Worksheet 1: Introduction

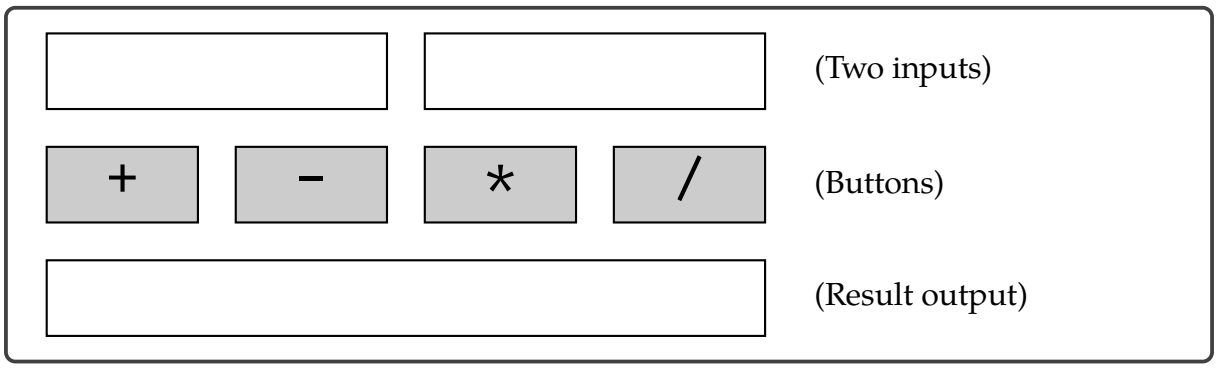
Updated: 1st August, 2018

This worksheet will give you an introduction to Android Studio and the stucture of Android projects.

Note: Some questions may occur up front:

- **Why Android and not iOS?** This is a matter of practicality. At the time of writing (to the best of the UC's knowledge) iOS development requires physical Mac hardware, not for technical reasons but legal ones. Android development can be done on virtually any modern PC, given enough RAM, disk space, etc.
- **Why the Android Studio (IDE) and not the command-line?** Previous units strongly emphasise the command-line, and it is possible to develop Android projects this way. However: (a) given your existing familiarity with the command-line, you should be able to figure it out from online documentation, if you want to; (b) We'd really like to explore more high-level concepts in this unit, and Android Studio simplifies things for us.
- **Can I install Android Studio on my own machine?** Most likely, yes. See developer.android.com/studio. Make sure to check the system requirements though!
- **Do I need an Android device?** No. All the prac work (and the assignment) can be tested using the Android emulator, which is installed along with Android Studio.
- **Can I use an Android device if I have one?** Probably. See developer.android.com/studio/debug/dev-options for instructions on enabling "developer options" on your phone/tablet. However, it's *still* probably better to use the emulator as a general-purpose testing platform.

To give you a gentle introduction to app design, we're going to make a four-function calculator that will end up looking vaguely like this:



(It doesn't have to be *exactly* this, but something functionally equivalent.)

1. Projects

Android Studio (like most IDEs) keeps track of whole "projects" at once, not just individual source code files. A project contains several source files, and other kinds of files, in a very specific directory structure. When you open Android Studio, it will give you several project-level options: start a new project, open an existing one, etc.

Note: There will almost certainly be changes to Android Studio over time. Don't be terribly surprised if the version of Android Studio you have does things *slightly* differently.

(a) **Start a new project.**

You will then be asked for:

- An application name; e.g. "Take My Money App".
- A company domain; e.g. "`curtin.edu.au`". (This helps ensure your source files are put into appropriate *packages*.)
- A project location; i.e. which directory to store all project files in.
- Various other things (where the defaults are probably fine).

(b) **Select a target device.**

Go with the defaults here, but it's important that you understand what's being asked. When developing software, you have to make complex choices about which platforms it will run on. We've already chosen Android, but there's more to it than that:

- First, note that there are a range of different kinds of devices that might run Android: phones, tablets, watches (and other wearable devices), TVs, cars, and possibly others. Different devices can obviously have different features. For now, we'll focus on just phones and tablets.
- Second, as the Android OS is updated, so too is the API (the "standard" classes and methods) that app developers have to work with. It routinely gets additional features that can either improve your productivity as an app developer, or let you do whole new things with your app.

  However, if you target a high (more recent) API version, you reduce your potential customer base. Only a tiny number of people, at any given point in time, will be using the most recent Android version (whatever it is). Many will be using a version that is years old.

  Android Studio has a "help me choose" option that shows, for each Android version, what percentage of all Android users are using that version or higher. It also shows what features are available in each version.

(c) **Add an activity.**

Activities are the first thing you'll learn about the structure of an Android app itself. An app can have one or more activities [1], and each is associated with a

[1] Or technically *zero* activities, if it's purely a service that runs in the background.

different UI screen.

