Programación Multimedia y Dispositivos Móviles

UD 8. Menús y preferencias de usuario

Javier Carrasco

Curso 2024 / 2025

Menús y preferencias de usuario

8. Menús y preferencias de usuario

Los **menús** son uno de elementos más comunes en las aplicaciones Android. Se recomienda utilizar la API `Menu` para dar al usuario la experiencia de elemento conocido, y no suponga tener que familiarizarse con cada nueva aplicación que se instale. Desde la versión Android 3.0 (API 11) ya no es necesario facilitar un botón exclusivo para el menú.

A pesar de los cambios sufridos a los largo de su vida, en diseño sobretodo, se siguen manteniendo los tres tipos de menú existentes, menú de **opciones**, **contextual** y **emergente**.

8.1. Definición de un menú XML

Para la creación de menús, ya sean de un tipo u otro, estos se basarán en un recurso XML para su definición, en vez de añadir código a las clases, separando así diseño y funcionalidad, por lo que únicamente se deberán cargar mediante el uso de la clase `Menu` en una actividad o fragmento.

Para definir menús, se deberá crear un archivo XML dentro del directorio `res/menu/` del proyecto y definirlo mediante los siguientes elementos:

- **<menu>** Define un menú como tal, será el contenedor de los elementos. Un elemento *<menu>* será el nodo raíz del archivo y podrá tener uno o más elementos *<item>* y *<group>*.
- **<item>** Creará un elemento de menú. Este elemento podrá contener elementos *<menu>* anidados para crear sub-menús.
- **<group>** Este contenedor es opcional e invisible para elementos *<item>*. Permitirá categorizar o agrupar los elementos del menú que compartan propiedades, como el estado de una actividad o visibilidad.

Para crear un XML que defina un menú, lo más rápido es utilizar el botón derecho sobre el proyecto y seleccionar la opción ***New > Android Resource File***.

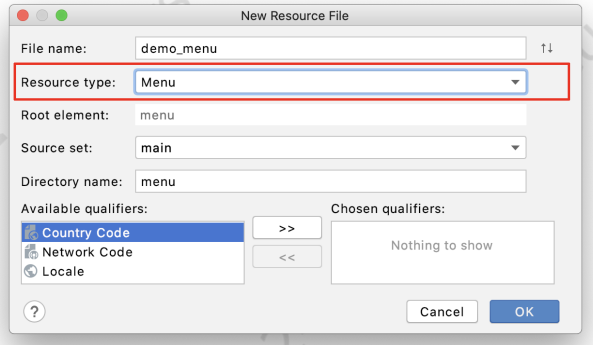

Figura 1

Tras indicar el nombre y tipo de *resource*, podrás ver una nueva sub-carpeta dentro de `res`, la sub-carpeta *"menu"*, donde se almacenarán todos los menús creados para la aplicación.

Una vez creado el fichero XML, ya se podrá editar y, al igual que con las *activities*, se podrá hacer desde el editor, o desde modo texto.

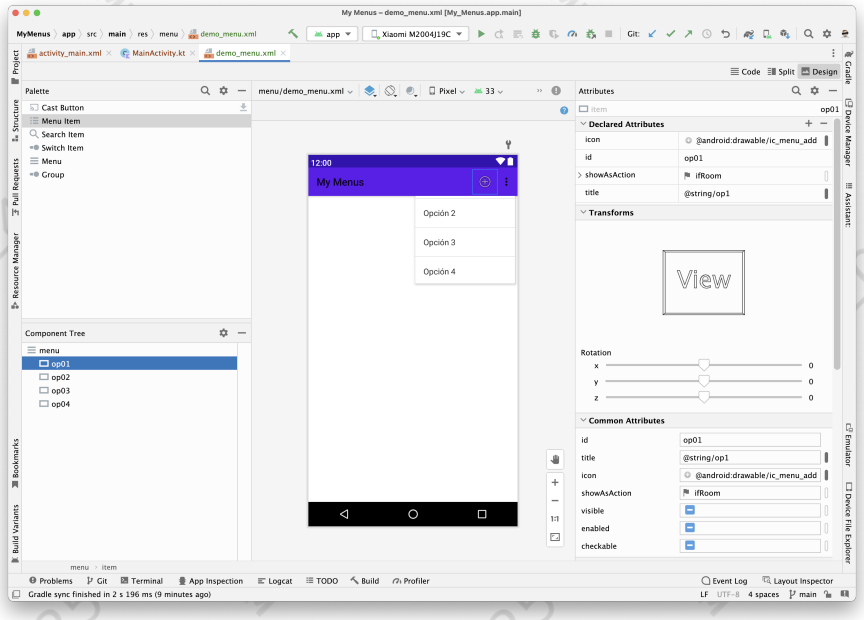

Figura 2

```xml
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto">

    <item
        android:id="@+id/op01"
        android:icon="@android:drawable/ic_input_add"
        android:title="@string/op1"
        app:showAsAction="ifRoom" />
    <item
        android:id="@+id/op02"
        android:title="@string/op2" />
    <item
        android:id="@+id/op03"
        android:title="@string/op3" />
    <item
        android:id="@+id/op04"
        android:title="@string/op4" />
</menu>
```

Las propiedades más comunes del elemento `<item>` son los siguientes:

- **android:id** identificador del recurso.
- **android:icon** icono a mostrar en la opción de menú.
- **android:title** título del elemento, texto que se mostrará para identificar la acción.
- **android:showAsAction** indica cuándo y cómo, deberá el elemento aparecer como un elemento de acción en la barra de la *app*. Esta propiedad dispone de cuatro opciones, puedes ver el resultado seleccionándolas en la vista diseño.

Ahora se añadirá un sub-menú a la *opción 2* para completar el menú de ejemplo.

```
<item
    android:id="@+id/op02"
    android:title="@string/op2">
    <menu>
        <item
            android:id="@+id/op021"
            android:title="@string/op21" />
        <item
            android:id="@+id/op022"
            android:title="@string/op22" />
    </menu>
</item>
```

8.2. Menú de opciones y barra de acción

Los menús de opciones y barra contienen las acciones, u opciones, que afectan a la aplicación. A continuación, se verá como darle vida al menú creado en el apartado anterior. En primer lugar, se deberá *"inflar"* el menú, para ello, habrá que sobrecargar el método `onCreateOptionsMenu()`, haciendo uso de la clase `MenuInflater` para crear objetos `Menu` a partir de ficheros XML.

```
// Se infla el menú para mostrarlo en la barra de acción.
override fun onCreateOptionsMenu(menu: Menu?): Boolean {
    val inflate = menuInflater
    inflate.inflate(R.menu.demo_menu, menu)
    return true
}
```

El siguiente paso será sobrecargar el método `onOptionsItemSelected()`, que permitirá conocer que opción del menú ha seleccionado el usuario.

```
// Método encargado de gestionar las opciones pulsadas del menú.
override fun onOptionsItemSelected(item: MenuItem): Boolean {
    return when (item.itemId) {
        R.id.op01 -> {
            Log.d("MENU", "${getString(R.string.op1)} seleccionada")
```

```
            myToast("${getString(R.string.op1)} seleccionada")
            true
        }
        // La opción op02 no haría falta, ya que abre el sub-menú.
        R.id.op021 -> {
            Log.d("MENU", "${getString(R.string.op21)} seleccionada")
            myToast("${getString(R.string.op21)} seleccionada")
            true
        }
        R.id.op022 -> {
            Log.d("MENU", "${getString(R.string.op22)} seleccionada")
            myToast("${getString(R.string.op22)} seleccionada")
            true
        }
        R.id.op03 -> {
            Log.d("MENU", "${getString(R.string.op3)} seleccionada")
            myToast("${getString(R.string.op3)} seleccionada")
            true
        }
        R.id.op04 -> {
            Log.d("MENU", "${getString(R.string.op4)} seleccionada")
            myToast("${getString(R.string.op4)} seleccionada")
            true
        }
        else -> false
    }
}
```

Con esto se obtendría un menú relativamente funcional. Si te fijas en la imagen, verás que la *opción 1*, a la que se le ha asignado signo **+** como icono, aparece en la barra, esto se debe a que tiene la propiedad `app:showAsAction="ifRoom"` activada en el XML. Todo lo hecho servirá para las opciones que se configuren para que aparezcan en la barra.

