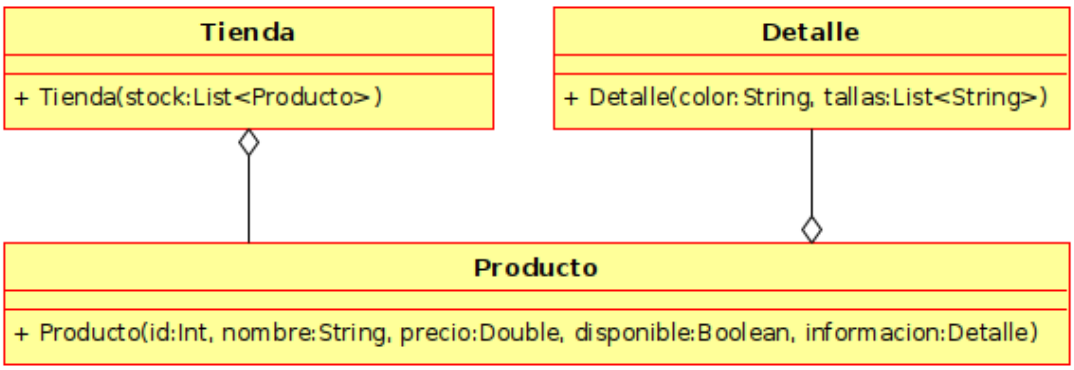
El archivo contiene un objeto que posee una clave llamada "productos", cuyo valor es una lista

```
{
  "productos": [
    {
      "id":1,
      "nombre" : "Camiseta",
      "precio": 19.99,
      "disponible": true,
      "detalles": {
        "color": "Rojo",
        "tallas": ["M","L","XL","XXL"]
      }
    },
    {
      "id":2,
      "nombre": "Pantalón",
      "precio": 29.99,
      "disponible": false,
      "detalles": {
        "color": "Azul",
        "tallas": ["L"]
      }
    },
}
```

Cada producto de la lista es un objeto que tiene como claves "id","nombre", "precio","disponible" y "detalles"

La clave "detalles" tiene como valor un objeto con claves "color" y "tallas"

- Observa que en el archivo **datos.json** podemos identificar estas tres clases:
  - Una es la que representa al objeto global, cuya clave es **productos**. Llamaremos **Tienda** a esta clase
  - Otra es la clase que representa un producto individual, y que tiene como variables de instancia "id","nombre","precio","disponible" y "detalles". Llamaremos **Producto** a esta clase
  - La última es la que representa los detalles de un producto, y que tiene las variables de instancia **color** y **tallas**. Llamaremos **Detalle** a esta clase



[1] En lugar de una clase convencional se suelen usar **data classes**, que son como los **records** de Java

- Crea en el proyecto un paquete llamado **modelo** y agrega a él una **data class** llamada **Detalle**, con este código fuente:

```
data class Detalle(
    val color:String,
    val tallas:List<String>
)
```

Una **data class** es una clase cuyo único objetivo es portar datos. En la definición de la clase aparece entre paréntesis la lista con dichos datos, y dicha lista es además el formato del constructor de la clase.

- A continuación, añade la anotación **@JsonClass** a la clase, de esta forma:

```
@JsonClass(generateAdapter = true)
data class Detalle(
    val color:String,
    val tallas:List<String>
)
```

La anotación **@JsonClass** sirve para indicar que esa clase se corresponde con un objeto de un archivo JSON.

La propiedad **generateAdapter** con valor **true** nos indica que se usará el plugin **kapt** para que **gradle** nos genere cierto código fuente asociado a la clase de forma automática.

- A continuación, escribimos y anotamos de igual forma la clase **Producto**

```
@JsonClass(generateAdapter = true)
data class Producto(
    val id:Int,
    val nombre:String,
    val precio:Double,
    val disponible:Boolean,
    val informacion:Detalle
)
```

- Como en la clase **Producto**, (de forma intencionada por motivos pedagógicos) hemos puesto que el objeto **Detalle** se llame **información,** y no coincide con el nombre que tiene en el archivo **datos.json**, que es **detalles**, debemos usar la etiqueta **@Json** en dicha variable de instancia para establecer la asociación:

```
@JsonClass(generateAdapter = true)
data class Producto(
    val id:Int,
    val nombre:String,
    val precio:Double,
    val disponible:Boolean,
    @Json(name="detalles") val informacion:Detalle
)
```

La anotación **Json** nos permite poner a una variable de instancia de la clase un nombre diferente al que tiene en el archivo **datos.json**

- Por último, programamos de forma similar la clase **Tienda**

```
@JsonClass(generateAdapter = true)
data class Tienda(
    @Json(name="productos") val stock:List<Producto>
)
```

- A continuación, en el mismo archivo de la clase **Tienda** vamos a añadir una función llamada **cargarJson** que cargue el archivo **datos.json**. Esta función necesitará como parámetro un objeto de tipo **context**, que nos da acceso a la carpeta **res** de la app

```
fun cargarJson(context: Context) =
    context             // objeto que representa la app
        .resources      // objeto que representa la carpeta res
        .openRawResource(R.raw.datos)   // obtiene un InputStream para datos.json
        .bufferedReader()   // obtiene un BufferedReader para el InputStream
        .readText()     // lee en un String todo el contenido del BufferedReader
```

El objeto **context** representa la app y posee un objeto llamado **context**, que representa la carpeta de recursos. Con su método **openRawResource** obtenemos un **InputStream** que da acceso al archivo cuyo id pasamos como parámetro.

- Siguiendo en el mismo archivo, vamos a añadir una función llamada **crearTienda**, que recibe un String con el contenido del archivo **datos.json** y utiliza la librería **Moshi** para obtener y devolver un objeto **Tienda** a partir de él.

```
fun crearTienda(json:String):Tienda{
    val moshi = Moshi.Builder().build() // obtenemos la librería Moshi
    val adapter = moshi.adapter(Tienda::class.java) // obtenemos un conversor de json a objetos
    val tienda:Tienda = checkNotNull(adapter.fromJson(json)) // convierte el String en un objeto
    return tienda
}
```

La librería **Moshi** permite obtener un **adapter,** que es un objeto que sabe convertir String en formato json a los objetos de Kotlin que hemos programado en el modelo.

La función **checkNotNull** recibe un objeto de un tipo nullable[2] y en caso de que no sea null, nos devuelve la versión no nullable de dicho objeto. Si es null, lanza una excepción.

- Por último, vamos a añadir a la clase **Tienda** un **companion object** que va a tener una función llamada **inicializar**, que recibirá un **context** y usará las dos funciones anteriores para devolver un objeto **Tienda.**

```
@JsonClass(generateAdapter = true)
data class Tienda(@Json(name="productos") val stock:List<Producto>){
    companion object{
        fun inicializar(context: Context):Tienda =
            crearTienda(cargarJson(context))
    }
}
```

[2] Los tipos **nullable** son los tipos cuyo nombre lleva un interrogante y admiten null. Ejemplo: **Int?**

En Kotlin no existen los métodos estáticos como tales, y la forma de conseguir un método asociado a una clase consiste en añadirle un **companion object**, que es un objeto que se adjunta a la clase y que tiene métodos.

