Mobile and Ubiquitous Computing

Labsheet 2 - Interactivity for Android

This labsheet assumes you have completed Labsheet 1. If you have not completed Labsheet 1 then please do so before starting this sheet.

Starting with Android programming

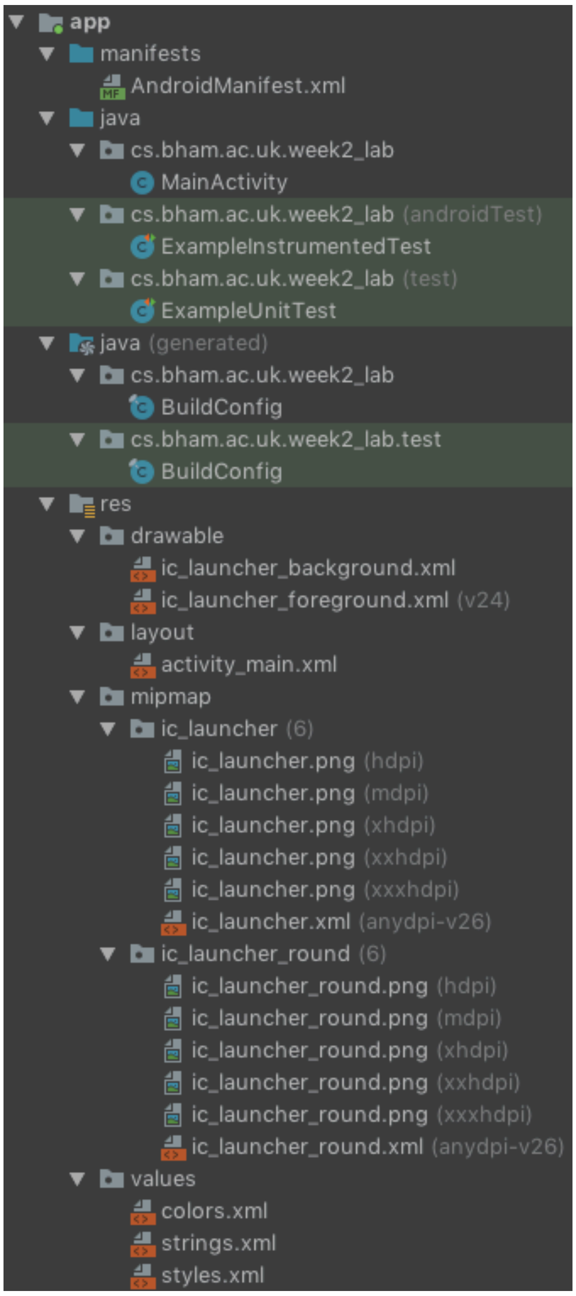
Last week we had a go at getting Android Studio up and running, getting a template application built and getting it running in the Android Emulator. This week we are actually going to start doing some programming.

Today, you will:

1. Have a go at editing a template application to add additional functionality to it.
2. Adding simple widgets to the app by adding *Views* to it.
3. Beginning to experiment with more complex *Views*.
4. Developing in class files to add behaviour to these *Views*.

The goal of this lab is to get you to start understanding how apps work so that we can start laying more complex functionality in future weeks.

Structure of a basic app



```
▼ app
  ▼ manifests
      AndroidManifest.xml
  ▼ java
    ▼ cs.bham.ac.uk.week2_lab
        MainActivity
    ▼ cs.bham.ac.uk.week2_lab (androidTest)
        ExampleInstrumentedTest
    ▼ cs.bham.ac.uk.week2_lab (test)
        ExampleUnitTest
  ▼ java (generated)
    ▼ cs.bham.ac.uk.week2_lab
        BuildConfig
    ▼ cs.bham.ac.uk.week2_lab.test
        BuildConfig
  ▼ res
    ▼ drawable
        ic_launcher_background.xml
        ic_launcher_foreground.xml (v24)
    ▼ layout
        activity_main.xml
    ▼ mipmap
      ▼ ic_launcher (6)
          ic_launcher.png (hdpi)
          ic_launcher.png (mdpi)
          ic_launcher.png (xhdpi)
          ic_launcher.png (xxhdpi)
          ic_launcher.png (xxxhdpi)
          ic_launcher.xml (anydpi-v26)
      ▼ ic_launcher_round (6)
          ic_launcher_round.png (hdpi)
          ic_launcher_round.png (mdpi)
          ic_launcher_round.png (xhdpi)
          ic_launcher_round.png (xxhdpi)
          ic_launcher_round.png (xxxhdpi)
          ic_launcher_round.xml (anydpi-v26)
    ▼ values
        colors.xml
        strings.xml
        styles.xml
```

Load your project from last week. If you've deleted it then follow Labsheet 1 to recreate it.

