

Programación Multimedia y Dispositivos Móviles

UD 8. Menús y preferencias de usuario

Javier Carrasco

Curso 2021 / 2022



Este obra está bajo una licencia de Creative Commons Reconocimiento 4.0 Internacional.
Última actualización: septiembre de 2021.

Versión impresa en Amazon: <https://www.amazon.es/dp/B08NM4XV4Q>

Menús y preferencias de usuario

8. Menús y preferencias de usuario.....	3
8.1. Definición de un menú XML.....	3
8.2. Menú de opciones y barra de app.....	5
8.3. Menú contextual y modo de acción contextual.....	6
Menú contextual.....	7
Modo de acción contextual.....	10
Menú de acción con selección múltiple.....	12
8.4. Menús emergentes.....	17
8.5. Preferencias.....	19
8.6. <i>Navigation drawer</i>	24

8. Menús y preferencias de usuario

Los menús son elementos 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 (nivel de API 11) ya no es necesario facilitar un botón exclusivo para el menú.

A pesar de los cambios sufridos a lo 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 sus elementos. Un elemento `<menu>` será el nodo raíz del archivo y podrá tener uno o más elementos `<item>` y `<group>`.
- **<item>** Creará un *item* de menú, representa un único elemento del menú. Este elemento podrá contener un elemento `<menu>` anidado para crear un submenú.
- **<group>** Este contenedor es opcional e invisible para elementos `<item>`. Permitirá categorizar o agrupar los elementos del menú para 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**.

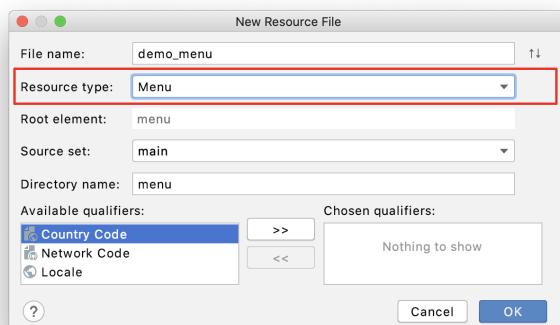


Figura 1

4 UNIDAD 8 MENÚS Y PREFERENCIAS DE USUARIO

Tras indicar el nombre y tipo de *resource* verás una nueva subcarpeta dentro de `res`, la subcarpeta menú, 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 el modo texto.

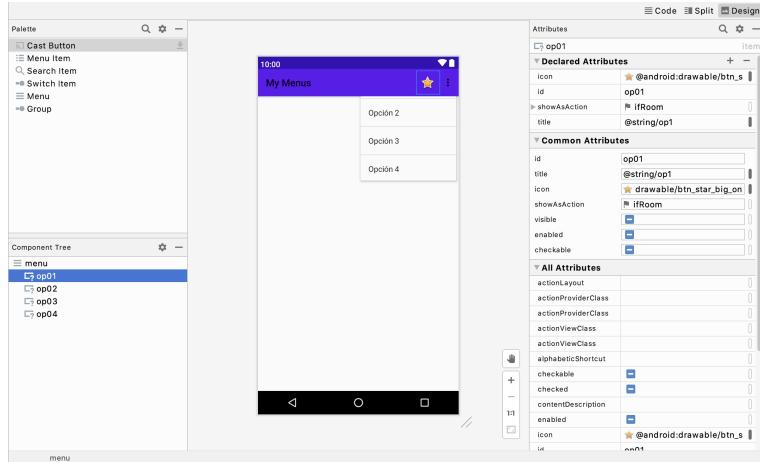


Figura 2

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <menu xmlns:android="http://schemas.android.com/apk/res/android"
3   xmlns:app="http://schemas.android.com/apk/res-auto">
4
5   <item
6     android:id="@+id/op01"
7     android:icon="@android:drawable/ic_menu_add"
8     android:title="@string/op1"
9     app:showAsAction="ifRoom" />
10
11  <item
12    android:id="@+id/op02"
13    android:title="@string/op2" />
14
15  <item
16    android:id="@+id/op03"
17    android:title="@string/op3" />
18
19  <item
20    android:id="@+id/op04"
21    android:title="@string/op4" />
22 </menu>
```

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

- **android:id** identificador del recurso.
- **android:icon** ícono a mostrar en la opción de menú.

- **android:title** título del elemento.
- **android:showAsAction** indica cuándo y cómo, deberá el elemento aparecer como un elemento de acción en la barra de app. Esta propiedad dispone de cuatro opciones, puedes ver el resultado seleccionándolas en la vista diseño.

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

```

1 <item
2   android:id="@+id/op02"
3   android:title="@string/op2">
4     <menu>
5       <item android:id="@+id/op021"
6           android:title="@string/op21" />
7       <item android:id="@+id/op022"
8           android:title="@string/op22" />
9     </menu>
10    </item>
```

8.2. Menú de opciones y barra de app

Los menús de opciones y de barra contienen las acciones, u opciones, que afectan a toda 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, se deberá sobrecargar el método `onCreateOptionsMenu()`.

```

1 /**
2 * Se infla el menú para mostrarlo en la barra de la aplicación.
3 */
4 override fun onCreateOptionsMenu(menu: Menu?): Boolean {
5     val inflate = menuInflater
6     inflate.inflate(R.menu.demo_menu, menu)
7     return true
8 }
```

El siguiente paso es sobrecargar el método `onOptionsItemSelected()`, éste permite conocer que opción del menú se ha seleccionado.