Los métodos del **companion object** se usan igual que los métodos estáticos de Java

- En **MainActivity** vamos a añadir un método llamado **testJson** que servirá para comprobar si se carga bien el archivo **datos.json**

```
1.  class MainActivity : AppCompatActivity() {
2.      override fun onCreate(savedInstanceState: Bundle?) {
3.          super.onCreate(savedInstanceState)
4.          setContentView(R.layout.activity_main)
5.          testJson()
6.      }
7.      private fun testJson(){
8.          // cargamos aquí el archivo datos.json
9.      }
10. }
```

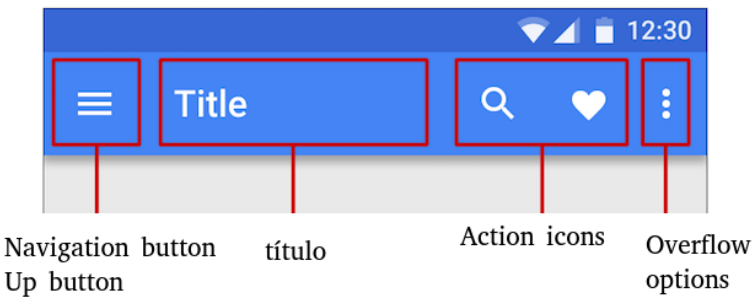
- Para cargar el archivo **datos.json** llamamos al método **inicializar** que está en el **companion object** de la clase **Tienda.** La llamada se hace igual que si fuese un método estático de Java:

```
1.  private fun testJson(){
2.      val tienda = Tienda.inicializar(this)
3.      Log.d("testJson","Cargados: ${tienda.stock}")
4.  }
```

- Ejecuta la app y comprueba que en **Logcat** se muestra la lista de productos que había en **datos.json**
- Una vez que compruebes que todo funciona bien, puedes borrar el método **testJson** y la línea de **onCreate** en la que se llama a dicho método.

3 – Diseño de menús

La barra de tareas de una app puede contener un menú con opciones que se muestran visibles en forma de iconos (**action icons)** o como texto que si no cabe, se muestra en el menú de desbordamiento (**overflow options**)



Es posible diseñar el menú de forma gráfica, o a mano, editando archivos xml

- Pulsa el botón derecho del ratón sobre la carpeta **res** y elige **new → android resource file**
- En la ventana que aparece, rellena estos datos:
    - Nombre de archivo: **menu_toolbar**
    - Tipo de recurso: **Menu**
- Observa que se abre el diseñador gráfico, y que pulsando los botones habituales podemos cambiar entre las vistas de diseñador y código fuente.
- Usando la vista de diseñador gráfico, arrastra un **Search Item (**su verdadero nombre es **SearchView**) a la barra de tareas y observa que aparece el **menú de desbordamiento** (tiene tres puntos verticales) con la opción **search**
- Selecciona la opción **search** y en la zona de atributos, configura estas opciones:
    - id: **txtBuscador**
    - title: **Buscar**
    - showAsAction: **always**

El atributo **showAsAction** nos indica si la opción deberá verse siempre en la barra de tareas, o si deberá formar parte del menú de desbordamiento.

- Añade ahora un **MenuItem** al menú, selecciónalo y rellena estos atributos:
    - id: **btnMostrarTodos**
    - title: **Mostrar todos**
    - showAsAction: **never**

4 – Menús en la barra de tareas

La colocación de un menú en una barra de tareas y la programación de sus opciones es algo que hay que hacer en código fuente.

El enfoque más reciente para ello consiste en poner a **MainActivity** un objeto **MenuProvider**

- Crea un paquete llamado **vista** y mueve el archivo **MainActivity** a él
- En dicho paquete, pulsa el botón derecho del ratón y elige **new → Fragment → Fragment (Blank)**
- En la pantalla que aparece rellena estos datos:
    - Nombre del fragment: **TiendaFragment**
    - Archivo de layout del fragment: **fragment_tienda**
- Se abrirá una ventana con el código fuente del archivo **TiendaFragment** y borra todo el código fuente obsoleto.
- Habilita el **view binding** y añade a **TiendaFragment** una variable de instancia de tipo **FragmentTiendaBinding**, que inicializarás en un método privado **inicializarBinding**

```
class TiendaFragment : Fragment() {
    private lateinit var binding:FragmentTiendaBinding
    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        inicializarBinding(inflater,container)
        return binding.root
    }
    private fun inicializarBinding(inflater: LayoutInflater, container: ViewGroup?){
        binding=FragmentTiendaBinding.inflate(inflater,container,false)
    }
}
```

- Abre **fragment_tienda.xml** con la vista de código fuente y bórralo todo
- Haz que el elemento principal sea un **LinearLayout** vertical que tenga una **MaterialToolbar** con estos atributos:
    - id: **material_toolbar**
    - ancho: El de su contenedor
    - alto: Se cogerá de la variable **actionBarSize** del tema
    - style: **Widget.MaterialComponents.Toolbar.Primary**

```
<LinearLayout
    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="match_parent"
    android:orientation="vertical">
    <com.google.android.material.appbar.MaterialToolbar
        android:layout_width="match_parent"
        android:layout_height="?attr/actionBarSize"
        android:id="@+id/material_toolbar"
        style="@style/Widget.MaterialComponents.Toolbar.Primary"/>
</LinearLayout>
```

- Abre el código fuente de **TiendaFragment** y crea un método auxiliar llamado **configurarToolbar**, que será llamado dentro de **onCreate**. Dentro de ese método, haremos que nuestra **MaterialToolbar** sea la barra de tareas de la app

```
class TiendaFragment : Fragment() {
    private lateinit var binding:FragmentTiendaBinding
    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        inicializarBinding(inflater,container)
        inicializarToolbar()
        return binding.root
    }
    private fun inicializarToolbar(){
        val a = activity as MainActivity
        a.setSupportActionBar(binding.materialToolbar)
    }
}
```

- Dentro de **inicializarToolbar** vamos a añadirle el menú que hemos creado en el archivo **menu_toolbar.xml**. Esto se hace llamando al método **addMenuProvider,** que requiere un objeto que implemente **MenuProvider**

```
private fun inicializarToolbar(){
    val toolbar:MaterialToolbar = findViewById(R.id.material_toolbar)
    a.setSupportActionBar(toolbar)
    a.addMenuProvider(object:MenuProvider{
        override fun onCreateMenu(menu: Menu, menuInflater: MenuInflater) {
            // este método debe crear el menú
        }
        override fun onMenuItemSelected(menuItem: MenuItem): Boolean {
            // este método decide el código que se ejecuta en cada opción del menú
        }
    })
}
```

La interfaz **MenuProvider** proporciona a **MainActivity** un menú, y tiene dos métodos:

- **onCreateMenu:** Su misión es usar **menuInflater** para crear el menú del archivo xml que hemos diseñado, y añadirlo al parámetro **menú**
- **onMenuItemSelected:** Su misión es usar un bloque **when** para detectar el id del **menuItem** que ha sido pulsado, y en base a eso, decidir el método que se llama.

