# First Android UI App

### Aims:

To introduce developing an App for Android

### Objectives:

* Getting started
* The Activity class
* findViewById

Review of Lecture Slides, and potential further reading

* How might you represent a Student record with XML elements and attributes?
* What is the role of setContentView?
* What is a widget and how might its ‘id’ be represented in XML?
* What is the role of findViewById?
* Explain the terms ‘anonymous class’ and ‘concrete realisation of an interface’. Why are these terms so important to understanding user interfaces and events in java / android?

## 1. Introduction

Using Android Studio we’ll develop our first Android Application with an interactive user interface and run it in an emulator.

## 2. Starting a new project

Let’s get started…

Open Android Studio and ‘Start a new Android Studio project’.

Choose an ‘Empty Activity’. Click [Next].

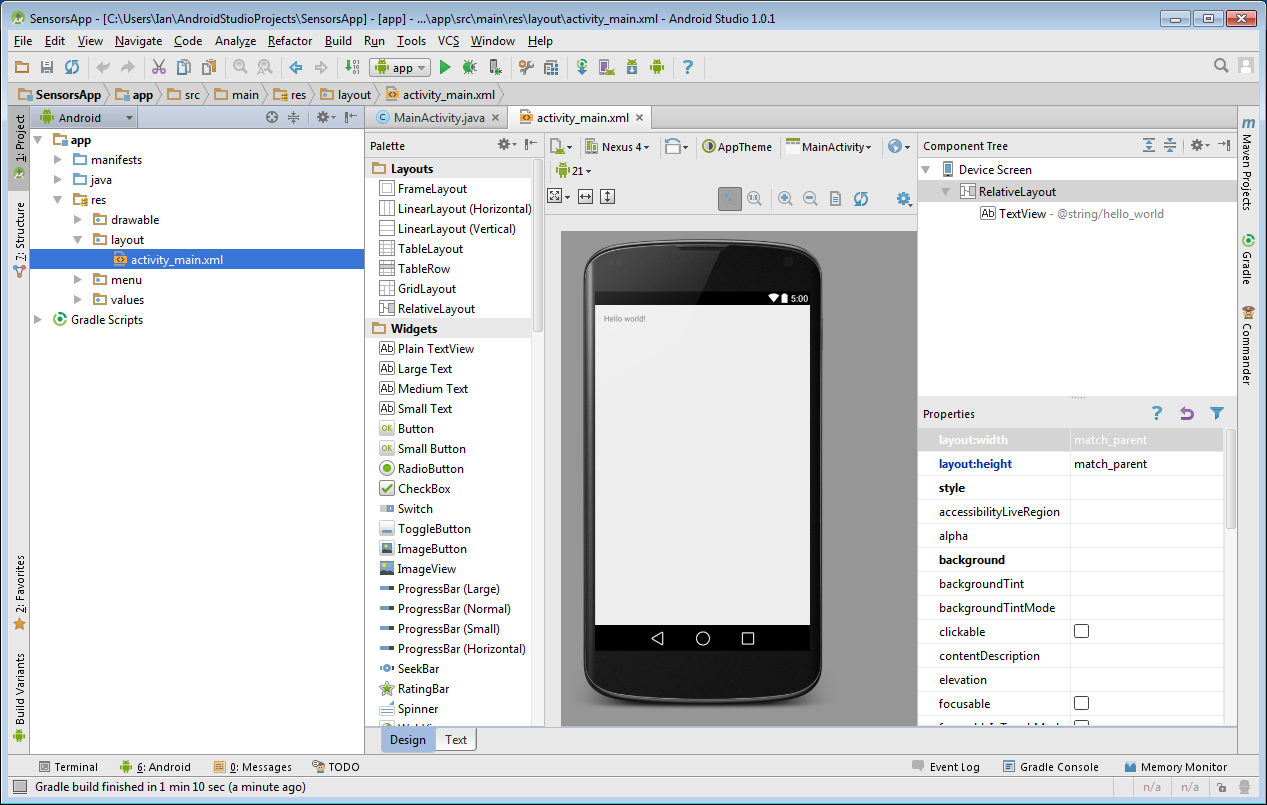
In the ‘Configure your new project’ dialog set your application

