

Tema 4. Elementos básicos en Android

4.1 Vinculación de vista

4.2 Hardcoded text

4.3 Elementos Layout

4.4 Botones

4.4.1 Eventos

4.4.2 Implementación anónima, lambdas en Kotlin, let y el operador Elvis

4.5 Toast

4.6 Etiquetas, cuadros de texto e imágenes

4.6.1 Internacionalización y localización

4.7 SnackBar

4.8 ScrollView

4.9 Adapter y AdapterView

4.10 Elementos complejos

4.11 RecyclerView y CardView

4.1 Vinculación de vista

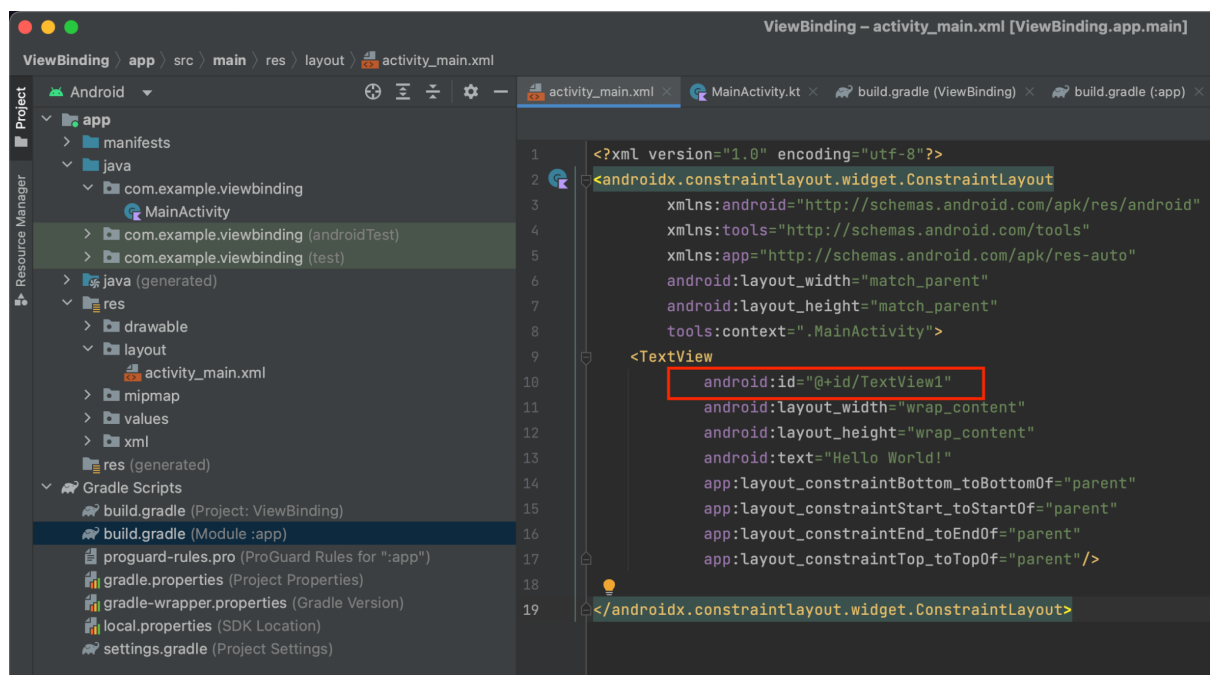
La vinculación de vista en Android con Kotlin, es una técnica que te permite acceder fácilmente a las vistas de tu diseño de interfaz de usuario (UI) desde tu código Kotlin.

Tenemos dos opciones:

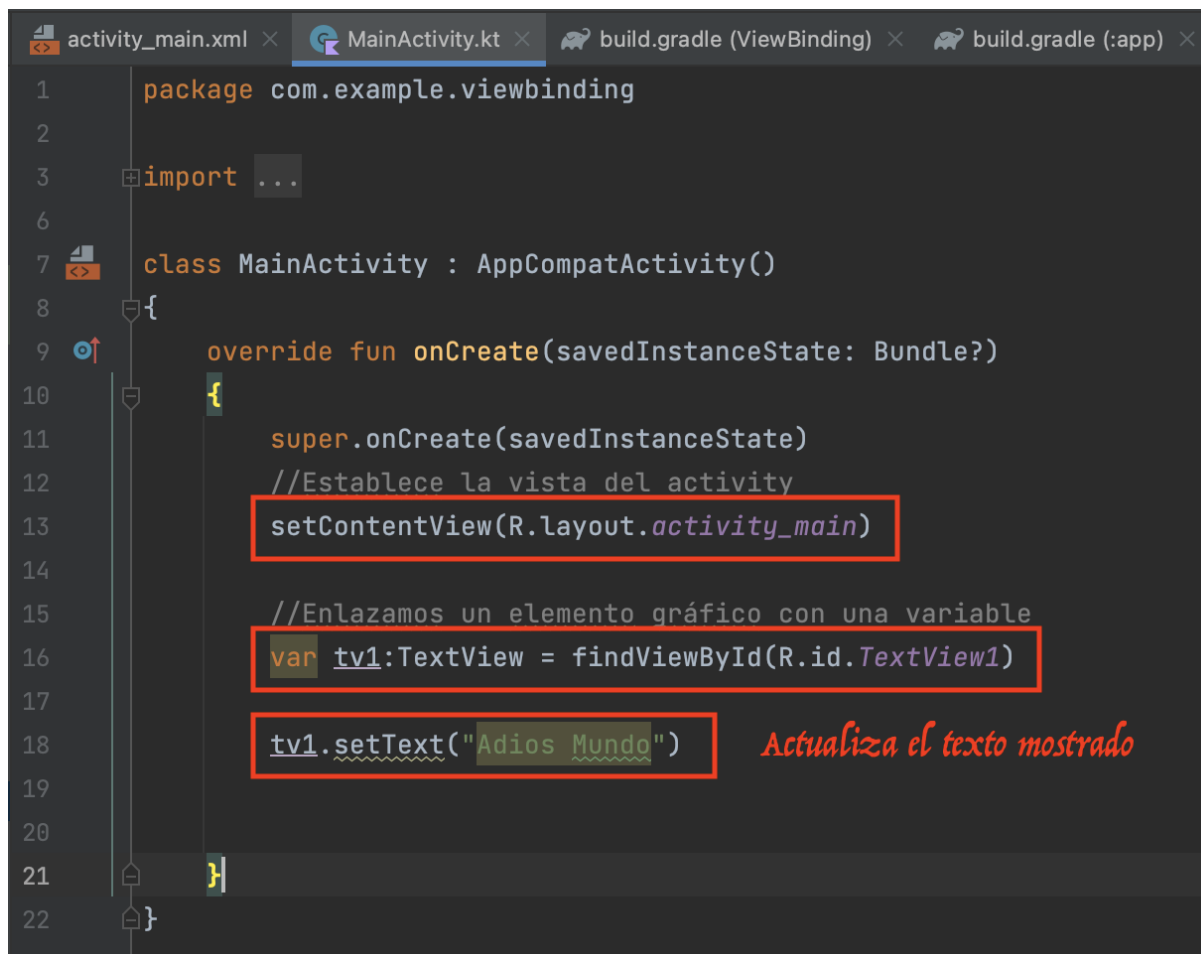
Opción 1. Función `findViewById`

Este es el método original de realizar la vinculación de vista.

