cpifp Los Enlaces

PROGRAMACIÓN MULTIMEDIA Y DISPOSITIVOS MÓVILES

2º DESARROLLO DE APLICACIONES MULTIPLATAFORMA

TEMA 5. RECYCLERVIEW Y DIALOG

cpifp Los Enlaces

T5. RECYCLERVIEW Y DIALOG

0. RECYCLERVIEW

RecyclerView facilita que se muestren de manera eficiente grandes conjuntos de datos, creando así listas dinámicas.

Cuando un elemento se desplaza fuera de la pantalla, reutiliza la vista para los elementos nuevos que se desplazaron y ahora se muestran en pantalla.

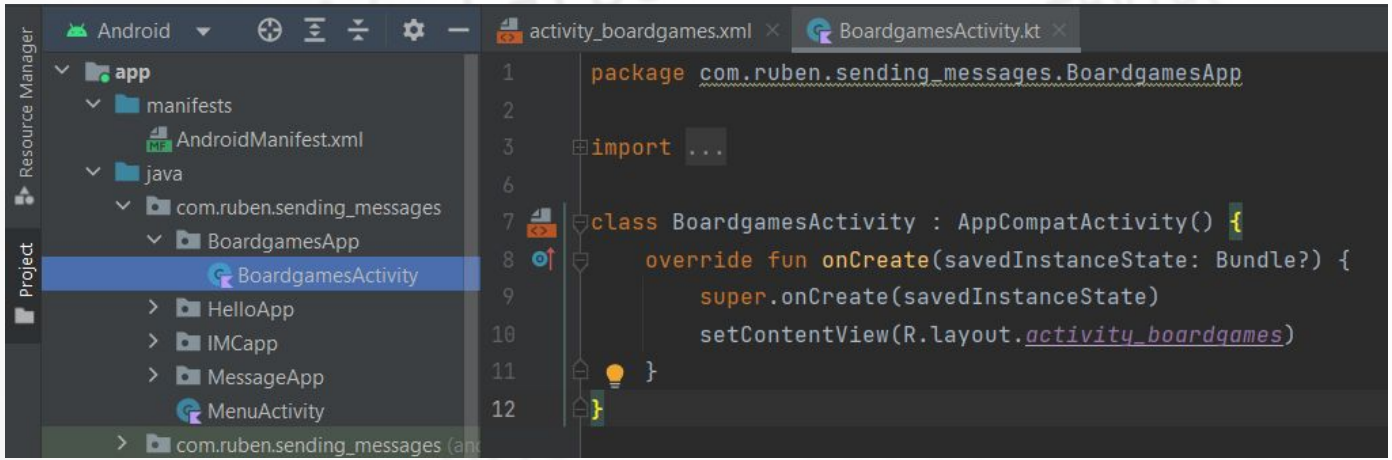
Esto mejora en gran medida el rendimiento y la capacidad de respuesta de tu app y reduce el consumo de energía.

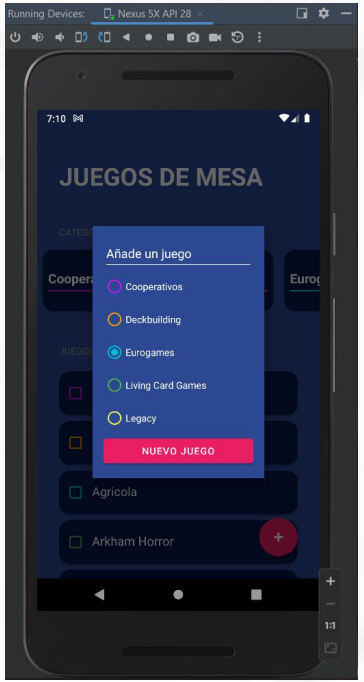
cpi'fp Los Enlaces

T5. RECYCLERVIEW Y DIALOG

1. APLICACIÓN BOARDGAMES

Vamos a crear una aplicación con dos RecyclerViews que organice un tipo de contenido por categorías y nos permita añadir nuevas entradas para dichas categorías.





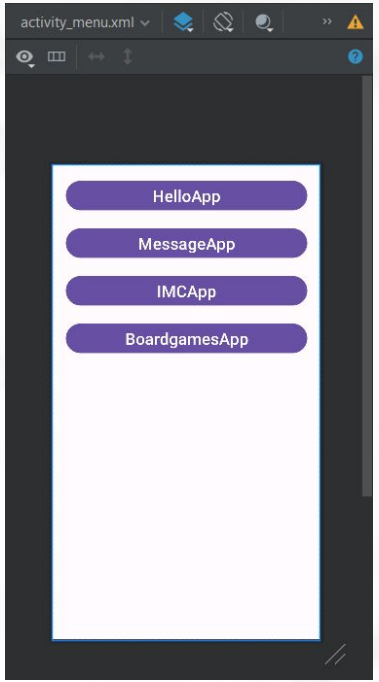
cpifp Los Enlaces

T5. RECYCLERVIEW Y DIALOG

1. APLICACIÓN BOARDGAMES

Lo primero que deberemos hacer es insertar el botón correspondiente en el menú que creamos anteriormente. Lo duplicamos en el layout y le añadimos la lógica.

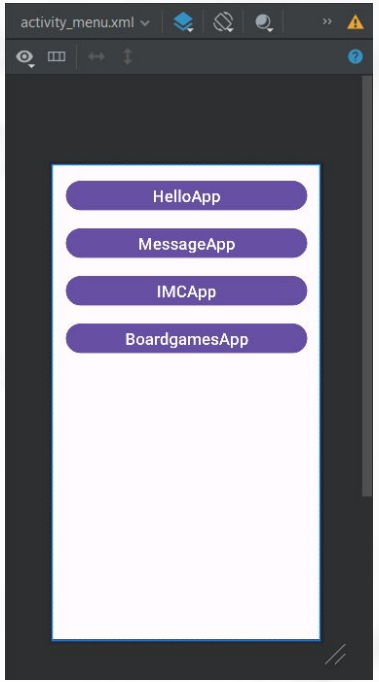
```
var btnBoardgamesApp = findViewById<Button>(R.id.btnBoardgamesApp)

btnBoardgamesApp.setOnClickListener{ navigateToBoardgamesApp() }


private fun navigateToBoardgamesApp(){
    var intent = Intent(this, BoardgamesActivity::class.java)
    startActivity(intent)
}
```



2. ESTILOS

Dentro del directorio res → values → themes podemos encontrar el archivo *themes.xml*. En él se pueden definir una serie de estilos para los layouts de la aplicación.