* name to ‘FirstUIApp’ or similar
* package name to com.your\_initials.firstuiapp
* save location to what works for you (e.g. F drive, one drive, or locally to be copied later)
* ‘Minimum API Level’ to for example to ‘API15:Android 4.0.3 (Ice Cream Sandwich).
* Language must be Java (Not Kotlin!)

Click [Finish].

Allow a minute or so to let Gradle complete executing tasks.

You will be presented with the Android Studio Integrated Development Environment.



Once working, in the resultant ‘Android ‘ pane to the left you’ll see the structure of your app by expanding the app directory’s view.

This includes

* manifests

Containing your AndroidManifest.xml file.

* java

Contains your package directory including your MainActvity.java source file.

* res

Contains directories for resources including the layout/activity\_main.xml file where you begin to define your user interface.

This default java/MainActivity.java source file extends ActionBarActivity class.

package com.example.ian.firstuiapp;

import android.support.v7.app.ActionBarActivity;

import android.os.Bundle;

public class MainActivity extends ActionBarActivity {

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

}

}

Save and Run the app. Note, you can do this from the main menu, keyboard shortcuts or the toolbar in your IDE.

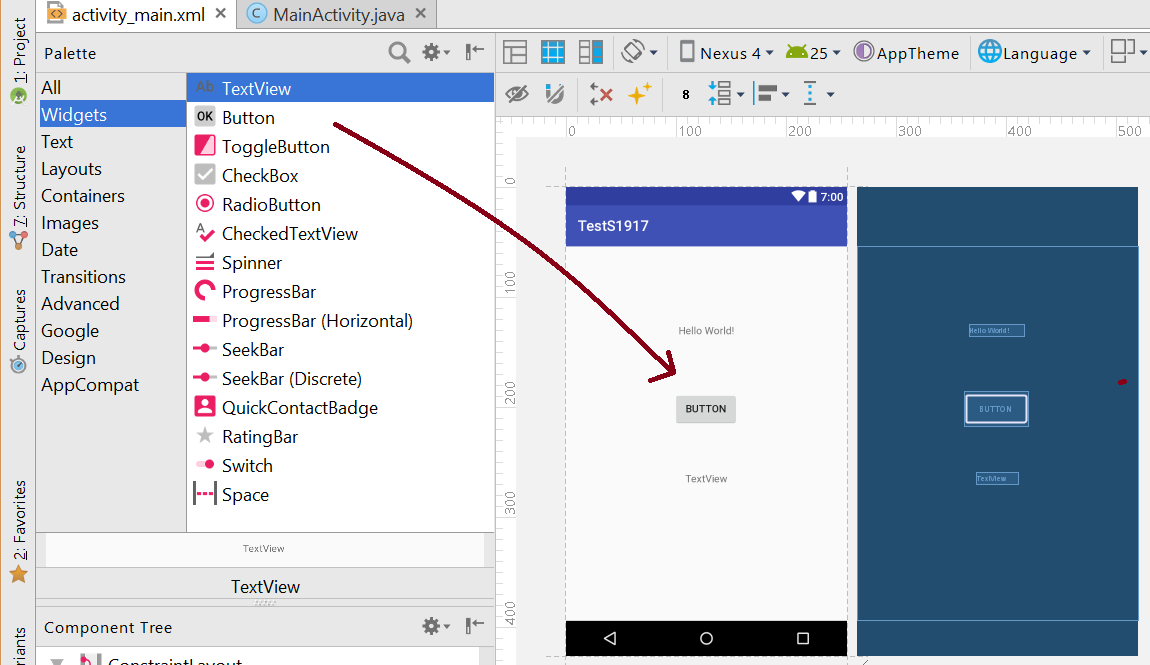
You’ll add to its OnCreate method some of your code to start creating your application.

