U3: GENERACIÓN DE INTERFACES. DISEÑO NATIVO. ANDROID STUDIO

OBJETIVOS

- Aprender a desarrollar interfaces nativas a través de la herramienta Android Studio.

RESULTADOS DE APRENDIZAJE

RA1. Genera interfaces gráficos de usuario mediante editores visuales utilizando las funcionalidades del editor y adaptando el código generado.

CRITERIOS DE EVALUACIÓN

a) Se han identificado las herramientas para diseño y prueba de componentes.

b) Se ha creado un interfaz gráfico utilizando las herramientas de un editor visual.

c) Se han utilizado las funciones del editor para ubicar los componentes del interfaz.

d) Se han modificado las propiedades de los componentes para adecuarlas a las necesidades de la aplicación.

e) Se ha analizado el código generado por el editor visual.

f) Se ha modificado el código generado por el editor visual.

g) Se han asociado a los eventos las acciones correspondientes.

h) Se ha desarrollado una aplicación que incluye el interfaz gráfico obtenido.

TABLA DE CONTENIDO

U3: GENERACIÓN DE INTERFACES. DISEÑO NATIVO. ANDROID STUDIO 1

1. JETPACK COMPOSE 3
2. CREACIÓN DE UN PROYECTO CON JETPACK COMPOSE 3
3. USO DEL EMULADOR 5
4. ARQUITECTURA BÁSICA DEL PROYECTO 6
5. PREVIEW DEL PROYECTO 9
6. MODIFICADORES EN COMPOSE 11
7. COMPONENTES BÁSICOS 12

TEXT 12

TEXTFIELD 13

OUTLINEDTEXTFIELD 15

BOTONES 16

8. LAYOUT EN COMPOSE 17

BOX 18

COLUMN 20

ROW 23

9. COMBINANDO LAYOUTS 24

1. JETPACK COMPOSE

Es un framework de **UI declarativa** para Android desarrollado por Google que permite crear interfaces de usuario de manera más eficiente y fluida. A diferencia del enfoque tradicional de diseño de interfaces basado en XML, Compose utiliza un enfoque declarativo que simplifica la actualización de la UI.

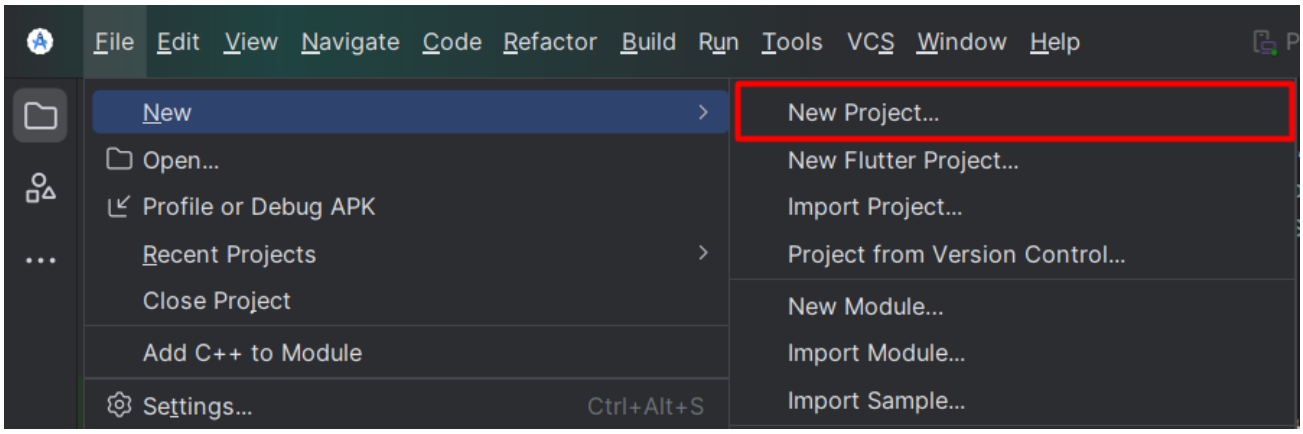
Jetpack Compose permite generar las interfaces en los archivos de kotlin (.kt) sin tener que enlazarlos con un .xml y así manejar interfaz/lógica en un mismo archivo.

- **Declarativo.** Definiremos como queremos la interfaz en un estado determinado.
- **Sincronización automática.** La UI se sincroniza con los datos directamente, sin tener que hacer cambios manuales. Cuando un dato que pertenece a un componente cambia, se actualizará sólo ese componente, lo que mejora el rendimiento y simplifica la lógica.
- **Composables.** Las funciones que generan partes de una interfaz se anotarán como @Composable. Los llamaremos los componentes.

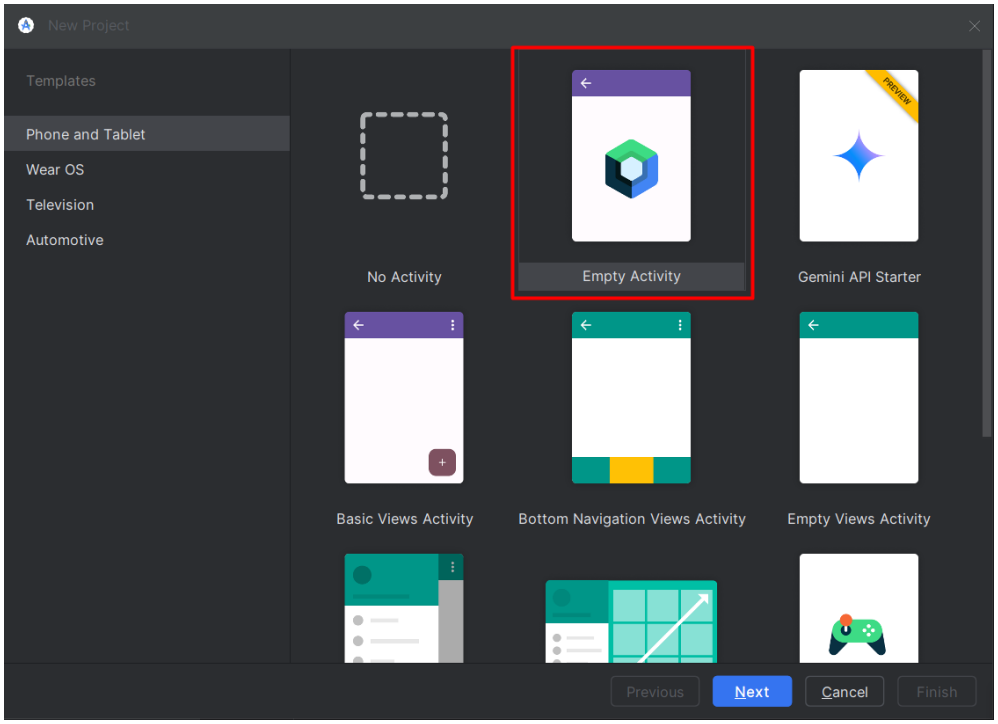
```kotlin
@Composable
fun Greeting(name: String) {
    Text(text = "Hello, $name!")
}
```

2. CREACIÓN DE UN PROYECTO CON JETPACK COMPOSE

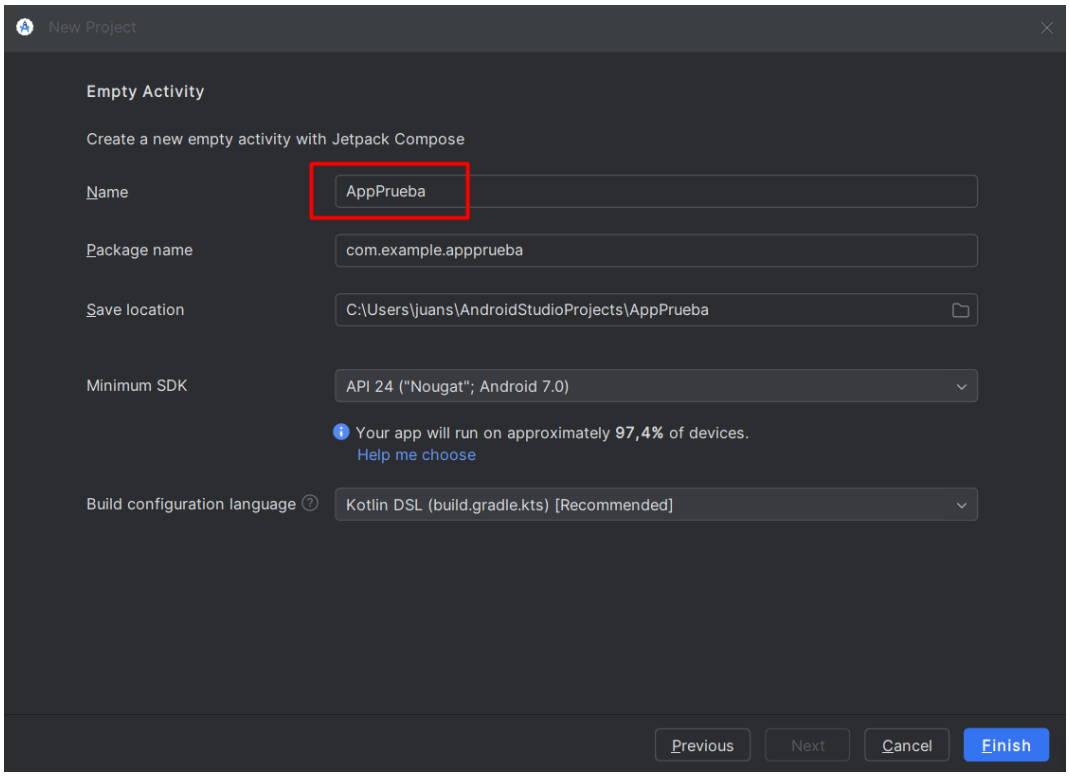
Una vez instalado Android Studio, procederemos a crear nuestro primer proyecto, para eso, iremos a la parte superior izquierda de la pantalla, “New -> New project…”



