Laboratorio di Applicazioni Mobili
Bachelor in Computer Science &
Computer Science for Management

University of Bologna

Declarative UI

Federico Montori
federico.montori2@unibo.it

Table of Contents

What is Compose

“Jetpack Compose is Android’s modern toolkit for building native UI. It simplifies and accelerates UI development on Android bringing your apps to life with less code, powerful tools, and intuitive Kotlin APIs.”

Jetpack Compose is a **Kotlin-based declarative UI** framework developed by Google and released in its stable version in mid 2021.

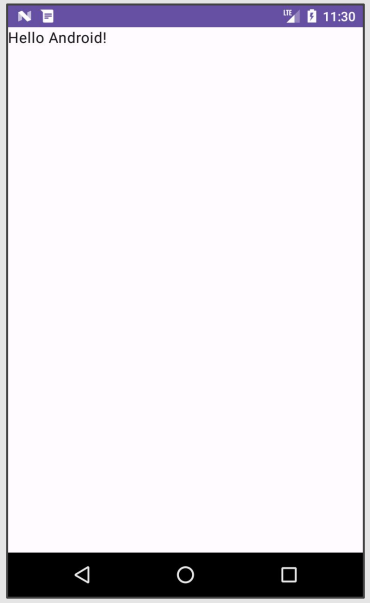
What is Compose

```
Surface( modifier = Modifier.fillMaxSize() ) {
            Greeting("Android")
        }

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

- Compose is a modern approach
- It allows you to write a lot less code
- It is based on the concept of **Composables**
- It is a **declarative UI approach**
  - What we have seen with Views can be defined as an **imperative UI approach**.

What is Compose

```
Surface( modifier = Modifier.fillMaxSize() ) {
            Greeting("Android")
        }

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



Elements in the UI are **Composable functions**, that take in data and emit UI elements.

- In this case the element **Text** renders to a TextView, but we don't need to know it!
- There will be **no XML Layouts**!
- Hierarchy is specified by calling composables within composables.

What is Compose

Android Views

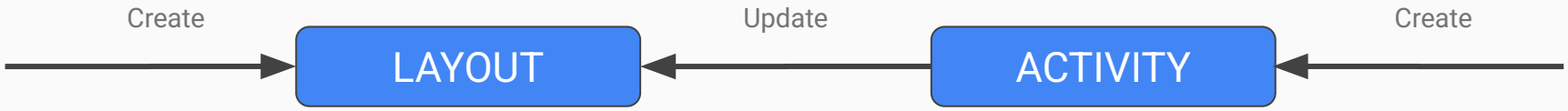
- Imperative Layout
- XML (plus referencing in Java/Kotlin)
- Inflate the views into an Activity (reverse referencing)
  - born for MVC
- Behavior dictated by the business logic
- Modification

Jetpack Compose

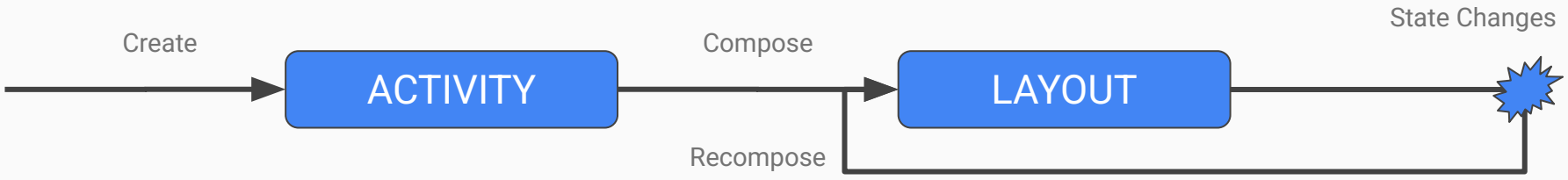
- Declarative Layout
- Kotlin specific
- Declare the views (composables) from the Activity
  - born for MVVM and modern cascade approaches
- Behavior embedded in the UI
- Recomposition

