



UD4.1 – Componentes Jetpack Compose

2º CFGS
Desarrollo de Aplicaciones Multiplataforma
2023-24

1.- Introducción

En esta unidad se van a estudiar los diferentes componentes que ofrece Jetpack Compose para diseñar interfaces.

Se estudiarán componentes de presentación e interacción como textos, imágenes, diálogos...

También se estudiarán componentes de layout que permiten organizar los elementos de la interfaz como columnas, rejillas, tarjetas...

1.- Introducción

En esta unidad se van a estudiar los diferentes componentes que ofrece Jetpack Compose para diseñar interfaces.

Se estudiarán componentes de presentación e interacción como textos, imágenes, diálogos...

También se estudiarán componentes de layout que permiten organizar los elementos de la interfaz como columnas, rejillas, tarjetas...

2.- Organizando el código

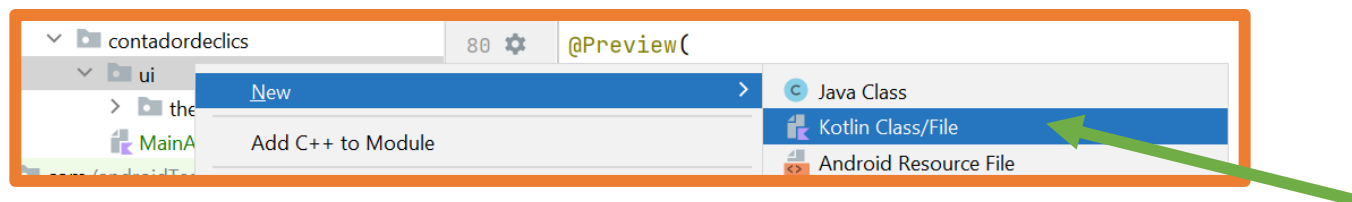
Antes de comenzar con el estudio de los componentes de Jetpack Compose es importante destacar que hay que organizar el código de la aplicación, por ello hay que tener en cuenta:

- Se pueden crear tantos archivos como se quiera.
- Si se tienen varias pantallas en la aplicación cada una tendrá su interfaz en un archivo.
- El archivo que contenga una clase que extienda de Activity no debe tener componentes en él.
- Se debe crear un componente con la base de la aplicación para poder utilizarlo en cualquier sitio.
- Todos los archivos que contengan componentes de la interfaz deberían incluirse en la carpeta ui.
- Si varios componentes forman parte de una zona/sección de la pantalla se debe crear un componente que los agrupe.
- Siempre que un componente se repita con las mismas características de estilo se debe crear un componente personalizado que lo extienda.

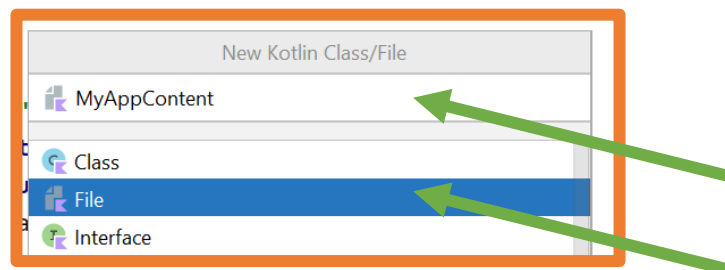
2.- Organizando el código

Para entender lo anterior se va a modificar el proyecto **Contador de clics** de la UD3 para cumplir con los puntos anteriores, además, a partir de ahora se harán indicaciones de dónde situar cada elemento que se estudie.

El primer paso consiste en crear un nuevo elemento Composable llamado **MyAppContent**:
Botón derecho sobre la carpeta ui → New → Kotlin Class/File.



Se indica el nombre del archivo: **MyAppContent**, se selecciona la opción File y se pulsa Enter.



2.- Organizando el código

Dentro del archivo **MyAppContent** se pondrá lo siguiente:

```
@Composable
fun MyAppContent(content: @Composable () -> Unit) {
    ContadorDeClicsTheme {
        Surface(
            modifier = Modifier.fillMaxSize(),
            color = MaterialTheme.colorScheme.background
        ) {
            content()
        }
    }
}
```

2.- Organizando el código

A continuación, se debe usar ese componente desde la función **onCreate** y desde la **previsualización**:

```
class MainActivity : ComponentActivity() {  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContent {  
            MyAppContent {  
                Content()  
            }  
        }  
    }  
}
```

```
@Composable  
fun ContentPreview() {  
    MyAppContent {  
        Content()  
    }  
}
```

De esta manera, siempre que se cree una pantalla nueva o se quiera previsualizar un componente se debe hacer uso del componente **MyAppContent** y se mantendrá el estilo de la aplicación.

2.- Organizando el código

Ahora es el turno de la función Content a la cual se le va a cambiar su nombre por **ClickCounter**. Se debe cambiar también las llamadas a ella tanto en la función **onCreate** como en la **previsualización**.

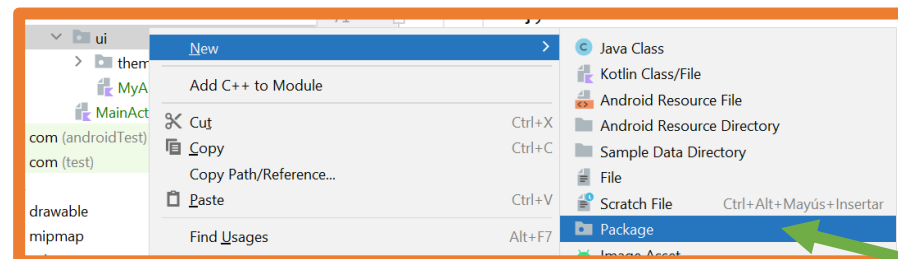
```
class MainActivity : ComponentActivity() {  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContent {  
            MyAppContent {  
                ClickCounter()  
            }  
        }  
    }  
}
```

```
@Composable  
fun ContentPreview() {  
    MyAppContent {  
        ClickCounter()  
    }  
}
```

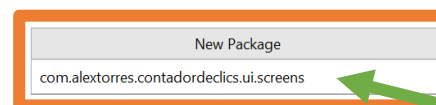
Para acabar se va a extraer la función **ClickCounter** a un archivo.

2.- Organizando el código

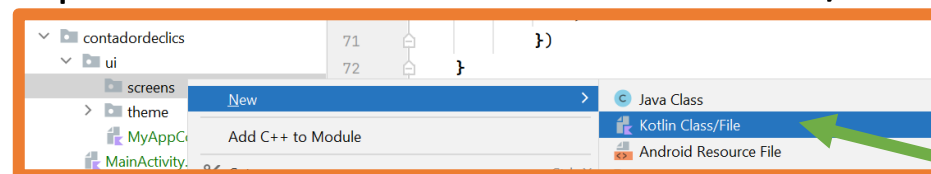
Botón derecho sobre la carpeta ui → New → Package.



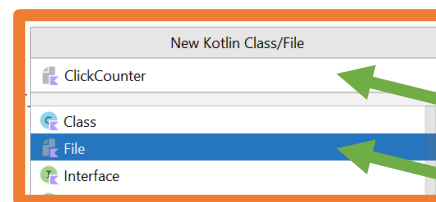
Se indica el nombre del paquete: **screens** y se pulsa Enter.



Botón derecho sobre la nueva carpeta screens → New → Kotlin Class/File:



Se indica el nombre del archivo: **ClickCounter**, se selecciona la opción File y se pulsa Enter.



2.- Organizando el código

La función **ClickCounter** y la previsualización **ContentPreview** se quitarán del archivo **MainActivity.kt** y se pondrán dentro del archivo **ClickCounter**:

```
27 @Composable
28 fun ClickCounter() {
29     var times by rememberSaveable { mutableStateOf( value: 0) }
30     Column(
31         modifier = Modifier.fillMaxSize(),
32         verticalArrangement = Arrangement.Center,
33         horizontalAlignment = Alignment.CenterHorizontally
34     ) { this: ColumnScope
35         Text(
36             // text = "has hecho clic $times veces",
37             text = stringResource(
38                 R.string.counter_text,
39                 times
40             ),
41             fontSize = 25.sp
42         )
43         Spacer(modifier = Modifier.height(20.dp))
44         Button(onClick = {
45             times++
46             Log.i( tag: "BOTON CLIC", msg: "Se ha pulsado el botón. Valor de times: $times")
47         },
48             content = { this: RowScope
49                 Text(
50                     text = stringResource(id = R.string.clickme),
51                     fontSize = 30.sp,
52                     modifier = Modifier.padding(16.dp),
53                 )
54             })
55     }
56 }
```

```
59 @Preview(
60     name = "Light Mode",
61     showBackground = true,
62     showSystemUi = true,
63 )
64 @Preview(
65     name = "Dark Mode",
66     showBackground = true,
67     showSystemUi = true,
68     uiMode = Configuration.UI_MODE_NIGHT_YES
69 )
70 @Preview(
71     showBackground = true,
72     name = "English",
73 )
74 @Preview(
75     showBackground = true,
76     name = "Spanish",
77     group = "locale",
78     locale = "es",
79 )
80 @Composable
81 fun ContentPreview() {
82     MyAppContent {
83         ClickCounter()
84     }
85 }
```

2.- Organizando el código

Por último, se cambiará el nombre de la previsualización **ContentPreview** por **ClickCounterPreview**:

```
@Composable
fun ClickCounterPreview() {
    MyAppContent {
        ClickCounter()
    }
}
```

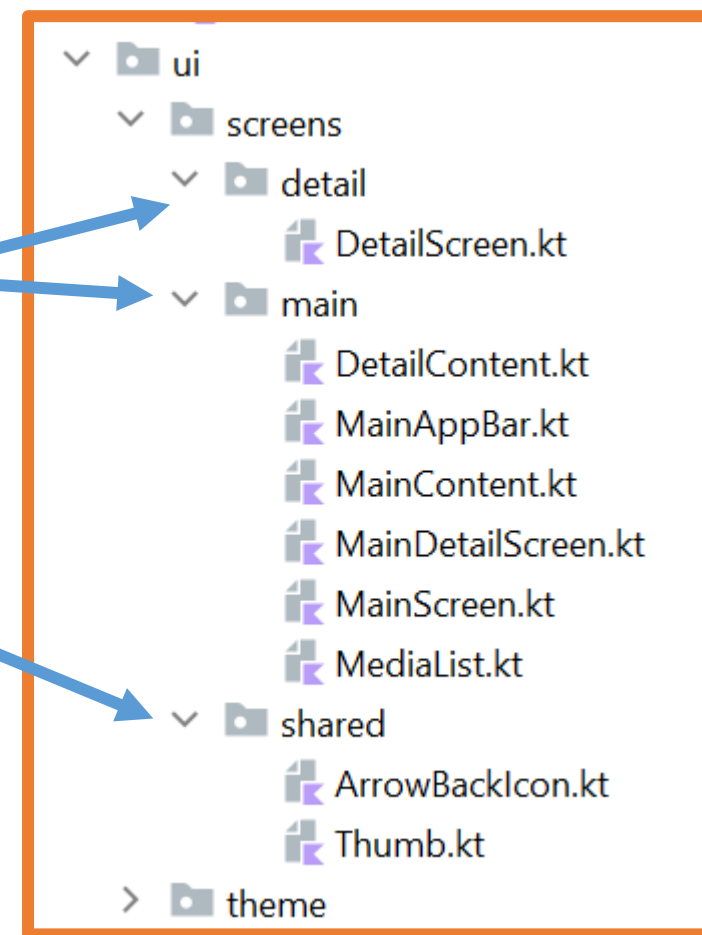
2.- Organizando el código

De esta manera el proyecto queda más organizado.

Si una pantalla se va a componer de diferentes archivos es recomendable crear un **nuevo paquete** (package) con el nombre de la pantalla dentro de la carpeta **screens**.

Si hay componentes que se comparten entre pantallas es recomendable crear un nuevo paquete llamado **common**, **share** o **partials** y guardar esos componentes en esa carpeta.