We're going to be focusing on **activities, layouts** and **views** in this this labsheet, something that I introduced to you in the Week 2 lecture.

An **activity** is a **single, defined thing that a user can do.** These are written in Java.

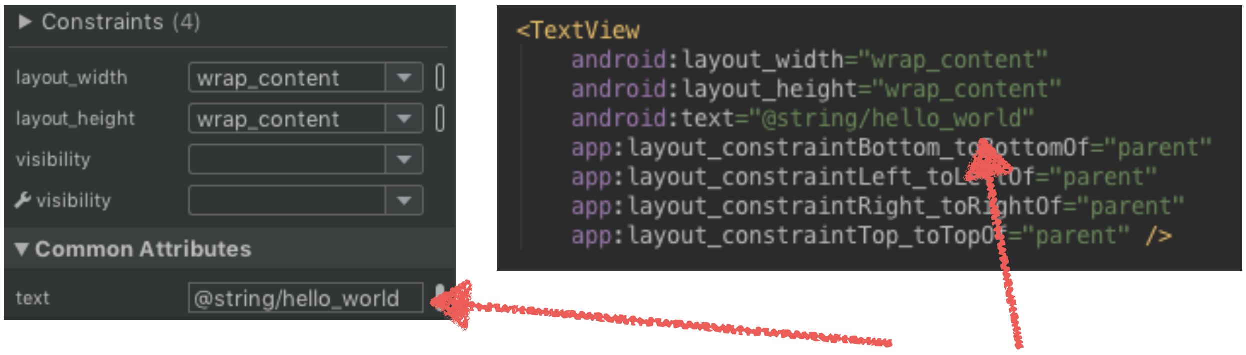
*A **layout** describes the **appearance of the screen.** These are written as XML files to denote arrangement of screen elements.*

There are a number of files in your project, which you can view in the project tree. See if you can find *MainActivity.java* and *activity_main.xml.* Open these. Note for the layout file, there are two view option: *Design* and *Text* view, as we discussed in the lecture. Explore both.

Refining your app

At the moment, your app says "Hello world!" in the middle of the screen. That's it. Let's make it more personal. Change the app to say "Hello <insert your name here>!" You can do this through the design editor (i.e., the GUI for designing interfaces), or the XML editor view for *activity_main.xml*. However, hard coding strings in this way is not recommended. It makes maintaining code harder (what if your name appears in 10 places in your application and then someone else takes over your code and wants to change the name?), but it also makes it more difficult to translate our app into a variety of different languages.

Let's change the Hello World text so rather than it being hard-coded, we keep it in a resources file that will make it easier to modify in future (and allow us to create translations). To do this, open *activity_main.xml* in replace your Hello World text with `@string/hello_world`. You can do this in design view or by editing the *activity_main.xml* file in the text editor.



```
▶ Constraints (4)
layout_width     wrap_content
layout_height    wrap_content
visibility
visibility
▼ Common Attributes
text             @string/hello_world
```