```

1 /**
2 * Método encargado de gestionar las opciones pulsadas del menú.
3 */
4 override fun onOptionsItemSelected(item: MenuItem): Boolean {
5     return when(item.itemId){
6         R.id.op01 -> {
7             Log.d("MENU", "${getString(R.string.op1)} seleccionada")
8             myToast("${getString(R.string.op1)} seleccionada")
9             true
10        }
11    }
```

6 UNIDAD 8 MENÚS Y PREFERENCIAS DE USUARIO

```
12 R.id.op02 -> { // Esta opción no haría falta, ya que abre un submenú.  
13     Log.d("MENU", "${getString(R.string.op2)} seleccionada")  
14     myToast("${getString(R.string.op2)} seleccionada")  
15     true  
16 }  
17  
18 R.id.op021 -> {  
19     Log.d("MENU", "${getString(R.string.op21)} seleccionada")  
20     myToast("${getString(R.string.op21)} seleccionada")  
21     true  
22 }  
23     ...  
24 else -> super.onOptionsItemSelected(item)  
25  
26 }
```

Con esto se obtendría un menú relativamente funcionando. Si te fijas en la imagen, verás que la *opción 1*, a la que se le ha asignado signo + como ícono, 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.

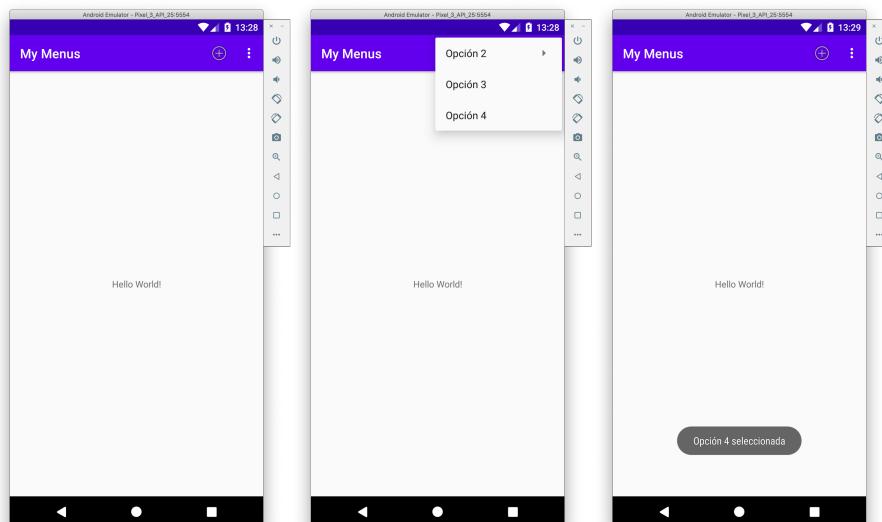


Figura 3

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.

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`.

```

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

La `activity_main.xml` únicamente contendrá un *ListView* para crear una lista de nombres. A continuación, puedes ver como quedaría la clase principal.

```

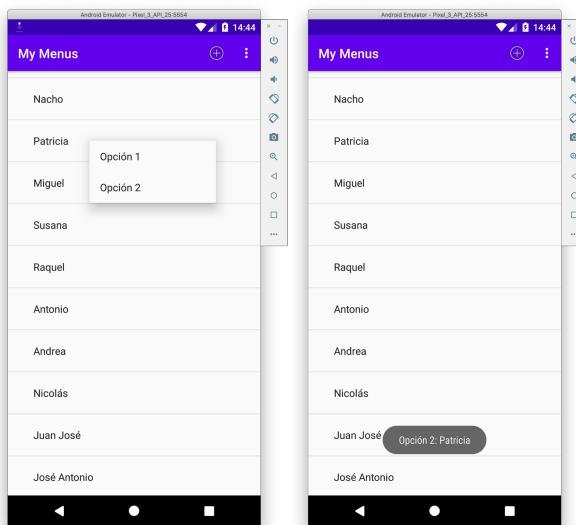
1 class MainActivity : AppCompatActivity() {
2     private lateinit var binding: ActivityMainBinding
3     private val personas =
4         arrayOf(
5             "Javier", "Pedro", "Nacho", "Patricia",
6             "Miguel", "Susana", "Raquel", "Antonio", "Andrea",
7             "Nicolás", "Juan José", "José Antonio", "Daniela",
8             "María", "Verónica"
9         )
10
11    override fun onCreate(savedInstanceState: Bundle?) {
12        super.onCreate(savedInstanceState)
13        binding = ActivityMainBinding.inflate(layoutInflater)
14        setContentView(binding.root)
15
16        // Se monta la vista para la lista de nombres.
17        val arrayAdapter: ArrayAdapter<String> =
18            ArrayAdapter(
19                this,
20                android.R.layout.simple_expandable_list_item_1,
21                personas
22            )
23
24        with(binding) {
25            // Se cargan los datos en la lista.
26            myListview.adapter = arrayAdapter
27        }
28    }
29}
```

8 UNIDAD 8 MENÚS Y PREFERENCIAS DE USUARIO

```
28 // Se registra el menú contextual al ListView.  
29 registerForContextMenu(myListView)  
30  
31 // Acción sobre el elemento de la lista pulsado.  
32 myListview.setOnItemClickListener { parent, view, position, id ->  
33     myToast(  
34         "Pulsado $id - " +  
35         "${myListView.getItemAtPosition(position)}"  
36     )  
37 }  
38 }  
39  
40 /**  
41 * Se "infla" el menú contextual con el resource.  
42 * Se ejecuta tras el registro.  
43 */  
44  
45 override fun onCreateContextMenu(  
46     menu: ContextMenu?,  
47     v: View?,  
48     menuInfo: ContextMenu.ContextMenuItem?  
49 ) {  
50     super.onCreateContextMenu(menu, v, menuInfo)  
51     val inflater = menuInflater  
52     inflater.inflate(R.menu.context_menu, menu)  
53 }  
54  
55 /**  
56 * Se comprueba la opción de menú seleccionada  
57 * y sobre que ítem se ha ejecutado.  
58 */  
59 override fun onContextItemSelected(item: MenuItem): Boolean {  
60     // Se obtiene el nombre de la persona, con  
61     // AdapterView.AdapterContextMenuInfo se obtiene la posición  
62     // sobre la que se ha hecho clic.  
63     val info = item.menuInfo as AdapterView.AdapterContextMenuInfo  
64     val posicion = info.position  
65     val nombre = personas[posicion]  
66     return when (item.itemId) {  
67         R.id.option01 -> {  
68             myToast("Opción 1: $nombre")  
69             true  
70         }  
71         R.id.option02 -> {  
72             myToast("Opción 2: $nombre")  
73             true  
74         }  
75         else -> super.onContextItemSelected(item)  
76     }  
77 }
```

```

78     private fun myToast(mensaje: String) {
79         Toast.makeText(
80             this,
81             mensaje,
82             Toast.LENGTH_SHORT
83         ).show()
84     }
85 }
```

**Figura 4**

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

```

1 <ImageView
2     android:id="@+id/imageView"
3     android:layout_width="wrap_content"
4     android:layout_height="wrap_content"
5     app:layout_constraintStart_toStartOf="parent"
6     app:layout_constraintTop_toTopOf="parent"
7     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.