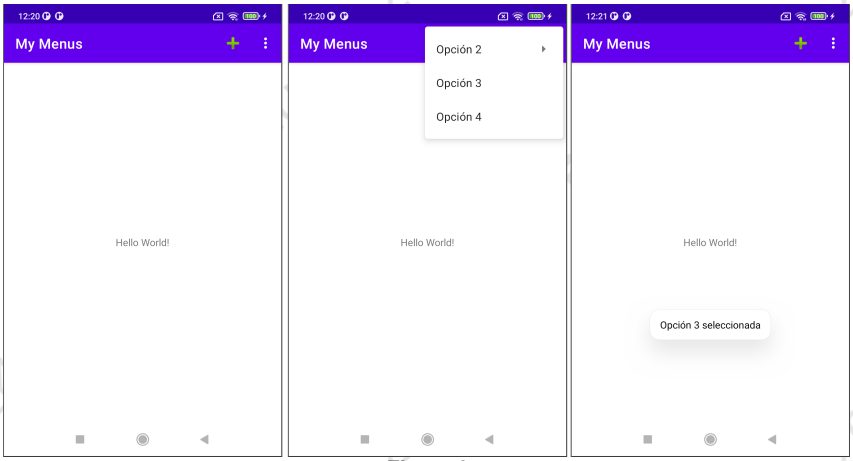

Figura 3

Si prefieres añadir tu propia *Toolbar*, haciendo uso de un tema *NoActionBar*, la forma de inflar el menú y gestionar sus opciones cambia ligeramente. Por ejemplo, con el *ViewBinding* activado, y habiendo añadido una *Toolbar* con nombre *mToolbar* como identificador, el inflado del menú se hará utilizando la siguiente línea.

```
binding.mToolbar.inflateMenu(R.menu.demo_menu)
```

El manejo de la opción seleccionada también varía, y como el inflado, se aplica directamente sobre la propia *Toolbar*.

```
binding.mToolbar.setOnMenuItemClickListener {
    return when (item.itemId) {
        R.id.op01 -> {
            Log.d("MENU", "${getString(R.string.op1)} seleccionada")
            myToast("${getString(R.string.op1)} seleccionada")
            true
        }
        ...
        else -> false
    }
}
```

8.3. Menú contextual y modo de acción contextual

Este tipo de menús afectará únicamente a elementos específicos dentro de la UI de la aplicación. Se puede asignar un menú contextual a cualquier tipo de vista, aunque es muy utilizado en *ListView*, *GridView* o *RecyclerView*, así como otras colecciones. Este tipo de menús permite realizar operaciones concretas sobre elementos concretos.

8.3.1. Menú contextual

El menú contextual aparece como una lista flotante similar a un cuadro de diálogo. Se muestra cuando se hace una pulsación larga sobre un elemento que admite este tipo de menús. Únicamente se podrá asignar una acción por elemento.

Para crearlo, en primer lugar deberás crear el contenido del menú contextual, siguiendo los pasos vistos para crear el menú anterior, crea el fichero `context_menu.xml`.

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item
        android:id="@+id/option01"
        android:title="@string/op1" />
    <item
        android:id="@+id/option02"
        android:title="@string/op2" />
</menu>
```

La `activity_main.xml` únicamente contendrá un *ListView* para mostrar una lista de nombres. A continuación, puedes ver el código que se añadirá a la clase principal (*MainActivity*). En primer lugar se añadirá una nueva propiedad llamada persona, un array, que contendrá los nombres a mostrar en la lista.

```
private val personas = arrayOf(
        "Javier", "Pedro", "Nacho", "Patricia", "Miguel", "Susana", "Raquel", "Antonio",
        "Andrea", "Nicolás", "Juan José", "José Antonio", "Daniela", "María", "Verónica",
        "Natalia")
```

En el método `onCreate` se montará la lista de nombres utilizando un *ListView* y la clase *ArrayAdapter*. Recuerda añadir el componente *ListView* en la actividad.

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

    // Se crea el adapter para la lista de nombres.
    val arrayAdapter: ArrayAdapter<String> = ArrayAdapter(
        this, android.R.layout.simple_expandable_list_item_1,
        personas.sortedArray()
    )

    with(binding) {
        // Se cargan los datos en la lista.
        myListView.adapter = arrayAdapter
        // Se registra el uso de un menú contextual al ListView.
        registerForContextMenu(myListView)
        // Acción sobre el elemento de la lista pulsado.
        myListView.setOnItemClickListener { parent, view, position, id ->
            myToast("Pulsado $id - ${myListView.getItemAtPosition(position)}")
        }
    }
}
```

Si lanzas la aplicación, tendrás un listado de nombres, ordenados, que al pulsar cada uno de ellos mostrará un *Toast* indicando la posición y el nombre pulsado. El siguiente paso será *"inflar"* el menú contextual, para ello se deberá sobrecargar el método `onCreateContextMenu`.

```
// Se "infla" el menú contextual con el resource. Se ejecuta tras el registro.
override fun onCreateContextMenu(
    menu: ContextMenu?,
    v: View?,
    menuInfo: ContextMenu.ContextMenuInfo?
) {
    menuInflater.inflate(R.menu.context_menu, menu)
}
```

Ahora, si mantienes la pulsación sobre cualquier elemento de la lista, aparecerá el menú

contextual sobre el elemento. Ya solo quedaría sobrecargar el método `onContextItemSelected` para conocer que opción del menú contextual ha pulsado el usuario.

```
// Se comprueba la opción de menú seleccionada y sobre que ítem se ha ejecutado.
override fun onContextItemSelected(item: MenuItem): Boolean {
    // Se obtiene el nombre de la persona, con ApaterView.AdapterContextMenuInfo se obtiene
    // la posición sobre la que se ha hecho clic.
    val info = item.menuInfo as AdapterView.AdapterContextMenuInfo
    val posicion = info.position
    val nombre = personas.sortedArray()[posicion]

    return when (item.itemId) {
        R.id.option01 -> {
            myToast("Opción 1: $nombre")
            true
        }
        R.id.option02 -> {
            myToast("Opción 2: $nombre")
            true
        }
        else -> false
    }
}
```