You also have a default XML based resource file ‘activity\_main.xml’ where you create your user interface layout. The IDE provides a drag and drop style designer with which to specify user interface layout and contents.

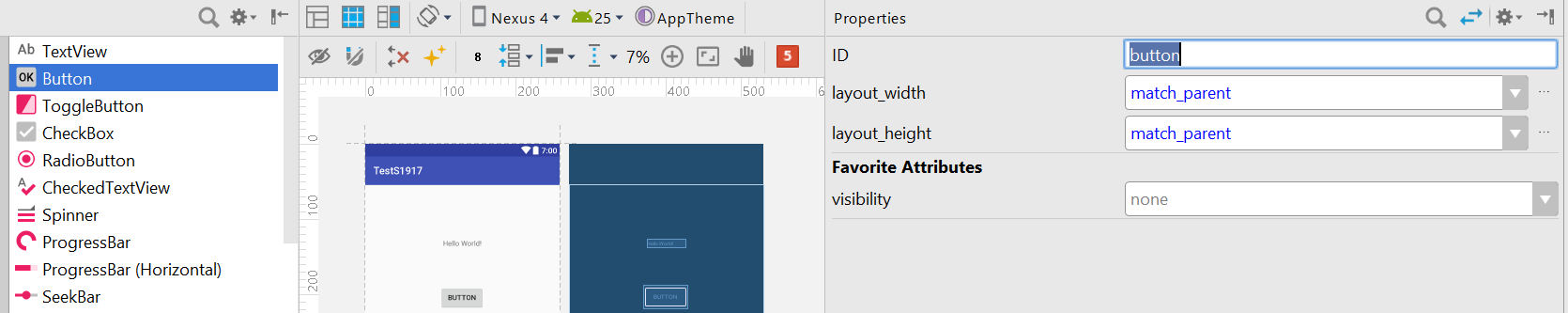
You will drag UI widgets such as buttons, text, images, etc to build the user interface and specify how they are constrained relative to each other. **Note, I will often refer to UI Widgets as ‘views’ because in code they are all subclasses of the ‘View’ class.**

Via the Editor or Package Explorer select res/layout/activity\_main.xml hence drag a TextView and a Button view onto the UI area. This UI area by default contains the parent view, in effect you drag views onto the parent view.

You can adjust a view’s properties such as its appearance and behaviour via its properties window (e.g. set Clickable true for Button).



Note from the Properties pane you can access and name the Ids of the views you added to the user interface. For example, give your views the names ‘button’ and ‘textView’ respectively, which you’ll use in code later.



Try running this application!

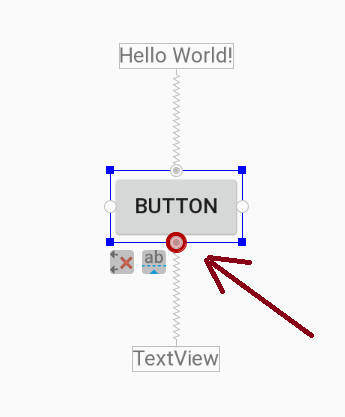


This will start or let you create an Android Virtual Device and upload FirstUIApp.apk to it, then install and run your app on this virtual device. Note the emulator is very slow to start, so try to keep it open when making changes to your app, or use a real device connected via USB.

If you were to run this application now you’d find you textView and button are in the top corner.

For a more useful user interface you need to utilise the **Constraint layout**.