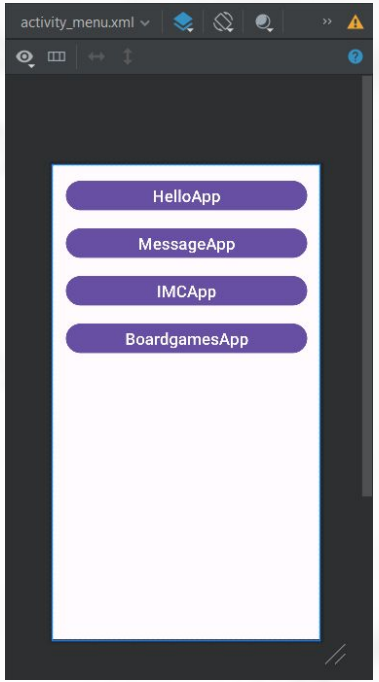
```
<style name="Theme.BoardgamesApp"
parent="Theme.MaterialComponents.DayNight.NoActionBar">
    <item name="colorPrimary">@color/purple_500</item>
    <item name="colorPrimaryVariant">@color/purple_700</item>
    <item name="colorOnPrimary">@color/white</item>
    <item name="colorSecondary">@color/teal_200</item>
    <item name="colorSecondaryVariant">@color/teal_700</item>
    <item name="colorOnSecondary">@color/black</item>
    <item name="android:statusBarColor" >@color/bgapp_background_app </item>
</style>
```



2. ESTILOS

En el archivo *AndroidManifest.xml* podremos insertar este estilo a las actividades que deseemos. En nuestro caso, lo aplicaremos sobre *BoardgamesActivity*.

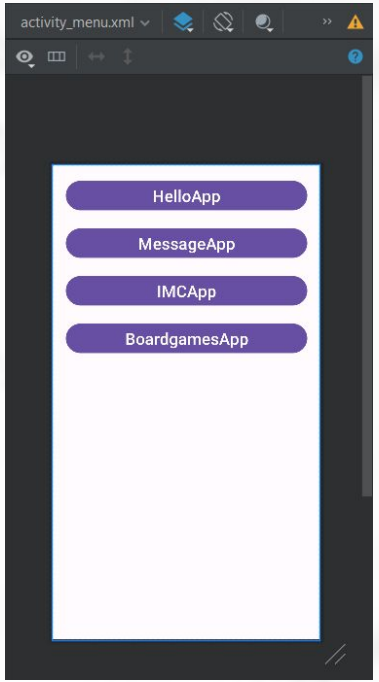
```
<activity
    android:name=".BoardgamesApp.BoardgamesActivity"
    android:theme="@style/Theme.BoardgamesApp"
    android:exported="false" />
```



2. ESTILOS

Además, añadiremos un estilo concreto para algunos elementos de la aplicación. Concretamente para los subtítulos que separarán las secciones.

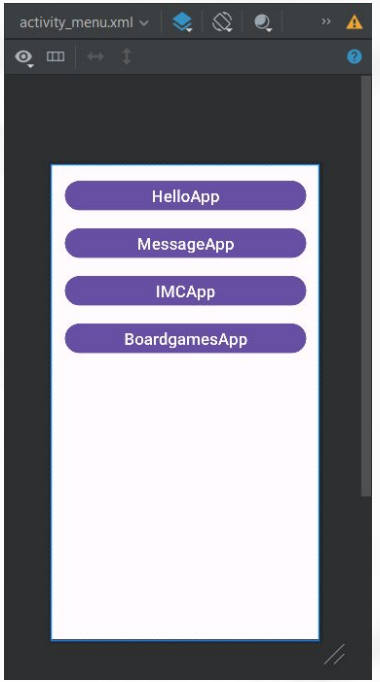
```
<style name="BoardgamesSubtitle">
    <item name="textAllCaps">true</item>
    <item name="android:textColor">@color/bgapp_subtitle_text</item>
    <item name="android:layout_marginHorizontal"
tools:targetApi="o">32dp</item>
</style>
```



2. ESTILOS

También definiremos una paleta de colores para los elementos comunes de la aplicación. Podemos diferenciarlos de los utilizados para otras aplicaciones.

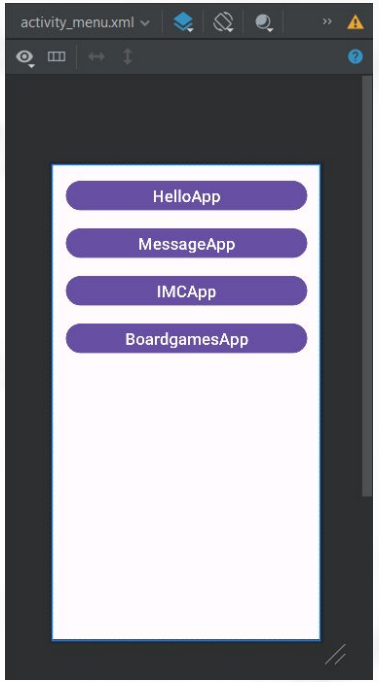
```
<!--BoardgamesApp-->
<color name="purple_200">#FFBB86FC</color>
<color name="purple_500">#FF6200EE</color>
<color name="purple_700">#FF3700B3</color>
<color name="teal_200">#FF03DAC5</color>
<color name="teal_700">#FF018786</color>

<color name="bgapp_background_card">#FF081A4A</color>
<color name="bgapp_background_disabled">#04335C</color>
<color name="bgapp_background_app">#FF2D4993</color>
```



2. ESTILOS

Y algunos colores más para determinados elementos específicos. Todos con el prefijo *bgapp* para que no haya conflictos con otros declarados anteriormente.

```
<color name="bgapp_subtitle_text">#959595</color>
<color name="bgapp_accent">#E91E63</color>

<color name="bgapp_deckbuilding_category">#FF9800</color>
<color name="bgapp_euro_category">#00BCD4</color>
<color name="bgapp_lcg_category">#4CAF50</color>
<color name="bgapp_cooperative_category">#D016F1</color>
<color name="bgapp_legacy_category">#EDF351</color>
```



3. LAYOUT

La distribución de esta aplicación va a ser sencilla, así que vamos a partir de un LinearLayout con orientación vertical a la que le vamos a asignar ya el color de fondo.

```
<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="match_parent"
    android:layout_height="match_parent"
    android:background="@color/bgapp_background_app"
    android:orientation="vertical"
    tools:context=".BoardgamesApp.BoardgamesActivity">
```

3. LAYOUT

Lo primero que vamos a añadir es un título con tamaño de fuente grande, color blanco y el texto “Juegos de mesa”, que almacenaremos en *strings.xml*.

