

Lab 1b: Hello Android

In case you found something to improve, please tell us!
<https://forms.gle/Nf27cXFf7AwaBL55A>

1 Introduction

This class teaches you how to build your first Android app. You will learn how to create an Android project and run a debuggable version of the app.

Note: The watch seemingly cannot connect to the WiFi. We present a workaround in the last page, demonstrating the import and use of a small custom app.

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. If not already done yet, install and start Android Studio. You can download it from the Android Studio website¹.
2. Click **Start a new Android Studio project** to create a new project.
3. Now you can select a device and an activity template, from which you begin building your app (Figure 1):
 1. **Form factors** correspond to the screen size and the device types. For this project, select **Phone and Tablet**.
 2. Select **Empty Activity**. Click on **Next**.
4. Fill in the form that appears (Figure 2):
 1. **Name** is the app name that appears to users.
 2. **Package name** is the package name for your app (following the same rules as packages in the Java programming language). 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 your related apps in a unique *Workspace* directory. It can be all the apps created for this course.

¹<https://developer.android.com/studio/>

4. **Language** used on the App programming. Make sure you select **Java**.
5. **Minimum API level** is the lowest version of Android that your app supports. The newest features might be available to older devices but at the cost of an increased app size because of the use of an additional library: **androidX.* artifacts**. 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**. This default configuration includes the **androidX.* artifacts** library mentioned before.

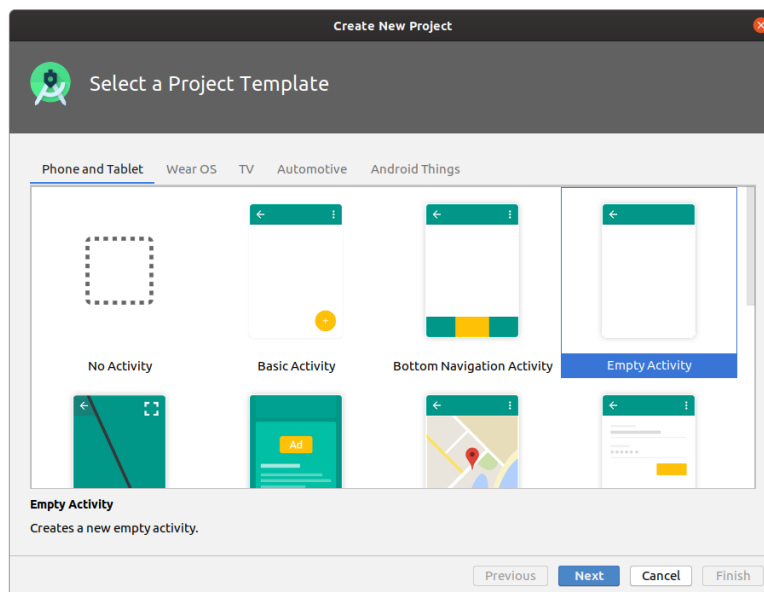


Figure 1: Wizard for creating a new project with Android Studio - choosing activity.

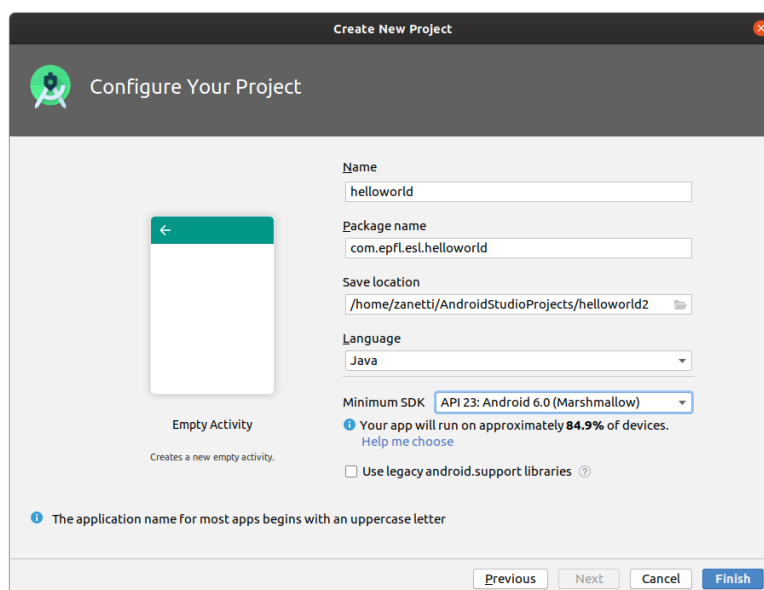


Figure 2: Wizard for creating a new project with Android Studio - configuring your project.

