Build a Basic Layout.

1.3: Build a Basic Layout.

▼ Build a Simple App with Text Composables.

What is Jetpack Compose?

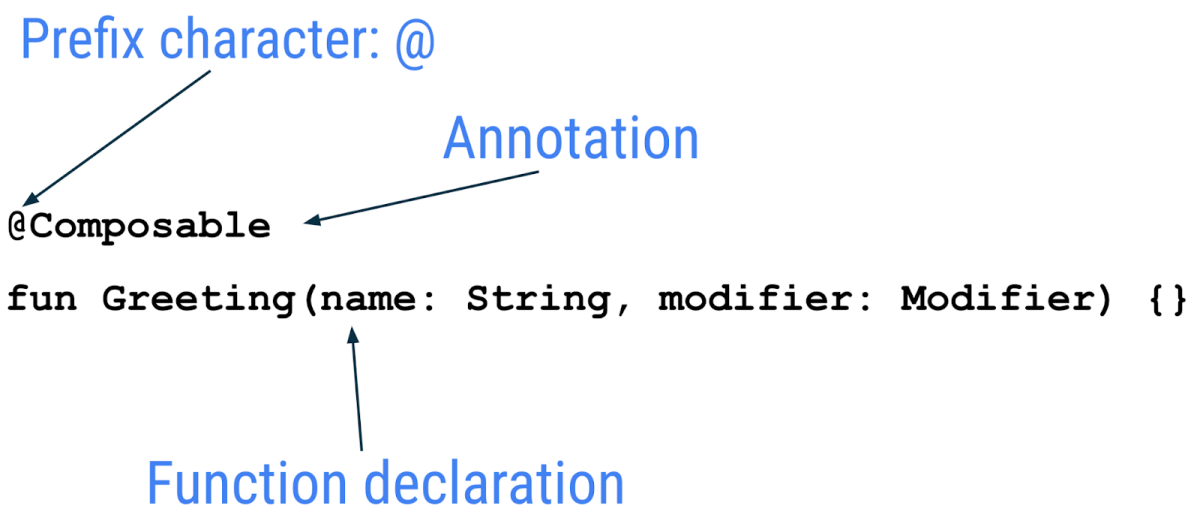
The user interface (UI) of an app is what you see on the screen: text, images, buttons, and many other types of elements, and how it's laid out on the screen. It's how the app shows things to the user and how the user interacts with the app.

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, you can build your UI by defining a set of functions, called composable functions, that take in data and describe UI elements.

Composable functions are the basic building blocks of a UI in Compose. They:

- Describe some part of your UI.
- Don’t return anything.
- Take some input and generates what’s shown on the screen.

Annotations are means of attaching extra information to code. An annotation is applied by prefixing its name (the annotation) with the `@` character at the beginning of the declaration you are annotating.



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.

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

The compose function that returns nothing and bears the `@Composable` annotation MUST be named using Pascal case. Pascal case refers to a naming convention in which the first letter of each word in a compound word is capitalized. The difference between Pascal case and camel case is that all words in Pascal case are capitalized.

The Compose function:

- *MUST* be a noun: `DoneButton()`
- *NOT* a verb or verb phrase: `DrawTextField()`
- *NOT* a nouned preposition: `TextFieldWithLink()`
- *NOT* an adjective: `Bright()`
- *NOT* an adverb: `Outside()`
- Nouns *MAY* be prefixed by descriptive adjectives: `RoundIcon()`

Add a New Text Element.

It's a best practice to have your Composable accept a `Modifier` parameter, and pass that `modifier` to its first child.

```
@Composable
fun GreetingText(message: String, from: String, modifier: Mod
    Text(
        text = message,
                fontSize = 100.sp,
                lineHeight = 116.sp,
    )
        Text(
        text = from
    )
}
```

```
@Preview(showBackground = true)
@Composable
fun BirthdayCardPreview() {
    HappyBirthdayTheme {
        GreetingText(message = "Happy Birthday Sam!")
    }
}
```

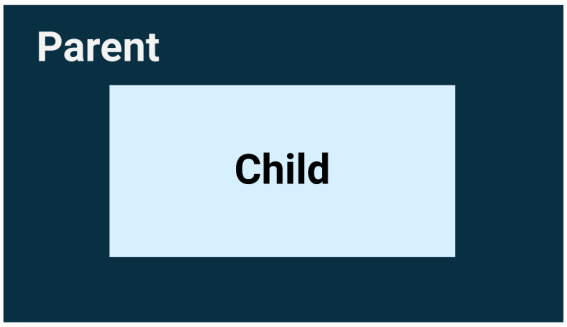
The *scalable pixels* (SP) is a unit of measure for the font size. UI elements in Android apps use 2 different units of measurement: *density-independent pixels* (DP), which you use later for the layout, and scalable pixels (SP).