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginHorizontal="32dp"
    android:layout_marginVertical="48dp"
    android:text="@string/boardgames"
    android:textAllCaps="true"
    android:textColor="@color/white"
    android:textSize="36sp"
    android:textStyle="bold" />
```

3. LAYOUT

A la seguida añadiremos un TextView con el estilo que hemos definido para los subtítulos y un RecyclerView que modificaremos posteriormente desde la lógica.

```
<TextView
    style="@style/BoardgamesSubtitle"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/categories" />

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

3. LAYOUT

Como vemos, Android Studio ya nos reserva un espacio para 10 ítems que podremos insertar en el RecyclerView. Colocamos ahora otro subtítulo para la siguiente sección.

```
<TextView
    style="@style/BoardgamesSubtitle"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="32dp"
    android:text="@string/games" />
```

3. LAYOUT

Y un último RecyclerView que tendrá un scroll vertical, a diferencia del primero en el que los elementos se desplazarán horizontalmente.

```
<androidx.recyclerview.widget.RecyclerView
    android:id="@+id/rvGames"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginHorizontal="32dp"
    android:layout_marginTop="16dp" />
```

4. LÓGICA

Empezaremos creando las variables asociadas a los RecyclerViews de nuestro layout y la función initComponents() que los inicialice como objetos.

```
private lateinit var rvCategories: RecyclerView
private lateinit var rvGames: RecyclerView

override fun onCreate(savedInstanceState: Bundle?){...}

private fun initcomponents() {
    rvCategories = findViewById<RecyclerView>(R.id.rvCategories)
    rvGames = findViewById<RecyclerView>(R.id.rvGames)
}
```

4. LÓGICA

Lo siguiente será crear una función initUI() que contendrá la clase Adapter necesaria para que el RecyclerView funcione, la cual necesitará otra clase ViewHolder que determinará el estilo con el que mostrar la información.

```
private lateinit var categoriesAdapter: CategoriesAdapter
private lateinit var gamesAdapter: GamesAdapter

override fun onCreate(savedInstanceState: Bundle?){...}

private fun initUI(){
    categoriesAdapter = CategoriesAdapter(categories)
    gamesAdapter = GamesAdapter(games)
}
```

4. LÓGICA

Previo a la creación de estas dos clases asociadas al RecyclerView debemos definir el tipo de objetos que se van a mostrar en estas listas dinámicas. Para ello, creamos una Kotlin Class llamada *GameCategory.kt*.

```
package com.ruben.aplicacionesPMDM.BoardgamesApp

sealed class GameCategory(var isSelected:Boolean = true) {
    object Deckbuilding : GameCategory()
    object Euro : GameCategory()
    object LCG : GameCategory()
    object Cooperative : GameCategory()
    object Legacy : GameCategory()
}
```

4. LÓGICA

Y otra Kotlin Class llamada *Game.kt* que recibirá 3 parámetros para poder crear objetos de una determinada categoría, si ésta se encuentra seleccionada, y añadirlos a su lista correspondiente.

```
package com.ruben.aplicacionesPMDM.BoardgamesApp

data class Game (val name:String, val category: GameCategory, var isSelected:Boolean
= false)
```

5. DATA CLASS Y SEALED CLASS

Las *data class* permiten definir los datos necesarios en una sola línea de código y proporcionan métodos generados automáticamente para acceder y modificar esos datos.

Pueden ser más **flexibles que las clases selladas** debido a que no tienen restricciones sobre dónde se pueden subclasificar, se pueden definir en un archivo y usarlas en otro sin ningún problema. Esto puede resultar útil para compartir datos entre diferentes partes del código base.

Uno de los principales inconvenientes es que **no proporcionan el mismo nivel de seguridad de tipos** que las clases selladas. Es posible que otro código defina subclases adicionales que entren en conflicto.

5. DATA CLASS Y SEALED CLASS

Una *sealed class* es un tipo especial de clase que sólo puede subclasificarse dentro del mismo archivo que dicha clase sellada. Esto facilita la definición de un conjunto fijo de estados posibles, ya que todas las subclases deben declararse en el mismo archivo.

Un beneficio de las clases selladas es que pueden ayudar a garantizar que el código sea **correcto y fácil de entender**. En una estructura *when*, el compilador de Kotlin se asegurará de cubrir todos los estados posibles.

Uno de los principales inconvenientes es que definir una clase sellada y todas sus subclases puede requerir más código que usar otras opciones, como clases de datos o enumeraciones.

6. RECYCLERVIEW.ADAPTER

Creamos una nueva Kotlin Class llamada *CategoriesAdapter* en el directorio de nuestra aplicación, la cual va a recibir una lista de objetos del tipo *GameCategory* que hemos definido anteriormente.

```
class CategoriesAdapter(private val categories: List<GameCategory>) :
    RecyclerView.Adapter<CategoriesViewHolder>() {
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
CategoriesViewHolder {
    }
    override fun getItemCount(): Int {
    }
    override fun onBindViewHolder(holder: CategoriesViewHolder, position: Int) {
    }
}
```