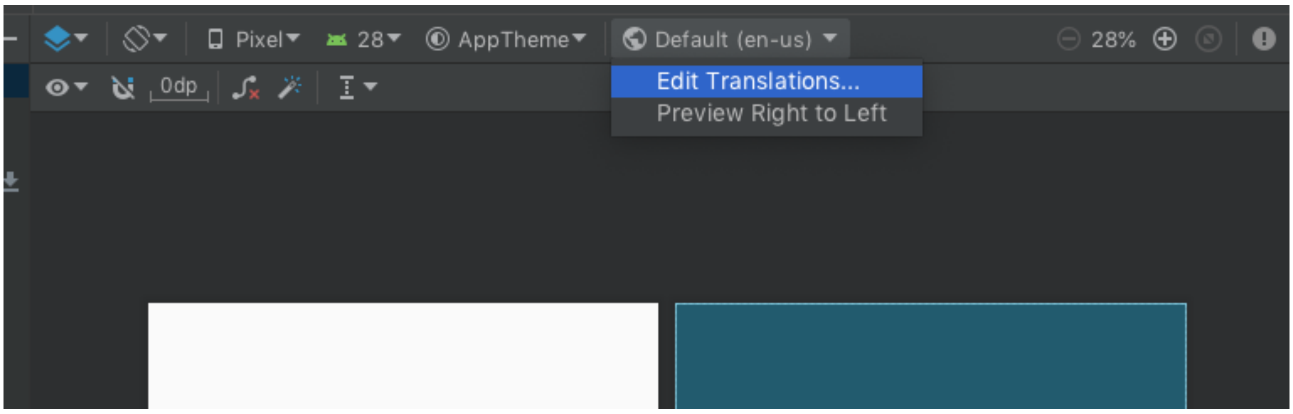
```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/hello_world"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
```

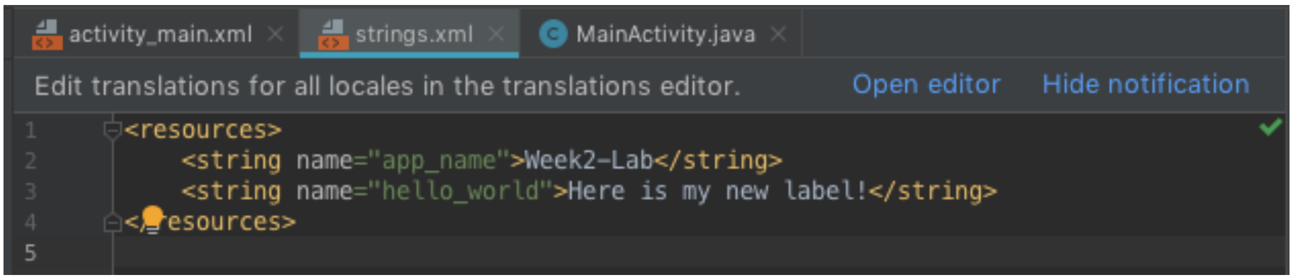
You can set the TextView text to `@string/hello_world` in the *Design* or *Text* editors.

This indicates that the app must look up a string value from a variable called `hello_world` in the strings resource file, s*trings.xml.* This is a resource file located in the *res* → *values* directory of your project tree (look at the image above. You will need to add a variable to this resource file named `hello_world` with the value you want this string to have, i.e.,

```
<resources>
    <string name="app_name">Week2-Lab</string>
    <string name="hello_world">Here is my new label!</string>
</resources>
```

We've improved the maintainability of our code already, so now let's do some translating. You can add this from the top of the Design interface by selecting the language drop-down and clicking *Edit Translations…* If you view *strings.xml* the editor might also offer you the option to a toolbar above the text.

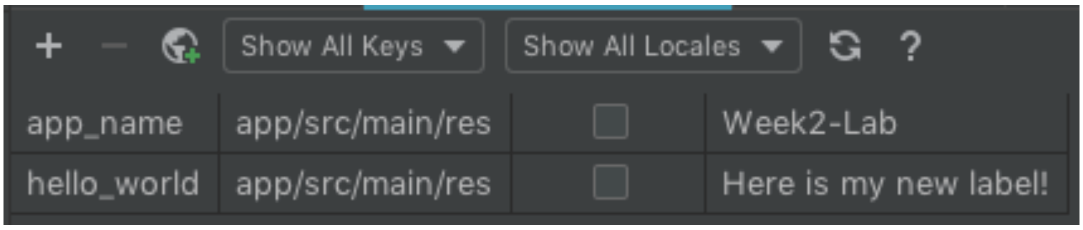




Edit translations for all locales in the translations editor.

```
<resources>
    <string name="app_name">Week2-Lab</string>
    <string name="hello_world">Here is my new label!</string>
</resources>
```