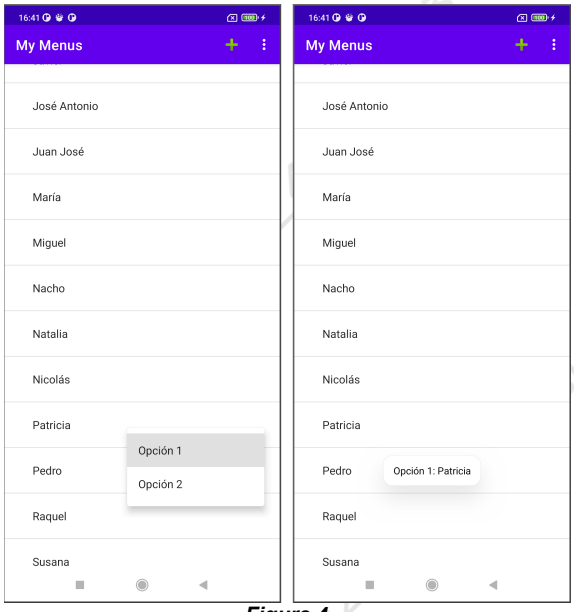

Figura 4

Ahora bien, si se quiere añadir este menú a un elemento en concreto, por ejemplo, una imagen de la UI como la siguiente.

```
<ImageView
```

```
    android:id="@+id/imageView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:srcCompat="@mipmap/ic_launcher" />
```

Tras sobrecargar `onCreateContextMenu()` y `onContextItemSelected()` como se ha visto, sin la necesidad de identificar sobre que *item* se ha ejecutado la acción, bastará con registrar el menú al elemento.

```
// Se registra el menú contextual al ImageView.
registerForContextMenu(imageView)
```

8.3.2. Modo de acción contextual

Este tipo de menú es una implementación del sistema `ActionMode`[1], que mostrará una serie de acciones contextuales en la barra de acción (*ActionBar*). Se partirá de un proyecto nuevo (*My Menus 2*) con una etiqueta en la UI del usuario, sobre la cual, tras una pulsación larga se activará el modo de acción. En primer lugar, deberá crearse el menú, y como deberán aparecer en la barra, se asignará un icono a cada opción, para que quede bonito.

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item
        android:id="@+id/option01"
        android:icon="@android:drawable/ic_menu_delete"
        android:title="@string/menu_op01" />
    <item
        android:id="@+id/option02"
        android:icon="@android:drawable/ic_menu_edit"
        android:title="@string/menu_op02" />
</menu>
```

En la clase principal, se creará la variable `actionMode`, esta se utilizará para controlar si el elemento `ActionMode` está activo, si no lo está su valor será `null`. Esta variable estará instanciada como una propiedad de la clase, como puedes ver a continuación.

```
private var actionMode: ActionMode? = null
```

Seguidamente, deberás implementar la interfaz `ActionMode.Callback` de `android.view.ActionMode` para configurar el modo de acción contextual. Si te fijas en el código, verás que muchos de los métodos ya se conocen, y su funcionamiento es muy parecido.

```
// Modo de acción contextual.
private val actionModeCallback = object : ActionMode.Callback {
    // Método llamado al selección una opción del menú.
    override fun onActionItemClicked(mode: ActionMode?, item: MenuItem?): Boolean {
```

1 *ActionMode* (https://developer.android.com/reference/android/view/ActionMode.html)

```
        return when (item!!.itemId) {
            R.id.option01 -> {
                Toast.makeText(applicationContext, R.string.menu_op01,
                    Toast.LENGTH_SHORT).show()

                binding.etiqueta.visibility = View.GONE
                // Se cierra el menú.
                actionMode!!.finish()
                true
            }
            R.id.option02 -> {
                Toast.makeText(applicationContext, R.string.menu_op02,
                    Toast.LENGTH_SHORT).show()
                true
            }
            else -> false
        }
    }

    // Llamado al crear el modo acción a través de startActionMode().
    override fun onCreateActionMode(mode: ActionMode?, menu: Menu?): Boolean {
        menuInflater.inflate(R.menu.context_menu, menu)
        return true
    }
    // Se llama cada vez que el modo acción se muestra, después de onCreateActionMode().
    override fun onPrepareActionMode(mode: ActionMode?, menu: Menu?): Boolean {
        return false
    }

    // Se llama cuando el usuario sale del modo de acción.
    override fun onDestroyActionMode(mode: ActionMode?) {
        actionMode = null
    }
}
```

Para este tipo de menú no se deberá realizar el registro del menú, sino llamar al método `startActionMode()` para que el sistema ejecute `ActionMode`. Para este ejemplo, se utilizará el método `setOnLongClickListener()` sobre la vista en cuestión.

```
binding.etiqueta.setOnLongClickListener {
    when (actionMode) {
        null -> {
            // Se lanza el ActionMode.
            actionMode = it.startActionMode(actionModeCallback)
            it.isSelected = true
            true
        }
        else -> false
    }
}
```

El resultado de este código puedes verlo en la imagen que se muestra a continuación.

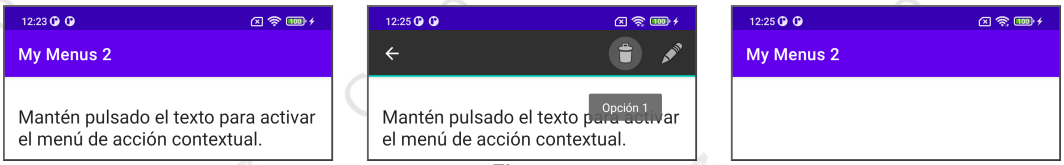

Figura 5

Menú de acción con selección múltiple

En el siguiente ejemplo, podrás ver como implementar el modo de acción sobre un *ListView*, creando un *BaseAdapter()* y permitiendo la **selección múltiple**. Partiendo de un proyecto nuevo (*My Menus 3*), crea un nuevo menú, `context_menu.xml`, parecido al visto en el ejemplo anterior.

```xml
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item
        android:id="@+id/optionDelete"
        android:icon="@android:drawable/ic_menu_delete"
        android:title="@string/menu_op_delete" />
    <item
        android:id="@+id/optionShare"
        android:icon="@android:drawable/ic_menu_share"
        android:title="@string/menu_op_share" />
</menu>
```

Seguidamente crea la vista (`item_layout.xml`) para personalizar los elementos del *ListView*, en esta se mostrará una etiqueta para los nombres y un *CheckBox* para poder hacer las selecciones.

```xml
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/myRelativeLayout"
    android:layout_width="match_parent"
    android:layout_height="50dp"
    android:orientation="vertical">

    <TextView
        android:id="@+id/person_name"
        android:layout_width="wrap_content"
        android:layout_height="match_parent"
        android:layout_alignParentStart="true"
        android:layout_marginStart="25dp"
        android:gravity="center"
        android:textSize="18sp"
        android:textStyle="bold"
        tools:text="nombre" />
```

```
    <CheckBox
        android:id="@+id/checkBox"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentEnd="true"
        android:layout_centerVertical="true"
        android:layout_marginStart="26dp"
        android:layout_marginEnd="25dp"
        android:visibility="gone" />
</RelativeLayout>
```

En primer lugar, se crearán las siguientes propiedades utilizando `companion object` en la `MainActivity`. La fuente de datos, el objeto `actionMode` que se utilizará para modificar algunas propiedades, como el título, una variable *bool* para controlar en que estado está, y la lista para almacenar la selección del usuario.