En Kotlin, esta interfaz se suele implementar no creando una clase, sino creando un objeto[3] que se pasa al método **addMenuProvider** de **MainActivity**

- Programa **onCreateMenu** de forma que llame al método **inflate** del objeto **menuInflater**, pasando de parámetro el id del diseño de nuestro menú.

```
addMenuProvider(object:MenuProvider{
    override fun onCreateMenu(menu: Menu, menuInflater: MenuInflater) {
        menuInflater.inflate(R.menu.menu_toolbar,menu)
    }
}
```

- Programa el método **onMenuItemSelected** para que, de momento devuelva **true**.

```
override fun onMenuItemSelected(menuItem: MenuItem): Boolean {
    return true
}
```

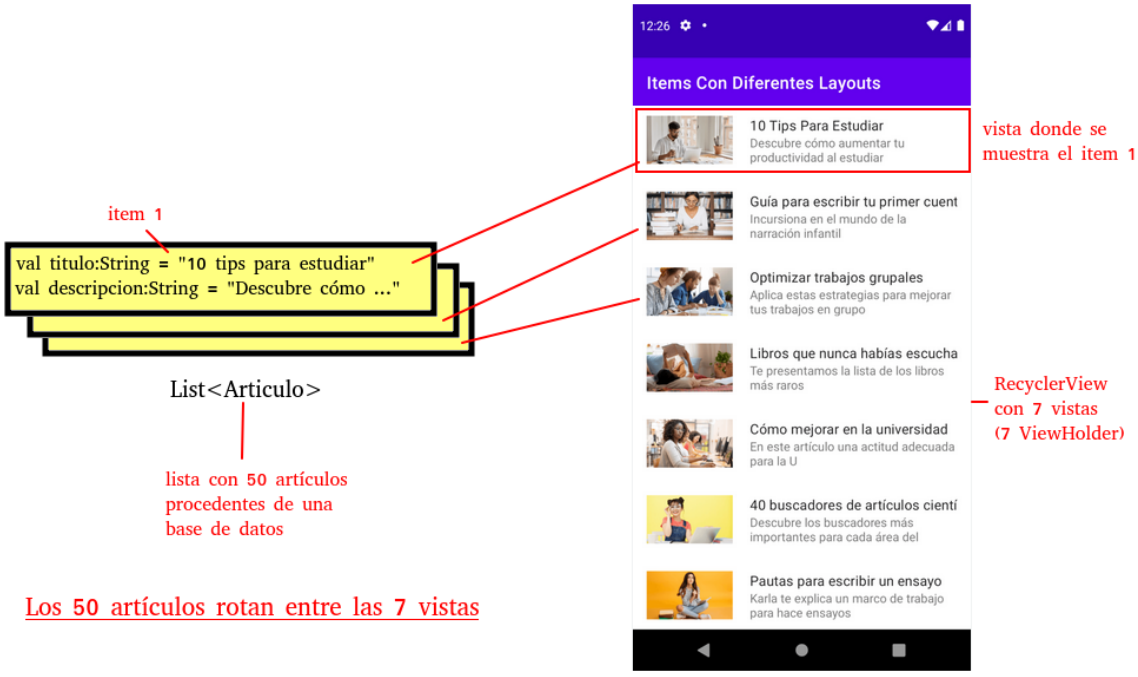
- Abre **activity_main.xml** con la vista de código fuente, borra todo lo que hay y pon como elemento raíz un **FragmentContainerView** que muestre a **TiendaFragment**

```
<androidx.fragment.app.FragmentContainerView
    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"
    android:id="@+id/fragment_container_view"
    android:name="dam.moviles.tiendaropa.controladores.TiendaFragment"/>
```

[3] En Kotlin es posible pasar a un método un objeto que implemente una interfaz utilizando la sintaxis **object:** ***Interfaz*** y sobreescribiendo los métodos que hay en la interfaz. Esto nos evita tener que hacer una clase completa que luego solo se usa en un único punto del programa.

5 – RecyclerView

Un **RecyclerView** es un elemento que muestra una serie de vistas que se corresponden con los ítems de una lista. O sea, cada ítem de la lista, se muestra en una vista del **RecyclerView**.

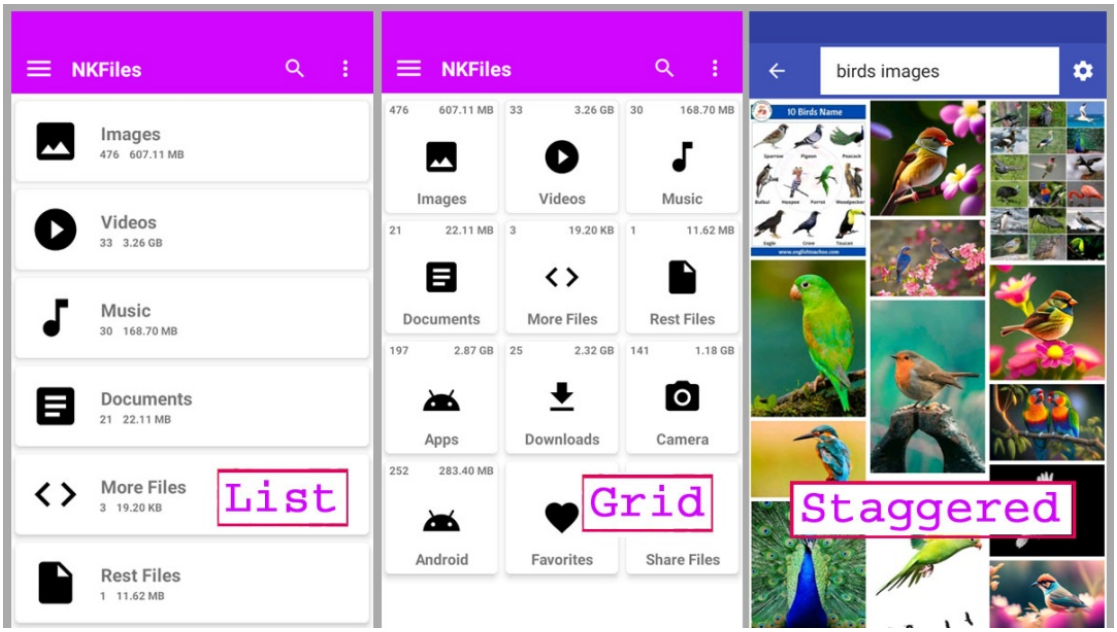


La novedad es que las vistas se reutilizan, de forma que solo los ítems que se ven en la pantalla están en una vista. Cuando un ítem sale de la pantalla, su vista es reutilizada para el nuevo ítem que entra en la pantalla. O sea, el conjunto de vistas está fijo, y los distintos ítems se van colocando en las vistas conforme aparecen en la pantalla. Así, se ahorra memoria mediante la reutilización de las vistas.

- Abre el archivo **fragment_tienda.xml**
- Añade a continuación de la **MaterialToolbar** un **RecyclerView** con estas características:
    - id: **lstResultados**
    - ancho y alto: El de su contenedor
    - layoutManager: **androidx.recyclerview.widget.LinearLayoutManager**

```
<androidx.recyclerview.widget.RecyclerView
    android:id="@+id/lstResultados"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:layoutManager="androidx.recyclerview.widget.LinearLayoutManager"/>
```

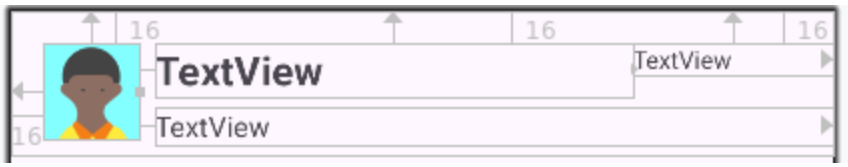
El **RecyclerView** necesita un **LayoutManager** que le indica cómo deberán disponerse los elementos de la lista. Admite varias posibilidades:



- Pulsa el botón derecho del ratón sobre la carpeta **res/layout** y elige **new** → **layout resource file**
- En la ventana que aparece, escribe **item_producto** y observa que se crea el archivo **item_producto.xml,** que se abre con el diseñador

El **RecyclerView** muestra vistas en su interior, así que es necesario hacer un diseño de cómo será la vista que muestre un producto de la tienda.