To start with, we'll add one. Android Studio tries to help by giving us a lot of different options. These are basically different code templates; the IDE will give you a small amount of code as a starting point. However, "crutches" like this aren't necessarily helpful in actually learning how things work.

So, go with the simplest one: "Empty Activity".

(d) **Configure activity.**

Here you'll be asked for:

- The Activity Name – the name of the class representing the activity.
- The Layout Name – the (partial) name of the file representing the UI (the visual front-end to the activity).

There's also a checkbox marked "Backwards Compatibility (AppCompat)". On balance, this is generally a good thing. AppCompat is one of several "support libraries". These give app developers access to an up-to-date API on devices that do not directly support it. The tradeoff is an increase in the installed size of your app; it takes up more memory because the support libraries are included in the installation.

(e) **Finish (and initial build).**

Once you finish these steps, Android Studio will initialise the new project. This could take a while (at least the first time), since it may need to download a few things.

You may notice some messages about "Gradle". Gradle is the build tool, comparable to an advanced version of Ant (if you've used that), or an extremely advanced version of Make. Bascially, it's the underlying command-line tool that Android Studio uses (mostly in the background) to compile, test and package your code into an app.

If everything (eventually) works, you should get a "Gradle build finished" message at the bottom of the window.

Gradle troubleshooting: If anything goes wrong at this point, you could be staring at an error message involving Gradle (e.g. "`Gradle sync failed`"), because Gradle is on the "front-line" of the battle, and will be the first to encounter most errors.

This typically happens when there's something missing, especially the Android SDK or parts thereof. If so, you can probably solve the issue by clicking on whatever "install" or "update" options are presented. You may need to do this multiple times, but hopefully only the first time you run Android Studio.

> **The Android SDK:** The Android Software Development Kit (SDK) is the set of tools and libraries needed to compile, test and package an Android project (apart from Android Studio and Gradle, which themselves aren't *technically* necessary).

(f) **Run.**

You should now see the proper IDE editor and a few lines of auto-generated code. But before we do anything with it, we're going to check out the emulator.

Click on the toolbar button marked with a green triangle, representing "run". (IDEs will automatically re-compile anything that needs re-compiling before running, because they're nice like that.)

You must select a "deployment target"; i.e. a device to run the app on. You should see a list of "Connected devices" (probably empty), and a list of "Virtual devices" (hopefully not empty). This is where the emulator comes in.

(If the virtual device list *is* empty, you should be able to create a new one. You just need to select a device to mimick (e.g. "Pixel XL"), and accept the defaults for the other settings.)

Select one of the virtual devices and click "OK". The emulator should start up and load the app equivalent of "Hello World".

> **The Emulator:** The virtual device has a complete Android OS installation, including all the standard Google apps. At a software level, it has all the same capabilities as a real Android device, though these are manipulated through software controls rather than connected to actual hardware capabilities.
>
> Your own app (when you run it) will be uploaded to the device, overwriting any previous version.
>
> Having *multiple* virtual devices allows lets you test your app across a variety of screen sizes and API versions, and it's generally a more controlled testing environment than if you use your own physical device.

2. The Activity

You should see (approximately) this in the IDE's editor:

```
package com.example.demo.myapplication;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;

public class MainActivity extends AppCompatActivity
{
    @Override
    protected void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
}
```

We're only going to take a brief look for now, just to understand what's going on.

`MainActivity` (or whatever you called it) is the initial Java code behind the newly-created app. Activities must extend the standard `Activity` class, but in this case we're extending `AppCompatActivity`, which itself extends `Activity`. (Recall that we chose to use the "AppCompat" library to help our app run on older devices.)

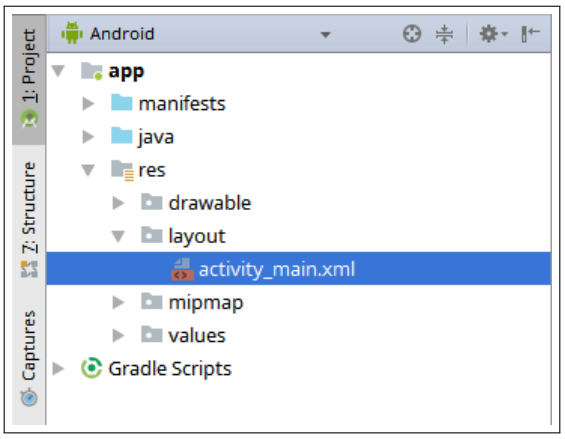
When the user launches the app, Android will first create an instance of `MainActivity`, and then call `onCreate()`. There's a lot of stuff going on, behind the scenes, before *our* code actually starts running.

Then we get to "`super.onCreate(savedInstanceState)`". We want to make sure that the superclass's version of this method gets to do its work (which could involve some important setting up).

