Chapter 03: Develop Android apps with Kotlin

Part 1

1-What is a user interface (UI)?. 2- Composable functions. 3- Add Paddings. 4- Arrange the text elements in a row and column (UI Hierarchy). 5- Resources in Jetpack Compose. 6- Layout Modifiers.

otlin - JetPack Compose

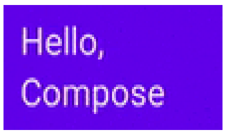
Kotlin is a modern statically typed programming language used by over 60% of professional Android developers that helps boost productivity, developer satisfaction, and code safety.

Jetpack Compose is a modern toolkit for building Android UIs. Compose simplifies and accelerates UI development on Android with less code, powerful tools, and intuitive Kotlin capabilities. With Compose, we can build our UI by defining a set of functions, called composable functions.

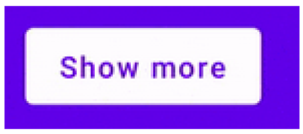
1. What is a user interface (UI)?

The user interface (UI) of an app is what you see on the screen: text, images, buttons, and many other types of elements, and how it's laid out on the screen. It’s how the app shows things to the user and how the user interacts with the app.

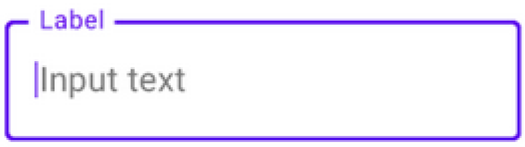
We present here a clickable button, text message, and text-input field where users can enter data.



Text message



Clickable button



Text-input field

2. Composable functions

Composable functions are the basic building block of a UI in Compose. A composable function:

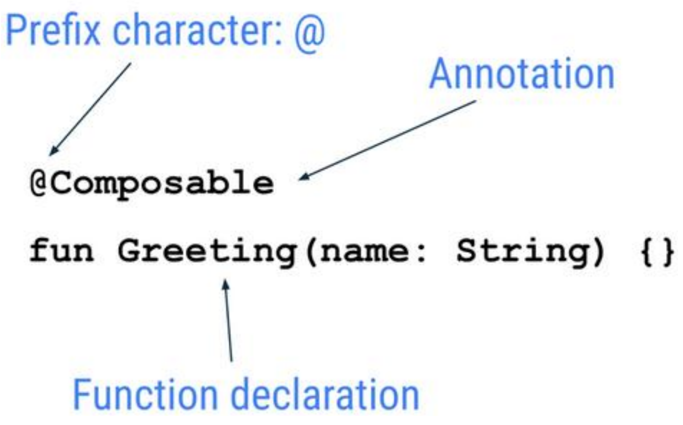
- Describes some part of our UI.
- Doesn't return anything.
- Takes some input and generates what's shown on the screen.
- Might emit several UI elements.

2.1 Annotations

Annotations are means of attaching extra information to code. This information helps tools like the Jetpack Compose compiler, and other developers understand the app's code.

An annotation is applied by prefixing its name (the annotation) with the **@** character at the beginning of the declaration. Different code elements, including properties, functions and classes, can be annotated.

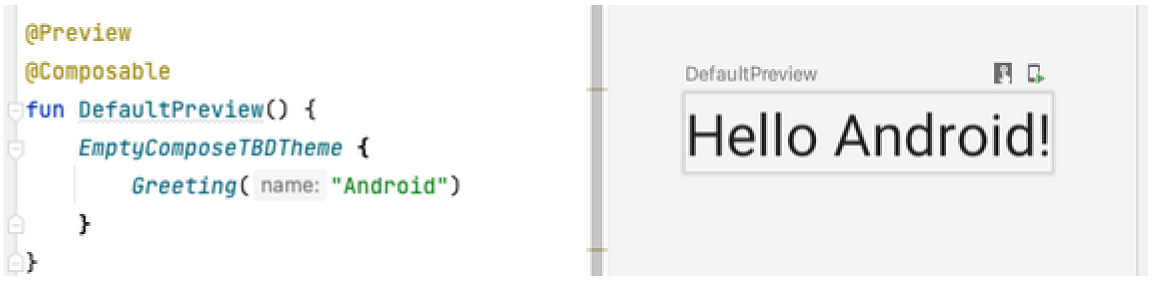
The following diagram is an example of annotated function:



```
@Composable
fun Greeting(name: String) {}
```

