Lab 1: Hello Android

In case you found something to improve, please tell us!
https://forms.gle/3U6WrZNyNx2nBXQ38

1 Introduction

This lab teaches you how to build your first Android app with Jetpack Compose. You will learn how to create an Android project and display composables on Android tablets/phones and smarwatches. It will be the first step towards building a complete sport tracking application, which you will develop step by step during the upcoming lab sessions.

2 Creating an Android project

An Android project contains all the files that comprise the source code for your Android app. The Android SDK tools are included in Android Studio. They make it easy to start a new Android project with a set of default project directories and files.

1. Start Android Studio from the class computer or from your own. In case you want to use your own computer, you can download and install Android Studio here[1]. Computers in class have the version **2024.3.1 Patch 2** pre-installed. Please always use this version to avoid compatibility issues.
2. Click on **New Project**.
3. Now you can select a device and an activity template, from which you begin building your app (Figure 1):
    1. Select **Phone and Tablet** from the options on the left, and **Empty Activity** from the list of items on the right. Click on **Next**.
4. Fill in the form that appears (Figure 2):
    1. **Name** is the app name, as shown to users. You can use **Sports Tracker** as your app name.
    2. **Package name** is the package name for your app. Your package name must be unique across all packages installed on an Android system.
    3. **Save location** is the directory where all the files will be stored for this Android Studio project. It is common practice to group all related apps e.g. all the apps created for this course, in a unique *Workspace* directory.

[1] https://developer.android.com/studio/archive

4. **Language** is the language used for App programming. Make sure you select **Kotlin**.
5. **Minimum SDK** is the lowest version of Android that your app supports. We recommend using the **API 23: Android 6.0 (Marshmallow)** or superior.

5. Leave all the details for the activity in their default state and click **Finish**.

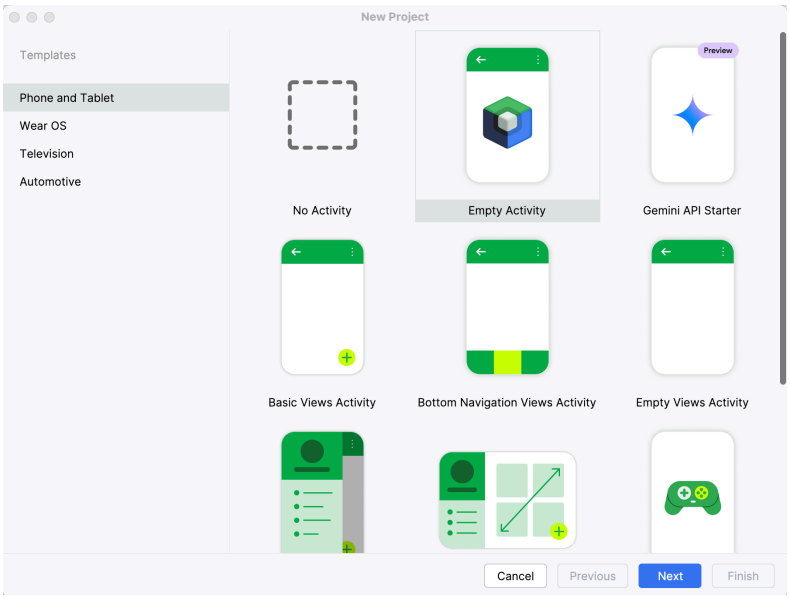

Figure 1: Wizard for creating a new project with Android Studio - choosing a template.