Elegiremos Empty Activity para crearlo

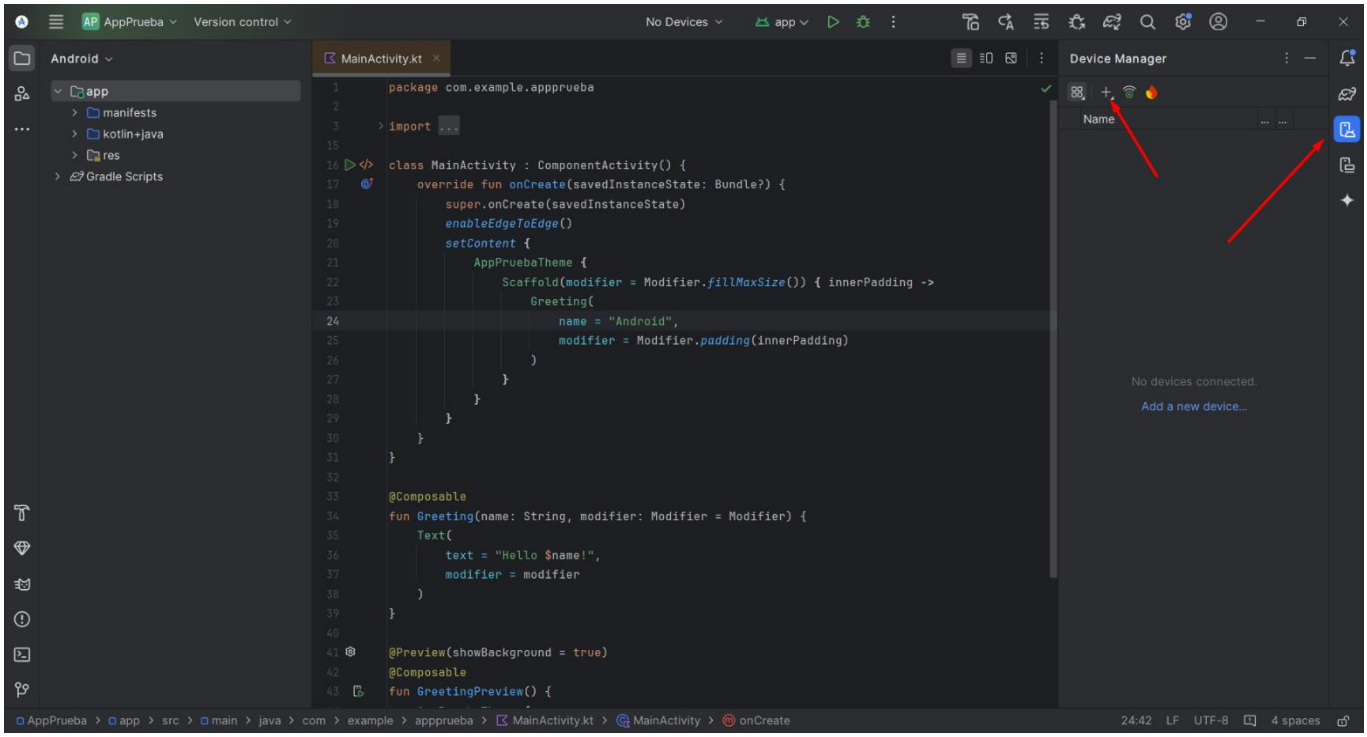


Cambiaremos el nombre para que sea acorde con el proyecto que vamos a realizar y daremos a Finish.

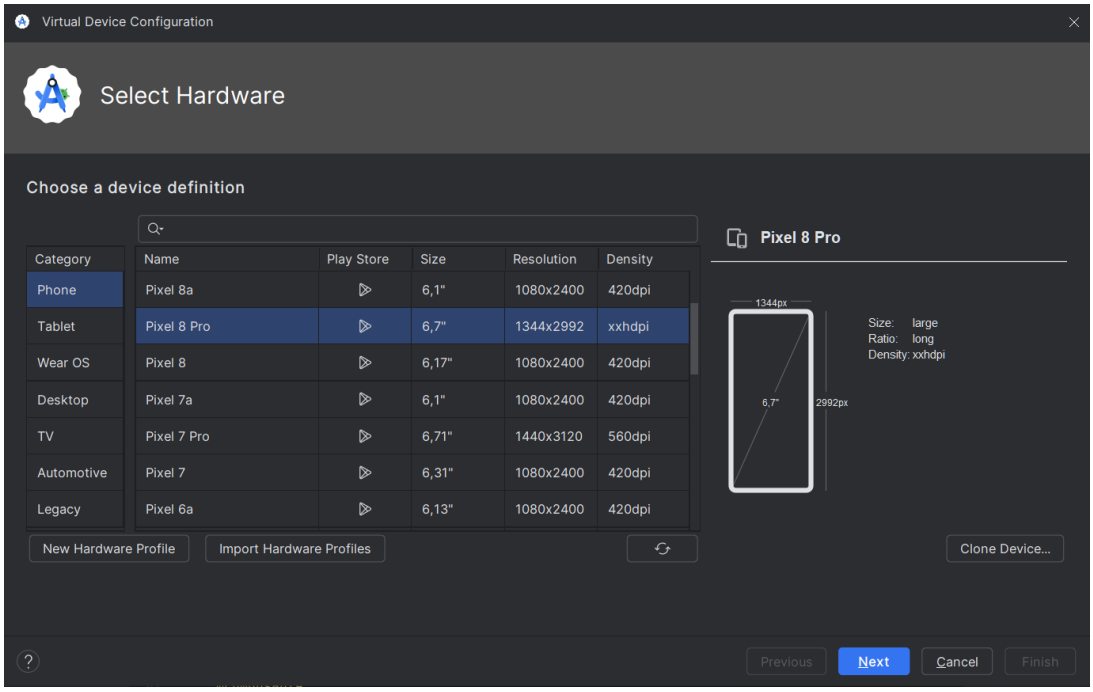


3. USO DEL EMULADOR

Para poder ver el resultado de nuestras aplicaciones mediante un emulador móvil en Android Studio, deberemos configurarlo. A la derecha de la pantalla principal, veremos el botón “Device Manager”, desde donde podremos gestionar los dispositivos que queremos emular.

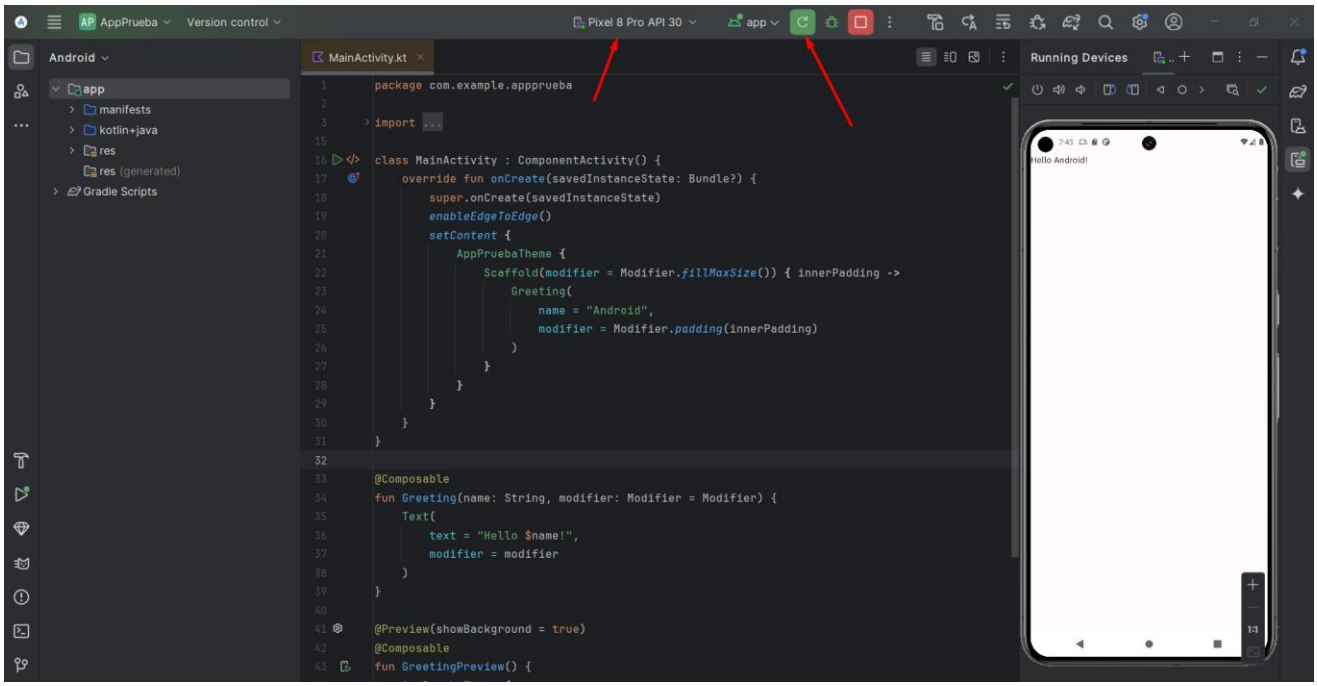


De la lista que nos aparece, elegiremos el dispositivo que necesitamos, en este caso, móvil.



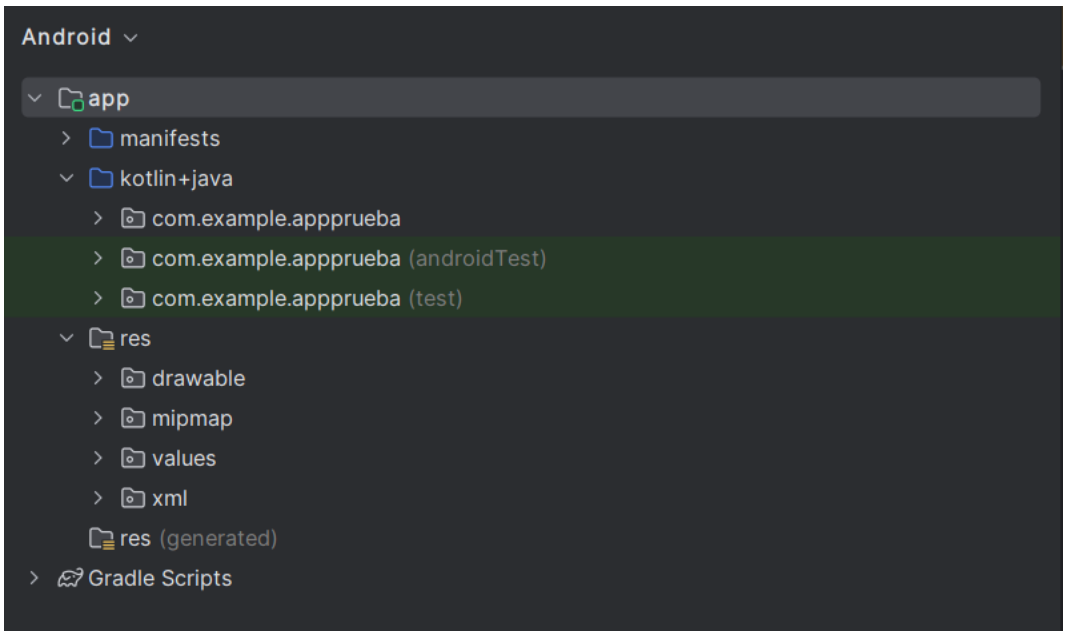
Una vez añadido, nos aparecerá en la parte superior de la pantalla. Vamos a ejecutar el proyecto por defecto que nos da Android Studio y así probar que efectivamente, el emulador funciona.