En primer lugar, es necesario tener asignado un **id**, al elemento gráfico declarado dentro del *layout*, que posteriormente vamos a vincular con una variable en el fichero de Kotlin.



En segundo lugar procedemos con la vinculación de vista, desde el fichero de Kotlin asociado a dicho layout.



```
1 package com.example.viewbinding
2
3 import ...
4
5
6
7 class MainActivity : AppCompatActivity()
8 {
9     override fun onCreate(savedInstanceState: Bundle?)
10    {
11        super.onCreate(savedInstanceState)
12        //Establece la vista del activity
13        setContentView(R.layout.activity_main)
14
15        //Enlazamos un elemento gráfico con una variable
16        var tv1:TextView = findViewById(R.id.TextView1)
17
18        tv1.setText("Adios Mundo")
19
20
21    }
22 }
```

Si nos decidimos por esta opción, será necesario crear una *variable*, por cada uno de los elementos gráficos que queramos vincular.

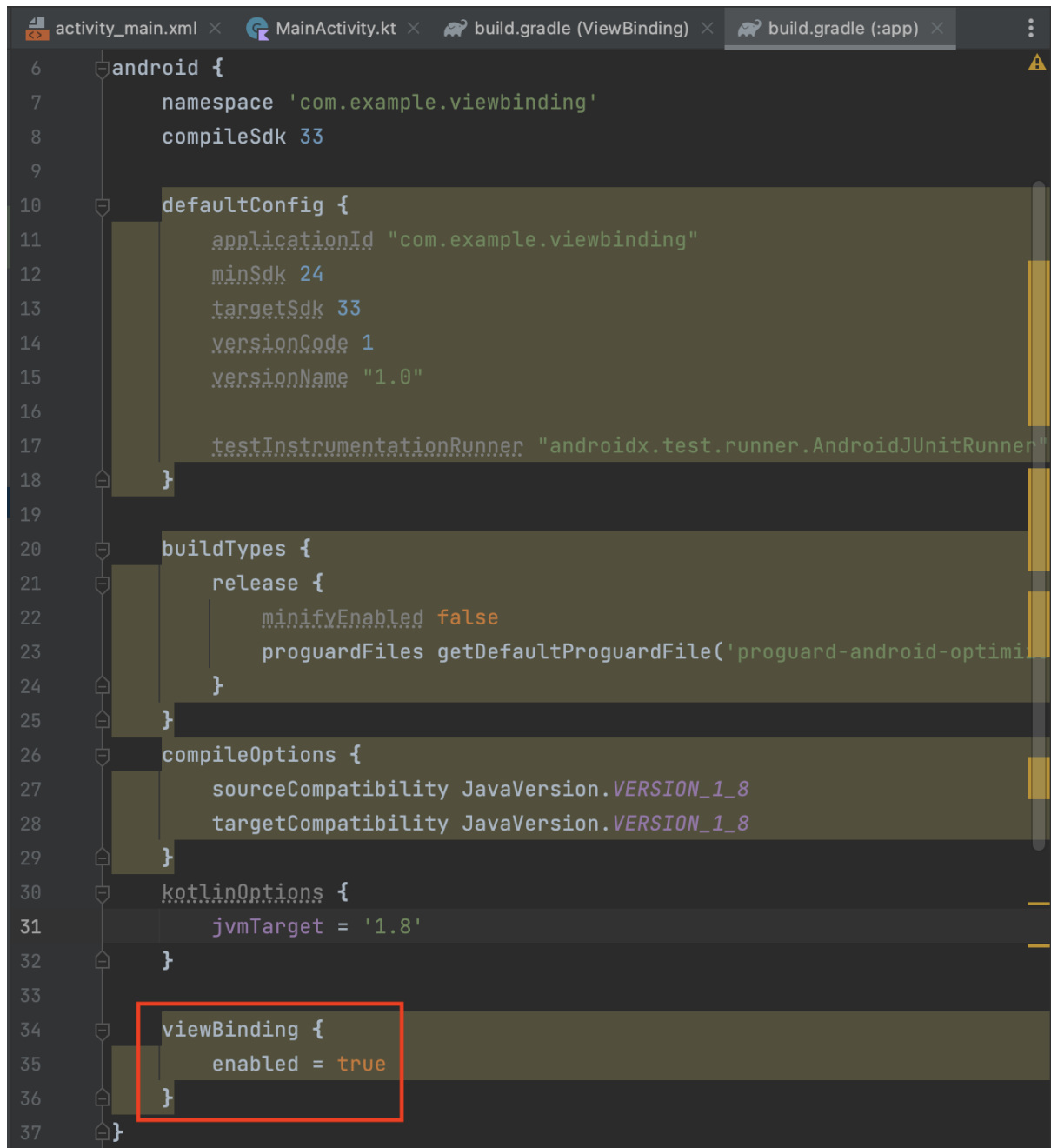
Opción 2. View Binding

La vinculación de vista en Android con Kotlin es una técnica que te permite acceder fácilmente a las vistas de tu diseño de interfaz de usuario (UI) desde tu código Kotlin sin necesidad de utilizar **findViewById**.

Aquí tienes un ejemplo de cómo realizar la vinculación de vista en Kotlin:

Supongamos que tienes un archivo XML llamado **activity_main.xml** que contiene un TextView con el id *TextView1*. Para vincular esta vista en Kotlin, sigue estos pasos:

Asegúrate de que tengas la configuración adecuada en tu archivo **build.gradle** (a nivel de app) para habilitar la vinculación de vista (**ViewBinding**):



```
6  android {
7      namespace 'com.example.viewbinding'
8      compileSdk 33
9
10     defaultConfig {
11         applicationId "com.example.viewbinding"
12         minSdk 24
13         targetSdk 33
14         versionCode 1
15         versionName "1.0"
16
17         testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
18     }
19
20     buildTypes {
21         release {
22             minifyEnabled false
23             proguardFiles getDefaultProguardFile('proguard-android-optimize')
24         }
25     }
26     compileOptions {
27         sourceCompatibility JavaVersion.VERSION_1_8
28         targetCompatibility JavaVersion.VERSION_1_8
29     }
30     kotlinOptions {
31         jvmTarget = '1.8'
32     }
33
34     viewBinding {
35         enabled = true
36     }
37 }
```

En tu archivo XML **activity_main.xml**, asegúrate de que la etiqueta layout tenga un id asignado. Por ejemplo:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/mainLayout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <TextView
        android:id="@+id/TextView1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintTop_toTopOf="parent"/>

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

En tu actividad (por ejemplo, **MainActivity.kt**), realiza la vinculación de vista de la siguiente manera:

```
package com.example.viewbinding

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.widget.TextView
import com.example.viewbinding.databinding.ActivityMainBinding

class MainActivity : AppCompatActivity() {
    private lateinit var binding: ActivityMainBinding // Declara una variable de vinculación

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

        binding = ActivityMainBinding.inflate(layoutInflater) // Infla la vista utilizando la clase de vinculación
        val view = binding.root

        //Establece la vista del activity
        setContentView(view)

        // Ahora puedes acceder a tus vistas directamente usando la variable de vinculación
        binding.TextView1.setText("¡Adiós, Mundo!")
    }
}
```

Asegúrate de que el import de **com.example.viewbinding.databinding.ActivityMainBinding** refleje la ubicación real de tu clase de vinculación, que se genera automáticamente a partir del nombre de tu archivo XML.