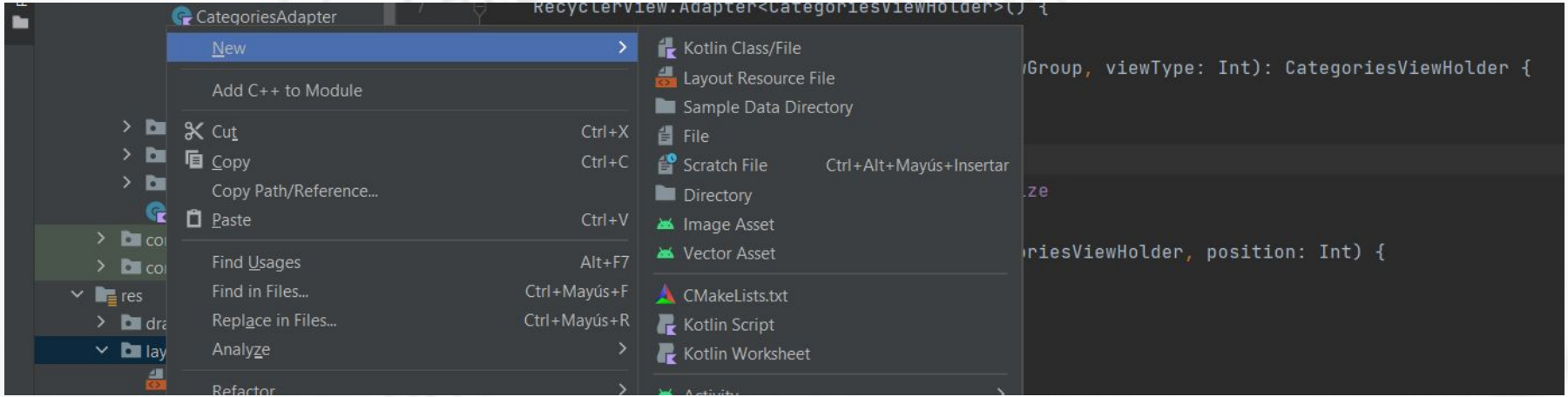
6. RECYCLERVIEW.ADAPTER

Esta clase *Adapter* necesita tres funciones internas que le suministren cierta información. La más sencilla es getItemCount(), que tiene que obtener el tamaño de la lista recibida como parámetro.

```
override fun getItemCount(): Int {
    return categories.size
}

//O simplificando...

override fun getItemCount() =  categories.size
```

6. RECYCLERVIEW.ADAPTER

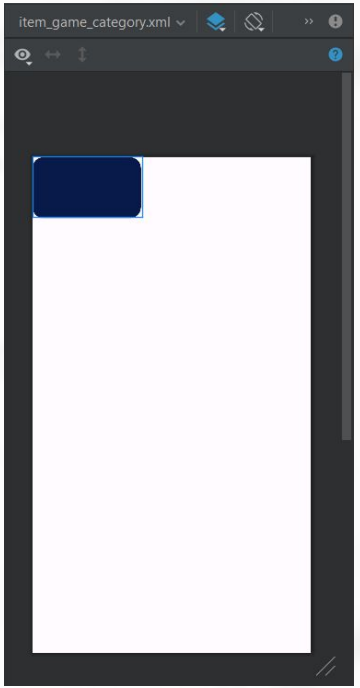
El método onCreateViewHolder() creará la vista que se mostrará cada ítem, pero para ello definiremos un *Layout Resource File* dentro del directorio res → layouts que llamaremos *item_game_category.xml*.



6. RECYCLERVIEW.ADAPTER

Como elemento padre, en vez de un ConstraintLayout, vamos a definir únicamente un CardView con bordes redondeados y un fondo de nuestra paleta de colores.

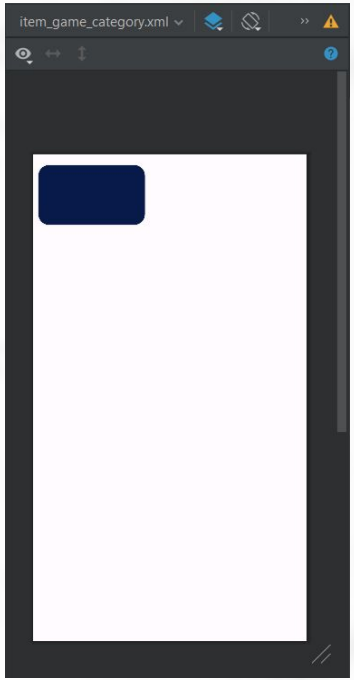
```
<?xml version="1.0" encoding="utf-8"?>
<androidx.cardview.widget.CardView
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/viewContainer"
    android:layout_width="160dp"
    android:layout_height="90dp"
    app:cardBackgroundColor="@color/bgapp_background_card"
    app:cardCornerRadius="16dp">
```



6. RECYCLERVIEW.ADAPTER

Dentro colocaremos un LinearLayout con orientación vertical. Además, añadiremos al CardView márgenes horizontal y vertical de 8dp y 16dp, respectivamente.

```
<androidx.cardview.widget.CardView...
   android:layout_marginHorizontal="8dp"
   android:layout_marginVertical="16dp"...>

   <LinearLayout
      android:layout_width="match_parent"
      android:layout_height="match_parent"
      android:gravity="center_vertical"
      android:layout_margin="8dp"
      android:orientation="vertical">
```



6. RECYCLERVIEW.ADAPTER

Dentro del LinearLayout añadimos un TextView con un texto de prueba y su identificador correspondiente, ya que este layout servirá de plantilla para el ViewHolder.

```
<TextView
    android:id="@+id/tvCategoryName"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textColor="@color/white"
    android:textSize="19sp"
    android:textStyle="bold"
    tools:text="PRUEBA" />
```

6. RECYCLERVIEW.ADAPTER

Y finalmente, como elemento decorativo, insertamos un View simple con un color blanco provisional que posteriormente cambiaremos según la categoría elegida.

```
<View
    android:id="@+id/divider"
    android:layout_width="match_parent"
    android:layout_height="2dp"
    android:layout_marginTop="4dp"
    tools:background="@color/white" />
```

6. RECYCLERVIEW.ADAPTER

En el método onCreateViewHolder() debemos ahora declarar una variable que tome esta plantilla y se la pase a la clase CategoriesViewholder() necesaria para que funcione el RecyclerView y que crearemos luego.

```
override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
CategoriesViewHolder {
    val view = LayoutInflater.from(parent.context).inflate(R.layout.item_game_category,
parent, false)
    return CategoriesViewHolder(view)
}
```

6. RECYCLERVIEW.ADAPTER

En el último método que nos queda del Adapter tenemos que añadirle una función que declararemos en el ViewHolder y que podemos llamar *render* porque le vamos a pasar la posición de cada ítem que tiene que pintar.

```
override fun onBindViewHolder(holder: CategoriesViewHolder, position: Int) {
    holder.render(categories[position])
}
```

7. RECYCLERVIEW.VIEWHOLDER

Lo primero que tenemos que incluir en el ViewHolder es la función *render* que hemos insertado en el Adapter. También tenemos que declarar los objetos del layout que querremos pintar desde este método.

```
class CategoriesViewHolder(view: View) : RecyclerView.ViewHolder(view) {

    private val tvCategoryName: TextView = view.findViewById(R.id.tvCategoryName)
    private val divider: View = view.findViewById(R.id.divider)

    fun render(gameCategory: GameCategory){

    }
}
```

