Programación Multimedia y Dispositivos Móviles

UD 4. Elementos básicos en Android

Javier Carrasco

Curso 2021 / 2022

Última actualización: septiembre de 2021.
Versión impresa en Amazon: https://www.amazon.es/dp/B08NM4XV4Q

Elementos básicos en Android

4. Elementos básicos en Android

Este capítulo centrará la atención en el desarrollo de aplicaciones **Android** mediante el uso de **Kotlin**. Se tratará de ver los elementos básicos que intervienen en el desarrollo de una aplicación móvil, aprendiendo sobre cada uno de ellos la forma de tratarlos. Algunos de los elementos que se verán a continuación ya se han introducido por el camino, pero no está de más volver a repasarlos.

4.1. Vinculación de vista

Desde este momento, al utilizar **Kotlin** en lugar de Java, se puede decir adiós, parcialmente, a la instrucción `findViewById`. **Kotlin** permite referenciar directamente elementos del *layout* por su nombre según el plugin que se utilice, aunque como se verá más adelante, en ocasiones es necesario hacer uso de `findViewById`.

Se comenzará con la siguiente versión **Java**, en la que se hace referencia a un botón creado en la UI para detectar la pulsación. Se utilizará este ejemplo como referencia para contrastarlo.

```
1  Button button = (Button) findViewById(R.id.button_send);
2
3  button.setOnClickListener(new View.OnClickListener() {
4      public void onClick(View v) {
5          // Do something in response to button click
6      }
7  });
```

Synthetic Binding

Kotlin facilita el acceso a la UI mediante los *imports* de *Synthetic Binding*[1], desde que se incluyó Kotlin como lenguaje nativo, no es necesario hacer nada especial para hacer uso de *Synthetic*, bastará con escribir el nombre del elemento (el id indicado en el XML) y Android Studio auto-importará a la clase la siguiente línea, en este caso sería para la `activity_main.xml`.

```
1  import kotlinx.android.synthetic.main.activity_main.*
```

Ahora, la gestión del evento clic sobre el botón que ha visto en Java, puedes observar que se limita a las siguientes líneas de código.

```
1  button_send.setOnClickListener {
2      // Do something in response to button click
3  }
```

Synthetic se encuentra *deprecated* desde la versión 1.4.20 de Kotlin, versión desde la que deja de utilizarle el plugin *kotlin-android-extensions*.

1 *Synthetic Binding* (https://kotlinlang.org/docs/tutorials/android-plugin.html)

View Binding

Esta nueva estrategia de vinculación de vista, *View Binding*[2], está disponible desde la versión 3.6. de Android Studio, y se necesita la versión 5.6.1., o superior, de *Gradle*, puedes ver tu versión en el archivo `gradle-wrapper.properties`. La idea es facilitar, todavía más, el acceso a la UI. Para su uso se deberá activar esta función en cada uno de los módulos del proyecto, por tanto, se deberá modificar el fichero `build.gradle` a nivel de `Module:app`.

```
android {
  ...
    // Android Gradle Plugin 3.6.0
    viewBinding {
        enabled = true
    }
}
```

Si se está trabajando con la versión cuatro se deberá utilizar la siguiente configuración en el *Gradle*, debido a que la versión anterior ya está *deprecated*.

```
// Android Gradle Plugin a partir de la versión 4.0
buildFeatures {
    viewBinding = true
}
```

El siguiente paso será *"inflar"* la vista que se esté utilizando, sería como establecer los atributos de los elementos. Se crea una propiedad para la clase principal, `MainActivity`, que contendrá el inflado.

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

Fíjate que se utiliza `lateinit` para poder inicializar la variable en el momento dado, en este caso, se iniciará en el método `onCreate()`. Como dato interesante, fíjate en el tipo de clase, `ActivityMainBinding`, se llama así debido a que se hace referencia a `activity_main.xml`. Si el fichero se llamase `referencias_layout.xml`, la clase sería `ReferenciasLayoutBinding`.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)

    // Estas dos líneas sustituyen a setContentView(R.layout.activity_main)
    binding = ActivityMainBinding.inflate(layoutInflater)
    setContentView(binding.root)

    binding.btHello.setOnClickListener { sayHello() }
}
```

2 *View Binding* (https://developer.android.com/topic/libraries/view-binding)

Mejoras con respecto a *findViewById*.

- Seguridad ante nulos debido a que se establecen referencias directas, esto permite que no se produzcan punteros nulos ante un ID inválido.
- Seguridad de tipos, los campos de cada clase tendrán tipos que coincidan con los tipos de las vistas al que hacen referencia en el fichero XML.

A medida que se vaya avanzando, se verá el ahorro de líneas de código que supone el uso de Kotlin con respecto a Java.

4.2. *Hardcoded text*

Cuando se empieza con el desarrollo de aplicaciones Android, uno de los primeros *warnings* con los que se suele tropezar es con el *Hardcoded text*.

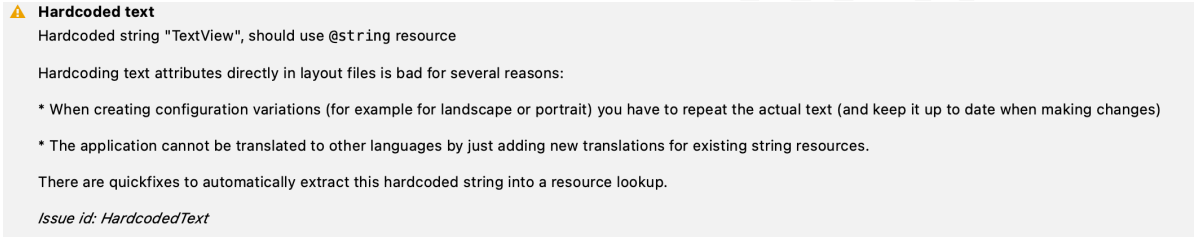

Figura 1

Esto es debido a que en Android, debe evitarse en medida de lo posible el uso de literales fijos en código o diseño. Para eliminar este tipo de avisos, basta con hacer uso del fichero `strings.xml`, donde se almacenarán todas las cadenas de texto que se vayan a emplear en la aplicación.

```
<resources>
    <string name="app_name">My Hardcoded String</string>
    <string name="miTexto">Este será mi texto a mostrar.</string>
</resources>
```

Se utilizará la constante `miTexto` en la propiedad `text` de los elementos que dispongan de ella, como etiquetas, botones, etc.

El uso de `strings.xml`, facilita la modificación de los literales en la aplicación, así como su segunda utilidad, aplicar traducciones. Para acceder al editor de traducciones[3], bastará con pulsar con el botón derecho sobre el fichero y seleccionar la opción ***Open Translations Editor***.

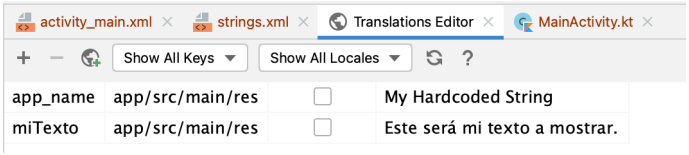

Figura 2

3 Editor de traducciones (https://developer.android.com/studio/write/translations-editor)

El funcionamiento es sencillo, se pulsará sobre la bola del mundo para seleccionar uno o varios idiomas nuevos. Aparecerán tantas columnas como idiomas, ya solo tendrás que añadir las traducciones en cada columna, esto ya no es automático.

Figura 3

Esto también producirá nuevos ficheros `strings.xml`, uno por cada una de las traducciones que se añadan a la aplicación, pasando a estar todos anidados en una nueva carpeta en `res/values/strings`.

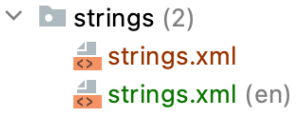

Figura 4

4.3. Botones

El botón clásico, ***button***, es uno de los elementos que más relación tiene con los usuarios, se utilizan generalmente para iniciar acciones. Estos se encontrarán en la sección ***Buttons > Button*** de la paleta en el *Layout Editor*.

Una vez colocado en el área de diseño, su XML puede tener el siguiente aspecto. Fíjate que la propiedad `android:text` hace referencia al fichero `strings.xml`.

```
<Button
    android:text="@string/myButton"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/button"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintStart_toStartOf="parent"/>
```

Recuerda el capítulo anterior, si se quiere que todos los botones tengan el mismo estilo, lo más fácil es crear un estilo y asignarlo a cada uno de ellos. Algunas de las propiedades de *buttons* que pueden resultar útiles.

- **clickable**, activa o desactiva el funcionamiento del botón, pero no aparece sombreado (como cuando está deshabilitado).
- **enabled**, activa o desactiva el botón, cuando está deshabilitado el botón aparece sombreado.
- **visibility**, permite mostrar u ocultar el botón.

El tipo de botón ***ImageButton*** funciona igual que el *button*, pero se puede añadir una imagen al botón mediante la propiedad **srcCompat** y no dispone de la propiedad *enabled*.