```
companion object {
    val personas: MutableList<String> = mutableListOf(
            "Javier", "Pedro", "Nacho", "Patricia", "Miguel", "Susana", "Raquel",
            "Antonio", "Andrea", "Nicolás", "Juan José", "José Antonio", "Daniela",
            "María", "Verónica", "Juan", "Carlos", "Isabel", "Óscar", "Víctor"
        )
    var actionMode: ActionMode? = null
    var isActionMode: Boolean = false
    var seleccion: MutableList<String> = ArrayList()
}
```

A continuación, crea la clase `ListViewAdapter.kt` que contendrá el *adapter* para el *ListView* heredando de la clase `BaseAdapter`. El método `getView()` *"inflará"* la vista utilizando el *layout* para los elementos de la lista, y el control de los elementos seleccionados.

```
class ListViewAdapter(var names: List<String>) : BaseAdapter() {

    override fun getView(position: Int, convertView: View?, parent: ViewGroup?): View {
        val bindingFila: ItemLayoutBinding
        // Se "infla" la vista.
        if (convertView == null) {
            bindingFila = ItemLayoutBinding.inflate(
                LayoutInflater.from(parent!!.context),
                parent,
                false
            )
        } else bindingFila = ItemLayoutBinding.bind(convertView)

        // Se asigna el nombre al TextView
        bindingFila.personName.text = this.getItem(position)

        // Se asigna como etiqueta del checkBox la posición en la que se encuentra.
        bindingFila.checkBox.tag = position
```

```
        if (MainActivity.isActionMode)
            bindingFila.checkBox.visibility = View.VISIBLE
        else bindingFila.checkBox.visibility = View.GONE

        // Se controla la selección del usuario mediante la lista selección.
        bindingFila.checkBox.setOnCheckedChangeListener { compoundButton, _ ->
            val pos: Int = compoundButton.tag.toString().toInt()
            Log.d("CHECKBOX", position.toString())

            // Se añade o elimina de la lista la selección.
            if (MainActivity.seleccion.contains(this.names[pos]))
                MainActivity.seleccion.remove(this.names[pos])
            else MainActivity.seleccion.add(this.names[pos])

            MainActivity.actionMode!!.title =
                "${MainActivity.seleccion.size} items seleccionados"
        }
        // Se devuelve la fila.
        return bindingFila.root
    }

    override fun getItem(position: Int) = this.names[position]

    override fun getItemId(position: Int) = position.toLong()

    override fun getCount() = this.names.size

    fun eliminarNombres(items: List<String>) {
        // Se elimina los elementos seleccionados.
        for (item in items)
            MainActivity.personas.remove(item)

        // Se notifica un cambio en la información mostrada en la lista.
        // Esto producirá la actualización de la vista.
        notifyDataSetChanged()
    }
}
```

Creadas las propiedades en la clase principal y la clase `ListViewAdapter`, en el método `onCreate()` de la clase principal, se añadirán las siguientes líneas de código. Se carga el *adapter* en el *ListView* y se crea el *listener*, ya utilizado en apartados anteriores, para cuando se pulse un elemento de la lista.

```
val adapter = ListViewAdapter(personas)
with(binding) {
    myListView.adapter = adapter
    myListView.setOnItemClickListener { _, _, position, _ ->
        Toast.makeText(applicationContext, personas[position], Toast.LENGTH_LONG).show()
    }
    ...
```

Y ahora se establecerá el modo de acción sobre el *ListView*, donde se volverán a ver métodos que ya se conocen.

```
with(binding) {
    ...
    with(myListView) {
        // Se establece la selección múltiple en la lista.
        choiceMode = ListView.CHOICE_MODE_MULTIPLE_MODAL

        setMultiChoiceModeListener(object : AbsListView.MultiChoiceModeListener {
            // Se llama cuando el usuario selecciona una opción del menú.
            override fun onActionItemClicked(mode: ActionMode?, item: MenuItem?): Boolean {
                return when (item!!.itemId) {
                    R.id.optionDelete -> {
                        Toast.makeText(
                            context, getString(R.string.txt_borrados, seleccion.size),
                            Toast.LENGTH_LONG).show()
                        adapter.eliminarNombres(seleccion)
                        mode!!.finish()
                        true
                    }
                    R.id.optionShare -> {
                        Toast.makeText(
                            context, R.string.menu_op_share,
                            Toast.LENGTH_LONG).show()
                        true
                    }
                    else -> false
                }
            }

            // Este método se invoca cuando la lista cambia a estado de selección.
            override fun onItemCheckedStateChanged(
                mode: ActionMode?,
                position: Int,
                id: Long,
                checked: Boolean
            ) {
                // En este ejemplo no se realiza ninguna acción.
            }

            // Se llama cuando se crea el modo acción, en este caso,
            // al crear setMultiChoiceModeListener().
            override fun onCreateActionMode(mode: ActionMode?, menu: Menu?): Boolean {
                menuInflater.inflate(R.menu.context_menu, menu)
                actionMode = mode
                isActionMode = true

                return true
            }
```

```
48              // Se llama cada vez que el modo acción se muestra, siempre
49              // después de onCreateActionMode().
50              override fun onPrepareActionMode(mode: ActionMode?, menu: Menu?): Boolean {
51                  return false
52              }
53
54              // Se llama cuando se sale del modo de acción.
55              override fun onDestroyActionMode(mode: ActionMode?) {
56                  // Se indica que no está activo el modo de acción.
57                  isActionMode = false
58                  // Se elimina el objeto ActionMode.
59                  actionMode = null
60                  // Se borra la selección del usuario.
61                  seleccion.clear()
62              }
63          })
64      }
65  }
```

El resultado de este código puedes verlo en la siguiente figura, salvando las distancias según versiones.

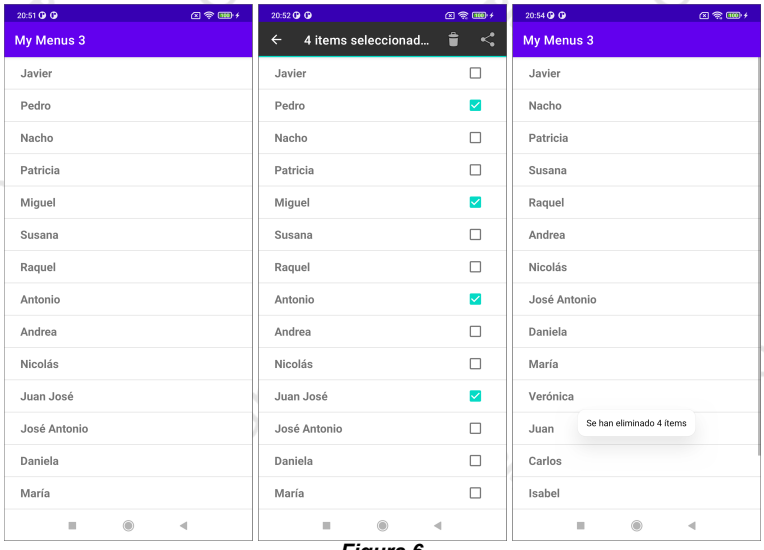

Figura 6

8.4. Menús emergentes (PopupMenu)

Este tipo de menús se encuentran anclados a una vista (*View)*. Aparecerán bajo la vista anclada si tiene espacio o sobre ella en caso contrario. A continuación, se verá con un ejemplo, comenzando un nuevo proyecto (*My Menus 4*), crea el nuevo menú (`acciones.xml`), además, se añadirá una agrupación al menú para ver como se haría.

Puedes ver a continuación como podría ser el XML del menú que actuará como menú emergente en la aplicación.