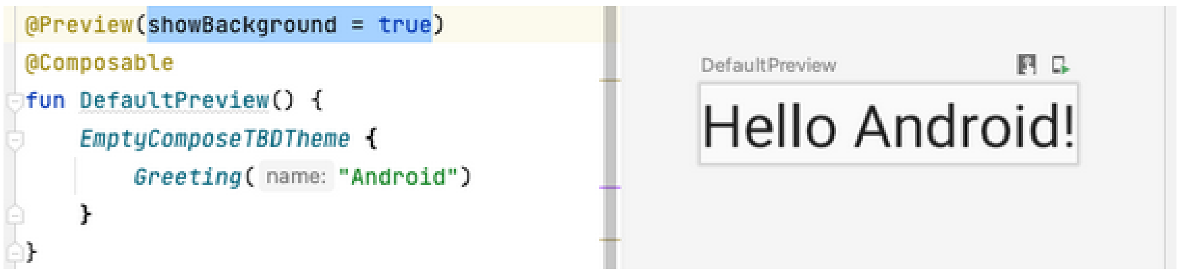
Annotations can take parameters. They provide extra information to the tools processing them. The following are some examples of **@Preview** annotations with and without parameters.

- *Annotation without parameters*



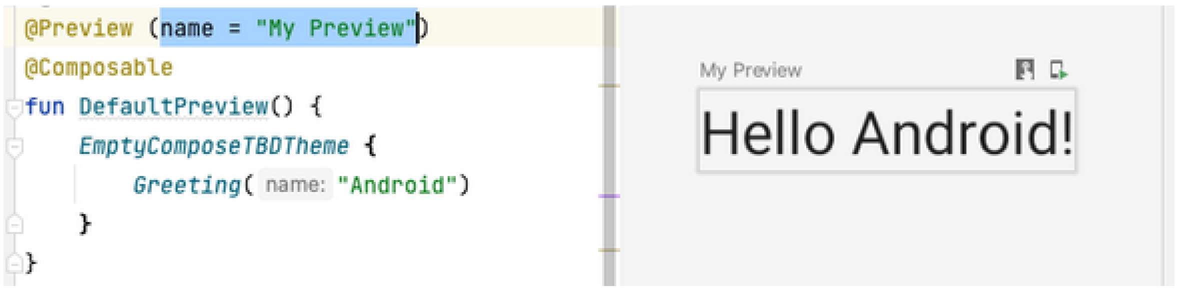
```
@Preview
@Composable
fun DefaultPreview() {
    EmptyComposeTBDTheme {
        Greeting( name: "Android")
    }
}
```

- *Annotation previewing background*



```
@Preview(showBackground = true)
@Composable
fun DefaultPreview() {
    EmptyComposeTBDTheme {
        Greeting( name: "Android")
    }
}
```

- *Annotation with a preview title*



```
@Preview (name = "My Preview")
@Composable
fun DefaultPreview() {
    EmptyComposeTBDTheme {
        Greeting( name: "Android")
    }
}
```

2.2 Example of Composable function

The Composable function is annotated with the @Composable annotation. All composable functions must have this annotation. This annotation informs the Compose compiler that this function is intended to convert data into UI. As a reminder, a compiler is a special program that takes the code you wrote, looks at it line by line, and translates it into something the computer can understand (machine language).

This code snippet is an example of a simple composable function that is passed data (the name function parameter) and uses it to render a text element on the screen.