Es importante mantener organizado siempre el código.



3.- Componentes Jetpack Compose

Los componentes de Jetpack Compose se pueden clasificar extraoficialmente en dos clases:

Layout

Permiten organizar los elementos de la interfaz gráfica

Box	Surface
Column	Row
Card	ConstraintLayout
Scaffold	AppBar
BottomBar	ModalDrawer
LazyColumn	LazyRow
LazyVerticalGrid	LazyHorizontalGrid
LazyVerticalStaggeredGrid	LazyHorizontalStaggeredGrid
...	

Presentación de información

Permiten mostrar al usuario información. El usuario puede interactuar con ellos.

Text	TextField
Button	Image
Spacer	Switch
Slider	CheckBox
RadioButton	ElevatedButton
Icon	IconButton
SnackBar	BadgedBox
Slider	FloatingActionButton
...	

3.- Componentes Jetpack Compose

En la unidad anterior ya se han utilizado algunos componentes de Jetpack Compose: Column, Row, Text, Button, Spacer, Surface.

Toda la interfaz de las aplicaciones se va a realizar con:

- **Componentes que ofrece Jetpack Compose.**
- **Componentes de Jetpack Compose propios** creados en base a extensiones y agrupaciones de los componentes que ofrece Jetpack Compose.

3.- Componentes Jetpack Compose

Los **componentes** Jetpack Compose son **funciones** etiquetadas con **@Composable**.

Esto se puede observar en los componentes que facilita Jetpack Compose:

```
@Composable ←  
fun Text(  
    text: String,  
    modifier: Modifier = Modifier,  
    color: Color = Color.Unspecified,  
    fontSize: TextUnit = TextUnit.Unspecified
```

```
@Composable ←  
fun Button(  
    onClick: () -> Unit,  
    modifier: Modifier = Modifier,  
    enabled: Boolean = true,  
    shape: Shape = ButtonDefaults.shape.
```

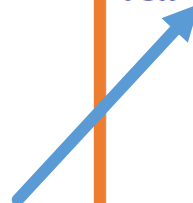
```
@Composable ←  
@NonRestartableComposable  
fun Spacer(modifier: Modifier) {  
    Layout({}, measurePolicy = SpacerMeasurePolicy, modifier = modifier)  
}
```

3.- Componentes Jetpack Compose

Para crear un componente propio se debe crear una función etiquetada con **@Composable**.

Además la documentación indica que:

- Si la función **@Composable** no devuelve nada (lo más habitual) su nombre debe comenzar con **mayúscula**.
- Si la función **@composable** devuelve algo su nombre debe comenzar en **minúscula**.



```
@Composable
fun Text(
    text: String,
    modifier: Modifier = Modifier,
    color: Color = Color.Unspecified,
    fontSize: TextUnit = TextUnit.Unspecified
```


4.- Material 3

La última versión de Jetpack Compose utiliza Material Design 3. Material Design es un estilo de diseño de interfaces de Google.

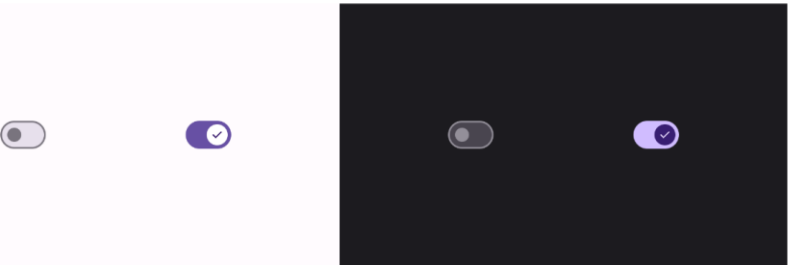
Todos los componentes Jetpack Compose se pueden encontrar en la [documentación oficial](#) con ejemplo visual y enlace a [Material Design 3](#): para ver cómo usar el componente y las guías de estilo:

Switch Added in 1.0.0

```
@Composable
fun Switch(
    checked: Boolean,
    onCheckedChange: ((Boolean) -> Unit)?,
    modifier: Modifier = Modifier,
    thumbContent: (@Composable () -> Unit)? = null,
    enabled: Boolean = true,
    colors: SwitchColors = SwitchDefaults.colors(),
    interactionSource: MutableInteractionSource = remember { MutableInteractionSource() }
): Unit
```

[Material Design Switch](#)

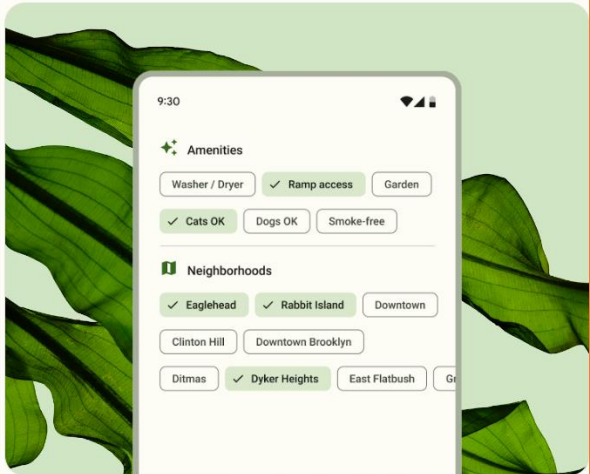
Switches toggle the state of a single item on or off.



Home
Get started
Develop
Foundations
Styles
Components
Blog

Chips

Chips help people enter information, make selections, filter content, or trigger actions



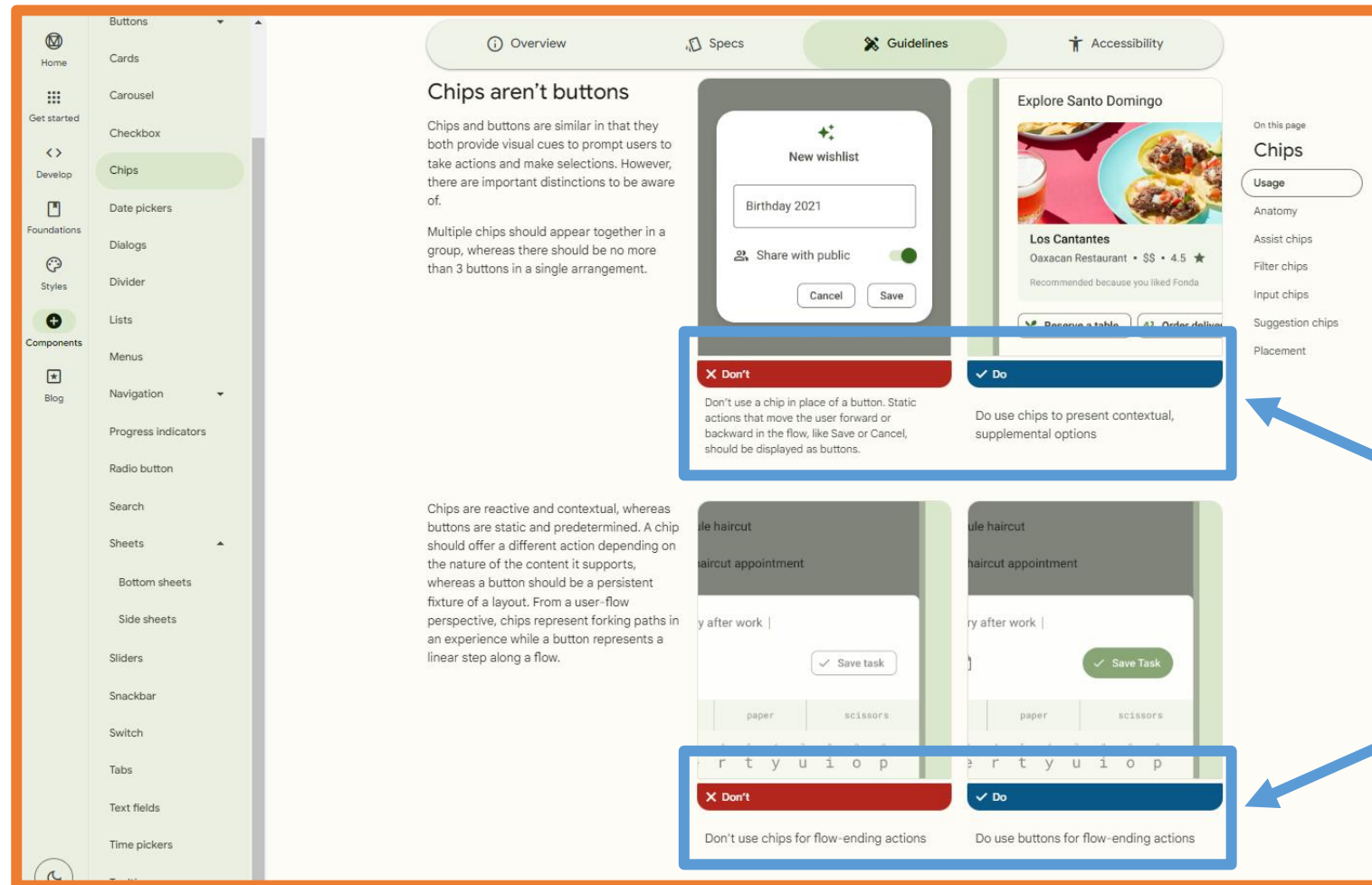
Overview Specs Guidelines Accessibility

- ★ Use chips to show options for a specific context
- ★ Four types: assist, filter, input, and suggestion
- ★ Chip elevation defaults to 0 but can be elevated if they need more visual separation

On this page
Chips
Resources
What's new

4.- Material 3

La web de [Material Design 3](#) ofrece información muy importante de cómo usar los componentes en el apartado **Guidelines** (guías de estilo):



5.- Componentes Text y BasicText

Los componentes **Text** y **BasicText** permiten mostrar una cadena de texto en la interfaz.

La diferencia es que **Text** utiliza los principios de **Material** y **BasicText** no.

Su uso básico ya se ha estudiado en clase y es el siguiente:

```
Text(text = "Hello Rick!")
```

5.- Componentes Text y BasicText

Es importante **conocer el funcionamiento de los componentes** de Jetpack Compose, se pulsa la tecla **CTRL** y sin soltar se hace clic izquierdo sobre el nombre de la clase.



También funciona teniendo el cursor en el nombre y pulsado CTRL+B.

De esta manera se abre en Android Studio el código del componente clicado.

Esta acción se puede hacer con cualquier clase, función, objeto...

5.- Componentes Text y BasicText

Todas las funciones disponen de documentación KotlinDoc donde se explican los parámetros que aceptan:

High level element that displays text and provides semantics / accessibility information.

The default `style` uses the `LocalTextStyle` provided by the `MaterialTheme` / components. If you are setting your own style, you may want to consider first retrieving `LocalTextStyle`, and using `TextStyle.copy` to keep any theme defined attributes, only modifying the specific attributes you want to override.

For ease of use, commonly used parameters from `TextStyle` are also present here. The order of precedence is as follows:

- If a parameter is explicitly set here (i.e, it is *not* `null` or `TextUnit.Unspecified`), then this parameter will always be used.
- If a parameter is *not* set, (`null` or `TextUnit.Unspecified`), then the corresponding value from `style` will be used instead.

Additionally, for `color`, if `color` is not set, and `style` does not have a color, then `LocalContentColor` will be used.

Params: `text` - the text to be displayed

`modifier` - the `Modifier` to be applied to this layout node

`color` - `Color` to apply to the text. If `Color.Unspecified`, and `style` has no color set, this will be `LocalContentColor`.

`fontSize` - the size of glyphs to use when painting the text. See `TextStyle.fontSize`.

`fontStyle` - the typeface variant to use when drawing the letters (e.g., italic). See `TextStyle.fontStyle`.

`fontWeight` - the typeface thickness to use when painting the text (e.g., `FontWeight.Bold`).

`fontFamily` - the font family to be used when rendering the text. See `TextStyle.fontFamily`.