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 various declarations in this file as you read more training classes.
- **app/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.
- **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**.
 - **layout/** This directory is for files that define your app's user interface.
 - **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 **build.gradle** files. They are scripts used by the build system to compile the app. This is the place where the project's dependencies are defined, where the Android API level is specified or where the C++ source files are found. 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 World!". The result is nothing exciting, but it's important that you understand how to run your app before you start developing.

4 Running your app

You can run your app on a real device or an emulated device. This section shows you how to install and run your app on a real device and on the Android emulator. Note that using

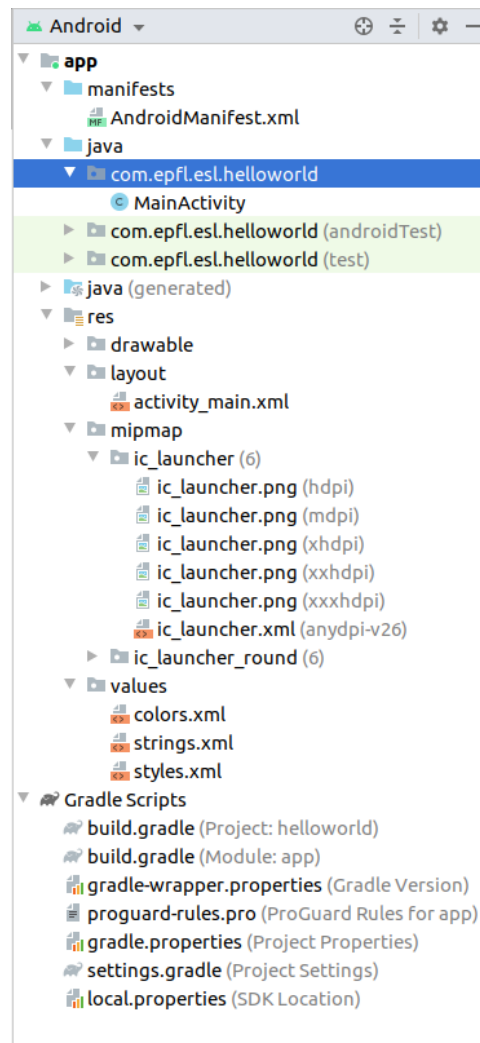


Figure 3: Structure of an Android project


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 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 the **USB debugging** one.
3. From the Android Studio toolbar, select the **app** configuration and click the **Run** button . 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 Virtual Device Manager by clicking on the **AVD Manager**  from the Android Studio toolbar.
2. A device might already be present by default in the list. If needed, in the Android Virtual Device Manager panel, click **Create Virtual Device**.
3. 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.
4. The list of all system images currently on your computer is displayed. If the Android version you need is not displayed, click **Show downloadable system images**. Select the version, which is running on your device and click **Download**. The image will be downloaded and installed.
5. In the final screen you can choose if you want your AVD to be displayed in portrait or landscape mode. Note that you can always change the orientation when the emulator is running.

Running the app is the same as using a real device: the emulated devices show the list of devices to choose from to run the app.

5 Learning the basics of Android Studio

Android Studio is a development environment that provides many little but useful tools to have a more pleasant experience. Here we will show you few of them, but feel free to

