Lesson 0 - Your First Android App

Introduction (Read this first)

You will complete this lesson outside of lesson time.

The instructions given in this lesson will show you how to do a “hello world” Android app. Assuming that you have a fresh installation of Android Studio, a significant amount of time is consumed in downloading components from the internet.

You should allocate 45 mins to 1 hour, together with a fast and cheap (hopefully free) internet connection for this task. Some downloads may take some time, so plan something else to do during that period.

Building android apps is computationally intensive. Your laptop can get warm very quickly and could possibly overheat (which I experienced). Do take the appropriate steps.

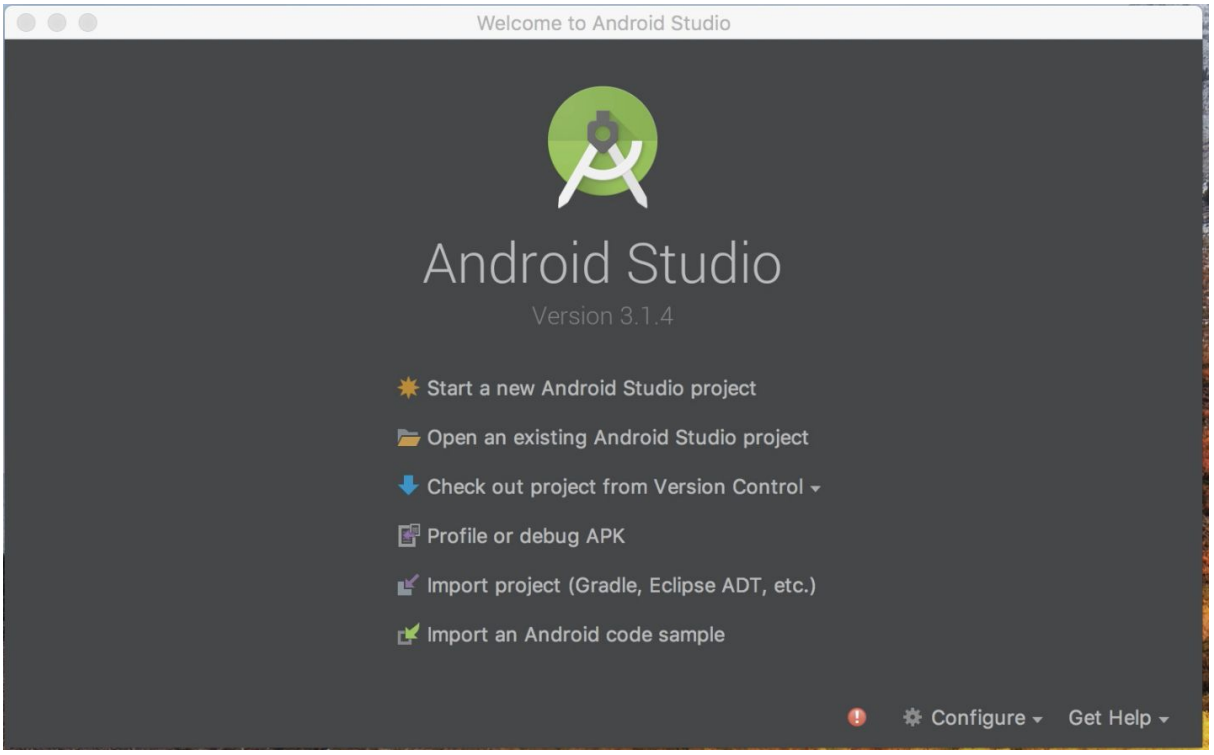
This tutorial was written with a freshly installed Android Studio in a new MacBook. Hence, what you see on your computer could be different, just respond accordingly to the instructions on the user interface.

Try to get an android phone if you do not have one. You may find that testing your app on a physical android mobile phone is much faster than using the emulator.

Create a “hello world” app

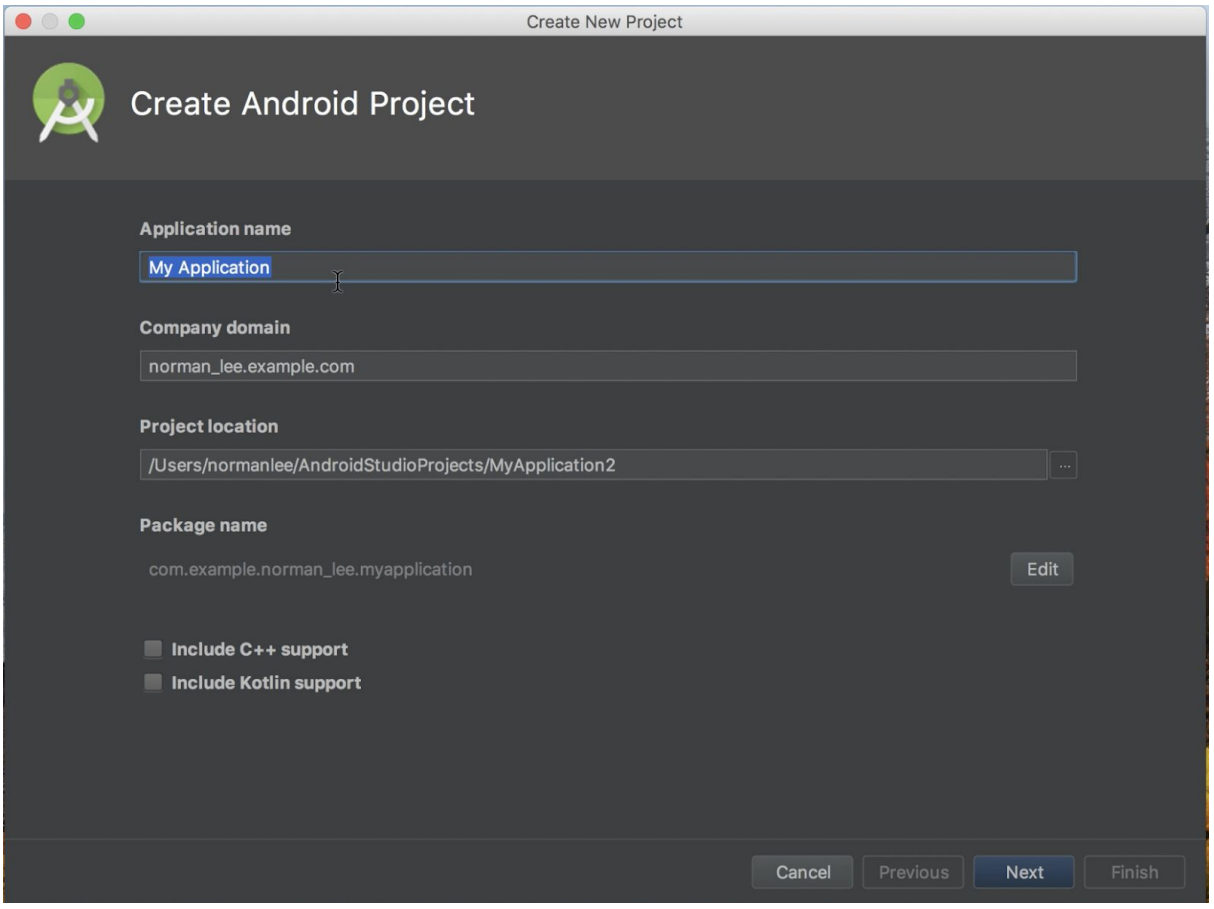
Start a new android studio project

Launch Android Studio from your computer and select **Start A New Android Studio Project**



Give your project a name

Give your project an **Application name** and state where you want it to be saved in the **Project location**. For starters, you may accept the defaults and carry on.

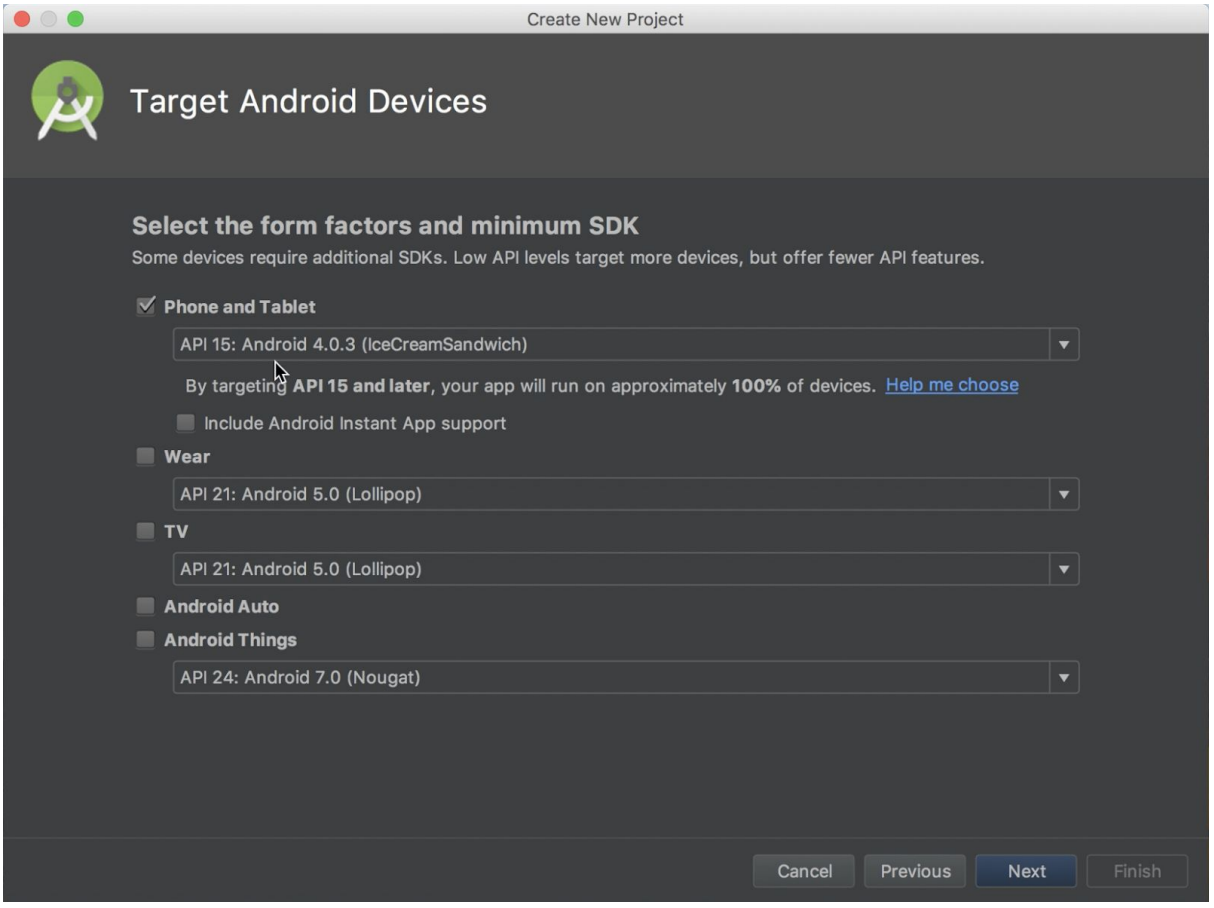


State what devices you want your android app to run on

Ensure **Phone and Tablet** is ticked. You will next have to decide the minimum **API Level** that you want to deploy your android app on.