- Diseña la siguiente interfaz, que será usada para mostrar la descripción de cada producto en el **RecyclerView**



  - **txtNombre:** Aquí se verá el nombre del producto. El tamaño del texto será de 20sp y aparecerá en negrita
  - **txtInfo:** Aquí se verá el color del producto y la lista con las tallas. El tamaño del texto será 14sp
  - **txtPrecio:** Aquí se verá el precio del artículo, si está disponible. Si no lo está, se verá "no disponible". Tendrá una anchura de 100dp y el tamaño del texto será de 12sp

- **imgProducto:** Muestra la imagen del producto. Tendrá 48dp de ancho y de alto, escalando la imagen, y estará separada 16dp del borde izquierdo de la pantalla

- Cuando termines el diseño, pon a **72dp** la altura del **ConstraintLayout**

6– RecyclerView.ViewHolder

Un **ViewHolder** es un recipiente donde se coloca la vista de un item de la lista. El **RecyclerView** crea la máxima cantidad de **ViewHolder** que se ve en la pantalla, de manera que cada uno de ellos porta una vista. Cuando el usuario sube o baja en la lista, los productos que aparecen se colocan en uno de los **ViewHolder** que se quedan libres, de forma que siempre se están mostrando los mismos **ViewHolder**, pero con diferentes productos que entran o salen de la pantalla.

Nuestro proyecto necesita una clase de tipo **ViewHolder** que transporte un **item_producto** y sepa rellenar sus campos con la información de un objeto **Producto**

- Crea un paquete llamado **vista**
- En el paquete **vista**, crea una clase llamada **ProductoHolder**, que herede de la clase **RecyclerView.ViewHolder** y que tenga una variable de instancia llamada **producto**, que será el objeto **Producto** que se muestra en el **ProductoHolder**

```
1.  class ProductoHolder(val binding:ItemProductoBinding)
2.        : RecyclerView.ViewHolder(binding.root){
3.      lateinit var producto:Producto
4.  }
```

La clase **RecyclerView.ViewHolder** es una clase abstracta que representa un **ViewHolder** que porta la vista pasada cuyo elemento raíz se pasa como parámetro en su constructor.

- Añade un método a la clase **ProductoHolder** llamado **mostrarProducto**, que recibe un producto, lo guarda en la variable de instancia **producto** y después rellena la vista que hay dentro de **ProductoHolder**

```
1.  fun mostrarProducto(p:Producto){
2.      producto = p
3.      binding.txtNombre.text=p.nombre
4.      binding.txtPrecio.text= if(p.disponible) p.precio.toString() else "No disponible"
5.      binding.txtInfo.text=
6.          "Color: ${p.informacion.color} Tallas: ${p.informacion.tallas.joinToString(",")}"
7.      binding.imgProducto.setImageDrawable(cargarFoto("p_${p.id.toString()}"))
8.  }
```

- Para cargar la imagen del producto, vamos a añadir un método auxiliar (que pondremos privado por ser auxiliar) que nos ayudará a cargar una imagen a partir del nombre que tiene el archivo en la carpeta **raw**. Dicho método necesitará obtener un **context**, porque recordemos que dicho objeto es el que permite acceder a la carpeta **res**

```
class ProductoHolder(val binding: ItemProductoBinding)
 : RecyclerView.ViewHolder(binding.root){
    fun vincularProducto(p:Producto){
        producto = p
        binding.txtNombre.text=p.nombre
        binding.txtPrecio.text= if(p.disponible) p.precio.toString()+"€" else "No disponible"
        binding.txtInfo.text=
           "Color: ${p.informacion.color} Tallas: ${p.informacion.tallas.joinToString(",")}"
        binding.imgProducto.setImageDrawable(cargarFoto("p_${p.id}"))
    }
    private fun cargarFoto(nombre:String): Drawable {
        // obtiene un objeto que representa la app
        val context = binding.root.context
        // obtiene la carpeta de recursos
        val carpetaRes = context.resources
        // obtengo el id que tiene la foto buscada, en la carpeta raw
        val idFoto = carpetaRes.getIdentifier(nombre,"drawable","dam.moviles.tiendaropa")
        // cargamos la imagen
        val imagen=carpetaRes.getDrawable(idFoto,null)
        return imagen
    }
}
```

7 – RecyclerView.Adapter

Un **Adapter** es un objeto que se conecta al **RecyclerView** y que tiene tres funciones:

- Indicar al **RecyclerView** cuál es la lista de objetos que debe mostrar
- Crear los **ViewHolder** que necesita el **RecyclerView**
- De forma opcional, pero recomendada, portar el código fuente que se ejecuta al pulsar un item de la lista. Este código fuente es una expresión lambda.

La clase abstracta **RecyclerView.Adapter<T>** representa un **Adapter** que trabaja con **view holders** de tipo **T**

- En el paquete **vista** crea una clase llamada **ProductoAdapter,** que herede de la clase **RecyclerView.Adapter<ProductoHolder>** y que tenga en su constructor una lista de productos y una lambda que procese un **ProductoHolder**

```
class ProductoAdapter(
    var productos:List<Producto>,
    val lambda: (ProductoHolder)->Unit  // código que se ejecuta al pulsar un ítem de la lista
) : RecyclerView.Adapter<ProductoHolder>(){
}
```

En Kotlin, el tipo de dato **(A) → B** representa un bloque de código (expresión lambda) que recibe un parámetro de tipo **A** y devuelve un objeto de tipo **B**

En nuestro caso, el tipo **(ProductoHolder) → Unit** representa un bloque de código que recibe como parámetro un objeto **ProductoHolder** y no devuelve nada[4].

[4] Realmente, si que devuelve algo. **Unit** es una clase de la cual, solo hay un único objeto, también llamado **Unit**. Kotlin permite hacer programación funcional más avanzada que Java, y eso implica que todos los métodos y funciones siempre deben devolver algo. Cuando el valor devuelto por el método es irrelevante, se devuelve el objeto **Unit**.

- Pasa el ratón por encima del error que aparece y selecciona la opción **implement members** y **Ok** en la ventana que aparece. *Si lo haces bien verás que aparecen tres métodos que hay que programar*

```
class ProductoAdapter(
    var productos:List<Producto>,
    val lambda: (ProductoHolder) -> Unit
) : RecyclerView.Adapter<ProductoHolder>(){
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ProductoHolder {
        TODO("Not yet implemented")
    }
    override fun getItemCount(): Int {
        TODO("Not yet implemented")
    }
    override fun onBindViewHolder(holder: ProductoHolder, position: Int) {
        TODO("Not yet implemented")
    }
}
```

La clase **RecyclerView.Adapter<T>** es abstracta, lo que significa que sus clases hijas (como **ProductoAdapter**) necesitan sobreescribir tres métodos:

- **onCreateViewHolder:** Este método debe crear y devolver un **view holder** de tipo **T** (en nuestro proyecto **T** es el tipo **ProductoHolder**)
- **onBindViewHolder:** Este método hace tres cosas:
  - Recupera de la lista de productos el objeto **Producto** de la posición pasada como segundo parámetro
  - Rellena la vista del **ProductoHolder** pasado como primer parámetro con los datos del objeto **Producto** recuperado anteriormente
  - Hace que al pulsar sobre un **ProductoHolder**, se ejecute la expresión lambda que hay definida en el constructor. Dicha expresión lambda procesa el **ProductoHolder** que ha sido pulsado
- **getItemCount:** Devuelve el número de productos de la lista de productos

