Lab 5 - Jetpack Compose[1]

Android is known for experiencing significant changes over the past years in terms of how the apps are structured, the language used for development, the tooling and libraries that speed up development, and the improvements in testing apps. However, one thing that has known little (or no) change in all these years is the Android UI toolkit. This has changed with the introduction of Jetpack Compose - version 1.0, stable and ready to use in production, was released in July 2021. Jetpack Compose is Android's modern toolkit for building native UI. It simplifies and accelerates UI development on Android, enabling the developer to quickly bring the apps to life with less code, powerful tools, and intuitive Kotlin APIs.

The main advantages Jetpack Compose brings are:

- **Less code**. Writing less code affects all stages of development: as a developer, you get to focus on the problem at hand, with less to test and debug and with less chances of bugs; as a reviewer or maintainer you have less code to read, understand, review and maintain.
- **Intuitive**. Compose uses a declarative API, which means that all you need to do is describe your UI - Compose takes care of the rest.
- **Accelerate development**. Compose is compatible with all your existing code: you can call Compose code from Views and Views from Compose. Most common libraries like Navigation, ViewModel and Kotlin coroutines work with Compose, so you can start adopting when and where you want.
- **Powerful**. Compose enables you to create beautiful apps with direct access to the Android platform APIs and built-in support for Material Design, Dark theme, animations, and more

In this lab assignment, you will start by getting familiar with the basics of Jetpack Compose, and then proceed to build an Android app that displays a set of message cards, each showing an image and text, and which are expandable when clicked. You will customize how the messages look and behave upon clicking (adding animations) by using Material Design.

Create a new app with support for Jetpack Compose

To start a new project that includes support for Jetpack Compose by default, Android Studio includes new project templates to help you get started:

1. If you're in the Welcome to Android Studio window, click Start a new Android Studio project. If you already have an Android Studio project open, select File > New > New Project from the menu bar.
2. In the Select a Project Template window, select **Empty Compose Activity** and click Next.
3. In the Configure your project window, do the following:
    a. Set the Name, Package name, and Save location as you normally would.
    b. Note that, in the Language dropdown menu, Kotlin is the only available option because **Jetpack Compose works only with classes written in Kotlin**.
    c. In the Minimum API level dropdown menu, select API level 21 or higher.
4. Click Finish.

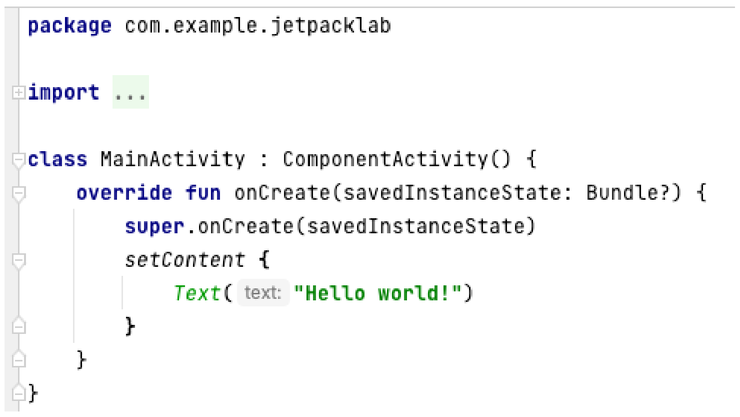
[1] Based on resources from https://developer.android.com/jetpack/compose/tutorial and https://www.jetpackcompose.app/

5. Verify that the project's build.gradle file is configured correctly, as described in Add Jetpack Compose toolkit dependencies.

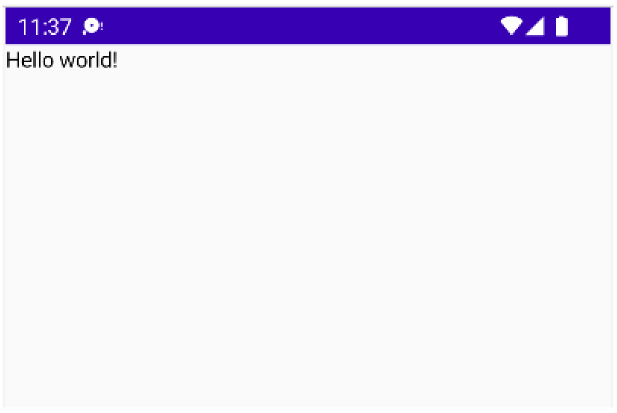
Composable functions

After creating a new app using the Empty Compose Activity template, it already contains some Compose elements, but let's disregard those for now and build it up step by step.

