

Jetpack compose beginner crash course

https://www.youtube.com/watch?v=6_wK_Ud8--0&ab_channel=PhilippLackner

Jetpack compose: it is a new UI framework where we build the whole UI in Kotlin (we don't use xml files for the view no more).

Generate a **new compose project**: File → new project → empty compose activity.

setContent function in compose is the equivalent to the setContent function of xml, it assigns a certain xml file to an activity.

A **composable** is a Kotlin function, when you create composable you have to use the '@Composable' annotation. For example:

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

DefaultPreview is the equivalent to the layout inspector in xml. By using the following function we can see what we are building. You have to annotate the function with 'Preview'.

```

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

```

In this case 'JetpackComposeCrashCourseTheme' gives the UI some theming attributes.

This form of building UI is different than using XML. In Compose we have a more **declarative** approach. In XML we command the view to do something, in Compose we have the concept of state, instead of changing the text with `greetingText.text = "` we change the name's state.

Using the command **Ctrl+P** inside a Composable shows all the properties of the Composable.

A **modifier** is used to modify our view. It is applicable for any composable, it has properties like background and padding. The properties of our modifier executes in order, it is not the same to set the background and then the padding than to set the padding and then the background.

```

@Composable
fun Greeting(name: String) {
    Text(
        text = "Hello $name!",
        color = Color.Blue,
        fontSize = 30.sp,
        modifier = Modifier
            .background(Color.Red)
            .padding(16.dp)
            .background(Color.Green)
    )
}

```

If we want to show two texts: when we have two composables the one called latter is what will be displayed in front. There are **linear composables** like 'Column' which arranges composables in a column, the column composable also have modifiers.

```

@Composable
fun Greeting(name: String) {
    Column(
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.Center,
        modifier = Modifier
            .size(400.dp)
    ) { this: ColumnScope
        Text(
            text = "Hello $name!",
            color = Color.Blue,
            fontSize = 30.sp,
        )
        Text(
            text = "Some other text",
            color = Color.Blue,
            fontSize = 30.sp,
        )
    }
}

```

We also have the Row layout and the Box layout which is the equivalent to the `FrameLayout` in XML

To add an image to the UI we use the **Image** composable. `painter = painterResource` is the equivalent in XML to set the image resource.

```

Image(
    painter = painterResource(id = R.drawable.ic_launcher_foreground),
    contentDescription = null,
    modifier = Modifier.background(Color.Black)
)

```

We can also use the `Icon` composable, it is like an image but vector only.

```
Icon(
    imageVector = Icons.Default.Add,
    contentDescription = null
)
```

Inside composable functions we can use if statements surrounding our composables to show it or not based on a condition or for loops to show a composable the number of times we want.

In XML we use RecyclerView to build **lazy loading** lists, only the items that the user sees on the screen are the ones rendered, in XML it is a complex code. In Jetpack Compose it is simpler, we only have to use the LazyColumn composable.

```
@Composable
fun Greeting(name: String) {
    LazyColumn(modifier = Modifier.fillMaxSize()) { this: LazyListScope
        items(count: 10) { this: LazyItemScope, i ->
            Icon(
                imageVector = Icons.Default.Add,
                contentDescription = null,
                modifier = Modifier.size(100.dp)
            )
        }
    }
}
```

The same way we have a LazyColumn we also have a LazyRow.

Modifier.fillMaxSize() is used to make the component fit the maxSize of its parent.

State is a value that can change over time. In Compose when a value is changed it looks through the UI to only change the composables that are

affected by that change, that is what we call **recomposition**.

The **remember** block is used to only evaluate the expressions inside during the composition, it won't run the lines inside during each recomposition of the code. This comes in hand when you for example have a variable with a default value, if you didn't declare it inside a remember block each time that the UI made a recomposition it will go back to its default value.