Daremos a “Run” en la parte superior.



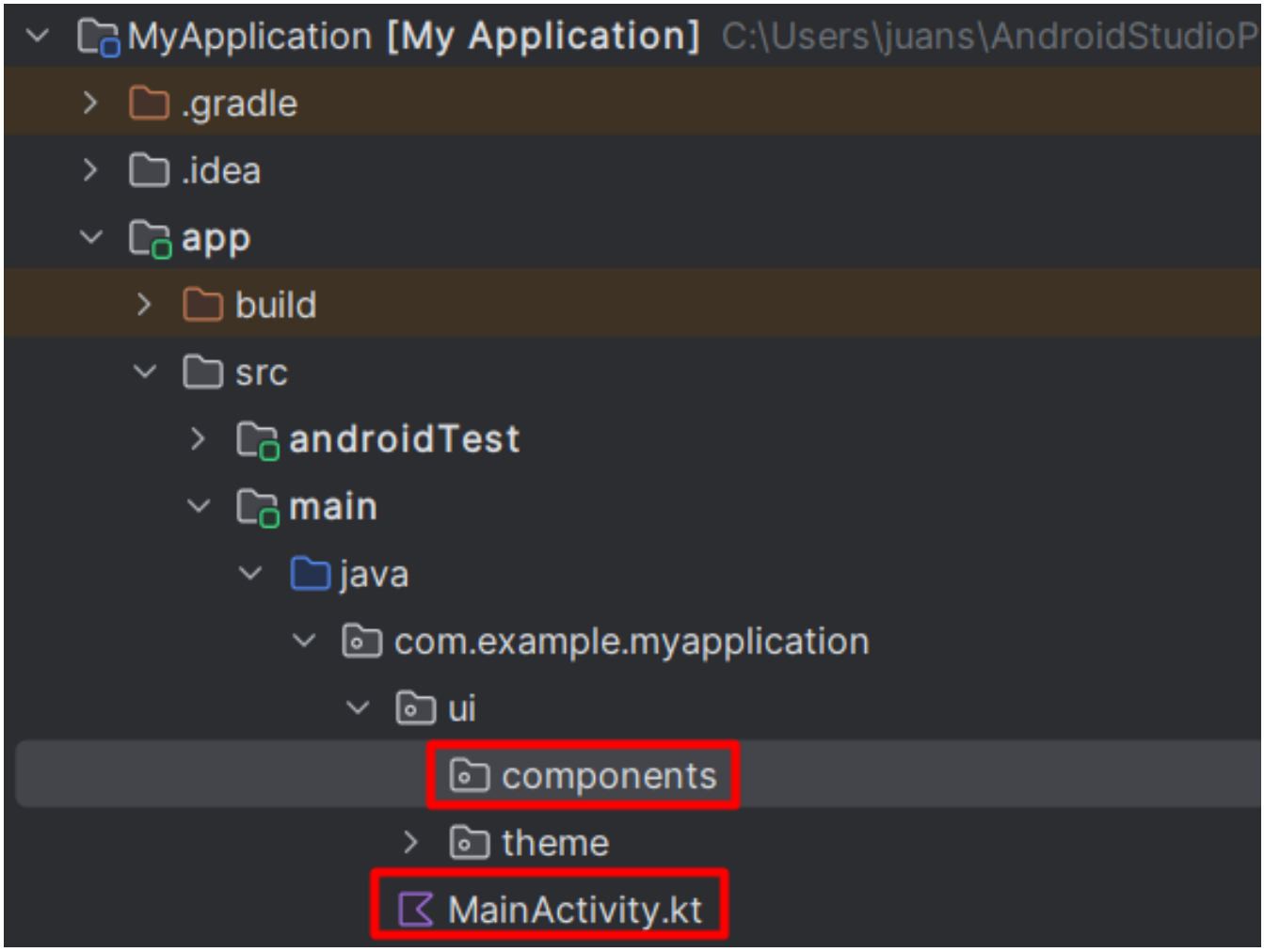
4. ARQUITECTURA BÁSICA DEL PROYECTO

Al crear el proyecto, podremos ver una serie de archivos y estructura que nos genera Android Studio por defecto.



- **Manifest:** Contiene el archivo AndroidManifest.xml. Contiene configuración e información esencial de la aplicación.
- **Kotlin+java:**
    - Com.example.appprueba: Código kotlin de nuestra aplicación
    - Com.example.apprueba (androidTest): Pruebas de la aplicación sobre un emulador Android.
    - Com.example.appprueba: Pruebas unitarias sobre el código.
- **Res**: Contendrá los recursos de la aplicación
    - Drawable: Imágenes, gráficos…
    - Mipmap: Iconos.
    - Values: Valores estáticos como cadenas de text, colores…
    - Xml: Archivo de configuración
- **Gradle Script**: Manejo de la construcción y dependencias del proyecto.

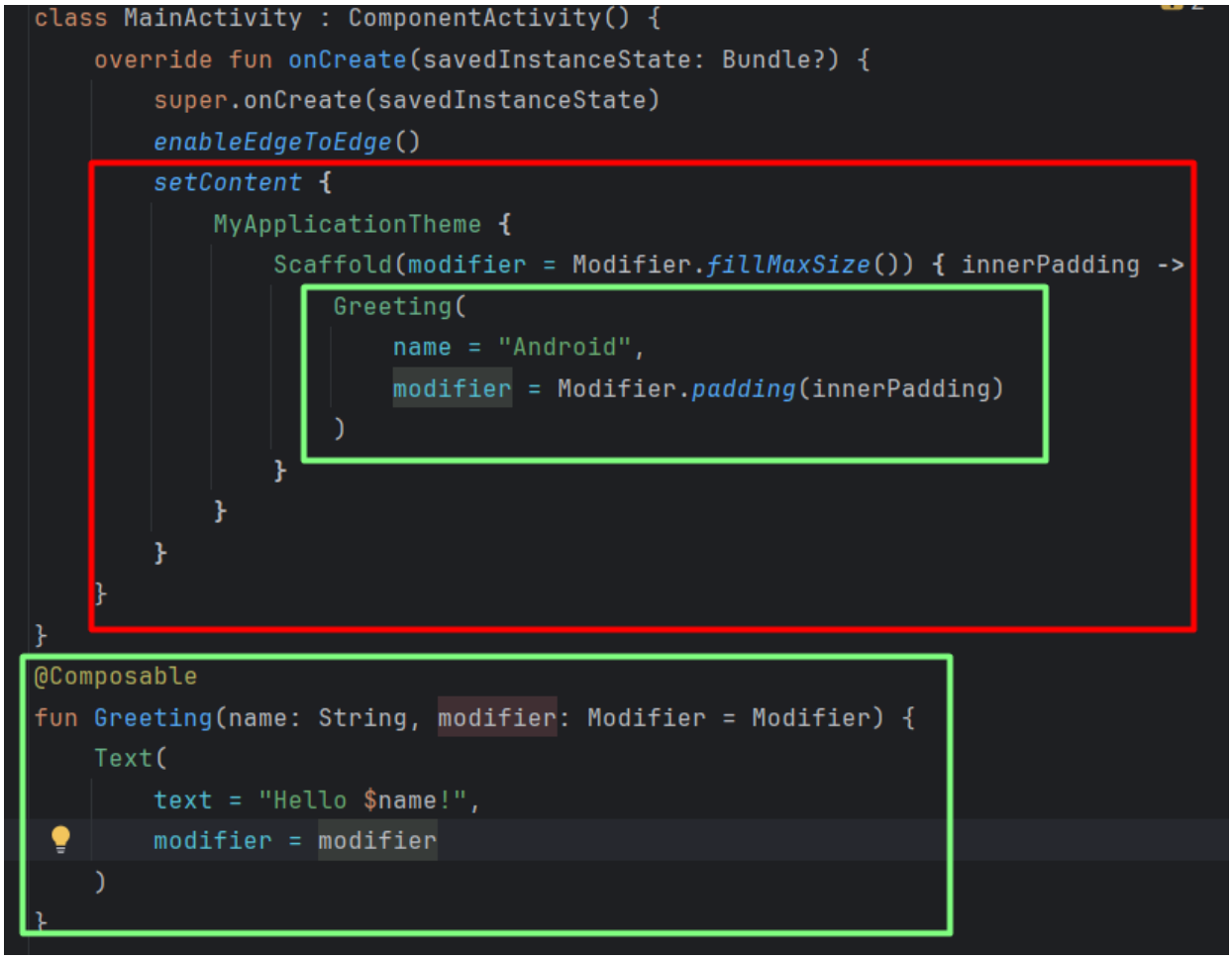
La parte en la que nos enfocaremos ahora será *com.example.appprueba* (este nombre cambiará con el nombre de tu proyecto). Aquí crearemos nuestras pantallas y componentes.



MainActivity.kt será la primera pantalla que tengamos en nuestra aplicación. Siempre que creemos una pantalla nueva, la llamaremos como "FuncionalidadActivity.kt".

Components es un paquete que deberemos crear nosotros y será donde añadiremos todos los componentes creados. Un componente puede ser tan pequeño como un simple botón o tan grande como una pantalla entera.