Con esto, has vinculado la vista **TextView1** de tu diseño de interfaz de usuario y puedes manipularla directamente en tu código Kotlin sin necesidad de usar **findViewById**.

La vinculación de vista hace que tu código sea más limpio y menos propenso a errores relacionados con los identificadores de vista.

!! NOTA IMPORTANTE !!

La variable **layoutInflater** es una instancia de la clase **LayoutInflater** en Android. Se utiliza para **inflar (convertir)** un archivo de diseño **XML** en una vista de objetos **View** que se pueden utilizar en la interfaz de usuario de tu aplicación.

Es una propiedad de instancia que puedes acceder dentro de una **actividad** (activity) o en otros contextos de la interfaz de usuario en Android. Está disponible a través de métodos como **getLayoutInflater()** o simplemente **layoutInflater**. Pertenecce al contexto (**Context**) en Android. Es una propiedad de instancia de la clase **Context**.

En una actividad (**Activity**), puedes acceder a **layoutInflater** directamente porque **Activity** es una subclase de **Context** y hereda sus propiedades y métodos.

Cuando trabajas con la vinculación de vista (**ViewBinding**) en Android, a menudo se usa **layoutInflater** para inflar el diseño XML en una vista que luego se establece como el contenido de una actividad o se utiliza en otros contextos de la interfaz de usuario.

En el ejemplo de uso anterior, **layoutInflater** se utiliza para inflar el diseño definido en **activity_main.xml** en un objeto de la clase **ActivityMainBinding**. Luego, **binding.root** se utiliza para obtener la vista raíz del diseño inflado, que puede establecerse como el contenido de la actividad.

En resumen, **layoutInflater** es una herramienta importante en Android que te permite crear objetos **View** a partir de archivos **XML** de diseño para que puedas trabajar con ellos programáticamente en tu aplicación.

4.2 Hardcoded text

"Hardcoded text" en Kotlin/Android se refiere a la práctica de incluir texto directamente dentro del código fuente de una aplicación en lugar de almacenarlo en recursos externos, como archivos de recursos de cadena (**strings.xml**).

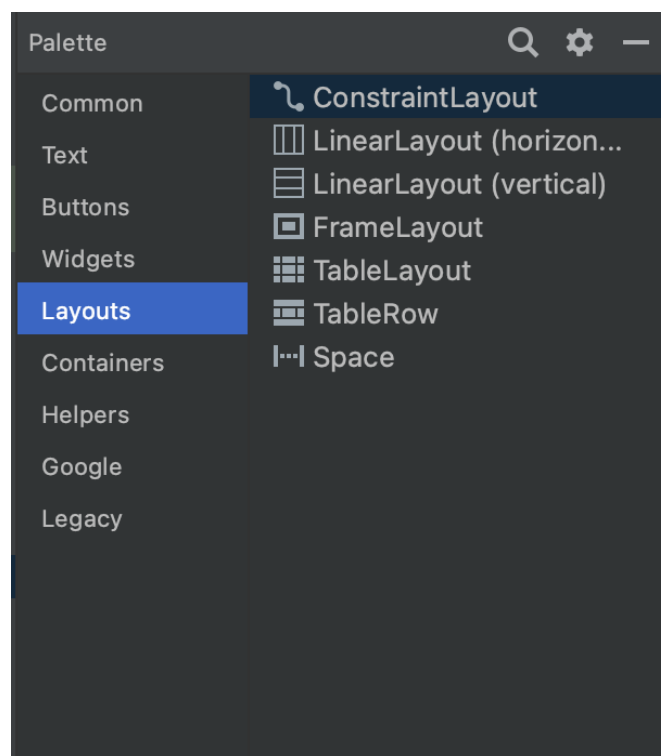
Esto significa que el texto se encuentra "duro" (hardcoded) en el código y no se puede cambiar fácilmente sin modificar el código fuente de la aplicación.

Aunque es posible utilizar texto directamente en el código, **se recomienda evitarlo** siempre que sea posible y, en su lugar, externalizar el texto a recursos de cadena. Aquí hay algunas razones para evitar el "hardcoded text":

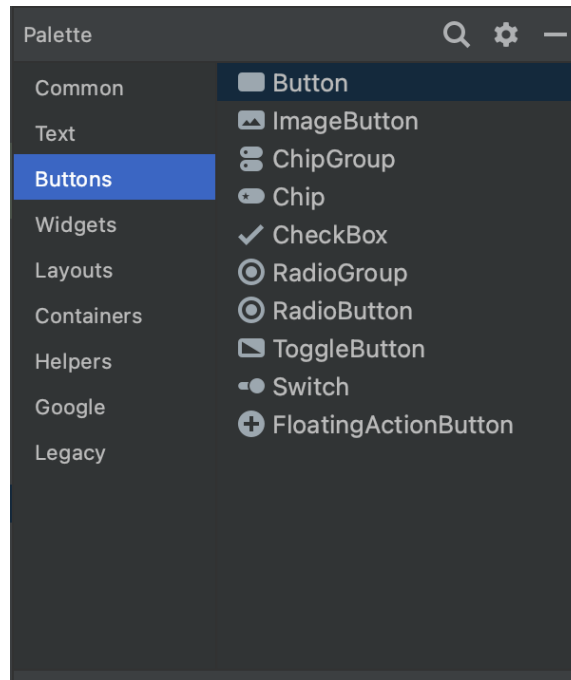
1. **Facilita la localización:** si deseas que tu aplicación sea internacionalizada y traducida a varios idiomas, es mucho más sencillo gestionar las traducciones si el texto se almacena en archivos de recursos de cadena. Los archivos de recursos de cadena permiten mantener diferentes versiones del mismo texto para diferentes idiomas.
2. **Facilita las actualizaciones:** si tienes texto hardcoded en varios lugares de tu código y necesitas cambiar ese texto (por ejemplo, para corregir un error ortográfico o para realizar una actualización importante), debes buscar y modificar manualmente cada instancia del texto en tu código. Esto puede ser propenso a errores y consumir mucho tiempo.
3. **Mejora la legibilidad y el mantenimiento del código:** al externalizar el texto a recursos de cadena, tu código se vuelve más legible y mantenible. El texto se encuentra en un solo lugar y es más fácil de gestionar.
4. Para externalizar texto a recursos de cadena en Android, puedes definir cadenas de texto en el archivo ***strings.xml*** dentro de la carpeta **res/values** de tu proyecto y luego acceder a ellas mediante su identificador en tu código Kotlin. Esto permite una mejor organización y mantenimiento del texto en tu aplicación.

4.3 Elementos Layout

Se han visto en el tema 2.