```xml
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item
        android:id="@+id/menu_save"
        android:icon="@android:drawable/ic_menu_save"
        android:title="@string/menu_op_save" />

    <!-- menu group -->
    <group android:id="@+id/group_delete">
        <item
            android:id="@+id/menu_archive"
            android:title="@string/menu_op_archive" />
        <item
            android:id="@+id/menu_delete"
            android:title="@string/menu_op_delete" />
    </group>
</menu>
```

A continuación, se creará una vista en la que se añadirá un *ImageButton* para simular la opción "más opciones" en la `activity_main.xml`. No se ha utilizado este tipo de elemento hasta el momento a lo largo del libro, pero funciona como un botón normal, solo que debe añadirse una imagen al botón.

```xml
<TextView
    android:id="@+id/textView"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    ...
    android:text="@string/txt_ejemplo"
    android:textAppearance="@style/TextAppearance.AppCompat.Large"
    ... />

<ImageView
    android:id="@+id/iv_moreOptions"
    ...
    android:background="?selectableItemBackgroundBorderless"
    android:contentDescription="@string/desc_btn"
    ...
    app:srcCompat="@drawable/more_options" />
```

En la clase `MainActivity.kt` se añadirá el *listener* para el *ImageButton* en el método `onCreate()` y el método encargado de "inflar" el menú.

```kotlin
override fun onCreate(savedInstanceState: Bundle?) {
    ...
    binding.ivMoreOptions.setOnClickListener {
        showPopupMenu(it)
```

```
    }
}

private fun showPopupMenu(v: View) {
    PopupMenu(this, v).apply {
        inflate(R.menu.acciones)
    }.show()
}
```

Llegados a este punto, ya se dispondría de un *PopupMenu* "inflado" en la vista totalmente accesible al pulsar sobre el *ImageButton*.

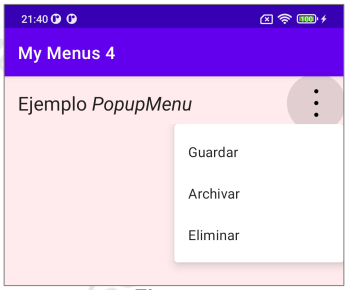

Figura 7

Ya solo faltaría capturar los eventos clic sobre las opciones del menú, existen varias formas de realizar esta operación. En este caso se optará por un nuevo método que se encargará de recoger la opción seleccionada por el usuario, de esta forma se pretende mantener el código más organizado y no sobrecargar el método encargado del "inflado".

```
// Manejador de eventos sobre el PopupMenu.
private fun managerClick(menuItem: MenuItem): Boolean {
    return when (menuItem.itemId) {
        R.id.menu_archive -> {
            showToast("Opción ${getString(R.string.menu_op_archive)}")
            true
        }
        R.id.menu_delete -> {
            showToast("Opción ${getString(R.string.menu_op_delete)}")
            true
        }
        R.id.menu_save -> {
            showToast("Opción ${getString(R.string.menu_op_save)}")
            true
        }
        else -> false
    }
}

private fun showToast(texto: CharSequence){
    Toast.makeText(this, texto, Toast.LENGTH_SHORT).show()
}
```

El método `showPopupMenu()` quedará como se muestra a continuación tras registrar el "manejador" de eventos.

```
private fun showPopupMenu(v: View) {
    PopupMenu(this, v).apply {
        inflate(R.menu.acciones)
        setOnMenuItemClickListener(::managerClick)
```

```
        }.show()
    }
```

Al igual que `setOnMenuItemClickListener()`, está el método `setOnDismissListener()` si quieres controlar el descarte del menú.

8.5. Navigation drawer

Un *Navigation drawer* es un elemento de UI que permite mostrar un menú deslizante. Este elemento se encuentra definido por Material Design[2]. La idea de este tipo de menú es facilitar el acceso a los elementos más importantes de una aplicación, siendo accesible en cualquier momento y desde cualquier punto de la aplicación.

Se recomienda su uso cuando la aplicación dispone de cinco o más elementos, así como cuando se hace uso del componente *Navigation*, que se verá más adelante.

La anatomía del *Navigation drawer* se compone de los siguientes elementos principales:

- El ***container***, como su nombre indica, contiene el contenido completo del *Navigation drawer*.
- Un ***headline***, es opcional, se utiliza para añadir información adicional, por ejemplo, sobre la aplicación, del propio usuario, de algún elemento, etc.
- El ***divider***, se una línea que se dibujará para separar secciones.
- El ***subtitle***, permite añadir una etiqueta de información, generalmente para identificar submenús.

Para ilustrar la aplicación de este componente se partirá de un nuevo proyecto, concretamente, se implementará un *Modal drawer*[3], que es la barra de menú deslizante que aparece desde la izquierda en los dispositivos móviles. El primer paso será añadir las siguientes dependencias al fichero `build.gradle (app)`.

```
// Material Design
implementation 'com.google.android.material:material:1.8.0'
// DrawerLayout
implementation 'androidx.drawerlayout:drawerlayout:1.1.1'
```

Es posible que la dependencia de *Material* ya esté añadida al proyecto. Seguidamente, se comenzará por crear los elementos que forman el *Navigation Drawer*, empezando por la cabecera, este será un *layout* normal que se creará en `res/layout`, el fichero XML se llamará `header_navigation_drawer.xml`, y su contenido será parecido al siguiente.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
```

2 *Navigation drawer* (https://material.io/components/navigation-drawer)
3 *Modal drawer* (https://material.io/components/navigation-drawer#modal-drawer)

```
4       android:layout_height="match_parent"
```

```
    android:orientation="vertical">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginStart="24dp"
        android:layout_marginTop="24dp"
        android:layout_marginEnd="24dp"
        android:text="@string/txtHeader1"
        android:textAppearance="@style/TextAppearance.AppCompat.Large"
        android:textColor="?attr/colorOnSurface"/>
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginStart="24dp"
        android:layout_marginEnd="24dp"
        android:layout_marginBottom="24dp"
        android:text="@string/txtHeader2" />
</LinearLayout>
```

Si te fijas, verás que es un *layout* simple con la información a mostrar, en este caso, no tiene asociada ninguna clase Kotlin. La creación de las opciones del menú es similar a las vistas anteriormente, pero se añade la etiqueta `group` para agrupar las opciones. El siguiente fichero XML, `navigation_drawer.xml`, se creará en `res/menu` tal y como se ha visto en otras ocasiones.

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <group
        android:id="@+id/group1"
        android:checkableBehavior="single">
        <item
            android:id="@+id/item11"
            android:checked="true"
            android:checkable="true"
            android:icon="@android:drawable/star_big_on"
            android:title="@string/txtMenu11" />
        <item
            android:id="@+id/item12"
            android:icon="@android:drawable/ic_menu_edit"
            android:checkable="true"
            android:title="@string/txtMenu12" />
        <item
            android:id="@+id/subgroup"
            android:title="@string/txtSubgrupo">
            <menu>
                <item
                    android:id="@+id/item21"
                    android:checkable="true"
                    android:icon="@android:drawable/ic_menu_mapmode"
                    android:title="@string/txtMenu21" />
```

```
                <item
                    android:id="@+id/item22"
                    android:checkable="true"
                    android:icon="@android:drawable/ic_menu_info_details"
                    android:title="@string/txtMenu22" />
            </menu>
        </item>
    </group>
</menu>
```

De este fichero destacar la propiedad `checkableBehavior` de la etiqueta `group`, generalmente se establece a ***single***, ya que funciona como un grupo y se vería la opción seleccionada en el menú, si se establece a ***all*** sería como un *checkbox*, que permite elegir varias opciones. Si se estable a ***none***, no se aplicará ningún efecto. A continuación, se modificará el tema de la aplicación para que no aparezca la *ActionBar* y hacer que las barras de estado e inferior sean transparentes, fíjate que se hará uso de *Material3*, esto deberás hacerlo en las dos versiones del tema.