Al crear el proyecto, en MainActivity.kt veremos que hay código de ejemplo.



De momento, en la clase MainActivity nos centraremos sólo en **SetContent** y lo que está dentro de **Scaffold.**

Lo que haya dentro de **SetContent** será por donde empiece a cargar la aplicación.

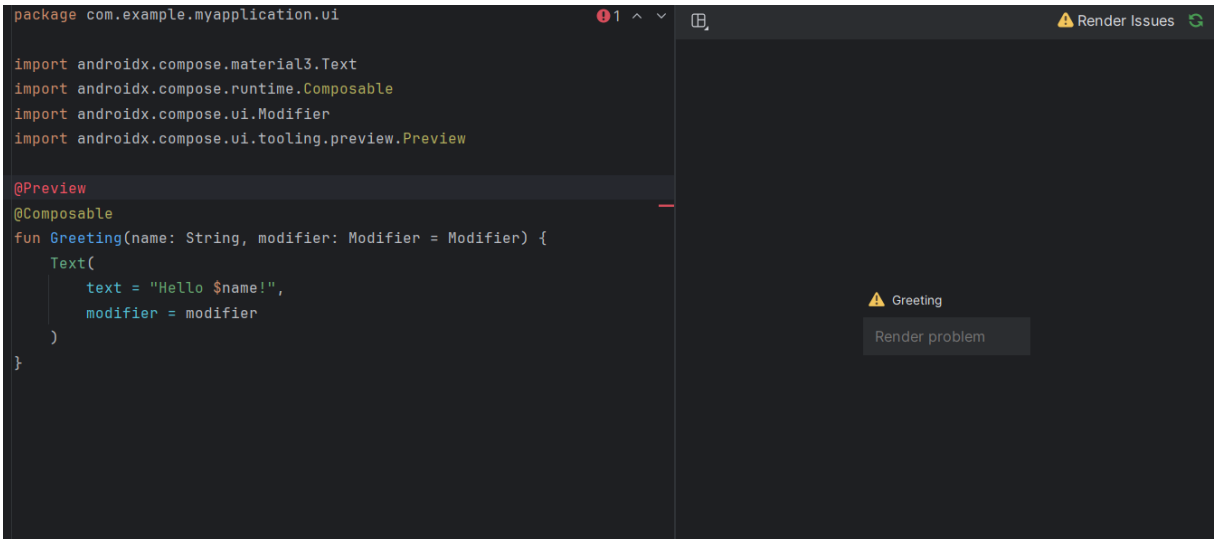
Vemos que hay una función "Greeting" con dos parámetros, name y modifier. Esa función está declarada abajo con esos parámetros y está anotada como @Composable, lo que nos dice que esa función devuelve un componente, en este caso, de tipo ***Text.*** Si ejecutamos, tendremos el resultado que esperamos, una pantalla sin nada más que un texto con "Hello Android".

5. PREVIEW DEL PROYECTO

Con tal de poder ir viendo el proyecto antes de ejecutarlo, con Composable tenemos la opción de las vistas **“Code”, “Split”, “Desing”.**

- **Code**: Para una vista únicamente del código
- **Split**: Vista de código y diseño
- **Desing**: Para ver únicamente el diseño

Para poder ver en la pantalla de diseño una función anotada con @Composable, tenemos que añadir la anotación @Preview.



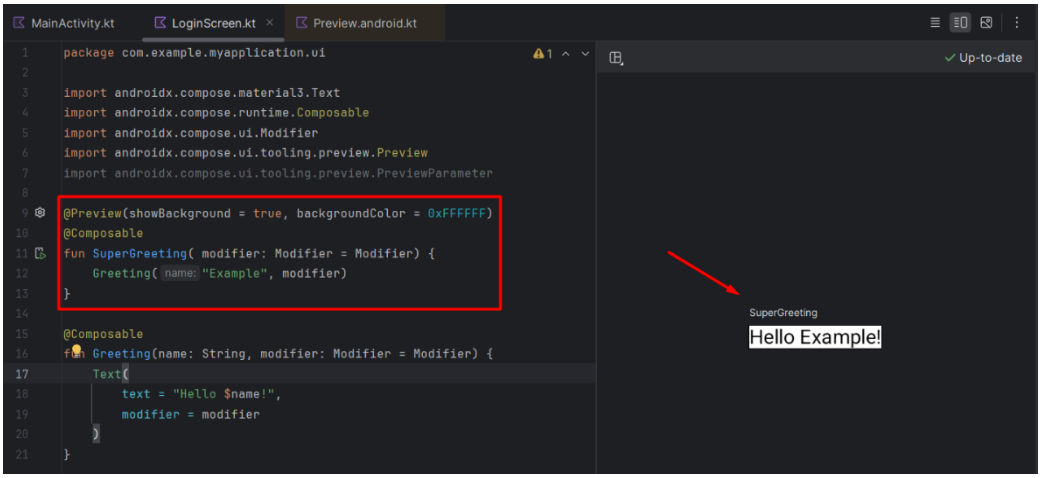
En este caso, nos aparecerá @Preview, indicándonos que existe un problema.

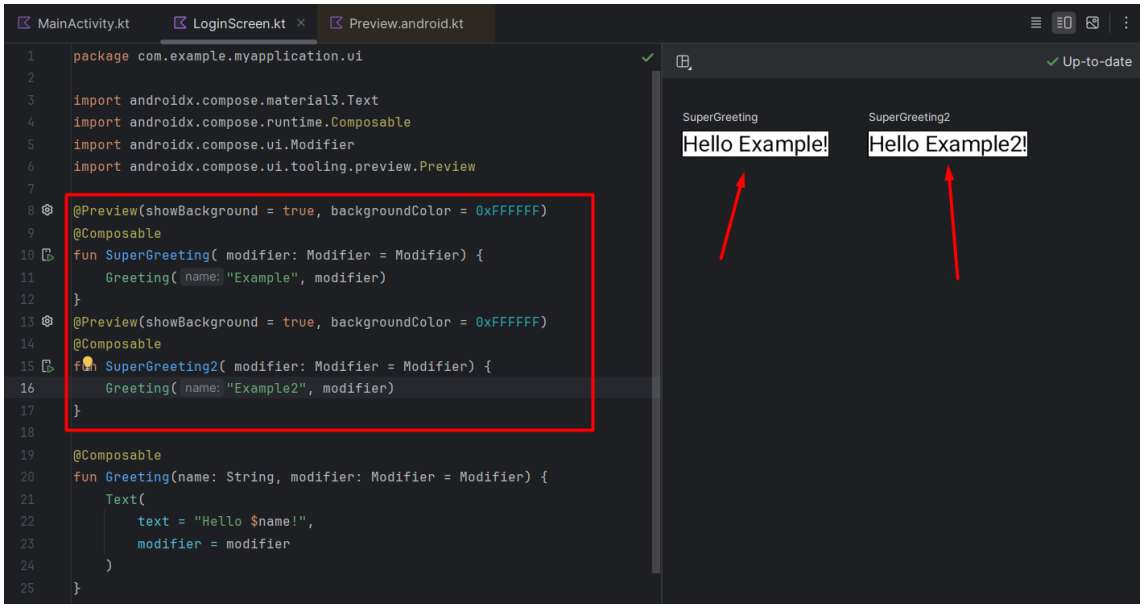
Nos indica que no puede renderizar una vista de un composable que no tiene parámetros estáticos, ya que no hay ninguna ejecución y no sabría como representar esa vista.

Podemos solucionarlo creando una “superFuncion”, que llamará a nuestra función real, con parámetros estáticos, en este caso “Example” para el nombre.

La @Preview tiene también dos parámetros, showBackground y backGroundColor, que nos dejará definir el color del fondo, que por defecto sería negro.

La función de Preview nos permite previsualizar varios componentes si quisiéramos:

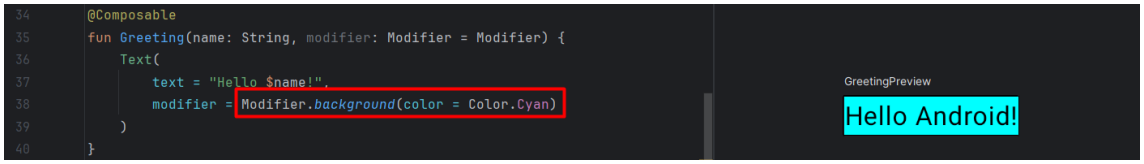




6. MODIFICADORES EN COMPOSE

Al crear el proyecto, en el archivo “MainActivity.kt” tendremos una función, con el componente “Text”.

Observamos que, para la creación del Text, a parte del parámetro *text*, que será la cadena de texto que mostrará, tiene un *modifier*. Este parámetro es muy importante en Compose ya que podremos utilizarlo en todos los componentes y es la forma que tenemos de definir su estilo (ancho, alto, espaciado…).