4.4 Botones



4.4.1 Eventos

Comencemos por ver algunos conceptos necesarios:

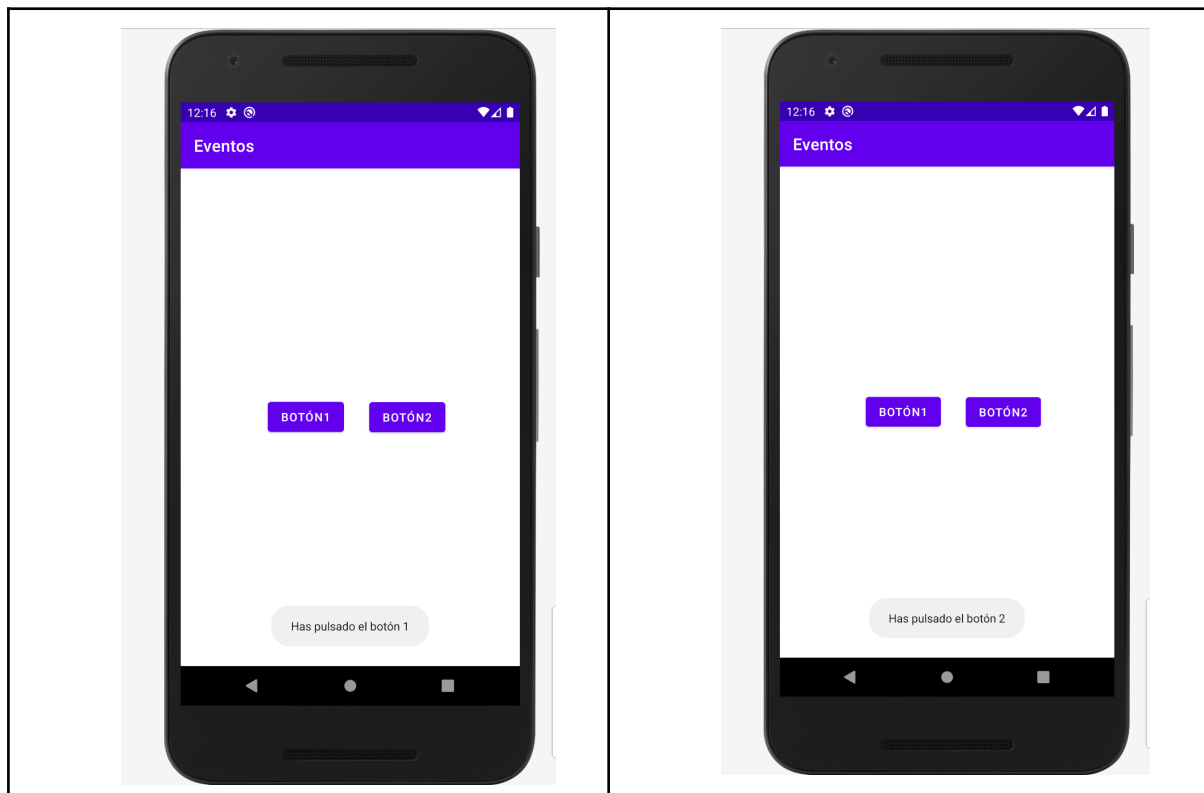
1. **Eventos:** son las acciones que los usuarios realizan sobre el dispositivo móvil o sobre algún objeto de la pantalla. (Ej. pulsar un botón, arrastrar un elemento por la pantalla, etc.)
2. **Escuchadores (Listeners):** son los componentes que gestionan los eventos. Normalmente existe un listener por evento. Se representan por clases abstractas donde el desarrollador debe implementar el método que recibirá el evento.
3. **Métodos** que registran los escuchadores: son los métodos que permiten registrar los listeners para que puedan ser utilizados. Puedes encontrarlos en la clase `android.view.View`. (Ej.: el método `setOnClickListener()` registra el listener `OnClickListener` que capturará los eventos cuando se pulsa sobre un elemento de la pantalla).

Para registrar los escuchadores podemos utilizar varias técnicas, pero vamos a centrarnos en las siguientes que seguiremos con un ejemplo muy sencillo.

1. **Implementación anónima:** creamos un objeto anónimo del escuchador con la implementación del evento dentro y su registro se hace desde la clase de la actividad. Al hacerlo de este modo, entran en juego lo que se conocen como lambdas. Te recomiendo este enlace para entender bien cómo se usan las lambdas en Kotlin.
2. **Implementación en la misma clase:** en este caso la clase de la actividad implementa la interfaz del escuchador y lo registramos desde aquí.

En el siguiente enlace puedes encontrar una descripción de cómo tratar los eventos en Android usando Kotlin: <https://developer.android.com/guide/topics/ui/ui-events?hl=es-419>
En el siguiente ejemplo de código, aparecen las dos formas disponibles para tratar el evento de click sobre un botón:

```
class MainActivity : AppCompatActivity(), View.OnClickListener {  
    override fun onCreate(savedInstanceState: Bundle?)  
    {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_main)  
  
        //Rescatamos el primer botón  
        val btn1: Button = findViewById(R.id.button) as Button  
        //Añadimos el evento de click en el botón1 a la lista de eventos a escuchar del MainActivity  
        btn1.setOnClickListener(this)  
        val btn2: Button = findViewById(R.id.button3) as Button  
        //Implementación anónima  
        btn2.setOnClickListener()  
        {  
            view -> Toast.makeText(context: this, text: "Has pulsado el botón 2", Toast.LENGTH_LONG).show()  
        }  
    }  
  
    //Este método se ejecuta al hacer click sobre el botón1  
    override fun onClick(p0: View?)  
    {  
        Toast.makeText(context: this, text: "Has pulsado el botón 1", Toast.LENGTH_LONG).show()  
    }  
}
```



4.4.2 Implementación anónima, lambdas en Kotlin, let y el operador Elvis

Implementación anónima

Una **implementación anónima** en Kotlin se refiere a la creación de una instancia de una clase o una interfaz *sin tener que declarar una clase separada que implemente esa interfaz o extienda esa clase*. En lugar de crear una clase con un nombre y luego implementar o extender la interfaz o clase deseada, puedes crear una implementación directamente en el lugar donde la necesitas de manera anónima.

Esto se logra utilizando **expresiones lambda o clases anónimas**, y es una característica poderosa y flexible en Kotlin que se utiliza para implementar interfaces o clases abstractas de manera concisa. Aquí hay un ejemplo de cómo se realiza una implementación anónima en Kotlin:

Supongamos que tienes una interfaz llamada **MiInterfaz**:

```
kotlin Copy code  
  
interface MiInterfaz {  
    fun miFuncion()  
}
```

Puedes crear una instancia anónima de esta interfaz y proporcionar una implementación para la función `miFuncion` de la siguiente manera:

```
kotlin Copy code  
  
val objetoAnonimo = object : MiInterfaz {  
    override fun miFuncion() {  
        // Implementación de la función miFuncion  
        println("Función miFuncion en una implementación anónima.")  
    }  
}  
  
// Llamando a la función miFuncion en la instancia anónima  
objetoAnonimo.miFuncion()
```

Expresiones Lambda en Kotlin

Una **expresión lambda** en Kotlin es una forma concisa de definir una función anónima que puede ser tratada como un valor. Las expresiones lambda se utilizan comúnmente para pasar código como argumento a funciones de orden superior, como *map*, *filter*, *forEach*, etc., o para definir interfaces funcionales y simplificar la escritura de código en situaciones donde solo se necesita una función temporal.

En Kotlin, la sintaxis básica de una **expresión lambda** es la siguiente:

kotlin

 Copy code


```
{ parámetros -> cuerpo de la lambda }
```

Aquí tienes una breve descripción de los elementos clave:

- **{}**: Los corchetes se utilizan para definir la expresión lambda.
- **parámetros**: Puedes especificar cero o más parámetros que se pasarán a la lambda. Si no tienes parámetros, puedes dejar los corchetes vacíos o utilizar un guión bajo (`_`) como marcador de posición.
- **cuerpo de la lambda**: Es el código que se ejecutará cuando la lambda se invoque. Puede consistir en una o más declaraciones.


