Programación Multimedia y Dispositivos Móviles

UD 4. Elementos básicos en Android

Javier Carrasco

Curso 2024 / 2025

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**. Tratarán de verse los elementos básicos que intervienen en el desarrollo de las aplicaciones móviles, 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 vez 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 (se inyecta) a un botón creado en la UI para detectar la pulsación sobre este. Se utilizará este ejemplo como referencia para contrastarlo.

```
Button button = (Button) findViewById(R.id.button_send);

button.setOnClickListener(new View.OnClickListener() {
    public void onClick(View v) {
        // Do something in response to button click
    }
});
```

4.1.1. View Binding

Esta estrategia de vinculación de vistas llamada *View Binding*[1] aparece con la versión 3.6. de Android Studio, y requiera la versión 5.6.1., o superior, de *Gradle*, puedes ver tu versión en el archivo `gradle-wrapper.properties`. La idea es facilitar el acceso a la UI. Para su uso se deberás 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 {
    ...
    viewBinding {
        enable = 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 realizará la inyección de la vista.

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

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

Fíjate que se utiliza `lateinit` para poder inicializar la variable en el momento que se necesite, en este caso, se iniciará en el método `onCreate()`. Como dato interesante, fíjate en el tipo de clase que crea el *binding*, `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`. Estas clases creadas se conocen como **clases de vinculación**., desde las cuales se podrá acceder a cualquier elemento de las vistas.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    binding = ActivityMainBinding.inflate(layoutInflater)
    // Se sustituye la referencia al recurso R.layout.activity_main
    // por el binding.
    setContentView(binding.root)
}
```

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 uno es con el *Hardcoded text*.

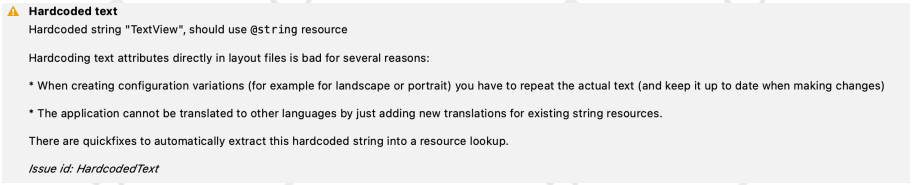

Figura 1

Esto es debido a que en Android, debe evitarse en medida de lo posible el uso de cadenas fijas 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, visto en capítulos anteriores.

```xml
<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[2], bastará con pulsar con el botón derecho sobre el fichero y seleccionar la opción ***Open Translations Editor***.

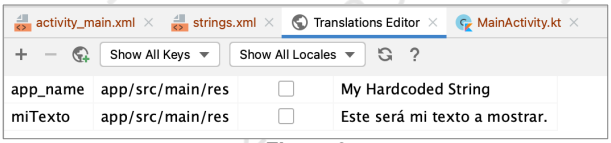

Figura 2

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.

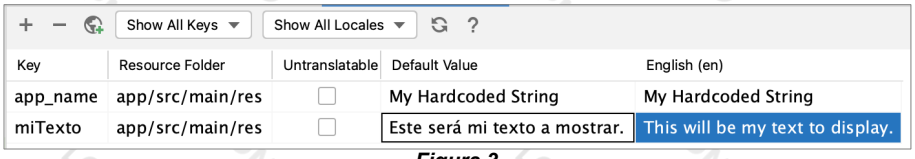

Figura 3

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`.

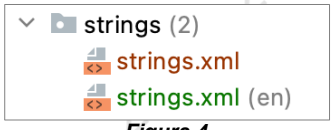

Figura 4

4.3. Elementos Layout

Cuando se trata de representar visualmente una pantalla, una UI, para el usuario, de tenerse muy claro que es un elemento *layout*[3], estos son capaces de representar el diseño gráfico de la interfaz de usuario. Este diseño será estructurado de manera jerárquica, formándose mediante el uso de *Views*, también conocidos como *widgets*, que son elementos simples, como un botón, una etiqueta, y por elementos *ViewGroup*, que contienen más *widgets*.