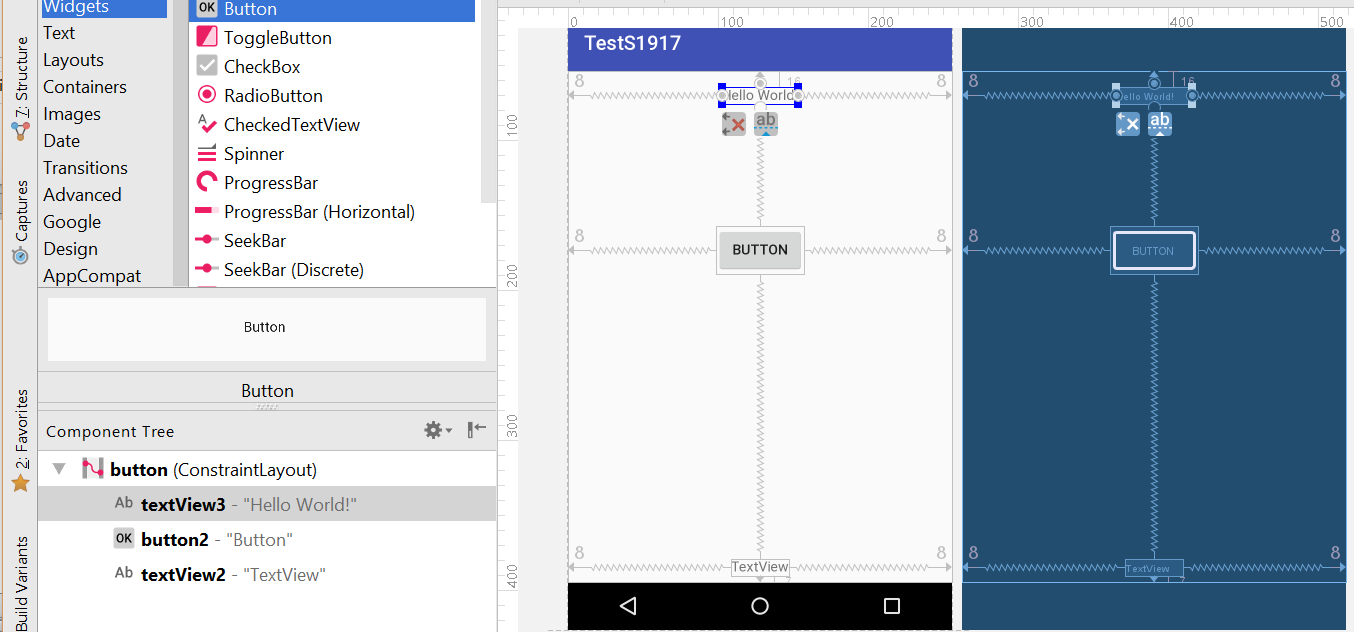
You can constrain the widgets you’ve just added relative to other widgets via the Design View.



When using ConstraintLayout, all views have four circles (called handles) top, bottom, left and right. You can click on and drag a handle from one view (e.g. button) to another view to add a constraint between them. For example, click on handles at the top and bottom of the Button widget, and attach to the surrounding parent. Constraints work like springs pulling views one way or another.

See in your own time <https://youtu.be/XamMbnzI5vE> and <https://developer.android.com/training/constraint-layout/index.html>

With practice you can position widgets relative to each other as you wish. Right clicking in the component tree gives access to more options. Alternatively you can edit the XML directly. Look at the XML text to see how the XML attributes work to constrain your views.



Try to build your user interface to look as follows. Running the app regularly helps you gain familiarity with the UI design tools.



To add interactivity, select MainActivity.java, and add to the MainActivity class the fields

TextView tv1;

Button b1;

You need to include import statements for these unresolved classes, to do so just hover the mouse over the class name underlined in red and let Android Studio suggest the import statement by clicking on a light bulb (if it appears). You can also **place cursor by the unresolved class name and hit Alt-Enter**.