First, we'll display a “Hello world!” text by adding a text element inside the `onCreate` method. You do this by defining a content block, and calling the `Text()` function. The `setContent` block defines the activity's layout where we call composable functions. Composable functions can only be called from other composable functions.



```
package com.example.jetpacklab

import ...

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            Text( text: "Hello world!")
        }
    }
}
```



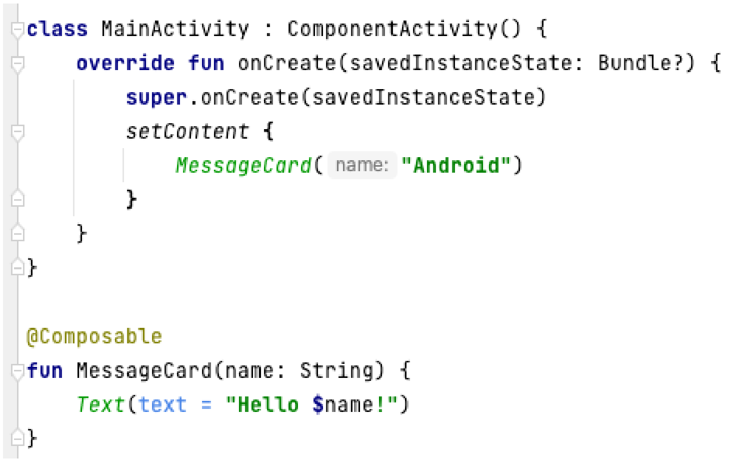
11:37

Hello world!

Jetpack Compose uses a Kotlin compiler plugin to transform these composable functions into the app's UI elements. For example, the `Text()` function that is defined by the Compose UI library displays a text label on the screen. Build and run your app in the emulator to see the result of this code.

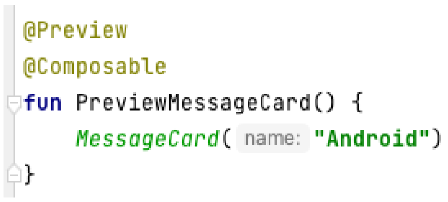
Define a composable function

Jetpack Compose is built around composable functions. These functions let you define your app's UI programmatically by describing how it should look and providing data dependencies, rather than focusing on the process of the UI's construction (initializing an element, attaching it to a parent, etc.). To create a composable function, just add the `@Composable` annotation to the function name. To try this out, define a `MessageCard()` function which is passed a name, and uses it to configure the text element.



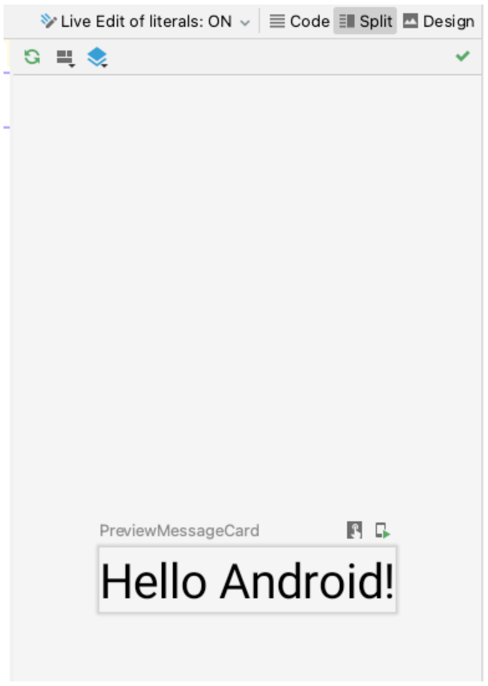
```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            MessageCard( name: "Android")
        }
    }
}

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

Preview your function in Android Studio



```
@Preview
@Composable
fun PreviewMessageCard() {
    MessageCard( name: "Android")
}
```

Android Studio lets you preview your composable functions within the IDE, instead of installing the app to an Android device or emulator. The composable function must provide default values for any parameters. For this reason, you can't preview the `MessageCard()` function directly. Instead, let's make a second function named `PreviewMessageCard()`, which calls `MessageCard()` with an appropriate parameter. Add the `@Preview` annotation before `@Composable`.



Rebuild your project. The app itself doesn't change, since the new `PreviewMessageCard()` function isn't called anywhere, but Android Studio adds a preview window. This window shows a preview of the UI elements created by composable functions marked with the `@Preview` annotation. To update the previews at any time, click the refresh button at the top of the preview window.

Layout design using JetPack Compose

UI elements are hierarchical, with elements contained in other elements. In Compose, you build a UI hierarchy by calling composable functions from other composable functions. So far we've built our first composable function and preview! To discover more Jetpack Compose capabilities, we're going to build a simple messaging screen containing a list of messages that can be expanded with some animations.

Let's start by making our message composable richer by displaying the name of its author and a message content. We will first define a Message data class with two parameters, name and body, both of type String.