The **API level** is the version of the android operating system. The lower the API level, the older the version, but the more devices that your app can run on. Explore the options. (You can read more about it here: https://developer.android.com/about/dashboards/)

For starters, accept the defaults.



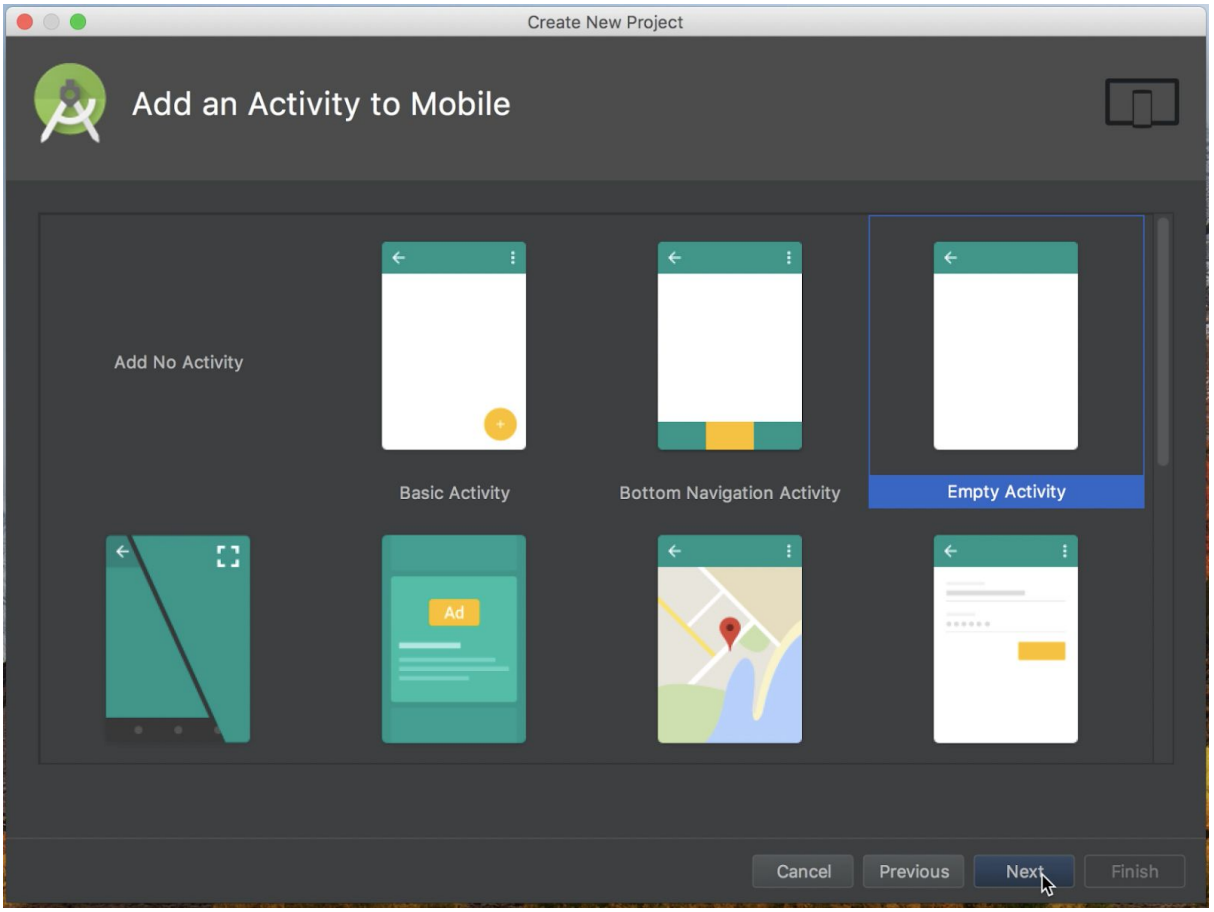
Start with an Activity

You will now have to add an **Activity**. The Android documentation defines an **Activity** as:

An activity is a single, focused thing that the user can do

In other words, it is a screen where the user interface (UI) resides on, for the user to interact with.

Make sure **Empty Activity** is selected, as it is the option with the least amount of code. The other options come pre-loaded with some android app features, and they are useful when you are more familiar with android app programming.



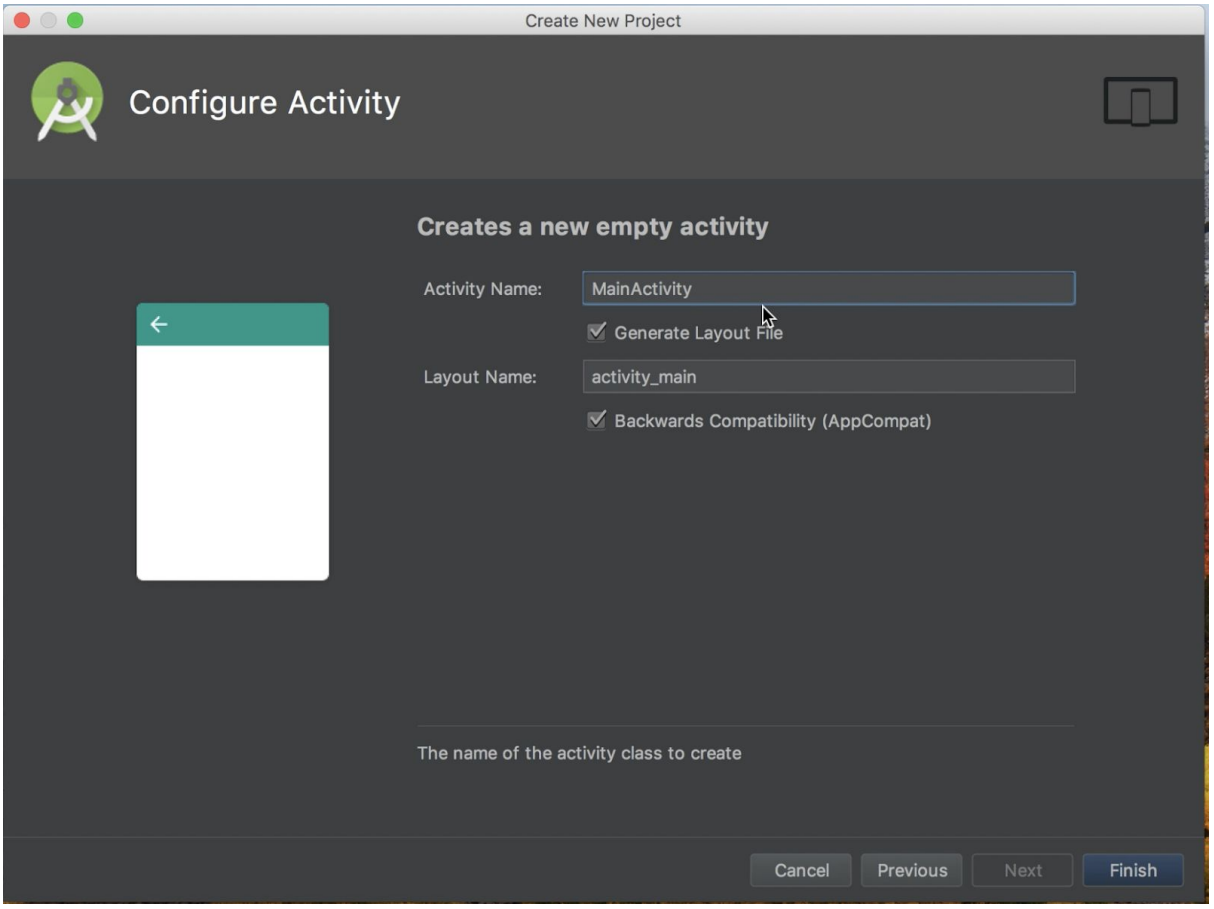
Name your activity

Your android app project will typically have many activities, and the entry point (i.e. the first activity that your user will see) is called **MainActivity**. Hence, just accept the defaults.

This creates two files:

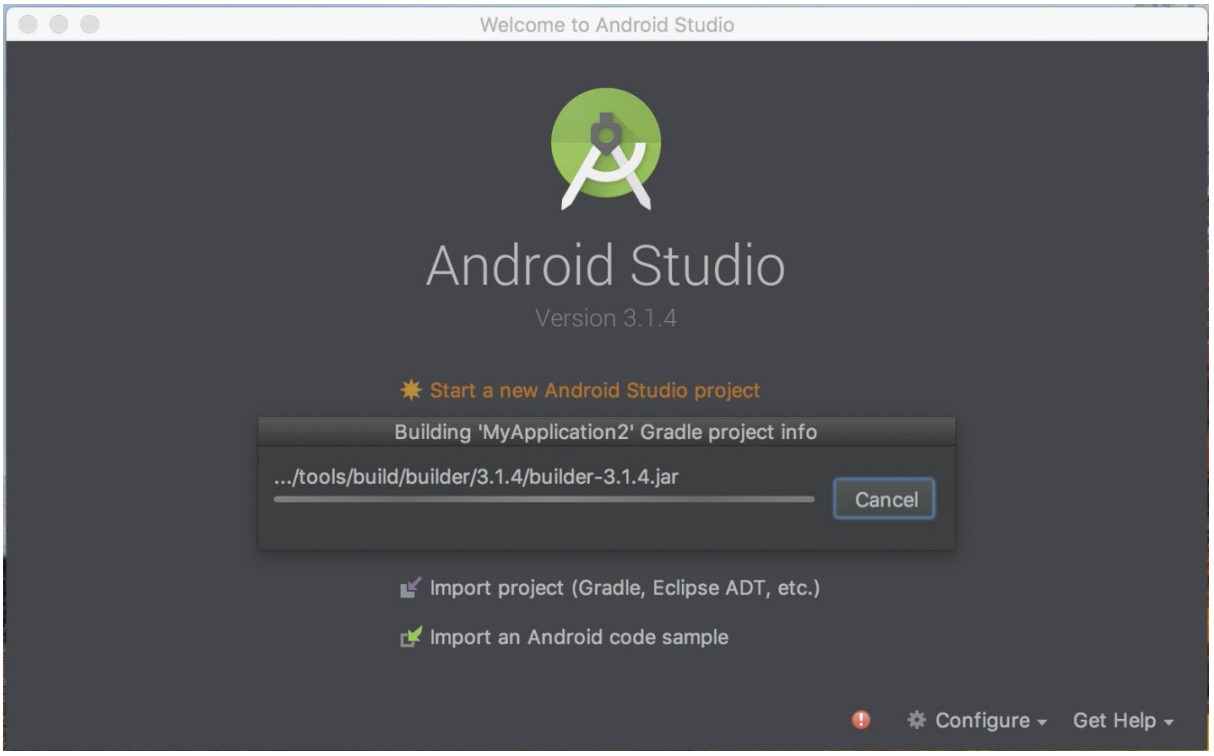
- An xml file named `activity_main.xml`
- A java class in a file named `MainActivity.java`

The layout of the UI is specified in the XML file and the logic (e.g. what happens when a button is pressed) is coded in the java file.