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.

Arrange the Text Elements in a Row and Column.

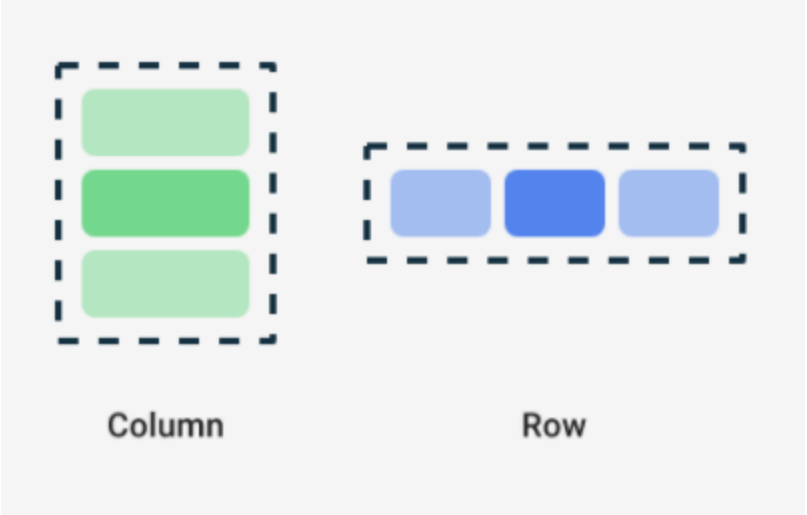
The UI hierarchy is based on containment, meaning one component can contain one or more components, and the terms parent and child are

sometimes used. The context here is that the parent UI elements contain children UI elements, which in turn can contain children UI elements.



The 3 basic layout elements (composables) in Compose are:

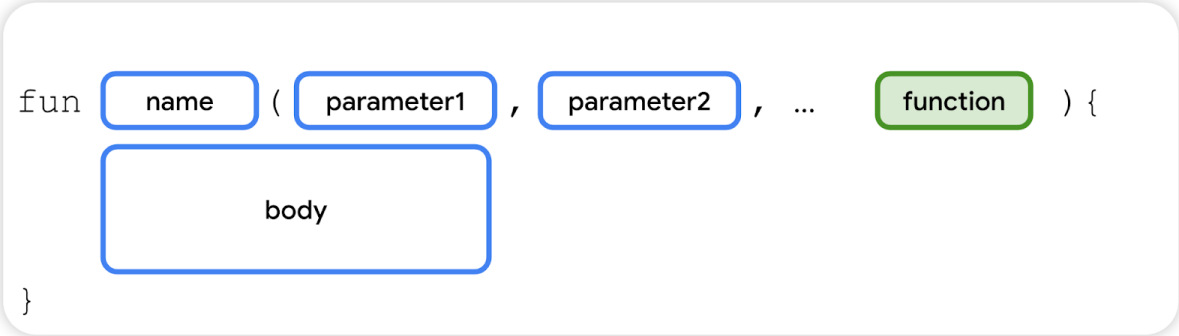
- Column.
- Row.
- Box.



💡 `Column`, `Row`, and `Box` are composable functions that take composable content as arguments, so you 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.

```
Row {
    Text("First Column")
    Text("Second Column")
}
```

Kotlin offers a special syntax for passing functions as parameters to functions, when the last parameter is a function:



When you pass a function as that parameter, you can use trailing lambda syntax. Instead of putting the function inside the parentheses, you can place it outside the parentheses in curly braces.

The trailing lambda syntax in Kotlin is a convenience feature that makes the code more readable, especially when the lambda expression is large or the last parameter of a function.