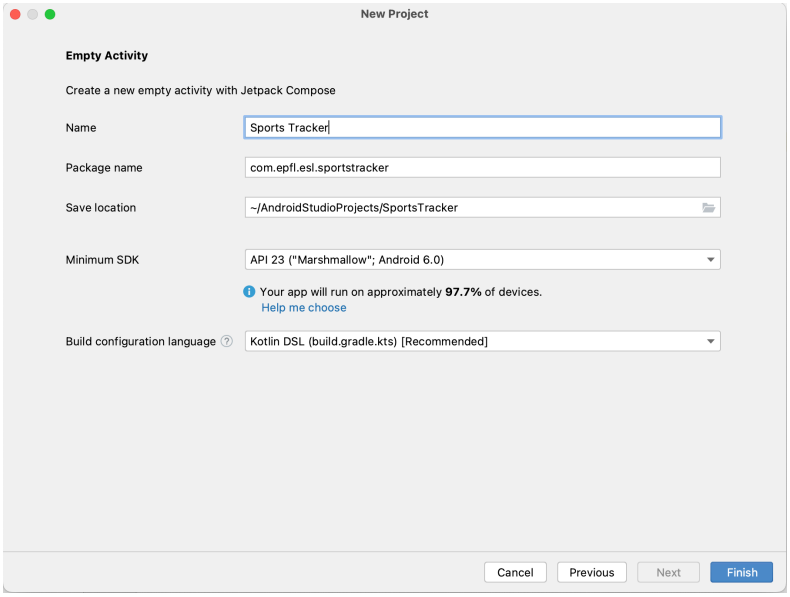

Figure 2: Wizard for creating a new project with Android Studio - configuring your project.

3 Fundamental notions

Before you run your app, you should be aware of a few directories and files in the Android project:

- **app/manifests/AndroidManifest.xml** The manifest file describes the fundamental characteristics of the app and defines each of its components. It also contains the access that your app requires to be used on a device such as body sensors (heart-rate), microphone, GPS, etc. You will learn about the various elements of this file in the next lab sessions.
- **app/kotlin+java/** This directory is for your app's main source files. By default, it includes an Activity class that runs when your app is launched, as well as a directory (**ui.theme**) containing the definition of the UI look and feel, such as the employed colors.
- **app/res/** Contains several sub-directories for app resources. For example:
  - **drawable/** Directory for *drawable* objects used in the UI of your app such as bitmap images. They should be stored in different folders depending on the resolution they are meant for (*hdpi*, *mdpi*, *xhdpi*...).

    Note: You might not see the different folders yet. In the future, you can create them when needed by right-clicking on the **res/** directory and then **New > Android resource directory**. In the menu, select *drawable* as the **Resource type** and *Density* as the **Available qualifier**. You can choose the desired density and click **Ok**.
  - **values/** This directory is for other various XML files that contain a collection of resources, such as string or color definitions.
- **Gradle scripts/** This directory contains the scripts used by the build system to compile the app. This is the place where the project's dependencies are defined and where the Android API level is specified. We will not go further than that use in our labs.

When you build and run the default Android app, the Activity class starts and loads a layout file that says "Hello Android!". The result is nothing exciting, but it's important that you understand how to run your app before you start developing.