```
<style name="Theme.MyNavigationDrawer" parent="Theme.Material3.DayNight.NoActionBar">
    ...
    <!-- Customize your theme here. →
    <item name="android:windowTranslucentStatus">true</item>
    <item name="android:windowTranslucentNavigation">true</item>
```

Ahora, se prepará la actividad principal para cargar el menú en la `activity_main.xml` y tener una primera impresión.

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

    <com.google.android.material.navigation.NavigationView
        android:id="@+id/myNavigationView"
        android:layout_width="wrap_content"
        android:layout_height="match_parent"
        app:headerLayout="@layout/header_navigation_drawer"
        app:menu="@menu/navigation_drawer"
        app:layout_constraintStart_toStartOf="parent" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

En este punto, si lanzas la aplicación, verás que ya tienes en pantalla el menú *Navigation*, pero de momento no se oculta ni tiene utilidad. Ahora, se terminará de montar la actividad principal, en este punto entra el *widget* `DrawerLayout`, que contendrá un `CoordinatorLayout` y el propio `NavigationView`. También se añadirá la *ActionBar* de manera manual.

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.drawerlayout.widget.DrawerLayout
    ...
```

```
    android:id="@+id/myDrawerLayout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:fitsSystemWindows="true"
    tools:context=".MainActivity"
    tools:openDrawer="start">

    <androidx.coordinatorlayout.widget.CoordinatorLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent">

        <com.google.android.material.appbar.AppBarLayout
            android:id="@+id/myAppBar"
            style="@style/Widget.MaterialComponents.AppBarLayout.PrimarySurface"
            android:layout_width="match_parent"
            android:layout_height="wrap_content">
            <androidx.appcompat.widget.Toolbar
                android:id="@+id/myToolBar"
                style="@style/Widget.MaterialComponents.Toolbar.PrimarySurface"
                android:layout_width="match_parent"
                android:layout_height="?actionBarSize" />
        </com.google.android.material.appbar.AppBarLayout>

        <!-- Screen content -->

    </androidx.coordinatorlayout.widget.CoordinatorLayout>

    <com.google.android.material.navigation.NavigationView
        android:id="@+id/myNavigationView"
        android:layout_width="wrap_content"
        android:layout_height="match_parent"
        android:layout_gravity="start"
        app:headerLayout="@layout/header_navigation_drawer"
        app:layout_constraintStart_toStartOf="parent"
        app:menu="@menu/navigation_drawer" />

</androidx.drawerlayout.widget.DrawerLayout>
```

El elemento raíz es un `DrawerLayout`, sobre el cual se dibujará el `NavigationView`, que viene a ser el propio *Modal drawer*. Dentro ya se dispone de un `CoordinatorLayout` para el resto de elementos de la vista, incluida la barra de aplicación, pero puedes utilizar el *layout* que más se adapte a tus necesidades.

Con respecto al `NavigationView`, fíjate en las propiedades `headerLayout` y `menu`, mediante las que se asocian la cabecera y las opciones del menú.

Otra propiedad que se ha utilizado en este *layout* es `fitsSystemWindows`, con esta propiedad a *true*, se consigue que el elemento no se oculte por la barra de estado del sistema, quedando esa parte transparente. Está disponible a partir de la API 21.

Debes tener en cuenta que, cuando se utiliza *NavigationView*, se suelen utilizar *fragments*, y no *activities*, ya que eso supondría cargar el *NavigationView* con su *DrawerLayout* en todas y cada una de ellas, y eso no sería práctico.

Con la actividad principal actualizada, observa el siguiente código, dónde se activará la *ActionBar*, se añadirá el botón "hamburguesa" y se vinculará el menú para que se despliegue cuando se pulse. También se desplegará al deslizar el dedo de izquierda a derecha. Observa ahora como quedaría la clase `MainActivity.kt`.

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

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

        // Se carga la barra de acción.
        setSupportActionBar(binding.myToolBar)
        // Se añade el botón "hamburgesa" a la toolbar y se vincula con el DrawerLayout.
        val toggle = ActionBarDrawerToggle(
            this,
            binding.myDrawerLayout,
            binding.myToolBar,
            R.string.txt_open,
            R.string.txt_close
        )
        binding.myDrawerLayout.addDrawerListener(toggle)
        toggle.syncState()

        // Maneja la pulsación del menú.
        binding.myNavigationView.setNavigationItemSelectedListener { menuItem ->
            Toast.makeText(applicationContext, menuItem.title, Toast.LENGTH_SHORT).show()
            menuItem.isChecked = true // Deja marcada la opción seleccionada.
            binding.myDrawerLayout.close() // Cierra el menú.
            true
        }
    }
}
```

A destacar de este código la variable `toggle`, que se utiliza para crear un objeto de tipo `ActionBarDrawerToggle`, esta permitirá asociar el *DrawerLayout* con la *ActionBar*, haciendo que aparezca botón hamburguesa.

El método `setNavigationItemSelectedListener()` utilizado en el *NavigationView* permite comprobar que opción del menú se ha pulsado y actuar en consecuencia, este método funciona exactamente igual que los métodos `onOptionsItemSelected()`, `onContextItemSelected()`, `onActionItemClicked()` o `setOnMenuItemClickListener()` vistos anteriormente para otros tipos de menú.

Para ver el efecto del menú se usará un *FrameLayout* con creación dinámica de *fragments*, visto en el capítulo anterior. Añade la siguiente vista en el *layout* de la actividad principal en el punto que aparece como `<!-- Screen content -->`, tras la vista *AppBarLayout* y el cierre del *CoordinatorLayout*.

```
<FrameLayout
    android:id="@+id/fragmentContainer"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:layout_behavior="@string/appbar_scrolling_view_behavior"
    tools:layout="@layout/fragment_page" />
```

Se creará un *Fragment Blank* con dos parámetros, título y subtítulo, que se irán modificando en función de la opción pulsada por el usuario en el menú. En el método `onCreate()` de la clase principal se añadirán las siguientes líneas para crear el *fragment* inicial.

```
// Se añade el fragment al FrameLayout.
supportFragmentManager.beginTransaction()
    .add(
        binding.fragmentContainer.id,
        crearFragment(
            "Inicio",
            "Fragment de inicio"
        )
    ).commit()
```

El método `crearFragment()` será como se muestra.

```
// Método encargado de crear los fragments.
private fun crearFragment(titulo: String, subtitulo: String): Fragment {
    val fragment = PageFragment()
    val bundle = Bundle()
    bundle.putString("titulo", titulo)
    bundle.putString("subtitulo", subtitulo)
    fragment.arguments = bundle

    return fragment
}
```

Por último, hay que añadir a cada opción del menú la creación del *fragment* correspondiente, esto se llevará a cabo en el método `setNavigationItemSelectedListener()` introducido anteriormente.