To understand this, let's break down the concepts involved:

1. **Lambda Expressions**: A lambda expression is essentially an unnamed (anonymous) function. It's defined with curly braces `{}` and can take parameters and return values.

2. **Functions as Parameters**: In Kotlin, functions can take other functions as parameters. This is useful in many scenarios, such as creating higher-order functions, which are functions that operate on functions (taking them as parameters or returning them).
3. **Trailing Lambda Syntax**: When the last parameter of a function is a lambda expression, Kotlin allows you to pass it outside of the function's parentheses. This is particularly useful in making the code look cleaner and more readable.

Imagine you have a function `doOperation` that takes 2 parameters, an integer, and a function (lambda expression):

```
fun doOperation(x: Int, operation: (Int) -> Unit) {
    // ... some code
    operation(x)
}
```

You can call this function and pass a lambda as the last parameter in two ways:

1. **Regular Syntax** (without trailing lambda):

```
doOperation(5, { number ->
    println(number * number)
})
```

2. **Trailing Lambda Syntax**:

```
doOperation(5) { number ->
    println(number * number)
}
```

In the trailing lambda syntax, since the lambda is the last parameter, you can omit the parentheses when calling the function, which makes it look cleaner.

In the context of Jetpack Compose, which makes extensive use of trailing lambdas, this becomes particularly elegant. Composable functions, like `Row`, often take a lambda as the last parameter to define their content.

Using the trailing lambda syntax, you can write composables in a way that resembles a domain-specific language (DSL), which is more readable and concise:

```
Row {
    Text("Some text")
    Text("Some more text")
    Text("Last text")
}
```

This is syntactic sugar that improves the readability of the code when dealing with function types as parameters. It's especially common in DSLs and UI frameworks like Jetpack Compose, where you often define UI elements in a hierarchical and nested manner.

```
@Composable
fun GreetingText(message: String, from: String, modifier: Mod
    Column(
        verticalArrangement = Arrangement.Center,
        modifier = modifier
    ) {
        Text(
            text = message,
            fontSize = 100.sp,
            lineHeight = 116.sp,
            textAlign = TextAlign.Center
        )
        Text(
            text = from,
            fontSize = 36.sp,
            modifier = Modifier
                .padding(16.dp)
```

```
                    .align(alignment = Alignment.End)
            )
        }
    }
```

Conclusion.

- Jetpack Compose is a modern toolkit for building Android UI. Jetpack Compose simplifies and accelerates UI development on Android with less code, powerful tools, and intuitive Kotlin APIs.
- The UI of an app is what you see on the screen: text, images, buttons, and many other types of elements.
- Composable functions are the basic building block of Compose. A composable function is a function that describes some part of your UI.
- The Composable function is annotated with the `@Composable` annotation; this annotation informs the Compose compiler that this function is intended to convert data into UI.
- The 3 basic standard layout elements in Compose are `Column`, `Row`, and `Box`. They are Composable functions that take Composable content, so you can place items inside. For example, each child within a `Row` will be placed horizontally next to each other.

▼ Add Images to Your Android App.

Set Up Your App.

Follow these steps to add an image to your Android app.

Add an Image Composable.

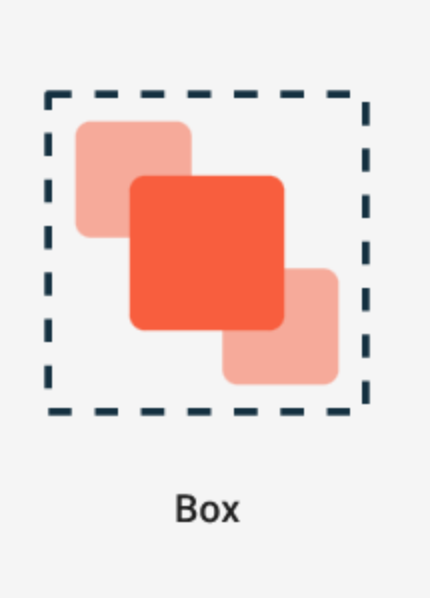
Just like you use a `Text` composable to display text, you can use an `Image` composable to display an image.