Por ejemplo, aquí hay una expresión lambda que toma dos números y devuelve su suma:

kotlin

 Copy code

```
val suma = { a: Int, b: Int -> a + b }
```

kotlin

 Copy code

```
val resultado = suma(5, 3)
println(resultado) // Imprimirá 8
```

Otro ejemplo puede ser usando la estructura de repetición **forEach**:

```
kotlin
Copy code

val numeros = listOf(1, 2, 3, 4, 5)

// Usando forEach con una expresión lambda para imprimir los números
numeros.forEach { numero ->
    println(numero)
}
```

Let y el operador Elvis

Supongamos que deseas realizar una acción en una vista en Android, pero la vista podría ser nula. Puedes utilizar una expresión lambda con **let** para manejar esto de manera segura:

```
kotlin
Copy code

val vista: View? = obtenerVistaPosiblementeNula()

vista?.let {
    // Realizar alguna acción en la vista (it)
    it.setBackgroundColor(Color.RED)
} ?: run {
    // Acción a realizar si la vista es nula
    // Por ejemplo, mostrar un mensaje de error
    println("La vista es nula.")
}
```

- **it**: en Kotlin, **it** es una variable implícita que se utiliza en ciertos contextos, como *lambdas* o *funciones* de orden superior, para referirse al único parámetro de la función si la función tiene solo un parámetro. Es una abreviatura que se utiliza para simplificar el código.
- **View?**: el signo de exclamación (?) se utiliza para indicar que la variable **View** puede tomar el valor **nulo**.
 - **View!**: el signo de exclamación (!) se utiliza para indicar que la variable **View** es de tipo **no nulo**. Si un tipo de dato termina con un signo de exclamación, significa que el valor no puede ser nulo y debe contener un valor válido.

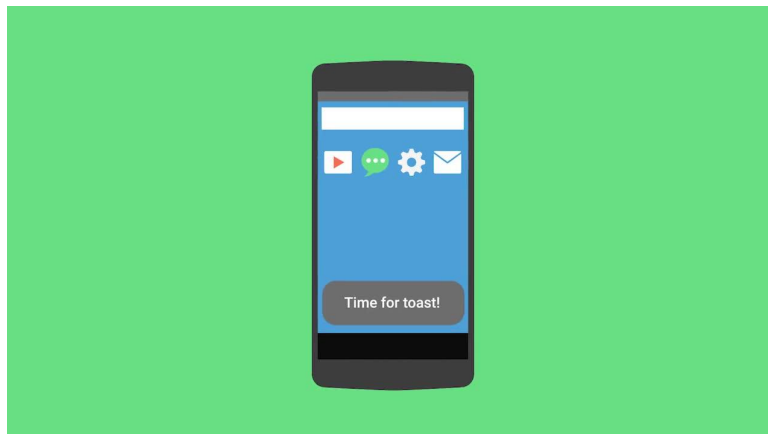
- **?:** el operador **Elvis** se utiliza en Kotlin para proporcionar un valor de respaldo en caso de que la expresión a la izquierda sea nula. Si **it** (en este contexto) es nulo, se utilizará un valor de respaldo.

4.5 Toast

Son mensajes que se muestran en pantalla durante sólo unos segundos para volver a desaparecer automáticamente, sin requerir ningún tipo de actuación.

En este enlace tienes la documentación oficial:

<https://developer.android.com/guide/topics/ui/notifiers/toasts?hl=es-419>



Para lanzar un mensaje de este tipo, se llama al método `makeText` desde la clase `Toast`. El primer argumento es el contexto, el segundo el texto a mostrar y el tercero el tiempo que quieres que se muestre. Este tiempo puede ser corto (**`Toast.LENGTH_SHORT`**) o largo (**`Toast.LENGTH_LONG`**).

Finalmente, debes añadir al final el método **`show()`** para que se muestre.

```
Toast.makeText( context: this, text: "Has pulsado el botón 1", Toast.LENGTH_LONG).show()
```

!! NOTA IMPORTANTE !!

El **"contexto"** en una aplicación Android se refiere a un objeto que proporciona información sobre el entorno en el que se encuentra la aplicación. El contexto es una parte fundamental en el desarrollo de aplicaciones **Android** y se utiliza para acceder a **recursos** y **servicios del sistema**, como archivos de recursos, bases de datos, preferencias compartidas y más. Es esencial comprender el contexto para interactuar adecuadamente con el sistema operativo Android.

En Android, el contexto generalmente se representa mediante objetos de la clase **Context**. Esta clase es una superclase de varias otras clases relacionadas con la aplicación, como **Activity**, **Service**, **Application**, entre otras. Por lo tanto, en la mayoría de los casos, puedes obtener un contexto de estas clases o subclases.

El contexto **proporciona información sobre aspectos** como la *ubicación* de la *aplicación* en el *sistema de archivos*, los *recursos disponibles*, el *ciclo de vida* de la aplicación y permite realizar diversas operaciones, como *iniciar actividades*, *crear vistas*, acceder a preferencias compartidas y mucho más.

Aquí hay algunas situaciones en las que se utiliza el contexto en una aplicación Android:

1. **Acceso a recursos:** puedes usar el contexto para acceder a recursos como cadenas, imágenes y diseños almacenados en archivos de recursos (res).
2. **Iniciar actividades:** para iniciar una nueva actividad, necesitas un contexto. Puedes usar un *intent* y el *contexto* para abrir otras actividades.
3. **Acceso a preferencias compartidas:** para acceder a las preferencias compartidas de tu aplicación, necesitas un contexto.
4. **Acceso a servicios del sistema:** el contexto se utiliza para acceder a servicios del sistema, como el administrador de notificaciones, el administrador de ubicación y otros.
5. **Crear vistas personalizadas:** algunas vistas personalizadas pueden requerir un contexto para su creación.
6. **Manejo de bases de datos:** cuando trabajas con bases de datos en Android, necesitas un contexto para crear o acceder a la base de datos.

En resumen, el contexto es una parte esencial en el desarrollo de aplicaciones Android, ya que proporciona información sobre el entorno de la aplicación y permite interactuar con los recursos y servicios del sistema de manera efectiva. Es importante entender cómo y cuándo utilizar el contexto en tu aplicación para un desarrollo adecuado.

4.6 Etiquetas, cuadros de texto e imágenes

Etiquetas (TextView)

Se empezará por las etiquetas ([TextView](#)), son la forma más sencilla de mostrar información al usuario dentro de un *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.