`letterSpacing` - the amount of space to add between each letter. See `TextStyle.letterSpacing`.

`textDecoration` - the decorations to paint on the text (e.g., an underline). See `TextStyle.textDecoration`.

`textDecorations`.

`textAlign` - the alignment of the text within the lines of the paragraph. See `TextStyle.textAlign`.

`lineHeight` - line height for the `Paragraph` in `TextUnit` unit, e.g. SP or EM. See `TextStyle.lineHeight`.

`lineHeight`.

`overflow` - how visual overflow should be handled.

`softWrap` - whether the text should break at soft line breaks. If false, the glyphs in the text will be positioned as if there was unlimited horizontal space. If `softWrap` is false, `overflow` and `TextAlign` may have unexpected effects.

`maxLines` - an optional maximum number of lines for the text to span, wrapping if necessary. If the text exceeds the given number of lines, it will be truncated according to `overflow` and `softWrap`. If it is not null, then it must be greater than zero.

`onTextLayout` - callback that is executed when a new text layout is calculated. A `TextLayoutResult` object that callback provides contains paragraph information, size of the text, baselines and other details. The callback can be used to add additional decoration or functionality to the text. For example, to draw selection around the text.

`style` - style configuration for the text such as color, font, line height etc.

```
@Composable
fun Text(
    text: String,
    modifier: Modifier = Modifier,
    color: Color = Color.Unspecified,
    fontSize: TextUnit = TextUnit.Unspecified,
    fontStyle: FontStyle? = null,
    fontWeight: FontWeight? = null,
    fontFamily: FontFamily? = null,
    letterSpacing: TextUnit = TextUnit.Unspecified,
    textDecoration: TextDecoration? = null,
    textAlign: TextAlign? = null,
    lineHeight: TextUnit = TextUnit.Unspecified,
    overflow: TextOverflow = TextOverflow.Clip,
    softWrap: Boolean = true,
    maxLines: Int = Int.MAX_VALUE,
    onTextLayout: (TextLayoutResult) -> Unit = {},
    style: TextStyle = LocalTextStyle.current
) {
    // NOTE(text-perf-review): It might be worthwhile writing a bespoke merge implementation that
    // will avoid reallocating if all of the options here are the defaults
    val mergedStyle = style.merge(
        TextStyle(
            color = textColor,
            fontSize = fontSize,
            fontWeight = fontWeight,
            textAlign = textAlign,
            lineHeight = lineHeight,
            fontFamily = fontFamily,
            textDecoration = textDecoration,
            fontStyle = fontStyle,
            letterSpacing = letterSpacing
        )
    )
    BasicText(
        text,
        modifier,
        mergedStyle,
        onTextLayout,
        overflow,
        softWrap,
        maxLines,
    )
}
```

5.- Componentes Text y BasicText

En la llamada a la función Text, al usar el nombre en los parámetros como recomienda Jetpack Compose, el orden de los mismos se puede alterar.

```
Text(  
    text = "Hola Rick!",  
    color = Color.Red,  
    fontSize = 25.sp,  
    fontStyle = FontStyle.Italic,  
    fontWeight = FontWeight.Bold,  
    fontFamily = FontFamily.Serif,  
    letterSpacing = 5.sp,  
    textDecoration = TextDecoration.Underline,  
    textAlign = TextAlign.End,  
    lineHeight = 2.em,  
    maxLines = 1,  
    softWrap = false,  
    overflow = TextOverflow.Ellipsis,  
    onTextLayout = { it: TextLayoutResult  
        Log.i( tag: "InfoText", msg: "Texto renderizado. tamaño ${it.size}")  
    },  
    modifier = Modifier.fillMaxWidth(),  
)
```

TextPreview

Hola Rick!

5.- Componentes Text y BasicText

Cuando se usan los componentes de Jetpack Compose, igual que sucede con cualquier clase que no sea propia, es importante conocer qué parámetros tiene y cómo funciona.

Con el cursor en una clase/función/parámetro/variable se pulsa CTRL+B y Android Studio abre el archivo donde se define para poder consultarlo.

Por ejemplo:

- text necesita un String.
- modifier necesita un Modifier.
- fontWeight necesita un FontWeight.

Navegando por los parámetros con CTRL+B se puede saber el tipo de dato que acepta. Esto también se puede consultar por internet en la documentación oficial.



```
fun Text(  
    text: String,  
    modifier: Modifier = Modifier,  
    color: Color = Color.Unspecified,  
    fontSize: TextUnit = TextUnit.Unspecified,  
    fontStyle: FontStyle? = null,  
    fontWeight: FontWeight? = null,  
    fontFamily: FontFamily? = null,  
    letterSpacing: TextUnit = TextUnit.Unspecified,  
    textDecoration: TextDecoration? = null,  
    textAlign: TextAlign? = null,  
    lineHeight: TextUnit = TextUnit.Unspecified,  
    overflow: TextOverflow = TextOverflow.Clip,  
    softWrap: Boolean = true,  
    maxLines: Int = Int.MAX_VALUE,  
    onTextLayout: (TextLayoutResult) -> Unit = {},  
    style: TextStyle = LocalTextStyle.current  
) {
```

5.- Componentes Text y BasicText

El único **parámetro obligatorio** es **text** que será la cadena a mostrar.

El resto de parámetros se pueden omitir (tienen asignado un valor), en ese caso Kotlin le pondrá los valores por defecto.

```
@Composable
fun Text(
    text: String,
    modifier: Modifier = Modifier,
    color: Color = Color.Unspecified,
    fontSize: TextUnit = TextUnit.Unspecified,
    fontStyle: FontStyle? = null,
    fontWeight: FontWeight? = null,
    fontFamily: FontFamily? = null,
    letterSpacing: TextUnit = TextUnit.Unspecified,
    textDecoration: TextDecoration? = null,
    textAlign: TextAlign? = null,
    lineHeight: TextUnit = TextUnit.Unspecified,
    overflow: TextOverflow = TextOverflow.Clip,
    softWrap: Boolean = true,
    maxLines: Int = Int.MAX_VALUE,
    onTextLayout: (TextLayoutResult) -> Unit = {},
    style: TextStyle = LocalTextStyle.current
) {
```


5.- Componentes Text y BasicText

Algunos ejemplos de Text con parámetros:


Con el parámetro **style** se pueden indicar diferentes configuraciones de estilo a la vez.

Aunque se pueden usar los parámetros específicos como **color**, **fontSize**...



6.- Modifier

Todos los componentes tienen un parámetro modifier.



```
@Composable
fun Text(
    text: String,
    modifier: Modifier = Modifier,
    color: Color = Color.Unspecified,
    fontSize: TextUnit = TextUnit.Unspecified
```

Por defecto el **parámetro modifier** contiene un objeto **Modifier** con la configuración básica.

6.- Modifier

El objeto **Modifier** es un **Companion Object** con muchas funciones de extensión para modificar los componentes

Además, es de tipo **builder** por lo que se pueden concatenar las llamadas a las funciones mediante el punto.

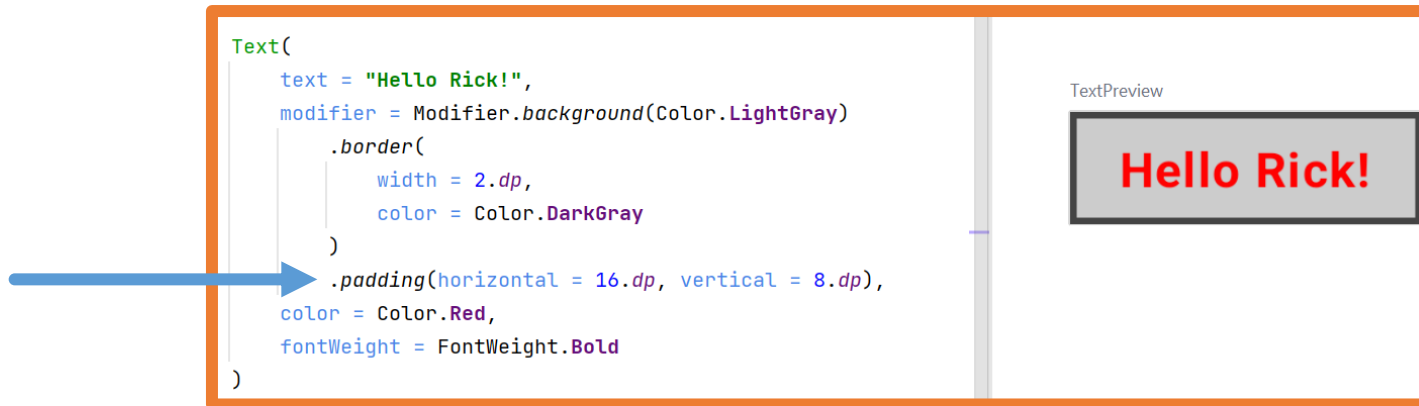
El **orden** de llamada a las funciones es **importante** y gracias a él se crean diferentes efectos.

Las funciones que contiene Modifier se pueden clasificar extraoficialmente en:

De posicionamiento y tamaño	fillMaxSize, fillMaxWidth, fillMaxHeight, wrapContentSize, wrapContentHeight, width, height, aspectRatio...
De funcionalidad	clickable, toggleable, horizontalScroll, verticalScroll...
De apariencia	padding, border, alpha...
Listeners (manejadores de eventos)	clickable, onFocusChanged, onKeyEvent, onSizeChanged...

6.- Modifier

Ejemplo de **Text** con **modifier**:



El orden de llamada a las funciones de Modifier altera el comportamiento:



6.- Modifier

Gracias a **Modifier** todos los componentes pueden ser **clickable**:



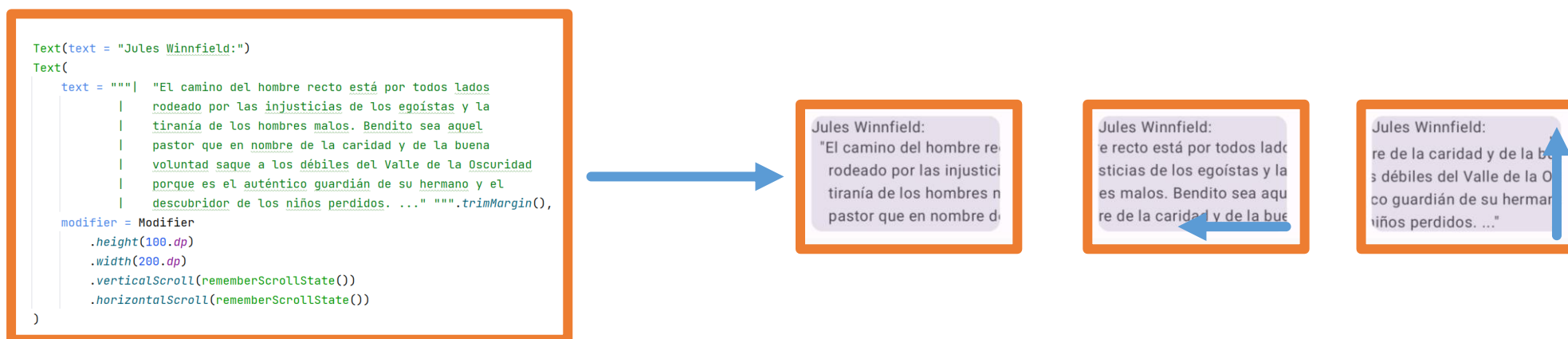
```
Text(  
    text = "Hello Rick!",  
    modifier = Modifier.background(Color.LightGray)  
        .border(  
            width = 2.dp,  
            color = Color.DarkGray  
        )  
        .padding(horizontal = 16.dp, vertical = 8.dp)  
        .clickable {  
            Log.i(tag: "Text", msg: "Se ha pulsado")  
        },  
    color = Color.Red,  
    fontWeight = FontWeight.Bold  
)
```

El orden en el que aparezca **clickable** también altera el comportamiento.

