Lab 2 - Kotlin basics. 2-activity app, layouts, buttons.

In this assignment you will create your own Android Kotlin app from scratch. You will learn how to start an Activity from another Activity via Intents, and how to pass data between activities via Intents. You will also learn about different layouts and UI widgets.

Your app is going to consist of the following activities:

- **RegistrationActivity** - This activity lets the user enter his/her name. From this activity the user is forwarded to:
- **ProfileActivity** - This activity shows the name entered earlier and shows a user picture. The user is also given an option to enter a message that will be emphasized on the screen.

Let's start programming the app!

Start Android Studio and create a new Kotlin language project with no Activities named `ProfileApp` with the package name `si.uni_lj.fri.pbd.lab2`

Browse around the tree structure in the window on the left side of the screen - those are your project files. Android Studio usually tries to build the app automatically. Inspect `build.gradle` files - there are two of them, one for your project, and one for the only module you have in the project ("app"). **Make sure your project uses API 31 as a compileSdkVersion.** Although your project does not contain much at the moment, it might be useful to check out the output window at the bottom of the screen. If any errors are reported, read the reports and try to fix them (hint: stackoverflow.com is your friend!).

We will add a screen that, once our app is started, shows the user the terms of use, lets him/her type in the name, and register. Create a new Empty Activity using the File->New->Activity menu. Call it `RegistrationActivity` , for the package name use the same domain name as before, and tick a box that indicates that a layout file will be created (name it: `activity_registration`) .

Android Studio will automatically open `RegistrationActivity.kt` a file where the code for this activity lives. The code is in Kotlin, so let's take a few seconds to get familiarized with this language:

- Kotlin is an object oriented language, thus, you have your classes, just like in Java. Here, we declared a class **RegistrationActivity**. By default, this declaration is public.
- Inheritance is supported by "`:`" following the class description. Thus, our **RegistrationActivity** inherits from **AppCompatActivity** class (i.e. ":" is equivalent to "extends" in Java)
- Functions (equivalent to "methods" in Java) are declared with the "`fun`" keyword. "`override`" keyword indicates that we are overriding the function originally declared in the super-class (i.e. in **AppCompatActivity**)
- Declaring arguments within a function is done by stating the name of the argument followed by its type. Here we have `savedInstanceState` variable of type `Bundle?`.
- Null safety is one of the greatest benefits of Kotlin. By default, Kotlin does not allow you to set a variable to `null`. To do that, you must explicitly state that the variable can be assigned a null value. You do that by appending a question mark after the type assignment. You can see that our `savedInstanceState` variable is of type `Bundle`, but can be set to null.
- Calling function is in this example done just like in Java: e.g. we call the `onCreate` method of the super-class with `savedInstanceState` parameter, and we call `setContentView` (also of the super-class, but no need for "super") with `R.layout.activity_registration` parameter.

Our class extends `AppCompatActivity` - this is for backwards compatibility with older Android versions. The code in the file tells the activity how to behave, but the looks of the activity are defined elsewhere. You should see `setContentView(R.layout.activity_registration)`. This line tells RegistrationActivity that it should render the user interface defined in the `activity_registration` file. Find this file in the `res` folder on the left. This is where layouts, written in XML language, live.

Open `activity_registration.xml`. You can switch between the text (XML) view and a rendered (Design) view by clicking on these icons: 

NOTE: To speed things up, you may download `activity_registration.xml` from ucilnica!

Add a `TextView` element to your layout by dragging it from the menu on the left to the layout screen. Since this view will hold the title of our activity we want to make the font size large. On the right you will see the Declared Attributes menu, where you can add `textAppearance` property and set it to `AppCompat.Large`.

See how your layout "looks" in XML.

NOTE: elements (Views) in the XML file have unique IDs. You should set those IDs to values that make it easy for you to tell which element you're referring to in the code. E.g. `android:id="@+id/text_title"` for the above view that will hold the title.

Add one more `TextView` below the view you have already created. This one will hold our consent terms.

Now add a `Plain Text` (aka `EditText`) element below. This is where the user will sign their name.

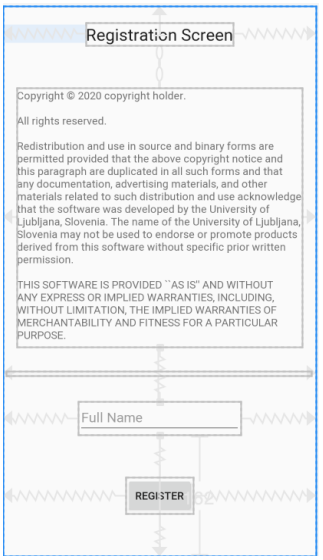
Then, add a button at the bottom. Finally, add a horizontal line between the `TextView` with our consent terms and the `EditText` where a user's signature will be typed. You can add such a line by putting a Horizontal Divider widget from the menu on the left.