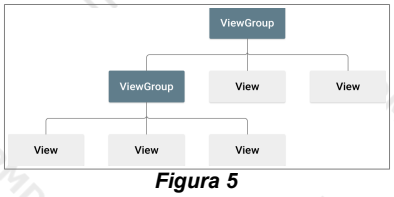

Figura 5

Si observas la figura con la jerarquía, verás que el elemento raíz de todo *layout* será un *ViewGroup*, los elementos más utilizados de este tipo son:

2 Editor de traducciones (https://developer.android.com/studio/write/translations-editor)
3 Diseños (https://developer.android.com/guide/topics/ui/declaring-layout.html)

- ***ConstraintLayout***[4], viene por defecto al crear una nueva vista, permiten cierta libertad para colocar los elementos, pero estos siempre deberán tener al menos dos puntos de referencia.
- ***LinearLayout***, se estructuran de manera vertical u horizontal, añadiendo cada elemento a continuación.
- ***FrameLayout***, muestra los elementos, hijos, a modo de pila, superponiéndolos.
- ***RelativeLayout***, es similar al *ConstraintLayout* en lo referente a libertad, pero cada hijo debe estar asociado a su hermano más inmediato.

Existen algunos más como el *TableLayout* o el *GridLayout* que puedes probar para estructurar las interfaces de usuario. Las propiedades más comunes y de uso habitual de estos elementos son las siguientes:

- El ***id***, que permitirá establecer un identificador para poder ser accedido si fuese necesario desde el código.
- Las propiedades ***layout_width*** y ***layout_height*** permiten establecer el tamaño, siendo los valores más comunes *match_parent*, para ajustarse al tamaño del padre y, *wrap_content* para ajustarse a su contenido.
- Otra propiedad, es ***orientation***, asociada al *LinearLayout*, esta permite establecer la dirección del elemento, vertical u horizontal.

Recuerda que un elemento *layout* puede contener a su vez otro de la misma naturaleza como se muestra en la estructura jerárquica.

4.4. Botones

El botón clásico, ***button***, es uno de los elementos que más relación tiene con los usuarios, se utiliza 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:id="@+id/button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/my_button" />
```

Recuerda el capítulo anterior, si quieres 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 son:

4 Diseño de una UI responsiva con *ConstraintLayout* (https://developer.android.com/training/constraint-layout)

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

Pues utilizar el botón de *Material* simplemente cambiando el nombre, en vez de utilizar *Button* puedes utilizar ***MaterialButton***, más acorde con el estilo *Material* y con las mismas propiedades.

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:id="@+id/imageButton"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    tools:srcCompat="@tools:sample/avatars" />
```

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:id="@+id/checkBox"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="CheckBox" />
```

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

```
<RadioGroup
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" >

    <RadioButton
        android:id="@+id/radioButton1"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="RadioButton1"
        android:checked="true"/>

    <RadioButton
        android:id="@+id/radioButton2"
        android:layout_width="match_parent"
```

```
        android:layout_height="wrap_content"
        android:text="RadioButton2" />
</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:id="@+id/toggleButton"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="ToggleButton"
    android:checked="true"/>

<Switch
    android:id="@+id/switch1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Switch"
    android:checked="false"/>
```

Una recomendación con respecto al *Switch*, es hacer uso de *SwitchMaterial* o *SwitchCompact*, de esta forma se utiliza el elemento de la librería *Material*, más acorde con el diseño y compatibilidades establecidas.

Se ha visto el XML generado al insertar los elementos en una *activity*, pero recuerda que dispones del *Layout Editor* para evitar escribir el código XML, aunque en ocasiones, puede resultar más rápido para encontrar determinadas propiedades.

A continuación, se verá el código **Kotlin** que se necesita para hacer referencia a los elementos insertados en el *layout*, y como hacer uso de algunas de las propiedades vistas. Se partirá de una *activity* vacía en la que en su clase asociada podrás encontrar el siguiente código inicial al crearla de nuevas.

Importante, a partir de este punto se hará uso del método *View Binding* visto en el punto 4.1. en la mayor parte del código, salvo excepciones en las que no pueda o sea aconsejable utilizar otro método.

```
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`.

Añade el siguiente código **Kotlin** a la clase asociada a la vista 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 se podrá aplicar a gran parte de los elementos con los que se trabajará en Android. Observa también como se añade el código al método *onCreate()*, recuerda el ciclo de vida de las *activities*.

```
override fun onCreate(savedInstanceState: Bundle?) {
    ...
    setContentView(binding.root)

    with(binding) { this:ActivityMainBinding
        button.setOnClickListener {
            checkBox.isChecked = true
            radioButton1.isChecked = true
            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. Además, se hace uso de la función estándar ***with***, la cual permite establecer un contexto de trabajo, en este caso haciendo uso del objeto *binding*, esto permite evitar tener que escribir *binding* delante de cada elemento de la vista, útil cuando son muchas líneas o muchas referencias a diferentes elementos, pero también puede usarse con otros contextos.

A continuación, se creará un *listener* para el *imageButton* para que desactive los activados anteriormente y active el *radioButton2*. Este nuevo *listener* también irá dentro del *scope* establecido con la función estándar *with*.