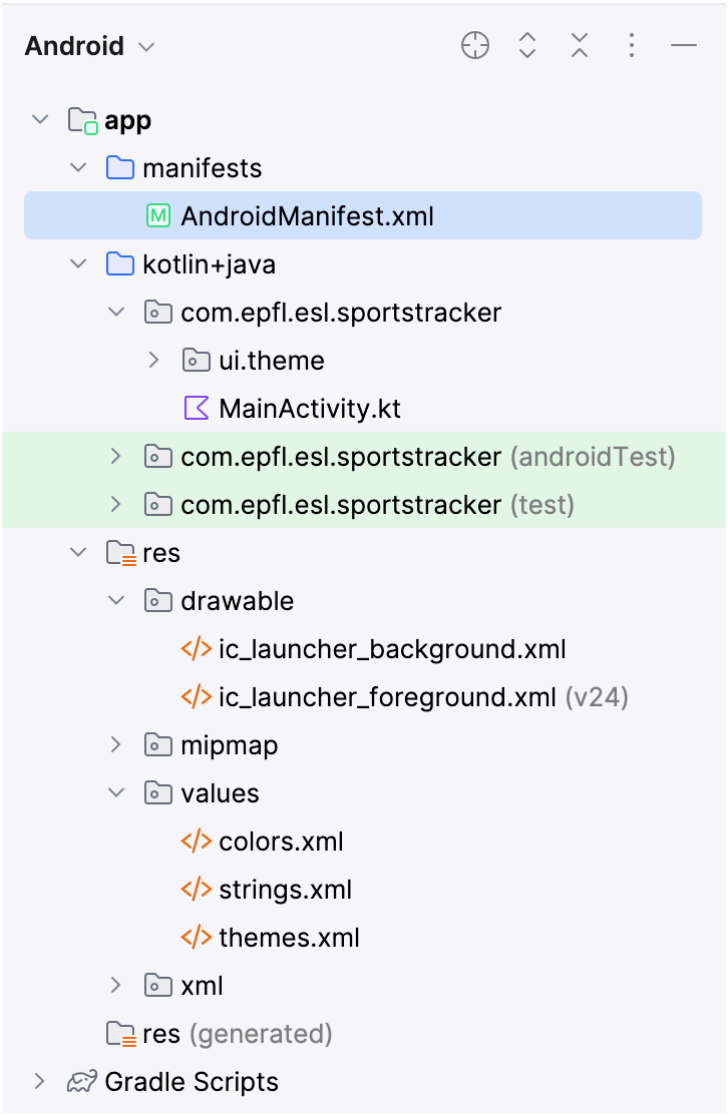

Figure 3: Structure of an Android project

4 Running your app

You can run your app on a real device or an emulated one. This section shows you how to install and run your app on a real device and on the Android emulator. Note that using an emulator may not provide all the features of a real device (camera, sensors, etc.). For this reason, it is always better to test your app (also) on a real device.

4.1 Run on a real device

If you have a real Android-powered device, here is how you can install and run your app:

1. Plug in your device to your development machine with a USB cable.
2. Enable **USB debugging** on your device:
    1. On Android 4.0 and newer, it's in **Settings > Developer options**.
    2. Developer options are hidden by default. To make them available, go to **Settings > About phone/tablet** and tap **Build number** at least seven times. Return to the previous screen to find **Developer options**. On a **smartwatch**, the build number is available going to **Settings > System > About**.
    3. Access **Developer options** and enable **USB debugging**.
3. From the Android Studio toolbar, select your device in the contextual menu that appear beside the **App** configuration button.

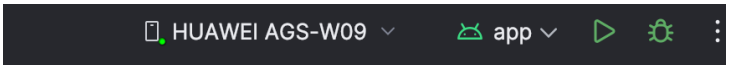

Figure 4: Selecting target device

1. Now click the **Run** button (green arrow in Figure 4). Android Studio installs the app on your connected device you choose and starts it.

When creating a project targeting a smartphone and a smartwatch, both show up in the *Run configuration* dropdown menu as ***Application*** and ***Wearable***, respectively.

That's how you build and run your Android app on a device!

4.2 Run on the emulator

An Android Virtual Device (AVD) is a device configuration for the Android emulator that allows you to model different devices. To run your app on the emulator you need to first create an Android Virtual Device (AVD).

To create an AVD:

1. Launch the Android Device Manager by clicking on the **Device Manager** from the Android Studio toolbar.
2. Device manager lists the details of all available devices (Virtual and Physical).
3. A device might already be present by default in the list. To add more virtual devices, in the Device Manager panel, click on the "+" button, then select **Create Virtual Device**. You can either fill in the details for the AVD or select an existing hardware model from the list. This way, it is easier to try the app on a device with a different form factor.

Running the app on emulated devices is the same as using a real device. Available emulated devices are shown in the "Run" drop-down menu.

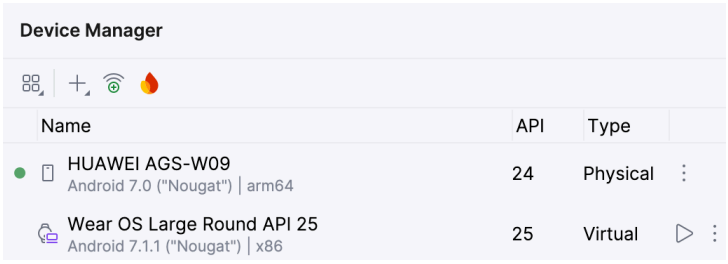

Figure 5: The Android Device Manager

5 Create a `Hello world` app

Android Studio is a development environment that provides many useful tools. Here we show a few of them, but feel free to explore around!

First, from the **app/kotlin+java/(my.package.name)** folder, open the **MainActivity** file. This is the file describing our app's user interface (UI) and functionality. You can see that there is a class called MainActivity with one method, a Greeting composable function outside of the class, and a preview function for the Greeting composable, which allows you to see a live preview of the composable without running the application.