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

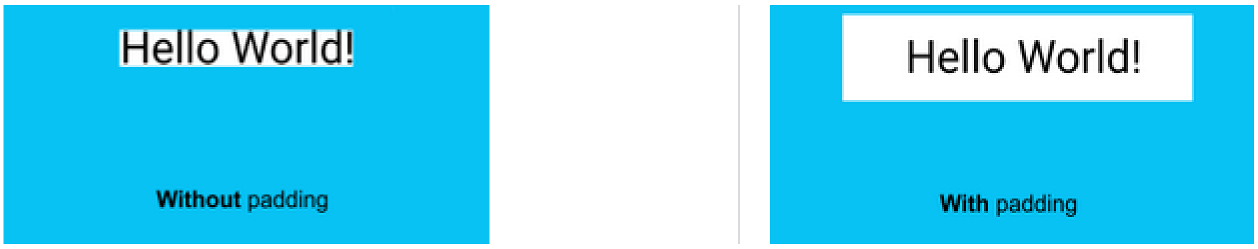
- **A few notes about the composable function:**

- Composable functions can accept parameters, which let the app logic describe or modify the UI. In this case, our UI element accepts a **String** so that it can greet the user by name.

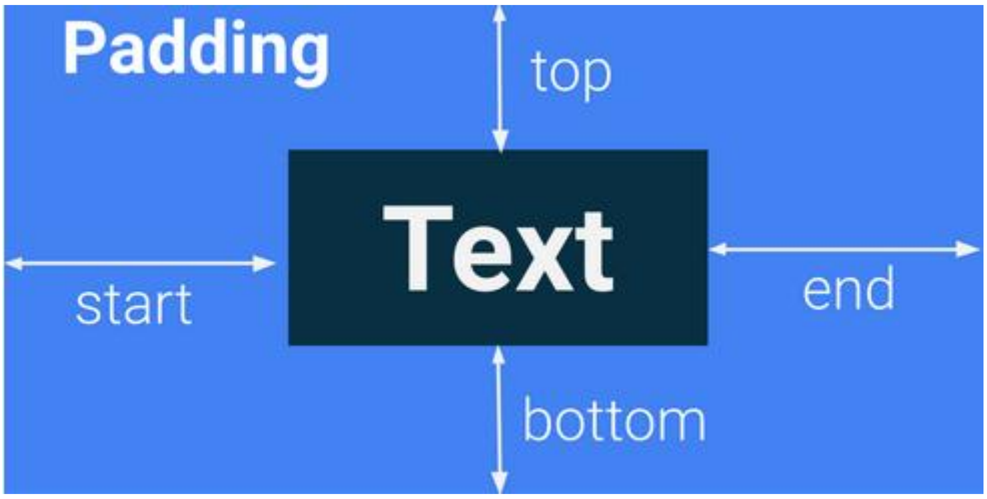
The function doesn't return anything. Composable functions that emit UI don't need to return anything because they describe the desired screen state instead of constructing UI elements. In other words, composable functions only describe the UI, they don't construct or create the UI, so there is nothing to return.

3. Add Paddings:

A UI element wraps itself around its content. To prevent it from wrapping too tightly, we can specify the amount of *padding* on each side.



Padding is used as a modifier, which means that you can apply it to any composable. For each side of the composable, the padding modifier takes an optional argument that defines the amount of padding in **dp** or **sp** unit.



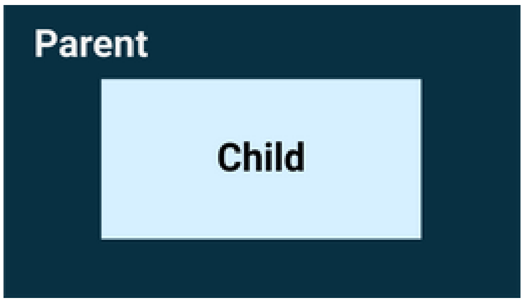
The **`scalable pixels (SP)`** is a unit of measure for the font size. UI elements in Android apps use two different units of measurement: **`density-independent pixels (DP)`**, which you use later for the layout, and scalable pixels (SP). By default, the SP unit is the same size as the DP unit, but it resizes based on the user's preferred text size under phone settings.
Example using Modifier:

```
// This is an example.
Modifier.padding(
    start = 16.dp,
    top = 16.dp,
    end = 16.dp,
    bottom = 16.dp
)
```

4. Arrange the text elements in a row and column (UI Hierarchy)

The UI hierarchy is based on containment, meaning one component can contain one or more components, and the terms parent and child are sometimes used. The context here is that the parent