Finally, we do "`setContentView(R.layout.activity_main)`". Recall that an activity is connected to a UI screen. This is how that connection is made. The term "`activity_main`" was the name we chose earlier for that UI. Did you notice that the words "Hello World" appeared when you ran the app, but they *don't* appear anywhere in the Java code?

3. The Layout File (UI)

There's another file lurking in the project called `activity_main.xml` (or whatever you chose as the "Layout Name" earlier). You can find this in the list of project files:



This is where you design the user interface itself, and it should look like (roughly) what you saw when you ran the app (but with additional markings). In particular, you should now be able to see "Hello World".

Here is where you can drag-and-drop UI elements (buttons, text fields, etc.) onto the screen to build up your app's UI. *However,* before you do anything else, find the two tabs that say "Design" and "Text", and select "Text". Now you should see something approximately like this:

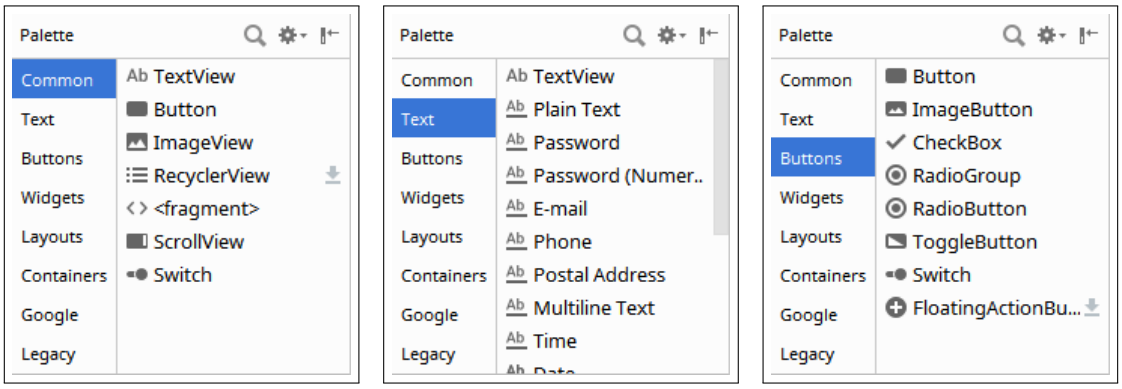
```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
  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

    <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" />

</android.support.constraint.ConstraintLayout>
```

This is an XML file (see Appendix), and it stores all the UI information. It's what the UI design really is, underneath. Any and all changes made to the graphical drag-and-drop view will show up here.

Now we'll build the actual calculator UI!

(a) **Add the UI elements.**

Switch back to the drag-and-drop ("Design") interface, and see if you can add the right set of elements to make a four-function calculator. Drag them from the *palette* (below) onto the mock mobile screen.



Keep the existing "TextView" element (for the result) and add two new "Plain Text" elements, and four "Button" elements. It doesn't have to look pretty, as long as they're all there.

(b) **Set the right attributes for each element.**

When you select any of the UI elements, you should be able to see and edit its *attributes*. Most of these will look quite esoteric for the moment, but a few are important:

ID – this is a bit like a field or method name. It's a name that you will use in the code to access each element. The IDE will pick default names (like "`editText2`", etc.), but you should replace these with more meaningful names, just as if you were naming fields or methods.

inputType – this exists for editable elements, and tells Android what sort of data is being edited. We *could* just accept plain text and then validate it outselves, but if Android knows that we want numerical data (for instance) then it can validate it for us. If you click on the "···" icon, you should see a long list of different input formats. Select both "numberSigned" and "numberDecimal" (for the two input elements).

text – for buttons, this is their visible label, so change it to "+", "−", "×" and "÷" as appropriate. For the input/output elements, delete the existing text (if any), since we'd like those elements to be blank initially.

onClick – for buttons, this is the name of a method in the activity to be called when the button is pressed. This is *one* way to set up interaction between the UI and the activity. Don't change this yet, because we'll explore a different approach below.

(c) Have a brief look back at the XML to see what's changed. You should now see several `<EditText ... />` and `<Button .../>` elements, and all the UI attributes you just changed should be reflected as XML attributes (e.g. `android:text="+"`).

4. Implementing Callbacks

Return to the Java code (`MainActivity.java`, or whatever you called it). We're going to add most of our code immediately *underneath* this line (which should already be there):

```
setContentView(R.layout.activity_main);
```

We inherits this method (and others) from the superclass. We said earlier that it connects the activity to the UI, but what it *really* does is cause the entire UI to be instantiated. The layout XML file is read, and each `<EditText ... />` element (for instance) becomes an `EditText` *object* in memory, which can display the appropriate representation of itself on the screen. (`EditText`, `TextView`, `Button`, etc. are all standard Android classes).