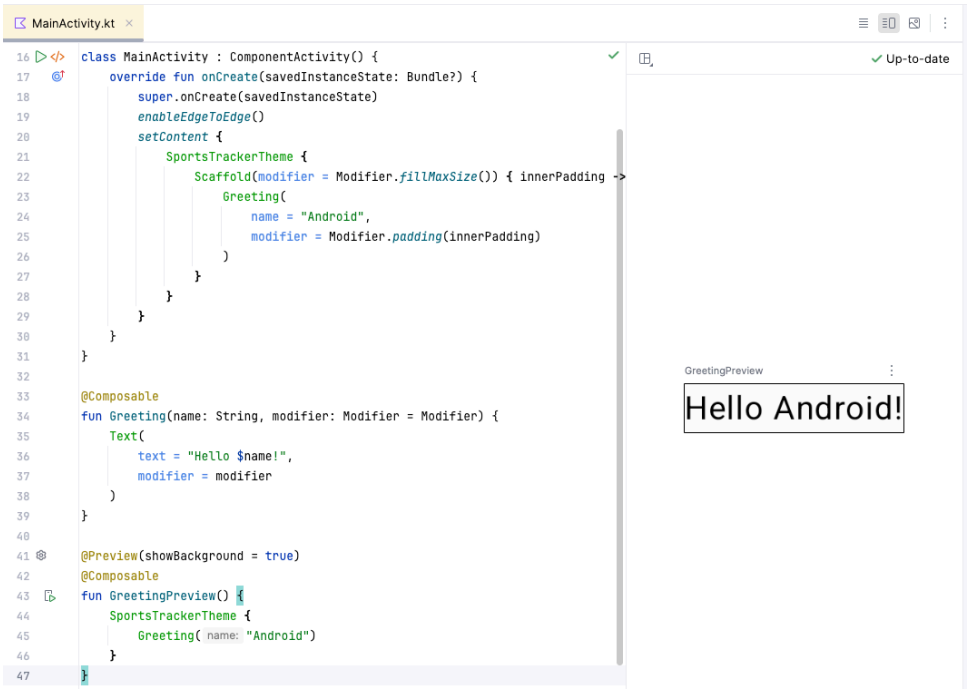

Figure 6: Split view with code and preview

Now select the Split tab button from from the top-right of the window, to be able to see the code and the live previews at the same time. Since this is the first time, you will first have

to build the application. Click on **Build refresh** and wait until it finishes. Let's check the live preview in action by changing the text in the Greeting composable to the following:

```
@Composable
fun Greeting(name: String, modifier: Modifier = Modifier) {
    Text(
        text = "This is my first app in $name!",
        modifier = modifier
    )
}
```

5.1 Tidying up the code style

Editing after editing, especially with multiple programmers, the code can become messy (irregular bracket position, spaces, unclean indentation). This can be fixed using **Code > Reformat code**. You will soon know the shortcut **Ctrl+Alt+L** by heart!

5.2 Renaming variables (the smart way)

The last problem one can find in our code is that the *name* we gave to the **`String resource`** variable is **objectively terrible**. It is so generic that it becomes unhelpful as we add more string resources in the future. One can manually rename all usages with a better name, or use a "find and replace" but can be error-prone. Let's use Android Studio **refactoring** abilities to rename it. To do so, right-click on it and select **Refactor > Rename**, which can be done with the shortcut **Shift+F6**. Let's rename it to "greeting_message".

5.3 The poor man's debug: outputting messages into Logcat

Even if outputting text is not the best way to debug, it's easy to do and is efficient to have an idea of the execution flow in the app. Using the **Log** library, it's easy to output text, which can be colorized depending on its importance. Add this code in the **MainActivity** class:

```
private  val TAG = "MainActivity"
// A function printing to logcat
private fun demoLogcat() {
    Log.v(TAG, "Verbose")
    Log.d(TAG, "Debug")
```

```
    Log.i(TAG, "Information")
    Log.w(TAG, "Warning")
    Log.e(TAG, "Error")
}
```

Note that you will have to import the *android.util.Log* library at the top of the *MainActivity.kt* file.

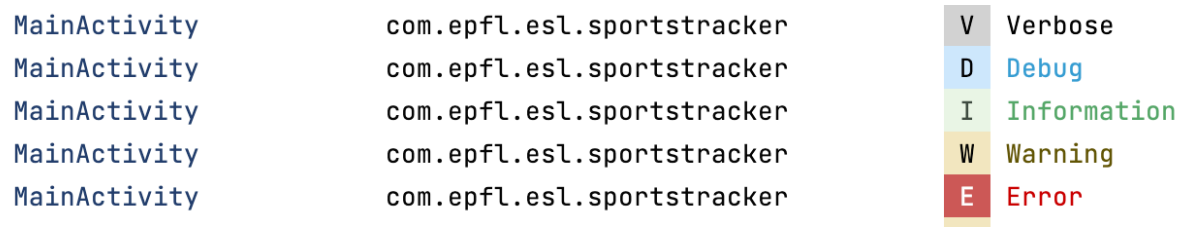

Figure 7: Logcat messages

Create a new **demoLogcat()** method call (for example in **onCreate(...)**, before the **setContent** block). Run the application again and open Logcat by cliccking on the cat icon in the bottom-left toolbar of Android Studio, and filter messages for the keyword "MainActivity". You will see something similar to figure 7.