The `painterResource()` function loads a drawable image resource and takes resource ID (`R.drawable.androidparty` in this case) as an argument.

```
@Composable
fun GreetingImage(message: String, from: String, modifier: Mo
        val image = painterResource(R.drawable.androidparty)

        Box(modifier) {
                Image(
                        painter = image,
                        contentDescription = null,
                        contentScale = ContentScale.Crop,
                        alpha = 0.5F
                )
        GreetingText(
        message = message,
        from = from,
        modifier = Modifier
            .fillMaxSize()
            .padding(8.dp)
        )
        }
}
```

`Box` layout is one of the standard layout elements in Compose. Use `Box` layout to stack elements on top of one another.



💡 You use the `ContentScale.Crop` parameter scaling, which scales the image uniformly to maintain the aspect ratio so that the width and height of the image are equal to, or larger than, the corresponding dimension of the screen.

Layout Modifiers.

Modifiers are used to decorate or add behavior to Jetpack Compose UI elements. For example, you 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.

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

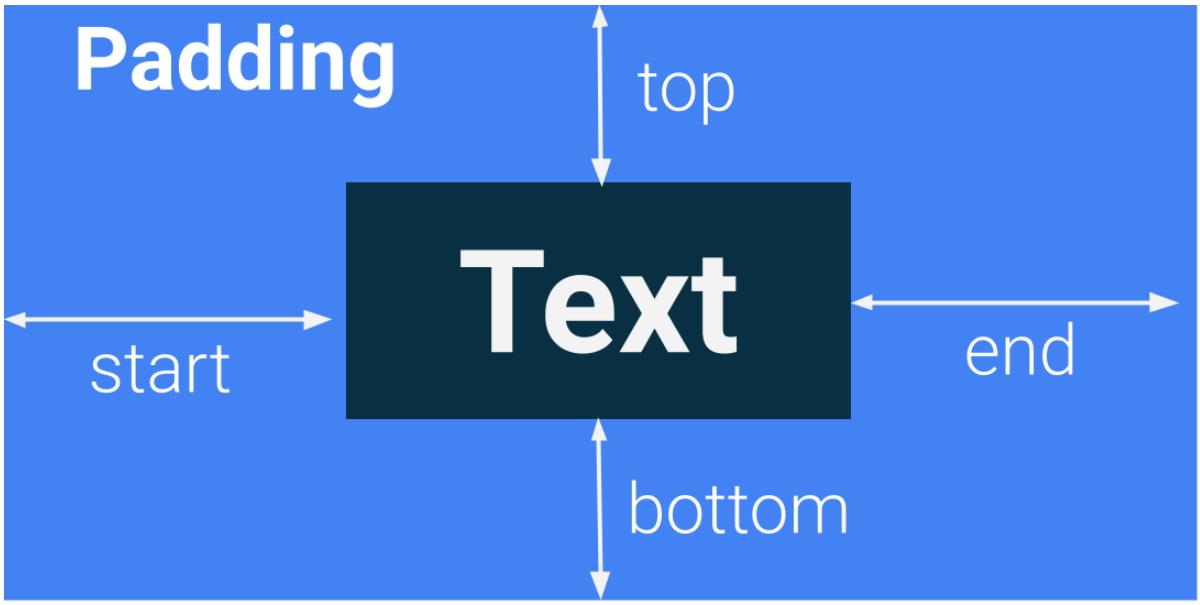
```
@Composable
fun GreetingText(message: String, from: String, modifier: Mod
    Column(
        verticalArrangement = Arrangement.Center,
```

```
            modifier = modifier
        ) {
            Text(
                text = message,
                fontSize = 100.sp,
                lineHeight = 116.sp,
                textAlign = TextAlign.Center
            )
            Text(
                text = from,
                fontSize = 36.sp,
                modifier = Modifier
                    .padding(16.dp)
                    .align(alignment = Alignment.End)
            )
        }
    }
```

The `verticalArrangement` property in the column is set to `Arrangement.Center`. The text content will be centered on the screen thus.

A UI element wraps itself around its content. To prevent it from wrapping too tightly, you 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.




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

Conclusion.

- The **Resource Manager** tab in Android Studio helps you add and organize your images and other resources.
- An `Image` composable is a UI element that displays images in your app.
- An `Image` composable should have a content description to make your app more accessible.
- Text that's shown to the user, such as the birthday greeting, should be extracted into a string resource to make it easier to translate your app into other languages.