The fields ‘tv1’ and ‘b1’ are references. You can bind these references to objects representing the TextView and Button you added to the user interface. To do this use ‘findViewById’. For example add the following to onCreate…

tv1 = (TextView) findViewById(R.id.*textView1*);

b1 = (Button) findViewById(R.id.*button1*);

Hence use the references to get or set the objects properties. For example add the following to onCreate…

tv1.setText("Hello, test application");

b1.setText("Click");

Next the button needs associated with it an event handler. You can instantiate a concrete realisation of the OnClickListener interface and associate this with the button via its setOnClickListener method. For example, add as a field of the MainActivity class

View.OnClickListener **b1Listener** = **new** View.OnClickListener() {  
 @Override  
 **public void** onClick(View v) {  
 **tv1**.setText(**"Button Clicked"**);  
 }  
};

*Note you can get Android Studio to generate much of the above for you. First type*

*OnClickListener b1Listener=* ***new*** *OnClickListener() { };*

*Then let Android studio suggest imports and implement method.*

Finally add the following to onCreate…

b1.setOnClickListener(b1Listener);

to associate your listener object with the button.

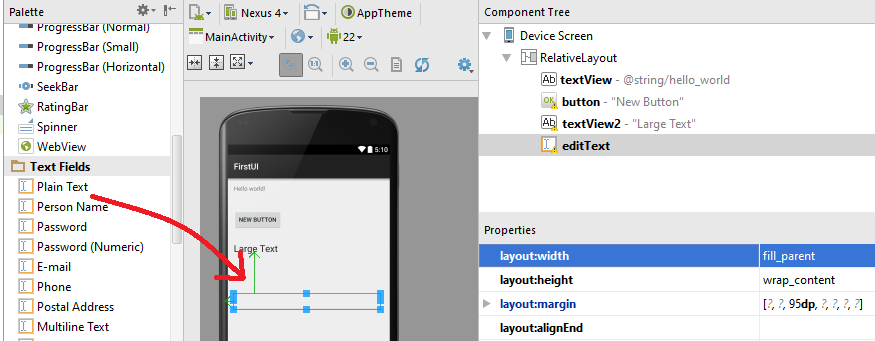
Your complete code should look like…

**package** com.example.firstuiapp;  
  
**import** androidx.appcompat.app.AppCompatActivity;  
  
**import** android.os.Bundle;  
**import** android.view.View;  
**import** android.widget.Button;  
**import** android.widget.TextView;  
  
**public class** MainActivity **extends** AppCompatActivity {  
 TextView **tv1**;  
 Button **b1**;  
  
 @Override  
 **protected void** onCreate(Bundle savedInstanceState) {  
 **super**.onCreate(savedInstanceState);  
 setContentView(R.layout.***activity\_main***);  
  
 **tv1** = (TextView)findViewById(R.id.***textView2***);  
 **b1** = (Button)findViewById(R.id.***button***);  
  
 **b1**.setOnClickListener(**b1Listener**);  
  
 }  
  
 View.OnClickListener **b1Listener** = **new** View.OnClickListener() {  
 @Override  
 **public void** onClick(View v) {  
 **tv1**.setText(**"Button Clicked"**);  
 }  
 };  
}

## EXERCISE

Try adding a text field to your app and display its contents when the button is clicked.

For example drag ‘Plain Text’ onto your user interface, and via its properties pane set the layoutWidth to fill\_parent.



In the MainActivity class create a field

EditText et;

In ‘OnCreate’ method bind the reference to the ui control

et= (EditText) findViewById(R.id.*editText1*);

In the ‘onClick’ button event handler you can use its value with for example…

tv1.setText(et.getText());

## EXERCISE

Create an app that converts popular measurement values (e.g. miles to kilometres, Fahrenheit to Celsius, etc). Google around for conversion formulae.

Note you can get the value of editable text with for example

**double** val = Double.*parseDouble*(et.getText().toString());

## Making Toast

A toast is a view containing a quick little message for the user. You can create and show a toast using the Toast class.