```
// Maneja la pulsación del menú.
binding.myNavigationView.setNavigationItemSelectedListener { menuItem ->
    menuItem.isChecked = true // Deja marcada la opción seleccionada.
    binding.myDrawerLayout.close() // Cierra el menú.

    val transaction = supportFragmentManager.beginTransaction().apply {
        replace(binding.fragmentContainer.id, crearFragment(
            menuItem.title.toString(),
```

```
                "Fragment de ${menuItem.title}"
            ))
            // Permite la vuelta "atrás".
            addToBackStack(null)
        }

        transaction.commit()
        true
    }
```

En este caso no es necesario evaluar cada una de las opciones, pero si fuese necesario, podría hacerse utilizando *when*.

```
when (menuItem.itemId) {
    (R.id.item11) -> {
        ...
        true
    }
    (R.id.item12) -> {
        ...
        true
    }
    ...
    else -> false
}
```

Si pruebas la aplicación, comprobarás que si pulsas el botón "atrás" del sistema con el menú abierto, la aplicación se cierra. Lo ideal sería que si el menú está abierto y se pulsa atrás, este se cierre, y si se pulsa con el menú cerrado, se cierre la aplicación, no se volverá al *fragment* anterior aún activando la opción *addToBackStack()*. Para poder hacer esto, bastará con sobrecargar el método `onBackPressed()`, pero como verás, ya se encuentra *"deprecated"*.

```
@Suppress("DEPRECATION")
override fun onBackPressed() {
    if (binding.myDrawerLayout.isOpen)
        binding.myDrawerLayout.close()
    else super.onBackPressed()
}
```

O puedes optar por la versión moderna del *BackPressed*, en la que habría que crear un *callback* al inicio de la clase.

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

    private val myBackPressed = object : OnBackPressedCallback(true) {

        override fun handleOnBackPressed() {
            if (binding.myDrawerLayout.isOpen)
                binding.myDrawerLayout.close()
            else finish()
```

```
        }

    }
    ...
```

Y después, habría que añadirlo al *dispatcher* en el método *onCreate()* de la misma clase con la siguiente instrucción.

```
onBackPressedDispatcher.addCallback(this, myBackPressed)
```

El resultado que se puede obtener con este código puedes verlo en la siguiente figura, aproximadamente, ya que dependerá de la API del dispositivo, o emulador, que estés utilizado y otros factores.

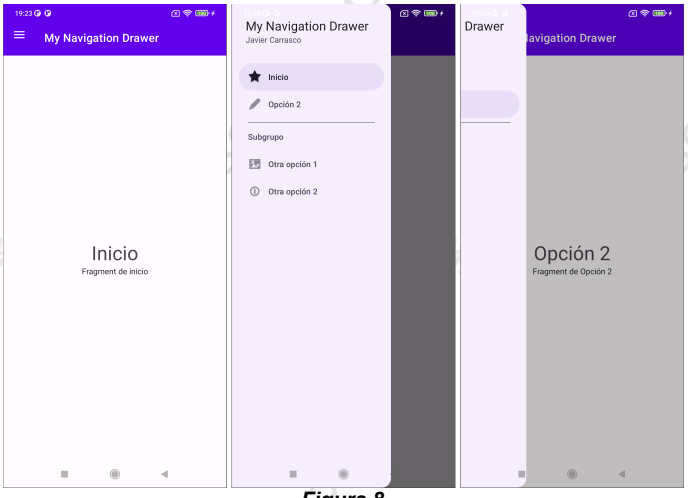

Figura 8

En este punto se ha detallado ***Modal drawer***, que suele utilizarse en dispositivos de pantalla limitada, bloqueando el uso de la aplicación mientras está abierto, quedando sobre el resto de elementos de la interfaz. Pero existen otros.

El ***Standard drawer*** permite la interacción simultánea entre la interfaz de la aplicación y el propio menú, están visibles al mismo tiempo.

Y los ***Bottom drawer***, que son menús anclados a la parte inferior de la pantalla, se suelen utilizar con barras de aplicación inferiores, abriéndose al pulsar el botón de navegación.

8.6. Preferencias

La idea de este punto es trabajar la persistencia de datos entre usos, concretamente las relacionadas con las preferencias de usuario, utilizando para ello la clase `SharedPreferences`, aunque también pueden almacenarse otro tipo de datos, como una puntuación de un videojuego por ejemplo.

Existen varias formas de utilizar las preferencias, pero desde aquí, se va a utilizar un sistema que puede resultar bastante sencillo, pensado para poder hacer uso de las preferencias desde cualquier parte de la aplicación creando una clase propia.

Partiendo de un nuevo proyecto (*My Preferences*), se creará un menú en la barra de aplicación, con la opción que de acceso a las "Preferencias". Al pulsar sobre esta opción deberá abrirse una nueva *activity* (`SettingsActivity.kt` y `activity_settings.xml`). El contenido del fichero XML para las preferencias puede parecerse al siguiente.

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

    <com.google.android.material.textfield.TextInputLayout
        android:id="@+id/textInputLayout"
        style="@style/Widget.MaterialComponents.TextInputLayout.OutlinedBox"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        ... >
        <com.google.android.material.textfield.TextInputEditText
            android:id="@+id/et_name"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:hint="@string/hint_name"
            android:imeOptions="actionDone"
            android:inputType="textPersonName"
            android:textAppearance="@style/TextAppearance.AppCompat.Medium" />
    </com.google.android.material.textfield.TextInputLayout>

    <Button
        android:id="@+id/btn_deletePrefs"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/txt_btn_deletePrefs"
        ... />
</androidx.constraintlayout.widget.ConstraintLayout>
```

A continuación, se creará al clase que se encargará de gestionar las preferencias, y donde se hará uso de la clase `SharedPreferences`. Crea una nueva clase en el proyecto llamada `Preferences.kt`.

```kotlin
class Preferences(context: Context) {
    val PREFS_NAME = "es.javiercarrasco.mypreferences"
    val SHARED_NAME = "shared_name"
    val prefs: SharedPreferences = context.getSharedPreferences(PREFS_NAME, MODE_PRIVATE)

    // Se crea la propiedad name que será persistente, además se modifica
```

```
    // su getter y setter para que almacene en SharedPreferences.
    var name: String
        get() = prefs.getString(SHARED_NAME, "").toString()
        set(value) = prefs.edit().putString(SHARED_NAME, value).apply()

    // Se eliminan las preferencias.
    fun deletePrefs() {
        prefs.edit().apply {
            remove(SHARED_NAME)
            apply()
        }
    }
}
```

Al usarse la clase `SharedPreferences`, debe hacerse uso del contexto de la aplicación para establecerlas mediante el método `getSharedPreferences()`, al cual se le pasará un identificador global (`PREFS_NAME`) para las preferencias y un modo, en este caso `MODE_PRIVATE`, sólo para uso de la aplicación. Existen otros modos de acceso, como `MODE_WORLD_READABLE` y `MODE_WORLD_WRITABLE`, que permiten acceder a las preferencias desde fuera de la aplicación, pero quedaron obsoletos a partir de la API 17 por tratarse de un tipo de acceso peligroso.

Aprovechando la potencia de Kotlin, se crea la propiedad `name` y se modifican su *getter* y *setter*. Para obtener una propiedad se utilizarán los *getters* (`getString()`, `getInt()`, etc) haciendo uso de una clave. En el ejemplo, el segundo parámetro de `getString()` es un valor por defecto, en caso de no encontrar la propiedad.

Para guardar valores se hace uso de los *setters* (`putString()`, `putInt()`, etc) mediante clave-valor. Pero, primero se deberá indicar que se están editando las preferencias (`edit()`) y terminando, para guardar, con los métodos `apply()` o `commit()`. También se ha creado un método para eliminar las propiedades, `remove()`, indicando la clave de la propiedad a eliminar.

Ahora se añadirá una nueva clase, en la que se verá un nuevo concepto que puede servir para otras operaciones. Crea la clase `SharedApp.kt`. Esta clase extenderá de la clase `Application`, lo que significa que será la primera en ejecutarse al iniciar la aplicación.