What is Compose

Android Views: runtime modification



Jetpack Compose: recomposition



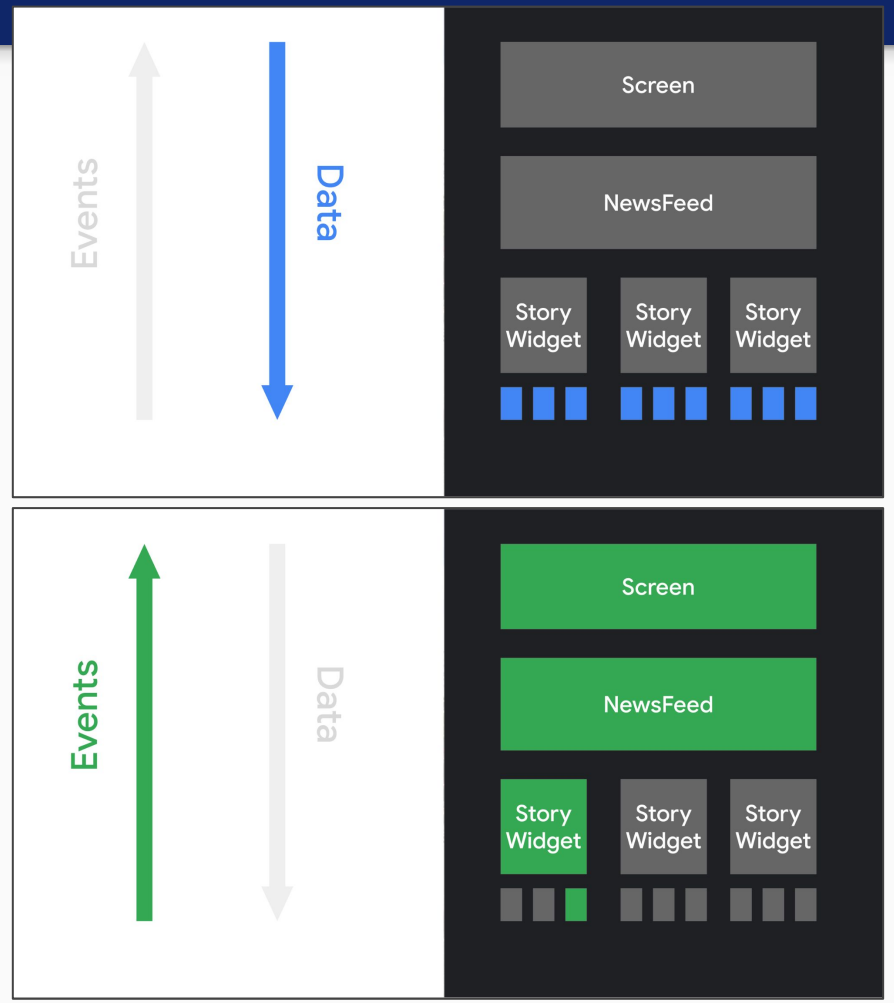
Logical references in only one direction… seems familiar?

More on states later…

What is Compose

Recomposition may seem inefficient, but…

- It smartly recomposes only parts of the layout that were actually affected.
- Composition and recomposition work within coroutines under the hood, which may trigger a pool of background threads…
  - Android Views is completely on the main thread.



Composables and Previews

As Compose is a complete redesign, you will need to import the whole BoM (Bill of Materials) by yourself…

```
implementation("androidx.activity:activity-compose:1.8.2")
implementation(platform("androidx.compose:compose-bom:2023.08.00"))
implementation("androidx.compose.ui:ui")
implementation("androidx.compose.ui:ui-graphics")
implementation("androidx.compose.ui:ui-tooling-preview")
implementation("androidx.compose.material3:material3")
androidTestImplementation(platform("androidx.compose:compose-bom:2023.08.00"))
androidTestImplementation("androidx.compose.ui:ui-test-junit4")
debugImplementation("androidx.compose.ui:ui-tooling")
debugImplementation("androidx.compose.ui:ui-test-manifest")
```