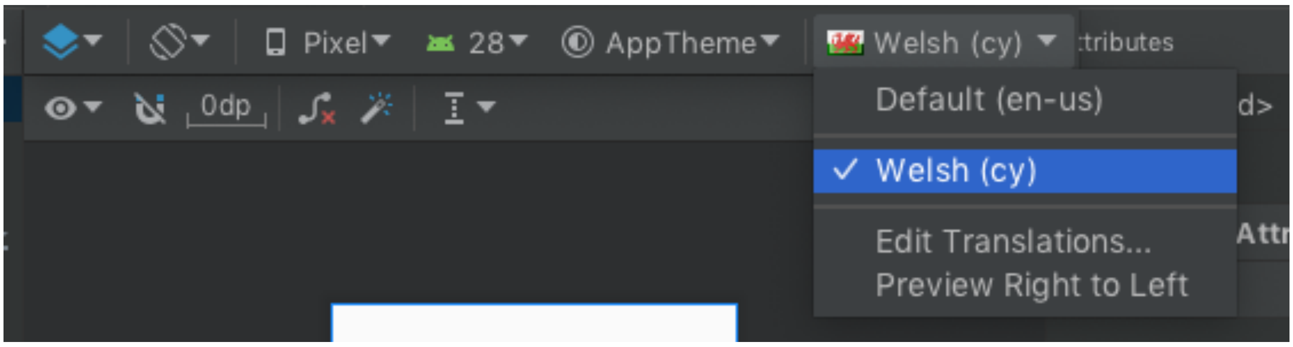
This Translations Editor shows us our current strings and offers us a way of adding another language by clicking the *Add Locale…* button.

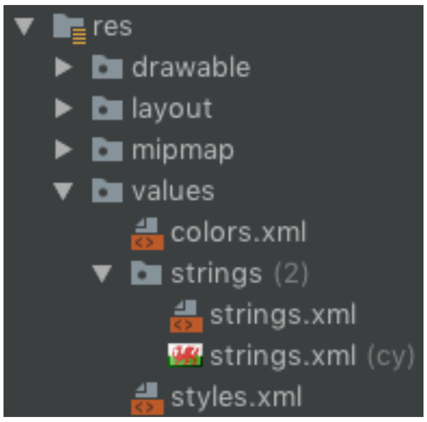


app_name	app/src/main/res	☐	Week2-Lab
hello_world	app/src/main/res	☐	Here is my new label!

Choose a language (preferably one that you understand) from the list. Another column should appear for you to provide a translation of the strings in each row. Add the translations or, if you don't know how to translate it, type in an alternative string in English and you can still observe the effects.

If you now go back to design view and click the languages menu again, you should now see the language you added as an option. If you select it, then your language. You should see the *TextView* text in the Design editor to match the language you have chosen.





Now you have a new language, your *strings.xml* file has migrated to a strings directory, which contains a strings.xml file for each of the languages your app uses. Editing these files allows direct input of translations. (Just as you might choose edit the original strings.)

Alternatively you can use the translation manager that you can access by selecting *Edit Translations* from that same *Language* drop-down in the design view.

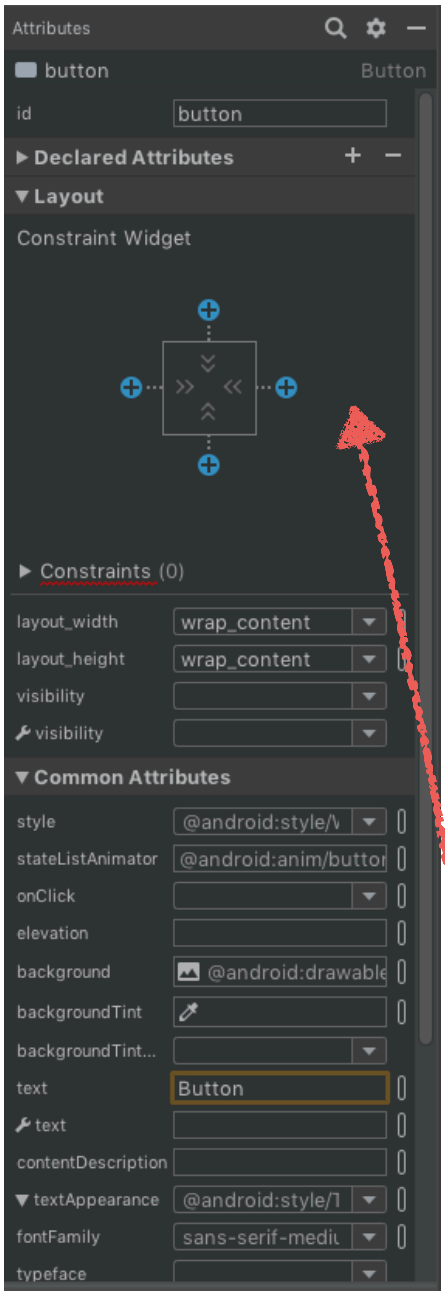
Once you've made the necessary changes, rerun the app, and marvel at the results! If there is more time, explore how to make more changes this further, for example by changing the colour of the text or the font used using the Design view attributes editor. Observe how this change made in the visual editor affects the content of the XML files!