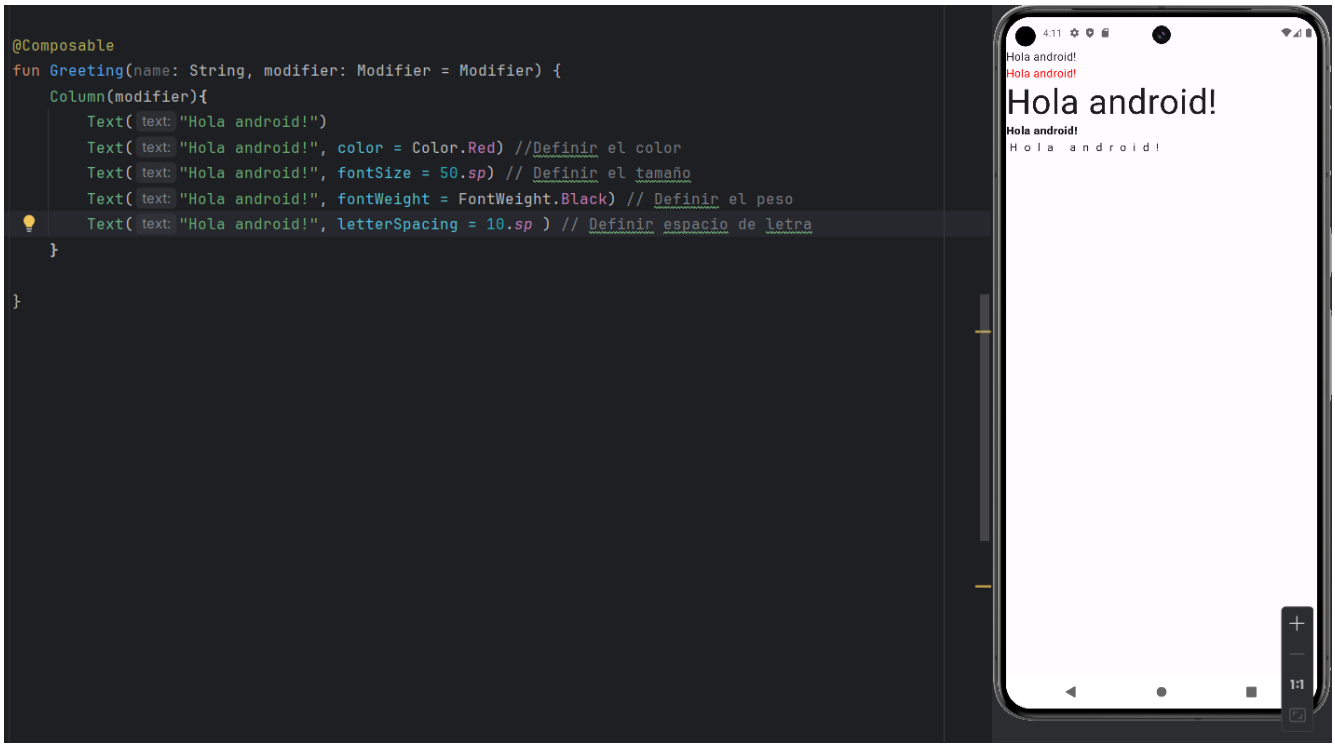
En el caso anterior, hemos añadido que el color de fondo sea Cyan.

7. COMPONENTES BÁSICOS

TEXT

Será el componente que más utilicemos y, como su nombre indica, contendrá el texto que queramos mostrar en las pantallas.

A continuación, se muestran varios (que no todos) modificadores que podremos utilizar para dar formato a nuestro texto. Se han introducido los Text en una Columna, que se verá más tarde en este documento, para mejor visibilidad de los elementos.



TEXTFIELD

Con TextField podremos controlar las entradas de texto del usuario, para su uso, deberemos de añadir un par de imports si no nos lo hace el asistente por defecto.

```
import androidx.compose.material3.TextField

import androidx.compose.runtime.mutableStateOf

import androidx.compose.runtime.remember

import androidx.compose.runtime.getValue

import androidx.compose.runtime.setValue
```

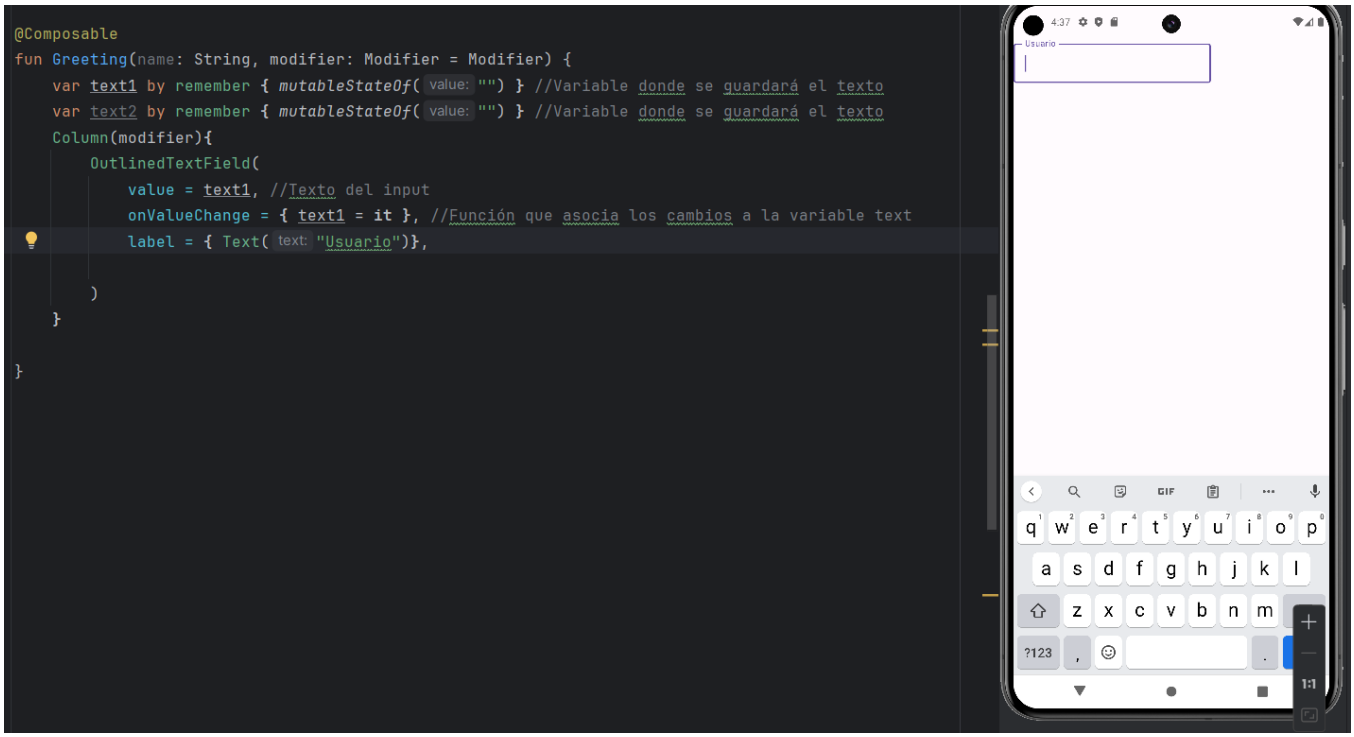
```
@Composable
fun Greeting(name: String, modifier: Modifier = Modifier) {
    var text by remember { mutableStateOf( value: "Input") } //Variable donde se guardará el texto
    Column(modifier){
        TextField(
            value = text, //Texto por defecto en el input
            onValueChange = { text = it }, //Función que asocia los cambios a la variable text
            label = { Text( text: "Label")} // Label que aparecerá en el input
        )
    }

}
```

Algunas modificaciones que podemos hacer a nuestro TextField para cambiar su aspecto podrían ser las siguientes.

```
@Composable
fun Greeting(name: String, modifier: Modifier = Modifier) {
    var text1 by remember { mutableStateOf( value: "") } //Variable donde se guardará el texto
    var text2 by remember { mutableStateOf( value: "") } //Variable donde se guardará el texto
    Column(modifier){
        TextField(
            value = text1, //Texto del input
            onValueChange = { text1 = it }, //Función que asocia los cambios a la variable text
            label = { Text( text: "Usuario")},

        )
        Spacer(Modifier.height(10.dp))
        TextField(
            value = text2, //Texto en del input
            onValueChange = { text2 = it }, //Función que asocia los cambios a la variable text
            label = { Text( text: "Contraseña")},
            shape = RoundedCornerShape(16.dp),
            maxLines = 1,
            colors = TextFieldDefaults.colors(
                focusedIndicatorColor = Color.Transparent,
                unfocusedIndicatorColor = Color.Transparent
            )


        )
    }

}
```

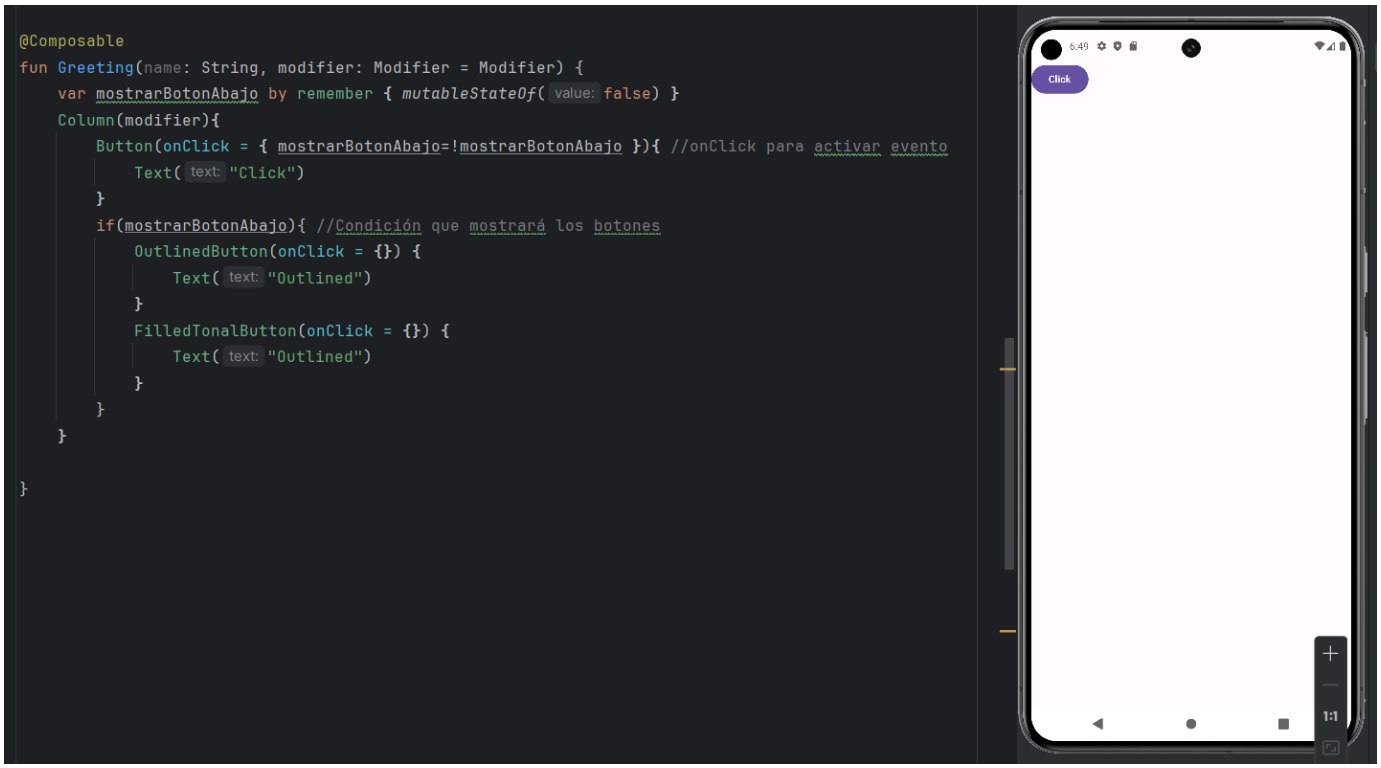
OUTLINEDTEXTFIELD

Una variante de TextField es la de OutlinedTextField.