Depending on the importance of the output, different *verbosity* level is used. An **error** happens only once in a while, as it should be used for a critical failure. A **warning** is for a failure, which does not prevent the execution to continue, and therefore it could happen more frequently. An **information** message can be displayed for example when going from one state to another, which may happen anytime. A **debug**-level message gives usually more details about what is happening. Finally, **verbose** is expected to output a lot of text, very likely that most of it is useless but still interesting to have a detailed vision of the execution. Android Studio can hide messages below a given importance in the Logcat viewer e.g. by filtering for "level:error".

5.4 Proper debugging

To find out what is precisely happening without having to log everything, it's best to use breakpoints. Click near a line number to set a break-point:

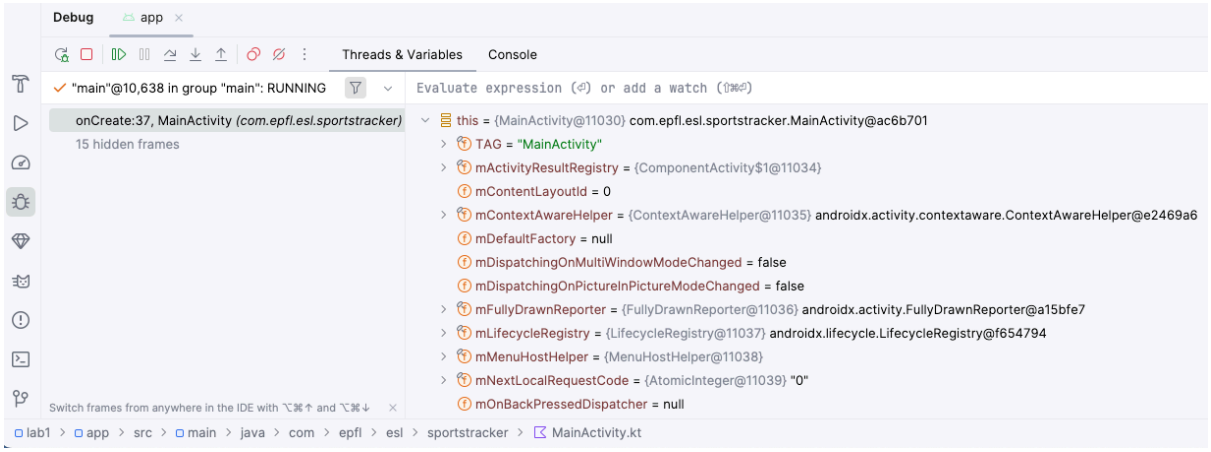

Figure 8: Debug window

Running the app in **debug** mode is done with the button. When the execution hits the break-point, it will **pause the execution** and you will be able to continue step-by-step, explore each variable internal value, execute an arbitrary expression, etc.

Now, you have the additional knowledge that makes Android programming a pleasant (or at least, efficient) experience. Feel free to explore the other features available, such as the Profiler (enabled by clicking on View -> Tool Windows -> Profiler in the top Android studio bar), which can be very helpful to study performance problems.

6 Create a static User Interface

We will build (UI) for an Android apps using hierarchies of composable layouts and composables, as shown in Fig. 9. Non-layout composables include **`Text`**s, **`Image`**s, and **`Button`**s. **`Layouts`** objects are invisible containers that define how the child composables are laid out, such as in a grid or a vertical list.

6.1 Creating composables and previews

There are several kinds of composable layouts, each with a different purpose. The Figure below presents some of the available layouts:

1. **`Column`**, **`Row`** and **`Box`** organize their childrens into a single vertical column, horizontal row, or on top of each other in a box. (Fig. 9).
2. **`ConstraintLayout`** allows you to specify the location of child objects relative to each other, or to their parent (Fig. 10).