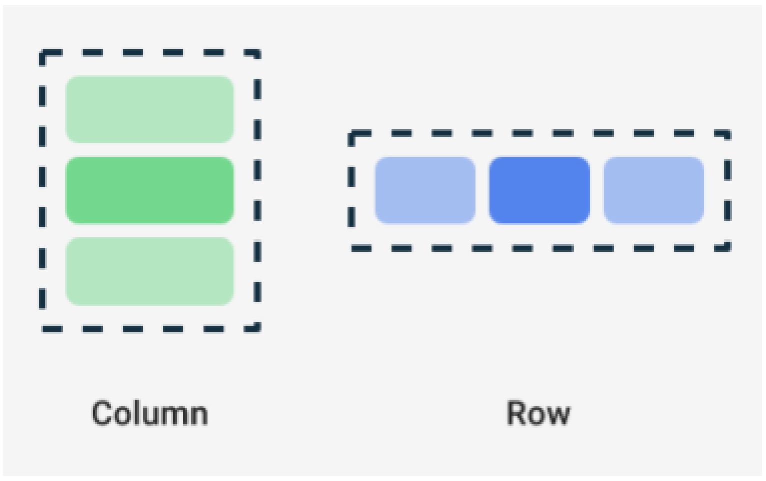
UI elements contain children UI elements, which in turn can contain children UI elements. In this section, we will learn about Column, Row and Box composables, which can act as parent UI elements.



The three basic and standard layout elements in Compose are Column, Row and Box composables.

4.1 Rows and Columns

Column, Row, and Box are composable functions that take composable content as arguments, so we can place items inside these layout elements.



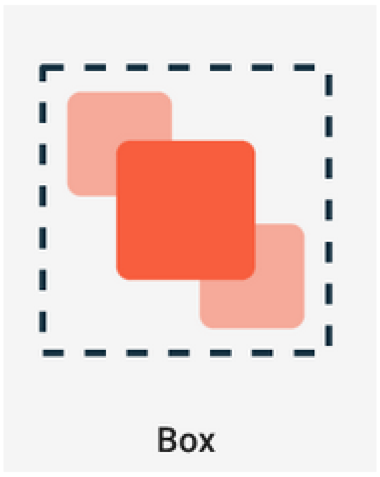
For example, each child element inside a Row composable is placed horizontally next to each other in a row.

```
// Don't copy.
Row {
    Text("First column")
    Text("Second column")
}
```

These text elements display next to each other on the screen as seen in this image. The blue borders are only for demonstration purposes and don't display.

4.2 Box Composable

Box layout is one of the standard layout elements in Compose. Use **Box** layout to stack elements on top of one another. **Box** layout also lets we configure the specific alignment of the elements that it contains.



A Box Composable can be used around UI elements. For example around an image and text composables:

First we declare the image with help of a `val` property as follows:

```
val image = painterResource(R.drawable.androidparty)
```

A Composable Image should be defined as:

```
Image(
    painter = image,
    contentDescription = null
)
```

Add Box layout:

```
Box {
    Image(
        painter = image,
        contentDescription = null
    )
            GreetingWithText(message = message, from = from)
}
```

In this case the Box composable incorporates the two objects or composable elements: the image and text.

5. Resources in Jetpack Compose

Resources are the additional files and static content that your code uses, such as bitmaps, user-interface strings, animation instructions, and more.

We should always separate app resources, such as images and strings, from our code so that we can maintain them independently. At runtime, Android uses the appropriate resource based on the current configuration. For example, we might want to provide a different UI layout based on the screen size or different strings based on the language setting.

5.1 Grouping resources

We should always place each type of resource in a specific subdirectory of our project's res/ directory. For example, here's the file hierarchy for a simple project:

```
MyProject/
    src/
        MyActivity.kt
    res/
        drawable/
            graphic.png
        mipmap/
            icon.png
        values/
            strings.xml
```