BOTONES

En Android disponemos también de algunos botones por defecto que podemos utilizar.



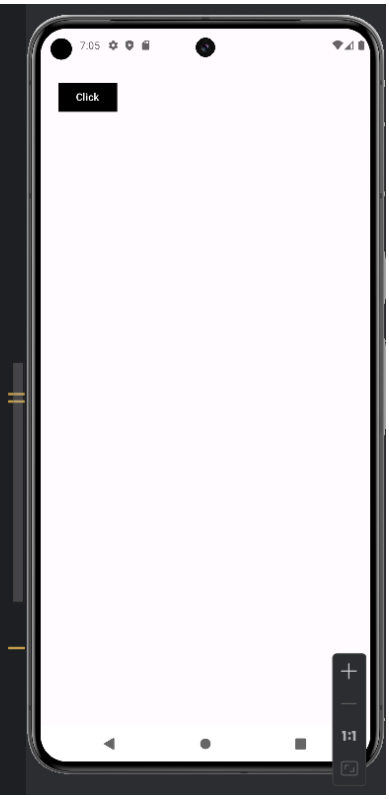
```
@Composable
fun Greeting(name: String, modifier: Modifier = Modifier) {
    var mostrarBotonAbajo by remember { mutableStateOf( value: false) }
    Column(modifier){
        Button(onClick = { mostrarBotonAbajo=!mostrarBotonAbajo }){ //onClick para activar evento
            Text( text: "Click")
        }
        if(mostrarBotonAbajo){ //Condición que mostrará los botones
            OutlinedButton(onClick = {}) {
                Text( text: "Outlined")
            }
            FilledTonalButton(onClick = {}) {
                Text( text: "Outlined")
            }
        }
    }

}
```


Las funciones en Kotlin que creamos para generar los componentes, también admite código “normal” que nos permitirá controlar nuestras pantallas. En este caso:

- Variable mostrarBotonAbajo, de tipo boolean.
- Al hacer click en el primer botón el booleano cambia de estado.
- Integramos una condición para que se muestren dos botones más.

Para personalizar nuestro botón, podemos pasarle un par de argumentos y modificarlos:

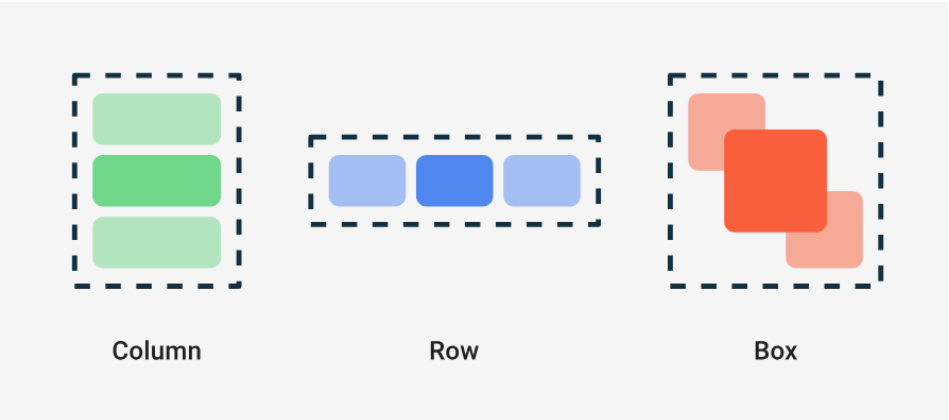
```
@Composable
fun Greeting(name: String, modifier: Modifier = Modifier) {
    var mostrarBotonAbajo by remember { mutableStateOf( value: false) }
    Column(modifier.padding(24.dp)){
        Button(onClick = { },
            border = BorderStroke(2.dp, Color.Black), //Añadimos borde al botón
            colors = ButtonDefaults.buttonColors( //Añadimos color al contenedor
                containerColor = Color.Black,
            ),
            shape = RoundedCornerShape(0.dp) // Cambiamos el radio
        ){ //onClick para activar evento
            Text( text: "Click")
        }
    }

}
```



- **onClick:** Al clickar en el botón, se ejecutará el código que contenga (se puede llamar a otras funciones).
- **Border:** Para cambiar el borde del botón. Recibe un objeto BorderStroke(tamaño, color).
- **Colors:** Por defecto, el color de nuestros botones será el que hayamos configurado en nuestros Temas. De momento no vamos a modificar nada de Temas, por lo que podemos modificar nuestros botones sobrescribiéndolo. En este caso, recibe el argumento "containerColor" black.
- Se ha añadido al modificador de la columna un **padding** de 24.dp.

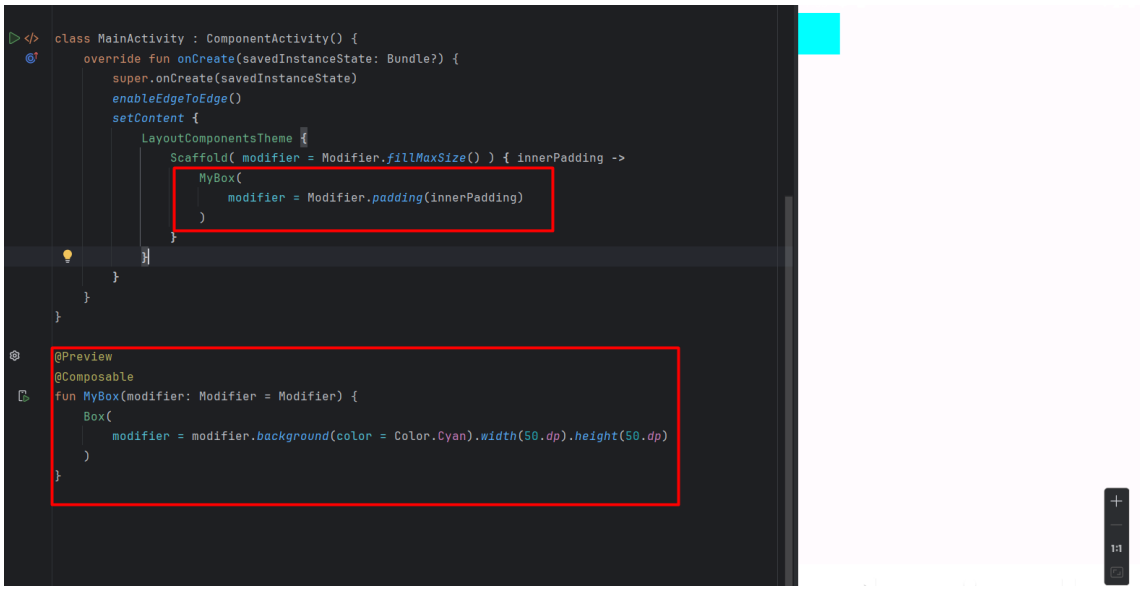
8. LAYOUT EN COMPOSE

Un layout es un contenedor que nos permite alinear los componentes dentro de nuestra vista.



BOX

El layout más sencillo de todos, que podremos posicionar en la pantalla según queramos. Se suelen utilizar como agrupación de otros componentes con tal de agruparlos de forma grupal.



```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContent {
            LayoutComponentsTheme {
                Scaffold( modifier = Modifier.fillMaxSize() ) { innerPadding ->
                    MyBox(
                        modifier = Modifier.padding(innerPadding)
                    )
                }
            }
        }
    }
}

@Preview
@Composable
fun MyBox(modifier: Modifier = Modifier) {
    Box(
        modifier = modifier.background(color = Color.Cyan).width(50.dp).height(50.dp)
    )
}
```

Se ha cambiado el ejemplo que da por defecto Android Studio con el componente Text y se ha añadido “MyBox” en su lugar. En caso de que queramos modificar un componente a nuestro gusto en Android (que será casi siempre), crearemos la función como “MyComponente”, de forma que la función devolverá un componente de tipo “Componente” con las modificaciones deseadas.

En este caso, MyBox devuelve un Box, donde hemos modificado mediante su modifier su tamaño y color, de forma que vemos un cuadrado de color Cyan en la vista de Diseño.

Modifiers

- **.background**. Para modificar el fondo del componente. En este caso el color.
- **Color.** Clase que contiene varios colores.
- **.width.** Para definir el ancho del componente.
- **.heigth.** Para definir el alto del componente.
- **dp.** Unidad de medida utilizada para que un componente se adapte bien a diferentes tamaños de pantalla.

Los componentes se pueden anidar, de forma que el componente exterior actuaría como “padre” del que se encuentra en el interior.

```
@Preview
@Composable
fun MyBox(modifier: Modifier = Modifier) {
    Box(Modifier.fillMaxSize()){
        Box(

            modifier = modifier.background(color = Color.Cyan)
                .width(50.dp)
                .height(50.dp)
                .align(alignment = Alignment.Center)

        )
    }

}
```

En este caso tenemos dos componentes anidados. Box actuaría como “padre” que además ocuparía todo el espacio disponible y sin ningún color. Box sería su “hijo” y se encontraría dentro del anterior componente.

Modifiers

- .**fillMaxSize()**. El componente ocupará el máximo espacio posible vertical y horizontalmente.
- **align.** Propiedad para posicionar un elemento respecto a su padre.
- **Alignment.Center.** Clase que contiene los posicionamientos. En este caso, para colocar en el centro respecto al componente superior.