Toast.*makeText*(MainActivity.**this**, **"hello!"**, Toast.***LENGTH\_LONG***).show();

Where MainActivity.**this** is the package context, often we can’t use ‘this’ alone because our toast code may be within an anonymous inner class such as a button handler, and the ‘this’ would refer to the anonymous class.

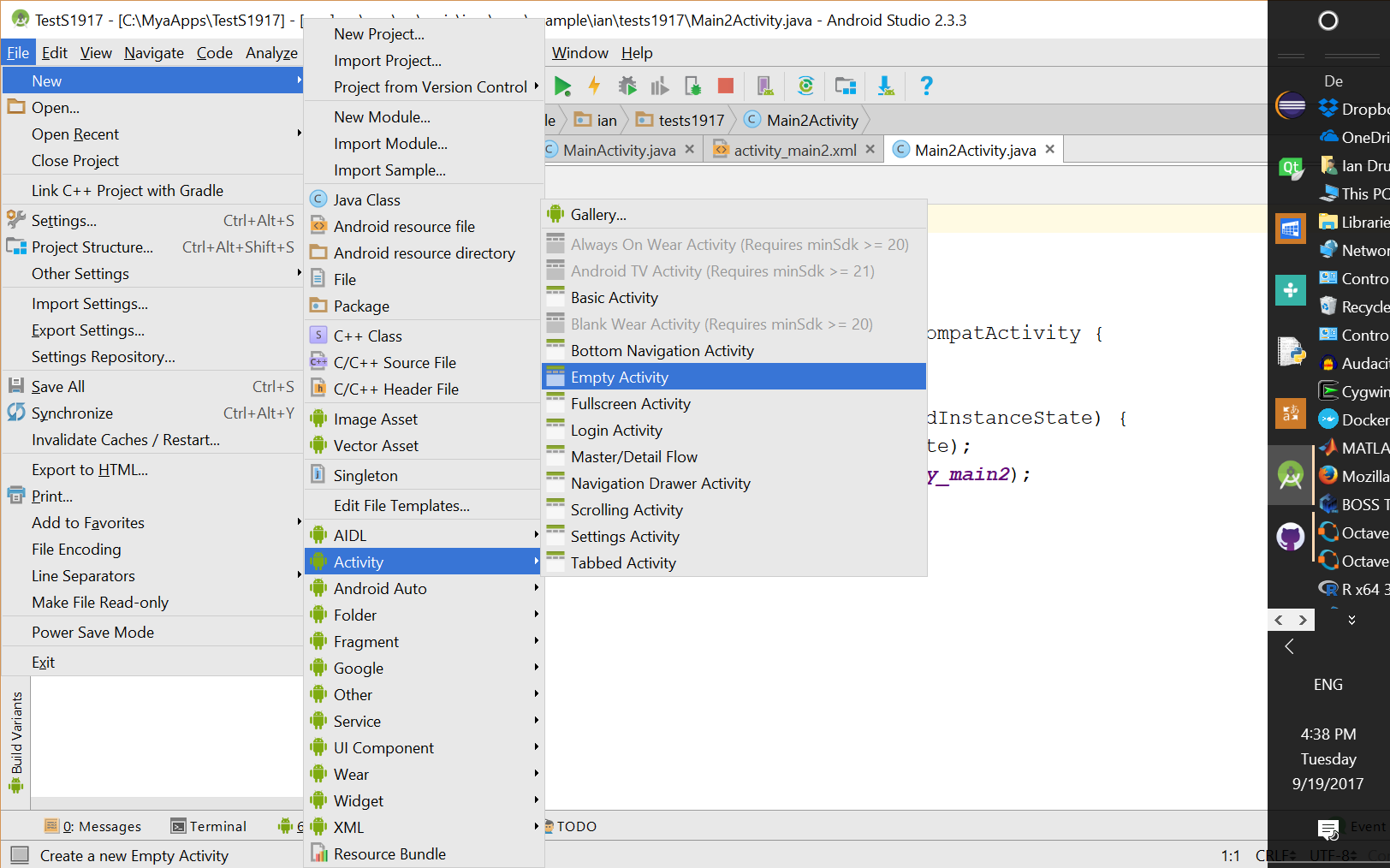
Try adding a Toast to your app.