Composables and Previews

Theming in Compose is scoped and based on Material3 libraries

- To access or modify theme properties you use **MaterialTheme.***

```
MyTheme {           // Everything inside this scope uses MyTheme
    // A surface container using the 'background' color from MyTheme
    Surface(
        modifier = Modifier.fillMaxSize(),
        color = MaterialTheme.colorScheme.background
    ) {
        Greeting("Android")
    }
}
```

Composables and Previews

The theme is then declared separately by instantiating a **MaterialTheme**

```
fun MyTheme(
   darkTheme: Boolean = isSystemInDarkTheme(), content: @Composable () -> Unit
) {
      MaterialTheme( // build elsewhere two objects of type LightColorScheme & DarkColorScheme
            colorScheme = when {
                  darkTheme -> DarkColorScheme
                  else -> LightColorScheme
            },
            typography = Typography,
            content = content
      )
}
```

Composables and Previews

You can use the **Preview** keyword to see the preview of any composable

- Pretty much equivalent to the layout inspector in Views
- You can preview even single widgets
- Preview is dynamic, it gets updated and recomposed, unlike static Views

```
@Preview(showBackground = true) // This preview shows only the widget, not the fullscreen
@Composable
fun GreetingPreview() {
    MyTheme {
        Greeting("Android")
    }
}
```

Composables and Previews

Widgets in Compose have their own parameters. Few of them are mandatory (like **text** for **Text**). For optional parameters we often use named arguments

```
@Preview(showBackground = true)
@Composable
fun GreetingPreview() {
    MyTheme {
        Greeting("Android")
    }
}
```

```
@Composable
fun Greeting(name: String) {
    Text(
        text = "Hello $name!",
        color = Color.Blue,
        fontSize = 30.sp
    )
}
```

Composables and Previews

Widgets in Compose have their own parameters. Few of them are mandatory (like **text** for **Text**). For optional parameters we often use named arguments

- An omnipresent one is **Modifier** which holds all general appearance

```
@Preview(showBackground = true)
@Composable
fun GreetingPreview() {
    MyTheme {
        Greeting("Android",
                modifier = Modifier
                        .background(Color.Red)
                        .padding(10.dp))
    }}
```

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

Notice how it gets injected here

Composables and Previews

Widgets in Compose have their own parameters. Few of them are mandatory (like **text** for **Text**). For optional parameters we often use named arguments

- Modifier parameters are applied sequentially!

```
@Preview(showBackground = true)
@Composable
fun GreetingPreview() {
   MyTheme {
      Greeting("Android",
            modifier = Modifier
                  .background(Color.Red)
                  .padding(10.dp))
   }}
```