Give descriptive ID to your elements, e.g. `btn_accept` to the Button, `edit_name` to the EditText, etc.

Now we want to add the right text to the fields. We are not going to hardcode values within the XML, because we might want to repeat text in different parts of the app or we might want to translate our app to different languages. In Android string values are usually stored in an XML file called `strings.xml`. We have already provided this file for you, so just download it from ucilnica and put it instead of your default `values/strings.xml`, inspect the contents of this file.

You should now refer to these values in the layout file. Open the `activity_registration.xml`, switch to Text tab, and replace `android:text` with appropriate references, e.g. `"@string/reg_consent_text"`. In the `EditText` element, add `android:hint="@string/reg_full_name_text"`. You will do similar with any other strings that appear in the app.

Everything you added so far is residing within a `ConstraintLayout` (see in the XML code). This layout allows you to set constraints among elements - defines how they will relate to each other (top/bottom/left/right). Experiment a bit with the elements until you get something like this:



NOTE: Designing Android User Interface (UI) should follow (current) best practices in Android UI programming. While we won't be able to cover it in this, you should definitely read about the guidelines here https://developer.android.com/design/

Back to our app - to link the "Register" button with an action, we have to capture `onClick` event. There are multiple ways of accomplishing this. For example, we can add the following property to the `Button` element in the XML layout file: `android:onClick="registerUser"`. This means that the `registerUser` function will be called.

We now go back to the **RegistrationActivity** to implement `registerUser` function. The function has to take a View object as an argument, i.e. its declaration should be:
`fun registerUser(view : View).`

In the function, we want to check if a user filled out the username and if not, show a warning. First, we need to access the `activity_registration`'s `EditText` element. To do so, we can use the `findViewById(int)` function of the Activity class. The function requires an ID that corresponds to the ID we used for the EditText element in the layout XML file and returns a View, which we should *cast* to EditText. In Kotlin we can do that in multiple ways, the most common being "as", i.e.:

```
val nameText = findViewById(R.id.edit_name) as EditText
```

A few things to note here:

- `val` is a Kotlin keyword indicating an immutable object, i.e. the one we cannot change after the initial assignment. Note - this doesn't mean we cannot change something within that object!
- we did not explicitly state what is the data type of `nameText`, it was automatically inferred after we cast the found object to `EditText`.

We can drill deeper into the returned (Edit)View to get the value of the entered text:

```
val fullName = nameText.text.toString()
```

Check if it's length is larger than zero. If not, we can show a warning via the EditText's `setError(String)` method. Remember that you should not hardcode any text that is going to be shown to the user.

Once the user fills out the requested info, and clicks the button, the entered information should be forwarded to another activity - **ProfileActivity**. To start an Activity from another Activity we use Intents. An intent takes the name of the class it needs to launch. We will bundle a string with the user's full name to the intent, like this:

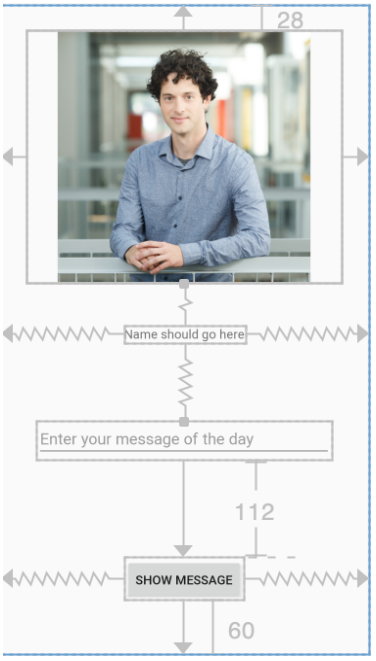
```
val intent = Intent(applicationContext, ProfileActivity::class.java)
intent.putExtra(EXTRA_NAME, fullName)
startActivity(intent)
```