Existen más manejadores de eventos: onFocusChanged, onKeyEvent...

6.- Modifier

Si el contenido de un componente no se puede mostrar entero por cualquier razón, gracias a **Modifier** se puede indicar con **verticalScroll** y **horizontalScroll** que el contenido sea deslizable para poder visualizarlo totalmente.



Si un componente con un **verticalScroll** o un **HorizontalScroll** está ocupando toda la pantalla, el resto de elementos no se visualizarán a menos que al componente con **Scroll** se le indique un tamaño.

Si en un elemento se puede hacer scroll vertical, dentro no debería tener un elemento con scroll vertical. Así, si en un elemento se puede hacer scroll horizontal, dentro no debería haber uno con scroll horizontal.

Más adelante se explicará el uso de **rememberScrollState**.

6.- Modifier

En la documentación se pueden encontrar todas las funciones de extensión disponibles para el objeto Modifier:

<https://developer.android.com/jetpack/compose/modifiers-list>

7.- Componente Button

Un **Button** permite lanzar una acción al interactuar con él.

Tiene dos parámetros obligatorios **onClick** y **content** que son funciones lambda, este último además es el último parámetro definido por lo que se puede extraer de los paréntesis.

```
Button(onClick = {  
    Log.i( tag: "Botón", msg: "Se ha pulsado el botón")  
}) { this: RowScope  
    Text( text: "Button")  
}
```

```
@Composable  
fun Button(  
    onClick: () -> Unit,  
    modifier: Modifier = Modifier,  
    enabled: Boolean = true,  
    shape: Shape = ButtonDefaults.shape,  
    colors: ButtonColors = ButtonDefaults.buttonColors(),  
    elevation: ButtonElevation? = ButtonDefaults.buttonElevation(),  
    border: BorderStroke? = null,  
    contentPadding: PaddingValues = ButtonDefaults.ContentPadding,  
    interactionSource: MutableInteractionSource = remember { MutableInteractionSource() },  
    content: @Composable RowScope.() -> Unit  
) {
```


7.- Componente Button

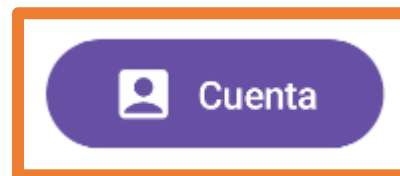
El parámetro **content** es una función lambda que acepta **@Composable**, eso significa que en el cuerpo de **content** se pueden poner componentes Jetpack Compose.

```
@Composable
fun Button(
    onClick: () -> Unit,
    modifier: Modifier = Modifier,
    enabled: Boolean = true,
    shape: Shape = ButtonDefaults.shape,
    colors: ButtonColors = ButtonDefaults.buttonColors(),
    elevation: ButtonElevation? = ButtonDefaults.buttonElevation(),
    border: BorderStroke? = null,
    contentPadding: PaddingValues = ButtonDefaults.ContentPadding,
    interactionSource: MutableInteractionSource = remember { MutableInteractionSource() },
    content: @Composable RowScope.() -> Unit
) {
```

```
    interactionSource: MutableInteractionSource = remember { Mut
    content: @Composable RowScope.() -> Unit
} {
```

Además, implementa **RowScope** (ámbito de columna) por lo que se comporta como una **columna** y los componentes que se incluyan se añadirán en fila.

```
Button(onClick = { /*TODO*/ }) { this: RowScope
    Icon(
        imageVector = Icons.Default.AccountBox,
        contentDescription = "Cuenta"
    )
    Spacer(modifier = Modifier.width(8.dp))
    Text(text = "Cuenta")
}
```



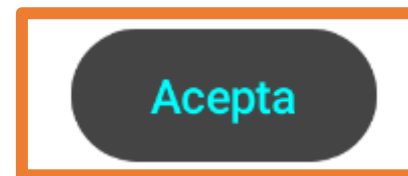
7.- Componente Button

Button define algunos de sus valores por defecto en la clase **ButtonDefaults** y para algunas modificaciones se debe usar esa clase:

```
@Composable
fun Button(
    onClick: () -> Unit,
    modifier: Modifier = Modifier,
    enabled: Boolean = true,
    shape: Shape = ButtonDefaults.shape,
    colors: ButtonColors = ButtonDefaults.buttonColors(),
    elevation: ButtonElevation? = ButtonDefaults.buttonElevation(),
    border: BorderStroke? = null,
    contentPadding: PaddingValues = ButtonDefaults.ContentPadding,
    interactionSource: MutableInteractionSource = remember { MutableInteractionSource() },
    content: @Composable RowScope() -> Unit
) {
```

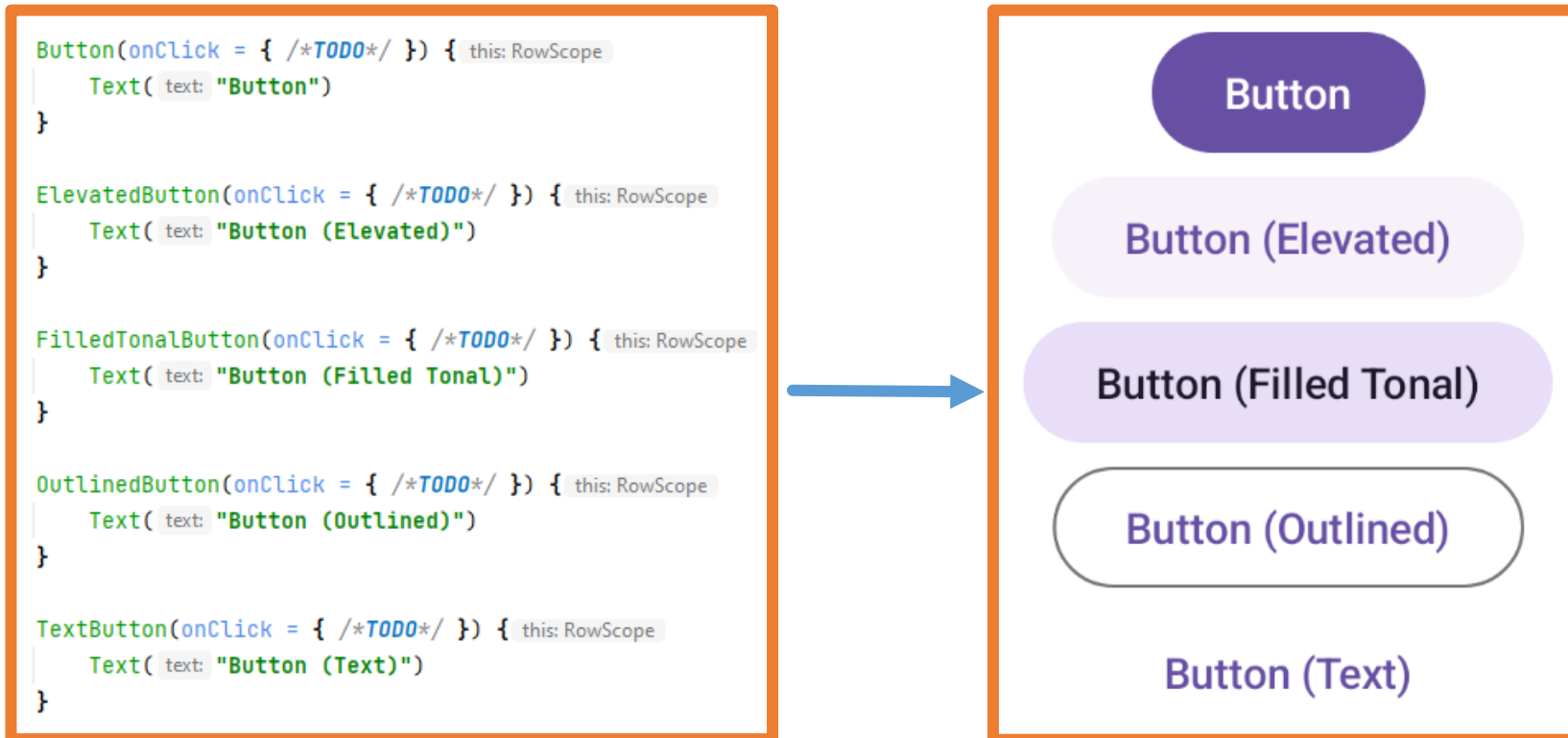
```
shape: Shape = ButtonDefaults.shape,
colors: ButtonColors = ButtonDefaults.buttonColors(),
elevation: ButtonElevation? = ButtonDefaults.buttonElevation(),
border: BorderStroke? = null,
contentPadding: PaddingValues = ButtonDefaults.ContentPadding,
```

```
Button(
    onClick = { /*TODO*/ },
    colors = ButtonDefaults.buttonColors(
        contentColor = Color.Cyan,
        containerColor = Color.DarkGray
    )
) { this: RowScope
    Text( text: "Acepta" )
}
```



7.- Componente Button

El componente Button tiene diferentes **variaciones** con estilo predefinido:



Internamente todos estos componentes son del tipo Button.

8.- Componentes propios

Para organizar el código es habitual crear componentes propios que extiendan o agrupen a componentes de Jetpack Compose.

```
// Text con un formato predefinido
@Composable
fun HeaderText(text: String) {
    Text(
        text = text,
        fontSize = 20.sp,
        fontWeight = FontWeight.Bold,
        textDecoration = TextDecoration.Underline,
        modifier = Modifier
            .background(Color.LightGray)
            .border(width = 1.dp, color = Color.DarkGray)
            .padding(horizontal = 16.dp, vertical = 8.dp)
    )
}
```

```
// Text para saludar que acepta Modifier
@Composable
fun HeaderGreetingText(
    name: String,
    modifier: Modifier = Modifier
) {
    Text(
        text = "Hola $name",
        modifier = modifier
    )
}
```

```
// Componente propio para mostrar a un usuario
@Composable
fun UserItemList(user: User) {
    Row(
        modifier = Modifier
            .fillMaxWidth()
            .padding(horizontal = 8.dp, vertical = 4.dp)
    ) { this: RowScope
        Text(user.name)
        Spacer(Modifier.width(10.dp))
        Text(text: "(")
        Icon(
            imageVector = Icons.Default.Mail,
            contentDescription = "mail",
            modifier = Modifier.size(18.dp)
        )
        Text(text: " ${user.mail}")
    }
}
```

```
HeaderText(text: "¡Bienvenido!")

HeaderGreetingText(
    name = "Rick",
    modifier = Modifier.background(Color.Yellow)
)

UserItemList(User(name: "Rick", mail: "rick@mail.com"))
```



8.- Componentes propios

Se puede diseñar un componente propio que combine un Modifier interno y el Modifier que reciba como parámetro.

```
// Text con un formato predefinido
@Composable
fun HeaderText(text: String) {
    Text(
        text = text,
        fontSize = 20.sp,
        fontWeight = FontWeight.Bold,
        textDecoration = TextDecoration.Underline,
        modifier = Modifier
            .background(Color.LightGray)
            .border(width = 1.dp, color = Color.DarkGray)
            .padding(horizontal = 16.dp, vertical = 8.dp)
    )
}
```

```
// Text con un formato predefinido que acepta modificaciones
@Composable
fun HeaderTextMod(text: String, modifier: Modifier = Modifier) {
    Text(
        text = text,
        fontSize = 20.sp,
        fontWeight = FontWeight.Bold,
        textDecoration = TextDecoration.Underline,
        modifier = Modifier
            .background(Color.LightGray)
            .border(
                width = 1.dp,
                color = Color.DarkGray
            )
            .padding(
                horizontal = 16.dp,
                vertical = 8.dp
            )
            .then(modifier)
    )
}
```

```
HeaderTextMod(text = ";Bienvenido!")

HeaderTextMod(
    text = ";Bienvenido!",
    modifier = Modifier.background(Color.Cyan)
)
```

El lugar donde se llame a **then** cambiará el comportamiento.

8.- Componentes propios

Los componentes propios permiten dividir el código para que esté más organizado.