Adding a button to your activity

So far, we've only looked at editing existing content from the template. Now we're going to look at adding additional components, or ***Views***, to our app.

There are two way of adding components to your app: you can use Text (i.e., XML) or Design editor. It's Important to get a feel for both. Dragging and dropping widgets is a good way to get started, but editing the XML is often essential for fine control.

First, let's add a button using the Design editor. There should be a palette containing GUI components on the left of the Design editor. Find, drag and drop a button onto the activity being displayed in the editor. Change the *text* of the of the button in the attributes pane so that it now displays 'Update' as its label. (You can use strings.xml or just hard code it, so long as you remember what best practice is for the future!)



Attributes

button — Button

id	button

▶ Declared Attributes

▼ Layout

Constraint Widget

▶ Constraints (0)

layout_width	wrap_content
layout_height	wrap_content
visibility	
visibility	

▼ Common Attributes

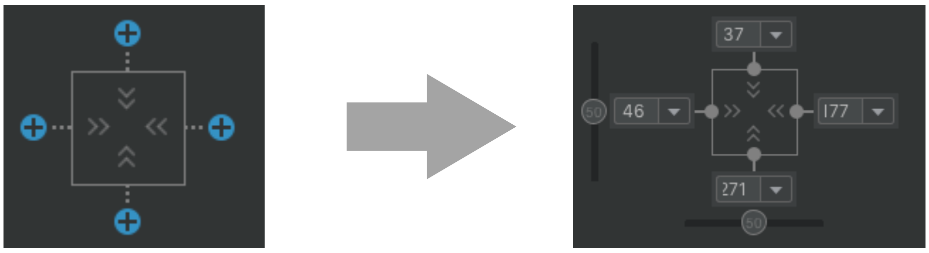
style	@android:style/W
stateListAnimator	@android:anim/buttor
onClick	
elevation	
background	@android:drawable
backgroundTint	
backgroundTint...	
text	Button
text	
contentDescription	
▼ textAppearance	@android:style/T
fontFamily	sans-serif-mediu
typeface	

It's important to note that changes in the design editor are reflected in the XML. You should see this if you switch to the *Text View*. Find the button you've just added in the XML. The Button has a few important properties, common to many, that you should note:

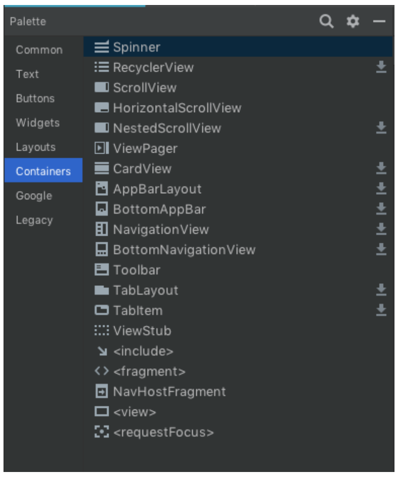
android:id This gives the component an identifying name, and importantly, enables you to control what components do via activity code, and also allows you to control where components are placed in the layout.

android:text specifies the text displayed

If you run the application as it is now, you will find that the button is not where you placed it, but instead will be in the top left hand corner. This is due to the *layout model* being used. We're not going to cover layouts in this labsheet, so for now all you need to know is that to get that button to appear exactly where you placed it, you will need to click the four buttons in the Constraint Widget in the Attributes sidebar.