7. RECYCLERVIEW.VIEWHOLDER

Y para estas categorías, cuando el Adapter pase por cada una recorriendo sus posiciones en el método onBindViewHolder, insertaremos un texto y el color asociado al *divider* que hemos diseñado en *colors.xml*.

```
when(gameCategory){
    GameCategory.Cooperative -> {
        tvCategoryName.text = "Cooperativos"
        divider.setBackgroundColor(getColor(divider.context, R.color.bgapp_cooperative_category))
    }
    GameCategory.Deckbuilding -> {
        tvCategoryName.text = "Deckbuilding"
        divider.setBackgroundColor(getColor(divider.context, R.color.bgapp_deckbuilding_category))
    }
    GameCategory.Euro -> {...
```

7. RECYCLERVIEW.VIEWHOLDER

Ahora, en la actividad principal *BoardgamesActivity* nos falta crear la lista de categorías que se le pasan al Adapter a través de la función initUI(). Si importamos la clase GameCategory solo escribiremos las subclases.

```
class BoardgamesActivity : AppCompatActivity() {

    private val categories = listOf(
        GameCategory.Cooperative,
        GameCategory.Deckbuilding,
        GameCategory.Euro,
        GameCategory.LCG,
        GameCategory.Legacy
    )
```

→

```
    private val categories = listOf(
        Cooperative,
        Deckbuilding,
        Euro,
        LCG,
        Legacy,
    )
```

7. RECYCLERVIEW.VIEWHOLDER

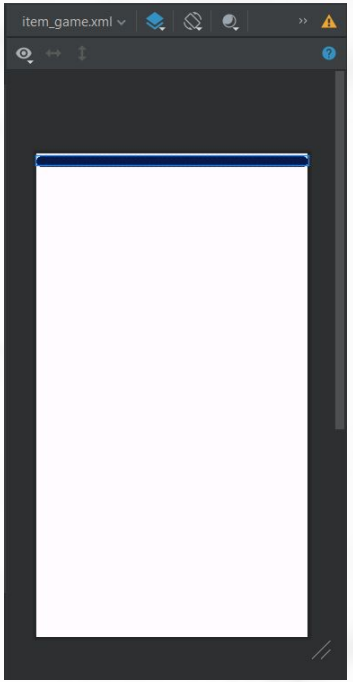
Por último, en la función initUI() debemos definir la orientación del RecyclerView importando la clase LinearLayoutManager() y asignarle el Adapter con la lista asociada a él. Ahora podemos ejecutar la aplicación.

```
private fun initUI(){
    categoriesAdapter = CategoriesAdapter(categories)

    rvCategories.layoutManager = LinearLayoutManager(this,
LinearLayoutManager.HORIZONTAL, false)

    rvCategories.adapter = categoriesAdapter
}
```

8. ITEM_GAME.XML

Para crear los ítems del segundo RecyclerView, con scroll vertical, realizamos unos pasos similares a los realizados anteriormente. Partimos de un CardView:

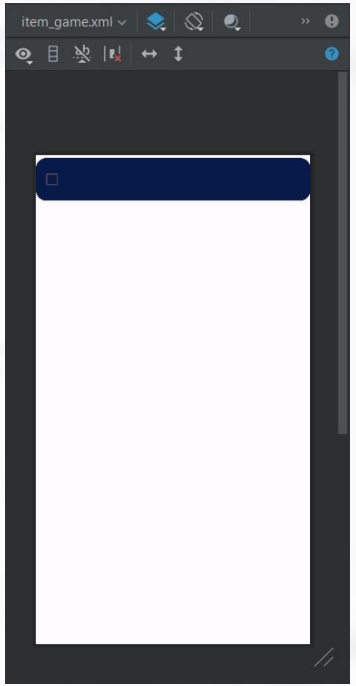
```
<androidx.cardview.widget.CardView
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginVertical="4dp"
    app:cardBackgroundColor="@color/bgapp_background_card"
    app:cardCornerRadius="16dp">
```



8. ITEM_GAME.XML

Dentro colocamos un LinearLayout horizontal con un elemento CheckBox que identificaremos para poder interactuar posteriormente.

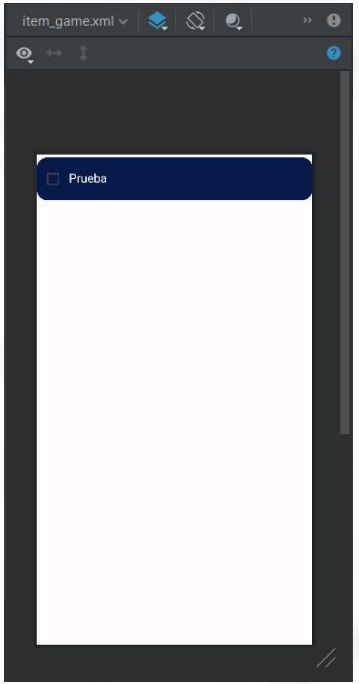
```
<LinearLayout
    android:paddingVertical="8dp"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal">

    <CheckBox
        android:id="@+id/cbGame"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
```



8. ITEM_GAME.XML

Y además un TextView que se ajuste al contenido con tamaño de fuente 18sp de color blanco con el texto “Prueba” provisional.

```
<TextView
    android:id="@+id/tvGame"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textColor="@color/white"
    android:textSize="18sp"
    tools:text="Prueba"/>
```



9. SEGUNDO RECYCLERVIEW

Pasos para crear un RecyclerView:

- Construir una clase que defina los elementos que se mostrarán en él.
- Definir la clase Adapter:
  - getItemCount() = elementos.size
  - onCreateViewHolder() → pasa la vista del layout definido a la clase ViewHolder
  - onBindViewHolder → recorre la lista de elementos dibujándolos mediante la clase Viewholder

9. SEGUNDO RECYCLERVIEW

Pasos para crear un RecyclerView:

- Definir la clase ViewHolder:
  - Declarar los objetos asociados a los Views del item_layout
  - Recorrer con *when* los elementos de la clase creada para ello
- Función principal:
  - Pasarle a la clase Adapter una lista de elementos
  - Definir su *layoutmanager*
  - Asignarle al RecyclerView del layout la clase Adapter definida

9. SEGUNDO RECYCLERVIEW

Repite los pasos realizados para el primer RecyclerView y crea el segundo definiéndolo con un scroll vertical y tomando como plantilla los elementos del layout *item_game.xml* que acabamos de diseñar.