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

      modifier = modifier
            .background(Color.Blue)
   )}
```

Layouts

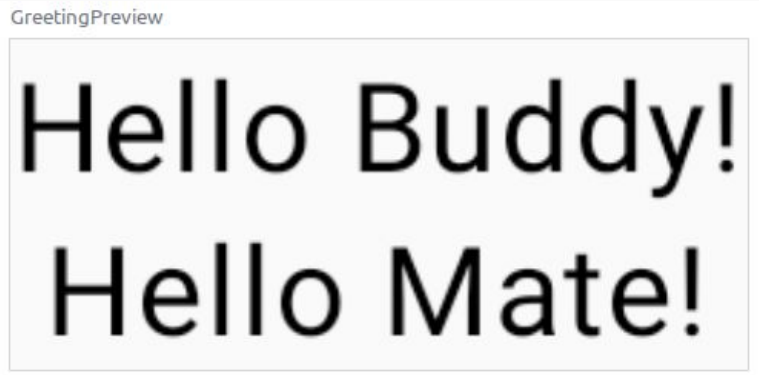
If you were to place two composables in the same scope, Android does not know how to arrange them.

- In Android Views we had **Layouts** which were just Views made for containing Views.
- In Compose we have composables that are made for containing other composables.
  - We do not have real layouts here, but we can kind of connect the two concepts…

Layouts

- A **Row()** is similar to a horizontal LinearLayout
- A **Column()** is similar to a vertical LinearLayout

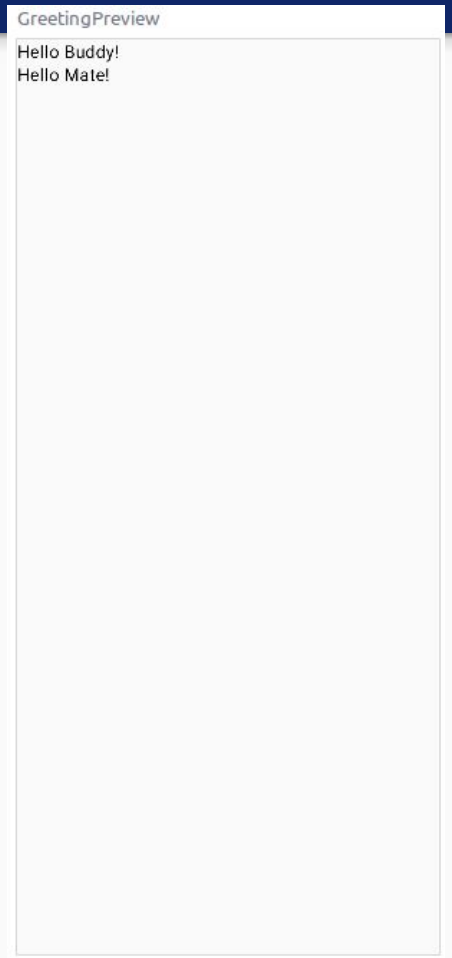
```
@Preview(showBackground = true)
@Composable
fun GreetingPreview() {
    MyTheme {
        Column( horizontalAlignment = Alignment.CenterHorizontally ) {
            Greeting("Buddy")
            Greeting("Mate")
        }
    }
}
/* Columns and Rows can have their Modifier too… */
```



Layouts

Old layout params are now contained in the modifier too. Example with “match parent”:

```
@Preview(showBackground = true)
@Composable
fun GreetingPreview() {
   MyTheme {
      Column( modifier = Modifier.fillMaxSize() ) {
         Greeting("Buddy")
         Greeting("Mate")
      }
   }
}
/* Columns and Rows can have their Modifier too… */
```

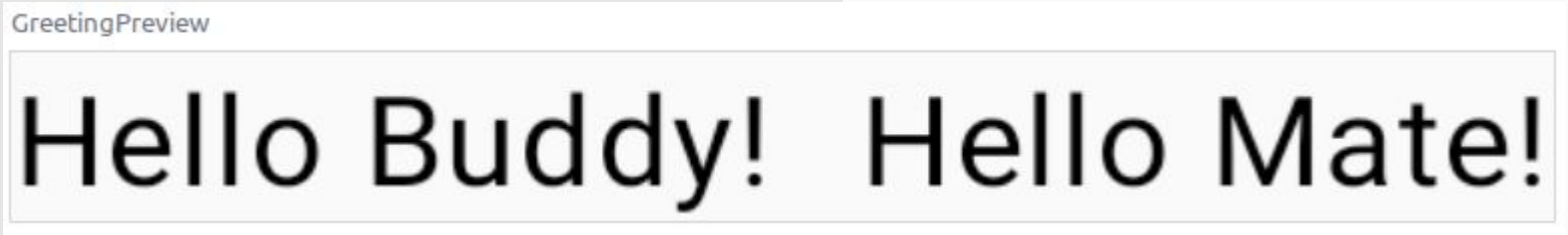


Layouts

Spacing between items is done with a **Spacer** item

- (horizontal for Row, vertical for Column)

```
@Preview(showBackground = true)
@Composable
fun GreetingPreview() {
    MyTheme {
        Row() {
            Greeting("Buddy")
            Spacer(modifier = Modifier.size(10.dp))
            Greeting("Mate")
        }
    }
}
```



Layouts

It is also important to understand the difference between:

- Alignment (align elements in the orthogonal direction of the layout)
- Arrangement (arrange elements in the direction of the layout)

So:

- Column has:
  - verticalArrangement property
  - horizontalAlignment property
- Row has:
  - horizontalArrangement property
  - verticalAlignment property

Layouts

- A **Box()** is similar to a FrameLayout
- A **Surface()** is also a Box with elevation and some additional checks