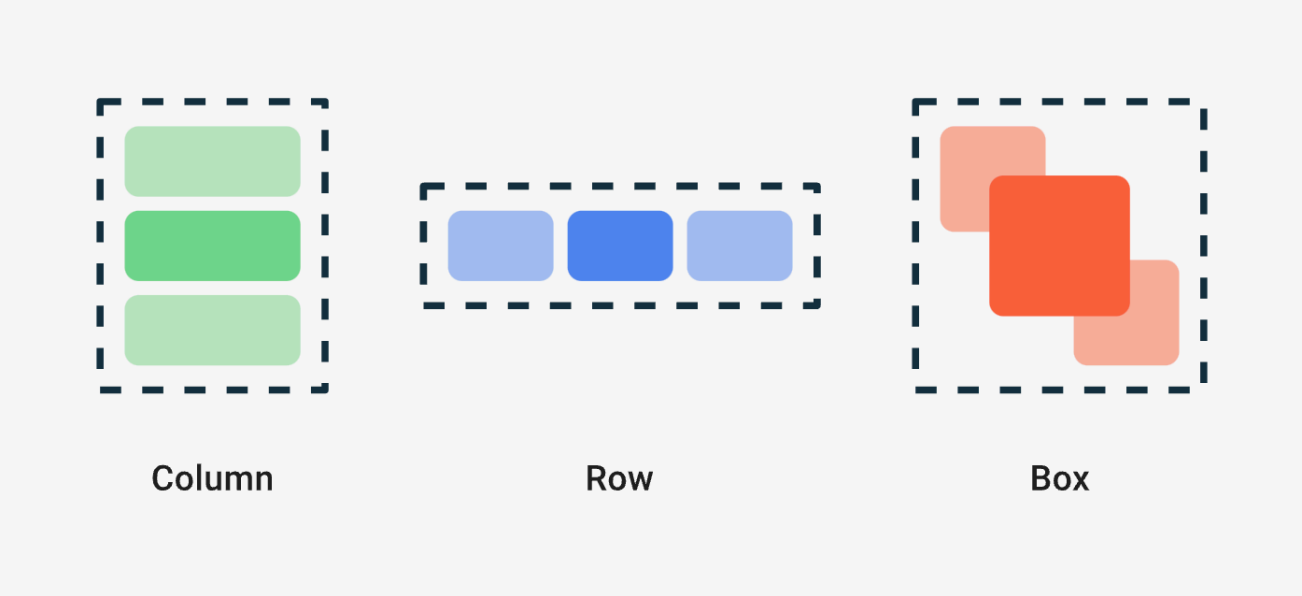

Figure 9: Standard composable layouts

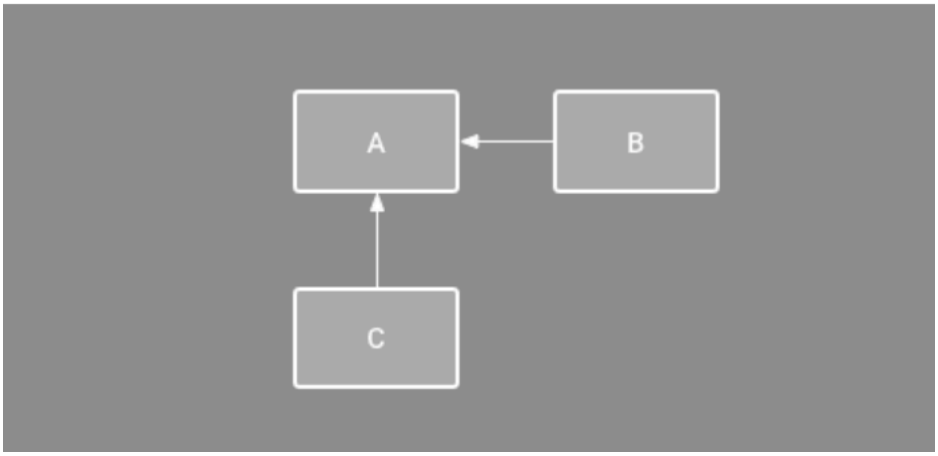

Figure 10: Constraint layout: B is constrained to always be to the right of A, and C is constrained below A.

You will be using a **Column** composable for this part of the lab. Open the layout file in **app/java+kotlin/com.epfl.esl.sportstracker/MainActivity.kt**. Switch to the **Code** or **Split** tab at the top right of the editor's window, and replace all the content with the following:

```
class MainActivity : ComponentActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
```

```
        setContent {
            SportsTrackerTheme {

            }
        }
    }
}
```