```
<ImageButton
    android:layout_width="wrap_content"
    android:layout_height="0dp"
    android:id="@+id/imageButton"
    android:layout_marginTop="8dp"
    android:contentDescription="@string/myButton"
    app:srcCompat="@mipmap/ic_launcher_round"
    app:layout_constraintTop_toBottomOf="@+id/button"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintEnd_toEndOf="parent"/>
```

El botón ***CheckBox*** permite marcar con un *tick* la opción que represente, a diferencia de *RadioButton* que se verá a continuación, es posible marcar más de uno. Además de las propiedades *clickable*, *enabled* y *visibility* vistas, es interesante conocer la propiedad **checked**, que permite marcar o desmarcar el *check*.

```
<CheckBox
    android:text="@string/myCheckBox"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/checkBox"
    android:layout_marginTop="8dp"
    app:layout_constraintTop_toBottomOf="@+id/imageButton"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintEnd_toEndOf="parent"/>
```

El tipo de botón ***RadioButton*** funciona exactamente igual que el tipo *CheckBox*, incluso dispone de las mismas propiedades. La diferencia radica en la agrupación que se haga de ellos mediante el elemento ***RadioGroup***, dentro de cada grupo, únicamente uno podrá estar marcado.

```
<RadioGroup
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/checkBox"
    app:layout_constraintEnd_toEndOf="parent">
    <RadioButton
        android:text="@string/myRadioButton1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/radioButton1"
        android:layout_marginTop="8dp"/>
    <RadioButton
        android:text="@string/myRadioButton2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/radioButton2"
        android:layout_marginTop="8dp"/>
</RadioGroup>
```

Los ***ToggleButton*** y los ***Switch*** tienen la misma funcionalidad, activado/desactivado, y comparten las mismas propiedades. La principal diferencia está en la personalización que permiten los *ToggleButton* (color, forma, texto, etc) frente a los *Switch*.

```
<ToggleButton
    android:text="@string/myToggleButton"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/toggleButton"
    android:layout_weight="1"
    android:layout_marginTop="8dp"
    app:layout_constraintTop_toBottomOf="@+id/radioGroup"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintEnd_toEndOf="parent"/>

<Switch
    android:text="@string/mySwitch"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/switch1"
    android:layout_marginTop="8dp"
    app:layout_constraintTop_toBottomOf="@+id/toggleButton"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintEnd_toEndOf="parent"/>
```

Se ha mostrado el XML generado el insertar los elementos en una *activity*, pero recordad que se dispone de *Layout Editor* para evitar escribir el código XML.

A continuación se verá el código **Kotlin** necesario para hacer referencia a los elementos del *layout*, y como hacer uso de las propiedades vistas. Se partirá de una *activity* vacía en la que se puede encontrar el siguiente código.

```
class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
    }
}
```

Es importante fijarse como se hace la unión, o inyección, entre la vista y el controlador, mediante la instrucción `setContentView(R.layout.activity_main)`, donde `activity_main` hace referencia a `activity_main.xml`, aunque también se puede hacer uso del método *View Binding* visto anteriormente.

Añade código **Kotlin** para que cuando el usuario pulse el *button* se marque el *checkBox*, el *radioButton1*, el *toggleButton* y el *switch1*. Para ello deberás hacer uso del evento *setOnClickListener* del *button*, pero no exclusivo, ya que lo se podrá aplicar a gran parte de los elementos con los que se trabajará en Android.

```
class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        button.setOnClickListener {
            checkBox.isChecked = true
            radioButton1.isChecked = true
            radioButton2.isChecked = false
            toggleButton.isChecked = true
            switch1.isChecked = true
        }
    }
}
```

Fíjate en el uso del método *isChecked* que posee cada uno de los elementos. Este método además, permite conocer el estado en que se encuentra el elemento. A continuación, crea un *listener* para el *imageButton* para que desactive los activados anteriormente y active el *radioButton2*.

```
class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        button.setOnClickListener {
            checkBox.isChecked = true
            radioButton1.isChecked = true
            radioButton2.isChecked = false
            toggleButton.isChecked = true
            switch1.isChecked = true
        }

        imageButton.setOnClickListener {
            if(checkBox.isChecked) {
                checkBox.isChecked = false
            }

            radioButton1.isChecked = false
            radioButton2.isChecked = true
            toggleButton.isChecked = false
            switch1.isChecked = false
        }
    }
}
```

Ordena el código utilizando una función de una manera básica, pasando como argumento un *string* indicando el nombre del elemento sobre el que se ha pulsado.

```
class MainActivity : AppCompatActivity() {
```

```
    private fun actionsButtons(name: String) {
        when (name) {
            "button" -> {
                checkBox.isChecked = true
                radioButton1.isChecked = true
                radioButton2.isChecked = false
                toggleButton.isChecked = true
                switch1.isChecked = true
            }
            else -> {
                checkBox.isChecked = false
                radioButton1.isChecked = false
                radioButton2.isChecked = true
                toggleButton.isChecked = false
                switch1.isChecked = false
            }
        }
    }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        button.setOnClickListener { actionsButtons("button") }

        imageButton.setOnClickListener { actionsButtons("imageButton") }
    }
}
```

Evidentemente, este no es el mejor código que se pueda escribir, pero servirá para ver el uso de funciones y una estructura que, cambia de nombre, `switch` en Java, en **Kotlin** es `when`, pero el funcionamiento básicamente el mismo, pero con más posibilidades que se irán viendo a medida que se vaya avanzando[4].

4.4. *Toast*

Los ***toasts*** son mensajes flotantes que se mostrarán al usuario en pantalla durante un determinado periodo de tiempo. Desaparecerá automáticamente y el usuario no interviene para nada. Ya se han utilizado para saludar en capítulos previos. Ahora se verá con algo más de detalle como crear ***toasts*** en **Kotlin**, ya que para estos no interviene el *layout*. Se puede lanzar un *toast* al realizar una pulsación sobre un botón. La instrucción para mostrar este tipo de mensajes flotantes es la siguiente.

```
Toast.makeText(
    applicationContext,
    "Este es un mensaje flotante!",
    Toast.LENGTH_SHORT
```

4 *Control flow* (https://kotlinlang.org/docs/reference/control-flow.html)

```
5   ).show()
```

También se puede formar la instrucción de esta otra manera.

```
1   Toast.makeText(
2       this,
3       "Este es un mensaje flotante!",
4       Toast.LENGTH_LONG
5   ).show()
```

Si fuese necesario, también es posible modificar la posición por defecto, cambiando la ubicación del mensaje utilizando la clase *Gravity*, utilizando el método `setGravity` del propio *Toast*.

```
1   val myToast = Toast.makeText(
2       applicationContext,
3       "Mensaje flotante con la posición modificada!",
4       Toast.LENGTH_SHORT)
5   myToast.setGravity(Gravity.CENTER,0,-300)
6   myToast.show()
```

Para crear un *toast* basta con utilizar el método *makeText* de la clase *Toast*. Este tiene tres argumentos, el contexto de la aplicación (*applicationContext* o *this*), el mensaje a mostrar, y la duración (*LENGTH_LONG* o *LENGTH_SHORT*). Finalmente, para mostrar el *toast* se deberá utilizar el método *show()*.

4.5. Etiquetas, cuadros de texto e imágenes

Se empezará por las **etiquetas** (*TextView*), son la forma más sencilla de mostrar información al usuario dentro de una *activity*. El texto de un *TextView* no puede ser editado directamente por el usuario, debe intervenir un botón o una acción que modifique su contenido.

```
1   <TextView
2       android:text="@string/myTextView"
3       android:layout_width="wrap_content"
4       android:layout_height="wrap_content"
5       android:id="@+id/textView"
6       android:layout_marginTop="64dp"
7       app:layout_constraintTop_toBottomOf="@+id/button"
8       app:layout_constraintStart_toStartOf="parent"
9       app:layout_constraintEnd_toEndOf="parent"/>
```

Algunas de las propiedades que pueden resultar útiles de las etiquetas son:

- **clickable**, activa o desactiva la opción de hacer click sobre la etiqueta, pero no aparece sombreada (como cuando está deshabilitada).
- **enabled**, activa o desactiva la etiqueta, cuando está deshabilitada aparece sombreada.
- **visibility**, permite mostrar u ocultar la etiqueta.
- **text**, permite obtener y modificar el texto de la etiqueta.

Ahora algo de código en Kotlin haciendo uso de un elemento *TextView*.