- Programa el método **onCreateViewHolder** de manera que cree y devuelva un **ProductoHolder.** Para hacer eso, usaremos la clase **ItemProductoBinding** para obtener un objeto **binding** a partir de la interfaz **item_producto.xml,** y pasarlo al constructor de **ProductoHolder**

```
override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ProductoHolder {
    // obtenemos un Inflater, que sabe crear objetos a partir del xml
    val inflater = LayoutInflater.from(parent.context)
    // obtenemos el objeto binding para item_producto.xml
    val binding = ItemProductoBinding.inflate(inflater,parent,false)
    // devolvemos un ProductoHolder que porta el objeto binding
    return ProductoHolder(binding)
}
```

- Programa el método **onBindViewHolder**, para que consulte de la lista de productos el de la posición indicada en el parámetro **position**, y luego use el método **vincularProducto** del **ProductoHolder** pasado como parámetro para ponerle el producto consultado de la lista. Además, hace que al pulsar el elemento raíz del **ProductoHolder** se ejecute la lambda que pasamos en el constructor de **ProductoAdapter**

```
override fun onBindViewHolder(holder: ProductoHolder, position: Int) {
    // recupero el Producto de la lista
    val producto = productos.get(position)
    // se lo pongo al holder con su método setProducto
    holder.mostrarProducto(producto)
    // código que se ejecuta al pulsar sobre el elemento raíz de la vista del ProductoHolder
    holder.binding.root.setOnClickListener{
         lambda(holder)   // se llama a la lambda, pasando como parámetro el ProductoHolder
    }
}
```

- A continuación, programa el método **getItemCount** para que devuelva el número de productos de la lista

```
override fun getItemCount(): Int = productos.size
```

- Por último, añadimos un método en **ProductoAdapter** que sirva para cambiar la lista de productos, y hacer que el **RecyclerView** se redibuje cuando suceda.

```
fun setListaProductos(p:List<Producto>){
    productos=p
    notifyDataSetChanged()  // hace que el RecyclerView se redibuje
}
```

Una vez que la clase **ProductoAdapter** está hecha, lo único que nos falta es ponerle un **ProductoAdapter** al **RecyclerView**

- Crea un paquete llamado **viewmodel** y dentro programa la clase **TiendaFragmentViewModel**, con una variable de instancia que sea una lista de productos. Esta variable se inicializará en un método auxiliar llamado **cargarProductos**, que necesitará un **context** como parámetro, ya que usa el método **inicializar** de la clase **Tienda**

```
class TiendaFragmentViewModel():ViewModel() {
    var productos:List<Producto> = emptyList()
    fun cargarProductos(context: Context){
        productos=Tienda.inicializar(context).stock
    }
}
```

El método **cargarProductos** se encarga de rellenar la lista de productos con todos los productos del archivo **datos.json.** Llamaremos a este método en **TiendaFragment**, justo tras crear el **view model**

- Abre **TiendaFragment** y añade una variable de instancia que sea el **TiendaFragmentViewModel**. Esta variable se inicializará en un método auxiliar

llamado **inicializarViewModel** dentro de **onCreateView**, y además, se llamará al método **cargarProductos**

```
class TiendaFragment : Fragment() {
    // resto de la clase omitido
    private lateinit var viewModel:TiendaFragmentViewModel
    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        inicializarBinding(inflater,container)
        inicializarViewModel()
        return binding.root
    }
    private fun inicializarViewModel(){
        viewModel=ViewModelProvider(this).get(TiendaFragmentViewModel::class.java)
        viewModel.cargarProductos(requireContext())
    }
}
```

Una vez creado el **view model**, llamamos a su método **cargarProductos** para cargar la lista de productos de la tienda. Sin embargo, si se produce un cambio estructural, este método vuelve a ser llamado y los datos volverían a cargarse. Para evitar esta duplicidad, introducimos una pequeña comprobación en el **view model**

- Modifica el método **cargarProductos**, para que solo se carguen los datos si la lista de productos está vacía[5]:

```
fun cargarProductos(context: Context){
    if(productos.isEmpty()) {
        productos = Tienda.inicializar(context).stock
    }
}
```

- Por último, añade a **TiendaFragment** un método auxiliar llamado **inicializarRecyclerView,** que es llamado dentro de **onCreateView**. En dicho método, le ponemos al **RecyclerView** un **ProductoAdapter** con la lista de productos recuperada de **datos.json** y el código fuente que se ejecuta al pulsar un item de la lista es un **Toast** que muestra el nombre del producto pulsado.

```
private fun inicializarRecyclerView(){
    val adapter=ProductoAdapter(productos){ holder ->
        Toast.makeText(
            requireContext(),
            "Seleccionado ${holder.producto.nombre}",
            Toast.LENGTH_SHORT
        ).show()
    }
    binding.lstResultados.adapter=adapter
}
```

- Ejecuta la app y comprueba que todo funciona correctamente

[5] La solución óptima para no usar este truco consiste en usar un **ViewModelProvider.Factory,** aunque ese enfoque es más complejo de implementar

8 – CardView

Un **CardView** es una vista contenedora que tiene un aspecto de carta. En este proyecto utilizaremos **MaterialCardView**, que es la implementación realizada por la guía **Material Design**.

- Abre el archivo **item_producto.xml** con la vista de código fuente
- Encierra todo el contenido del archivo en una etiqueta **MaterialCardView**

```
<com.google.android.material.card.MaterialCardView
    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_marginBottom="8dp"
    android:layout_width="match_parent"
    android:layout_height="72dp">
    <!-- Aquí viene el ConstraintLayout -->
</com.google.android.material.card.MaterialCardView>
```

- Ejecuta la app y verás que los productos están encerrados en una carta

9 – Configuración del tema

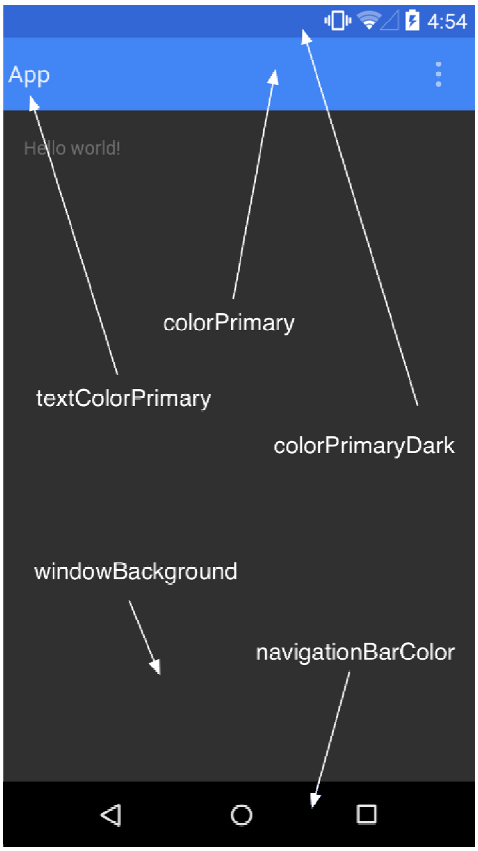
Hasta ahora hemos usado el tema predefinido para nuestra app, pero podemos cambiar el tema a nuestro gusto, en el archivo **theme.xml.** Pese a su nombre, en realidad dicho archivo contiene **estilos**

Un **estilo** es una colección de parejas **clave-valor** que definen características de dibujado. Los estilos se pueden aplicar a cualquier cosa: botones, listas, checkbox, e incluso a la propia app. Se llama **tema** al estilo que utiliza la app.

Las claves de un estilo no pueden tener el nombre que queramos, sino que tienen que coincidir con los nombres que han puesto los desarrolladores de los componentes a los que se quiere aplicar dicho estilo. Por ejemplo, si un componente necesita un parámetro llamado “cornerSize”, no es posible aplicar un estilo que no tenga definida una clave llamada “cornerSize”.