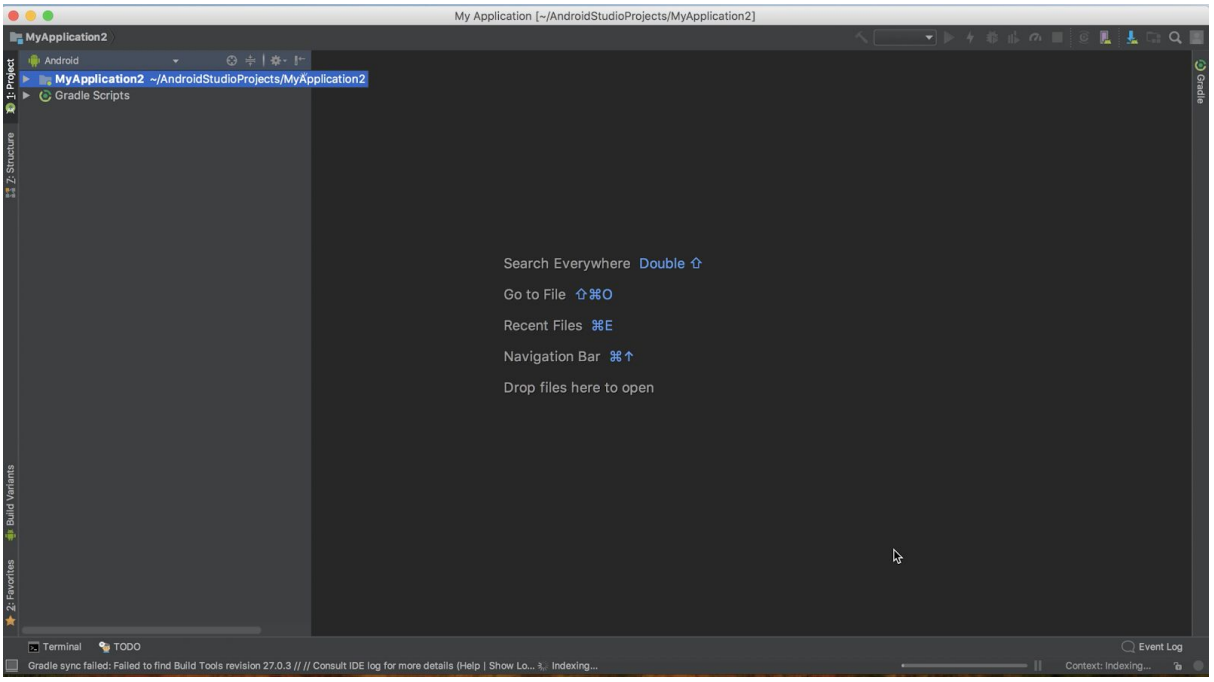
Wait for a while and install some stuff

While android studio is getting your project ready, the screen should look like this.

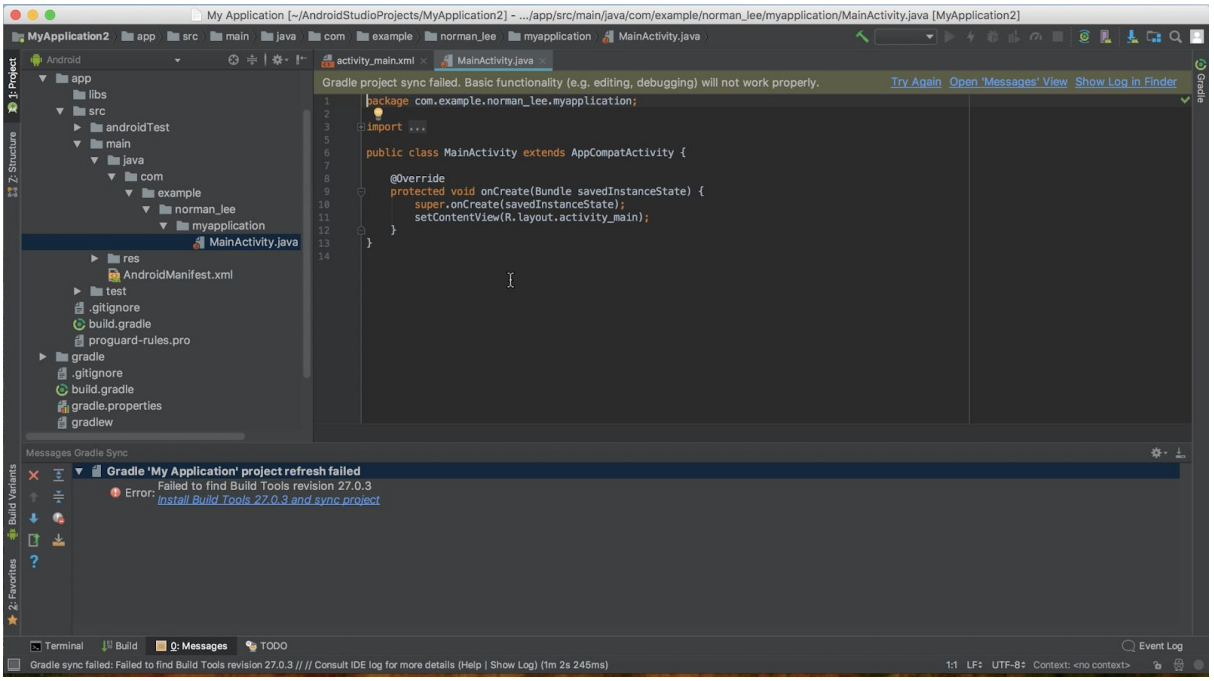


When the installation is done the screen should look like this.

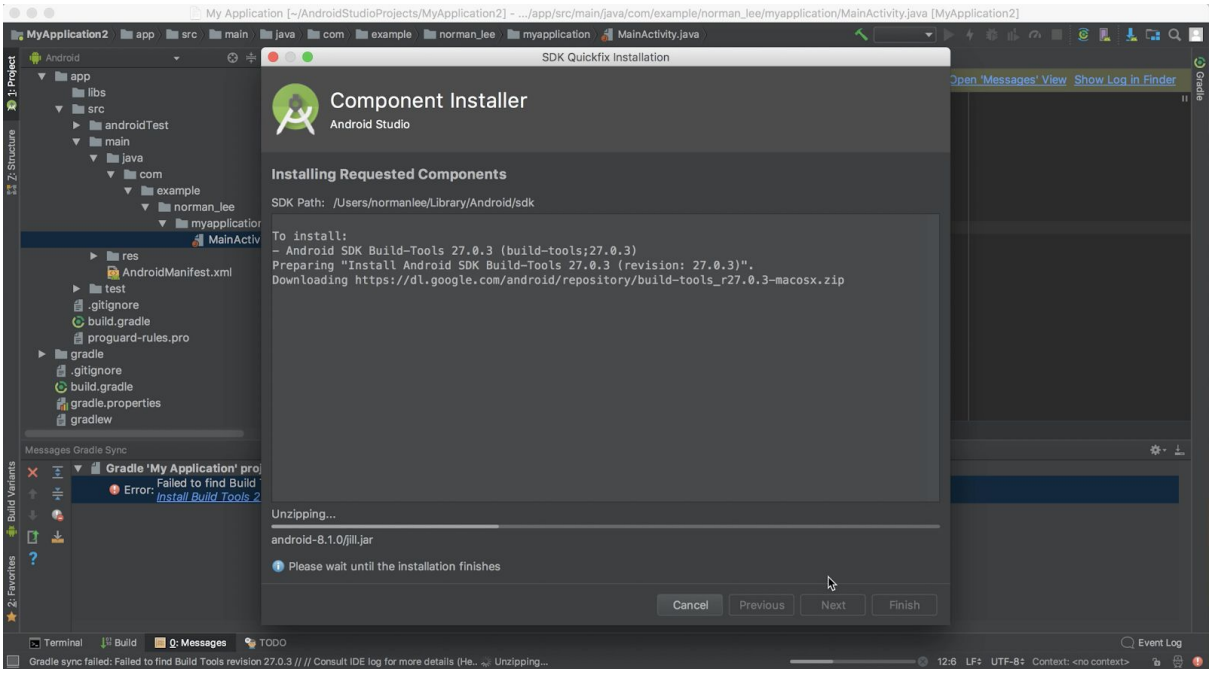
Click on the arrow beside **MyApplication** to expand the folder tree.



After expanding the folder tree, you will see some messages. At this stage, Android Studio does not have all the components necessary to build your app. Click on **Install Build Tools** and wait for a while.



After you click **Install Build Tools**, you will see this display. Click **Finish** when the download is completed.



Examine the folder structure

You'll notice that the folder tree now looks different and will look like the image below. This is the correct folder tree to see.

You can select different views (red box below). What you should select is the **Android** view.

There are two folders, **app** (see point 1 below) and **res** (see point 3 below).

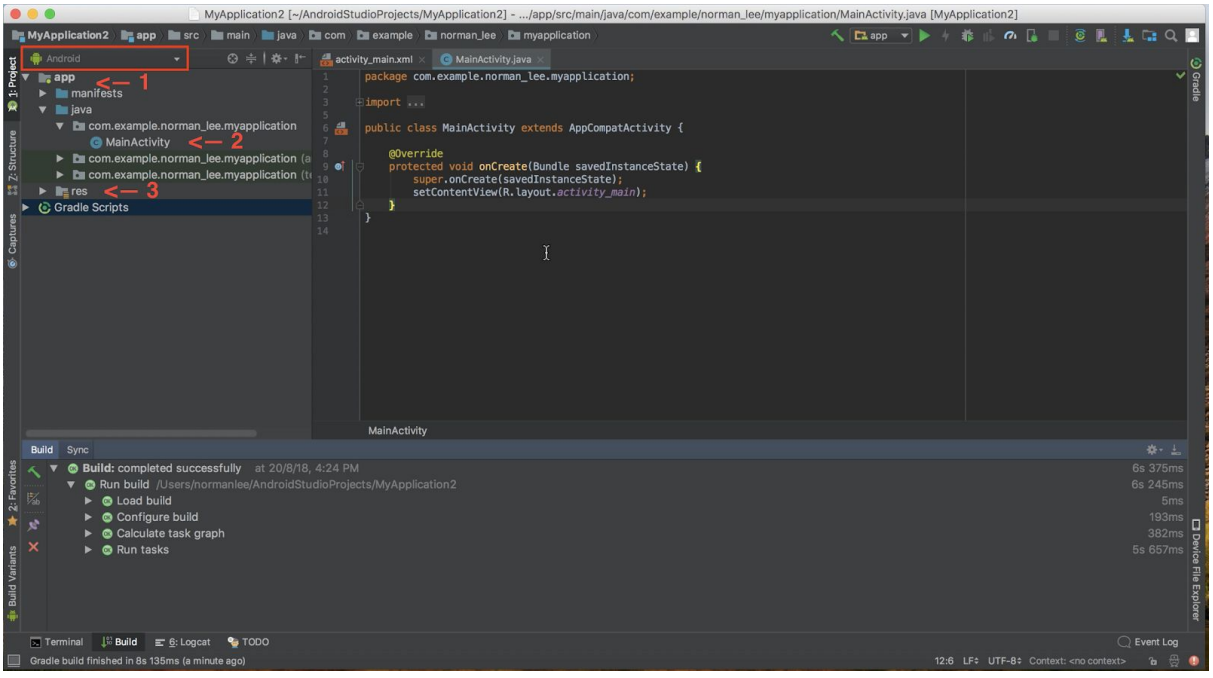
Recall earlier that when an **activity** is created, you get two files:

- An xml file named `activity_main.xml`
- A java class in a file named `MainActivity.java`

The **app** folder contains the java code and the java code `MainActivity.java` is found in the package folder within the java folder (see point 2).