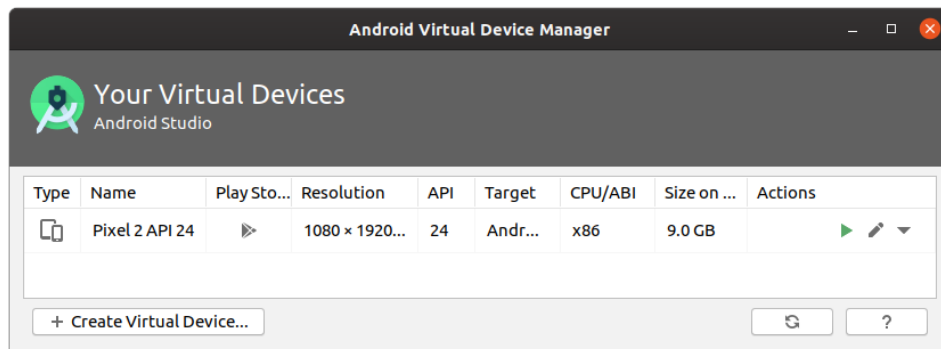
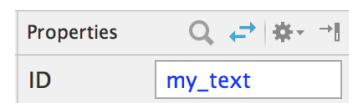
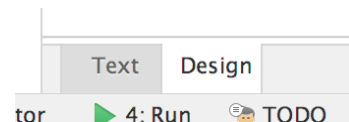


Figure 4: The Android Virtual Devices (AVD) manager

explore the software.

First, open the file `app/res/layout/activity_main.xml`. This is the file describing the user interface (UI) of our app, read by Android to do the layout of the different UI elements (called Views). In the future, we will see how to edit it manually.

Now, select the **Design** tab from the bottom of the window. Click on the text element saying "Hello World!". From the properties panel on the right, give it an ID, such as `my_text`. This is the unique identifier of this View, which we can access from the java code we are about to write.



From the `app/java/my.package.name` folder, open the `MainActivity` file. Complete the only existing function so that it looks like this:

```
@Override protected void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    TextView view = (View) findViewById(R.id.my_text);  
    view.setText("This is my first app!");  
    setContentView(R.layout.activity_main);  
}
```

With the two lines added, between `super.onCreate(...)` and `setContentView(...)`, we are trying to replace the "Hello World!" text with our own text. Unfortunately, it has several problems, which we will fix in the upcoming sections.

5.1 Fixing errors and code cleanup

The code we gave you has some problems, one of which prevents building the app as it is a type error: the result of `findViewById(R.id.my_text)` is casted to the `View` type, whereas the variable `view` has the type `TextView`. When using the keyboard shortcut **Alt+Enter** while the edit cursor is on the line, a contextual menu will suggest you different ways to fix the problem.

Once fixed, there is one more thing to do: casting the result of `findViewById(...)` used to be required for API level below 26, but now the explicit cast is optional, we can drop it altogether. When your edit cursor is in the greyed (`TextView`), you can choose the *Remove redundant cast(s)*. Choosing *Cleanup code...* can apply this on the whole file.

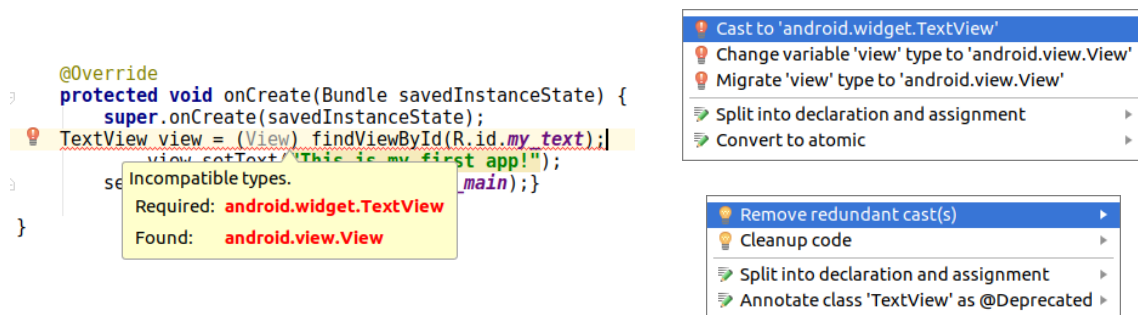


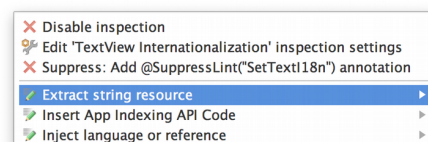
Figure 5: Fixing and cleaning the code

5.2 Applying suggestions to follow “Good practices”

Another problem of the current code is the use of a **hard-coded string** in the java code. It is not a

problem for debug purposes (i.e. write to a log file) but it should be avoided for released apps. Indeed, it’s a good practice to separate the app’s text strings in a resource file and the java code, the same way the display layout is separate from the java code.