Recuerda que la clase *Game.kt* con la que hacer la lista de elementos que pasarle al Adapter ya está creada.

Crea un elemento *game* de prueba para cada categoría y pásaselos al Adapter mediante una lista para comprobar cómo los pinta al ejecutar la aplicación.

9. SEGUNDO RECYCLERVIEW

La función render() dentro del ViewHolder tendrá la siguiente forma:

```
fun render(game: Game) {
    tvGame.text = game.name

    val color = when(game.category){
        GameCategory.Cooperative -> R.color. bgapp_cooperative_category
        GameCategory.Deckbuilding -> R.color. bgapp_deckbuilding_category
        GameCategory.Euro -> R.color. bgapp_euro_category
        GameCategory.LCG -> R.color. bgapp_lcg_category
        GameCategory.Legacy -> R.color. bgapp_legacy_category
    }

    cbGame.buttonTintList =
ColorStateList.valueOf(ContextCompat.getColor( cbGame.context, color))
}
```

10. BOTÓN PARA AÑADIR ELEMENTOS

Queremos insertar un botón en la aplicación que permita añadir nuevos juegos dentro de sus categorías. Para ello vamos a crear un FrameLayout encima del LinearLayout.

```
<FrameLayout
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">

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
```

10. BOTÓN PARA AÑADIR ELEMENTOS

Y tras este último colocaremos el botón situado en la parte inferior derecha de la pantalla. El LinearLayout seguirá ocupando todo el ancho y alto de la pantalla.

```
<com.google.android.material.floatingactionbutton.FloatingActionButton
    android:id="@+id/fabAddGame"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="end|bottom"
    android:layout_margin="32dp"
    android:src="@android:drawable/ic_input_add"
    android:contentDescription="Añade un juego"
    app:backgroundTint="@color/bgapp_accent"
    app:tint="@color/white"/>
```

10. BOTÓN PARA AÑADIR ELEMENTOS

Pasamos a la lógica y lo primero que debemos hacer es declarar este botón e inicializarlo. Necesitaremos el método setOnclickListener() para la interacción con el mismo y queremos mostrar un diálogo al pulsarlo.

```
private lateinit var fabAddGame: FloatingActionButton

private fun initComponents() {
    rvCategories = findViewById(R.id.rvCategories)
    rvGames = findViewById(R.id.rvGames)
    fabAddGame = findViewById(R.id.fabAddGame)
}

private fun initListeners() { fabAddGame.setOnClickListener{ showDialog() }
}
```

11. DIALOG

Los diálogos son vistas que se superponen al contenido y toman el foco principal de la aplicación:

https://developer.android.com/guide/topics/ui/dialogs?hl=es-419

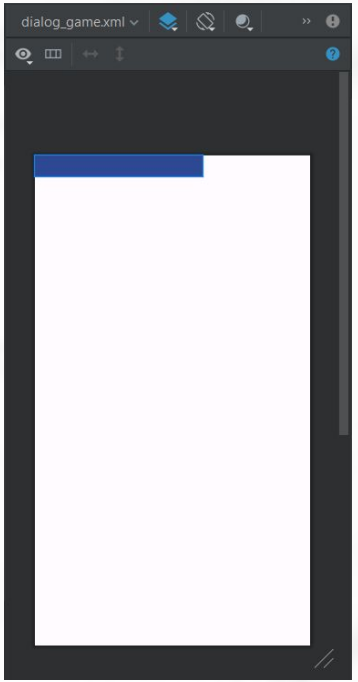
Vamos a declarar la variable que lo va a contener y lo primero que debemos hacer es crear la vista que va a mostrar, la cual agregaremos posteriormente con el método setContentView().

```
private fun showDialog() {
    val dialog = Dialog(this)
    dialog.setContentView(R.layout.dialog_game)
}
```

11. DIALOG

Creamos el archivo *dialog_game.xml* dentro del directorio /res/layouts. Vamos a utilizar un LinearLayout vertical de anchura 250dp y cuya altura se ajuste al contenido.

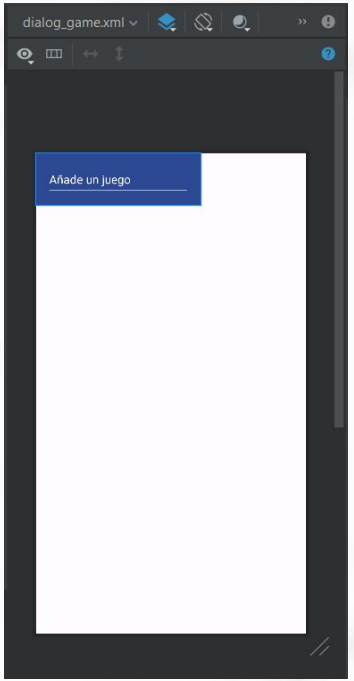
```
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="250dp"
   android:layout_height="wrap_content"
   android:padding="16dp"
   android:background="@color/bgapp_background_app"
   android:orientation="vertical">
```



11. DIALOG

En la parte superior pondremos un EditText en el que el usuario podrá introducir el nombre del juego de mesa que quiera añadir a la lista.

```
<EditText
    android:id="@+id/etGame"
    android:layout_width="match_parent"
    android:layout_height="48dp"
    android:backgroundTint="@color/white"
    android:hint="@string/dialog_add_game"
    android:textColor="@color/white"
    android:textColorHint="@color/white"
    android:importantForAutofill="no"
    android:inputType="text" />
```

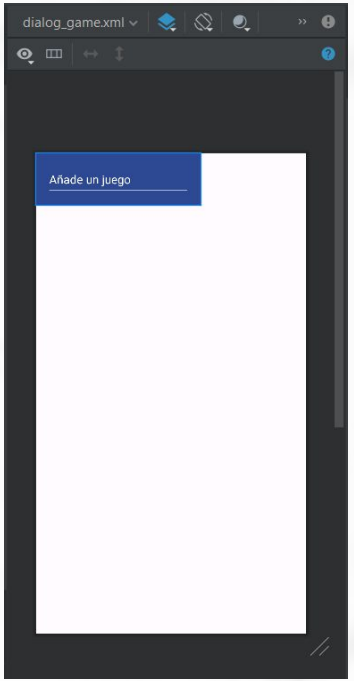


T5. RECYCLERVIEW Y DIALOG

11. DIALOG

Ahora vamos a ver como poner una lista de elementos con respuesta única, mediante la vista RadioGroup, para que el usuario seleccione la categoría del juego.