```

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

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 aplicación.

Se partirá de un proyecto con una imagen 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 ícono a cada opción, para que quede bonito.

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <menu xmlns:android="http://schemas.android.com/apk/res/android">
3      <item
4          android:id="@+id/option01"
5          android:icon="@android:drawable/ic_menu_delete"
6          android:title="@string/menu_op01" />
7      <item
8          android:id="@+id/option02"
9          android:icon="@android:drawable/ic_menu_edit"
10         android:title="@string/menu_op02" />
11  </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, según puedes ver a continuación.

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

A continuación, se deberá 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.

```

1  /**
2  * Modo de acción contextual.
3  */
4  private val actionModeCallback = object : ActionMode.Callback {
5      // Método llamado al seleccionar una opción del menú.
6      override fun onActionItemClicked(mode: ActionMode?, item: MenuItem?) {
7          : Boolean {
8              return when (item!!.itemId) {
9                  R.id.option01 -> {
10                      Toast.makeText(
11                          applicationContext,
12                          R.string.menu_op01,
13                          Toast.LENGTH_LONG
14                      ).show()
15                      return true
16                  }
17              }
18          }
19      }
20  }
```

1 `ActionMode` (<https://developer.android.com/reference/android/view/ActionMode.html>)

```

17         R.id.option02 -> {
18             Toast.makeText(
19                 applicationContext,
20                 R.string.menu_op02,
21                 Toast.LENGTH_LONG
22             ).show()
23             return true
24         }
25     else -> false
26     }
27 }
28
29 // Llamado cuando al crear el modo acción a través de startActionMode().
30 override fun onCreateActionMode(mode: ActionMode?, menu: Menu?): Boolean {
31     val inflater = menuInflater
32     inflater.inflate(R.menu.context_menu, menu)
33     return true
34 }
35
36 // Se llama cada vez que el modo acción se muestra,
37 // después de onCreateActionMode().
38 override fun onPrepareActionMode(mode: ActionMode?, menu: Menu?): Boolean {
39     return false
40 }
41
42 // Se llama cuando el usuario sale del modo de acción.
43 override fun onDestroyActionMode(mode: ActionMode?) {
44     actionMode = null
45 }
46 }
```

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 el elemento en cuestión.

```

1 binding.imageView.setOnLongClickListener {
2     when (actionMode) {
3         null -> {
4             // Se lanza el ActionMode.
5             actionMode = it.startActionMode(actionModeCallback)
6             it.isSelected = true
7             true
8         }
9         else -> false
10    }
11 }
```

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

12 UNIDAD 8 MENÚS Y PREFERENCIAS DE USUARIO

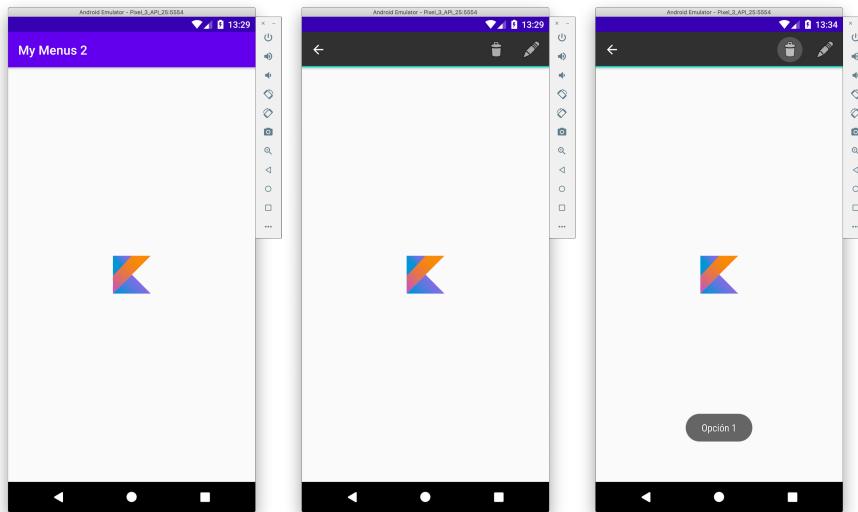


Figura 5

Menú de acción con selección múltiple

En el siguiente ejemplo, se verá como implementar el modo de acción sobre un *ListView*, creando un *BaseAdapter()* y permitiendo la **selección múltiple**.

Partiendo de un proyecto nuevo, crea un nuevo menú, `context_menu.xml`, muy parecido al visto en el ejemplo anterior.

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <menu xmlns:android="http://schemas.android.com/apk/res/android">
3     <item
4         android:id="@+id/optionDelete"
5         android:icon="@android:drawable/ic_menu_delete"
6         android:title="@string/menu_op_delete" />
7     <item
8         android:id="@+id/optionShare"
9         android:icon="@android:drawable/ic_menu_share"
10        android:title="@string/menu_op_share" />
11 </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.

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
3     xmlns:tools="http://schemas.android.com/tools"
4     android:id="@+id/myRelativeLayout"
5     android:layout_width="match_parent"
```

```

6      android:layout_height="50dp"
7      android:orientation="vertical">
8
9      <TextView
10         android:id="@+id/contact_name"
11         android:layout_width="wrap_content"
12         android:layout_height="match_parent"
13         android:layout_alignParentStart="true"
14         android:layout_marginStart="25dp"
15         android:gravity="center"
16         android:textSize="18sp"
17         android:textStyle="bold"
18         tools:text="nombre" />
19
20     <CheckBox
21         android:id="@+id/checkBox"
22         android:layout_width="wrap_content"
23         android:layout_height="wrap_content"
24         android:layout_alignParentEnd="true"
25         android:layout_centerVertical="true"
26         android:layout_marginStart="26dp"
27         android:layout_marginEnd="25dp"
28         android:visibility="gone" />
29 </RelativeLayout>

```

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.