```
@Preview(showBackground = true)
@Composable
fun GreetingPreview() {
   MyTheme {
      Box( modifier = Modifier.size(150.dp) ) {
         Greeting("Buddy")
         Greeting("Mate",
              modifier = Modifier.align(Alignment.BottomEnd))
      }
   }
}
```

Layouts

Compose is very efficient with nested layouts

- … unlike Views, where a flat layout is always the best choice
- In fact, we have a **ConstrainedLayout** in Compose, but it is hardly used.
  - Most of the times you can get away with nesting Rows and Columns

How to setup your production UI? Just replace the **setContentView**:

```
class MainActivity : ComponentActivity() {
  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContent {
      MyTheme { /* My Compose Body */ }
    }
  }
}
```

Dynamic Layouts

Layouts in Compose are dynamic with respect to the programming logic.
For example, a visibility attribute is here represented by a conditional:

```
@Preview(showBackground = true)
@Composable
fun GreetingPreview() {
    MyTheme {
        Column {
            Greeting("Buddy")
            Greeting("Mate")
        }
    }
}
```

```
@Composable
fun Greeting(name: String) {
if (name.length >= 5)
            Text(
                text = "Hello $name!",
            )
}
```

GreetingPreview

Hello Buddy!

Dynamic Layouts

Similarly a **ListView** would be here represented by a simple loop

```
@Preview(showBackground = true)
@Composable
fun GreetingPreview() {
   MyTheme {
      Column {
         for ( i in 1..5 ) {
            Greeting("Buddy")
         }
      }
   }
}
```

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

GreetingPreview

Hello Buddy!
Hello Buddy!
Hello Buddy!
Hello Buddy!
Hello Buddy!

Dynamic Layouts

What about the **RecyclerView**?
In Android Views the RecyclerView is quite painful to implement, because you need to deal with its optimization, even though it gives out enough level of detail.

- We needed to specify a layout for each member and inflate it into a container.
- We needed to define the behavior of each view element.
- We finally had to deal with all the modifications going on with the list, which delivers a lot of pain when keeping track of the indices.

Dynamic Layouts

What about the **RecyclerView**?
In Compose we just make a lazy loading Column and implement the lambda associated to each element.

```
LazyColumn {
    items(count = 5) { i ->
        Greeting("Buddy number ${i + 1}")
    }
}
```

GreetingPreview

Hello Buddy number 1!
Hello Buddy number 2!
Hello Buddy number 3!
Hello Buddy number 4!
Hello Buddy number 5!

States

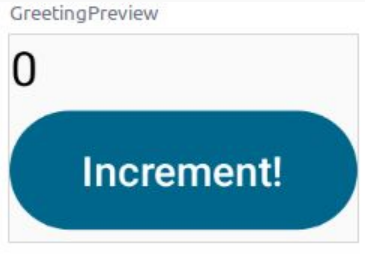
What triggers recomposition? State changes!

“A state is just a value (a variable) that can change over time and to which one or more composables depend on.”

Any responsive UI is founded upon states, because the change of a state triggers the recomposition of every composable using that state.

States

Think about a simple UI where we have a text label which displays the value of a variable and a button that increments that variable.

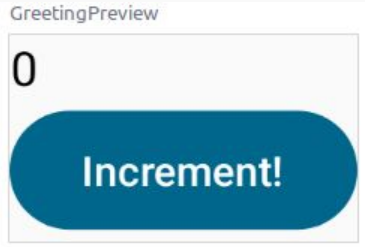



```
var number = 0
Column {
    Text(text = "$number")
        Button(onClick = {
            number++
        }) {
            Text(text = "Increment!")
        }
}
```

However this will not work, because the text field will be updated only through recomposition. There is **no way** in which we can modify it directly.

States

States are similar to LiveData, with all Composable functions using their values listening for their changes. Use **mutableStateOf**