```
button.setOnClickListener {
    Toast.makeText(
        applicationContext,
        "Texto de la etiqueta: ${textView.text}",
        Toast.LENGTH_SHORT
    ).show()
}
```

Al pulsar el botón de la *activity* se mostrará un *Toast*, éste contendrá una parte de texto fija concatenada con el contenido del *TextView*, recogiéndola con su método *text* (`textView.text`).

Para solicitar información al usuario se puede hacer uso de los ***EditText***, como se puede ver en el *Layout Editor*, en la sección *Text* de la paleta, se disponen de varios modelos, *PlainText*, *Password*, *E-mail*, *Phone*, etc. A continuación, puedes ver dos ejemplos de *EditText*, uno para texto plano y otro para *E-mail*.

```
<EditText
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:ems="10"
    android:id="@+id/editText"
    android:layout_marginTop="32dp"
    android:inputType="none"
    android:hint="@string/yourTextHere"
    android:importantForAutofill="no"
    app:layout_constraintTop_toBottomOf="@+id/textView"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    tools:targetApi="o"/>

<EditText
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:inputType="textEmailAddress"
    android:ems="10"
    android:id="@+id/editText2"
    android:layout_marginTop="32dp"
    android:hint="@string/yourEmailHere"
    android:importantForAutofill="no"
    app:layout_constraintTop_toBottomOf="@+id/editText"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    tools:targetApi="o"/>
```

A continuación, algunas propiedades que pueden resultar útiles de los *EditText* son:

- **clickable**, activa o desactiva la opción de hacer click sobre el *EditText*, pero no aparece sombreado (como cuando está deshabilitado).

- **enabled**, activa o desactiva el *EditText*, cuando está deshabilitada aparece sombreado.
- **visibility**, permite mostrar u ocultar el *EditText*.
- **text**, permite obtener y modificar el texto del *EditText*.
- **hint**, permite añadir un texto sombreado para dar pistas sobre la información que se espera. Se borra automáticamente al escribir.
- **inputType**[5], permite cambiar el tipo de *EditText*, desde código deberá utilizarse la clase `InputType` para seleccionar el tipo.

Que se indique el tipo de *EditText* no significa que el texto introducido será comprobado, esto indica al sistema el tipo de teclado a mostrar más conveniente. Ahora se añadirá algo de código para ver como se puede acceder a las propiedades de los *EditText*.

```
button.setOnClickListener {
    var message: String = ""

    if (!editText.text.isEmpty()) {
        textView.text = editText.text
        message += "Texto de EditText: ${editText.text}\n"
    }

    message += "Texto de la etiqueta: ${textView.text}"
    Toast.makeText(
        applicationContext,
        message,
        Toast.LENGTH_SHORT
    ).show()
}
```

Los ***ImageView*** permitirán mostrar imágenes en las *activities*, o iconos que se necesiten añadir a los *layouts*. Se pueden mostrar, utilizando la propiedad `app:srcCompat`, imágenes que se hayan añadido al proyecto (ver capítulo 2) o desde un recurso en Internet.

```
<ImageView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/imageView"
    android:layout_marginTop="32dp"
    android:contentDescription="@string/imageText"
    app:srcCompat="@mipmap/ic_launcher"
    app:layout_constraintTop_toBottomOf="@+id/editText2"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintBottom_toBottomOf="parent"/>
```

5 *Input type* (https://developer.android.com/reference/android/text/InputType)

Propiedades que pueden resultar útiles de los *ImageView*.

- ***clickable***, activa o desactiva la opción de hacer click sobre el *ImageView*.
- ***visibility***, permite mostrar u ocultar el *ImageView*.
- ***contentDescription***, permite añadir un texto descriptivo a la imagen.
- ***srcCompat***, permite indicar la imagen que se debe cargar. Desde código, el método para ello es `setImageResource()`, y se hará uso de la clase `R.mipmap` o `R.drawable`.

```
button.setOnClickListener {
    imageView.setImageResource(R.mipmap.ic_launcher_round)
}
```

Trabajar con imágenes, sobretodo cuando se trata de cargar una desde código en tiempo de ejecución puede resultar algo tedioso, sobretodo si esas imágenes se encuentran en la nube. Por ello existen una serie de librerías externas, como *Glide*[6] o *Picasso*[7], que permiten que esta tarea sea algo más sencilla. A continuación se muestran los pasos para utilizar *Glide*.

En primer lugar se deberá añadir la dependencia en el fichero *gradle* a nivel de aplicación (*Module: app*).

```
dependencies {
    ...
    implementation 'com.github.bumptech.glide:glide:4.11.0'
}
```

Sincronizado el *Gradle* ya se podrá hacer uso de *Glide* para manejar las imágenes en los componentes *ImageView*.

```
// Se carga la imagen en el ImageView.
Glide.with(this)
    .load("https://www..../android-studio-logo.jpg")
    .override(100,100) // Ajusta el tamaño.
    .centerCrop() // Centra la imagen.
    .into(imageView) // Contenedor.
```

4.6. SnackBar

Las ***SnackBar*** se incluyen a partir de la versión 5 de Android (API 21) junto con *Material Design* y, vienen a tener la misma funcionalidad que las *Toasts*, aparecer en pantalla durante un periodo de tiempo para luego desaparecer, pero a diferencia de los *Toast* pueden contener un botón de acción en forma de texto para recibir *feedback* por parte del usuario.

Para poder utilizarlas se debe añadir una dependencia en el fichero *gradle* a nivel de aplicación (*Module: app*). También es importante mencionar, que la API mínima del proyecto para utilizar *SnackBars* debe ser la 21.

6 *Glide* (https://bumptech.github.io/glide/)
7 *Picasso* (https://square.github.io/picasso/)

```
dependencies {
    ...
    implementation 'com.google.android.material:material:1.1.0'
}
```

No se ha comentado hasta ahora, pero, si se está trabajando con una instalación limpia de **Android Studio**, concretamente con la versión 3.4.2, durante la creación de un nuevo proyecto, la opción ***Use androidx.* artifacts*** debería estar marcada por defecto, si no es así, asegúrate de marcarla para coincidir con los ejemplos mostrados.

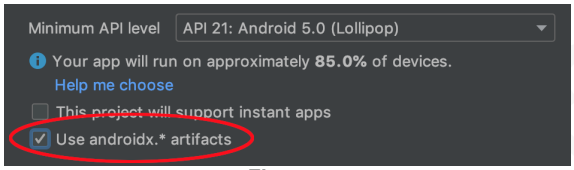

Figura 5

Pero si no estás utilizando **androidx**, la dependencia que debes añadir al *gradle* será la siguiente. En el momento de la redacción de este libro se dispone de la versión `28.0.0`.

```
dependencies {
    ...
    implementation 'com.android.support:design:28.0.0'
}
```

El siguiente código **Kotlin** mostrará un *SnackBar* al pulsar sobre el botón identificado como *button*. Es importante resaltar `root_layout`, es el identificador que recibe el *ConstraintLayout* de la *activity* (`android:id="@+id/root_layout"`), que hace de raíz, a diferencia de *Toast* se necesita conocer la vista sobre la que mostrar el *SnackBar*, no el contexto.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

    button_simple.setOnClickListener {
        showSnackSimple()
    }

    button_action.setOnClickListener {
        showSnackAction()
    }
}

private fun showSnackSimple() {
    Snackbar.make(
        root_layout,
        "Mi primer Snackbar!",
        Snackbar.LENGTH_LONG
    ).show()
}
```

```
21
22  private fun showSnackAction() {
23      // Cambiamos el color de fondo del layout
24      root_layout.setBackgroundColor(Color.YELLOW)
25
26      Snackbar.make(
27          root_layout,
28          "Mi SnackBar con acción!",
29          Snackbar.LENGTH_LONG
30      ).setAction(
31          "Deshacer" // Texto del botón
32      ) { // Acciones al pulsar el botón "Deshacer"
33          root_layout.setBackgroundColor(Color.WHITE)
34      }.show()
35  }
```

El resultado que se debe obtener mediante este código se parecerá a la imagen que se muestra en la figura 65.