```
with(binding) {
    ...
    imageButton.setOnClickListener {
        if (checkBox.isChecked)
            checkBox.isChecked = false

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

Ordena el código utilizando una función de manera básica, pasándole como argumento un *string* en el que se indicará el tipo del elemento pulsado. Observa el código completo.

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

    private fun actionButtons(name: String) {
```

```
        with(binding) {
            when (name) {
                button.javaClass.name -> {
                    checkBox.isChecked = true
                    radioButton1.isChecked = true
                    toggleButton.isChecked = true
                    switch1.isChecked = true
                }
                else -> {
                    if (checkBox.isChecked)
                        checkBox.isChecked = false

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

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityMainBinding.inflate(layoutInflater)
        // Se sustituye la referencia al recurso R.layout.activity_main
        // por el binding.
        setContentView(binding.root)

        with(binding) {
            button.setOnClickListener { actionButtons(button.javaClass.name) }
            imageButton.setOnClickListener { actionButtons(imageButton.javaClass.name) }
        }
    }
```

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, y que en **Kotlin** es `when`, pero el funcionamiento básicamente el mismo, pero con más posibilidades que se irán viendo a medida que se vaya avanzando[5].

4.5. Toast

Los ***toasts*** son mensajes flotantes que se mostrarán al usuario en pantalla durante un determinado periodo de tiempo. Desaparecerán automáticamente y el usuario no interviene para nada. Ya se han utilizado para saludar en capítulos anteriores. 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, al terminar una acción, etc. La instrucción para mostrar este tipo de mensajes flotantes será la siguiente.

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

```
Toast.makeText(
    context: this,
    text: "Este es un mensaje flotante",
    Toast.LENGTH_SHORT
).show()
```

Fíjate en el parámetro ***context***, en este caso se ha utilizado *this*, esto es debido a que se ha lanzado el *toast* desde la propia clase (*MainActivity*), pero en ocasiones, este *this* es posible que no haga referencia a la clase, por ejemplo, si estás dentro del *with(binding)*, el *this* hará referencia a *ActivityMainBinding*, por lo que daría error. Para especificar un contexto diferente, como el de la actividad sobre la que mostrar el *toast*, se tendría que utilizar `this@MainActivity` u otros métodos como se verá más adelante.

También es posible modificar la posición por defecto, cambiando su ubicación mediante el método `setGravity` de la propia clase *Toast* y utilizando la clase *Gravity*.

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

Como has podido ver, para crear un *toast* bastará 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*). Para hacer visible el *toast* deberás utilizar el método *show()*.

Por último, como habrás visto, se ha utilizado como contexto `applicationContext`, este devolverá el contexto global de la aplicación que se esté ejecutando, en el caso del *toast*, mostrará el mensaje sobre la aplicación en pantalla. Esta forma de obtener el contexto solo debe utilizarse cuando se necesita un contexto separado del ciclo de vida del contexto actual, como se ha comentado anteriormente.

4.6. Etiquetas, cuadros de texto e imágenes

Se comenzará viendo las **etiquetas** (*TextView*), estas son la manera más sencilla de mostrar información al usuario dentro de una *activity*. El texto contenido en un *TextView* no puede ser editado directamente por el usuario, no es un cuadro de texto, pero sí puedes ser modificado mediante una acción que modifique su contenido.

```
<TextView
    android:id="@+id/textView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/my_textView" />
```

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). Suele activarse junto con la propiedad *focusable*.
- **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*, observa como se hace uso del método *text* de la etiqueta y como se concatena con la cadena en el mensaje del *Toast* mediante el uso de `${…}`.

```kotlin
binding.button.setOnClickListener {
    Toast.makeText(
        this,
        "Texto de la etiqueta: ${binding.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*. Al igual que se ha dicho con los botones, puedes sustituir un *TextView* por un ***MaterialTextView***.

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

```xml
<EditText
    android:id="@+id/editText"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:hint="@string/yourTextHere"
    android:importantForAutofill="no"
    android:inputType="none" />

<EditText
    android:id="@+id/editText2"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:hint="@string/yourEmailHere"
    android:importantForAutofill="no"
    android:inputType="textEmailAddress" />
```

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). Suele activarse junto con la propiedad focusable.
- **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**[6], 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 más conveniente a utilizar. Ahora se añadirá algo de código para ver como se puede acceder a las propiedades de los *EditText*.
- **imeOptions**, esta es una propiedad que puede resultar de mucha utilidad cuando se está creando un formulario, permite modificar el comportamiento de la tecla "Intro" del teclado, por ejemplo, al pulsar "Intro", hacer que el foco pase al siguiente campo a completar (*actionNext*).