Es importante que los componentes propios **sean lo más pequeños posible** teniendo nombres **semánticos** para que sean auto explicativos.

Además, se pueden crear componentes que agrupen a otros para poder reutilizarlos todos juntos y organizar el código.

Los ejemplos anteriores son perfectos para entender esto.

9.- Componente Spacer

El componente **Spacer** permite dejar un espacio entre otros componentes.

El componente Spacer solo admite el parámetro **modifier**.

```
Text( text: "Hola Rick")

Spacer(modifier = Modifier.height(30.dp))

Text( text: "Hello there!")
```

Hola Rick
Hello there!

```
Spacer(modifier = Modifier.width(15.dp))
Spacer(
    modifier = Modifier
        .width(15.dp)
        .height(40.dp)
)
```

Es recomendable crearse un componente propio que extienda a Spacer para simplificar su uso:

```
@Composable
fun MySpacer(width: Int = 0, height: Int = 0) {
    Spacer(
        modifier = Modifier.size(
            width = width.dp,
            height = height.dp
        )
    )
}
```

```
@Composable
fun MySpacer(size: Int) {
    Spacer(
        modifier = Modifier.size(size.dp)
    )
}
```



```
MySpacer(height = 50)
MySpacer(width = 15)
MySpacer(width = 15, height = 30)
MySpacer(size = 50)
```

10.- Imágenes

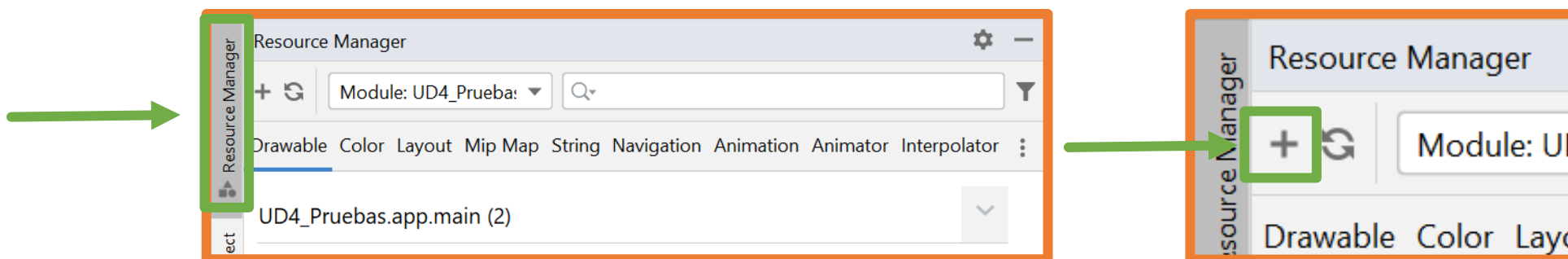
En una aplicación Android las imágenes se tienen que importar previamente en el proyecto o bien conseguirlas de internet.

Las imágenes que se importen al proyecto deben estar en uno de los siguientes formatos: vectorial, jpg, png y webp.

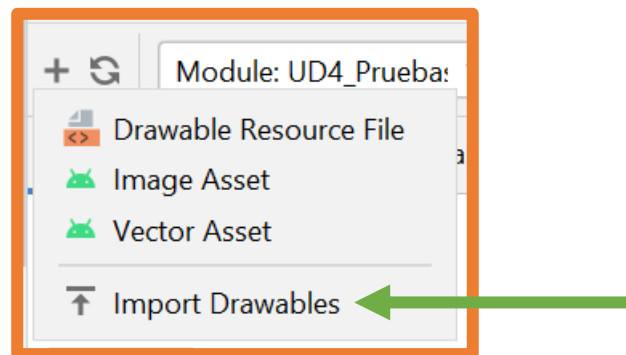
Con respecto a imágenes matriciales (jpg, png y webp) es preferible el uso de imágenes en formato **webp** por ello se explicará como convertir imágenes añadidas al proyecto en formato jpg y png a formato webp.

10.- Imágenes

Para añadir imágenes al proyecto se debe abrir el panel **Resource Manager** en la parte izquierda y desde él pulsar el botón +.

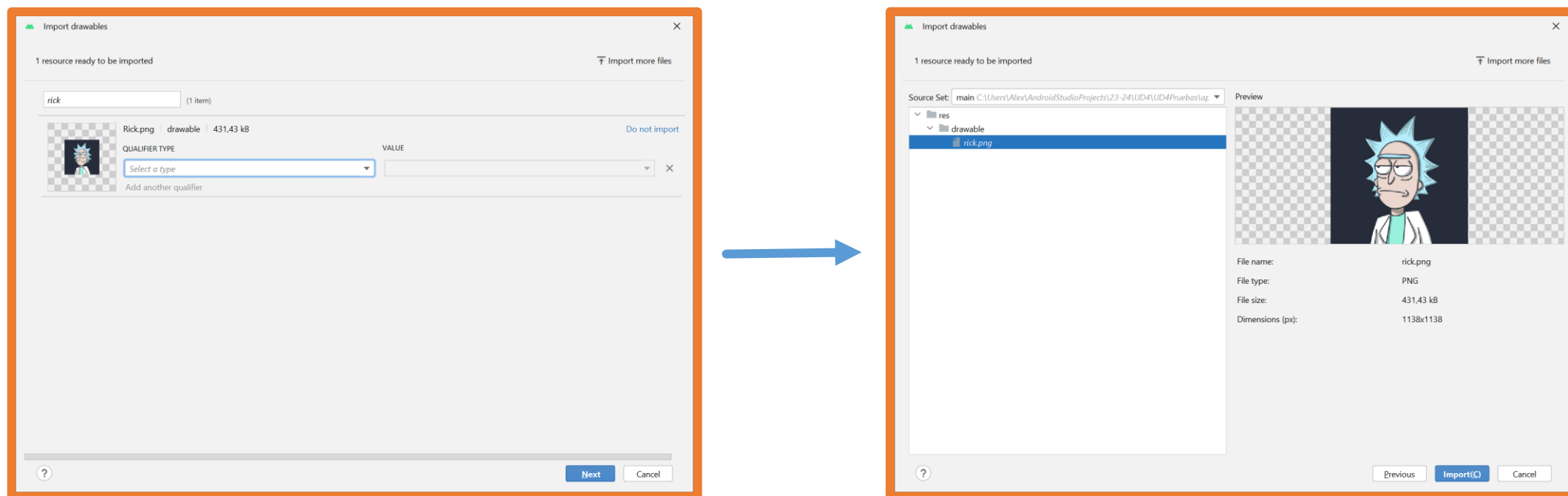


Para añadir imágenes matriciales (jpg, png y webp) se debe seleccionar la opción **Import Drawables**.



10.- Imágenes

Una vez seleccionada la imagen aparece la ventana de importación:

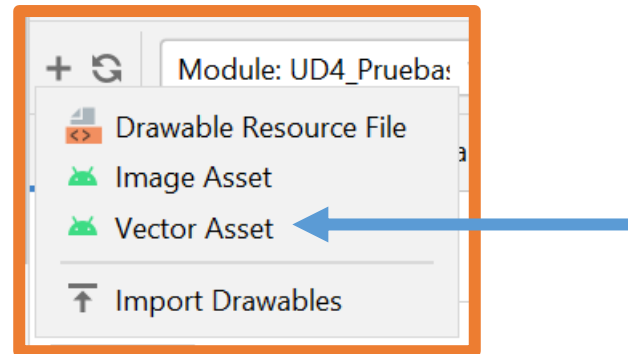


Si se quiere se puede seleccionar un **QUALIFIER**, esto permitiría cargar imágenes de diferentes tamaños según las características del dispositivo.

Por último se debe pulsar el botón **Import** y la imagen ya estará disponible.

10.- Imágenes

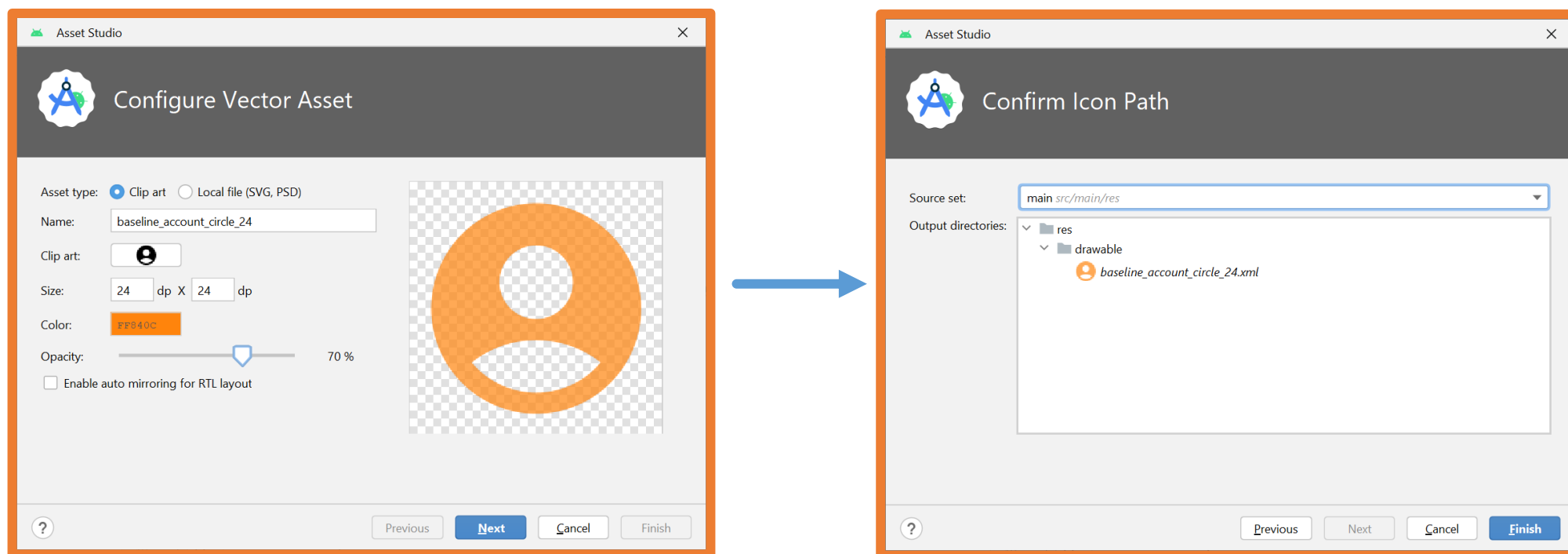
Para añadir imágenes vectoriales, generalmente iconos ofrecidos por Google, se debe seleccionar la opción **Vector Asset**.



Se abrirá una ventana que permite elegir el icono y configurar su nombre, tamaño, color y opacidad.

10.- Imágenes

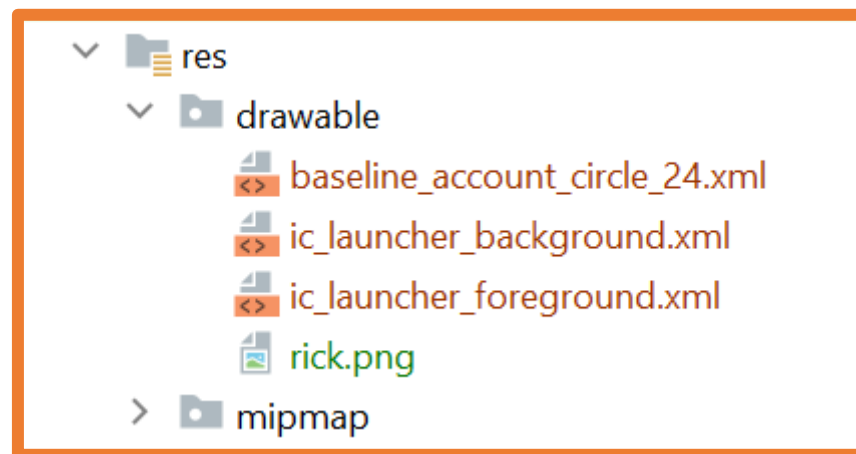
Para añadir imágenes vectoriales, generalmente iconos ofrecidos por Google, se debe seleccionar la opción **Vector Asset**.