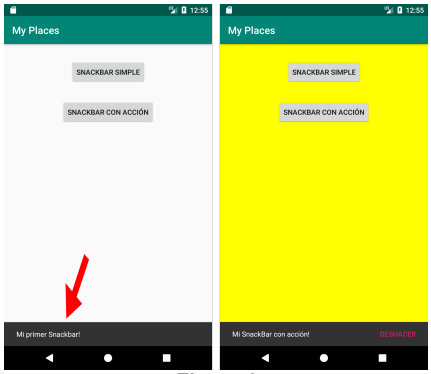

Figura 6

Ejercicios propuestos

4.6.1. Crea una aplicación Android, utilizando **Kotlin**, cuya interfaz contenga un *EditText*, un *Switch* y un *Button*. El diseño depende de ti. El funcionamiento será el siguiente, al pulsar el botón, se deberá comprobar que el *EditText* no está vacío, si lo está se mostrará un *Toast* indicando el problema. Si no está vacío, con el *Switch* en *off* se mostrará un *Toast* con el texto del *EditText*, si el *Switch* está en *on*, se mostrará un *SnackBar* simple con el texto del *EditText*.

4.7. ScrollView

Un ***ScrollView*** es un tipo de *layout* que permite añadir a las vistas desplazamiento vertical, de manera automática. Este tipo de *layout* puede resultar útil cuando la vista que se quiere mostrar excede del espacio visible del dispositivo.

El uso de *ScrollView* es una buena opción para añadir desplazamientos, pero no debe utilizarse con *ListViews*, *GridViews* o *RecyclerViews*, ya que estos se encargarán de su propio desplazamiento vertical.

El *ScrollView* dispone de la propiedad `android: fillViewport` para definir si el componente debe estirar su contenido para llenar la ventana o no.

Cuando se añade un *ScrollView* a la vista, éste lleva asociado un *LinearLayout* que contendrá el resto de elementos a mostrar.

El siguiente ejemplo muestra un *ScrollView* que contiene una serie de botones y un texto para poder recrear una vista más grande que la parte visible del dispositivo.

```
<?xml version="1.0" encoding="utf-8"?>
<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:fillViewport="true"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent">

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="vertical">

        <Button
            android:id="@+id/button"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:text="@string/myTextButton" />

        <Button
            android:id="@+id/button2"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:text="@string/myTextButton" />

       <TextView
```

```
31              android:id="@+id/textView"
32              android:layout_width="match_parent"
33              android:layout_height="wrap_content"
34              android:text="@string/myText"
35              android:textAppearance="@style/TextAppearance.AppCompat.Large" />
36
37          <Button
38              android:id="@+id/button3"
39              android:layout_width="match_parent"
40              android:layout_height="wrap_content"
41              android:text="@string/myTextButton" />
42
43          <Button
44              android:id="@+id/button4"
45              android:layout_width="match_parent"
46              android:layout_height="wrap_content"
47              android:text="@string/myTextButton" />
48
49          <Button
50              android:id="@+id/button5"
51              android:layout_width="match_parent"
52              android:layout_height="wrap_content"
53              android:text="@string/myTextButton" />
54
55      </LinearLayout>
56  </ScrollView>
```

Éste código muestra la barra de desplazamiento de manera automática, no habría que hacer nada más en cuanto a código se refiere.

Si necesitas conseguir el mismo efecto, pero en sentido horizontal, deberás utilizar el componente ***HorizontalScrollView***[8], que funciona de igual manera que *ScrollView*.

También existe ***NestedScrollView***[9], este permite anidar desplazamientos, algo que de otra manera no podría hacerse, ya que el sistema no podría decidir que debe desplazar. Este elemento añade desde la API 23, el evento *OnScrollChangeListener()*, este *callback* se emite cuando las coordenadas X e Y de la vista cambien.

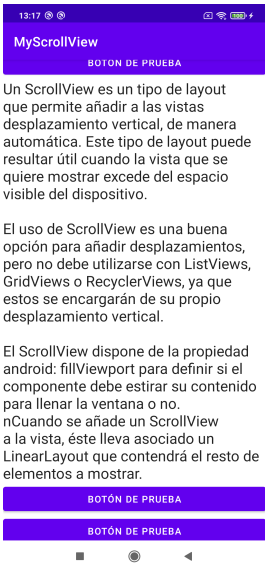

Figura 7

8 *HorizontalScrollView* (https://developer.android.com/reference/kotlin/android/widget/HorizontalScrollView)
9 *NestedScrollView* (https://developer.android.com/reference/kotlin/androidx/core/widget/NestedScrollView)

4.8. Adapter y AdapterView

Antes de seguir con elementos, o *widgets*, más complejos, se introducirán una serie de conceptos que ayudarán a entender mejor lo que está por venir.

- ***Adapter***[10], es un objeto que hará las funciones de puente entre un *AdapterView* y los datos de una vista. El *Adapter* se encargará del acceso a cada elemento y construir sus vistas. Entendiendo por vistas, aquello que mostramos al usuario.
- ***AdapterView***[11], es la vista, cuyos hijos vienen determinados por el *Adapter*. Muestra los elementos cargados en un adaptador, los elementos que se deben cargar suelen venir de una fuente de datos basada en un *array*.

Cuando se utilice la clase *AdapterView*, se dispondrá de una serie de métodos que se deben utilizar, estos dispondrán de unos *callbacks* que habrá que sobrecargar, en función del elemento y acción a gestionar. Algunos de los más comunes son:

- **AdapterView.OnItemClickListener**
  - **onItemClick(AdapterView<?> parent, View view, int position, long id)**, este *callback* será invocado cuando un elemento sea pulsado dentro de este *AdapterView*.
    - Sus parámetros son:
      - **parent**, indica el *AdapterView* donde se ha producido el clic, también se puede encontrar como `p0`.
      - **view**, es la vista dentro del *AdapterView* en la que se hizo el clic, también se puede encontrar como `p1`.
      - **position**, es la posición de la vista, del elemento, sobre el que se ha hecho el clic, también se puede encontrar como `p2`.
      - **id**, indica el identificador de la fila que contiene el elemento pulsado, también se puede encontrar como `p3`.
- **AdapterView.OnItemSelectedListener**
  - **onItemSelected(AdapterView<?> parent, View view, int position, long id)**, funciona exactamente igual que `onItemClick()`.
  - **onNothingSelected(AdapterView<?> parent)**, será invocado cuando la selección desaparezca de la vista. El parámetro `parent` contiene el contenedor que ya tiene el elemento no seleccionado.

En el siguiente punto se verá como utilizar estos conceptos con los elementos, o *widgets*, que hacen uso de ellos y se pondrá en práctica los métodos vistos.

10 *Adapter* (https://developer.android.com/reference/kotlin/android/widget/Adapter.html)
11 *AdapterView* (https://developer.android.com/reference/kotlin/android/widget/AdapterView.html)

4.9. Elementos complejos

A continuación se mostrarán una serie de elementos que hacen uso de los *Adapters* y *AdapterViews*. Para los ejemplos de cada uno de los elementos se partirá de un proyecto nuevo con una actividad vacía, API mínima 21 y marcada la opción *androidx*.

ListView

Un ***ListView***[12] es un *widget* que permite mostrar elementos de un *array* como una lista *scrollable* (desplazable) al usuario.

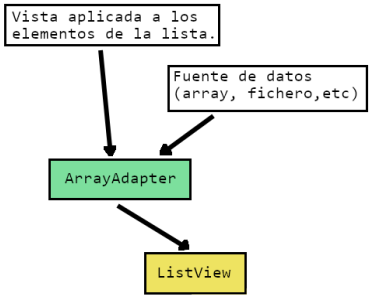

Figura 8

Como se muestra en el esquema, se necesitan dos elementos clave para montar el *Adapter*, en primer lugar la fuente de datos (un *array*, un fichero, una base de datos, etc), y en segundo, un *layout* para dar formato a los *items* de la lista, esto último se conoce como *ListView* con vista personalizada.

En primer lugar se creará la vista de los *items*, para ello se necesita un fichero XML. Se creará un nuevo `res/layout`, utilizando la opción **File > New > Android resource file**. Tal cual aparece en la siguiente imagen.