Now let's create a new composable that we will call "HomeScreen" outside of the MainActivity class. Type "comp" and press TAB, this will automatically generate a `@Composable` function, where you will have to add the name. Add a function call in the blank space that we left inside the **SportsTrackerTheme** with our new Composable, **`HomeScreen()`**. Let's add a function parameter to our composable, **`modifier: Modifier = Modifier`**. Also, let's add a **Surface** composable inside of our new composable with parameters **`modifier = modifier.fillMaxSize()`** and **`color = MaterialTheme.colorScheme.background`**. Surface is used mainly to assist with material theming (Google's design approach) and with better light/night mode integration. **`modifier = modifier.fillMaxSize()`** fill the whole available screen, both height, and width, to be used by our composable. Finally, let's add a Preview, so we can see live updates without running our app. Type "prev" and press TAB, this will generate a Preview, while you have to add a name. Name it "HomeScreenPreview". Add a **SportsTrackerTheme** block and inside the block call the HomeScreen function.

Below is the resulting code.

```
class MainActivity : ComponentActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            SportsTrackerTheme {
                HomeScreen()
            }
        }
    }
}

@Composable
fun HomeScreen(modifier: Modifier = Modifier) {
    Surface(
        modifier = modifier.fillMaxSize(),
```

```
        color = MaterialTheme.colorScheme.background
    ) {

    }
}

@Preview
@Composable
fun HomeScreenPreview() {
    SportsTrackerTheme {
        HomeScreen()
    }
}
```

Now we can add GUI elements to your **HomeScreen** composable!

6.2 Add elements to a Composable

Let's start with adding a **Column** inside the Surface block. If you see that the "Column" keyword becomes red, press **ALT+ENTER** and import the approriate function (*androidx.compose.foundation.layout.Column*).

In order to have the element children of Column centered, we need to add **verticalArrangment** and **horizontalAlignment** as parameters (see code below).

Now let's add an **Image**. First, you will have to download it from Moodle (**ic_logo.png**). Then, open the Resource Manager tab in the project navigation pane on the left side of the screen by clicking on the button, and click on the '+' sign to open a drop-down menu. Select "Import Drawables". Type "Image" inside the **Column** composable and you will see a popup with 3 Image constructors. In our case, we are going to use the one with the painter function parameter.

```
    Column(
            verticalArrangement = Arrangement.Center,
            horizontalAlignment = Alignment.CenterHorizontally
    ) {
        Image(
            painter = painterResource(id = R.drawable.ic_logo),
            contentDescription = "Logo Sports Tracker"
        )
    }
```

Let's add a Text after the image, with the text "This is my first app!".

```
Text(text = "This is my first app!")
```

The result is shown in Fig. 11.

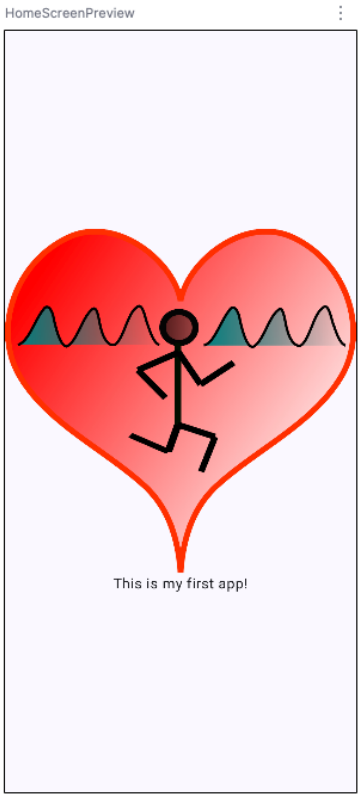

Figure 11: View of the initial Layout

6.3 Customizing the `Views` sizes

As you can see in Fig. 11, the text size is too small comparing to the Image's size. You can easily change the text size by adding the following line as an extra parameter in the Text composable.

```
style = TextStyle(fontSize = 36.sp)
```

Here, *sp* stands for scale-independent pixels. It is recommended to use this unit for every specified text size in the layout; because it allows the application to adjust the text size based on the user's font size preference.

The second problem with the appearance of our app is that the Text is too close to the Image. To increase the distance between these two views, you can add *padding* to the Text or add a **Spacer** composable between them. This line of code `Spacer(modifier = modifier.size(16.dp))` between the Image and the Text, adds an empty space of 16dp between them.

You can use padding instead of a Spacer. Insert this code as a parameter to the Text composable: `modifier = modifier.padding(16.dp)`. Padding pushes the boundaries of the Text in all directions with the specified amount. You can also use padding in one direction only, for example, top padding, which is needed in our case: `modifier = modifier.padding(top = 16.dp)`. You can see the differences between padding and Spacer by adding background color to the Text, applying `.background(color = Color.Green)` to its modifier. Using padding, the Green background is also extended, while using the spacer it is not.

dp stands for density-independent pixels. The Android system converts this unit to an actual density based on the screen device. Except for the text sizes, it is recommended to use *dp* for specifying any size to the layout elements.