Por último se debe pulsar el botón **Finish** y la imagen ya estará disponible.

10.- Imágenes

Las imágenes añadidas, sean del tipo que sean, se pueden consultar en la carpeta res.



Si se han seleccionado QUALIFIERS las imágenes aparecerán organizadas según eso QUALIFIERS.

10.- Imágenes

Conversión de imágenes matriciales a webp.

Google recomienda el uso de imágenes webp por estar más optimizadas y ocupar menos espacio.

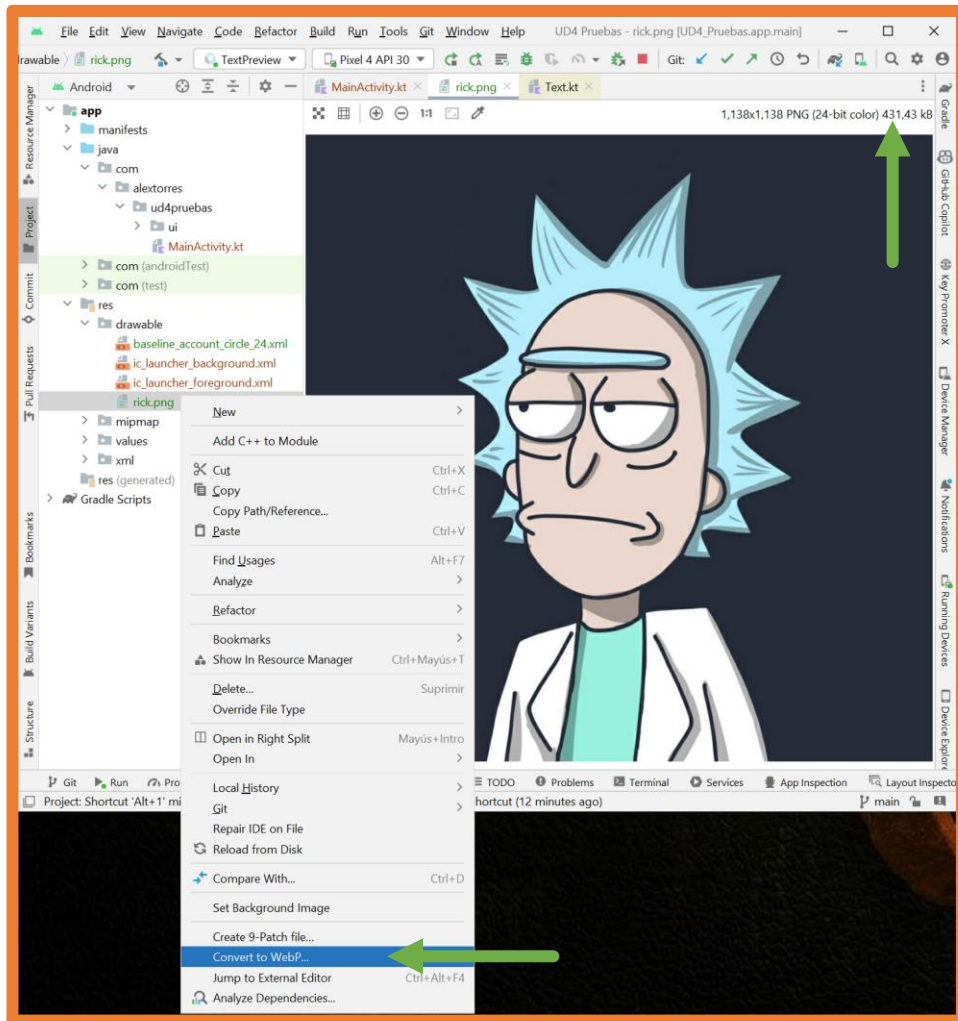
Con imágenes en formato webp se puede reducir el tamaño de la aplicación hasta un 80%.

El formato webp es compatible a partir de la API 14 con pérdida de calidad y de la API 18 sin pérdida de calidad.

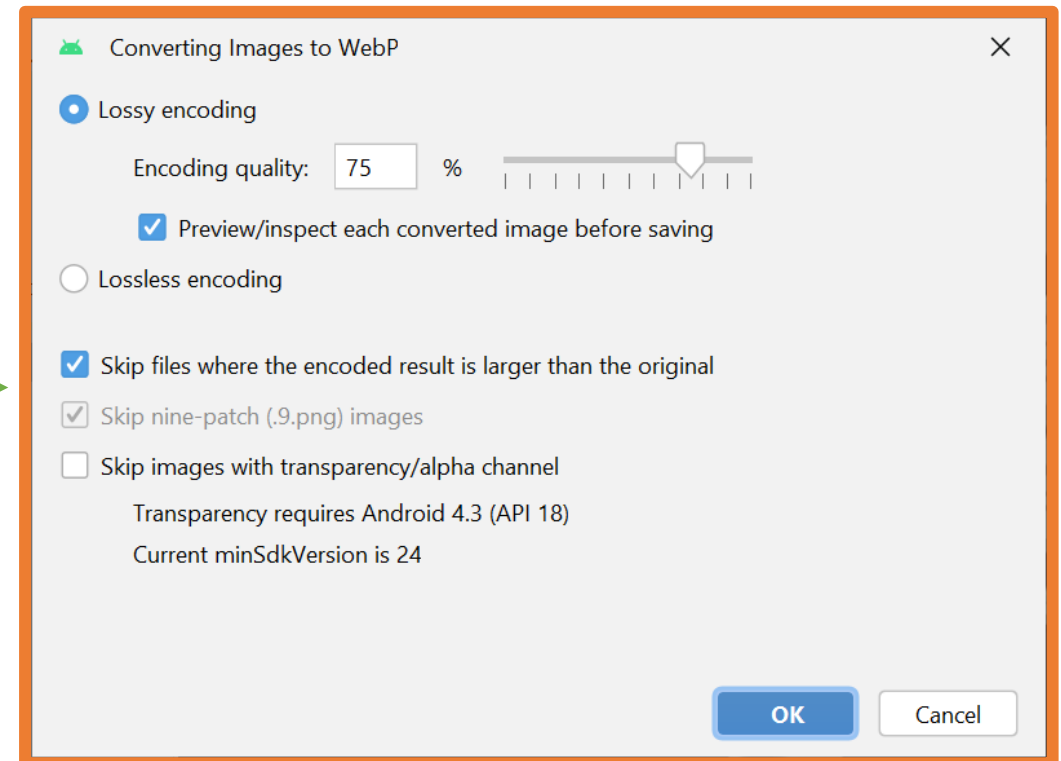
Como en el curso se trabaja con la API 24 no hay problemas de calidad.

10.- Imágenes

Sobre la imagen en el proyecto se hace clic derecho:

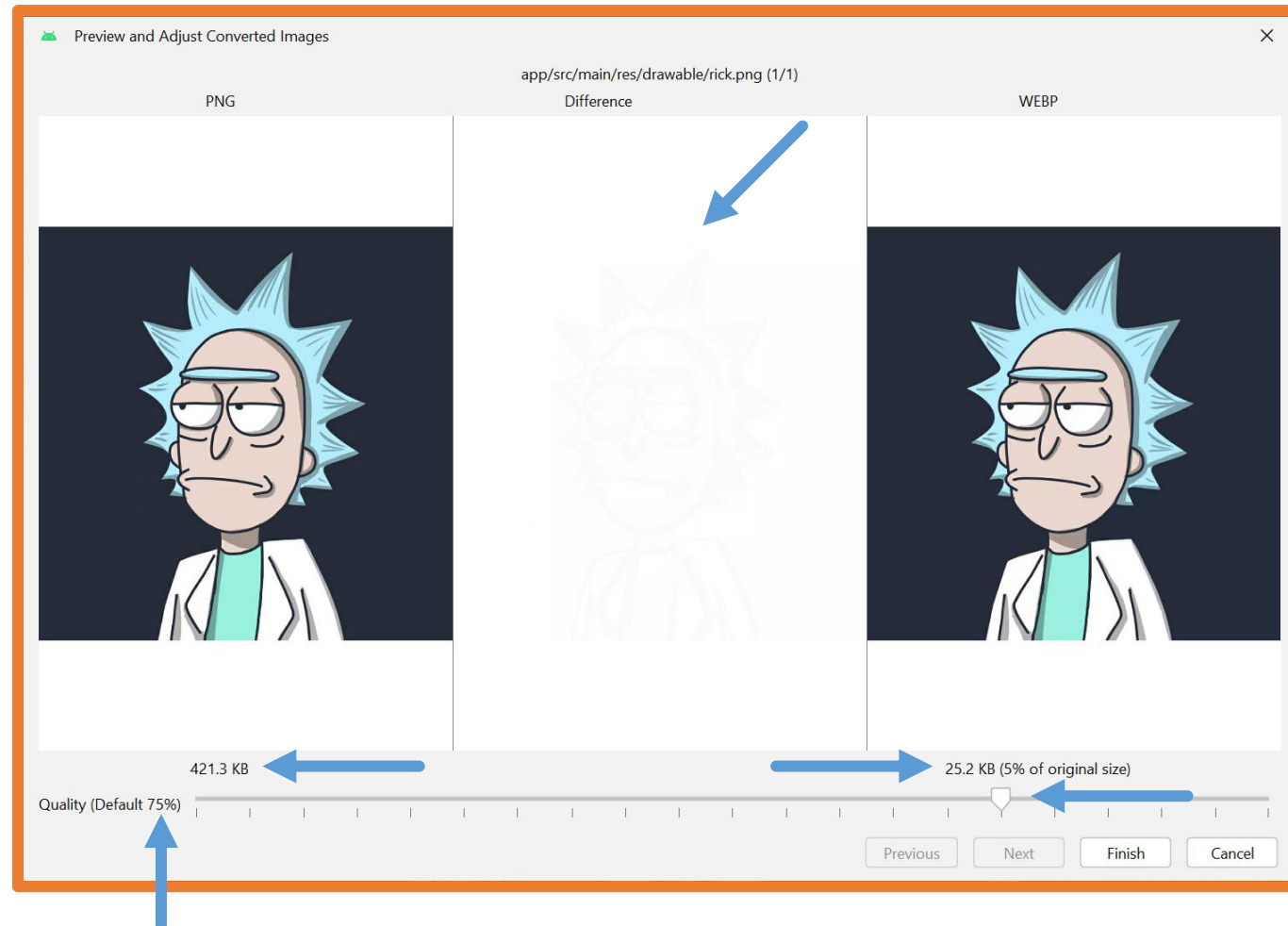


Se eligen las opciones que se necesiten:
100% o Lossless encoding → máxima calidad.