One benefit of separating the text is to easily be able to provide a translated app, because Android will automatically use the user’s locale if the translated strings are provided by the app. Using the same **Alt+Enter** shortcut, it’s easy to extract the string as a resource.



5.3 Tidying up the code style

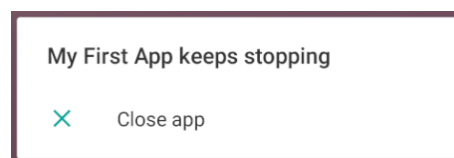
Editing after editing, especially with multiple programmers, the code can be messy (irregular bracket position, spaces, unclean indentation). This can be fixed using **Code > Reformat code**. You will soon know the shortcut **Ctrl+Alt+L** by heart!

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    TextView view = findViewById(R.id.my_text);
    view.setText(R.string.my_custom_message);
    setContentView(R.layout.activity_main);
}
```

5.4 Using Logcat

You might have tried to run the code but the application crashes! The system logs everything happening and this will be useful to find out where the bug is.

If you open the Android Monitor (or Logcat) tab in the bottom toolbar of Android Studio, you will see the stack trace that triggered the crash:



```
/ch.epfl.esl.myfirstapp D/AndroidRuntime: Shutting down VM
/ch.epfl.esl.myfirstapp E/AndroidRuntime: FATAL EXCEPTION: main
Process: ch.epfl.esl.myfirstapp, PID: 4260
java.lang.RuntimeException: Unable to start activity ComponentInfo{ch.epfl.esl.myfirstapp}: java.lang.NullPointerException: Attempt to invoke virtual method 'void ar
    at android.app.ActivityThread.performLaunchActivity(ActivityThread.java:2817)
    at android.app.ActivityThread.handleLaunchActivity(ActivityThread.java:2892)
    at android.app.ActivityThread.-wrap11(Unknown Source:0)
    at android.app.ActivityThread$H.handleMessage(ActivityThread.java:1593)
    at android.os.Handler.dispatchMessage(Handler.java:105)
    at android.os.Looper.loop(Looper.java:164)
    at android.app.ActivityThread.main(ActivityThread.java:6541) <1 internal calls>
    at com.android.internal.os.Zygote$MethodAndArgsCaller.run(Zygote.java:240)
    at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:767)
Caused by: java.lang.NullPointerException: Attempt to invoke virtual method 'void ar
    at ch.epfl.esl.myfirstapp.MainActivity.onCreate(MainActivity.java:13)
    at android.app.Activity.performCreate(Activity.java:6975)
    at android.app.Instrumentation.callActivityOnCreate(Instrumentation.java:1213)
    at android.app.ActivityThread.performLaunchActivity(ActivityThread.java:2770)
    at android.app.ActivityThread.handleLaunchActivity(ActivityThread.java:2892)
    at android.app.ActivityThread.-wrap11(Unknown Source:0)
    at android.app.ActivityThread$H.handleMessage(ActivityThread.java:1593)
    at android.os.Handler.dispatchMessage(Handler.java:105)
    at android.os.Looper.loop(Looper.java:164)
    at android.app.ActivityThread.main(ActivityThread.java:6541)
    at java.lang.reflect.Method.invoke(Native Method) <2 more...>
```

Figure 6: Logcat capture of a crash

In our case, there is not too much to read from that text. Most of the time, clicking the first blue link takes us where the crash happened: **MainActivity.java**, line **13**. The line before explains what kind of crash happened. In that context, it's a **NullPointerException**. In other words, we are trying to use an uninitialized (null) pointer to an object.

The crash happened when we called **view.setText(...)**, which means that our **view** variable is null. The only reason why that can happen is because **findViewById(...)** couldn't

find the element we are looking for. This is because the layout is built only later, thanks to the `setContentView(...)` call. Moving this `setContentView(...)` function call immediately after the `super.onCreate(...)` and before the `findViewById(...)` will solve the problem:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    TextView view = (TextView) findViewById(R.id.my_text);
    view.setText(R.string.my_custom_message);
}
```

5.5 Renaming variables (the smart way)

The last problem one can find in our code is that the *name* we gave to the `TextView` variable is **objectively terrible**. It is so generic that it becomes unhelpful. 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**.