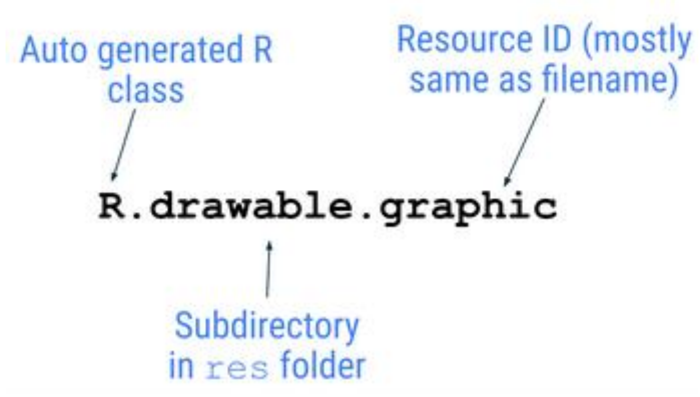
As we can see in this example, the res/ directory contains all the resources in subdirectories, which includes a drawable/ directory for an image resource, a mipmap/ directory for launcher icons, and a values/ directory for string resources. To learn more about the usage, format, and syntax for app resources, see Resource types overview[1].

5.2 Accessing resources

Jetpack Compose can access the resources defined in our Android project. Resources can be accessed with resource IDs that are generated in our project's R class.

An R class is an automatically generated class by Android that contains the IDs of all resources in the project. In most cases, the resource ID is the same as the filename. For example, the image in the previous file hierarchy can be accessed with this code:

[1] https://developer.android.com/guide/topics/resources/available-resources


Auto generated R class
Resource ID (mostly same as filename)
R.drawable.graphic
Subdirectory in res folder

5.3 The R class:

Le SDK Android construit automatiquement cette classe statique appelée **R**. Elle ne contient que des constantes groupées par catégories. Elle est générée automatiquement (dans le dossier generated) par ce que nous mettons dans le dossier **`res`** : des interfaces, icons, images, chaînes. . . Certaines de ces ressources sont des fichiers **XML**, d'autres sont des images PNG.

Par exemple, le fichier **`res/values/strings.xml`** :