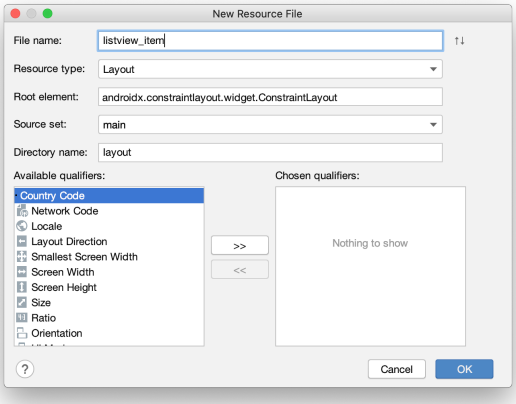

Figura 9

12 *ListView* (https://developer.android.com/reference/android/widget/ListView)

Una vez creado el fichero `listview_item.xml`, edita el fichero para que contenga el siguiente código.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Cada elemento de la lista se mostrará con el siguiente TextView -->
<TextView xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/titulo"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:padding="20dp"
    android:textSize="30sp"
    android:textStyle="bold"
    tools:text="Nombre persona" />
```

Como puedes ver, es un simple *TextView*, este será el formato que tendrá cada uno de los elementos de lista. A continuación, añade un *ListView* al fichero `activity_main.xml`.

```
<ListView
    android:id="@+id/myListView"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
```

Ya se puede crear el código fuente para establecer la relación entre ambos elementos y cargar la información. Para este ejemplo, los datos se cargarán desde un *array* estático.

```
class MainActivity : AppCompatActivity() {
    // Fuente de datos para el ListView.
    private val nombres = arrayOf(
        "Javier", "Nacho", "Patricia", "Miguel", "Susana", "Rosa", "Juan",
        "Pedro", "Asunción", "Antonio", "Lorena", "Verónica", "Paola",
        "Esteban", "Andrea", "María"
    )
    private lateinit var binding: ActivityMainBinding

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

        // Se crea el Adapter uniendo la vista y los datos.
        val adapter = ArrayAdapter(this, R.layout.listview_item, nombres)

        // Se asigna el Adapter al Listview.
        binding.myListView.adapter = adapter

```

```
        // Se utiliza un AdapterView para conocer el elemento pulsado.
        binding.myListView.onItemClickListener =
            object : AdapterView.OnItemClickListener {
                override fun onItemClick(
                    parent: AdapterView<*>?,
                    view: View?,
                    position: Int,
                    id: Long
                ) {
                    Toast.makeText(
                        applicationContext,
                        "${binding.myListView.getItemAtPosition(position)}",
                        Toast.LENGTH_SHORT
                    ).show()
                }
            }
    }
}
```

Fíjate que los pasos están bastante claros, la línea `val adapter = ArrayAdapter(this, R.layout.listview_item, nombres)` es donde se produce la unión entre vista y datos, para en la línea siguiente asignarla al *ListView* en el que se mostrará la información.

No te preocupes por la cantidad de código, parte de éste lo auto-completa el propio IDE según vayas escribiendo.

Una vez se lance la aplicación se podrá ver el resultado, y debe ser muy similar al que se muestra en la imagen que se muestra en la figura. Cada vez que se haga clic sobre un elemento de la lista deberá mostrarse un *toast* con el nombre pulsado.

Es posible que, según las circunstancias que se den, se necesiten obtener los datos desde otra fuente diferentes, o varias, por ejemplo, un *array* de datos almacenado en el fichero `strings.xml` dentro de `res/values`.

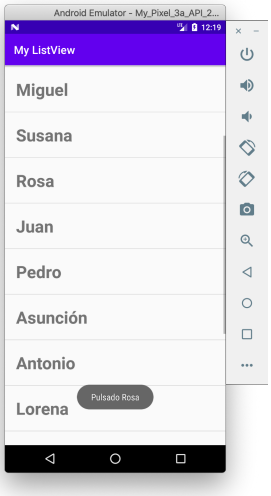

Figura 10

A continuación, puedes ver el *array* utilizado en el ejemplo.

```
<resources>
    ...
    <string-array name="array_nombres">
        <item>Javier</item>
        <item>Nacho</item>
        <item>Patricia</item>
        <item>Miguel</item>
        <item>Susana</item>
        <item>Rosa</item>
        <item>Juan</item>
        <item>Pedro</item>
```

```
        <item>Asunción</item>
        <item>Antonio</item>
        <item>Lorena</item>
        <item>Verónica</item>
        <item>Paola</item>
        <item>Esteban</item>
        <item>Andrea</item>
        <item>María</item>
    </string-array>
</resources>
```

En tal caso, únicamente se deberá modificar la unión entre vista y datos para que adopte la siguiente forma.

```
val nombres2 = resources.getStringArray(R.array.array_nombres)
// Se crea el Adapter uniendo la vista y los datos.
val adapter = ArrayAdapter(this, R.layout.listview_item, nombres2)
```

GridView

El funcionamiento básico de un ***GridView***[13] es exactamente igual al mostrado para el *ListView*, la principal diferencia entre ambos radica en que el *GridView* permite mostrar la información en formato de tabla. A continuación se muestra el código que se debe utilizar si se quiere un *GridView* simple.

Tras comenzar un nuevo proyecto, crea el fichero `gridview_item.xml` y edita el fichero para que contenga el siguiente código.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Cada elemento de la lista se mostrará con el siguiente TextView -->
<TextView xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/titulo"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:padding="20dp"
    android:textSize="20sp"
    android:textStyle="bold"
    tools:text="Nombre persona" />
```

Como puedes ver, para este ejemplo, el código es exactamente igual que el utilizado para el *ListView*. A continuación, añade un *GridView* al fichero `activity_main.xml`, donde se encontrará una ligera diferencia con respecto al ejemplo anterior.

```
<GridView
    android:id="@+id/myGridView"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
```

13 *GridView* (https://developer.android.com/reference/android/widget/GridView)

```
    android:numColumns="auto_fit"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
```

En los *GridView*, una propiedad importante es `android:numColumns`, esta permitirá ajustar automáticamente el número de columnas utilizando `auto_fit` o especificar un valor fijo si fuese necesario.

Fíjate que el código **Kotlin** que se va a utilizar es exactamente el mismo que el empleado en el anterior ejemplo, cambiando *ListView* por *GridView*.

```
class MainActivity : AppCompatActivity() {
    // Fuente de datos para el GridView.
    private val nombres = arrayOf(
        "Javier", "Nacho", "Patricia", "Miguel", "Susana", "Rosa", "Juan",
        "Pedro", "Asunción", "Antonio", "Lorena", "Verónica", "Paola",
        "Esteban", "Andrea", "María"
    )
    private lateinit var binding: ActivityMainBinding

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

        // Se crea el Adapter uniendo la vista y los datos.
        val adapter = ArrayAdapter(this, R.layout.gridview_item, nombres)

        // Se asigna el Adapter al GridView.
        binding.myGridView.adapter = adapter

        // Se utiliza un AdapterView para conocer el elemento pulsado.
        binding.myGridView.onItemClickListener =
            object : AdapterView.OnItemClickListener {
                override fun onItemClick(
                    parent: AdapterView<*>?,
                    view: View?,
                    position: Int,
                    id: Long
                ) {
                    Toast.makeText(
                        applicationContext,
                        "${binding.myGridView.getItemAtPosition(position)}",
                        Toast.LENGTH_LONG
                    ).show()
                }
            }
    }
}
```

El resultado que se obtenga diferirá con respecto al *ListView*, pero el funcionamiento será exactamente igual al que se ha visto.

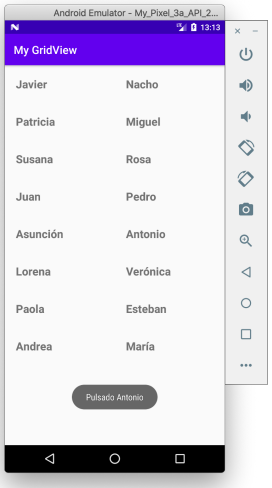

Figura 11

A continuación, se ampliará el uso de *GridView*, también valdría para el *ListView*, añadiendo algo más de información a cada uno de los elementos que forman la lista. Para ello se utilizará la clase abstracta ***BaseAdapter***, hasta ahora se había utilizado *ArrayAdapter*, que viene bien para listas o *arrays*. *ArrayAdapter* hereda de la clase *BaseAdapter*, pero al tratarse de una clase abstracta, será necesario escribir algo más de código.

Para este ejemplo, comienza por crear un nuevo proyecto, *My GridView 2*, por ejemplo, y crea un nuevo fichero llamado `gridview_item.xml`, pero esta vez, su contenido será como el que se muestra a continuación.

```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="150dp"
    android:layout_height="wrap_content"
    android:background="#eee"
    android:gravity="center"
    android:orientation="vertical"
    android:padding="15dp">

    <ImageView
        android:id="@+id/image"
        android:layout_width="150dp"
        android:layout_height="150dp"
        android:contentDescription="@string/desc_imagen"
        app:srcCompat="@mipmap/ic_launcher" />

    <TextView
        android:id="@+id/tvName"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:gravity="center"
        android:textSize="20sp"
        tools:text="Descripción" />
</LinearLayout>
```

La idea de este *layout* es conseguir algo parecido a lo que podrás ver en la imagen que se muestra en la siguiente figura.