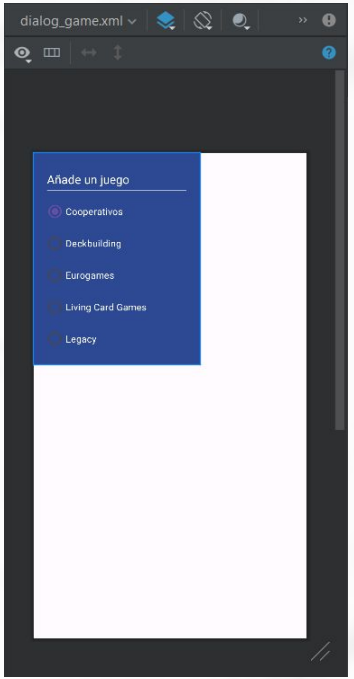
```
<RadioGroup
    android:id="@+id/rgCategories"
    android:layout_width="match_parent"
    android:layout_height="wrap_content">
```



11. DIALOG

Entre las etiquetas de apertura y cierre de RadioGroup podemos insertar tantos elementos RadioButton como necesitemos, en nuestro caso 5.

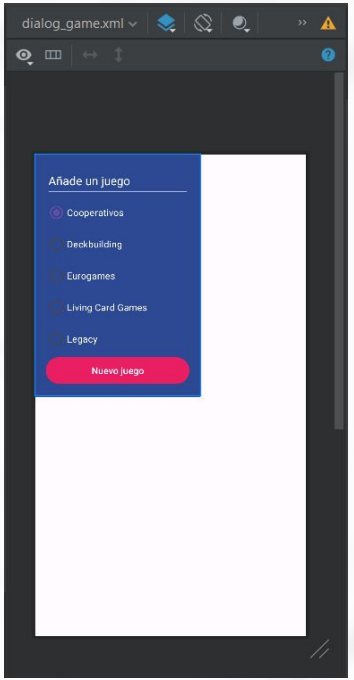
```
<RadioButton
    android:id="@+id/rbCooperative"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:buttonTint="@color/bgapp_cooperative_category"
    android:checked="true"
    android:text="@string/dialog_cooperative_category"
    android:textColor="@color/white" />
```



11. DIALOG

Y por último, detrás del RadioGroup, diseñaremos un Button que añada el juego a la lista cuando el usuario haya escrito el nombre y seleccionado la categoría.

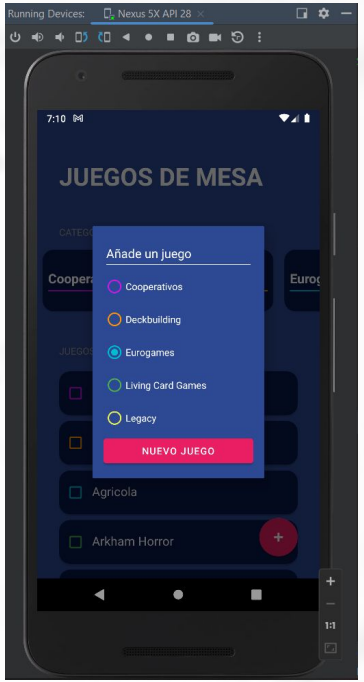
```
<Button
    android:id="@+id/btnAddGame"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="@string/dialog_newgame_button"
    android:textColor="@color/white"
    android:backgroundTint="@color/bgapp_accent"/>
```



11. DIALOG

Si queremos ejecutar la aplicación y comprobar el funcionamiento de nuestro Dialog hasta el momento, podemos usar el método show() para el mismo.

```
private fun showDialog() {
    val dialog = Dialog(this)
    dialog.setContentView(R.layout.dialog_game)
    dialog.show()
}
```



11. DIALOG

Dentro de la función showDialog() declararemos las variables necesarias para interactuar con cada uno de los elementos que hemos identificado en el layout. El comando dialog.show() lo dejaremos al final de la función.

```
private fun showDialog() {
    val dialog = Dialog(this)
    dialog.setContentView(R.layout.dialog_game)

    val btnAddGame: Button = dialog.findViewById(R.id.btnAddGame)
    val etGame: EditText = dialog.findViewById(R.id.etGame)
    val rgCategories: RadioGroup = dialog.findViewById(R.id.rgCategories)

    dialog.show()
}
```

11. DIALOG

Iniciaremos un método setOnClickListener() para el Button que recoja el nombre del juego y la categoría seleccionada por el usuario únicamente cuando el botón haya sido pulsado.

```
btnAddGame.setOnClickListener {
    val currentGame = etGame.text.toString()
    if(currentGame.isNotEmpty()){
        val selectedId = rgCategories.checkedRadioButtonId
        val selectedRadioButton: RadioButton = rgCategories.findViewById(selectedId)
        val currentCategory: GameCategory = when(selectedRadioButton.text){
            getString(R.string.dialog_cooperative_category) -> Cooperative
            getString(R.string.dialog_deckbuilding_category) -> Deckbuilding
            getString(R.string.dialog_euro_category) -> Euro
            getString(R.string.dialog_lcg_category) -> LCG
```

11. DIALOG

Y todavía dentro de la estructura if(), añadiremos a la mutableListOf *games* (declarada al principio del código) un objeto Game con el nombre introducido por el usuario y la categoría seleccionada.

```
                getString(R.string.dialog_lcg_category) -> LCG
                else -> Legacy
            }
            games.add(Game(currentGame, currentCategory))
        }
    }
    dialog.show()
}
```

11. DIALOG

El problema es que hay que comunicarle al Adapter que la lista se ha actualizado y, para ello, crearemos una función updateGames() que se encargue de esto.

```
                getString(R.string.dialog_lcg_category) -> LCG
                else -> Legacy
            }
            games.add(Game(currentGame, currentCategory))
            updateGames()
            dialog.hide()
        }
    }
    dialog.show()
}
```

11. DIALOG

Dicha función contendrá un método notifyDataSetChanged() sobre el Adapter. Existen otros métodos como notifyItemInserted() pero más adelante necesitaremos indicar que ha cambiado el conjunto de datos.

```
private fun updateGames(){
    gamesAdapter.notifyDataSetChanged()
}
```