```
data class Message(val author: String, val body: String)
```

We then need to change our composable parameter to accept a Message object instead of a String, and add another Text composable inside the MessageCard composable.

```
@Composable
fun MessageCard(msg: Message) {
    Text(text = msg.author)
    Text(text = msg.body)
}
```

Make sure to update code for `PreviewMessageCard()` as well as the call to `MessageCard` within the MainActivity's `setContent` block. Now update the preview: you will see the two text elements inside the content view. However, since we haven't provided any information about how to arrange them, the text elements are drawn on top of each other, making the text unreadable. Let's fix that!

If you were to use classic Android, you would probably think of using something like `LinearLayout` or `RelativeLayout`. This might make you wonder "What is the equivalent of LinearLayout in Jetpack

Compose? Well, JetPack compose provides several layout composables to arrange your elements. The `Column` composable lets you arrange elements vertically. Add `Column` to the `MessageCard()` function by wrapping the two `Text` elements inside a `Column {...}` block. You can also use `Row` to arrange items horizontally and `Box` to stack elements.

Note: If you are interested in a quick way to find the corresponding Compose APIs for an existing classic Android functionality without searching through the documentation, a very helpful tool can be found here: https://www.jetpackcompose.app/What-is-the-equivalent-of-ImageView-in-Jetpack-Compose

Now that we fixed how the two text elements are being displayed, let's enrich our message card by adding a profile picture of the sender. Use the Resource Manager to import an image of your choice (Tools - Resource Manager - “+” icon - Import drawable, the image will then show up in the res - drawable folder). In the `MessageCard` function, add a `Row` composable to have a well structured design and an `Image` composable inside it, alongside the `Column` composable:

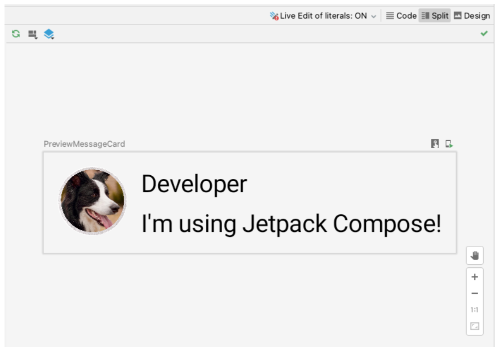
```
Row {
    Image(
        painter = painterResource(R.drawable.pic),
        contentDescription = "Profile picture",
    )
    Column {
        Text(text = msg.author)
        Text(text = msg.body)
    }
}
```

Update the preview - you will notice that while the message layout has the desired structure, its elements aren't well spaced and the image is too big! In order to customize how a composable looks, Jetpack Composes uses modifiers - these allow you to change the composable's size, layout, appearance or add high-level interactions, such as making an element clickable. You can also chain them to create richer composables.

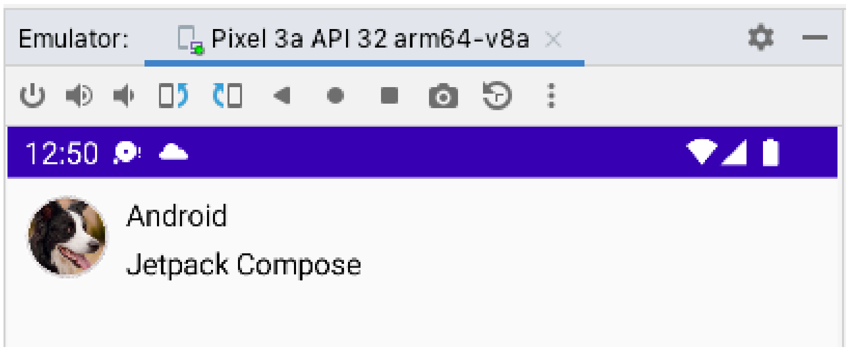
Let’s use some modifiers to improve our layout. We add these modifiers as parameters to the composables we already have in our layout. Try the following:

- Add a padding around our message:
  `Row(modifier = Modifier.padding(all = 8.dp))`
- Set the image size to 40 dp and clip the image to be shaped as a circle:
  `modifier = Modifier.size(40.dp).clip(CircleShape)`
- Add a horizontal space between the image and the column using the Spacer composable:
  `Spacer(modifier = Modifier.width(8.dp))`
- Add a vertical space between the author and message texts in the Column composer:
  `Spacer(modifier = Modifier.height(4.dp))`

You should now see a significant improvement in how the message layout looks after updating the preview. It’s also a good moment to check out the app in the emulator also:



Developer
I'm using Jetpack Compose!



Android
Jetpack Compose

Create a list of messages

Having just one message doesn't look that exciting, so let's change our app to have multiple messages. We need to create a `MessageList` function that will show multiple messages. For this use case, we can use Compose's `LazyColumn` and `LazyRow`. These composables render only the elements that are visible on screen, so they are designed to be very efficient for long lists, being very similar to the RecyclerView you are used to from the previous labs and Mini App 1. At the same time, these composables avoid the complexity of RecyclerView with XML layouts.