Un estilo puede usar otro estilo como “padre”, de forma que reciba todas las claves-valor definidas en él, y luego modifique las que le interese.

Como el estilo que forma el tema de una app tiene una cantidad grandísima de claves, Android dispone de temas completos predefinidos, de los que podremos cambiar las claves que nos interese, como por ejemplo:



- Abre el archivo **themes.xml**, que está en la carpeta **res/values/themes** y observa que el nombre del tema de nuestra app es **Base.Theme.TiendaRopa**, y como "tema padre" usa el tema **Theme.Material3.DayNight.NoActionBar**
- Cambia el tema base a **Theme.MaterialComponents.Light.DarkActionBar**
- Ejecuta la app y comprueba que aparece la barra de tareas por defecto, con color negro
- Añade dentro de la etiqueta **style** la siguiente etiqueta **item**, para cambiar los colores principal y secundarios de la app:

```
<resources xmlns:tools="http://schemas.android.com/tools">
    <style name="Base.Theme.TiendaRopa" parent="Theme.Material3.DayNight.NoActionBar">
        <item name="colorPrimary">#ADCEE4</item>
        <item name="colorSecondary">#E6F0F8</item>
    </style>
</resources>
```

Cuando añadimos una clave a un tema, estamos cambiando su valor respecto al valor que tiene esa clave en el tema padre

- Ejecuta la app y comprueba que tras cambiar el color principal de la app, cambia el color de la barra de tareas
- Abre el archivo **themes.xml** y añade una etiqueta **style** que definirá las características que tendrán todos los **MaterialCardView.** Llamaremos **estiloCardView** a dicho estilo.

```
<style name="estiloCardView" parent="Widget.MaterialComponents.CardView">
        <item name="cardBackgroundColor">?attr/colorSecondary</item>
        <item name="cardElevation">4dp</item>
        <item name="cardCornerRadius">10dp</item>
</style>
```

Algunos elementos, como los **MaterialCardView** poseen su propio estilo (en los **MaterialCardView** es **Widget.MaterialComponents.CardView**), que es necesario configurar si queremos cambiar su apariencia

- Por último, vamos a indicar que todas las **MaterialCardView** tengan el estilo que hemos llamado **estiloCardView.** Eso lo haremos añadiendo la clave **materialCardViewStyle** al estilo del tema de la app

```
<style name="Base.Theme.Cifrador" parent="Theme.Material3.DayNight.NoActionBar">
    <item name="colorPrimary">#FF0000</item>
    <item name="materialCardViewStyle">@style/resultadoCardView</item>
</style>
```

- Ejecuta la app y comprueba que las **MaterialCardView** tienen ahora un color de fondo.

10 – ShapeableImageView

Los **ImageView** siempre se muestran de forma rectangular. Un **ShapeableImageView** es un **ImageView** cuyo borde puede adoptar distintas formas diferentes, según el estilo que le apliquemos

- Abre el archivo **themes.xml**
- Añade un estilo llamado **circular**, que incluye la clave **cornerSize** para definir el porcentaje de doblado de los bordes (esto es similar al border-radius de css), que será de un **50%**

```
<resources>
    <style name="bordeCircular">
        <item name="cornerSize">50%</item>
    </style>
</resources>
```

- Vete al código fuente de **item_producto.xml** y cambia la etiqueta **ImageView** por un **ShapeableImageView** de esta forma:

```
<com.google.android.material.imageview.ShapeableImageView
        android:id="@+id/imgProducto"
        android:layout_width="48dp"
        android:layout_height="48dp"
        android:layout_marginStart="16dp"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:shapeAppearanceOverlay="@style/bordeCircular"
        tools:srcCompat="@tools:sample/avatars" />
```

El atributo **app:shapeAppearanceOverlay** nos permite indicar el estilo que se aplicará al **ShapeableImageView**.

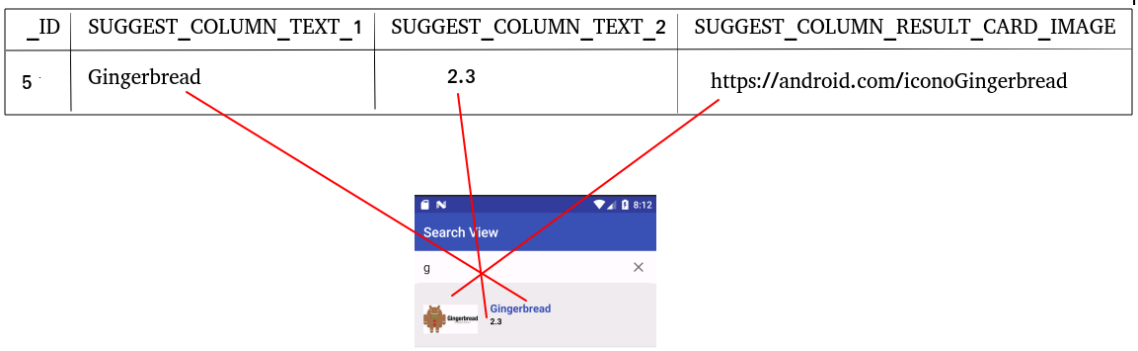
11 – SearchView

Un **SearchView** es un componente muy potente diseñado para hacer buscadores y usarse solo, o integrado en la barra de tareas. El usuario teclea aquello que quiere buscar, y el **SearchView** muestra sugerencias de búsqueda. Su uso es complejo, debido a que está pensado para ser usado con bases de datos, y consta de los siguientes pasos:

- Diseñar un archivo xml para la vista que tendrá una sugerencia
- Hacer un método que calcule las sugerencias que le corresponden a un **String** y rellene con ellas un **Cursor**, que es un objeto similar a una tabla de una base de datos relacional. La clase **SearchManager** define los nombres de columnas posibles del **Cursor**

_ID	SUGGEST_COLUMN_TEXT_1	SUGGEST_COLUMN_TEXT_2	SUGGEST_COLUMN_RESULT_CARD_IMAGE
5	Gingerbread	2.3	https://android.com/iconoGingerbread

- Hacer un método que rellene un objeto **CursorAdapter** que haga corresponder cada columna del **Cursor** con cada elemento de la vista de la sugerencia



- Programar eventos cuando en el **SearchView** se modifique o se envíe el texto escrito, y cuando se seleccione o se pulse sobre una sugerencia.

Para complicar más las cosas, como el **SearchView** es un componente antiguo y está diseñado para ser usado en Java, es muy frecuente el paso de objetos que implementen interfaces en las llamadas a los métodos (así es como se programaba tradicionalmente Android en Java, antes de la aparición de Kotlin)

Hay una versión mucho más sencilla denominada **AutoCompletableEditText**, que gestiona automáticamente las sugerencias.