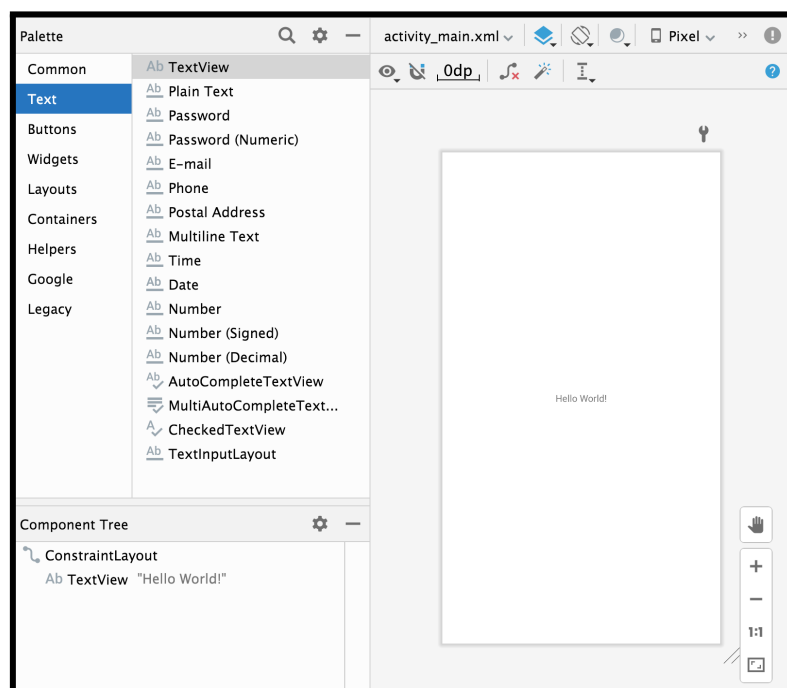
```
<TextView
    android:id="@+id/TextView1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Hello World!"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintTop_toTopOf="parent"/>
```

Algunas de las propiedades más útiles son:

- **autolink**: se utiliza para crear enlaces.

```
android:autoLink=""
android:visibility="web"
android:layout_width="all"
android:layout_height="email"
android:text="http://www.google.com"
app:layout_constraintBottom_toBottomOf="parent"
app:layout_constraintStart_toStartOf="parent"
app:layout_constraintEnd_toEndOf="parent"
app:layout_constraintTop_toTopOf="parent" android:enabled="true"/>
```

- **visibility (visible/invisible)**: permite mostrar u ocultar la etiqueta.
- **text**: permite obtener y modificar el texto de la etiqueta.



Cuadros de texto (EditText)

Se utilizan para pedir información al usuario. Actualmente, material design recomienda usar ***TextInputEditText*** dentro de un bloque ***TextInputLayout***.

```
<com.google.android.material.textfield.TextInputLayout
    android:id="@+id/textField"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:hint="@string/label">

    <com.google.android.material.textfield.TextInputEditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"

    />

</com.google.android.material.textfield.TextInputLayout>
```

Uno de los atributos más interesantes dentro de *TextInputEditText* es [*inputType*](#). Éste permite identificar el tipo de texto esperado en ese cuadro de texto.

```
android:inputType="textPassword"
```

Imágenes (ImageView)

Permiten mostrar imágenes o iconos en las activities.

Las imágenes deben estar añadidas en el proyecto (*res/drawable*) u obtenerse desde un recurso de Internet.

```
<ImageView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    app:srcCompat="@drawable/baseline_content_paste_24"
    android:id="@+id/imageView"
    app:layout_constraintTop_toBottomOf="@+id/password"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"/>
```

El atributo ***srcCompat*** indica la ruta hasta la imagen.

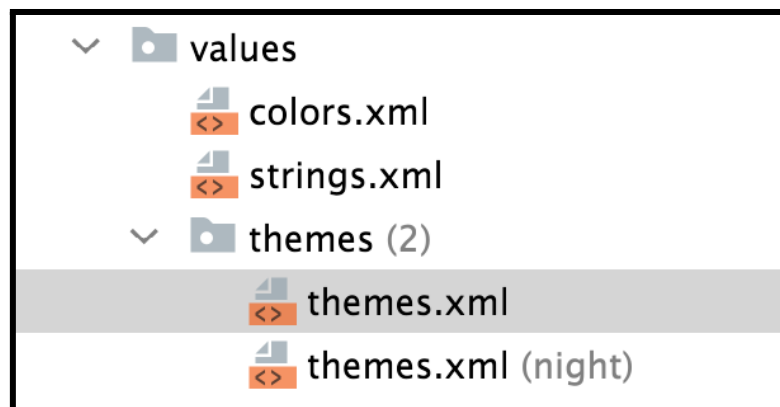
4.6.1 Internacionalización y localización

1. Temas y estilos

La apariencia de una aplicación es un detalle muy importante a tener en cuenta para que tenga éxito o no. Hay que cuidar por ejemplo los colores que se utilizan como fondo o para el tipo de fuente, el tamaño, etc.

Un **estilo** y un **tema** son dos conceptos muy similares, ya que ambos recopilan las características de apariencia común a los objetos a los que se vayan a aplicar. La única diferencia es que un estilo se puede asociar a objetos de tipo **View** mientras que un tema se asocia a objetos de tipo **Activity**.