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

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

    message += "Texto de la etiqueta: ${binding.textView.text}"

    Toast.makeText(
        this,
        message,
        Toast.LENGTH_SHORT
    ).show()
}
```

Los *EditText* en Android disponen de una propiedad llamada `error` que puede resultar de gran utilidad, a esta, si se le asigna un texto muestra un signo de *warning* en el propio campo a modo de aviso y, cuando se pulse sobre signo de aviso se mostrará el texto asignado.

```
binding.editText.error = "Campo requerido"
```

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

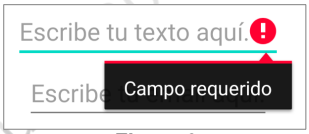

Figura 6

Este tipo de cuadros de texto, por motivos de accesibilidad, se recomienda que vayan acompañados de una etiqueta que pueda ser leída e identifique que dato debe ir en el cuadro de texto. Esto evitará los *warnings* que aparecen en el diseño del *layout*, pero no es obligatorio.

Ahora bien, se dispone de otro tipo de cuadro de texto que facilita esto, los *TextInputLayout* y pertenecen a la librería de *Material*. Estos son un elemento compuesto formado por una etiqueta y un cuadro de texto que trabajan juntos.

```
<com.google.android.material.textfield.TextInputLayout
    android:id="@+id/myInputLabel"
    style="@style/Widget.MaterialComponents.TextInputLayout.OutlinedBox"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:hint="@string/txt_Label">

    <com.google.android.material.textfield.TextInputEditText
        android:id="@+id/myInput"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />
</com.google.android.material.textfield.TextInputLayout>
```

Tanto la etiqueta como el cuadro de texto, disponen de las propiedades vistas. También fíjate en la propiedad ***style*** de la etiqueta, que permite cambiar el diseño del *widget*. Como detalle, interesante, la propiedad *error*, puedes utilizarla en el cuadro de texto…

```
binding.button.setOnClickListener {
    binding.myInput.error = "Error!!!"
}
```

… lo que daría como resultado la siguiente imagen.

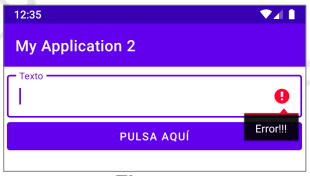

Figura 7

O puedes utilizarla sobre la etiqueta…

```
binding.button.setOnClickListener {
    binding.myInputLabel.error = "Error!!"
}
```

… obteniendo el siguiente resultado.

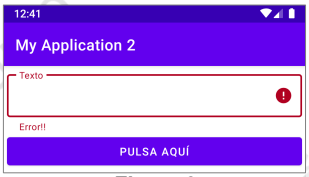

Figura 8

A diferencia de los *EditText*, la propiedad *Text* de los *TextInputEditText* es de tipo ***Editable***, por lo que no puedes asignarle un texto directamente mediante un *String*, para ello deberás utilizar los métodos de los que dispone.

```
binding.myInput.text?.clear() // Borra el contenido.
binding.myInput.setText(R.string.app_name) // Asigna un recurso string.
binding.myInput.setText("hola") // Asigna un texto.
binding.myInput.append("adiós") // Añade texto.
```

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 3) o desde un recurso en Internet.

```
<ImageView
    android:id="@+id/imageView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:contentDescription="@string/android_kotlin"
    app:srcCompat="@drawable/androidkotlin" />
```

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`.

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

Trabajar con imágenes, especialmente cuando se trata de cargar una en tiempo de ejecución puede resultar algo tedioso, sobre todo si esas imágenes se encuentran en la nube. Por ello, existen una serie de librerías externas, como *Glide*[7] o *Picasso*[8], que permiten que esta tarea sea algo más sencilla. A continuación se muestran los pasos para utilizar *Glide*.

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

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

```
dependencies {
    …
    // Glide
    implementation 'com.github.bumptech.glide:glide:4.14.2'
}
```

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://...Kotlin_vs_java.jpg") // Recurso.
    .override(300, 300) // Ajusta el tamaño.
    .centerCrop() // Centra la imagen.
    .into(binding.imageView) // Contenedor.
```

Aunque se tratará más adelante, para poder cargar la imagen es necesario dar permiso de uso de Internet a la aplicación, para ello, abre el fichero `AndroidManifest.xml` y añade la siguiente línea antes de la etiqueta *application*.

```
<uses-permission android:name="android.permission.INTERNET"/>
```

En ocasiones puede resultar incómodo el teclado que se muestra en pantalla, y puede ser interesante ocultarlo cuando se pierda el foco del componente que ha hecho aparecer el teclado.