The **res** folder (see point 3) contains resources for the app. This includes xml files, images and icons.

Expand the **res** folder and look for `activity_main.xml` file under the **layout** subfolder.



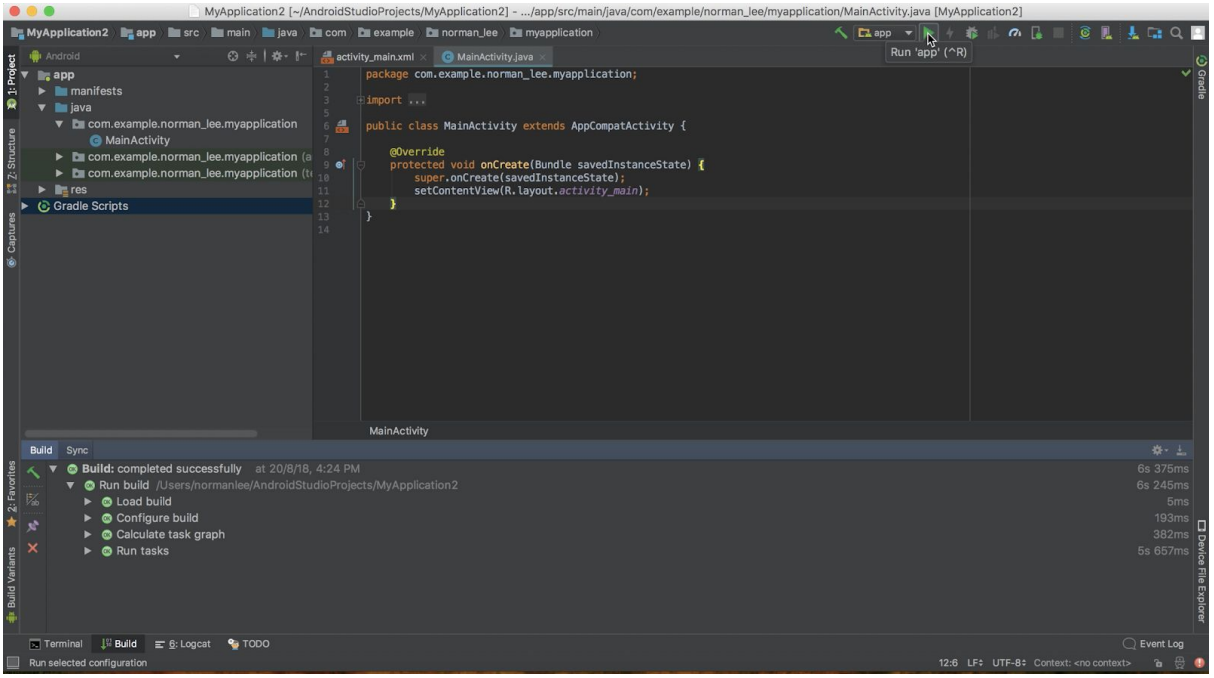
Test your App

If you have an android phone, follow the instructions:
https://developer.android.com/studio/run/device

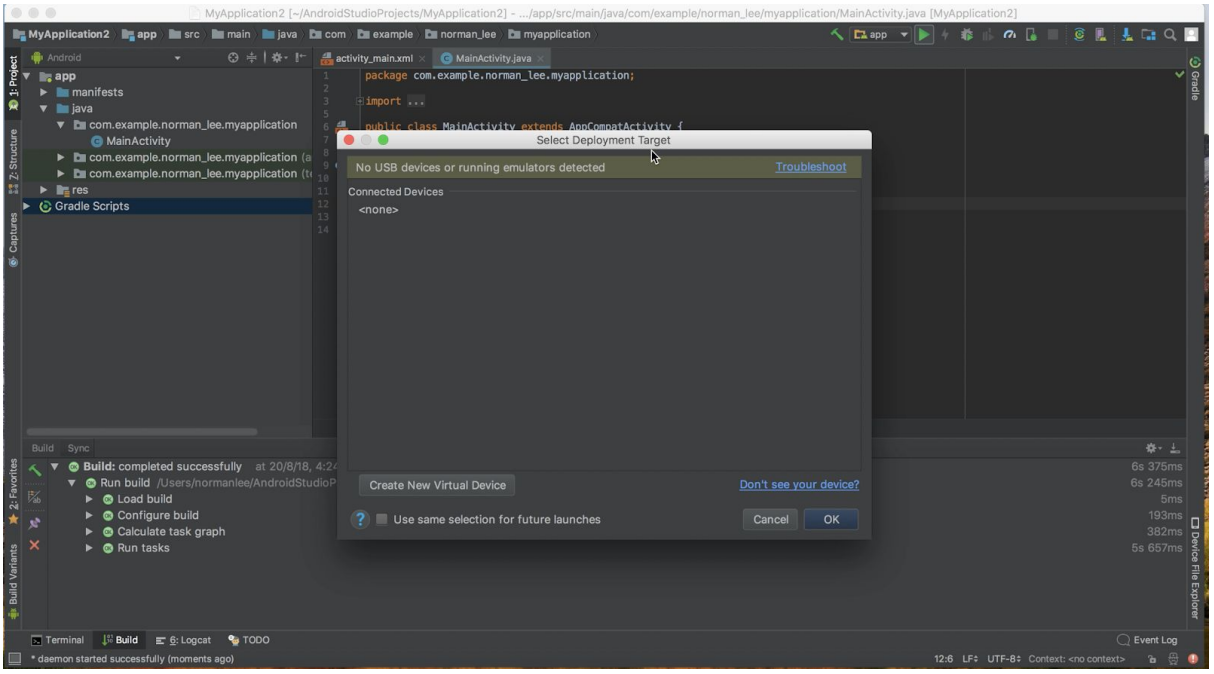
If you wish to use the emulator, continue with the rest of this guide. You will need at least 30 minutes more from now if you are doing a fresh installation.

Installing the Emulator

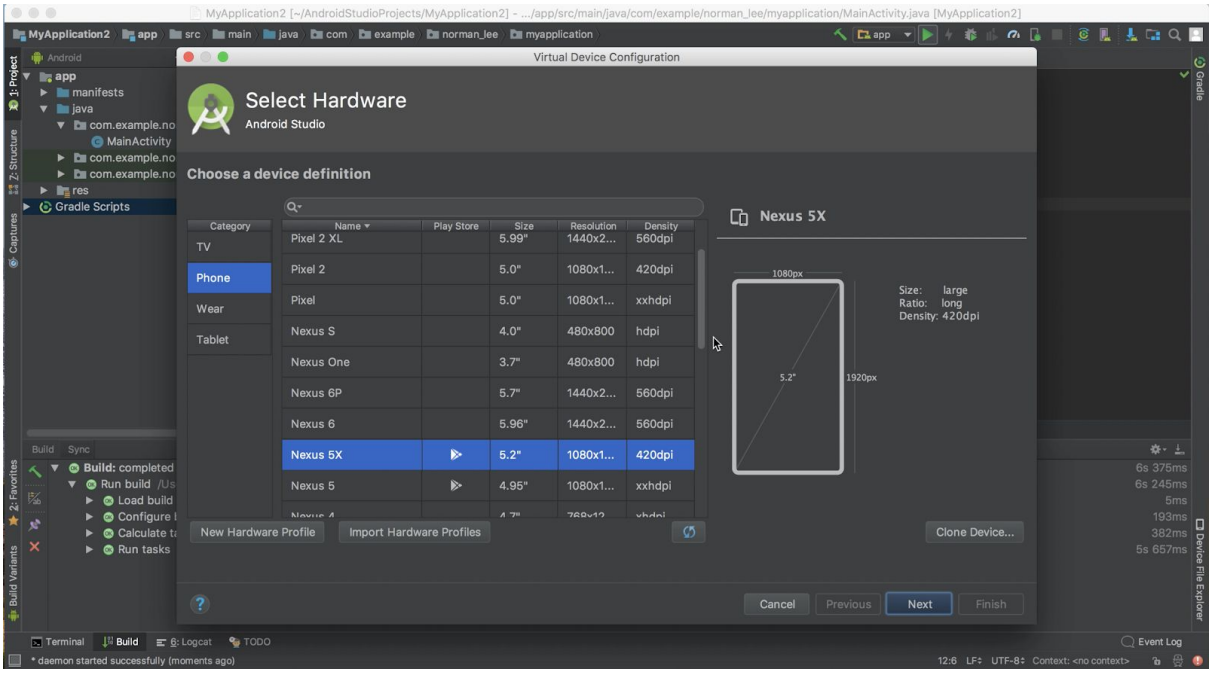
Go to the Run button and click it. Make sure “app” is shown beside it.