With that in mind, in order to finish the calculator, we need to do the following:

(a) **Get references to the newly-created GUI objects.**

To do this, we rely on another inherited method, as follows:

```
private EditText operand1;

...                          Inherited method
operand1 = (EditText) findViewById(R.id.operand1);
                                          The ID attribute
```

Do this for each UI element, substituting the names and IDs as appropriate.

> **Note:** Use fields to hold references to UI elements; they go at the top of the class as usual. (Sometimes a local variable might suffice instead, but using fields is simpler and sort-of conventional.)

> **Note:** We need the `(EditText)` downcast because, `findViewById()` can in principle return any `View` object. We must tell the compiler that we know it's actually `EditText`, a subclass of `View`.

(b) **Register and receive button events.**

Creating the callback typically looks like this:

```
// Where 'plusButton' is the ID you assigned to that "+" UI button.
plusButton.setOnClickListener(new View.OnClickListener()
{
    @Override
    public void onClick(View v)
    {
        ... // Do something when the button is pressed.
    }
});
```

For those unfamiliar with *anonymous classes*, this is one. It's just a short-hand notation for:

- Creating a new subclass of the `View.OnClickListener` interface,
- Making an object of it, and
- In this case, passing that object to `setOnClickListener()`.

When the button is actually pressed, it will invoke the `onClick()` method, since that's now been registered.

Repeat this for the other three buttons too.

(c) **Access the input (`EditText`) objects.**

Although our two `EditText` objects are constrained to take numbers only, they're still fundamentally text-based, and they will give us Strings. And they'll also do it a bit indirectly, so we need something like this:

```
Editable editable = operand1.getText();
String str = editable.toString();
double number = Double.parseDouble(str);
```

This ought to be merely a one-liner of course. I've just expanded it to make it clearer what's going on.

"`Editable`": GUI elements are often represented by *two* classes: one to control the actual inputting and outputting (e.g. `EditText`, and one to hold the data while it's being edited (a subclass of the `Editable` interface, in this case).

This may seem unnecessarily convoluted, but it's done for reasons of flexibility and separation-of-concerns. Android's UI is effectively the culmination of decades of UI expertise, and while a lot of it is going to seem convoluted at first, there's virtually always a good reason for the way things have been designed.

(d) **Modify the output (`TextView`) object.**

`TextView` is just about the simplest of all GUI elements. It doesn't really do anything other than display text, but that's all we need it for.

Now that all the other pieces have fallen into place, we can set the output text as follows:

```
double result = ...;
resultBox.setText(Double.toString(result));
```

You should be able to complete the functionality from here. Then you just need to run and test!

Appendix

Extra notes on XML: The eXtensible Markup Language (XML) is a way of representing hierarchical data. It's extremely similar to HTML (but not exactly the same). XML is a foundation on which other languages can be based; in this case, Android has a particular XML-based language for describing user interface designs.

Some basic XML concepts (most of which also apply to HTML):

- "**<xyz>**" is a *tag*.
- "**<xyz>** … **</xyz>**" is an *element* (defined by opening and closing tags). Elements are the building blocks of the data, and can contain a mixture of other elements and (sometimes) free-form text.
- "**<xyz />**" is a shortcut for "<xyz></xyz>"; i.e. an empty element.
- "**key="value"**" is an *attribute* (when placed inside an opening tag). This is another way to attach particular information to an element.
- "**ns:name**" is a name with a *namespace* of "ns" (when used as element or attribute name). Elements and attributes can be grouped into namespaces for organisational reasons.
- "**<?xml** … **?>**" is the XML declaration, containing metadata that helps decode the remainder of the document).
- Not shown above, we often need *character entities* to represent certain special characters. Most notably:

  "**&**" is a literal ampersand (&);
  "**<**" is a literal less-than sign (<);
  "**>**" is a literal greater-than sign (>);
  "**'**" is a literal single-quote (');
  "**"**" is a literal double-quote (").

- Not shown above, "**<!--** … **-->**" is a comment.

Resources and R.java:

As part of the build process, immediately *before* compilation, Gradle will auto-generate a source file called R.java [a].

Normally you don't need to actually see this file, but it's good to have a brief look at it just so you know. It can be a bit tricky to find, but the following Unix command should report its location, when run from within the project directory:

```
[user@pc]$ find -name R.java -not -path '**/android/**'
```

The class is basically just a gigantic list of integer constants (and nested classes that are also gigantic lists of integer constants).

Its purpose is to tie together the Java and XML code, and other resources. For instance, when you write "`R.layout.activity_main`" in Java, this evaluates to an auto-generated integer ID that represents the `activity_main.xml` file. Everything in the `res/` directory is a *resource*, and, when needed, you refer to it by the appropriate constant from the `R` class. The build process takes care of creating (and interpreting) these constants for you.

[a] This isn't inherently what Gradle does; it's just being told to do this by the build script that it's working from.

End of Worksheet