Las siguientes dos líneas de código te permitirán ocultar el teclado una vez ejecutadas, ya bien sea aplicándolas al quitar el foco del componente, al ponerlo en otro o al pulsar un botón.

```
val imm = getSystemService(INPUT_METHOD_SERVICE) as InputMethodManager
imm.hideSoftInputFromWindow(currentFocus!!.windowToken, 0)
```

4.7. 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* del usuario.

El siguiente código *Kotlin* mostrará un *SnackBar* al pulsar sobre el botón identificado como *button1*. Es importante resaltar el uso del método ***root*** del objeto *binding* que hace referencia a la raíz de la UI, a diferencia de *Toast* se necesita conocer la vista sobre la que mostrar el *SnackBar*, no el contexto, y así se puede obtener la vista.

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

    // Snackbar simple.
    binding.button1.setOnClickListener { showSnackSimple() }
    // Snacbar con acción.
    binding.button2.setOnClickListener { showSnackAction() }
}

// Método encargado de mostrar un Snackbar simple.
private fun showSnackSimple() {
    Snackbar.make(binding.root, "Mi primer Snackbar!", Snackbar.LENGTH_LONG).show()
}

// Método encargado de mostrar un Snackbar capaz de recibir feedback.
private fun showSnackAction() {
    // Se cambia el color de fondo del layout
    binding.root.setBackgroundColor(Color.YELLOW)
    Snackbar.make(
        binding.root,
        "Mi SnackBar con acción!",
        Snackbar.LENGTH_LONG
    ).setAction(
        "Deshacer" // Texto del botón
    ) { // Acciones al pulsar el botón "Deshacer"
        binding.root.setBackgroundColor(Color.WHITE)
    }.show()
}
```

El resultado que deberías obtener con el código visto deberá parecerse a la imagen que se muestra en la siguiente figura.

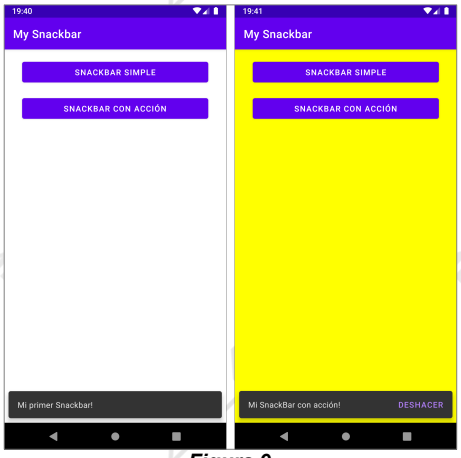

Figura 9

4.8. 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 de la pantalla.

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 y se verán más adelante.

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, de esta forma se intenta recrear una vista más grande que la parte visible del dispositivo.

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

    <ScrollView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_marginStart="5dp"
        android:fillViewport="true"
        android:scrollbarSize="10dp"
        android:scrollbarStyle="insideInset" >

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

            <TextView
                android:id="@+id/texto"
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:text="@string/myText" />

        </LinearLayout>
    </ScrollView>
</androidx.constraintlayout.widget.ConstraintLayout>
```

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***[9], que funciona de igual manera que *ScrollView*.

También existe ***NestedScrollView***[10], 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.

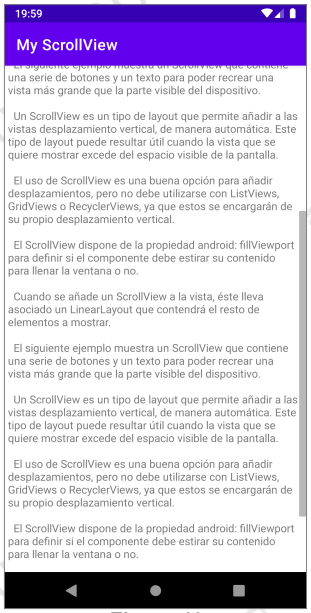

Figura 10

4.9. RatingBar

Estos elementos visuales permiten mostrar una valoración mediante una barra de estrellas o, permitir al usuario elegir la puntuación de una manera más visual.