```
var number = mutableStateOf(0)
Column {
        Text(text = "$number")
                Button(onClick = {
                        number.value++
                }) {
                        Text(text = "Increment!")
                }
}
```

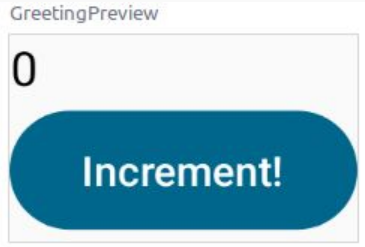
Modifying its value implies calling the *.value* member such as in LiveData.

However this will not work properly, because recomposing this will also trigger the reinstantiation of **number.**

How to make the instantiation unique?

States

States are similar to LiveData, with all Composable functions using their values listening for their changes. Use **mutableStateOf**



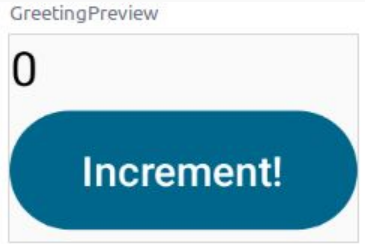

```
var number = remember {mutableStateOf(0)}
Column {
        Text(text = "$number")
                Button(onClick = {
                        number.value++
                }) {
                        Text(text = "Increment!")
                }
}
```

How to make the instantiation unique?

The **remember** block makes sure that the instantiation is called only once and every other time is going to be skipped.

States

States are similar to LiveData, with all Composable functions using their values listening for their changes. Use **mutableStateOf**




```
var number by remember {mutableStateOf(0)}
Column {
        Text(text = "$number")
                Button(onClick = {
                        number++
                }) {
                        Text(text = "Increment!")
                }
}
```

We can make the syntax less verbose by using the delegation syntax.

In this case we do not need to call the value attribute, but we treat the State syntactically as a normal variable.

States

In some cases you may need **two-way bindings** which imply that the same element that causes the state change also needs to be updated.

```
var textField by remember { mutableStateOf("") }
OutlinedTextField(
        value = newTodoTitle,
        onValueChange = {text ->
                textField = text
        })
```

In this case the text field displays the content of a state which gets updated by the text field itself as soon as someone starts typing.

States

States are **Observable Types**, but they are not LiveData:

- A LiveData can be observed by anyone
- A State is only observed by all Composables that use it

```
interface MutableState<T> : State<T> {
    override var value: T
}
```

If your State depends on a LiveData, then you should replicate it or better use the **observeAsState** Compose plugin to LiveData utilities.

Side Effects

In Compose sometimes we want to call functions or perform actions from a Composable that influence something outside a composable.

- These may be called again within every recomposition and we can't control that. For this reason we wrap these actions into **Side Effects**

```
LaunchedEffect(key1 = myState) {
        /* Non-composable action */
}
```

For example, **LaunchedEffect** is a side effect that executes its body whenever **myState** changes (use **true** if you want to execute it only once).

- It runs within a coroutine scope

Side Effects

Another useful one is **DisposableEffect**: similar to LaunchedEffect, but it gives you a scope in which you can declare what happens when the Composable leaves the composition.