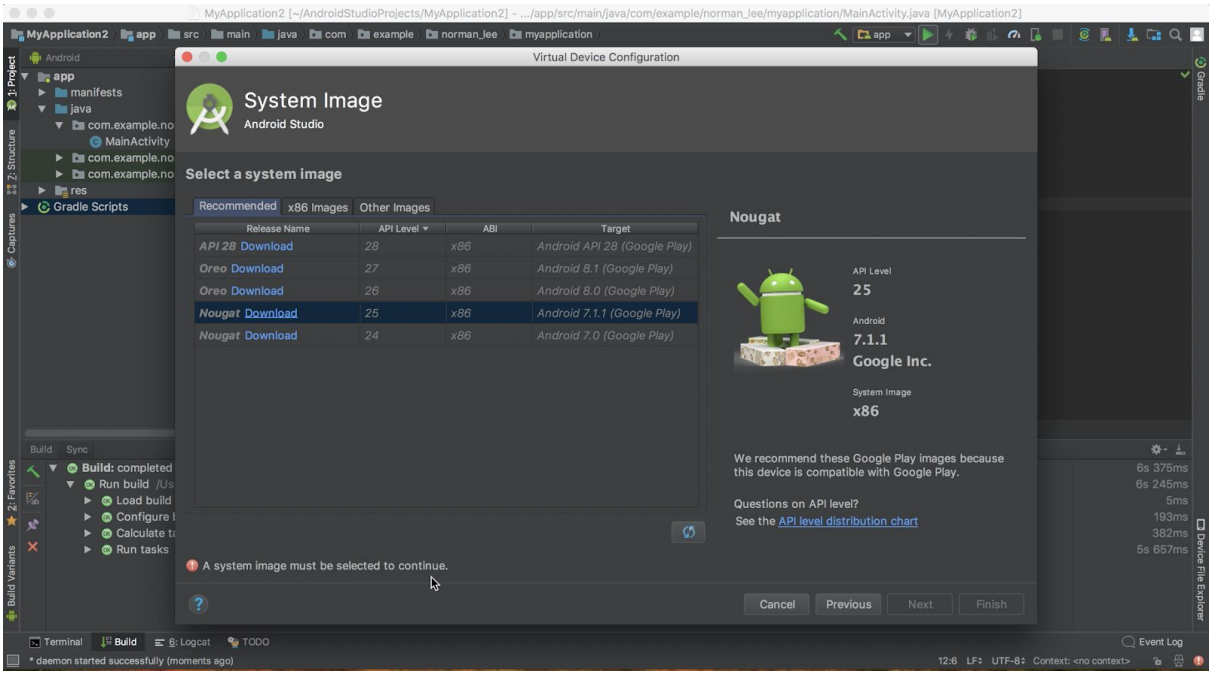
In the following window you should select **Create New Virtual Device**.



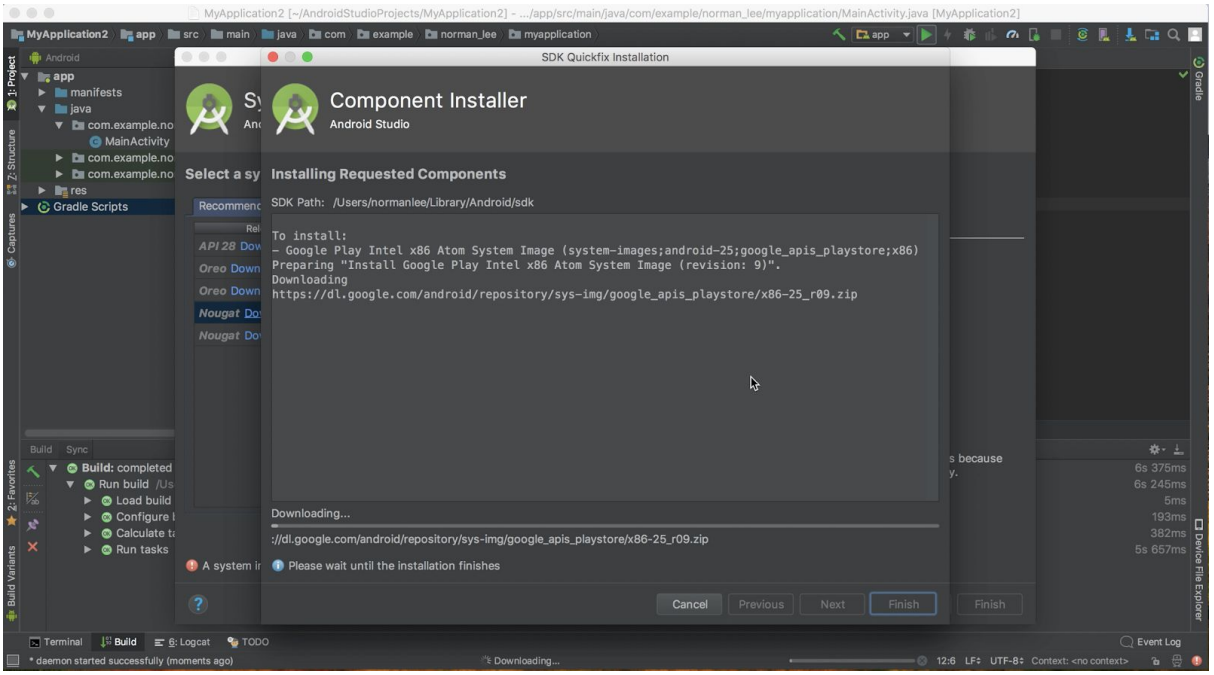
Many different devices are available for you with different screen sizes. You can explore by scrolling up and down. Just accept the defaults by clicking next.



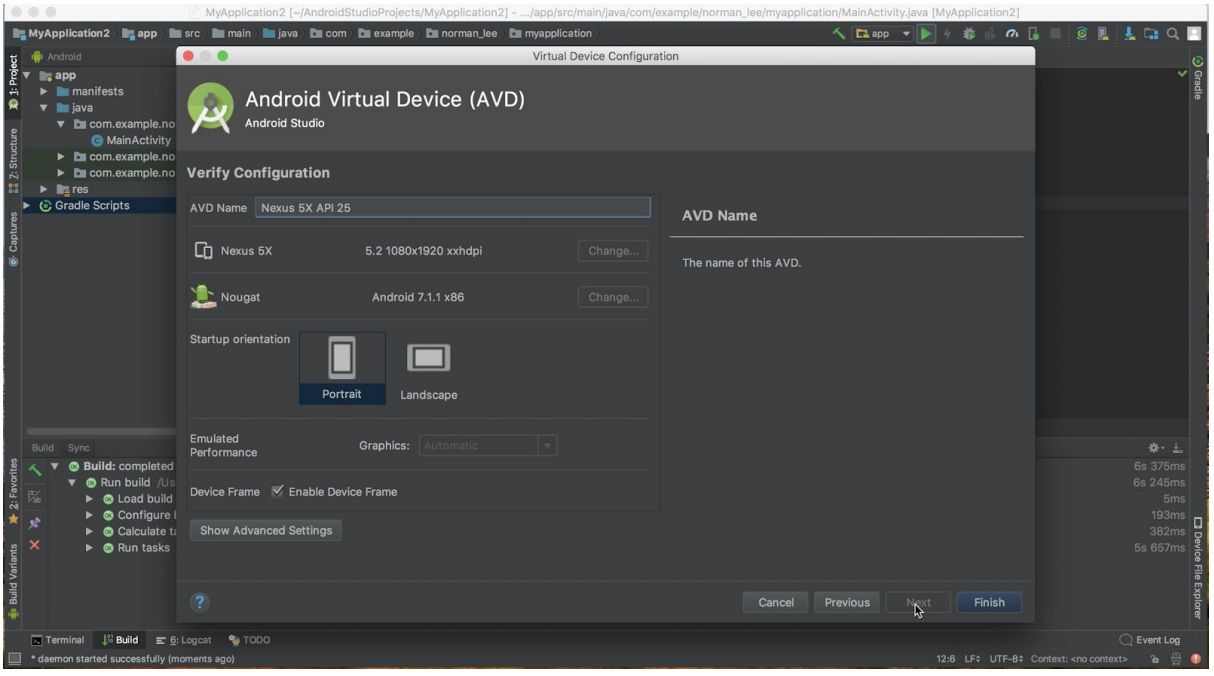
You are brought to this screen, where you have a choice of API levels (remember the API level is the version of the android OS). Choose API Level 25, we are not likely to need higher levels. After selecting the API Level, click on the Download link.



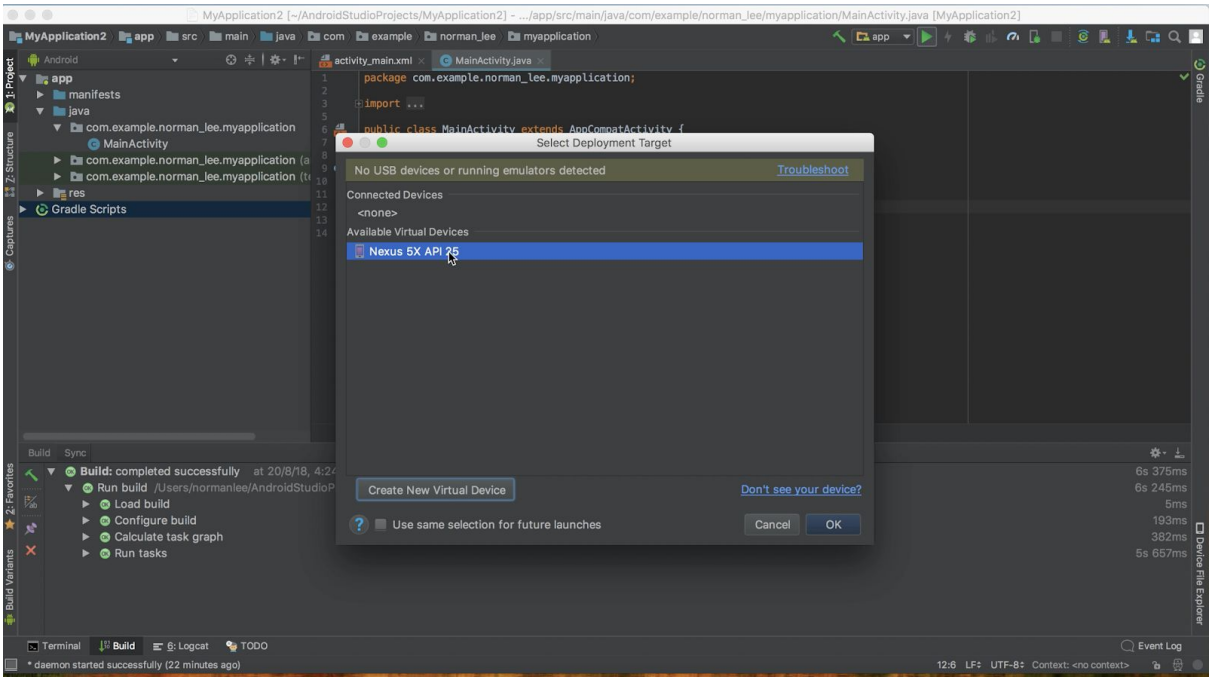
The following window pops up, and I had to wait for 20 minutes for the download to complete. If this is the case for you, find something else to do in the meantime. Click **Finish** when the download completes.