- Pulsa el botón derecho del ratón sobre la carpeta **res/layout** y elige **new** → **android layout file**

- En la ventana que aparece, escribe como nombre de archivo **sugerencia** y se abrirá el diseñador para dibujar el aspecto que tendrá la vista donde se mostrará una sugerencia de la búsqueda.
- Abre **sugerencia.xml** con la vista de código fuente y deja simplemente un **TextView** como único elemento

```
<TextView
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_height="match_parent"
    android:layout_width="match_parent"
    android:id="@+id/txtSugerencia"/>
```

- A continuación, añade a **TiendaFragment** dos métodos auxiliares que nos permitirán obtener un **Cursor** con las sugerencias, y también obtener elñ **CursorAdapter** que hemos comentado anteriormente

```
class TiendaFragment : Fragment() {
    // resto de la clase omitido
    fun getSugerencias(texto:String): Cursor {
        // este método rellena un Cursor con las sugerencias correspondientes al texto
    }
    fun getCursorAdapter():CursorAdapter{
        // este método rellena un CursorAdapter asociando cada columna del Cursor
        // con un elemento de la vista definida en sugerencia.xml
    }
}
```

- Comienza a programar el método **getSugerencias** creando un **MatrixCursor**, que es un **Cursor** con forma de tabla, y le pondremos las columnas:
    - **BaseColumns._ID** → Representa un número identificador único
    - **SearchManager.SUGGEST_COLUNN_TEXT_1** → Esta columna representa el texto principal de la sugerencia

```
fun getSugerencias(texto:String):Cursor{
    val cursor = MatrixCursor(arrayOf(BaseColumns._ID, SearchManager.SUGGEST_COLUMN_TEXT_1))
}
```

- A continuación, recorremos la lista de productos de la tienda y nos quedamos con aquellos que contienen el texto pasado como parámetro, de manera que no se repitan dos productos iguales. Añadimos al cursor una fila formada por la posición de esos productos y su nombre. *Usaremos programación funcional para hacer esto, porque con bucles el código sería demasiado complicado*

```
fun getSugerencias(texto:String):Cursor{
    val cursor = MatrixCursor(arrayOf(BaseColumns._ID, SearchManager.SUGGEST_COLUMN_TEXT_1))
    viewModel.productos   // partimos de todos los productos
        .map { p -> p.nombre }  // pasamos cada producto a su nombre
        .filter { n -> n.contains(texto, ignoreCase = true) } // aplicamos el filtro
        .distinct() // eliminamos duplicados
        .forEachIndexed { i, n -> cursor.addRow(arrayOf(i,n)) } // añadimos al cursor
    return cursor
}
```

- Programa ahora el método **getCursorAdapter**, que nos devolverá un objeto **SimpleCursorAdapter** rellenado de la forma que se indica en los comentarios del método:

```
fun getCursorAdapter():CursorAdapter =
    SimpleCursorAdapter(
        requireContext(), // contexto de la app
        R.layout.sugerencia,   // archivo xml de la sugerencia
        null,   // aquí es necesario pasar null
        arrayOf(SearchManager.SUGGEST_COLUMN_TEXT_1), // array con las columnas del Cursor
        intArrayOf(R.id.txtSugerencia), // array con las equivalencias de las vistas
        CursorAdapter.FLAG_REGISTER_CONTENT_OBSERVER   // constante para notificar eventos
    )
```

Lo más destacado del constructor de **SimpleCursorAdapter** es que pasamos, dentro de **arrayOf** las columnas del **Cursor** que nos interesa asociar con elementos de la interfaz. Cada posición de dicho array está asociada a la correspondiente posición del array que pasamos con **intArrayOf**, donde se encuentran los elementos de la interfaz dibujada en **sugerencia.xml**

- En el método **inicializarToolbar**, observa que el método **onCreateMenu** recibe un parámetro llamado **menu**, que es el menú de la barra de tareas.

```
override fun onCreateMenu(menu: Menu, menuInflater: MenuInflater) {
    menuInflater.inflate(R.menu.menu_toolbar,menu)
}
```

- Usa el método **findItem** del objeto **menu** para recuperar el **SearchView**.

```
override fun onCreateMenu(menu: Menu, menuInflater: MenuInflater) {
    menuInflater.inflate(R.menu.menu_toolbar,menu)
    val txtBuscador: SearchView = menu.findItem(R.id.txtBuscador).actionView as SearchView
}
```

El método **findItem** del **menu** devuelve un **ItemView**, que es un elemento del menú. Para acceder a su vista en la barra de tareas usamos su propiedad **actionView**, a la que deberemos hacer casting a **SearchView**.

- Para configurar todos los eventos que puedan suceder en **txtBuscador** (escribir algún término de búsqueda, elegir una sugerencia, etc) vamos a hacer un método privado llamado **inicializarBuscador** que se encargue de ello

```
class TiendaFragment : Fragment() {
    // resto de la clase omitido
    private fun inicializarToolbar(){
        val a = activity as MainActivity
        a.setSupportActionBar(binding.materialToolbar)
        a.addMenuProvider(object:MenuProvider{
            override fun onCreateMenu(menu: Menu, menuInflater: MenuInflater) {
                menuInflater.inflate(R.menu.menu_toolbar,menu)
                val txtBuscador: SearchView = menu.findItem(R.id.txtBuscador)
                                                  .actionView as SearchView
                inicializarBuscador(txtBuscador)
            }
            override fun onMenuItemSelected(menuItem: MenuItem): Boolean {
                return true
            }
        })
    }
    private fun inicializarBuscador(buscador:SearchView){
        // asociamos los eventos al buscador
    }
}
```

- En el método **inicializarBuscador**, llama al método **setOnQueryTextListener** del **buscador** pasando como parámetro un objeto que implemente **OnQueryTextListener**, lo que obliga a programar dos métodos:

```
1.  private fun inicializarBuscador(buscador:SearchView){
2.      buscador.setOnQueryTextListener(object: OnQueryTextListener{
3.          override fun onQueryTextSubmit(texto: String?): Boolean {
4.              TODO("Not yet implemented")
5.          }
6.          override fun onQueryTextChange(texto: String?): Boolean {
7.              TODO("Not yet implemented")
8.          }
9.      })
10. }
```

- **onQueryTextSubmit:** Se ejecuta cuando el usuario pulsa el botón de enviar la consulta
- **onQueryTextChange:** Se ejecuta cada vez que el usuario teclea una letra en el buscador, y sirve para proporcionar sugerencias según lo que haya escrito en ese momento

- Programa **onQueryTextSubmit**, para que llame a un método auxiliar llamado **consultar** pasándole el texto escrito en el **SearchView**

```
1.  private fun inicializarBuscador(buscador:SearchView){
2.      buscador.setOnQueryTextListener(object: OnQueryTextListener{
3.          override fun onQueryTextSubmit(texto: String?): Boolean {
4.              consultar(texto)
5.              return true
6.          }
7.          override fun onQueryTextChange(texto: String?): Boolean {
8.              return true
9.          }
10.     })
11. }
12. private fun consultar(texto:String?){
13.     // aquí hacemos que se consulten los productos cuyo texto se pasa como parámetro
14. }
```

- Programa el método **consultar** de forma que si el String recibido es **null**, el **RecyclerView** muestre todos los productos, y si no, muestre aquellos cuyo nombre contiene dicho String. *Para hacerlo, bastará asignar al **adapter** del **RecyclerView** la lista de productos filtrada*

```
1.  private fun consultar(texto:String?){
2.      // aquí hacemos que se consulten los productos cuyo texto se pasa como parámetro
3.      var lista = viewModel.productos
4.      if(texto!=null){
5.          lista = viewModel.productos
6.              .filter { p -> p.nombre.contains(texto, ignoreCase = true) }
7.      }
8.      val adapter = binding.lstResultados.adapter as ProductoAdapter
9.      adapter.setListaProductos(lista)
10. }
```

- Ejecuta la app y comprueba que el buscador funciona correctamente cuando pulsamos el botón de enviar. Comprueba también que tras la búsqueda, el buscador sigue abierto y no se cierra.