```
DisposableEffect(key1 = myState) {
    /* Non-composable action to be called when myState changes */
    onDispose {
        /* Execute this when the composable leaves the composition (e.g. free resources) */
    }
}
```

Multiplatform

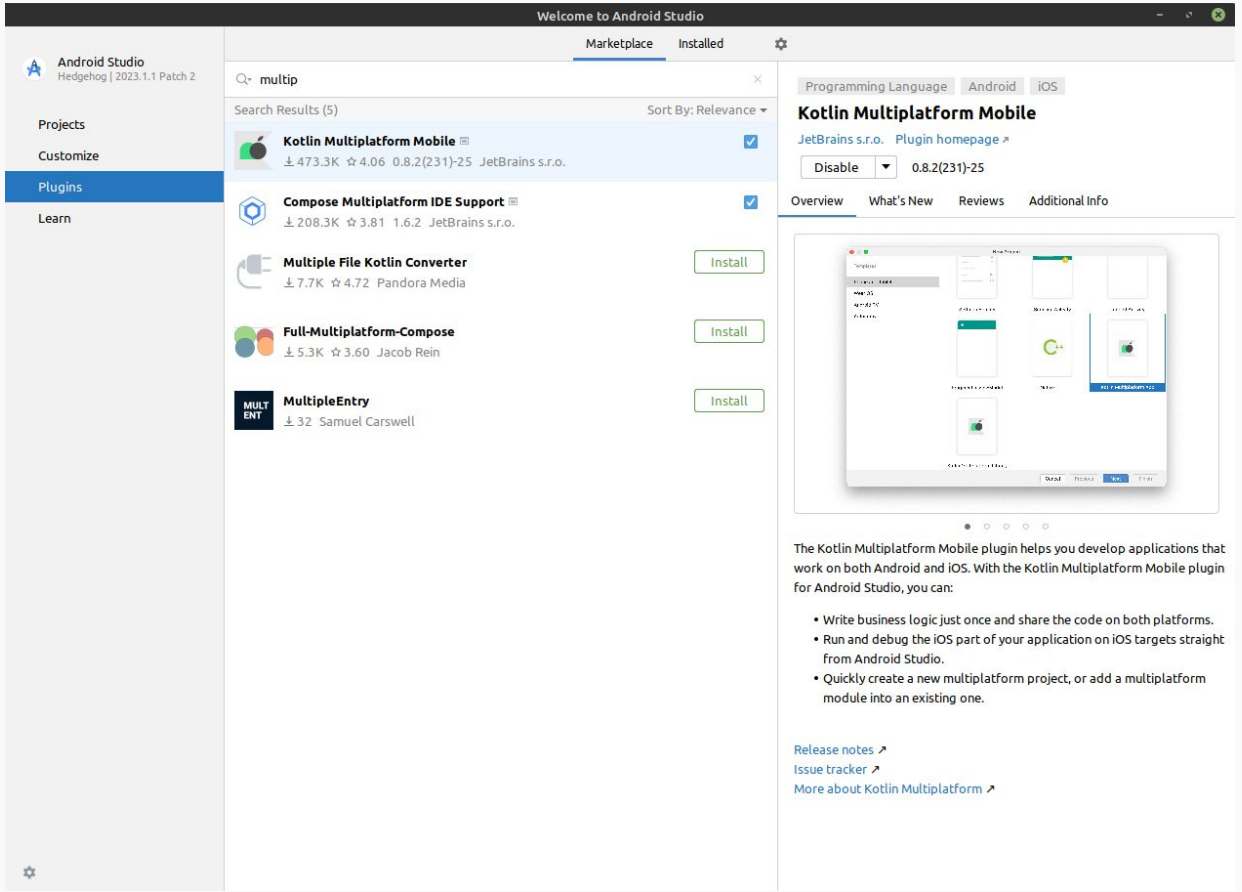
“Kotlin Multiplatform (KMP) is an open-source technology by JetBrains for flexible cross-platform development. It allows you to create applications for various platforms and efficiently reuse code across them while retaining the benefits of native programming. With Kotlin Multiplatform, you can develop apps for Android, iOS, desktop, web, server-side, and other platforms.”

Does this mean that I can write a single codebase for Android an iOS?

- Largely, yes, however the technology is still not far from being in Alpha
- For deploying the iOS app you still need a Mac…

Multiplatform

To start developing a Kotlin Multiplatform App, first install the multiplatform Plugin.



Multiplatform

A KMM can share between Android and iOS:

- The data logic
- The business logic
- The **Compose UI**

However, you will still have to set up an Activity on one side and a MainViewController on the other side to import your Compose UI.

The setup of the dependencies on Gradle is still quite painful!!

Questions?

federico.montori2@unibo.it