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">Exemple</string>
    <string name="message">Bonjour !</string>
<resources>
```

Cela rajoute automatiquement deux chaînes dans **`R.string`** : app_name et message.

6. Layout Modifiers

Modifiers are used to decorate or add behavior to Jetpack Compose UI elements. For example, we can add backgrounds, padding or behavior to rows, text, or buttons. To set them, a composable or a layout needs to accept a modifier as a parameter.

In the section 3, we learned about modifiers and used the padding modifier (Modifier.padding) to add space around Text composable. Modifiers can do a lot and we will see that in this and upcoming pathways.

For example, this Text composable has a Modifier argument that changes the background color to green.

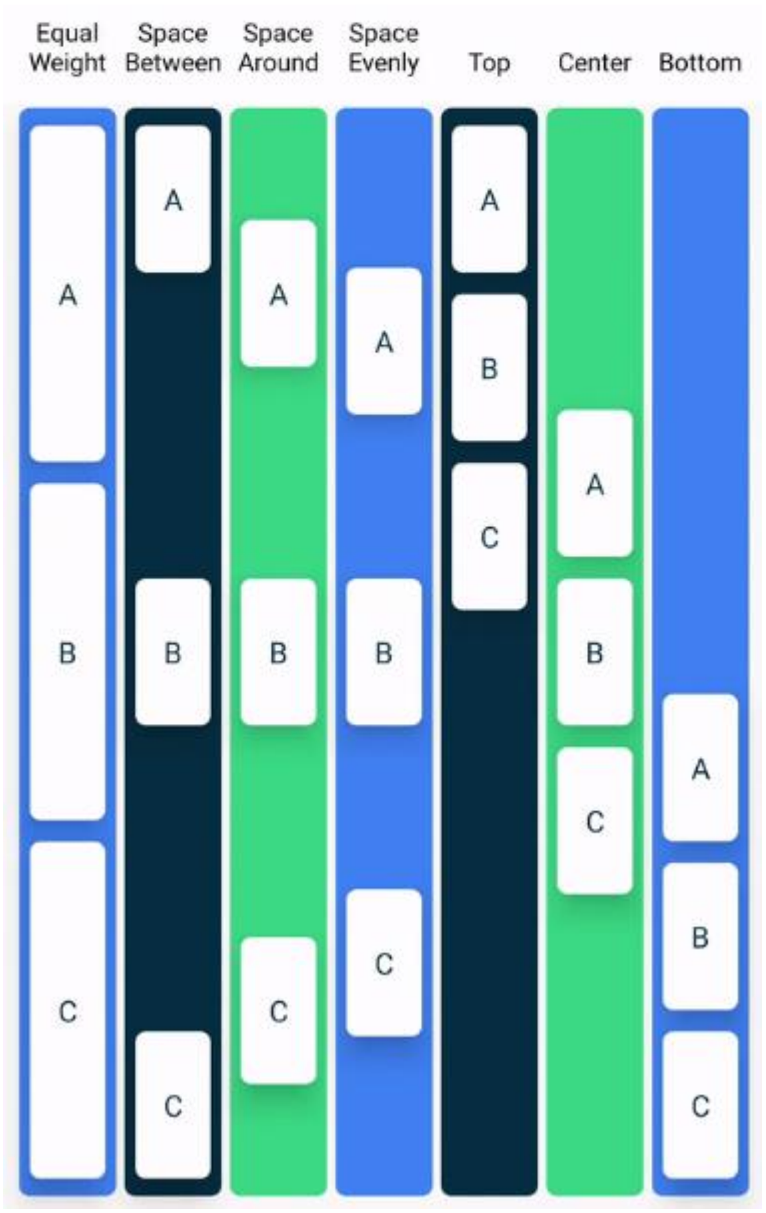
```
//Example
Text(
    text = "Hello, World!",
    // Solid element background color
    modifier = Modifier.background(color = Color.Green)
)
```

Similar to the above example, we can add Modifiers to layouts to position the child elements using arrangement and alignment properties.

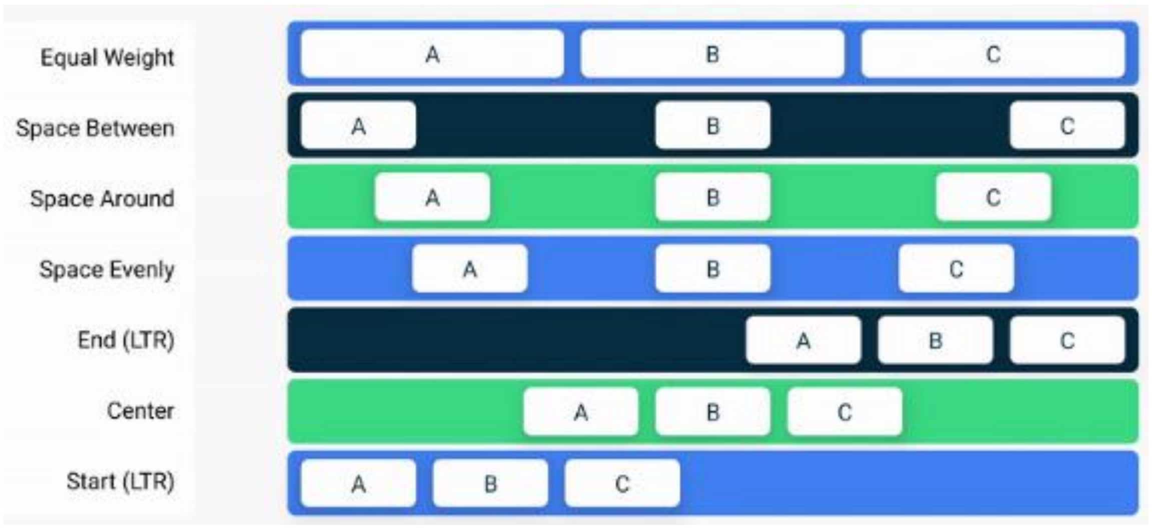
To set children's position within a Row, set the **horizontalArrangement** and **verticalAlignment** arguments. For a Column**,** set the **verticalArrangement** and **horizontalAlignment** arguments.

The arrangement property is used to arrange the child elements when the size of the layout is larger than the sum of its children.

For example: when the size of the Column is larger than the sum of its children sizes, a verticalArrangement can be specified to define the positioning of the children inside the Column. Below is an illustration of different vertical arrangements:



Similarly, when the size of the Row is larger than the sum of its children sizes, a **horizontalArrangement** can be specified to define the positioning of the children inside the Row. Below is an illustration of different horizontal arrangements:



The alignment property is used to align the child elements at the start, center, or end of layout.