```

1  class ListViewAdapter(
2     context: Context,
3     var names: List<String>
4 ) : BaseAdapter() {
5
6     private val inflater: LayoutInflator =
7         context.getSystemService(
8             Context.LAYOUT_INFLATER_SERVICE
9         ) as LayoutInflator
10
11    override fun getView(
12        position: Int,
13        convertView: View?, parent: ViewGroup?
14    ): View {
15        val viewFila = inflater.inflate(R.layout.item_layout, parent, false)
16
17        // Se asigna el nombre al TextView
18        val nombre = viewFila.findViewById(R.id.person_name) as TextView
19        nombre.text = this.getItem(position)
20
21        // Se asigna como etiqueta del checkBox la posición en la que

```

14 UNIDAD 8 MENÚS Y PREFERENCIAS DE USUARIO

```
22 // se encuentra.
23 val check = viewFila.findViewById(R.id.checkBox) as CheckBox
24 check.tag = position
25
26 if (MainActivity.isActionMode) {
27     check.visibility = View.VISIBLE
28 } else {
29     check.visibility = View.GONE
30 }
31
32 // Se controla la selección del usuario mediante la lista selección.
33 check.setOnCheckedChangeListener { compoundButton, _ ->
34     val position = compoundButton.tag as Int
35     Log.d("CHECKBOX", position.toString())
36
37 // Se añade o elimina de la lista la selección.
38 if (MainActivity.seleccion.contains(this.names[position])) {
39     MainActivity.seleccion.remove(this.names[position])
40 } else {
41     MainActivity.seleccion.add(this.names[position])
42 }
43
44 MainActivity.actionMode!!.title =
45     "${MainActivity.seleccion.size} items seleccionados"
46 }
47
48 // Se devuelve la fila.
49 return viewFila
50 }
51
52 override fun getItem(position: Int): String {
53     return this.names[position]
54 }
55
56 override fun getItemId(position: Int): Long {
57     return position.toLong()
58 }
59
60 override fun getCount(): Int {
61     return this.names.size
62 }
63
64 fun eliminarNombres(items: List<String>) {
65     // Se eliminan los elementos seleccionados.
66     for (item in items) {
67         MainActivity.personas.remove(item)
68     }
69
70     // Se notifica un cambio en la información mostrada en la lista.
71     // Esto producirá la actualización de la vista.
```

```

72     }
73 }
74 }
```

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

```

1 companion object {
2     val personas: MutableList<String> =
3         mutableListOf(
4             "Javier", "Pedro", "Nacho", "Patricia", "Miguel",
5             "Susana", "Raquel", "Antonio", "Andrea", "Nicolás",
6             "Juan José", "José Antonio", "Daniela", "María",
7             "Verónica", "Juan", "Carlos", "Isabel", "Óscar", "Víctor"
8         )
9     var actionMode: ActionMode? = null
10    var isActionMode: Boolean = false
11    var seleccion: MutableList<String> = ArrayList()
12 }
```

Una vez creadas las propiedades de la clase principal, en el método `onCreate()` de la clase, 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.

```

1 val adapter = ListViewAdapter(this, personas)
2 with(binding) {
3     myListview.adapter = adapter
4     myListview.setOnItemClickListener { _, _, position, _ ->
5         Toast.makeText(
6             applicationContext,
7             personas[position],
8             Toast.LENGTH_LONG
9         ).show()
10    }
11 }
```

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

```

1 with(binding) {
2     ...
3     with(myListview) {
4         // Se establece la selección múltiple en la lista.
5         choiceMode = ListView.CHOICE_MODE_MULTIPLE_MODAL
6
7         setMultiChoiceModeListener(object :AbsListView.MultiChoiceModeListener {
```

16 UNIDAD 8 MENÚS Y PREFERENCIAS DE USUARIO

```
8 // Se llama cuando el usuario selecciona una opción del menú.
9 override fun onActionItemClicked(mode: ActionMode?, item: MenuItem?) {
10    : Boolean {
11        return when (item!!.itemId) {
12            R.id.optionDelete -> {
13                Toast.makeText(
14                    context,
15                    R.string.menu_op_delete,
16                    Toast.LENGTH_LONG).show()
17                adapter.eliminarNombres(selección)
18                mode!!.finish()
19                return true
20            }
21            R.id.optionShare -> {
22                Toast.makeText(
23                    context,
24                    R.string.menu_op_share,
25                    Toast.LENGTH_LONG).show()
26                return true
27            }
28            else -> false
29        }
30    }
31
32 // Este método se invoca cuando la lista cambia a estado de
33 // selección.
34 override fun onItemSelectedChanged(
35     mode: ActionMode?,
36     position: Int,
37     id: Long,
38     checked: Boolean
39 ) {} // En este ejemplo no se realiza ninguna acción.
40
41 // Se llama cuando se crea el modo acción, en este caso,
42 // al crear setMultiChoiceModeListener().
43 override fun onCreateActionMode(mode: ActionMode?, menu: Menu?) {
44    : Boolean {
45        val inflater = menuInflater
46        inflater.inflate(R.menu.context_menu, menu)
47
48        actionMode = mode
49        isActionMode = true
50
51        return true
52    }
53
54 // Se llama cada vez que el modo acción se muestra, siempre
55 // después de onCreateActionMode().
56 override fun onPrepareActionMode(mode: ActionMode?, menu: Menu?) {
57    : Boolean {
```

```

58     return false
59 }
60 // Se llama cuando se sale del modo de acción.
61 override fun onDestroyActionMode(mode: ActionMode?) {
62     // Se indica que no está activo el modo de acción.
63     isActionMode = false
64
65     // Se elimina el objeto ActionMode.
66     actionMode = null
67
68     // Se borra la selección del usuario.
69     seleccion.clear()
70 }
71 }
72 }
73 }
```

El resultado que se obtendrá puedes verlo en la siguiente figura.

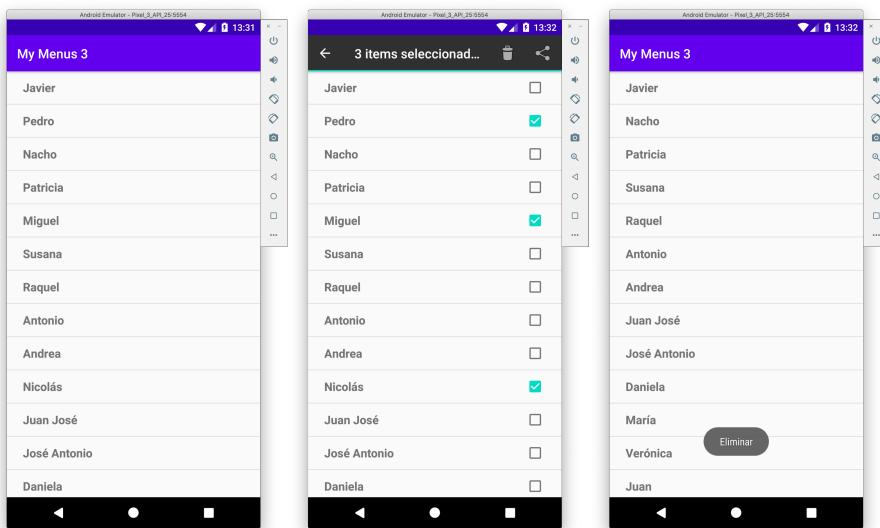


Figura 6

8.4. Menús emergentes

Este tipo de menús se encuentran anclados a una *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, crea el nuevo menú (`acciones.xml`), además, se añadirá una agrupación al menú para ver como podría hacerse.

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

18 UNIDAD 8 MENÚS Y PREFERENCIAS DE USUARIO

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <menu xmlns:android="http://schemas.android.com/apk/res/android">
3     <item
4         android:id="@+id/menu_save"
5         android:icon="@android:drawable/ic_menu_save"
6         android:title="@string/menu_op_save" />
7     <!-- menu group -->
8     <group android:id="@+id/group_delete">
9         <item
10            android:id="@+id/menu_archive"
11            android:title="@string/menu_op_archive" />
12         <item
13            android:id="@+id/menu_delete"
14            android:title="@string/menu_op_delete" />
15     </group>
16 </menu>
```