Figura 11

El código para añadir este elemento puede resultar como el siguiente.

```
<RatingBar
    android:id="@+id/ratingBar"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:numStars="5"
    android:rating="3"
    android:stepSize="1" />
```

La propiedad `numStars` indica el número de estrellas que se mostrarán, destacar que el ancho deberá estar ajustado a *wrap_content* para que se vea correctamente el número de estrellas. `Rating` es el valor, tanto el establecido de inicio, como el que asigne el usuario, y `stepSize`, indicará el valor de incremento de la puntuación. Para recoger el valor de la puntuación desde código, bastará con acceder a la propiedad *Rating*, que devuelve un valor de tipo *float*.

```
val puntuación: Float = binding.ratingBar.rating
```

Es posible que las estrellas, según la temática de la aplicación que estés desarrollando no se adapte a tus necesidades. Para solucionar esto, puedes personalizar el aspecto de tu *RatingBar*. En primer lugar necesitarás tres imágenes, una que represente un elemento vacío, otra a mitad y otra con el elemento lleno.

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

Una vez los tengas, deberás importarlos al proyecto como *drawables*.

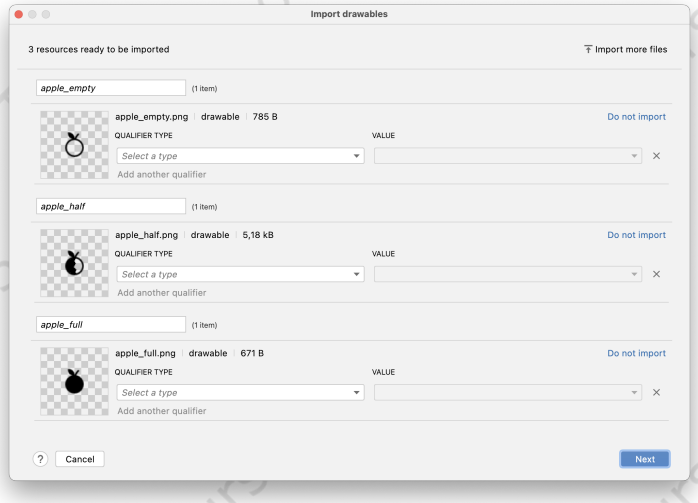

Figura 12

El siguiente paso será crear un nuevo *Drawable* (***File > New > Android Resource File***) y con ***layer-list*** como elemento raíz.

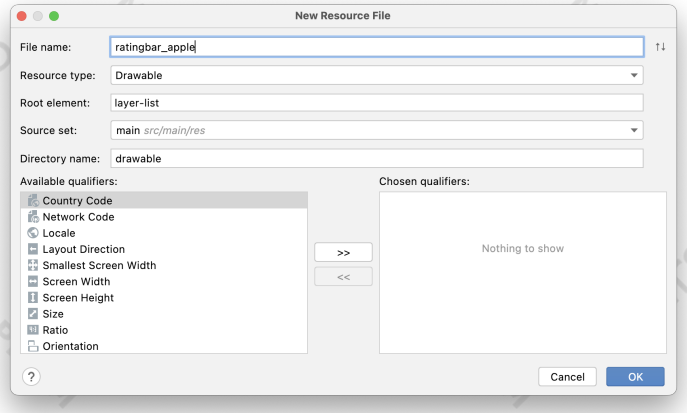

Figura 13

El resultado del nuevo recurso tras añadir las nuevas imágenes debe ser algo parecido al siguiente código.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <layer-list xmlns:android="http://schemas.android.com/apk/res/android">
3      <item android:id="@android:id/background" android:drawable="@drawable/apple_empty" />
```

```
    <item android:id="@android:id/secondaryProgress"
        android:drawable="@drawable/apple_half" />
    <item android:id="@android:id/progress" android:drawable="@drawable/apple_full" />
</layer-list>
```

Observa que los identificadores que se utilizan para los ítems ya están definidos por Android. A continuación, deberás crear un nuevo estilo para la *RatingBar* en `themes.xml`.

```
<style name="AppleRatingBar" parent="@style/Widget.AppCompat.RatingBar">
    <item name="minHeight">48dp</item>
    <item name="maxHeight">48dp</item>
    <item name="android:progressDrawable">@drawable/ratingbar_apple</item>
</style>
```

En este ejemplo se ha puesto la altura mínima y máxima a 48dp porque el tamaño de las imágenes es de 48x48. Ya solo quedaría establecer el estilo a la *RatingBar* con la propiedad *Style*.