## An app with multiple activities.

An app can have multiple activities that get pushed onto or popped of an activity stack. This is useful when you want to present several user interfaces. We will see the role of ‘intents’ in achieving this. An ‘intent’ is an abstract description of an operation to be performed, it can be used to start another activity or enable communication between multiple activities (and other services).

Create a new project and Empty activity, add to it a label and a button and define, instantiate and an event handler and associate it with the button.

Next create a new activity to add to the existing project via File->New->Activity->Empty Activity



A new java source file Main2Activty, and XML file activity\_main2.xml are created.

You can switch activities by creating an ‘intent’. For example in the button’s event handler ‘onClick’, declare and instantiate an intent, and call the startActivity method.

Intent intent1 = **new** Intent(MainActivity.**this**,Main2Activity.**class**);

startActivity(intent1);

Where MainActivity.**this** is the package context and Main2Activity.**class** is the context of the new activity.

You can escape the second activity, popping it off the activity stake by calling within Main2Activity the method

finish();

*Note if you instead call within Main2Activity something like*

Intent intent2 = **new** Intent(Main2Activity.**this**,MainActivity.**class**);

startActivity(intent2);

*You’ll just put more instances on the activity stack, which is not what you want.*

## Passing values between activities.

You can send ‘extras’ (such as strings, ints, doubles, arrays, etc) to pass values between activities.

For example within MainActivity edit code with additional lines in bold…

Intent intent1 = new Intent(MainActivity.this,Main2Activity.class);  
**int val=123;  
intent1.putExtra("MyVariableName",val);**  
startActivity(intent1);

And in Main2Activity add…

Bundle extras = getIntent().getExtras();

**if**(extras!=**null**)  
{  
 **int** val=extras.getInt(**"MyVariableName"**);  
 Toast.*makeText*(Main2Activity.**this**, **"val="**+val, Toast.***LENGTH\_LONG***).show();  
}

Try passing other variable types between activities.

## EXERCISE

Create an app with a starting screen that asks for the user’s name and age, then lets the user select with buttons a variety of user interfaces to do different tasks.

For example ...

* Create an activity asking for Gender and Height together with a [Submit] button. Hence it tells the user by name if they are tall, short or medium for their age.
* Create an activity asking for some lifestyle data (e.g. how many units of alcohol, cigarettes, takeaways, etc a week the user consumes). Hence it tells the user by name their life expectancy.
* Dream up your own additional activities to create.

## An alternative to event handling with anonymous inner classes

Instead of associating an [OnClickListener](http://developer.android.com/reference/android/view/View.OnClickListener.html) with the button in your activity, you can assign a method to your button in the XML layout, using the [android:onClick](http://developer.android.com/reference/android/R.attr.html#onClick) attribute. For example in res/layout//activity\_main.xml you could edit the button’s XML to include

android:onClick="sayHello"

The new XML for the button should look like

<Button  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:text="New Button"  
 android:id="@+id/button"  
 android:layout\_below="@+id/textView"  
 android:layout\_alignParentLeft="true"  
 android:layout\_alignParentStart="true"  
 android:layout\_marginTop="41dp"  
 **android:onClick="sayHello"** />

And define the sayHello() method in your button’s parent class, in this case FirstUIApp.

**public void** sayHello(View v)  
{  
 **tv1**.setText(**"hello from button"**);  
}

taking care to comment out

//b1.setOnClickListener(b1Listener);

Give it a try, though I recommend you instead use anonymous inner class definition for event handling. If you find the concept difficult I suggest you invest the time to learn it (it will be assessed) the implementation of anonymous inner classes is quite easy with a little practice.