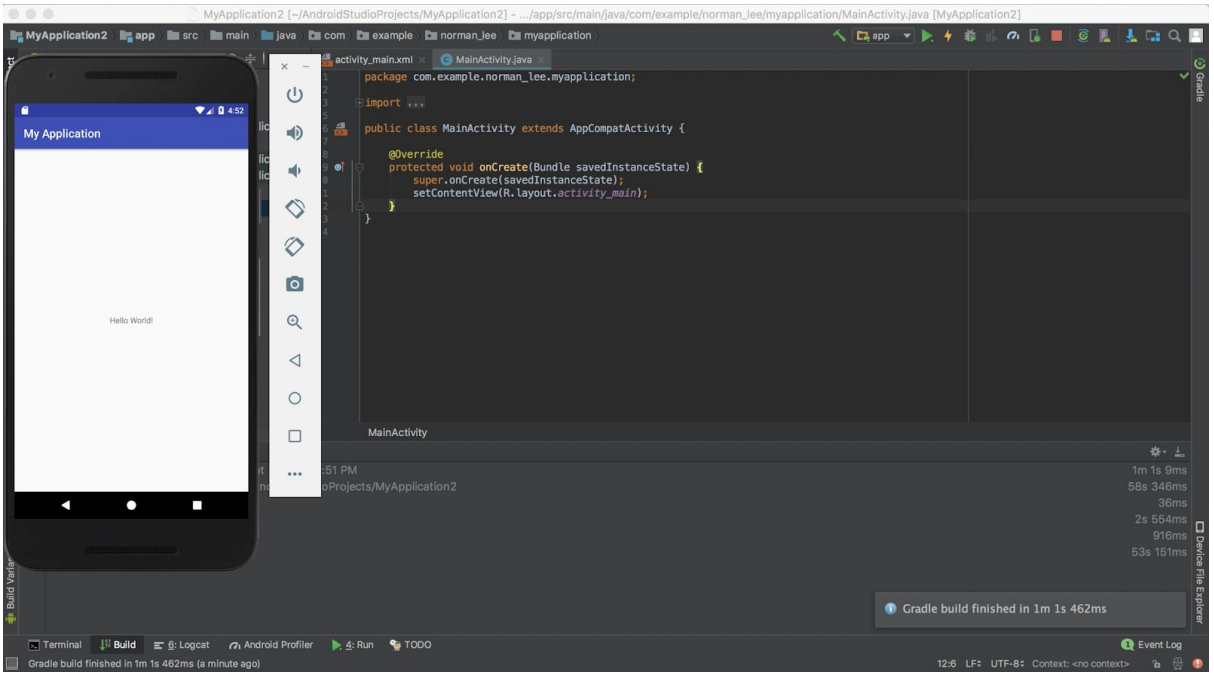
Next you are prompted for the settings of the emulator. Just accept the defaults and click Finish.



Now, an emulator is available in the window. Select it and click ok.



The emulator should now display the app. Congratulations, you have just completed your very first android app!



Further Reading

- Android Developer Fundamentals Concepts Section 1.1 - this was written in 2016 and changes have occurred since then, but it still gives a good overview of the process https://google-developer-training.gitbooks.io/android-developer-fundamentals-course-concepts/content/en/Unit%201/11_c_create_your_first_android_app.html

Lesson 1 - Random Images

Objectives

- Modify the xml layout file to specify LinearLayout, TextView and Button widgets, their id attribute and layout attributes
- Describe nested classes and anonymous classes in java
- use the instance method findViewById()
- Describe the R class in android
- Explain what is meant by inflating the layout
- Write java code in onCreate()
- Write java code to modify the text attribute of a widget
- Write java code to implement a callback

Explaining the XML Layout File

Linear Layout

Edit the XML file generated for you by replacing **`ConstraintLayout`** with **`LinearLayout`**.

In Linear Layout:

- The widgets are stacked in sequence according to the orientation.
- Two possible orientations: **`horizontal`** and **`vertical`**
- If no orientation attribute is specified, the default orientation is horizontal.

```
<LinearLayout
android:orientation = ”vertical”
 >
```

The three TextView widgets are stacked vertically.

```
<LinearLayout
android:orientation = ”horizontal”
>
```

The three TextView widgets are stacked horizontally.

TextView Widget

An XML tag for a basic TextView Widget is shown below.

```
<TextView
    android:id="@+id/myTextView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center"
    android:gravity="end"
    android:text="second"/>
```

- The **`id` attribute** enables you to give a unique ID to each widget in the XML layout file. This allows you to access the widget through the java code.
- The **`text` attribute** specifies the text that the widget should contain.

Button Widget

A possible XML tag of a basic Button widget is shown below.

```
<Button
    android:id="@+id/myButton"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="Click Me"/>
```

Here, I remove some attributes, showing you the minimum necessary to specify a widget.

How are the TextView and Button classes related?

Have a look at the documentation

https://developer.android.com/reference/android/widget/Button

Sizing A Widget

For the **`layout_width`** and **`layout_height`** attributes

- **`wrap_content`** sizes the widget to fit the content
- **`match_parent`** sizes the widget to fit the screen size

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="AA"/>
```

The widget sizes itself to fit its content.

```
<TextView
    android:layout_width=”match_parent"
    android:layout_height="wrap_content"
    android:text="AA"/>
```

The width of the widget is equal to the width of the screen.

Alignment

Note the difference between the two:

- To align a widget within a layout, use the **`layout_gravity`** attribute
- To align the contents of a widget within itself, use the **`gravity`** attribute

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content”
    android:layout_gravity=”center"
    android:text="AA"/>
```

The widget aligns itself to the centre of the layout.

```
<TextView
    android:layout_width=”match_parent"
    android:layout_height="wrap_content”
    android:gravity=”center"
    android:text="AA"/>
```

The contents of the widget aligns itself in the centre of the widget.

To understand the difference:

- In which scenario would the **`gravity`** attribute have no effect?
- In which scenario would the **`layout_gravity`** attribute have no effect?

The Java you need to know

ArrayList

```
List<Integer> a = new ArrayList<>();
a.add(1);
a.add(2);
a.add(1,3);
a.add(5);
System.out.println(a.toString());
```

What is printed on the screen?

Private vs Public

```
class Point2D{
    private double x;
    private double y;

    Point2D(){
        //code not shown}

    Point2D(double x, double y){
        this.x = x;
        this.y = y; }

    public double getX() { return x; }
    public double getY() { return y; }
}

class Point3D extends Point2D{

    private double z;

    Point3D( double x, double y, double z ){
    }
}
```

Complete the constructor for Point3D.

Recall Overriding vs Overloading

We see **overriding** and **overloading** in android very often, so it is good to recap these concepts.

To override a method in a super-class, the **method signature** in the subclass must be the same.

The **`@Override`** annotation allows the compiler to help you check if you have got this condition correct.

```
class Dog{
    public void bark(){ System.out.println("woof"); }
    public void drool(){ System.out.println("drool");}
}

class Hound extends Dog{
    public void sniff(){ System.out.println("sniff ");}
    @Override public void bark(){ System.out.println("growl");}
    public void drool(int time){ System.out.println("drool" + time);}
}
```

Given **`Hound h = new Hound();`**

- What will you see on the screen for **`h.bark()`** ?
- What will you see on the screen for **`h.drool(1)`** ?
- What will you see on the screen for **`h.drool()`** ?

Given **`Dog g = new Hound();`**

- What will you see on the screen for **`h.bark()`** ?
- What will you see on the screen for **`h.drool(1)`** ?
- What will you see on the screen for **`h.drool()`** ?

Recall Polymorphism

```
class A {
   void f(int x){System.out.println("A");}
}

class B extends A{

   void f(int x){System.out.println("Bf");}
   void g(int x){System.out.println("Bg");}
}
```

Which of the following statements is illegal?

(a) `A x = new A();`

(b) `A x = new B();`

(c) `B x = new B();`

(d) `B x = new A();`

For the code **`x.f(1)`**, which two statements above initialize **`x`** such that **`Bf`** is printed out?

Interfaces

An Interface is like a contract for the implementations of classes and helps in maintenance of software.

```
interface I {
   void m(int x);
}

class K implements I{
   void m(int x){System.out.println("m");}
}
```

Which of the following statements is/are legal?

`(i) K x = new K();`

`(ii) K x = new I();`

`(iii) I x = new K();`

`(iv) I x = new I();`

(a) (i) only

(b) (i) and (ii)

(c) (i) and (iii)

(d) (i), (ii) and (iii)

Bearing in mind interface **`I`** and class **`K`** implements **`I`** (defined above)
Which method below is better?

```
void firstMethod(K k){ //do something;}
void secondMethod(I i){ //do something;}
```

A method that takes in an interface is more flexible.
It will be able to accept any object that implements that interface.
Suppose you create a new class implementing **I** that has a better implementation of **m**, you are able to pass it to **`secondMethod`** without having to change its signature.

Next Question ..

```
class A {

   void f(int x){System.out.println("Af");}
   void h(int x){System.out.println("Ah");}
}

class B extends A{

   void f(int x){System.out.println("Bf");}
   void g(int x){System.out.println("Bg");}
```

Given

`A x = new B();`

Which of the following can subsequently be executed?

`x.f(1); //statement (i)`

`x.g(1); //statement (ii)`

`x.h(1); //statement (iii)`

(a) (i) only

(b) (i) and (ii)

(c) (i) and (iii)

(d) (i), (ii) and (iii)

Exceptions

```
public class TestExceptions1 {

   public static void main(String[] args){
       try{
           f(-1);
           System.out.print("R");
       }catch(Exception e){
           System.out.print("S") ;
       }
   }

   static void f(int x) throws Exception {
      if( x < 0) throw new Exception();
      System.out.print("P");
   }
}
```

In the code above, what is printed out? (open-ended)

(a) S

(b) PRS

(c) RS

(d) PR

```
public class TestExceptions2 {

   public static void main(String[] args){
       try{
           f(-1);
           System.out.print("R");
       }catch(Exception e){
           System.out.print("S") ;
       }
   }

   static void f(int x) throws Exception {
       try{
           if( x < 0) throw new Exception();
           System.out.print("P");
       }catch( Exception e){
           System.out.print("Q");
       }
   }
}
```

In the code above, what is printed out?

(a) Q **(b)** S

(c) QR **(d)** QRS

Points to note

- When an **exception** is thrown, the Java runtime searches through the **call stack** to find the first method that will handle the exception.
- The `finally` block is always executed regardless of what happens in the `try` block.
- It is good programming practice to specify exactly the type of exception that is handled in each catch block, as you will have specific details of the exception that occurred. Hence code examples here are not good …

Random Class

In many applications it is useful to generate random numbers.

In Java, you do it by getting an instance of the **`Random`** class.

In this class there are three useful methods

- **`nextInt()`** gives you an integer between 0 and 232 (exclusive)
- **`nextInt(n)`** gives you an integer between 0 and n (exclusive)
- **`nextDouble()`** gives you a double between 0.0 and 1.0

```
Random r = new Random();
r.nextInt();
r.nextInt(100);
r.nextDouble();
```

Random number generators usually need to be initialized with a seed.

If you need the sequence of random numbers to be the same, you use the same seed.