5.6 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 colored depending on its importance:

```
// It's common to create class member named TAG of type String
// Class fields usually at the top of the class
// Here is a small explanation of the following line:
//   - private: it is not visible/accessible from other classes
//   - final: it can only be assigned once (it won't change)
//   - this: it's a reference of the current instance of the class
private final String TAG = this.getClass().getName();

// A function printing to logcat
private void demo_logcat() {
    Log.v(TAG, "Verbose");
    Log.d(TAG, "Debug");
}
```

```

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


```

Add the class field **TAG** of type **String** in the class, along with the **demo_logcat()** method. Create a new **demo_logcat()** method call (for example in **onCreate()** ..)). The Logcat output is as displayed on the right, which we have customized in the Android Studio's settings to make it more colorful. You can import this color scheme by using the files available on Moodle:

```


/ch.epfl.esl.myfirstapp V/MainActivity: Verbose
/ch.epfl.esl.myfirstapp D/MainActivity: Debug
/ch.epfl.esl.myfirstapp I/MainActivity: Information
/ch.epfl.esl.myfirstapp W/MainActivity: Warning
/ch.epfl.esl.myfirstapp E/MainActivity: Error


```

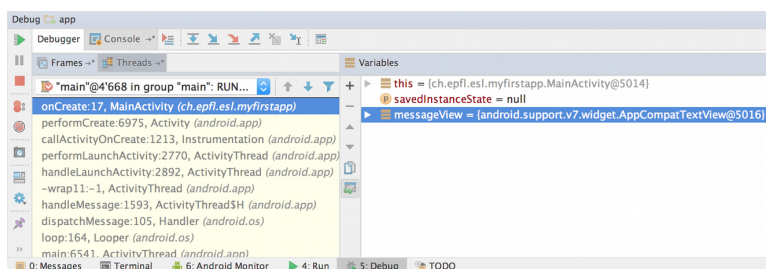
1. Open Android Studio's settings (called preferences on Mac),
2. Access **Editor > Color scheme > Android Logcat**,
3. Click on the gear icon  to show more actions, then select **Import Scheme**,
4. Choose between dark and light scheme and **Apply**.

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.

5.7 Proper debugging

To find out what is exactly happening without having to log everything, it's best to use breakpoints. Click near the line number to set a breakpoint: **17**  **demo_logcat();**

Now, 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 the Android programming a pleasant (or at least, efficient) experience. Feel free to explore the other features available, such as the Profiler in case of performance problems, as they might save you some time in the future.

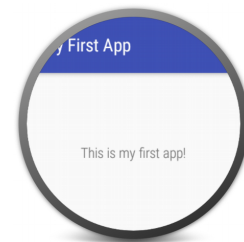
6 Run the app on a smartwatch

To illustrate the similarity between programming for an Android smartphone and an Android wear smartwatch, we will transform the current project so as to run it on a watch. Add the following line in the **AndroidManifest.xml**, in the **<manifest>** tag, above the **<application>** tag:

```
<uses-feature android:name="android.hardware.type.watch" />
```

Now, if you try to run the app on the smartphone, the device selector will complain about the **missing WATCH** feature. On the other hand, it will accept a smartwatch (whether real or emulated). If you do not see the smartwatch in the device selector window, check if the USB debugging is enabled. If it is not, follow the explanations in the section “Run on a real device”.

Please **keep in mind** that it is advised to use the dedicated Android wear libraries, which provide solutions for handling round screens and the other smartwatches particularities.



7 Using a custom watch app to connect to EPFL's WiFi

From the user interface of the watch, it is impossible to connect to EPFL's wifi. To do so, we created a small app that asks for the login (gaspar) and password to register both **EPFL** and **eduroam** networks. You can open the *Connect2Wifi* app sources in Android Studio, compile and run it on the watch. This way, you will get faster OS and app updates!