Relying on the `fillMaxSize()` is not really flexible to create the layout we aim for. You can set a fixed size for each element in the layout. Particularly, in this layout, you can limit the Image's size by setting `256dp` as the size of the height in its modifier: `modifier = modifier.height(256.dp)`. Now the layout is much better!

7 Create a new module for a smartwatch

Let's create a new module targeting the smartwatch. To create a new module, from **File -> New**, select *New Module...*. Then, on the new window, select **Wear OS**. Then, name the **Application/Library name** as same as the name you chose for the application. For **Module name** enter *wear*, edit the **Package name** to set the package name exactly as same as the application (ex. `com.epfl.esl.sportstracker`). Set the minimum API level as 28. Click "Next", then select "Empty Wear App", enable the Launcher Activity Icon in the next screen and click "Finish". In the new *wear*'s module `build.gradle` file change the "minSdk" to 26 to be able to connect to the Huawey smartwatch.

Now, in this project, we have two different modules for the different devices. To make the names more meaningful, please change the *app* module's name into *mobile*. For this purpose, you need to right-click on the *app -> Refactor -> Rename*. Then, in the **Select Refactoring** window, choose **Rename module**. Press *OK*, enter *mobile* as the new name, and now you have two modules: *mobile* and *wear*.

7.1 Building the Android Wear layout

For the Android Wear App, you will use a new type of layouts: the **constraint layout**. The constraint layout allows you to position your widgets relative to one another and to their parents in the horizontal and vertical axis. The general concept is to constrain a given side of a widget to another side of any other widget or the parent.

You will notice that Android Studio has automatically created a layout for you based on the template from the App Creation Wizard.

In the *wear*'s module **build.gradle** file, add the following line to dependencies to guarantee the compatibility with the constraint layout.

```
    implementation("androidx.constraintlayout:constraintlayout-compose:1.0.1")
```

Delete the content of **wear/kotlin+java/com.epfl.esl.sportstracker.presentation/MainActivity.kt** and replace it with the following Kotlin code to define our **ConstraintLayout**:

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            SportsTrackerTheme {
                HomeScreen()
            }
        }
    }
}

@Composable
fun HomeScreen(modifier: Modifier = Modifier) {
    ConstraintLayout ( modifier = Modifier.fillMaxSize()){

    }
}

@Preview(device = WearDevices.LARGE_ROUND, showSystemUi = true)
@Composable
fun HomeScreenPreview() {
    HomeScreen()
}
```

This defines a white container that takes the full available space in the screen. Next, you will add an **Image** and a **Text** inside the empty ConstraintLayout block. To do so, first, import the same **ic_logo.png** image as before in the Android Wear module.

Then, copy the code of the **Image** and **Text** in the app (mobile) module, and paste in the **ConstraintLayout** block.

We would like the image and the text to be centered in the layout, with the image at the top of the text. To achieve this, set the layout constraints as follows:

```
val (image, text) = createRefs()

Image(
    painter = painterResource(id = R.drawable.ic_logo),
    contentDescription = "Logo Sports Tracker",
    modifier = modifier
        .size(100.dp)
        .constrainAs(image) {
            top.linkTo(parent.top)
            start.linkTo(parent.start)
            end.linkTo(parent.end)
            bottom.linkTo(text.top)
        }
)
Text(
    text = "Hello World!",
    style = TextStyle(fontSize = 24.sp),
    modifier = modifier
        .constrainAs(text) {
            top.linkTo(image.bottom)
            start.linkTo(parent.start)
            end.linkTo(parent.end)
            bottom.linkTo(parent.bottom)
        }
)
```

start.linkTo() and **end.linkTo** allow to define how the elements are positioned horizontally. In this case, you constrained them to be aligned to the parent container in the left and right side, with some margins.

We define how the elements are positioned vertically using similar definitions. In this case, you set the image and the text to be aligned vertically, in the center of the container, with

the bottom of the image linked to the top of the text and vice-versa.

Notice that you also set the size(width and height) of the **`Image`** to **`100dp`**, this is because the image appears too big when the height and width are without modifier.

The resulting Android Wear layout should look like in Fig. 12.

Figure 12: Android Wear Layout