Mobile Computing

Practice # 1

First Android Application (in Kotlin)

1. Create an Android project in Android Studio for a first application

a) In Android Studio windows select **Projects** and then click **New Project**. A form should appear.

In the form choose the Phone and Tablet tab at the left side, and then 'Empty Activity' (this creates an 'empty' skeleton project). Click on **Next** at the bottom.

b) Fill the **next** form form in this way:

Name is the name of your application. The project directory, and the name visible in Android Studio are composed from this name, removing the spaces. If it is composed by several words, capitalize them. For instance **First App**

Package name is the unique identification of your application and usually contains also your organization and personal identifier inside it. It should be specified as a domain name in reverse (several sections separated by a dot): for instance, starting with ***org*** or ***com*** (non-profit or commercial organizations), organization name, developer or team name, and finally the app name. In my case, for instance, it could be: ***org.feup.apm.firstapp***. Choose one appropriate for you.

Each Android app should have a **unique** identity (when submitted to the app store), expressed as this Java package, grouping the classes you write.

Save location is the directory where all the project files are stored. You should use a directory with the Application Name, inside some other directory.

Language specifies the programming language. It can be Java or Kotlin. Choose Kotlin.

Minimum SDK is minimum android API version where your application will run. That means it will not install on devices with an API level older than the one specified. Also, you **cannot** call in your code any method defined in newer versions of the API (unless you use an external static support library with it). So, this setting should be a balance between the features we want and can use, and the base existing devices that we want to support. For now, specifying as minimum API, the level 25 (Nougat 7.1.1) guaranties support to about 85% of the world Android devices …

Leave the other settings as they are (not selecting 'Use legacy android.support libraries'), and click the **Finnish** button.

b) After Android Studio creates your project, wait for the first build to finish. Then examine the files in the project directories (app folder in the Android logical structure), specially the AndroidManifest.xml, the source (a Kotlin file), and the resource files (inside the subdirs of the **res** directory).

You can see in Android Studio, under the app/java project directory, three namespaces: one is the chosen project namespace, and the other two are for testing (using TDD) with two different frameworks (android test (espresso) and unit tests with JUnit). For now, you can delete those two projects (start with the files inside each).

Also, going to the File → Project Structure → Dependencies → app tab (or the corresponding button on Android Studio toolbar) dialog box, you should delete the espresso-core, and the two junit external dependencies (libraries). This will lower the app size.

We will use the 'Up Navigation' functionality that was automatically available only after API level 16. As we are supporting Android versions with API 25 or higher, we don´t need to use an external library for that.

c) Now we will add the code for the first activity in our application. The file and class are already created with the name **MainActivity**. Just open it in the editor.

d) Try to run the application in several emulators and real devices (connected through USB to the development computer). You should see a screen with only an Action Bar (title) and a line of text in the middle of the screen. The style of the app was already configured by Android Studio following the recommended design guidelines and using a derived Activity class (***AppCompatActivity***) from a support library, guaranteeing uniformity across API levels (the base ***Activity*** class in the Android library has changed along the API versions).

2. Modify the interface and behavior

Modify the generated project in the following way:

i. Open the res/layout/activity_main.xml file in design mode. You can see that it contains a ***ConstraintLayout*** and a ***TextView***.
ii. In the Component Tree select the **ConstraintLayout** and in the context menu (mouse right button) use Convert View … to convert it to a **LinearLayout**.
iii. Delete the **TextView**
iv. Adjust the properties to obtain the following (see the file in Code mode).

```
<LinearLayout
   xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:app="http://schemas.android.com/apk/res-auto"
   xmlns:tools="http://schemas.android.com/tools"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   tools:context=".MainActivity">
</LinearLayout>
```

v. You can now delete, in the Project Structure Dependencies tab, the **constraint layout support library**, as we do not need it anymore.

Try now to use the visual editor (design tab) in the **res/layout/activity_main.xml** file **to**:

a. Add a text field (PlainText, implemented as an EditText view with the id edt_message) empty (delete the text property) but with a hint ("Enter a message") inside the LinearLayout. All the interface strings should be first defined in the strings.xml resource file (res\values), and referenced with the syntax @string/<string name>. For the hint string use edt_message_hint as its name.
b. Add a button, also inside the LinearLayout but following the text field, with a label "Send" (put it also in the strings resource with name bt_send) and an **id** of also bt_send.
c. Try that the combination of text field and button occupy the screen width with the button with a minimum size:

For that, the property layout_width of the **EditText** view can be defined as zero: 0dp, but with the property layout_weight as 1.
Android tries to allocate space to interface elements proportional to their weights, but using the minimum to be visible. When a weight is not specified it is assumed to be 0 (delete the weight value for the button if needed). Also try to put some margin on the right side of the button.
d. The strings and layout XML files should now contain:
Strings:

```
<resources>
   <string name="app_name">First App</string>
   <string name="edt_message_hint">Enter a message</string>
   <string name="bt_send">Send</string>
```

```
</resources>
```

Layout:

```
<LinearLayout
   xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:tools="http://schemas.android.com/tools"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   tools:context=".MainActivity">

  <EditText
     android:id="@+id/edt_message"
     android:layout_width="0dp"
     android:layout_height="wrap_content"
     android:layout_weight="1"
     android:hint="@string/edt_message_hint"
     android:inputType="text" />

  <Button
     android:id="@+id/bt_send"
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:layout_marginEnd="5dp"
     android:text="@string/button_send"/>
</LinearLayout>
```

e. Run the application and confirm the layout, verifying the **EditText** box behavior and functionality on the screen.

3. Create a second activity

a) Using the Android Studio File/New menu entry, create another activity. Follow File → New → Activity → Empty Activity.

b) In the **EmptyActivity** form, fill it this way:

Activity Name: SecondActivity
Deselect all the checkboxes in the form (don't generate a layout and this is not a launcher activity). Click ***Finnish***.

c) Verify that the activity is a new file, and that it is already in the AndroidManifest. Add (in the Manifest) the following property to this activity element: android:parentActivityName=".MainActivity".

4. Send and receive a message while navigating to the Second activity

This second activity should display the message sent with the intent that activates the activity (this intent will be built and sent by the **MainActivity** send button listener).

So here, we need to get the activation intent, extract the message, and show it in this activity screen. We could specify a layout in a XML layout file, but instead, and because this one is extremely simple (only a text in a TextView), we'll do it in code. It is always possible to build layouts in code, but it is not very practical nor recommended, because it can promote some confusion between interface and business logic (not in this case).

Edit the **onCreate()** method of the **SecondActivity** as follows:

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

  // Create a TextView and fill it with the string that comes on the Intent
  val tv = TextView(this)
  tv.text = intent.getStringExtra(EXTRA_MESSAGE)
  tv.textSize = 30.0F

  // Set the TextView as the Activity screen
  setContentView(tv)
 }
```

The EXTRA_MESSAGE constant string will be defined in the main activity file.

5. Respond to the Send button and activate the SecondActivity activity.

a) We need to add a listener (aka callback function) to respond (being automatically called by Android) when the user clicks the Button on the main activity. This will be the handler of the onClick event of the button.

For that we need first to obtain the button as an object in the code (usually in onCreate()), and then calling setOnClickListener() and supplying the code to act as the callback (usually an anonymous function written immediately, using the lambda notation).

Obtain the button object calling findViewById:

```
findViewById<Button>(R.id.bt_send)
```

(use Alt+Enter to update the import statements in the .kt files)

b) Add the listener to respond to the click button event calling setOnClickListener() on the Button object.

The function parameter can be a function (the listener or callback; Kotlin (as Java) allows that if the corresponding interface contains only a single abstract method (SAM)) that will be called when the user clicks the button (or can be an object that implements the corresponding interface). It receives a parameter and executes the listener instructions. The better is using the lambda function literal with the operator -> (denotes function).

c) The listener code must build an explicit intent pointing to the second activity, extract the text from the **EditText** box and attach it to the intent. After that, we can start the second activity. The listener definition, attached to the click event of the button should be:

```
findViewById<Button>(R.id.bt_send).setOnClickListener { _ ->                // the parameter is not used
  val message = findViewById<EditText>(R.id.edt_message).text.toString()
  val intent = Intent(this, SecondActivity::class.java)
  intent.putExtra(EXTRA_MESSAGE, message)
  startActivity(intent)
 }
```

For a click listener, the handler receives a parameter containing the View where the click was performed. In this case we don't use it, so in the lambda we can maintain it anonymous using a _.

d) Extra information attached to intents have a name (pair name/value). One last thing is to define that name. Let's do it as a constant top-level string in this file. Add the constant EXTRA_MESSAGE as:

```
const val EXTRA_MESSAGE = "org.feup.apm.firstapp.MESSAGE"
```

The <package name> is usually included in the string name to make that name unique.

6. Run and use your first two activities application

Try in some emulators and real devices.