```
class SharedApp : Application() {
    companion object {
        lateinit var preferences: Preferences
            private set // El setter será privado y utiliza la implementación por defecto.
    }

    override fun onCreate() {
        super.onCreate()
        preferences = Preferences(applicationContext)
    }
}
```

Una vez creada, asegurarte de que aparece en el fichero *Manifest*, debe estar como una propiedad de *application*, si no es así, deberás añadirla.

```
<application
    android:name=".SharedApp"
    ...
```

Esta clase lo que hace es instanciar un objeto de clase `Preferences`, pasando como contexto la aplicación, con esto, ya estaría disponible el uso de `SharedPreferences` de una manera fácil. Observa ahora como se hace uso de este objeto desde la clase `SettingsActivity.kt`.

```
const val EMPTY_VALUE = ""

class SettingsActivity : AppCompatActivity() {
    private lateinit var binding: ActivitySettingsBinding
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivitySettingsBinding.inflate(layoutInflater)
        setContentView(binding.root)

        // Se comprueba si existen propiedades creadas para cargarlas.
        ConfigView()

        // Botón para eliminar las preferencias.
        binding.btnDeletePrefs.setOnClickListener {
            binding.etName.text = null
            SharedApp.preferences.deletePrefs()
            onBackPressed()
        }
    }

    // Se controla si se pulsa el botón "Atrás" de la barra de app.
    // Se controla como una opción de menú más.
    override fun onOptionsItemSelected(item: MenuItem): Boolean {
        return when (item.itemId) {
            android.R.id.home -> {
                onBackPressed()
                true
            }
            else -> super.onOptionsItemSelected(item)
        }
    }

    // Método que indica si se pulsa el botón "Atrás" del propio sistema operativo.
    override fun onBackPressed() {
        // Gracias a la clase Preferences, la asignación se encarga de editar y guardar.
        if (!binding.etName.text!!.isEmpty())
            SharedApp.preferences.name = binding.etName.text.toString().trim()
        super.onBackPressed()
    }
```

```
    // Muestra el valor de la propiedad en el EditText.
    fun showPrefs() {
        binding.etName.setText(SharedApp.preferences.name)
    }

    // Comprueba si existe la propiedad.
    fun configView() {
        if (isSavedName()) showPrefs()
    }

    fun isSavedName() = SharedApp.preferences.name != EMPTY_VALUE
}
```

La idea de la preferencia `nombre` es que ésta aparezca en la barra de título de la actividad principal, y si ésta no existe, deberá mostrarse el nombre de la aplicación.

Es importante destacar que ***onBackPressed*** se encuentra *deprecated* desde la API 33. A partir de ahora habrá que implementar `OnBackPressedCallback`, que será el nuevo sistema para detectar la pulsación o el gesto para retroceder, si está habilitado, o `onBackPressedDispatcher`, que invocará al *callback* cuando sea necesario. Observa como quedaría la clase *SettingsActivity* para esta nueva versión.

Se eliminará el método `onBackPressed()` de la primera versión, así como las llamadas al mismo, que serán sustituidas por `onBackPressedDispatcher.onBackPressed()`. En el método `onCreate()` de la clase se implementará el *callback*.

```
onBackPressedDispatcher.addCallback(this, object : OnBackPressedCallback(true) {
    override fun handleOnBackPressed() {
        Log.d("onBackPressedCallback", "handleOnBackPressed")
        if (!binding.etName.text!!.isEmpty())
            SharedApp.preferences.name = binding.etName.text.toString().trim()

        finish()
    }
})
```

Si te has fijado en el *EditText* utilizado, se ha empleado la propiedad ***imeOptions*** para indicar que debe hacerse al pulsarse la tecla *Intro* del teclado. Puedes controlar desde código esa pulsación, así como qué tecla se pulse, por ejemplo, para que vuelva automáticamente sin pulsar el botón “atrás” de la *ActionBar*. Observa como se controlaría el evento.

```
binding.etName.setOnEditorActionListener { textView, actionId, keyEvent ->
    if ((actionId == EditorInfo.IME_ACTION_DONE) ||
        (keyEvent.keyCode == KeyEvent.KEYCODE_ENTER)
    ) {
        onBackPressedDispatcher.onBackPressed()
    }

    true
}
```

Ahora, para actualizar la barra de la aplicación cada vez que se cambie la propiedad, se añadirá en la clase `MainActivity.kt` el siguiente código para el método `onResume()`.

```kotlin
// Este método se ejecuta cada vez que la activity vuelve al primer plano.
override fun onResume() {
    super.onResume()

    if (SharedApp.preferences.name != "")
        supportActionBar?.title = SharedApp.preferences.name
    else supportActionBar?.title = resources.getString(R.string.app_name)
}
```

Ahora, si se cierra la aplicación, los datos seguirán estando disponibles al volverla a abrir. El resultado de este código lo podrás ver en la figura siguiente.

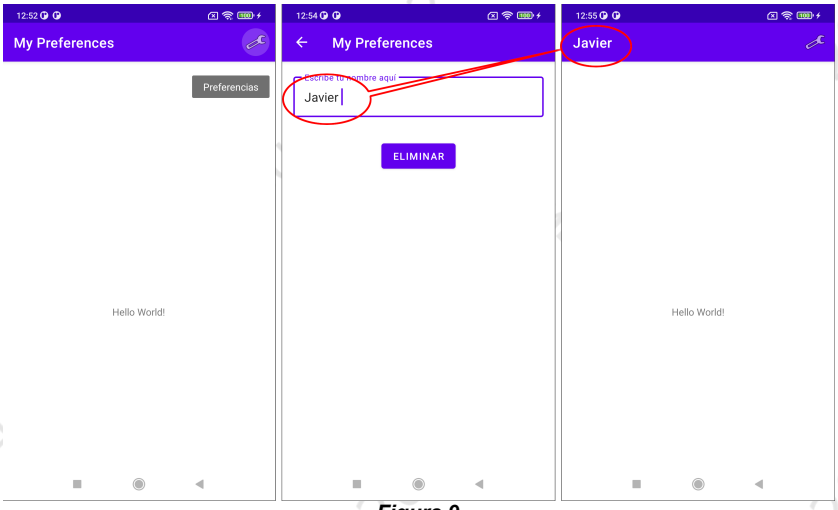

Figura 9

Cuando se hace uso de `SharedPreferences` se genera un fichero XML que almacena esas propiedades, el contenido viene a ser como se muestra a continuación.

```xml
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
    <string name="shared_name">Javier</string>
</map>
```

Este fichero se guarda en el propio dispositivo, o emulador, en mi caso, se encuentra en el directorio:

```
/data/data/es.javiercarrasco.mypreferences/shared_prefs/es.javiercarrasco.mypreferences.xml
```

Para poder explorar los archivos del emulador puedes utilizar la opción ***Tool Windows > Device File Explorer*** del menú ***View*** de Android Studio, o mediante el panel lateral derecho.