Si al componente Box no se le definen ni ancho ni alto, ocupará únicamente el espacio que necesite. En caso de no tener componentes internos no se verá en la pantalla, y si los tuviese, se adaptaría a lo que ocupen ellos.

```
@Preview
@Composable
fun MyBox(modifier: Modifier = Modifier) {
    Box(Modifier.fillMaxSize()){
        Box(

            modifier = modifier.background(color = Color.Cyan)
                .align(alignment = Alignment.Center)

        ){
            Text( text: "Ejemplo de clase");
        }
    }

}
```

COLUMN

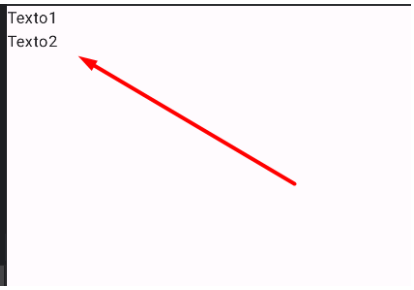
Por defecto, los elementos que vayamos poniendo en Android se solaparían uno encima de otro si no hacemos que su disposición en la pantalla sea diferente.

```
@Preview
@Composable
fun MyColumn(modifier: Modifier = Modifier){
    Text( text: "Texto1")
    Text( text: "Texto2")
}
```

En este caso vemos que tenemos dos componentes “Text”, pero que al ponerlos seguidos se solapan en la vista.

Controlar esto de forma manual constantemente se hace imposible, por eso utilizaremos el layout “Column”. Este layout hará que todos los elementos en su interior se dispongan de forma vertical, uno encima de otro.

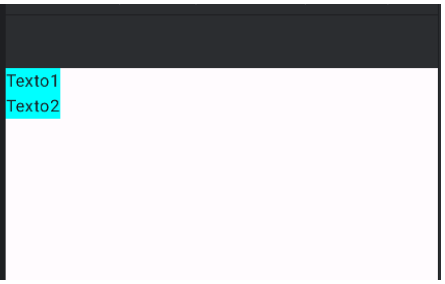
```
@Preview
@Composable
fun MyColumn(modifier: Modifier = Modifier){
    Column(){
        Text( text: "Texto1")
        Text( text: "Texto2")
    }

}
```



Una vez añadido el componente “Column”, sus hijos se disponen de forma vertical.

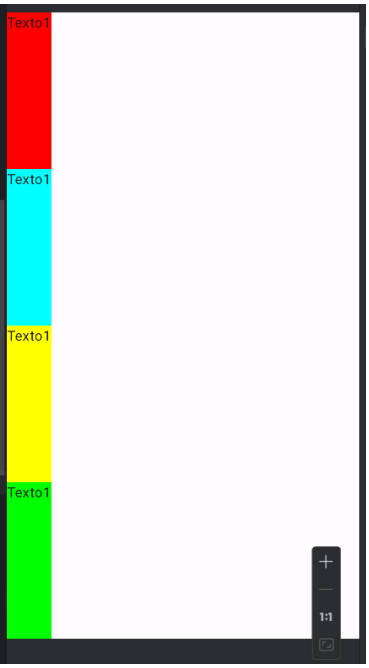
Al igual que en el componente Box, la columna será del tamaño que tengan sus “hijos”. Aquí podemos ver un ejemplo, marcado por el color Cyan de la columna.

```
@Preview
@Composable
fun MyColumn(modifier: Modifier = Modifier){
    Column( modifier.background(color = Color.Cyan)){
        Text( text: "Texto1")
        Text( text: "Texto2")
    }
}
```



En algunos casos, querremos que todos los elementos de nuestras columnas ocupen toda la pantalla **verticalmente**, independientemente del tamaño de sus elementos internos.

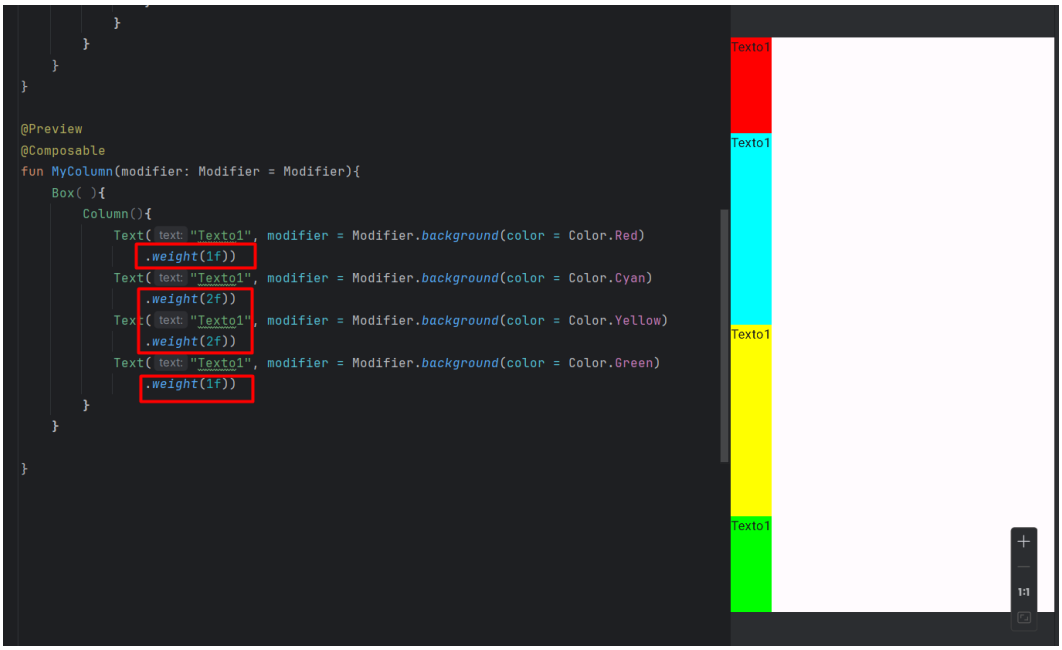
```
        }
    }
}

@Preview
@Composable
fun MyColumn(modifier: Modifier = Modifier){
    Box( ){
        Column(){
            Text( text: "Texto1", modifier = Modifier.background(color = Color.Red)
                .weight(1f))
            Text( text: "Texto1", modifier = Modifier.background(color = Color.Cyan)
                .weight(1f))
            Text( text: "Texto1", modifier = Modifier.background(color = Color.Yellow)
                .weight(1f))
            Text( text: "Texto1", modifier = Modifier.background(color = Color.Green)
                .weight(1f))
        }
    }

}
```



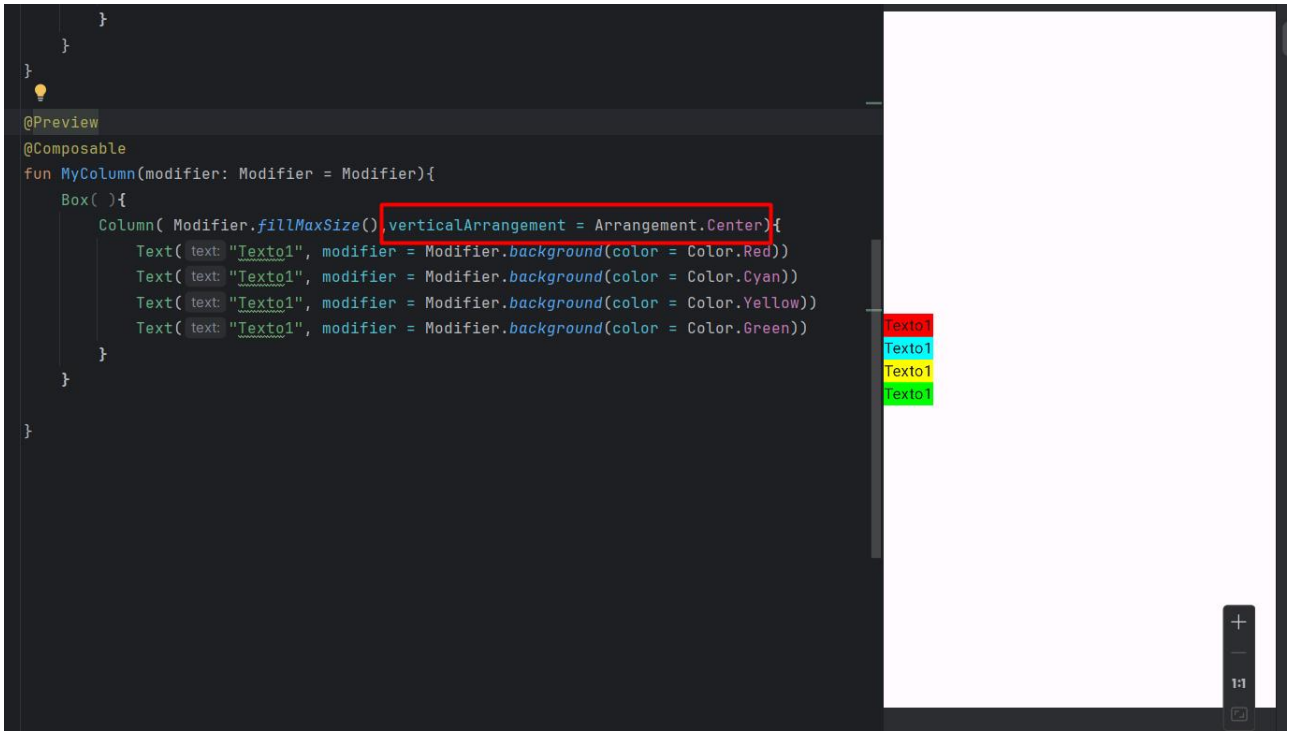
Modifiers

- .**weight.** Nos deja indicar la altura de un elemento de la columna. En el ejemplo, un peso 1f en cada elemento divide la pantalla a partes iguales.

Distintos pesos en los elementos de la columna conllevarían a distintos tamaños como podemos ver en el siguiente ejemplo.



Otra forma que podríamos utilizar para cambiar la disposición de elementos en una columna sería con el parámetro “verticalArrangement”.