A continuación, se añadirá un *ImageButton* a la actividad `activity_main.xml`. No se ha utilizado este tipo de elemento hasta el momento a lo largo del libre pero, funciona como un botón normal, solo que debe añadirse una imagen al botón.

```
1 <ImageButton
2     android:id="@+id/imageButton"
3     android:layout_width="wrap_content"
4     android:layout_height="wrap_content"
5     android:contentDescription="@string/desc_btn"
6     android:onClick="showPopup"
7     app:layout_constraintStart_toStartOf="parent"
8     app:layout_constraintTop_toTopOf="parent"
9     app:srcCompat="@android:drawable/ic_dialog_dialer" />
```

Resaltar la propiedad `onClick`, esta permite asociar un método al evento del botón, importante mencionar que dicho método no debe devolver ningún valor y, debe tener como único parámetro la vista, en caso contrario no se podría asignar. De esta forma no es necesario crear el *listener* del botón en la clase. Ahora, en la clase `MainActivity.kt` se añade el siguiente método.

```
1 fun showPopup(v: View) {
2     PopupMenu(this, v).apply {
3         inflate(R.menu.acciones)
4         setOnMenuItemClickListener {
5             when (it.itemId) {
6                 R.id.menu_archive -> {
7                     Toast.makeText(this@MainActivity,
8                         "Opción ${getString(R.string.menu_op_archive)}",
9                         Toast.LENGTH_SHORT).show()
10                true
11            }
12             R.id.menu_delete -> {
```

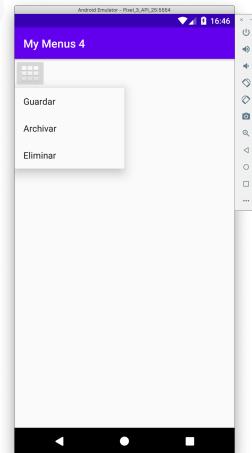


Figura 7

```

13     Toast.makeText(this@MainActivity,
14             "Opción ${getString(R.string.menu_op_delete)}",
15             Toast.LENGTH_SHORT).show()
16         true
17     }
18     R.id.menu_save -> {
19         Toast.makeText(this@MainActivity,
20                 "Opción ${getString(R.string.menu_op_save)}",
21                 Toast.LENGTH_SHORT).show()
22         true
23     }
24     else -> false
25   }
26 }
27 }.show()
28 }
```

8.5. Preferencias

La idea de este punto es trabajar la persistencia de datos, concretamente las relacionadas con las preferencias de usuario, usando la clase `SharedPreferences`, aunque también podrían almacenarse otro tipo de datos, como una puntuación en 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, creando 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 será el siguiente.

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <androidx.constraintlayout.widget.ConstraintLayout
3   xmlns:android="http://schemas.android.com/apk/res/android"
4   xmlns:app="http://schemas.android.com/apk/res-auto"
5   xmlns:tools="http://schemas.android.com/tools"
6   android:layout_width="match_parent"
7   android:layout_height="match_parent"
8   tools:context=".SettingsActivity">
9
10 <TextView
11   android:id="@+id/tv_name"
12   android:layout_width="wrap_content"
13   android:layout_height="wrap_content"
14   android:layout_marginStart="8dp"
15   android:layout_marginTop="8dp"
16   android:labelFor="@+id/et_name"
17   android:text="@string/pref_name"
18   android:textSize="24sp"
```

20 UNIDAD 8 MENÚS Y PREFERENCIAS DE USUARIO

```
19        app:layout_constraintStart_toStartOf="parent"
20        app:layout_constraintTop_toBottomOf="@+id/appBarLayout" />
21    <EditText
22        android:id="@+id/et_name"
23        android:layout_width="0dp"
24        android:layout_height="wrap_content"
25        android:layout_marginStart="16dp"
26        android:layout_marginTop="8dp"
27        android:layout_marginEnd="8dp"
28        android:ems="10"
29        android:importantForAutofill="no"
30        android:inputType="textPersonName"
31        app:layout_constraintEnd_toEndOf="parent"
32        app:layout_constraintStart_toEndOf="@+id/tv_name"
33        app:layout_constraintTop_toBottomOf="@+id/appBarLayout" />
34    <Button
35        android:id="@+id/btn_deletePrefs"
36        android:layout_width="wrap_content"
37        android:layout_height="wrap_content"
38        android:layout_marginStart="158dp"
39        android:layout_marginTop="24dp"
40        android:layout_marginEnd="165dp"
41        android:text="@string/txt_btn_deletePrefs"
42        app:layout_constraintEnd_toEndOf="parent"
43        app:layout_constraintStart_toStartOf="parent"
44        app:layout_constraintTop_toBottomOf="@+id/et_name" />
45    </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 `Preferences.kt`.

```
1 class Preferences(context: Context) {
2     val PREFS_NAME = "es.javiercarrasco.mypreferences"
3     val SHARED_NAME = "shared_name"
4     val prefs: SharedPreferences = context.getSharedPreferences(
5         PREFS_NAME, MODE_PRIVATE
6     )
7
8     // Se crea la propiedad name que será persistente, además se modifica
9     // su getter y setter para que almacene en SharedPreferences.
10    var name: String
11        get() = prefs.getString(SHARED_NAME, "").toString()
12        set(value) = prefs.edit().putString(SHARED_NAME, value).apply()
13
14    // Se eliminan las preferencias.
15    fun deletePrefs() {
16        prefs.edit().apply {
17            remove(SHARED_NAME)
18            apply()
19        }
20    }
21}
```

```
20     }
21 }
```