Adding a spinner to your activity



Following the same method as for adding a Button, this time add a Spinner. You'll find this in the *Containers* subsection of the design Palette. Set the height and width as you want but don't forget to add the constraints in the Constraint Widget (re-read the previous section if you don't know what this means).

The next thing to do is get the spinner some content. In general we'll want the spinner to display a list of values. That we can select from.

We can do this in the same way that we have set text labels, by using adding a **resource** to *strings.xml*. We are going to use a `string-array` to contain the possible values for the spinner.

Add a **string array** inside the `<resources></resources>` elements in *strings.xml.*

```
<string-array name="spinneroptions">
    <item>Hello to the world!</item>
    <item>Everyone around the world - Hello!</item>
    <item>There are so many ways to say hello!</item>
</string-array>
```

Now we need to connect our *Spinner* to this `string-array` so that Android knows to populate the spinner with these values. In the Design view, select your spinner and look for the `entries` attribute in the list. Instead of something like `@string/string_name` as we have done previously, now we must use `@array/array_name` in this case, we need it to read `@array/spinneroptions`. It's also possible to do this by editing the XML to add an `android:entries` attribute to the spinner that references the string-array you've just created (i.e., `@array/spinneroptions`).

Getting your app to do something

You now have an **activity** with three **views** on it. Currently these views do nothing, they just sit there. We're going to finish this Labsheet by getting some actual interactivity into our app. It will work like this:

1. User chooses a type of 'Hello World' greeting from the spinner
2. User clicks the button
3. The contents of our TextView (i.e., label), will be updated with the currently selected item in the Spinner at the moment we clicked the Button.

To do this we must:

1. Adjust *activity_main.xml* to specify which method in the activity will get called when the button is clicked.
2. Create that method in *activity_main.java* and populate it with code to change the value of the TextView to the selected value of the Spinner.

Let's have a go at doing this. Select the button in the Design editor and in the onClick attribute enter the name of a method `onClickUpdate`. This method doesn't exist yet, so we need to go to *MainActivity.java* to create it. This will contain an "onCreate" method already underneath it, create a public method that returns a void value, called `onClickUpdate`, that takes a single argument, called `view`, whose type is `View`. All methods that respond to a button click must be public, have a void return type, and take a single `View` as a parameter. Ensure that `android.view.View` is imported too. (Android Studio will offer to do this for you.)

Now we need something for this method to do. First we need to get a reference from both the spinner and the text view components. This will allow us to retrieve the value of the chosen greeting from the spinner, and display the text in the TextView. Use the following code to start this process:

```
TextView currentgreeting = (TextView) findViewById(R.id.textView);
```

We're passing `R.id.textView` into the `findViewById`, which, recall from the lecture helps us to map activity resources defined in our XML activity file to our Java class file. The 'R' here is a special Java class that enables you to retrieve references to resources in your app.

We can follow a similar process to get a reference to the Spinner:

```
Spinner greetings = (Spinner) findViewById(R.id.spinner);
```

The `Spinner` object we have, `greetings`, has several class methods, including one called `getSelectedItem,` which returns an `object` with a `toString` method. The `TextView` object we have, `currentgreeting`, which has a class method called `setText`. You should be able to work out what the final line needed is to finish this off. If not, take a look at the sample code.

Extension task

So, the button now does something, so we have interactive behaviour in our app! Nice one! If you have finished and have plenty of time left, then have a go at adding more complexity to the `onClickUpdate` method that we have just created. Try changing the spinner items so that rather than containing the values themselves, they contain 'Greeting 1', 'Greeting 2', 'Greeting 3' etc.

Then change your `onClickUpdate` method so that based on the selected option it sets the *TextView* to a particular text. So that if someone selects ‘Greeting 2’ the text in the *TextView* becomes “Welcome to my very cool app.”. If they select ‘Greeting 3’ something else comes up. *Hint – you can do this with if/else statements or a switch statement.*