Modifiers

- **verticalArrangement.** Mediante este parámetro de la columna, podemos decidir como se disponen los elementos en la pantalla.
    - Arrangement.Center
    - Arrangement.Top
    - Arrangement.Bottom
    - Arrangement.SpaceAround
    - Arrangement.SpaceEvenly

ROW

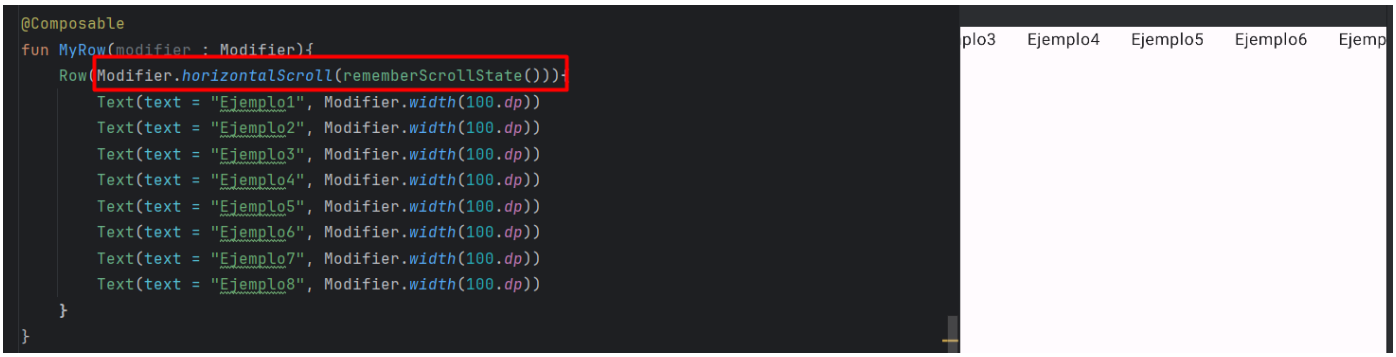
Al igual que con el anterior componente alineábamos los elementos de forma vertical, con Row los alinearemos de forma horizontal.

```
    }
}

@Composable
fun MyRow(modifier : Modifier){
    Row(){
        Text(text = "Ejemplo1")
        Text(text = "Ejemplo2")
    }
}
```

En este caso, cada una de las filas ocupa el tamaño del elemento que se encuentre en su interior. Podemos cambiar esto dándole un **tamaño fijo** a nuestros elementos.

```
@Composable
fun MyRow(modifier : Modifier){
    Row(){
        Text(text = "Ejemplo1", Modifier.width(200.dp))
        Text(text = "Ejemplo2", Modifier.width(200.dp))
    }
}
```

Aquí no habría problema ya que únicamente tenemos dos elementos y con el tamaño que les hemos dado, se muestran bien en la pantalla. Si quisiéramos añadir más elementos en una misma fila, o directamente son un número dinámico de elementos, podemos hacer fácilmente un **slider**.

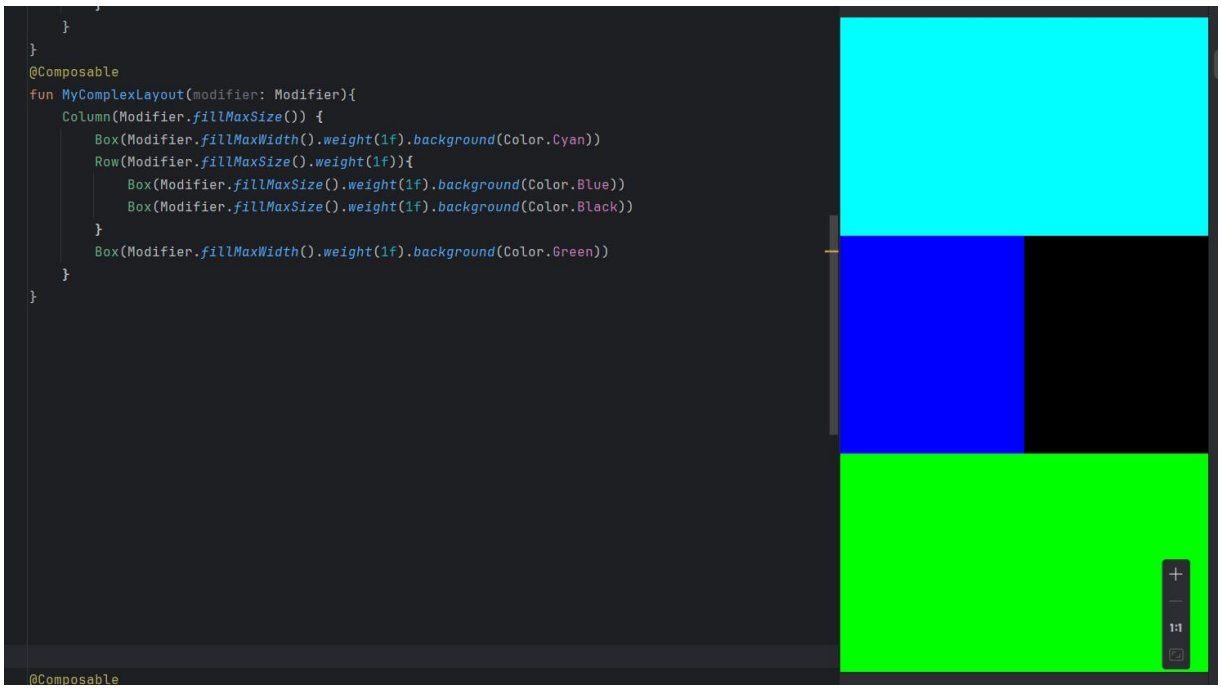


Modifier

.horizontalScroll. Si los elementos de una fila (o una columna) no caben en el espacio proporcionado, podemos crear un slide.

9. COMBINANDO LAYOUTS

El uso de estos layouts no tiene sentido si no se hace una combinación entre ellos para distribuir todos los elementos en la pantalla. Un ejemplo de uso sería el siguiente



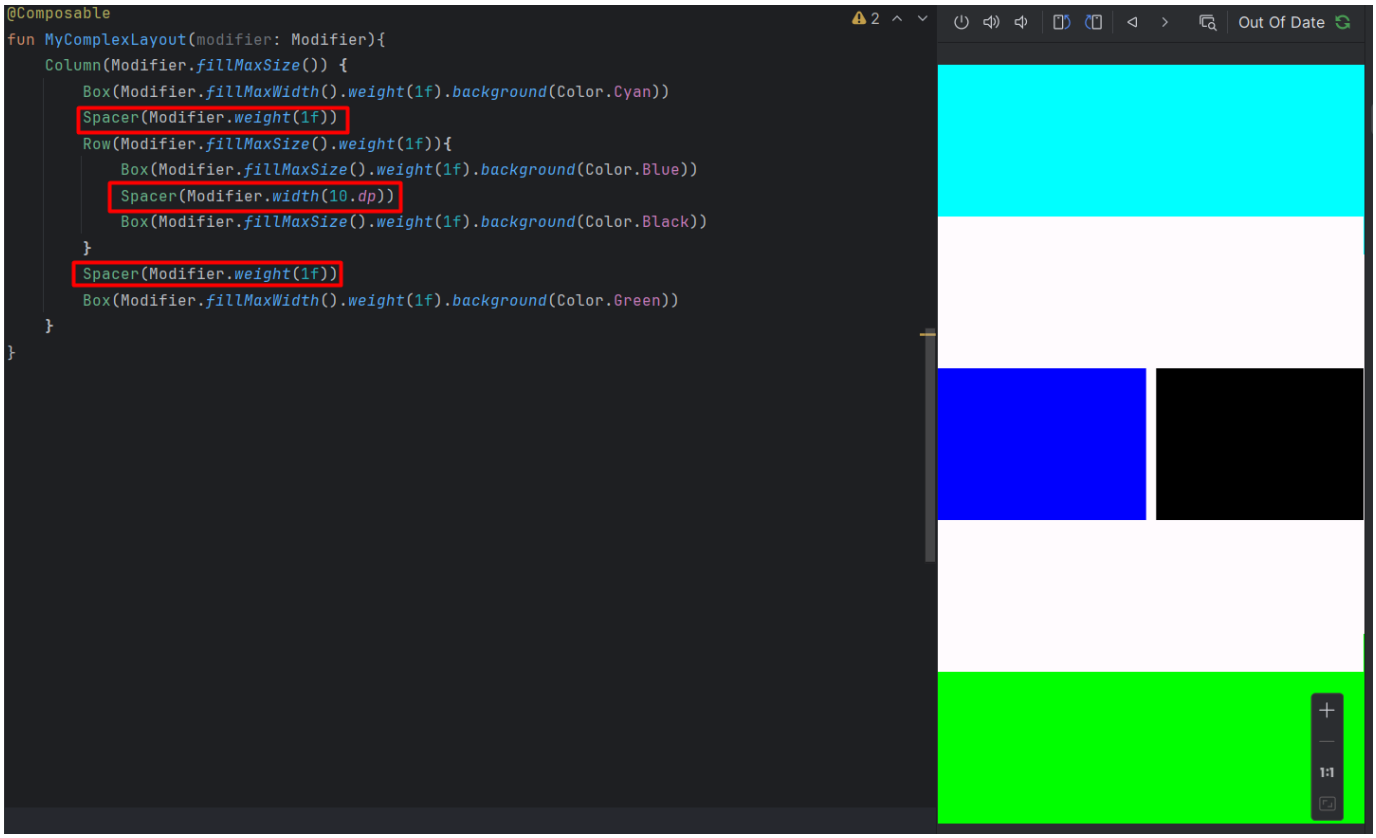
- Una columna que servirá como contenedor para todos los demás. Sus elementos hijos podrán ocupar toda la pantalla ya que tenemos el modificador. *fillMaxSize*.

- Una primera box, que ocupa el 33% de la columna debido al modificador. weight y color Cyan por *.background.*
- Un segundo layout Row, que ocupa también el 33% verticalmente y dentro, dos contenedores Box, ocupando el 50% cada uno.
- Finalmente, otro Box que ocupa el 33% de la pantalla.

SPACER

Es un componente que nos permite generar un espacio en la pantalla con tal de alinear bien nuestros elementos. En muchos casos nos ayudará a crear márgenes o espacios de forma sencilla, en vez de utilizar Margin o Padding.

Al ejemplo anterior del Layout combinado, le hemos añadido Spacers para ver cómo se utilizan.



¿Por qué no utilizar un box y darle el tamaño que le damos al Spacer?

El componente Spacer tiene la notación @NonRestartableComposable, lo que hace más eficiente el refresco de componentes que lo contengan, ya que no invertirán esfuerzos en su actualización.