Figura 12

El siguiente paso será crear un *GridView*, en este caso, en la `activity_main.xml`, además, deberás ajustar su anchura para que case con la que se le ha dado a la imagen.

```
<GridView
    android:id="@+id/myGridView"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:columnWidth="150dp"
    android:horizontalSpacing="15dp"
    android:numColumns="auto_fit"
    android:verticalSpacing="15dp"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
```

Ahora viene la parte de programación, lo primero que se necesitará es crear una clase para representar la información, para este ejemplo, `MyItems.kt` (**File > New > Kotlin File/Class**).

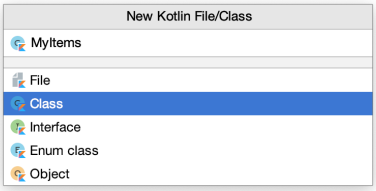

Figura 13

Convierte la clase en una *data class* utilizando el siguiente código.

```
// Clase encargada de almacenar la información de un item.
data class MyItems(val name: String, val image: Int) {
    private var n: String? = null
    private var img: Int? = null
    init {
        this.n = name
        this.img = image
    }
}
```

Recuerda el uso de `?` al declara una variable, se utiliza para indicar que una variable puede contener valor nulo. Ahora, crea otra nueva clase llamada `ItemAdapter` que extenderá de **BaseAdapter** y se encargará de “inflar” los items.

```
// Clase ItemAdapter que hereda de la clase abstracta BaseAdapter.
class ItemAdapter : BaseAdapter {
    var context: Context? = null
    var itemsList = ArrayList<MyItems>()

    // Se crea un constructor al que le pasamos el contexto
    // y la lista de elementos.
    constructor(
        context: Context,
        itemsList: ArrayList<MyItems>
    ) : super() {
        this.context = context
        this.itemsList = itemsList
    }

    // Devuelve la vista de cada elemento al adaptador.
    override fun getView(
        position: Int,
        convertView: View?,
        parent: ViewGroup?
    ): View {
        val item = this.itemsList[position]
        val inflator = context!!.getSystemService(
            Context.LAYOUT_INFLATER_SERVICE
        ) as LayoutInflater
        val binding = GridviewItemBinding.inflate(inflator)

        binding.image.setImageResource(item.image)
        binding.tvName.text = item.name

        return binding.root
    }

    override fun getItem(position: Int): Any {
        return itemsList[position]
    }

    override fun getItemId(position: Int): Long {
        return position.toLong()
    }

    override fun getCount(): Int {
        return itemsList.size
    }
}
```

En este fragmento de código aparece una nueva clase conocida como `ViewGroup`, ésta representa un objeto invisible en el que se organizarán las vistas individuales de cada elemento (*View*), las contendrá.

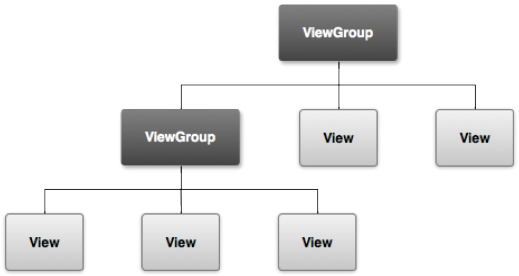

Figura 14

El método `getView()`, donde se utiliza el *LayoutInflater*, también se puede escribir así,

```
var inflator = LayoutInflater.from(context)
```

en vez de,

```
var inflator = context!!.getSystemService(
    Context.LAYOUT_INFLATER_SERVICE
) as LayoutInflater
```

¿Qué significa *"Inflar"*?, básicamente es añadir a un elemento contenedor, en este caso el *GridView* a través del adaptador, elementos de otra vista, que viene a ser `gridview_item.xml`.

Vuelve a la clase `MainActivity.kt` para terminar de escribir el código que falta.

```
class MainActivity : AppCompatActivity() {
    private lateinit var binding: ActivityMainBinding
    var adapter: ItemAdapter? = null
    var itemsList = ArrayList<MyItems>()

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

        // Se crea la fuente de datos con imágenes de muestra.
        itemsList.add(
            MyItems(
                "Estrella", R.drawable.abc_ic_star_black_48dp))
        itemsList.add(
            MyItems(
                "Botón Check", R.drawable.abc_btn_check_material))
        itemsList.add(
            MyItems(
                "1/2 estrella", R.drawable.abc_ic_star_half_black_48dp))
        itemsList.add(
            MyItems(
                "Launcher Back", R.drawable.ic_launcher_background))
        itemsList.add(
            MyItems(
```

```
                "Launcher Icon", R.drawable.ic_launcher_foreground))
        itemsList.add(
            MyItems(
                "Launcher", R.mipmap.ic_launcher))

        // Se generamos el adaptador.
        adapter = ItemAdapter(this, itemsList)

        // Asignamos el adapter
        binding.myGridView.adapter = adapter
    }
}
```

Para saber que vista ha pulsado el usuario, se puede hacer de varias formas, la primera, se podría añadir un evento `setOnClickListener()` sobre cada una de las vistas de la siguiente forma, en la misma función `getView()` de `ItemAdapter()` se añadirán las siguientes líneas.

```
// Pulsación sobre la vista.
binding.root.setOnClickListener {
    Toast.makeText(
        context,
        "${binding.tvName.text}",
        Toast.LENGTH_LONG
    ).show()
}
```

La otra opción es la que se ha estado utilizando hasta ahora, con una pequeña variación, ya que la vista individual de cada elemento ahora es más compleja. Para este nuevo método se necesitará añadir lo siguiente `import androidx.core.view.get` o pulsar `Alt+Intro` para añadir las dependencias. En este caso se implementará la funcionalidad en el método `onStart()`.

```
override fun onStart() {
    super.onStart()
    binding.myGridView.onItemClickListener =
        object : AdapterView.OnItemClickListener {
            override fun onItemClick(
                parent: AdapterView<*>?,
                view: View?,
                position: Int,
                id: Long
            ) {
                val bindingItem = GridviewItemBinding.bind(view!!)
                Toast.makeText(
                    applicationContext,
                    "Pulsado ${bindingItem.tvName.text}",
                    Toast.LENGTH_SHORT
                ).show()
            }
        }
}
```

Fíjate como se hace la referencia, en este caso, al texto del *TextView*. También se pueden utilizar *lambdas* para generar el mismo código, algo de uso muy común en **Kotlin**.

```
override fun onStart() {
    super.onStart()
    binding.myGridView.onItemClickListener =
        AdapterView.OnItemClickListener { parent, view, position, id ->
            val bindingItem = GridviewItemBinding.bind(view!!)

            Toast.makeText(
                applicationContext,
                "Pulsado ${bindingItem.tvName.text}",
                Toast.LENGTH_SHORT
            ).show()
        }
}
```

El resultado que se debería obtener tras estas líneas de código debe ser algo similar a lo que se puede ver en la figura.

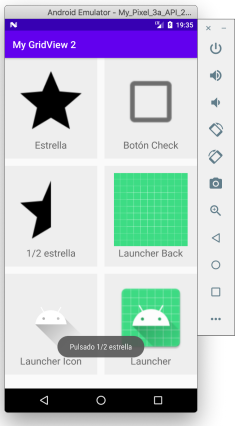

Figura 15

La principal diferencia entre estos dos sistemas para controlar que *item* se ha pulsado, ya bien sea un *ListView* o un *GridView*, es la siguiente; el primero, está implementado dentro del método `getView()`, únicamente funcionará si se hace clic sobre la propia imagen, el segundo, se encuentra implementado en el método `onStart()`, y funcionará si se pulsa sobre el *item* completo, no solo sobre la imagen.

Spinner

Los ***spinner***[14] son elementos desplegables que permiten seleccionar un elemento de una lista múltiple. Tiene una menor personalización que los anteriores, pero será necesario utilizar la clase *Adapter*.

Se partirá del *string-array* de nombres utilizado para el *ListView* de ejemplo (se podría añadir un primer elemento al *string-array* del tipo "Selecciona un elemento") y, se añade al fichero `activity_main.xml` un *spinner*.

```
<Spinner
    android:id="@+id/mySpinner"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toEndOf="@+id/tv_selecciona"
    app:layout_constraintTop_toTopOf="parent" />
```

Una vez esté añadido, ve a la clase `MainActivity.kt` para añadir los elementos a la lista y el código necesario para saber que elemento es seleccionado por el usuario.