If not, one way to get a changing seed is to use the Date object.

```
Date d = new Date();
Random r = new Random(d.getTime());
```

Nested Classes

A class definition can contain class definitions. We call these classes **nested classes**.

```
public class OuterClass {
   // code not shown

   class InnerClass{
       //code not shown
   }

}
```

This is typically done when you have classes that logically depend on the outer class and are used together with the outer class.

Inner Class

A nested class that is not declared static is called an **Inner Class**.

- To instantiate an inner class, you need an instance of the outer class, which is usually called the **enclosing class**.
- The inner class can access all methods and variables of the enclosing outer class.

```
public class OuterClass {

   int a;
   OuterClass(){  a = 10;  }
   void outerPrintA(){  System.out.println(a);  }

   class InnerClass{
       int c;

       InnerClass(){ c = 100; }

       void innerPrintA(){  System.out.println(a);  }

       OuterClass giveBackOuter(){  return OuterClass.this; }
   }
}
```

Activity. For **`OuterClass`**, complete the main function below to illustrate the following properties.

```
public class TestOuterClass {
   public static void main(String[] args){
       //Instantiate OuterClass
       OuterClass outerClass = new OuterClass();

       //Instantiate the InnerClass
       OuterClass.InnerClass innerClass = outerClass.new
InnerClass();

       //Show that InnerClass can access variables in OuterClass

       //Show that InnerClass stores a reference to OuterClass
   }
}
```

Static Nested Classes

By declaring a nested class as static, it is known as a **static nested class.**

- It can only access static variables and methods in the outer class.
- It can be instantiated without an instance of the outer class.

A static nested class behaves like a top-level class and is a way to organize classes that are used only by some other classes.

Activity.

- Modify **`OuterClass.java`** by declaring **`InnerClass`** as static and adjusting other parts of the class accordingly e.g. which other variables must be static? Which methods do not work anymore?
- Write code to show that you can instantiate OuterClass and InnerClass separately.

Nested Interface & Anonymous Classes

Recall that interfaces make your code reusable. We may nest interfaces as well. Recall also that Interfaces are inherently static. In the following code, any object that implements **`SomeClassExample3.MyInterface`** can be passed to **display()**.

```
public class SomeClassExample3 {

    interface MyInterface{
        void printSomething();
    }

    int a;

    SomeClassExample3(){
        a = 10;
    }

    public void display(MyInterface myInterface){
        myInterface.printSomething();
    }
}
```

So we may write the following code to make use of it.

```
public class OuterClassExample3 {
    SomeClassExample3 someClassExample3;

    OuterClassExample3(){
        someClassExample3 = new SomeClassExample3();
    }
    public void callMe(){
        someClassExample3.display( new InnerClass());
    }
    class InnerClass implements SomeClassExample3.MyInterface{
        @Override
        public void printSomething(){
            System.out.println("Hello");
        }
    }
}
```

Anonymous Class

Often, if the Inner Class is used only once, an alternative is an **Anonymous Inner Class**, to avoid declaring too many classes.
In the **`OuterClassExample3`** above, the following code may be used instead.

```
someClassExample3.display(new SomeClassExample3.MyInterface() {
   @Override
   public void printSomething() {
       System.out.println("hello");
   }
});
```

As you can see, we have an **anonymous class** because

- We do not name the class that implements the interface
- We do not assign a variable name to the class that implements the interface

We see nested interfaces, nested static classes and anonymous classes in Android programming frequently.

Further Reading

- **Nested Classes and Anonymous Classes at Oracle's Java Tutorial**
    - https://docs.oracle.com/javase/tutorial/java/javaOO/nested.html
    - https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html

- ***Bloch, Effective Java,* Item 22.**

onCreate is called when the Activity is first launched

Within the **`MainActivity`** class, you would see this code

```
@Override
protected void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   setContentView(R.layout.activity_main);
}
```

The **`onCreate`** method is called whenever your Activity is first launched e.g. when the user clicks on your app icon.

This method is part of the methods in the **Android activity life cycle**, which will be discussed in the next lesson.

You write code in **`onCreate`** to implement what you want the user to see when the activity is launched.

The R class contains resource IDs to the resources in the res folder.

When the app is compiled, a **R class** is generated that contains IDs to the resources in the **res** folder.

Since **`activity_main.xml`** is stored in the layout folder, its R class reference is **`R.layout.activity_main`**.

In onCreate, the layout is first inflated

`R.layout.activity_main` is passed to the **`setContentView`** method to **inflate the layout**. In this process, Android reads the XML code in the layout file and instantiates objects in the memory that represent each of the widgets on the Activity.

More examples of Resource IDs

Widget ID

If your widget has the following attribute

```
android:id =”@+id/myWidget”
```

then it can be accessed by **`R.id.myWidget`**.

Images in drawables

If you have an image stored in the drawable folder named **pikachu.png**, then it can be accessed by **`R.drawable.pikachu`**.

Note that image filenames must all be lowercase.

Clicker Question - What type of class is the R class?

The R class contains nested classes. True/False.

Seeing the R class (Optional)

The R class is generated for you but you may view it in your project as follows

- Change to Project View
- Access the folder path: app/build/generated/source/r/debug/<your.package.name>

In general, we don’t actually have to do anything to this file

Use findViewById() method to assign a widget to a variable

If a widget has an id attribute **`myTextView`**, then the corresponding reference in the R class is **`R.id.myTextView.`**

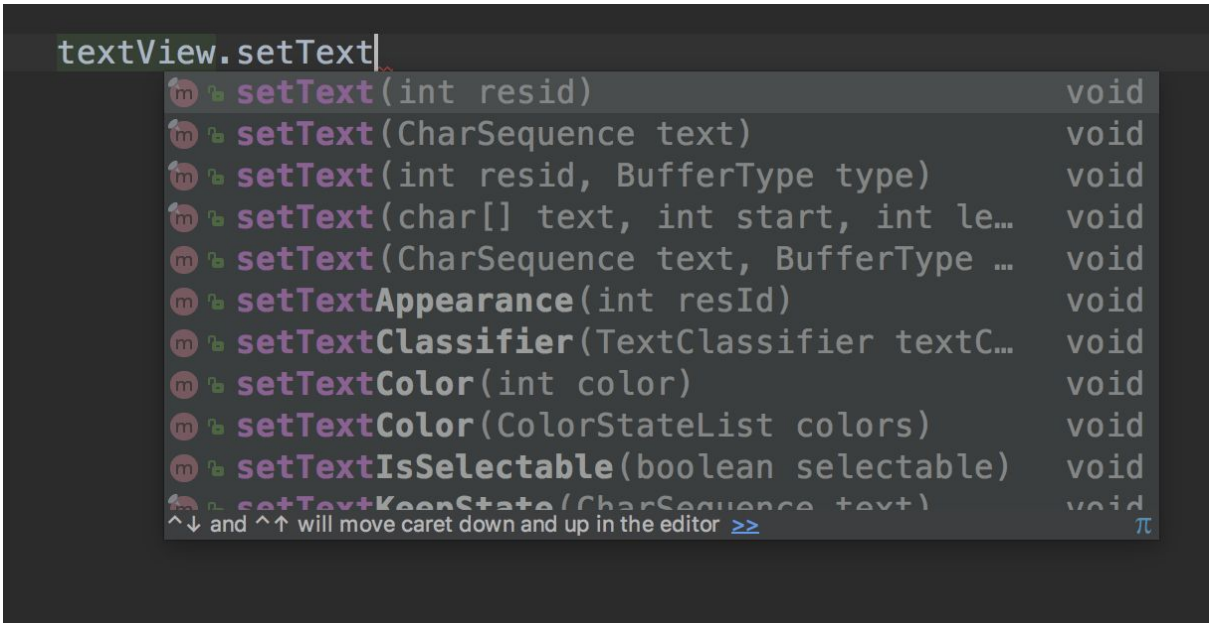
This reference is then passed to **`findViewById()`** , which returns a reference to the widget. This can then be assigned to a variable. A sample code is as follows:

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

When typing the code out in Android Studio, press **Alt + Enter (Windows)** or **Option + Enter (Mac)** to import the necessary library.

Use the dot operator to see what methods are available for that widget

Once you have a variable, use the dot operator to see what methods are available. For example, you can see that the setText method is an overloaded method, it can take in another resource ID or a character array.



You can control a widget's properties in Java

Some methods, like the **`setText`** method, enable you to control a widget's properties programmatically in Java. For example:

```
textView.setText("My New String")
```

This action thus replaces what is written in the XML layout file.

When a View object is clicked, what happens next is specified by calling setOnClickListener()

Typically, we want Button to be clicked, but it is also possible to have other widgets clicked, including TextView, LinearLayout etc.

The input to **`setOnClickListener`** is an object of a class that implements the **`View.OnClickListener`** **interface**. There is one method to implement, called **`onClick`**. You may implement this in several ways, here I list three ways:

Choice 1. As an inner class in `MainActivity`.

This method shows you clearly what you are doing. But it may cause your MainActivity.java to become bloated with inner classes that you use only once.

```
public class MainActivity extends AppCompatActivity {

   Button button;

   @Override
   protected void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
       setContentView(R.layout.activity_main);

       button = findViewById(R.id.myButton1);
       button.setOnClickListener( new ClickMe());
   }

   //*** this is an inner class ***
   class ClickMe implements View.OnClickListener{

       @Override
       public void onClick(View v) {
               //code goes here
       }
   }
}
```

Choice 2 (Recommended). As an anonymous class that is defined in the input to setOnClickListener():

This is the recommended method because it is used very frequently, and in many other situations.

```
public class MainActivity extends AppCompatActivity {

   Button button;

   @Override
   protected void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
       setContentView(R.layout.activity_main);

       button = findViewById(R.id.myButton1);

       button.setOnClickListener(new View.OnClickListener() {
           @Override
           public void onClick(View v) {
                 //code goes here
           }
       });
   }
}
```

Choice 3. Define an instance method in MainActivity specifying what is to be done. Then specify it as an attribute in the widget.

Although it looks straightforward and easy, I don't recommend this choice, for the following reasons.

- Many code examples used in teaching android use anonymous classes instead
- This does not work in many other situations.

Define a instance method in MainActivity with any name you like and the following signature.