The `putExtra` method takes the name of the extra resource (labeled with a public String constant `EXTRA_NAME`), and the value that needs to be sent to another activity (a user's full name). But, where is that constant? We need to create it. In Kotlin, we can create constants within a *companion object*. This object is shared among all instances of the class, the same as a static field or method would be shared in Java. Use the reserved word const for your compile-time constants. At the beginning of your RegistrationActivity put:

```
companion object {
    const val EXTRA_NAME = "si.uni_lj.fri.pbd.lab2.FULL_NAME"
}
```

We still don't have the target activity, so create it! Just like before, create a new Empty Activity, call it ProfileActivity (for package name use the same as you used earlier), the layout file should be called `activity_profile`. To complete **ProfileActivity**, we need to draw its interface. Open `activity_profile` and put an `ImageView` object on it, as well as a `TextView` object below. Underneath that add an `EditText` object. Finally add a button. The `TextView` object will show a user's name, while the `EditText` button will let the user input their "message of the day". Clicking on the button will show the message up close using a Toast.

You should mimic the process from the first activity, and your end result should look something like this (I dragged and dropped my photo in `res/drawables` and referred to it using `@android:src` property of ImageView):



You may want to try your app at this moment - run it in the emulator and see whether it transitions to the second activity. Perhaps you need to fix the Manifest file?

Remember that we sent the user's name with our Intent from RegistrationActivity. To read it in the ProfileActivity, we will unpack the "extra" from the Intent. In the `onCreate` do the following:

```
// Get the name from the intent
val intent = getIntent() as Intent
val fullName = intent.getStringExtra(RegistrationActivity.EXTRA_NAME) as
String
```

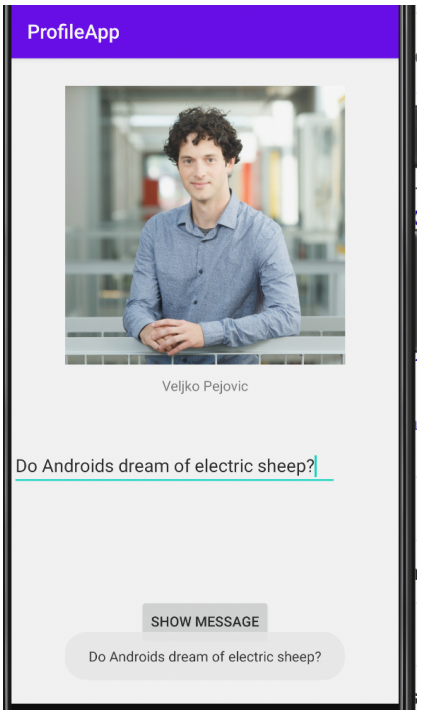
Then, find the **TextView** that should contain the name and set the text to the above variable `fullName`.

Finally, we should define what happens when a user clicks on the button in **ProfileActivity**. In the first activity we specified the function to be called when a button is clicked on directly in the layout (via GUI, but also possible via XML). An alternative is to define what happens when you click on a button in the Activity itself, more specifically in its `onCreate` method.

In the `onCreate` method of **ProfileActivity** use `findViewById` to find the button, and then use `setOnClickListener` to programmatically set the behavior on a click. This is the code that works **in Java:**

```
Button msgButton = (Button) findViewById(btn_msg);
msgButton.setOnClickListener(new View.OnClickListener() {
      @Override
      public void onClick(View v) {
            Context context = getApplicationContext();
            EditText editMsg = (EditText) findViewById(R.id.edit_msg);
            String msg = editMsg.getText().toString();
            int duration = Toast.LENGTH_LONG;
            Toast toast = Toast.makeText(context, msg, duration);
            toast.show();
      }
});
```

And it will show a Toast with the message that the user put in the EditText. An example can be seen in the screenshot below:



In Kotlin, you should be able to accomplish the above with fewer lines of code - do it!

That's it! We're done with programming. If your app still has problems recognising all the Activities, you just have to ensure that the app knows what all of its parts are. We dig into `AndroidManifest.xml` now. Each activity should be listed there. The main one is **RegistrationActivity**, it's the first that gets launched:

```
<activity
    android:name=".RegistrationActivity"
    android:label="@string/title_registration_activity" >
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

Any other Activities need to be listed, but no special intent-filters are needed.

Test your app. If you see any problems use the debugging tools including Logcat, Traceview and UI hierarchy viewer.

If you were not present in the lab for solving this assignment you should commit your solution to a private repository named **PBD2023-LAB-2** in your Bitbucket account. User **pbdfrita (pbdfrita@gmail.com)** must be added as a read only member of this repository. The code must be committed by Sunday (March 5th) 23:59.

Happy coding!