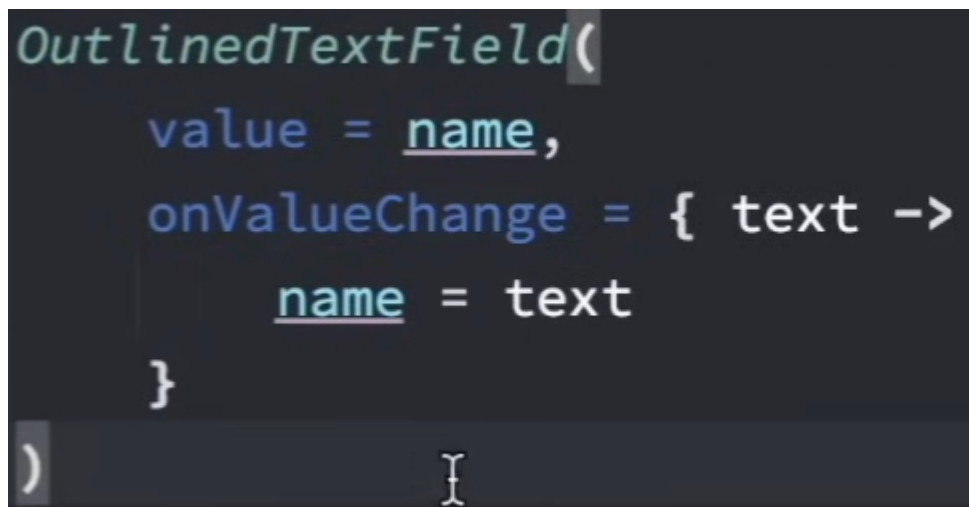
Using the 'by' keyword when defining a variable instead of '=' avoids us to have to write variable.value every time and we can write only variable to access the value.

The following code is an example of the typical counter app:

```
setContent {  
    JetpackComposeCrashCourseTheme {  
        var count by remember {  
            mutableStateOf( value: 0)  
        }  
        Column(  
            modifier = Modifier.fillMaxSize(),  
            verticalArrangement = Arrangement.Center,  
            horizontalAlignment = Alignment.CenterHorizontally  
        ) { this: ColumnScope  
            Text(  
                text = count.toString(),  
                fontSize = 30.sp  
            )  
            Button(onClick = {  
                count++  
            }) { this: RowScope  
                Text(text = "Click me!")  
            }  
        }  
    }  
}
```

In an **OutlinedTextField** we have to declare the value and the `onValueChange`. This works as a two way binding, the same state that causes the state change also triggers the `onValueChange` which changes the value of the text. In the following case name would be a variable like:

```
var name by remember {  
    mutableStateOf("")  
}
```

A screenshot of a code editor showing the configuration of an `OutlinedTextField`. The code is as follows:

```
OutlinedTextField(  
    value = name,  
    onValueChange = { text ->  
        name = text  
    }  
)
```

The variable `name` is underlined in both places, indicating it's a state variable. The cursor is positioned at the end of the closing parenthesis.

The following example is about a screen that lets you input some text and add it to a list that displays below the input field:

```
setContent {  
    JetpackComposeCrashCourseTheme {  
        var name by remember {  
            mutableStateOf("")  
        }  
        var names by remember {  
            mutableStateOf(listOf<String>())  
        }  
  
        Column(  
            modifier = Modifier  
                .fillMaxSize()  
                .padding(16.dp)
```

```

        ) {
            Row(
                modifier = Modifier.fillMaxWidth()
            ) {
                OutlinedTextField(
                    value = name,
                    onChange = { text ->
                        name = text
                    },
                    modifier = Modifier.weight(1f)
                )
                Spacer(modifier = Modifier.width(16.dp))
                Button(onClick = {
                    if(name.isNotBlank()) {
                        names = names + name
                        name = ""
                    }
                }) {
                    Text(text = "Add")
                }
            }
        }
    }

    // Call the lazy loading list composable
    NameList(names = names)
}

// Reusable list of names composable
@Composable
fun NameList(
    names: List<String>,
    modifier: Modifier = Modifier
) {
    LazyColumn(modifier) {
        items(names) { currentName ->
            Text(
                text = currentName,
            )
        }
    }
}

```



```
        modifier = Modifier
            .fillMaxWidth()
            .padding(16.dp)
    )
    Divider()
}
}
```

Now take action! take a design in Dribbble and try to build it.