14 *Spinner* (https://developer.android.com/guide/topics/ui/controls/spinner)

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

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

        // Se crea el Adapter, uniendo la fuente de datos con una vista
        // por defecto para el spinner de Android.
        val adapter = ArrayAdapter.createFromResource(
                this,
                R.array.array_nombres,
                android.R.layout.simple_spinner_item
        )

        // Se especifica el diseño que debe utilizarse para mostrar la lista.
        adapter.setDropDownViewResource(
                android.R.layout.simple_spinner_dropdown_item
        )

        // Se carga el Adapter en el Spinner.
        binding.mySpinner.adapter = adapter

        binding.mySpinner.onItemSelectedListener =
            object : AdapterView.OnItemSelectedListener {
                override fun onNothingSelected(p0: AdapterView<*>?) {}

                override fun onItemSelected(
                        p0: AdapterView<*>?, p1: View?,
                        p2: Int, p3: Long
                ) {
                    if (p2 == 0)
                        Toast.makeText(
                                applicationContext,
                                "Ningún elemento seleccionado.",
                                Toast.LENGTH_LONG
                        ).show()
                    else Toast.makeText(
                            applicationContext,
                            "${binding.mySpinner.getItemAtPosition(p2)}!",
                            Toast.LENGTH_LONG
                    ).show()

                    Log.d("MySpinner",
                            "${binding.mySpinner.getItemAtPosition(p2)}!")
                }
            }
    }
}
```

El resultado que se debería obtener con todo lo visto hasta aquí sería algo similar a lo que se puede ver en la figura mostrada.

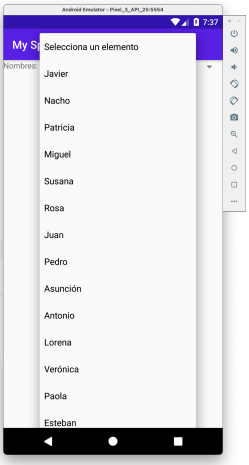

Figura 16

También se puede utilizar la clase ***Log***[15] para mostrar información en la consola de Android Studio en lugar de utilizar siempre *toasts* o *snack bars* por ejemplo.

Gracias a esta clase, es posible ver información de nuestra aplicación mediante el *Logcat* del IDE, lo que permite realizar trazas del código. Puedes utilizar el método `d` para *debug*, `e` para mostrar errores, `w` para los *warnings*, etc.

```
1   Log.d("MySpinner",
2           "Selected ${mySpinner.getItemAtPosition(p2)}!"
3   )
```

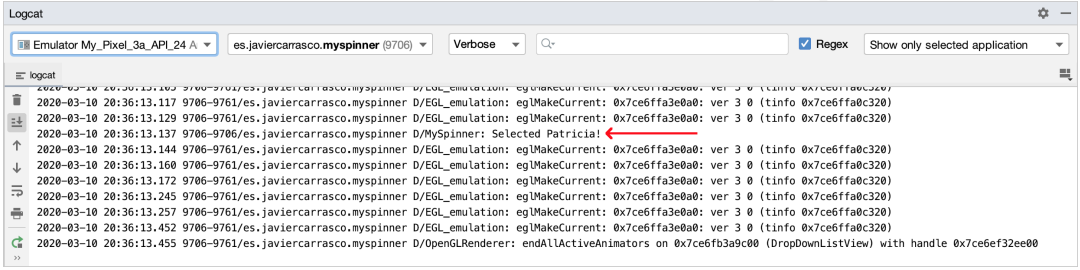

Figura 17

Ejercicios propuestos

4.9.1. Crea una aplicación **Android+Kotlin** llamada ***My Candies***, con API mínima 21. Crea una lista con los siguientes datos: caramelos, chupa-chups, corazones, pirufrutas, rosquillas, frutimelos, ositos, habichuelas y lacasitos. La fuente de datos a tu elección. Cuando pulsemos cualquier elemento de la lista, lanza un *toast* con el nombre y una línea en el *log*.

4.9.2. Crea una nueva versión del ejercicio anterior, ***My Candies 2***, con API mínima 21. Utilizando los mismos datos, deberás mostrarlos utilizando un *GridView*, añadiendo una imagen para cada elemento, puedes utilizar imágenes que encuentres por Internet para cargarlas en la *app*. Cuando pulsemos cualquier elemento, se lanzará un *toast* con el nombre y una línea en el *log*.

15 *Log* (https://developer.android.com/reference/android/util/Log)

> **NOTA:** Durante el desarrollo de estos ejercicios, es posible que te encuentres con el error `ArrayIndexOutOfBoundsException`, esto es debido a que no se actualiza la vista al desplazarla. No te preocupes, esto no ocurrirá con `RecyclerView`.

4.10. RecyclerView y CardView

Los *widgets* ***RecyclerView***[16] *y* ***CardView***[17] aparecen en **Android** a partir de la versión *Lollipop* (**API 21** Android 5.0) con la intención de hacer la vida del programador algo más fácil. Debes saber, que haciendo uso de estos dos elementos se cumplirá con la idea de *Material Design* de Google.

Empieza añadiendo un **RecyclerView** a nuestra vista principal, fíjate que en el *Layout Editor* aparece con un símbolo de descarga junto al componente, esto indica que se añadirá la librería `androidx.recyclerview:recyclerview:+` al *build.gradle (Module: app)*, se deberá indicar que sí. Se obtendrá algo así.

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

Ahora se creará la vista individual para cada uno de los elementos, en este caso se mostrará la imagen de un animal, su nombre común y su nombre científico. Para tal caso se creará un nuevo *layout* llamado `item_animal_list.xml`.

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

    <ImageView
            android:layout_width="150dp"
            android:layout_height="150dp"
            android:id="@+id/iv_animalImage"
            android:contentDescription="@string/desc_image"
            app:srcCompat="@mipmap/ic_launcher_round"/>

    <TextView
            android:text="@string/name_text"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:id="@+id/tv_nameAnimal"
            android:layout_alignParentTop="true"
            android:layout_marginTop="10dp"
```

16 RecyclerView (https://developer.android.com/reference/androidx/recyclerview/widget/RecyclerView)
17 CardView (https://developer.android.com/reference/kotlin/androidx/cardview/widget/CardView)

```
            android:layout_marginStart="5dp"
            android:layout_toEndOf="@+id/iv_animalImage"
            android:textSize="30sp"/>

    <TextView
            android:text="@string/latin_text"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:id="@+id/tv_latinName"
            android:layout_alignBottom="@+id/iv_animalImage"
            android:layout_alignParentEnd="true"
            android:layout_toEndOf="@+id/iv_animalImage"
            android:layout_marginStart="5dp"
            android:layout_marginTop="10dp"
            android:layout_below="@+id/tv_nameAnimal"
            android:textSize="18sp"/>

</RelativeLayout>
```

Figura 18

Ahora se creará el modelo de datos que contendrá la información sobre los animales, para eso se creará la clase `MyAnimals.kt` con el siguiente código.

```
data class MyAnimals(val name: String, val latin: String, val image: Int) {
    var animalName: String? = null
    var latinName: String? = null
    var imageAnimal: Int? = null

    init {
        this.animalName = name
        this.latinName = latin
        this.imageAnimal = image
    }
}
```

Ahora se creará el *adapter* para el *RecyclerView*, hasta ahora se ha estado colocando todo el código en la clase *MainActivity.kt*, pero hay que empezar a modularizar, crea una nueva clase llamada `RecyclerAdapter.kt`, que heredará de `RecyclerView` y deberás sobrecargar sus métodos.

