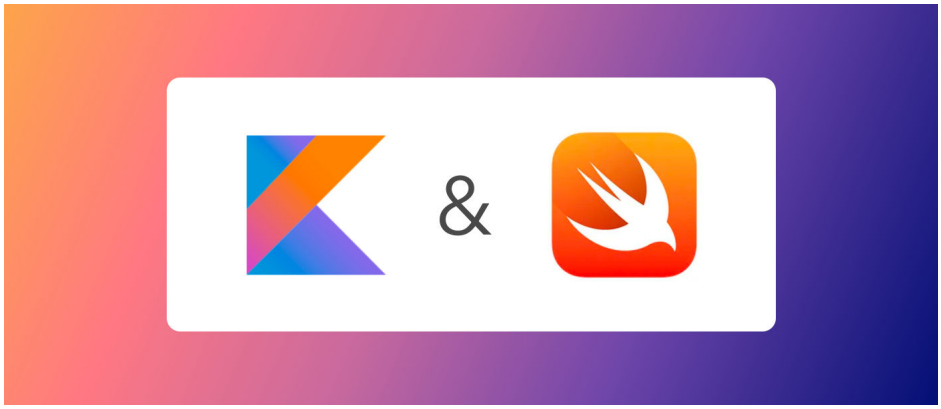
M3 - Mobile

M-MOB-300

Mobile

Create a StopWatch



2.0

{EPITECH.}

Mobile

language: Kotlin / Swift

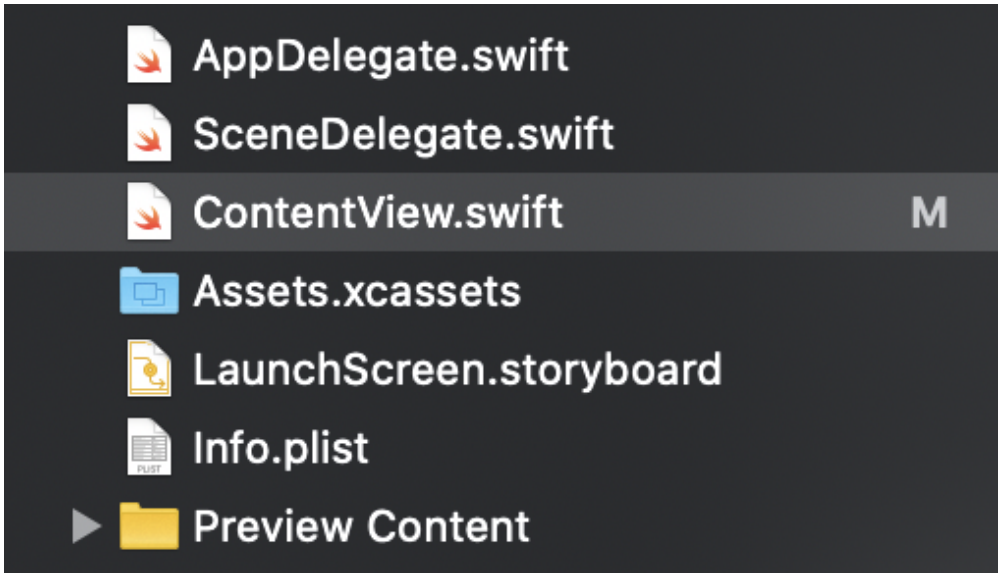
- Your repository must contain the totality of your source files, but no useless files (binary, temp files, obj files,...).
- All the bonus files (including a potential specific Makefile) should be in a directory named *bonus*.
- Error messages have to be written on the error output, and the program should then exit with the 84 error code (0 if there is no error).

Today we will see how to create a StopWatch App. For the workshop, we will be using Swift and Kotlin. We will, therefore, be creating a native mobile app.

SET UP THE WORKSPACE

+ iOS

To create a native iOS mobile app, you will need macOS and XCode. Create a new Single View Project in XCode. Let's start by understanding the files created.



AppDelegate.swift: Handles Lifecycle Events (we won't be touching this file during this workshop)
SceneDelegate.swift: Handles which Scene should be displayed, in our case, the default View will appear.
ContentView.swift: This is code for your view; we will be doing all the code in this file.
Assets.xcassets: Is where all the assets will be loaded from
LaunchSceen.storyboard: It is the storyboard of your app.
Info.plist: It contains info on the app.
During this WS, we will only be touching **ContentView** file.

+ ANDROID

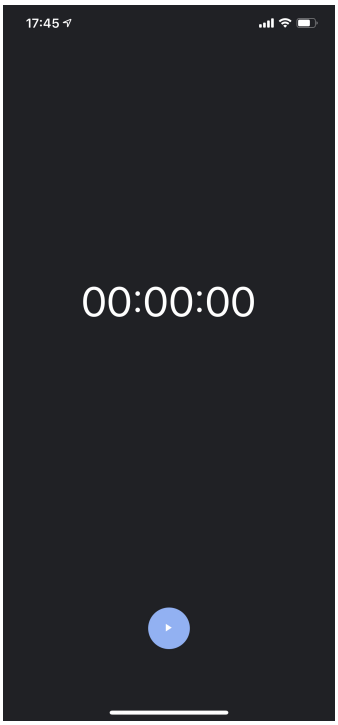
To create a native Android mobile app, you will need Android Studio and any OS that supports Android Studio. Once the project created, you will see a lot of folders are present; in this WS, we will be working on two files.
app/src/main/res/layout/activity_main.xml: Where we write the UI Code
app/src/main/java/com/example/myapplication/MainActivity.kt: where we write the app's logic.

OBJECTIVE

We will now see the different steps to creating a mobile App.

+ DESIGN

Before coding the app, you will first have to design how the app will look and feel. We call this designing the UI/UX of the app. In our case, we don't have the time to do this; we will base our app on the default clock app on your phone.



+ CODING THE UI

Once designing UI/UX is done, you will need to implement it.

iOS

You will first want to add the different elements to your view (button, labels…).
Your **ContentView** file should be similar to the image below.

```swift
import SwiftUI

struct ContentView: View {
    var body: some View {
        Text("Hello, World!")
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

The first struct **ContentView** is the code to your view. All the elements you want to display can be found in

the **body**. If you try to add multiple elements, XCode might show an error. To solve this, you will need to add a VerticalStack.

```
import SwiftUI

struct ContentView: View {
    var body: some View {
        VStack {
            Text("Hello, W!")
               .font(.title)
            Text("Hello, W!")
        }
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

With the **VStack**, you will be able to multiple elements. Try adding various Text elements and buttons. You may also want to try personalizing the different items you created (change the size, color, etc…)
Documentation for SwiftUI

ANDROID

You will first want to add the different elements to your view (button, labels…).
In **app/src/main/res/layout/activity_main.xml**, you will be able to add the different elements you want. You may find it easier to use the Graphical Editor, or you may add the elements yourself in the xml file.

```xml
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

Documentation for UI in Android Studio.

Code the App's Logic

Once you finished the UI, you may now code how the app works. In our case, you need to be able to start, stop, and then restart the timer.
The intricate part is displaying the elapsed time, having the perfect balance between being precise, and consuming too many resources.

Furthermore

Once you are done, call us to show us the final product. You may continue the project by, for example, adding a timer?