Al utilizar la clase `SharedPreferences` se debe hacer uso del contexto de la aplicación para establecerlas mediante el método `getSharedPreferences()`, al cual se le pasará un identificador (`PREFS_NAME`) global 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 están obsoletos desde la API 17 por ser 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 está 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 va a extender de la clase `Application`, lo que significa que será la primera en ejecutarse al iniciar la aplicación.

```
1 class SharedApp : Application() {
2     companion object {
3         lateinit var preferences: Preferences
4     }
5
6     override fun onCreate() {
7         super.onCreate()
8         preferences = Preferences(applicationContext)
9     }
10 }
```

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

```
1 <application
2     ...
3     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`.

22 UNIDAD 8 MENÚS Y PREFERENCIAS DE USUARIO

```
1 const val EMPTY_VALUE = ""
2
3 class SettingsActivity : AppCompatActivity() {
4     private lateinit var binding: ActivitySettingsBinding
5
6     override fun onCreate(savedInstanceState: Bundle?) {
7         super.onCreate(savedInstanceState)
8         binding = ActivitySettingsBinding.inflate(layoutInflater)
9         setContentView(binding.root)
10
11         // Se comprueba si existen propiedades creadas para cargarlas.
12         configView()
13         // Botón para eliminar las preferencias.
14         binding.btnDeletePrefs.setOnClickListener {
15             SharedApp.preferences.deletePrefs()
16             onBackPressed()
17         }
18     }
19
20     // Se controla si se pulsa el botón "Atrás" de la barra de app.
21     // Se controla como una opción de menú más.
22     override fun onOptionsItemSelected(item: MenuItem): Boolean {
23         return when (item.itemId) {
24             android.R.id.home -> {
25                 onBackPressed()
26                 true
27             }
28             else -> super.onOptionsItemSelected(item)
29         }
30     }
31
32     // Método que indica si se pulsa el botón "Atrás" del
33     // propio sistema operativo.
34     override fun onBackPressed() {
35         // La clase Preferences se encarga de editar y guardar en la asignación.
36         if (!binding.etName.text.isEmpty())
37             SharedApp.preferences.name = binding.etName.text.toString()
38
39         super.onBackPressed()
40     }
41
42     // Muestra el valor de la propiedad en el EditText.
43     fun showPrefs() {
44         binding.etName.hint = SharedApp.preferences.name
45     }
46
47     // Comprueba si existe la propiedad.
48     fun configView() {
49         if (isSavedName()) showPrefs()
50     }
```

```

51     fun isSavedName(): Boolean {
52         val myName = SharedApp.preferences.name
53         return myName != EMPTY_VALUE
54     }
55 }
```

La idea de la preferencia `nombre` es que ésta aparezca en la barra de título, y si no está, que muestre el nombre de la aplicación. Para refrescar la barra de la aplicación una vez cambiada la propiedad, se añadirá en la `MainActivity.kt` el siguiente código.

```

1 // Este método se ejecuta cada vez que la activity vuelve al primer plano.
2 override fun onResume() {
3     super.onResume()
4     if (SharedApp.preferences.name != "") {
5         supportActionBar?.title = SharedApp.preferences.name
6     } else supportActionBar?.title = resources.getString(R.string.app_name)
7 }
```

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.

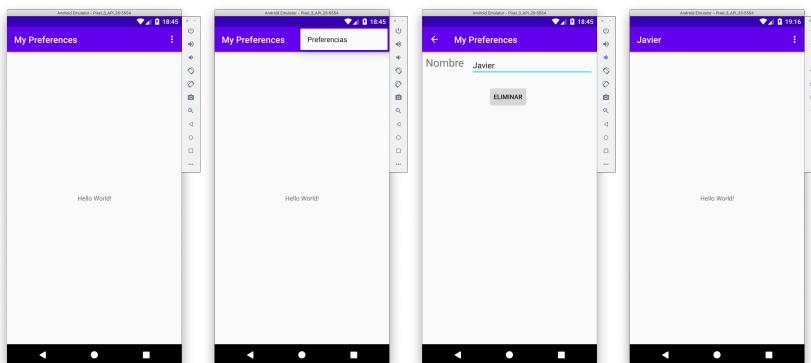


Figura 8

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.

```

1 <?xml version='1.0' encoding='utf-8' standalone='yes' ?>
2 <map>
3     <string name="shared_name">Javier</string>
4 </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.

8.6. Navigation drawer

Un *Navigation drawer* es un elemento de UI que permite mostrar un menú deslizante. Este elemento se encuentra definido por Material Design². 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 de *Navigation* (capítulo 7).

La anatomía de este elemento se compone de los siguientes elementos principales:

- El **container**, como su nombre indica, contiene el contenido completo del *Navigation drawer*.
- Un **header**, 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)`.

```
1 // Material Design
2 implementation 'com.google.android.material:material:1.4.0'
3 // DrawerLayout
4 implementation 'androidx.drawerlayout:drawerlayout:1.1.1'
```

El siguiente paso será desactivar la barra de aplicación, o *ActionBar*, del tema que viene por defecto.

```
1 <style name="Theme.MyNavigationDrawer"
2         parent="Theme.MaterialComponents.DayNight.NoActionBar">
```

Esto se hará en los dos ficheros XML para el tema de la aplicación, también deberás añadir al estilo la siguiente línea, en ambos ficheros (modo normal y modo oscuro).

```
1 <!-- Customize your theme here. -->
2 <item name="android:windowTranslucentStatus">true</item>
```

Esta propiedad a `true` en una ventana, solicita que la barra de estado sea transparente. Para ver el efecto del menú se usará un *ViewPager2* con creación dinámica de *fragments*, visto en el capítulo anterior, pero no se utilizará esta vez el *TabLayout*, ya que no se crearán pestañas.

2 *Navigation drawer* (<https://material.io/components/navigation-drawer>)

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

En primer lugar se creará los elementos que forman el *Navigation Drawer*, empezando por la cabecera, esta será un *layout* normal que se creará en `res/layout`, el fichero XML se llamará `header_navigation_drawer.xml`.