Esta clase se encargará de recorrer la lista de animales que se le pase, para que, utilizando una clase interna se rellenarán los campos. Observa el código siguiente.

```
class RecyclerAdapter : RecyclerView.Adapter<RecyclerAdapter.ViewHolder>() {
    var animals: MutableList<MyAnimals> = ArrayList()
    lateinit var context: Context

    // Constructor de la clase. Se pasa la fuente de datos y el contexto
    // sobre el que se mostrará.
    fun RecyclerAdapter(animalsList: MutableList<MyAnimals>, contxt: Context) {
        this.animals = animalsList
        this.context = contxt
    }

    // Este método se encarga de pasar los objetos, uno a uno al ViewHolder
    // personalizado.
    override fun onBindViewHolder(holder: ViewHolder, position: Int) {
        val item = animals.get(position)
        holder.bind(item, context)
    }

    // Es el encargado de devolver el ViewHolder ya configurado.
    override fun onCreateViewHolder(
        parent: ViewGroup,
        viewType: Int
    ): ViewHolder {
        val layoutInflater = LayoutInflater.from(parent.context)
        return ViewHolder(
            ItemAnimalListBinding.inflate(
                layoutInflater,
                parent,
                false
            ).root
        )
    }

    // Devuelve el tamaño del array.
    override fun getItemCount(): Int {
        return animals.size
    }

    // Esta clase se encarga de rellenar cada una de las vistas que se inflarán
    // en el RecyclerView.
    class ViewHolder(view: View) : RecyclerView.ViewHolder(view) {

        // Se usa View Binding para localizar los elementos en la vista.
        private val binding = ItemAnimalListBinding.bind(view)

        fun bind(animal: MyAnimals, context: Context) {
            binding.tvNameAnimal.text = animal.animalName
            binding.tvLatinName.text = animal.latinName
            binding.ivAnimalImage.setImageResource(animal.imageAnimal!!)

```

```
            itemView.setOnClickListener {
                Toast.makeText(
                    context,
                    animal.animalName,
                    Toast.LENGTH_SHORT
                ).show()
            }
        }
    }
}
```

A continuación, fíjate como quedaría la clase principal `MainActivity.kt`, donde se han añadido dos nuevos métodos, uno para configurar el *RecyclerView*, y otro que se encargará de generar la fuente de datos.

```
class MainActivity : AppCompatActivity() {
    private lateinit var binding: ActivityMainBinding
    private val myAdapter: RecyclerAdapter = RecyclerAdapter()

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

        setUpRecyclerView()
    }

    // Método encargado de configurar el RV.
    private fun setUpRecyclerView() {
        // Esta opción a TRUE significa que el RV tendrá
        // hijos del mismo tamaño, optimiza su creación.
        binding.myRVAnimals.setHasFixedSize(true)

        // Se indica el contexto para RV en forma de lista.
        binding.myRVAnimals.layoutManager = LinearLayoutManager(this)

        // Se genera el adapter.
        myAdapter.RecyclerAdapter(getAnimals(), this)

        // Se asigna el adapter al RV.
        binding.myRVAnimals.adapter = myAdapter
    }

    // Método encargado de generar la fuente de datos.
    private fun getAnimals(): MutableList<MyAnimals> {
        val animals: MutableList<MyAnimals> = arrayListOf()

        animals.add(MyAnimals("Cisne", "Cygnus olor", R.mipmap.cisne))
        animals.add(MyAnimals("Erizo", "Erinaceinae", R.mipmap.erizo))
        animals.add(MyAnimals("Gato", "Felis catus", R.mipmap.gato))
```

```
        animals.add(MyAnimals("Gorrión", "Passer domesticus", R.mipmap.gorrion))
        animals.add(MyAnimals("Mapache", "Procyon", R.mipmap.mapache))
        animals.add(MyAnimals("Oveja", "Ovis aries", R.mipmap.oveja))
        animals.add(MyAnimals("Perro", "Canis lupus familiaris", R.mipmap.perro))
        animals.add(MyAnimals("Tigre", "Panthera tigris", R.mipmap.tigre))
        animals.add(MyAnimals("Zorro", "Vulpes vulpes", R.mipmap.zorro))

        return animals
    }
}
```

El resultado que se debería obtener sería algo muy parecido a la imagen que muestra la figura anexa, fíjate que se muestra el *toast* con el nombre del elemento que ha sido pulsado.

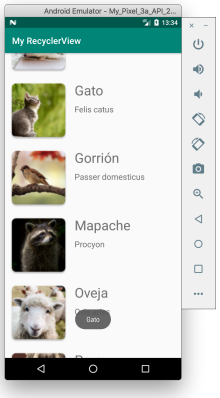

Figura 19

Observa que, al hacer clic sobre cualquiera de los elementos de la lista resultante, sobre cualquier parte del item, debe lanzarse un *toast* indicando el nombre común del animal. Esto se consigue mediante el código utilizado dentro del método `bind()` de la clase `ViewHolder` dentro de la clase `RecyclerAdapter`.

Si en vez de un *ListView*, prefieres utilizar un G*ridView*, vista en modo cuadrícula, únicamente deberás modificar la línea en la que se indica el contexto con forma de lista por esta otra que se muestra continuación, en la que se establece el contexto en forma de rejilla.

```
// Se indica el contexto para RV en forma de grid.
myRVAnimals.layoutManager = GridLayoutManager(this, 2)
```

El segundo parámetro de la clase `GridLayoutManager()` se utilizará para indicar el número de columnas que se quieran mostrar en el *RecyclerView*.

Los ***CardView***, como cualquier *ViewGroup*, se podrán añadir a los *layouts*, *activities* o *fragments* (que se verán más adelante). Para añadir un *CardView*, al igual que con *RecyclerView* se deberá importar una librería, pero ya se encargará **Android Studio**.

La principal ventaja de utilizar los *CardView*, además de agrupar los elementos en modo tarjeta, es la posibilidad de manipular el sombreado y elevación del *layout*, así como la posibilidad de redondear las esquinas.

Utilizando el código visto para el *RecyclerView*, para añadir un *CardView*, únicamente tendrás que modificar el fichero `item_animal_list.xml` de la siguiente forma.

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.cardview.widget.CardView
        xmlns:app="http://schemas.android.com/apk/res-auto"
        xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
```

```
        android:id="@+id/cv_item"
        android:padding="15dp"
        android:layout_margin="5dp"
        android:clickable="true"
        android:foreground="?android:attr/selectableItemBackground"
        app:cardCornerRadius="10dp"
        app:cardElevation="8dp"
        android:focusable="true">
    <RelativeLayout
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:padding="10dp"
            android:id="@+id/root_layout_cv"
            android:background="@android:color/holo_orange_light">
        <ImageView
                android:layout_width="150dp"
                android:layout_height="150dp"
                android:id="@+id/iv_animalImage"
                android:contentDescription="@string/desc_image"
                app:srcCompat="@mipmap/ic_launcher_round"/>
        <TextView
                android:text="@string/name_text"
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:id="@+id/tv_nameAnimal"
                android:layout_marginTop="10dp"
                android:layout_marginStart="5dp"
                android:layout_toEndOf="@id/iv_animalImage"
                android:textSize="30sp"/>
        <TextView
                android:text="@string/latin_text"
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:id="@+id/tv_latinName"
                android:layout_marginStart="5dp"
                android:layout_marginTop="10dp"
                android:layout_toEndOf="@id/iv_animalImage"
                android:layout_below="@id/tv_nameAnimal"
                android:textSize="18sp"/>
    </RelativeLayout>
</androidx.cardview.widget.CardView>
```

A continuación, se comentarán una serie de propiedades que pueden ayudar en el diseño de los *CardView*.

- **app:cardCornerRadius**, permite especificar el redondeo de las esquinas.
- **app:cardElevation**, eleva la *CardView*, a mayor elevación, mayor proyección de la sombra.
- Por defecto, las *CardView* no tiene el efecto *Ripple*, para ello se deberán añadir las dos siguientes atributos,

```
android:clickable="true"
android:foreground="?android:attr/selectableItemBackground"
```

Estos dos atributos se pueden añadir a otro tipo de elementos de un *layout*, como un *TextView*, un *ImageView*, etc, y poder aplicar el efecto.

¿Qué es el efecto *Ripple*? Es un *feedback*, devuelve una respuesta, generado sobre el elemento para dar sensación de movimiento sobre dicho elemento al ser pulsado.

El resultado que se obtendría debería ser similar al que se muestra en la imagen que tienes a continuación.

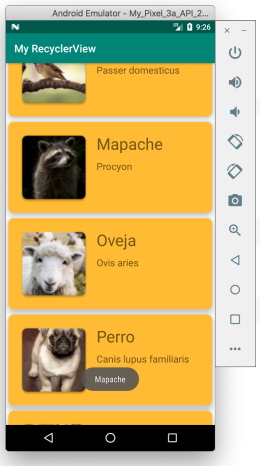

Figura 20

Ejercicios propuestos

4.10.1. Crea una nueva versión del ejercicio **4.9.1.** (*My Candies*) utilizando un *RecyclerView* con *CardView* para crear la lista.

4.10.2. Crea una nueva versión del ejercicio **4.9.2.** (*My Candies 2*) utilizando un *RecyclerView* con CardView para crear la cuadrícula.

4.10.3. Modifica el ejemplo visto para *RecyclerView* y *CardView* para que, en vez de mostrar un *Toast* al pulsar sobre un item, se muestre un *Snackbar*.