10.- Imágenes

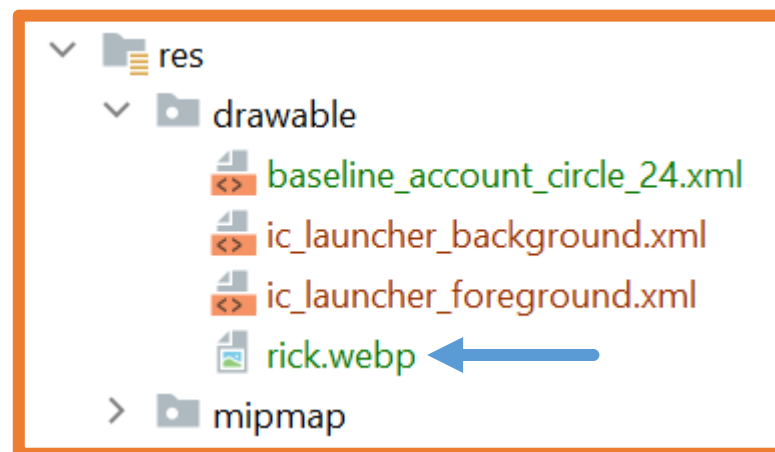
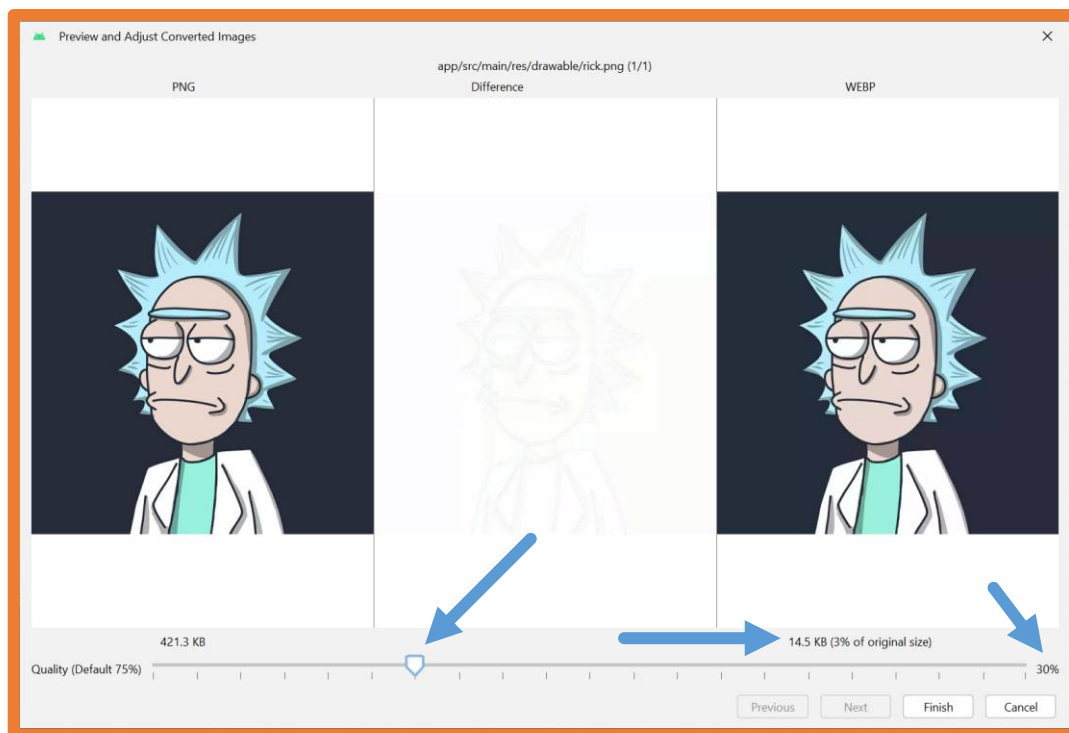
Se muestra una previsualización donde se puede ver la calidad final, el tamaño final y la diferencia. También se permite modificar la calidad final.



10.- Imágenes

La imagen del ejemplo a una calidad del 30% no tiene casi pérdida y pasa a ocupar 14,5 KB en lugar de los 421,3 KB originales (un 3,5% del tamaño original).

Para finalizar se pulsa el botón **Finish** y se convertirá la imagen a formato webp.



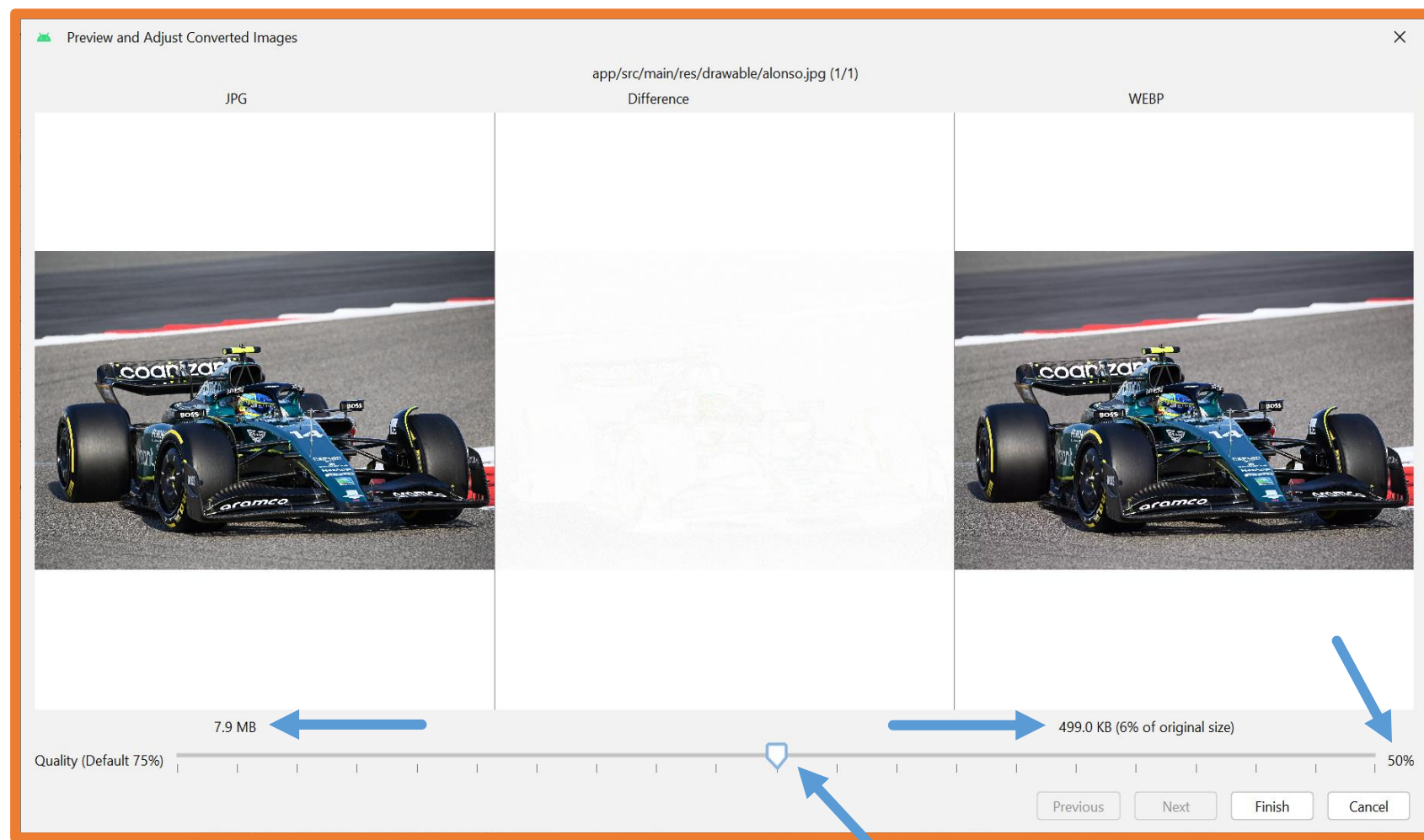
10.- Imágenes

Con imágenes más grande se ve con más detalle esa optimización.

Imagen original: 7,9 MB

Imagen final: 400 KB

Se ha reducido al 6,1%



11.- Componente Image

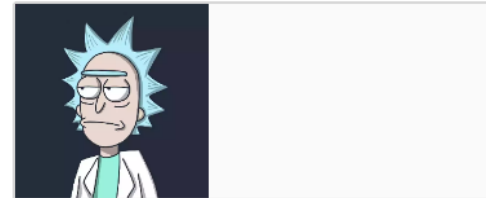
El componente **Image** permite cargar imágenes importadas al proyecto.

La función tiene dos parámetros obligatorios:

- painter: sirve para indicar el recurso a cargar.
- contentDescription: descripción de la imagen.

```
Image(  
    painter = painterResource(id = R.drawable.rick),  
    contentDescription = "Rick",  
)
```

ImagePreview



```
Image(  
    painter = painterResource(id = R.drawable.rick),  
    contentDescription = "Rick",  
    colorFilter = ColorFilter.tint(  
        color = Color.Red,  
        blendMode = BlendMode.Color  
    )  
)
```

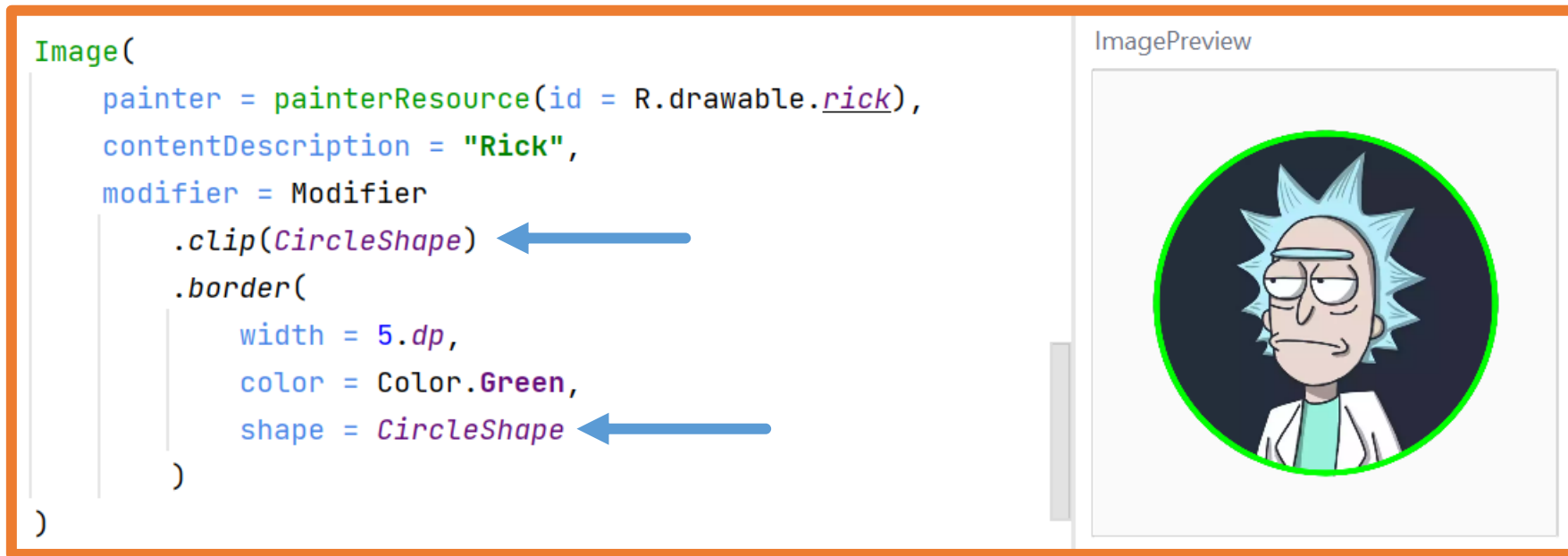
ImagePreview



11.- Componente Image

Con la función **Modifier.clip()** se puede dar forma a **cualquier componente**.

Si se asigna borde se debe asignar la misma forma que a la función clip.



11.- Componente Image

Si se indica a la imagen que ocupe todo el espacio disponible con el parámetro **ContentScale** se puede indicar cómo se debe comportar la imagen a la hora de rellenar el espacio:

Con **Crop**: se ajusta a la dimensión más pequeña del contenedor.

```
Image(  
    painter = painterResource(id = R.drawable.rick),  
    contentDescription = "Rick",  
    modifier = Modifier.fillMaxSize(),  
    contentScale = ContentScale.Crop  
)
```

ImagePreview



11.- Componente Image

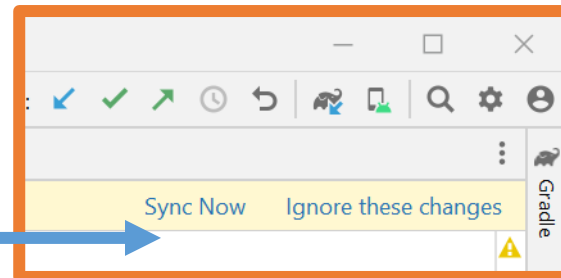
Con la librería **Coil** y el componente **AsyncImage** se pueden cargar imágenes de **internet**.