12. EXPRESIONES LAMBDA

Las expresiones lambda usan una sintaxis concisa para definir una función sin la palabra clave `fun`. Puedes almacenar una expresión lambda directamente en una variable.

```
fun main() {
    val trickFunction = trick
    trick()                 //Muestra por pantalla "No treats!!"
    trickFunction()         //Muestra por pantalla "No treats!!"
}
val trick = {
    println("No treats!")
}
//https://developer.android.com/codelabs/basic-android-kotlin-compose-function-types-and-lambda?hl=es-419#2
```

12. EXPRESIONES LAMBDA

Nuestro objetivo ahora es que los juegos seleccionados en la lista del segundo RecyclerView muestren su nombre tachado. Para ello, vamos a crear una nueva función onGameSelected() que pasarle al Adapter.

```
private fun onGameSelected(position:Int){
    games[position].isSelected = !games[position].isSelected
    updateGames()
}
```

12. EXPRESIONES LAMBDA

Cada uno de los elementos debería tener un setOnClickListener() pero eso sería muy engorroso y para eso están las funciones lambda. A nuestro GamesAdapter() le pasamos esta función como variable.

```
class GamesAdapter(private val games: List<Game>, private val onItemSelected: (Int)
-> Unit) :
    RecyclerView.Adapter<GamesViewholder>() {

    ...

}
```

12. EXPRESIONES LAMBDA

Y desde éste, hacemos que el ViewHolder ejecute nuestra función sobre cada posición de la lista de juegos. El atributo *itemView* hace referencia a toda la vista del RecyclerView.

```
override fun onBindViewHolder(holder: GamesViewholder, position: Int) {
    holder.render(games[position])
    holder.itemView.setOnClickListener{ onItemSelected(position) }
}
```

12. EXPRESIONES LAMBDA

Por último, para pasar la función lambda como parámetro, tenemos que ir a la inicialización del Adapter en la función initUI() y escribirla entre llaves. Especificamos el parámetro *position*, aunque sería suficiente con *it*.

```
private fun initUI(){
    categoriesAdapter = CategoriesAdapter(categories)
    rvCategories.layoutManager = LinearLayoutManager(this,
LinearLayoutManager.HORIZONTAL, false)
    rvCategories.adapter = categoriesAdapter

    gamesAdapter = GamesAdapter(games) {position -> onGameSelected(position)}
    rvGames.layoutManager = LinearLayoutManager(this)
    rvGames.adapter = gamesAdapter
}
```

12. EXPRESIONES LAMBDA

Para tachar el nombre del juego al marcarlo, en GamesViewHolder() insertaremos las siguientes líneas de código dentro de la función render().

```
fun render(game: Game) {
    if (game.isSelected) {
        tvGame.paintFlags = tvGame.paintFlags or Paint.STRIKE_THRU_TEXT_FLAG
    } else {
        tvGame.paintFlags = tvGame.paintFlags and Paint.STRIKE_THRU_TEXT_FLAG.inv()
    }

    cbGame.isChecked = game.isSelected

    tvGame.text = game.name
```

12. EXPRESIONES LAMBDA

Crea otra función lambda onCategorieSelected() para interactuar con CategoriesAdapter(). Añádele el siguiente método:

```
categoriesAdapter.notifyItemChanged(position)
```

Pasa la función como parámetro al Adapter y añádelo también como parámetro en el método onBindViewHolder() para la función render().

```
override fun onBindViewHolder(holder: CategoriesViewHolder, position: Int) {
    holder.render(categories[position], onItemSelected)
}
```

12. EXPRESIONES LAMBDA

Ahora debemos declarar este parámetro en la función render() de nuestro CategoriesViewHolder() y añadir una variable que almacene el CardView de los elementos de la clase *GameCategory*.

```
private val tvCategoryName: TextView = view.findViewById(R.id.tvCategoryName)
private val divider: View = view.findViewById(R.id.divider)
private val viewContainer: CardView = view.findViewById(R.id.viewContainer)

fun render(gameCategory: GameCategory, onItemSelected: (Int) -> Unit){

    when(gameCategory){
        GameCategory.Cooperative -> {
            tvCategoryName.text = "Cooperativos"
            divider.setBackgroundColor(getColor(divider.context, R.color.bgapp...
```

12. EXPRESIONES LAMBDA

Haremos que cada categoría cambie de color si está seleccionada o no y además deberemos añadir un método setOnClickListener() para cada uno de los elementos recorridos por el Adapter.

```
val color = if (gameCategory.isSelected) {
    R.color.bgapp_background_card
} else {
    R.color.bgapp_background_disabled
}

viewContainer.setCardBackgroundColor(ContextCompat.getColor(viewContainer.context,
color))

itemView.setOnClickListener { onItemSelected(layoutPosition) }
```

12. EXPRESIONES LAMBDA

Y además queremos que muestre solo los juegos de las categorías que estén seleccionadas. Para ello, nos vamos a la función updateGames() y creamos una lista que aplique este filtro.

```
private fun updateGames(){
    val selectedCategories: List<GameCategory> =categories.filter { it.isSelected }

    gamesAdapter.notifyDataSetChanged()
}
```

12. EXPRESIONES LAMBDA

Una vez se hayan filtrado las categorías, filtraremos los juegos de dichas categorías. Es decir, de la lista de juegos debe comprobar si en la lista de categorías seleccionadas está el atributo *category* de los objetos *Game*.

```
private fun updateGames(){
    val selectedCategories: List<GameCategory> =categories.filter { it.isSelected }
    val newGames = games.filter { selectedCategories.contains(it.category) }

    gamesAdapter.notifyDataSetChanged()
}
```

12. EXPRESIONES LAMBDA

Y finalmente, le pasamos esta nueva lista de juegos al Adapter. Pero habrá que tener en cuenta que nuestra variable *games* en GamesAdapter la hemos declarado como *private val* y habrá que cambiarla a *var*.

```
private fun updateGames(){
    val selectedCategories: List<GameCategory> =categories.filter { it.isSelected }
    val newGames = games.filter { selectedCategories.contains(it.category) }
    gamesAdapter.games = newGames

    gamesAdapter.notifyDataSetChanged()
}
```

13. EJECUCIÓN DE LA APLICACIÓN

Una vez ejecutada la aplicación y comprobado que su funcionamiento es el correcto conviene repasar todos los conceptos aprendidos en este tema:

- RecyclerView y sus subclases.
- Funciones lambda y dialog.
- Sealed class y data class.
- Métodos nuevos para distintas clases de objetos.

Es importante organizar un proyecto conforme se va elaborando. Se debe hacer uso de funciones y parcelar el código continuamente.