```
<RatingBar
    android:id="@+id/ratingBar"
    style="@style/AppleRatingBar"
    ... />
```

Con esto, ya puedes personalizar tus *RatingBar*. El resultado que obtendrías sería algo parecido a la siguiente imagen.

Figura 14

4.10. Vista horizontal

Llegados este punto, la *app* de ejemplo dispone ya de varias vistas y, si giras el dispositivo, poniéndolo en horizontal, estos se adaptarán a la pantalla tal y como se han ubicado inicialmente.

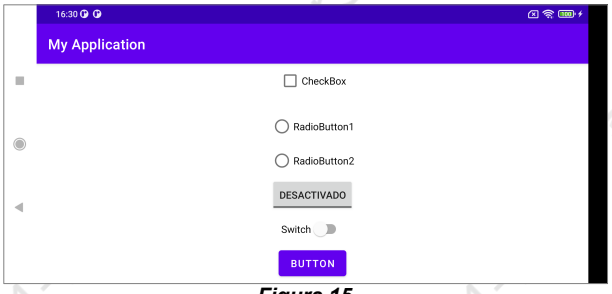

Figura 15

Es posible que esta disposición no sea la deseable, observa como los elementos que deberían ir debajo del botón no aparecen, y como no se ha añadido un *ScrollView*, no se puede desplazar hacia arriba. Es por eso por lo que se puede diseñar una nueva pantalla para cuando el dispositivo cambie de posición. Para ello, abre el *layout* que quieras adaptar y selecciona la opción ***Create Landscape Qualifier*** tal como se muestra la figura que aparece a continuación.

Ahora, podrás observar cómo se dispondrá de dos archivos *XML* con el mismo nombre, pero diferenciados por el indicador *(land)* en el explorador de proyecto. Recuerda que aunque tengan el mismo nombre, deberás editarlos por separado, uno se corresponderá con la vista vertical, y el otro (*land*) con la vista horizontal.

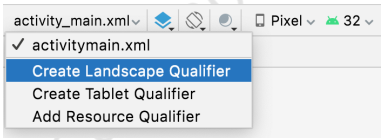

Figura 16

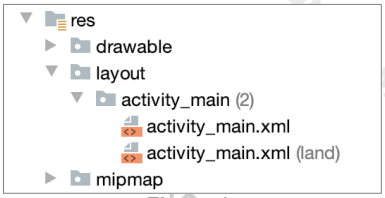

Figura 17

En este caso, se dividirá la pantalla en dos, al 50%, haciendo uso de una *Guideline* vertical y modificando su propiedad *layout_constraintGuide_percent* con un valor de 0.5, este elemento lo puedes encontrar en la paleta. Se dejará una parte de los elementos en la columna de la izquierda, y el resto en la derecha.

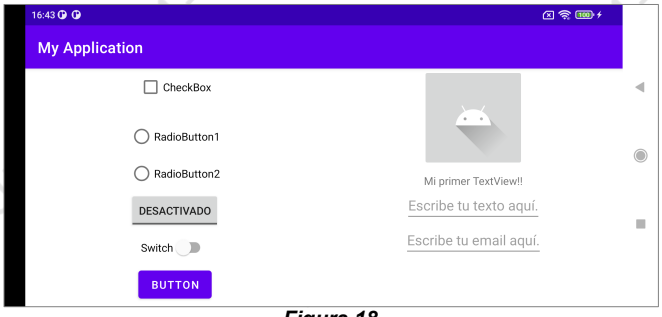

Figura 18

Ahora se añadirá un nuevo botón bajo el último *TextView* (el del email), la particularidad de este botón será que no estará disponible en la vista vertical, únicamente aparecerá cuando la vista se encuentre en horizontal (*landscape*).

Siguiendo el esquema visto, lo más probable es que la aplicación produzca un error, pero *Kotlin* ofrece una serie de herramientas para solventar este tipo de problemas. El código para el nuevo botón en la vista horizontal lanzará simplemente un *Toast*.

```
1  binding.btnLandscape.setOnClickListener {
2      Toast.makeText(this, "Botón de la vista horizontal", Toast.LENGTH_SHORT).show()
3  }
```

Pero este código, si se deja así, no se ejecutará la aplicación, se producirá un error de compilación, esto es debido a la protección ante nulos que ofrece *Kotlin*. Para solucionar este problema deberá hacerse una pequeña modificación en el código, pero que dará la salida buscada para poder ejecutar.

En primer lugar, se utilizará el operador interrogación `?` para indicar que existe la posibilidad de encontrar un nulo en el elemento en cuestión, el botón, y seguidamente se aplicará el método `let` para que, en caso de no ser nulo, indicar el código que deberá ejecutarse.

Ya con el nuevo botón en la vista horizontal, la *activity* mostrará dos UI en la aplicación según la posición en la que se encuentre el dispositivo.

Esta es una opción muy buena a tener en cuenta, ya que cuando se crean diferentes vistas para un mismo *layout*, como es el caso de los dos *layouts* para `activity_main.xml`, se sigue disponiendo una una sola clase para gestionarlos, de ahí que deba controlarse la existencia de elementos presentes en una vista pero no en la otra.

El código *Kotlin* adaptado a esta nueva situación de la aplicación puede ser como el que se muestra a continuación.