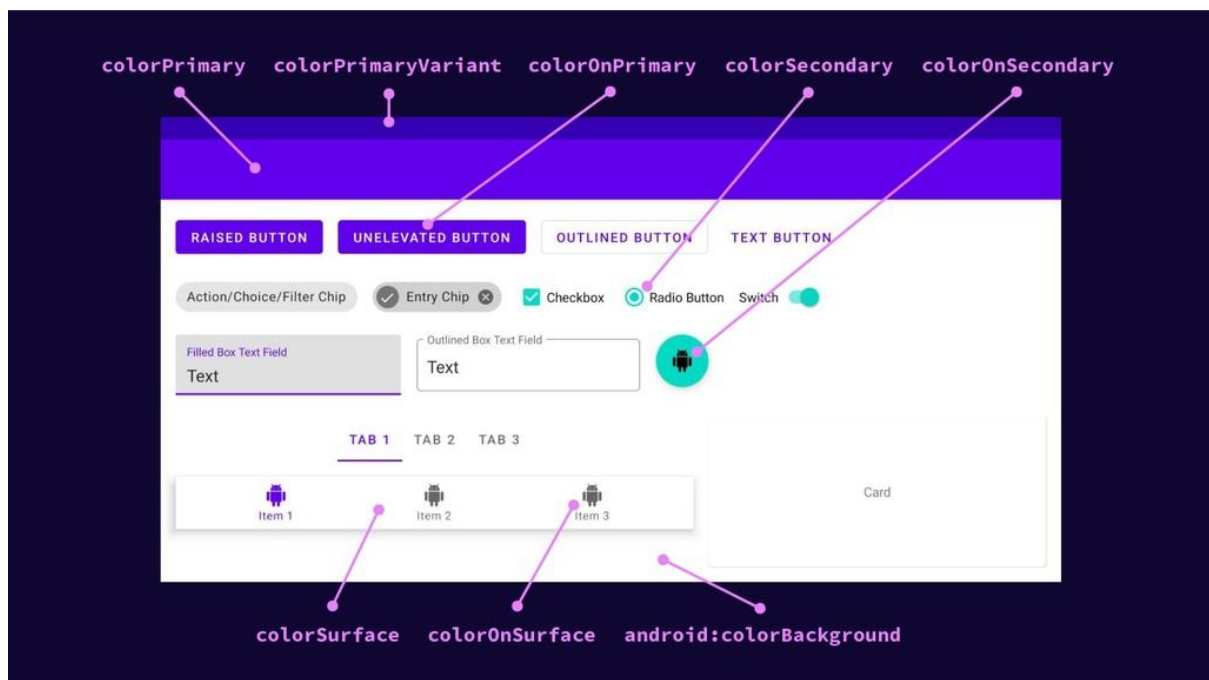
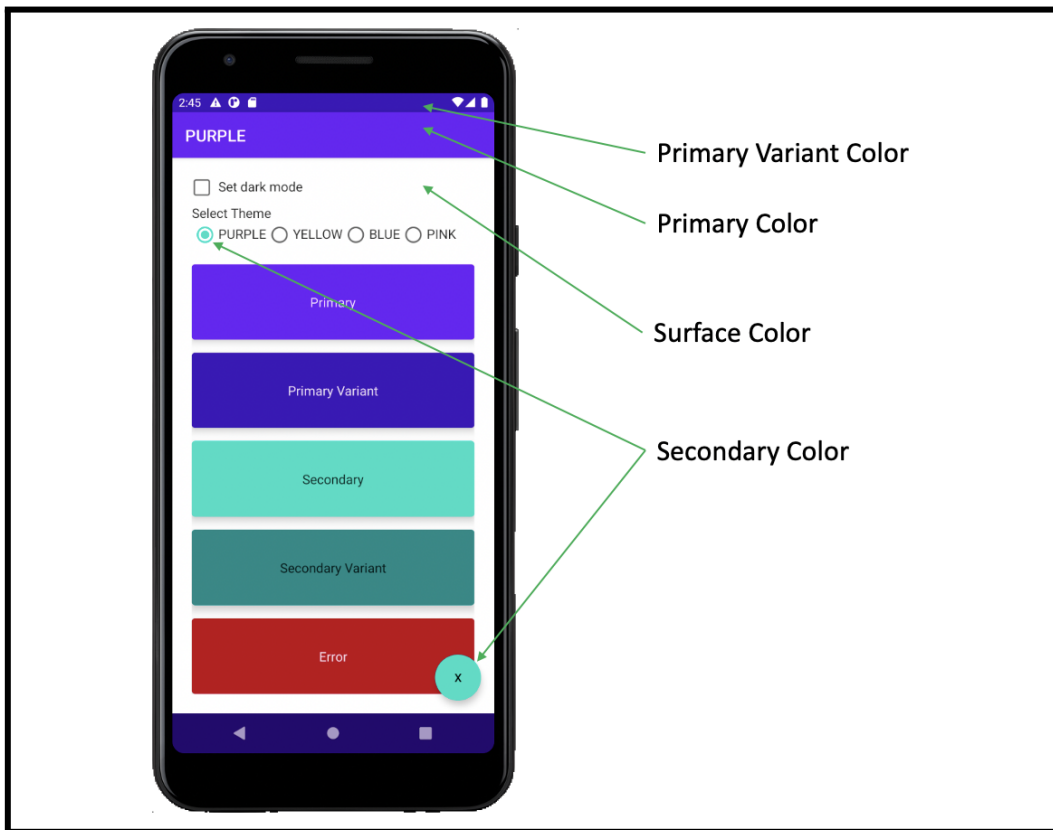
Los estilos se almacenan en la carpeta de recursos **/res/values/** en la carpeta **/themes/**. Existen estilos ya predefinidos y el desarrollador puede crear otros nuevos heredando de un estilo padre y creando los ítems correspondientes a los atributos XML.



```
<resources xmlns:tools="http://schemas.android.com/tools">
    <!-- Base application theme. -->
    <style name="Theme.Tutoria25_10" parent="Theme.MaterialComponents.DayNight.DarkActionBar">
        <!-- Primary brand color. -->
        <item name="colorPrimary">@color/purple_500</item>
        <item name="colorPrimaryVariant">@color/purple_700</item>
        <item name="colorOnPrimary">@color/white</item>
        <!-- Secondary brand color. -->
        <item name="colorSecondary">@color/teal_200</item>
        <item name="colorSecondaryVariant">@color/teal_700</item>
        <item name="colorOnSecondary">@color/black</item>
        <!-- Status bar color. -->
        <item name="android:statusBarColor" tools:targetApi="l">?attr/colorPrimaryVariant</item>
        <!-- Customize your theme here. -->
    </style>
</resources>
```


En los estilos son muy interesantes los parámetros sobre los colores de la aplicación. Los colores se definen en el fichero `res/values/colors.xml`

En las siguientes imágenes podemos ver cómo se aplican a una interfaz de usuario:



Documentación oficial: <https://developer.android.com/develop/ui/views/theming/themes>

Cómo implementar un tema (material.io):

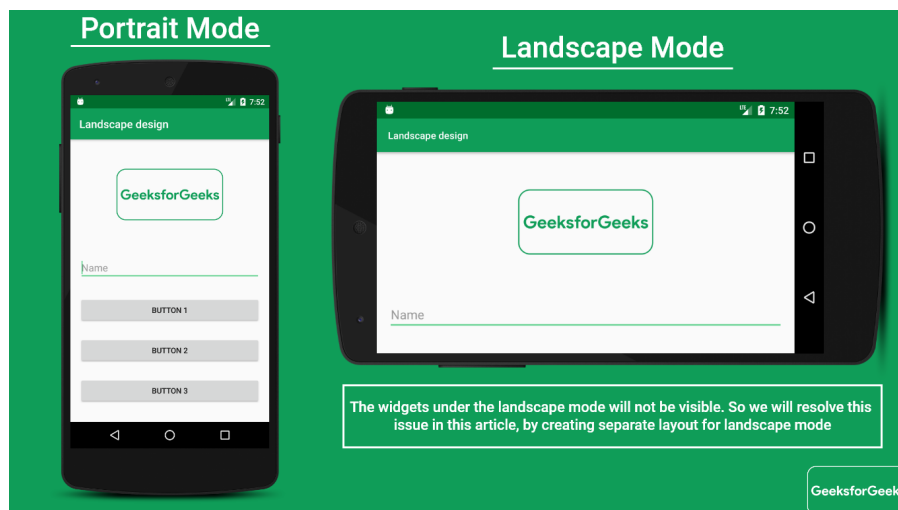
<https://m2.material.io/design/material-theming/implementing-your-theme.html#color>

2. Vista vertical (portrait) y vista horizontal (landscape)

En este apartado vamos a ver cómo crear en nuestra app, vista vertical (portrait) y una vista horizontal (landscape) distinta.

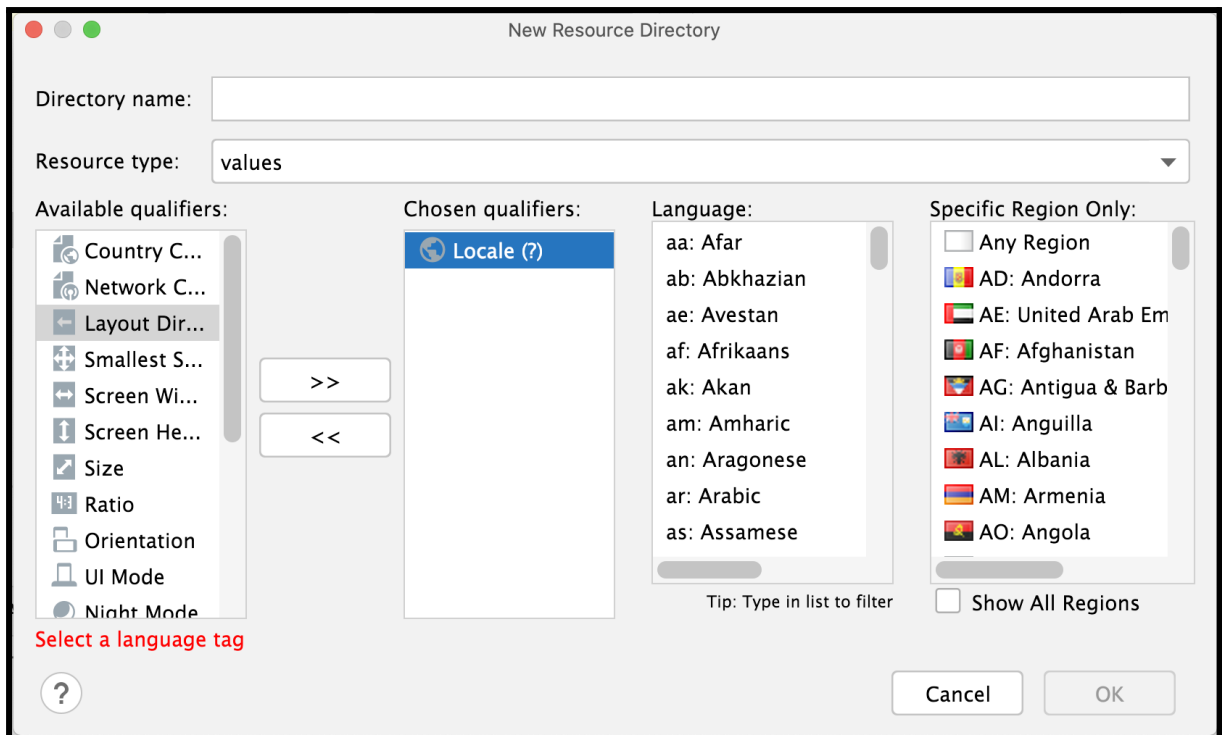
En el siguiente tutorial tienes explicados los pasos con detalle:

<https://www.geeksforgeeks.org/designing-the-landscape-and-portrait-mode-of-application-in-android/>

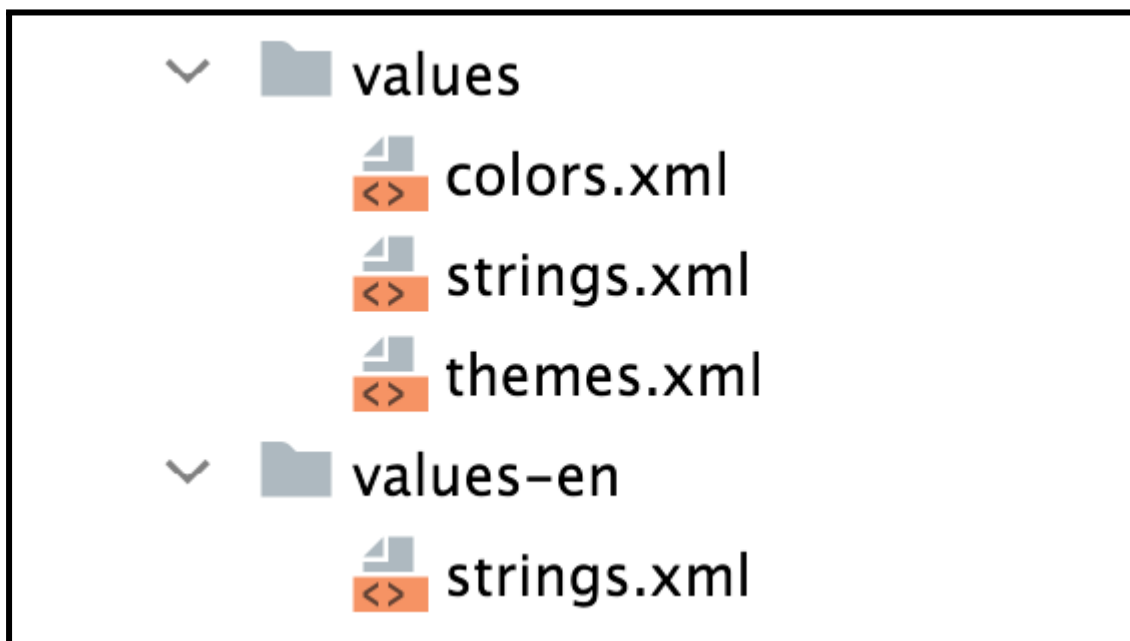


3. Idiomas

Los pasos son los mismos que en el caso anterior, pero ahora cuando elijamos el recurso de Android Studio, debemos elegir **“Locale”**, y a continuación, el idiomas que deseemos:



En nuestro proyecto se verán las dos carpetas así, después de copiar el fichero strings.xml a la nueva carpeta `values-en`:



4.7 SnackBar

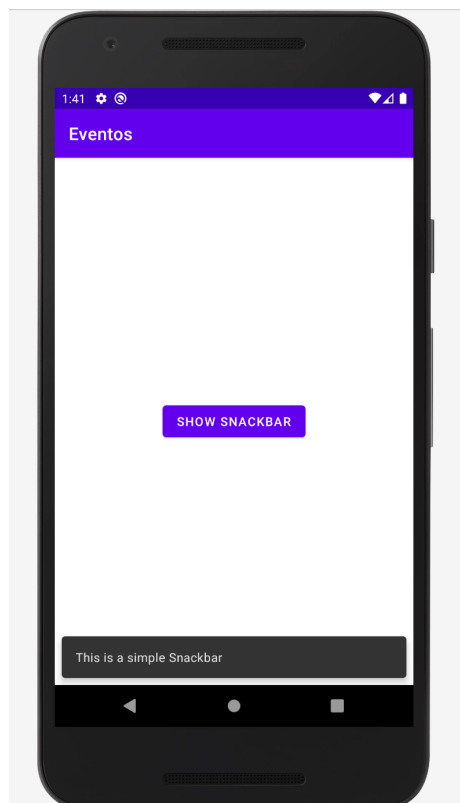
Es parecido a un Toast, pero aparece como una barra en la parte inferior.

Puedes encontrar cómo usarlos en la web de **material.io**:
<https://m3.material.io/components/snackbar/overview>

Se incluye un ejemplo de código, lanzando el **snackbar** después de pinchar en un botón:

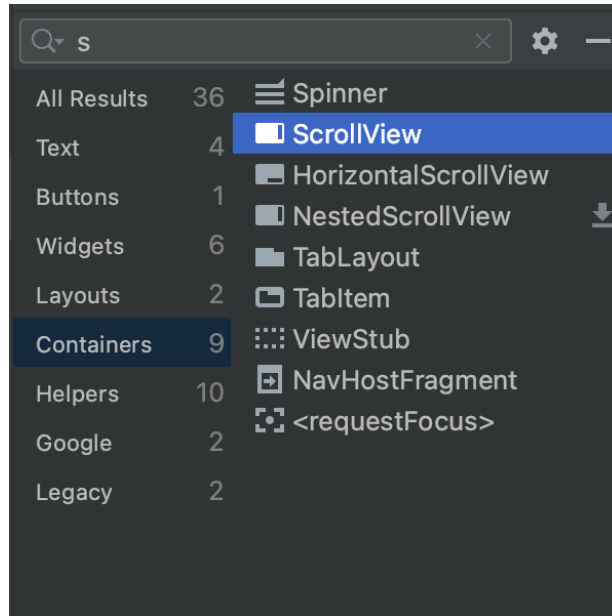
```
//Rescatamos el botón
val btn4: Button = findViewById(R.id.button4)

btn4.setOnClickListener { it: View!
    val snack = Snackbar.make(it,
        text: "This is a simple Snackbar", Snackbar.LENGTH_LONG)
    snack.show()
}
```



4.8 ScrollView

El [ScrollView](#) en Android te permite albergar una jerarquía de views con el fin de desplazar su contenido a lo largo de la pantalla, cuando sus dimensiones exceden el tamaño de la misma.



Os dejo un [vídeo](#) donde explican muy bien cómo usarlo.