Se deben **dar permisos de acceso a internet** a la aplicación, para ello en el archivo del manifiesto: manifest → **AndroidManifest.xml** se debe añadir la siguiente línea:

```
xmlns:tools="http://schemas.android.com/tools">  
  
<uses-permission android:name="android.permission.INTERNET"/>  
  
<application
```

A continuación, se añade la dependencia al archivo **build.gradle.kts (Module: app)** y se hace clic en **Sync Now** en la parte superior derecha.

```
implementation("androidx.compose.material3:material3")  
  
implementation("io.coil-kt:coil-compose:2.0.0")  
  
testImplementation("junit:junit:4.13.2")
```



Cuando finalice la sincronización ya se podrá usar el componente **AsyncImage**.

11.- Componente Image

Como las previsualizaciones (@Preview) no tienen acceso a internet se debe lanzar la aplicación para ver el resultado.

```
AsyncImage(  
    model = "https://loremflickr.com/400/400/dragonball",  
    contentDescription = "Imagen Dragon Ball",  
    modifier = Modifier.fillMaxSize()  
)
```



AsyncImage también tiene el parámetro contentScale para indicar cómo rellena la imagen el espacio disponible.

11.- Componente Image

Se puede dar forma a las imágenes con funciones de **AsyncImage** y con la función **Modifier.clip()**.

```
AsyncImage(  
    model = ImageRequest.Builder(LocalContext.current)  
        .data("https://loremflickr.com/600/600/deathnote?random=10")  
        .transformations(CircleCropTransformation())  
        .build(),  
    contentDescription = "Imagen Death Note",  
    modifier = Modifier.fillMaxSize()  
)
```



```
AsyncImage(  
    model = ImageRequest.Builder(LocalContext.current)  
        .data("https://loremflickr.com/600/600/deathnote?random=10")  
        .transformations(  
            RoundedCornersTransformation(  
                topLeft = 100f,  
                topRight = 100f,  
                bottomLeft = 50f,  
                bottomRight = 20f  
            )  
        )  
        //RoundedCornersTransformation(100f)  
    ).build(),  
    contentDescription = "Imagen Death Note",  
    modifier = Modifier.fillMaxSize()  
)
```



```
AsyncImage(  
    model = "https://loremflickr.com/600/600/deathnote?random=10",  
    contentDescription = "Imagen Death Note",  
    modifier = Modifier.clip(RoundedCornerShape(50.dp))  
        modifier = Modifier.clip(CircleShape)  
)
```



Coil ofrece a función **crossfade** que indica a la imagen que aparezca poco a poco:

```
AsyncImage(  
    model = ImageRequest.Builder(LocalContext.current)  
        .data("https://loremflickr.com/600/600/deathnote?random=10")  
        // .crossfade(true)  
        .crossfade( durationMillis: 2000)  
        .build(),  
    contentDescription = "Imagen Death Note",  
)
```



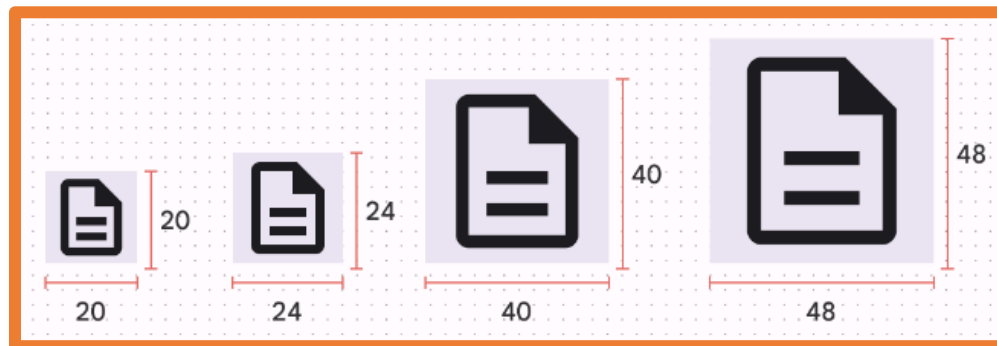
12.- Componente Icon

El componente **Icon** permite representar un icono en la aplicación, es similar a **Image** pero formando parte de **Material Design**.

Solo permite imágenes **vectoriales** o **rasterizadas** (convertidas a webp).

Solo se mostrará con un color.

Su tamaño por defecto es de 24dp pero soporta los siguientes tamaños:



12.- Componente Icon

En el ejemplo al usar una imagen como Icon se pierden los colores.



Con la clase **Icons** se pueden utilizar los iconos del sistema.

12.- Componente Icon

Lo más habitual es usar los iconos vectoriales de Material Design.

Android Studio solo carga por defecto algunos iconos. Si se quiere tener disponible todo el paquete de iconos se debe añadir la siguiente dependencia al archivo **build.gradle.kts (Module: app)**:

```
implementation("androidx.compose.material:material-icons-extended")
```

Recuerda pulsar **Sync Now** tras añadir la dependencia.

Una vez añadida la dependencia ya están disponibles [todos los iconos](#).

```
Icon(  
    imageVector = Icons.Rounded.Play,  
    contentDescription = "Play"  
)
```



12.- Componente Icon

En la clase **Icon** existen subclases de iconos:

- **Default:** como usar Filled.
- **Filled:** icono relleno del mismo color.
- **Outlined:** icono solo con los bordes.
- **TwoTone:** icono con dos colores.
- **Sharp:** icono con las esquinas anguladas.
- **Rounded:** icono con las esquinas redondeadas.

```
Icon(  
    imageVector = Icons.Default.PlayArrow,  
    contentDescription = "Play",  
)  
Icon(  
    imageVector = Icons.Outlined.PlayArrow,  
    contentDescription = "Play",  
)  
Icon(  
    imageVector = Icons.TwoTone.PlayArrow,  
    contentDescription = "Play",  
)  
Icon(  
    imageVector = Icons.Sharp.PlayArrow,  
    contentDescription = "Play",  
)  
Icon(  
    imageVector = Icons.Rounded.PlayArrow,  
    contentDescription = "Play",  
)
```



12.- Componente Icon

Con el parámetro **tint** se puede cambiar la tonalidad del icono.

```
Icon(  
    imageVector = Icons.Default.PlayArrow,  
    contentDescription = "Play",  
    tint = Color.Red  
)  
Icon(  
    imageVector = Icons.Rounded.PlayArrow,  
    contentDescription = "Play",  
    tint = Color.Green  
)  
Icon(  
    imageVector = Icons.TwoTone.PlayArrow,  
    contentDescription = "Play",  
    tint = Color.Blue  
)
```

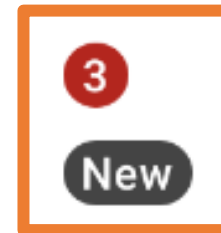


13.- Componente Badge

El componente **Badge** permite mostrar información dinámica como el número de mensajes pendientes.

Un Badge solo debería tener un icono o un texto corto.

```
Badge { this: RowScope  
  Text( text: "3")  
}  
  
Badge(  
  containerColor = Color.DarkGray,  
  contentColor = Color.White  
) { this: RowScope  
  Text( text: "New")  
}
```



14.- Componente BadgedBox

El componente **BadgedBox** permite mostrar información dinámica como el número de mensajes pendientes **sobre otro elemento** como un icono.

Se usan habitualmente en barras de navegación.

```
BadgedBox(  
  badge = {  
    this: BoxScope  
    Badge {  
      this: RowScope  
      Text( text: "123")  
    }  
  }) {  
    this: BoxScope  
    Icon(  
      Icons.Filled.Mail,  
      contentDescription = "Correo"  
    )  
  }  
)
```



```
BadgedBox(  
  badge = {  
    this: BoxScope  
    Badge(  
      containerColor = Color.LightGray,  
      contentColor = Color.Red  
    ) {  
      this: RowScope  
      Text( text: "123")  
    }  
  }) {  
    this: BoxScope  
    Icon(  
      Icons.Filled.Mail,  
      contentDescription = "Correo",  
      tint = Color.Red  
    )  
  }  
)
```



15.- Componentes Divider

Existen tres componentes de tipo Divider:

- **Divider**: disponible hasta material3 v1.1.1, **obsoleto en material3 v1.2.0**.
- **HorizontalDivider**: disponible en material3 v1.2.0.
- **VerticalDivider**: disponible en material3 v1.2.0.

Como la versión 1.2.0 se encuentra en Alpha, aún se puede utilizar el componente Divider.

Si se quiere usar HorizontalDivider y VerticalDivider hay que actualizar la dependencia en **build.gradle.kts (Module)** y sincronizar.

15.- Componentes Divider

Divider (línea horizontal), **HorizontalDivider** y **VerticalDivider** dibujan una línea que permite separar elementos generalmente en filas (Row) y columnas (Column).

```
Column { this: ColumnScope
    Text( text: "1")
    HorizontalDivider()
    Text( text: "2")
    HorizontalDivider()
    Text( text: "3")
}

Spacer(modifier = Modifier.height(8.dp))

Row(modifier = Modifier.height(50.dp)) { this: RowScope
    Text( text: "1")
    VerticalDivider()
    Text( text: "2")
    VerticalDivider()
    Text( text: "3")
}
```



Los tres componentes disponen de los mismos parámetros: modifier, thickness y color.

```
@Composable
fun HorizontalDivider(
    modifier: Modifier = Modifier,
    thickness: Dp = DividerDefaults.Thickness,
    color: Color = DividerDefaults.color,
) = Canvas(modifier.fillMaxWidth().height(thickness))
```

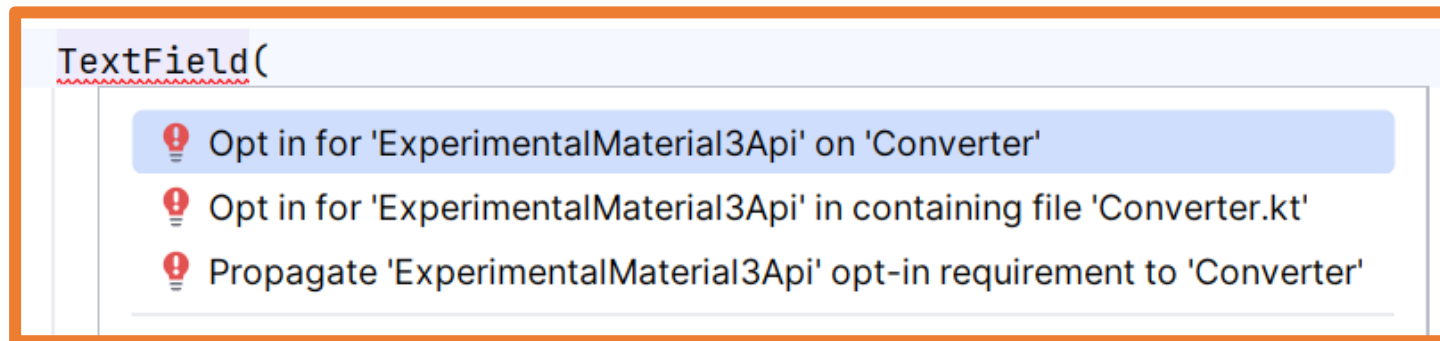
16.- ExperimentalMaterial3Api

Hay ocasiones que los componentes de Jetpack Compose se encuentran en fase experimental pero se pueden utilizar sin ningún problema.

En esas situaciones se debe indicar en Android Studio esta característica.

El propio Android Studio en la ayuda contextual si se quiere indicar en:

- Al inicio del componente que contiene al componente experimental
- En todo el archivo actual
- Si se quiere propagar y que se tenga que indicar en el componente padre al componente que contiene al componente experimental.



Práctica

Actividad 1

Perfil personal.