- Añade un método privado llamado **cerrarBuscador** que vacía el contenido del **SearchView** y lo "iconiza" (hace que se vuelva a mostrar como icono). Este método es llamado al finalizar el método **setOnQueryTextSubmit**

```
1.  private fun inicializarBuscador(buscador:SearchView){
2.      buscador.setOnQueryTextListener(object: OnQueryTextListener{
3.          override fun onQueryTextSubmit(texto: String?): Boolean {
4.              consultar(texto)
5.              cerrarBuscador(buscador)
6.              return true
7.          }
8.          override fun onQueryTextChange(texto: String?): Boolean {
9.              return true
10.         }
11.     })
12. }
13. private fun cerrarBuscador(buscador:SearchView){
14.     buscador.setQuery("",false)
15.     buscador.isIconified=true
16. }
```

- **<u>Paso opcional avanzado</u>:** *Podemos evitar hacer el método **cerrarBuscador** usando una característica de Kotlin llamada **extension function**, que consiste en "inyectar" a una clase un método. En este caso, podemos inyectar a la clase **SearchView** un método **cerrar()** de esta forma (el siguiente código se escribe en un archivo llamado **Extensiones.kt** en el paquete **vista**)*

```
1.  // añade un método llamado "cerrar" a la clase "SearchView"
2.  fun SearchView.cerrar(){
3.      setQuery("",false)
4.      isIconified=true
5.  }
```

*De esa forma, el método **onQueryTextSubmit** simplemente llamaría a ese método, así:*

```
1.  override fun onQueryTextSubmit(texto: String?): Boolean {
2.      consultar(texto)
3.      buscador.cerrar()
4.      return true
5.  }
```

- Vamos ahora a programar lo que sucede cuando se va escribiendo en el buscador. En ese caso, queremos que salga la lista de sugerencias. Comenzamos asignando a la propiedad **suggestionAdapter** del buscador lo que nos devuelve el método **getCursorAdapter**

```
1.  private fun inicializarBuscador(buscador:SearchView){
2.      buscador.suggestionsAdapter = getCursorAdapter()
3.      buscador.setOnQueryTextListener(object: OnQueryTextListener{
4.          override fun onQueryTextSubmit(texto: String?): Boolean {
5.              consultar(texto)
6.              buscador.cerrar()
7.              return true
8.          }
9.          override fun onQueryTextChange(texto: String?): Boolean {
10.             return true
11.         }
12.     })
13. }
14.
```

- A continuación, programamos **onQueryTextChange** para que nos devuelva un **Cursor** con las sugerencias correspondientes al texto que esté escrito en el buscador (ya teníamos hecho el método **getSugerencias** que nos lo da) y se lo ponemos al **suggestionsAdapter** del buscador

```
override fun onQueryTextChange(texto: String?): Boolean {
    if(texto!=null) {
        val sugerencias:Cursor = getSugerencias(texto)
        buscador.suggestionsAdapter.changeCursor(sugerencias)
    }
    return true
}
```

- Ejecuta la app y comprueba que aparecen las sugerencias esperadas al escribir texto en el buscador, pero que al pulsarlas, no pasa nada.
- Para hacer que al pinchar en una sugerencia suceda algo, hay que programar el evento **setOnSuggestionListener** del **buscador**, al final de **inicializarBuscador.** Dicho método recibe un objeto que implementa **OnSuggestionListener**, lo que obliga a programar dos métodos:
    - **onSuggestionSelect:** Se activa cuando el usuario selecciona una sugerencia. Normalmente no nos interesará hacer nada con él
    - **onSuggestionClick:** Se activa cuando el usuario pulsa una sugerencia. Aquí programaremos lo que ocurre cuando se pulsa la sugerencia.

```
private fun inicializarBuscador(buscador:SearchView){
        buscador.suggestionsAdapter = getCursorAdapter()
        buscador.setOnQueryTextListener(object: OnQueryTextListener{
              // cuerpo del método omitido
        })
        buscador.setOnSuggestionListener(object:OnSuggestionListener{
            override fun onSuggestionSelect(p0: Int): Boolean {
                return true
            }
            override fun onSuggestionClick(p0: Int): Boolean {
                return true
            }
        })
}
```

- Programa el método **onSuggestionClick** para que obtenga el **Cursor** asociado a la sugerencia seleccionada por el usuario, recupere su texto y llame al método **consultar** con dicho texto.

```
// p0 tiene la posición de la sugerencia pulsada
override fun onSuggestionClick(p0: Int): Boolean {
    // obtenemos el cursor correspondiente a la sugerencia pulsada
    val cursor = buscador.suggestionsAdapter.getItem(p0) as Cursor
    // el cursor tiene la forma (id,texto), así que recuperamos la columna 1
    val sugerencia = cursor.getString(1)
    // llamamos al método consultar
    consultar(sugerencia)
    // borramos el buscador
    cerrarBuscador(buscador)
    return true
}

```

- Ejecuta la app y comprueba que el buscador funciona correctamente

Ejercicios

Realiza la app de una agenda que sirva para apuntar tareas pendientes (*como siempre, la app constará de un **Fragment** en **MainActivity** y hará uso de **view binding** y **view model***). Las tareas estarán guardadas en un archivo dentro de la carpeta de la app (ver el ejercicio del bloc de notas) llamado **tareas.json**, que tendrá un formato como el de este ejemplo:

```
{
    "tareas":[
        {
            "nombre":"Hacer los deberes de móviles",
            "fecha_entrega": "24/10/2024",
            "realizado":true
        },
        {
            "nombre":"Estudiar para el examen de móviles",
            "fecha_entrega": "6/11/2024",
            "realizado":false
        },
        {
            "nombre":"Cita con el médico",
            "fecha_entrega": "28/10/2024",
            "realizado":false
        }
    ]
}
```

Deberás crear las clases apropiadas para que **Moshi** pueda recuperar la lista de tareas que hay en el archivo. Dicha lista se cargará al iniciarse la app.

Por otra parte, la interfaz de la app tendrá estos elementos (puedes dibujarlos como prefieras):

- Una **MaterialToolbar** (no es necesario que tenga un **SearchView**)
- Un **EditText** para escribir el nombre de la tarea
- Un **EditText** para escribir la fecha de entrega tope de la tarea
- Un **Button**, que creará una nueva tarea y la añadirá a la lista de tareas
- Un **RecyclerView**, que mostrará todas las tareas de la agenda, o aquellas cuyo nombre está escrito en el buscador. La vista asociada a cada tarea tendrá estos elementos:
    - Un **TextView** con el nombre de la tarea
    - Otro **TextView** con la fecha de entrega de la tarea
    - Un **CheckBox** de solo lectura que estará activado si la tarea está realizada
    - Un **Button** que cuando se pulse, servirá para marcar la tarea como realizada.

Por último, se sobreescribirá el método **onDestroy**, para que se guarde en el archivo **tareas.json** la lista de tareas actualizada.

Pistas:

- *El archivo **tareas.json** no puede estar en la carpeta **raw**, porque todo el contenido de la carpeta **res** es de solo lectura. Deberás crear el archivo **tareas.json** usando el código que vimos en el ejercicio del bloc de notas*

- *La librería **Moshi** sabe guardar el archivo **tareas.json**. Para ello, sigue estos pasos:*

    - *Obtén el objeto **Moshi** y su adapter*
    - *Llama al método **toJson** del adapter, para obtener un **String** que sea el json del objeto que contiene la lista de tareas actualizada*
    - *Usa librerías estándar de Java para guardar dicho **String** en el archivo **tareas.json***

*Recuerda que el archivo **tareas.json** no puede estar en la carpeta **res**, porque es de solo lectura. Hay que crearlo y leerlo desde la carpeta que Android le asigna a la app.*

Opcional: Incluye un **SearchView** en la barra de tareas para buscar tareas y que muestre sugerencias