```
binding.btnLandscape?.let {
    it.setOnClickListener {
        Toast.makeText(this, "Botón de la vista horizontal", Toast.LENGTH_SHORT).show()
    }
}
```

Ahora la aplicación el nuevo botón en la vista horizontal totalmente funcional.

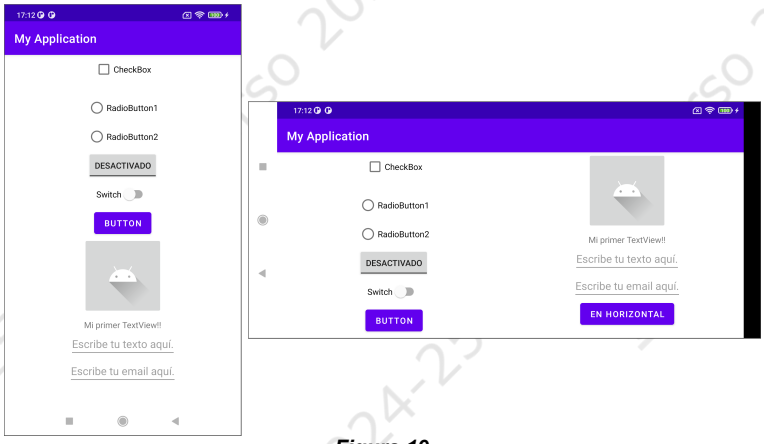

Figura 19

Por último, puede resultar útil conocer en un momento dado en que posición se encuentra el dispositivo móvil. Para obtener esta información se pueden utilizar las siguientes líneas, puedes añadirlas en el método *onStart* para asegurarte de que siempre se ejecuten al repintar la *activity*.

```
val estado = if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.R)
    this.display?.rotation
else {
    @Suppress("DEPRECATION")
    this.windowManager.defaultDisplay.rotation
}
Log.i("DISPLAY", estado.toString())
```

Se debe utilizar el chequeo de versión debido a que el uso de defaultDisplay está marcado como deprecad desde la API 30 (Android R).

En este caso, la variable `estado` podría obtener los siguientes valores según la posición del dispositivo. Se tendrá como referencia la posición del auricular del dispositivo o la cámara frontal.

- **0** en vertical a 90º (auricular en la parte superior).
- **1** en horizontal a 180º.
- **2** en vertical a 270º.
- **3** horizontal a 0º o 360º.

Una segunda forma para controlar la orientación puede ser la que se muestra a continuación:

```
if (resources.configuration.orientation == Configuration.ORIENTATION_LANDSCAPE) {
    // TO-DO
    Log.i("DISPLAY", resources.configuration.orientation.toString())
}
```

La diferencia con el método anterior es que, únicamente distingue entre dos posiciones, vertical (1) y horizontal (2).

Otra opción es evitar la rotación directamente, existen dos formas de hacerlo, la primera mediante el *manifest*, para ello, se deberá añadir la siguiente propiedad a la *activity*.

```
<activity
    android:name=".MainActivity"
    ...
    android:screenOrientation="portrait">
```

Asigna el valor en función de las necesidades, *portrait*, *landscape*, *fullSensor*, etc. El inconveniente es que se debe añadir para todas las actividades, no hay opción de hacerlo nivel de aplicación.

La segunda opción, y seguramente más cómoda, ya que no debes ir al *manifest* por cada *activity* creada, es añadir la siguiente línea en el método *onCreate* de las clases que lo requieran.

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

Como en la primera opción, dispones de múltiples opciones de configuración para la pantalla. Verás que el IDE lo marca, indicando simplemente que el usuario puede perder usabilidad.

Y llegados a este punto, es posible que cuando la aplicación esté en horizontal, no te interese que se vea la *ActionBar*, puedes ocultarla desde código con la siguiente línea, haciendo uso del objeto *supportActionBar*.

```
supportActionBar!!.hide()
```