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <androidx.constraintlayout.widget.ConstraintLayout
3     xmlns:android="http://schemas.android.com/apk/res/android"
4     xmlns:app="http://schemas.android.com/apk/res-auto"
5     android:id="@+id/linearLayout"
6     android:layout_width="match_parent"
7     android:layout_height="150dp"
8     android:background="#00B0FF">
9
10    <ImageView
11        android:id="@+id/imageView"
12        android:layout_width="64dp"
13        android:layout_height="64dp"
14        android:layout_marginStart="8dp"
15        android:layout_marginTop="24dp"
16        android:layout_marginBottom="8dp"
17        android:contentDescription="@string/txtHeader3"
18        app:layout_constraintBottom_toBottomOf="parent"
19        app:layout_constraintStart_toStartOf="parent"
20        app:layout_constraintTop_toTopOf="parent"
21        app:srcCompat="@mipmap/ic_profile_foreground" />
22
23    <TextView
24        android:id="@+id/textView3"
25        android:layout_width="wrap_content"
26        android:layout_height="wrap_content"
27        android:layout_marginStart="8dp"
28        android:text="@string/txtHeader1"
29        android:textAppearance="@style/TextAppearance.AppCompat.Medium"
30        android:textStyle="bold"
31        app:layout_constraintBottom_toTopOf="@+id/textView4"
32        app:layout_constraintStart_toEndOf="@+id/imageView"
33        app:layout_constraintTop_toTopOf="@+id/imageView"
34        app:layout_constraintVertical_chainStyle="packed" />
35
36    <TextView
37        android:id="@+id/textView4"
38        android:layout_width="wrap_content"
39        android:layout_height="wrap_content"
40        android:layout_marginStart="8dp"
41        android:text="@string/txtHeader2"
42        android:textColor="@color/material_on_surface_emphasis_medium"
43        android:textSize="12sp"
44        android:textStyle="italic"
45        app:layout_constraintBottom_toBottomOf="@+id/imageView"
46        app:layout_constraintStart_toEndOf="@+id/imageView"
47        app:layout_constraintTop_toBottomOf="@+id/textView3" />
48
49    <TextView
```

26 UNIDAD 8 MENÚS Y PREFERENCIAS DE USUARIO

```
47     android:id="@+id/textView5"
48     android:layout_width="wrap_content"
49     android:layout_height="wrap_content"
50     android:layout_marginStart="8dp"
51     android:layout_marginEnd="8dp"
52     android:text="@string/txtHeader4"
53     android:textSize="12sp"
54     app:layout_constraintEnd_toEndOf="parent"
55     app:layout_constraintStart_toStartOf="parent"
56     app:layout_constraintTop_toBottomOf="@+id/imageView" />
57 </androidx.constraintlayout.widget.ConstraintLayout>
```

Si te fijas, verás que es un *layout* normal con 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 realizar agrupaciones de opciones. El siguiente fichero XML, `navigation_drawer.xml`, se creará en `res/menu` tal y como se ha visto ya.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <menu xmlns:android="http://schemas.android.com/apk/res/android">
3      <group
4          android:id="@+id/group1"
5          android:checkableBehavior="single">
6          <item
7              android:id="@+id/item11"
8              android:checked="true"
9              android:icon="@android:drawable/star_on"
10             android:title="@string/txtMenu11" />
11          <item
12              android:id="@+id/item12"
13              android:icon="@android:drawable/ic_menu_edit"
14              android:title="@string/txtMenu12" />
15          <item
16              android:id="@+id/item13"
17              android:icon="@android:drawable/ic_menu_compass"
18              android:title="@string/txtMenu13" />
19          <item
20              android:id="@+id/item14"
21              android:icon="@android:drawable/ic_menu_preferences"
22              android:title="@string/txtMenu14" />
23      </group>
24
25      <group
26          android:id="@+id/group2"
27          android:checkableBehavior="single">
28          <item
29              android:id="@+id/subgroup"
30              android:title="@string/txtSubgrupo">
31                  <menu>
```

```

32     <item
33         android:id="@+id/item21"
34         android:icon="@android:drawable/ic_menu_mapmode"
35         android:title="@string/txtMenu21" />
36     <item
37         android:id="@+id/item22"
38         android:icon="@android:drawable/ic_menu_info_details"
39         android:title="@string/txtMenu22" />
40     </menu>
41   </item>
42 </group>
43 </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 establece a **none**, no se aplicará ningún efecto.

El siguiente paso será preparar la actividad principal para cargar los elementos que se utilizarán en la vista, `activity_main.xml` quedará como sigue.

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <androidx.drawerlayout.widget.DrawerLayout
3      xmlns:android="http://schemas.android.com/apk/res/android"
4      xmlns:app="http://schemas.android.com/apk/res-auto"
5      xmlns:tools="http://schemas.android.com/tools"
6      android:id="@+id/myDrawerLayout"
7      android:layout_width="match_parent"
8      android:layout_height="match_parent"
9      android:fitsSystemWindows="true"
10     tools:openDrawer="start">
11
12     <androidx.coordinatorlayout.widget.CoordinatorLayout
13         android:layout_width="match_parent"
14         android:layout_height="match_parent"
15         tools:context=".MainActivity">
16
17         <com.google.android.material.appbar.AppBarLayout
18             style="@style/Widget.MaterialComponents.AppBarLayout.PrimarySurface"
19             android:layout_width="match_parent"
20             android:layout_height="wrap_content">
21
22             <com.google.android.material.appbar.MaterialToolbar
23                 android:id="@+id/myToolBar"
24                 style="@style/Widget.MaterialComponents.Toolbar.PrimarySurface"
25                 android:layout_width="match_parent"
26                 android:layout_height="? actionBarSize" />
27         </com.google.android.material.appbar.AppBarLayout>
28
29         <androidx.viewpager2.widget.ViewPager2
30             android:id="@+id/myViewPager2"

```

28 UNIDAD 8 MENÚS Y PREFERENCIAS DE USUARIO

```
31         android:layout_width="match_parent"
32         android:layout_height="match_parent"
33         app:layout_behavior="@string/appbar_scrolling_view_behavior" />
34     </androidx.coordinatorlayout.widget.CoordinatorLayout>
35
36     <com.google.android.material.navigation.NavigationView
37         android:id="@+id/myNavigationView"
38         android:layout_width="wrap_content"
39         android:layout_height="match_parent"
40         android:layout_gravity="start"
41         app:headerLayout="@layout/header_navigation_drawer"
42         app:menu="@menu/navigation_drawer" />
43     </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. También verás el `ViewPager2` para mostrar los *fragments*, pero puedes utilizar cualquier otro sistema para mostrarlos.

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 oculte 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. Observa ahora como quedaría la clase `MainActivity.kt`.