```
@Composable
fun MessageList(messages: List<Message>) {
    LazyColumn {
        items(messages) { message ->
            MessageCard(message)
        }
    }
}

@Preview
@Composable
fun PreviewMessageLit() {
    MessageList(SampleData.messageListSample)
}
```

Take a look at the above code. Notice that `LazyColumn` has an `items` child. It takes a `List` as a parameter and its lambda receives a parameter we've named message (we could have named it whatever we want) which is an instance of Message. In short, this lambda is called for each item of the provided List. In order to speed up things, we will use a hard-coded dataset to populate our MessageList. Download the SampleData.kt file from ucilnica and import it in your project. Open it and check what it contains. Also, download the imageResources.zip archive, unpack it, and import all image files to your project (they should show up in your res - drawable folder). Each message will show a different profile pic, so update your Message data class and add an additional parameter `val imgR: String`.

Animate messages while expanding

We want not just to display the list of messages, but also add a bit of interactivity. For that, we will add the ability to expand a message to show a longer one, animating both the content size and the background color. To implement all this, we will use Material Design. This is a design system created by Google to help teams build high-quality digital experiences for Android, iOS, Flutter, and the web. It is inspired by the physical world and its textures, including how they reflect light and cast shadows. Material surfaces reimagine the mediums of paper and ink.

The good news is that Jetpack Compose provides an implementation of Material Design and its UI elements out of the box. The Empty Compose Activity generates a default theme for your project that allows you to use and customize MaterialTheme. This theme is named based on your project name with the "Theme" ending, i.e. JetpackLabTheme. To use it, you first need to import it (i.e. `import com.example.jetpacklab.ui.theme.JetpackLabTheme`). Second, wrap your `MessageCard` function call with the Material theme created in your project, (e.g. `JetpackComposeLabTheme(){...}`). Do it both in the `@Preview` and in the `setContent` function.

Now we need to store the local UI state, i.e. we need to keep track of whether a message has been expanded or not. To keep track of this state change, we have to use the Jetpack Compose functions `remember` and `mutableStateOf`. Composable functions can store local state in memory by using `remember`, and track changes to the value passed to `mutableStateOf`. Composables (and their children) using this state will get redrawn automatically when the value is updated (i.e. any changes to state automatically update the UI). This is called **recomposition**. It happens in a very efficient manner and only the composables that use this value are recomposed; the rest remain as is. There are similar concepts in other declarative UI frameworks like React, and Compose is inspired from these reactive frameworks.

Before the Column composer, declare a variable `isExpanded` that we will use to keep track if the message is expanded or not:

```
var isExpanded by remember { mutableStateOf(false) }
```

Now we can change the background of the message content based on isExpanded when we click on a message. We will use the Surface composable but instead of just toggling the background color of it, we will animate the background color by gradually modifying its value from `MaterialTheme.colors.surface` to `MaterialTheme.colors.primary` and vice versa. To do so, we will first define the `animateColorAsState` function that will set the surfaceColor variable (you can do this right below the line where you define `isExpanded`):

```
val surfaceColor: Color by animateColorAsState(
   if (isExpanded) MaterialTheme.colors.primary else
MaterialTheme.colors.surface,
)
```

We then configure a Surface composable that wraps the Column composable by specifying the shape, the color (which will be changed gradually as defined above), and a modifier that uses animateContentSize to animate the message container size smoothly:

```
Surface(
   shape = MaterialTheme.shapes.medium,
   elevation = 1.dp,
   // surfaceColor color will be changing gradually from primary to surface
   color = surfaceColor,
   // animateContentSize will change the Surface size gradually
   modifier = Modifier
        .animateContentSize()
        .padding(1.dp)
){ Column(...)}
```

Next, we configure the Text composable that shows the message body. We will add a parameter to it so that it should display all the text content if the message is expanded, otherwise just the first line:

```
maxLines = if (isExpanded) Int.MAX_VALUE else 1
```

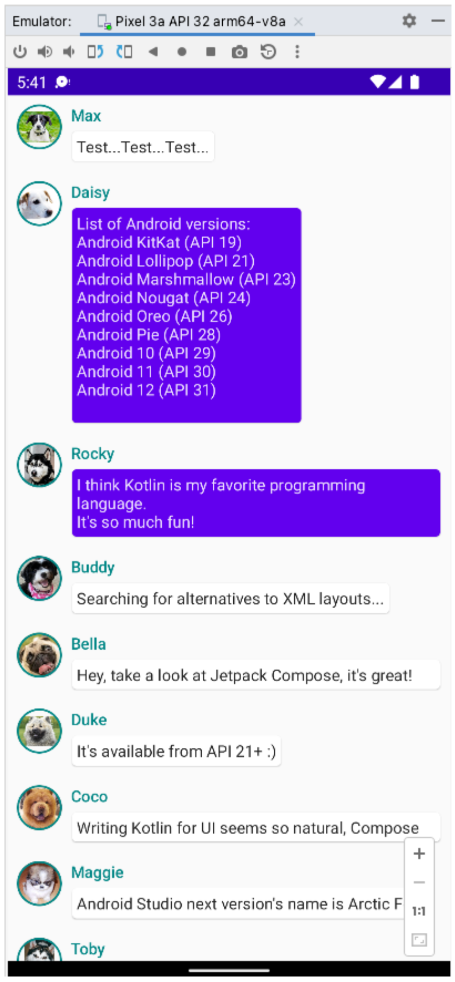
We can format it by adding additional parameters: a modifier that specifies padding along all edges and also a parameter to specify style formatting options:

```
modifier = Modifier.padding(all = 4.dp),
style = MaterialTheme.typography.body2
```

Finally, we will add the clickable modifier to handle click events on the Column composable and toggle the `isExpanded` variable when we click on the Column.

```
modifier = Modifier.clickable { isExpanded = !isExpanded }
```

Now test your app in the emulator. Click each message. You should see how it expands through the animation you defined in your code. However, all messages still display the same profile picture. Let's change this! In the Image composable, change the `painterResource` parameter to the resource indicated by the message's `imgR` property: `painterResource(context.resources. getIdentifier( msg.imgR,"drawable", context.packageName))`. You will need to get the context beforehand. In a composable function, you can get it in the following way: `val context = LocalContext.current.` Now refresh the preview. It should show a different image for each message. Test your app in the emulator!



Max
Test...Test...Test...

Daisy
List of Android versions:
Android KitKat (API 19)
Android Lollipop (API 21)
Android Marshmallow (API 23)
Android Nougat (API 24)
Android Oreo (API 26)
Android Pie (API 28)
Android 10 (API 29)
Android 11 (API 30)
Android 12 (API 31)

Rocky
I think Kotlin is my favorite programming language.
It's so much fun!

Buddy
Searching for alternatives to XML layouts...

Bella
Hey, take a look at Jetpack Compose, it's great!

Duke
It's available from API 21+ :)

Coco
Writing Kotlin for UI seems so natural, Compose

Maggie
Android Studio next version's name is Arctic F

Toby

Congratulations, you've finished the Jetpack Compose lab! You've built a simple app showing a list of expandable & animated messages containing an image and texts, designed using Material Design principles, and you've done it all in under 100 lines of code! Think about how you would implement this without JetPack Compose. How many lines of code (Kotlin + xml) would it take? :-)

If you did not attend and work on this solution during the in-person lab slots, your code must be committed to a private repository named **PBD2023-LAB-5** in your Bitbucket account. User **pbdfrita** must be added as a read-only member of this repository. The code must be committed by Monday (April 10th) 23:59.

Happy coding!