```
1 class MainActivity : AppCompatActivity() {
2     private lateinit var binding: ActivityMainBinding
3
4     override fun onCreate(savedInstanceState: Bundle?) {
5         super.onCreate(savedInstanceState)
6         binding = ActivityMainBinding.inflate(layoutInflater)
7         setContentView(binding.root)
8
9         // Se carga la barra de acción.
10        setSupportActionBar(binding.myToolBar)
11
12        // Se añade el botón hamburguesa a la toolbar y
13        // se vincula con el DrawerLayout.
14        val toggle = ActionBarDrawerToggle(
15            this,
16            binding.myDrawerLayout,
17            binding.myToolBar,
18            R.string.txt_open,
```

```
19         R.string.txt_close
20     )
21     binding.myDrawerLayout.addDrawerListener(toggle)
22     toggle.syncState()
23
24     // Se crea el adapter.
25     val adapter = ViewPager2Adapter(supportFragmentManager, lifecycle)
26
27     // Se crean los fragments.
28     adapter.addFragment(
29         createFragment(
30             "Pantalla inicio",
31             "Esta es la pantalla de inicio de la aplicación"
32         )
33     )
34     /* ... se crean cuatro fragments similares */
35
36     // Se asocia el adapter al ViewPager2.
37     binding.myViewPager2.adapter = adapter
38
39     // Control sobre la opción seleccionada.
40     binding.myNavigationView.setNavigationItemSelectedListener {
41         it.isChecked = true
42         when (it.itemId) {
43             R.id.item11 -> {
44                 // Se carga el fragment en el ViewPager2.
45                 binding.myViewPager2.setCurrentItem(0)
46
47                 // Se cierra el Drawer Layout.
48                 binding.myDrawerLayout.close()
49                 true
50             }
51             R.id.item12 -> {
52                 binding.myViewPager2.setCurrentItem(1)
53                 binding.myDrawerLayout.close()
54                 true
55             }
56             R.id.item13 -> {
57                 binding.myViewPager2.setCurrentItem(2)
58                 binding.myDrawerLayout.close()
59                 true
60             }
61             R.id.item14 -> {
62                 binding.myViewPager2.setCurrentItem(3)
63                 binding.myDrawerLayout.close()
64                 true
65             }
66             R.id.item21 -> {
67                 Toast.makeText(
68                     applicationContext,
```

30 UNIDAD 8 MENÚS Y PREFERENCIAS DE USUARIO

```
69                     getString(R.string.txtMenu21),
70                     Toast.LENGTH_SHORT
71                 ).show()
72
73             binding.myDrawerLayout.close()
74             true
75         }
76     R.id.item22 -> {
77         Toast.makeText(
78             applicationContext,
79             getString(R.string.txtMenu22),
80             Toast.LENGTH_SHORT
81         ).show()
82         binding.myDrawerLayout.close()
83
84         true
85     }
86     else -> false
87 }
88 }
89
90
91 // Método encargado de crear fragments.
92 private fun createFragment(name: String, texto: String): Fragment {
93     val fragment = PageFragment()
94     val bundle = Bundle()
95
96     bundle.putString("name", name)
97     bundle.putString("texto", texto)
98     fragment.arguments = bundle
99
100    return fragment
101 }
102 }
```

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

El método `setNavigationItemSelectedListener()` 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.

Por último, se ha creado el método `createFragment()` para ordenar el código durante la creación de los *fragments* que se utilizan en la aplicación.

El resultado de todo este código puedes verlo en la siguiente figura o utilizando el código que encontrarás en el repositorio del libro.

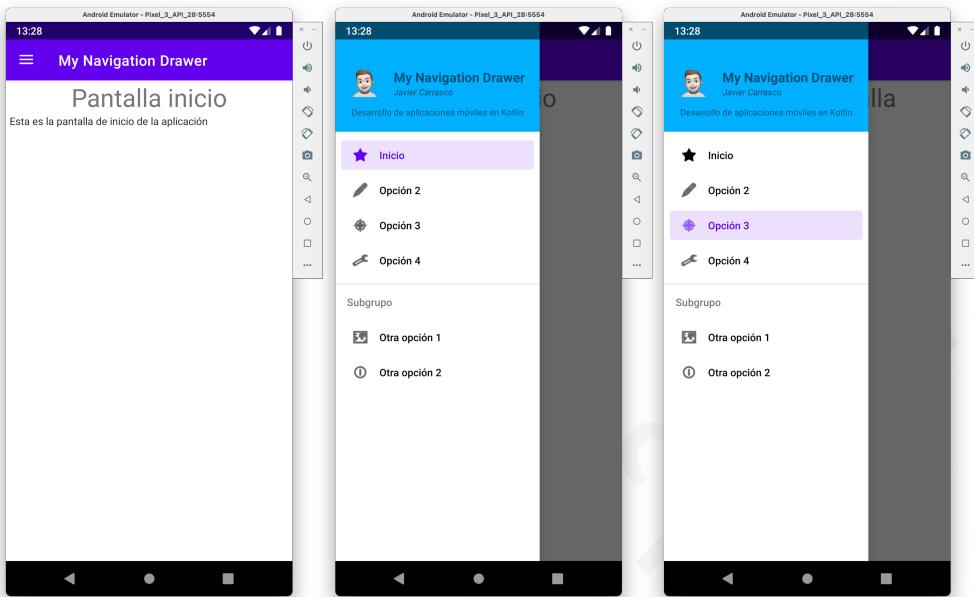


Figura 9

Si has probado la aplicación, habrás comprobado 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. Para poder hacer esto, bastará con sobre cargar el método `onBackPressed()`.

```

1 override fun onBackPressed() {
2     if(binding.myDrawerLayout.isOpen){
3         binding.myDrawerLayout.close()
4     }else super.onBackPressed()
5 }
```

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 estando 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.

Los **Bottom drawer** 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.