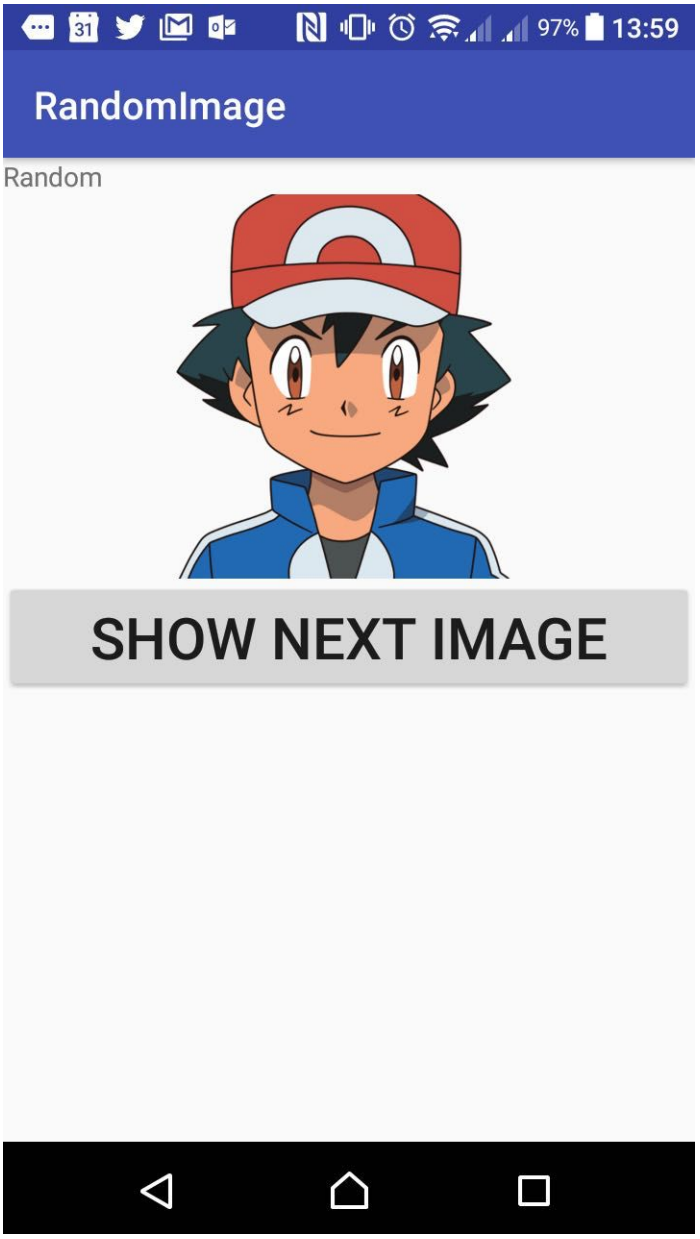
```
public class MainActivity extends AppCompatActivity {

   Button button;

   @Override
   protected void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
       setContentView(R.layout.activity_main);

       button = findViewById(R.id.myButton1);

   }
   //this method is what myButton1 will do
   public void whenClicked(View view){
        //code here
   }
}
```

Then in the XML file, the button widget will have the **onClick** attribute.

```
<Button
   android:id="@+id/myButton1"
   android:layout_width="match_parent"
   android:layout_height="wrap_content"
   android:onClick="whenClick"
   android:text="Click Me"/>
```

Making our App

What the app should look like



The strategy with this app:

- Store all images in the res/drawables folder. Put the Image IDs in an ArrayList
- When the Button is clicked, retrieve the image ID from the ArrayList in sequence.
- Use the image ID to retrieve the image and place it in the ImageView widget.

Code Stump For MainActivity.java

Copy the following and paste it in **MainActivity.java** in the project that you generated in Lesson 0. Import any classes as needed.

```
//TODO 1.1 Put in the images in the drawables folder
//TODO 1.2 Go to activity_main.xml and put in the layout

public class MainActivity extends AppCompatActivity {

    ArrayList<Integer> images;
    Button charaButton;
    ImageView charaImage;
    int count = 0;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        //TODO 1.3 Instantiate an ArrayList object
        //TODO 1.4 Add the image IDs to the ArrayList
        //TODO 1.5 Get references to charaButton and charaImage
using findViewById
        //TODO 1.6 For charaButton, invoke the
setOnClickListenerMethod
        //TODO 1.7 Create an anonymous class which implements
View.OnClickListener
        //TODO 1.8 Write code to randomly select an image ID from
the ArrayList and display it in the ImageView
        //TODO 1.9 [On your own] Create another Button to always
display the first image

    }
}
```

TODO 1.1 and 1.2 - Images and Layout

After putting the images in the drawables folder, you are ready to modify **`activity_main.xml`**

Layout

- Replace the tag **`ConstraintLayout`** with **`LinearLayout`**
- In the list of attributes, specify that its **`orientation`** is vertical

Widgets

- Put one TextView widget containing the text “Random Images”
- Put one ImageView widget
    - assign any image to it using the **`src`** attribute
    - Give it the id **`charaImage`**

- Put one Button widget
    - Its text shall be “Show Next Image” or anything else you like
    - Give it the id **`charaButton`**

Compile the app and view it on your phone. You should see three widgets.
If you don’t, you may have forgotten to specify that the linear layout is vertical.

TODO 1.3 and 1.4 - Instantiate the ArrayList object and add the image IDs to it

You would have learnt how to instantiate an ArrayList object in week 1.
How to write the image IDs using the R class has also been explained earlier.

```
images = new ArrayList<Integer>();
images.add(R.drawable.ashketchum);
images.add(R.drawable.bartsimpson);
```

TODO 1.5 - Get references to the charaButton and charaImage widgets

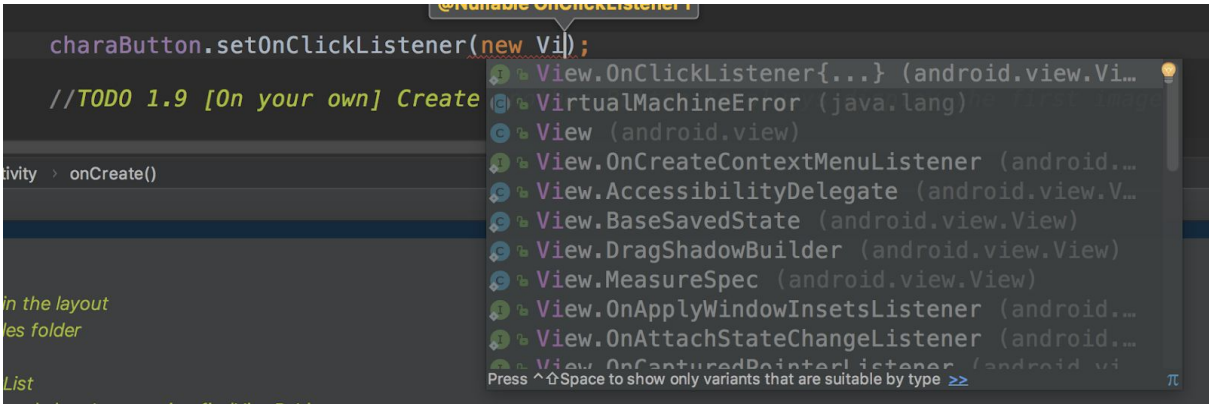
Recall that you use findViewById() and you use the R class to reference the widget IDs.

```
charaButton = findViewById(R.id.charaButton);
charaImage = findViewById(R.id.charaImage);
```

TODO 1.6 & 1.7 For charaButton, Invoke the setOnClickListener() and create an anonymous class which implements View.OnClickListener

After typing out the first few characters, select the first one from the drop-down list.

The code stump is automatically created.



We create an anonymous class instead of the other methods.

This is by far the most common way that I have seen.

TODO 1.8 Write code to randomly select an image ID from the ArrayList and display it in the ImageView

From the code stump generated, within **`onClick`**, write code to retrieve a random element from the ArrayList. The `Random` class will be helpful.
Once you have done so, you assign it to `charaImage` using the `setImageResource()` method.

```
charaImage.setImageResource(id);
```

Solution Code for TODO 1.6 - 1.8

I present the solution where the images are accessed from the array in sequence. Modify it such that it is accessed randomly using the Random class.

```
charaButton.setOnClickListener(

        new View.OnClickListener() {
            @Override
            public void onClick(View view) {

                int index = (count )% images.size();
                count = count + 1;

                int id = images.get(index);
                charaImage.setImageResource(id);
            }
        }
);
```

TODO 1.9 [Try Yourself] Create another Button to always display the first image

To test yourself to see if you have understood what to do, try this.

- Add another button to the layout.
- When this button is clicked, it will display a particular image of your choice.

Lesson 2 - Exchange Rate App

Objectives

- Describe the static factory method and builder design pattern in Java
- From an EditText widget, extract data and specify input settings
- Use the logcat to display messages to track the behaviour of an
- Describe what a Toast is and write code to display toasts
- Explain what is an Explicit Intent and write code to implement it
- Explain what is an Implicit Intent and write code to implement it
- Modify the android manifest to change the app name and to specify a parent activity
- Describe the Android activity life cycle
- Describe and modify the code needed for an Options Menu
- Save app data using the SharedPreferences class
- Explain the purpose of Instrumented testing and write code to implement instrumented testing using the Espresso framework

Introduction

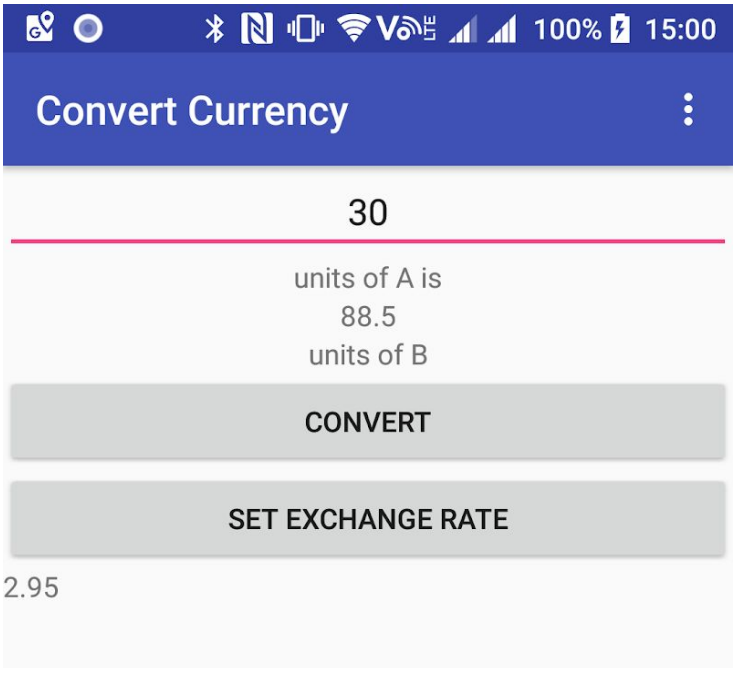
In this series of lessons, we will build an app that is handy for travelling. I often travel with people who want to know exactly how much an item would cost in their home currency.

MainActivity

When the app is launched for the very first time, the following Activity is seen, with a default exchange rate of 2.95 i.e. 1 unit of A buys 2.95 units of B.

- The screenshot shows that the user has entered '30' and obtained a result of 88.5 after clicking **Convert**.
- You can see the default exchange rate displayed below.
- To set the actual exchange rate, the user would have to click on **Set Exchange Rate**, which brings the user to **SubActivity**.
- The three dots on the top right open an **Options Menu**.

The UI is kept bare in order to concentrate on the coding.



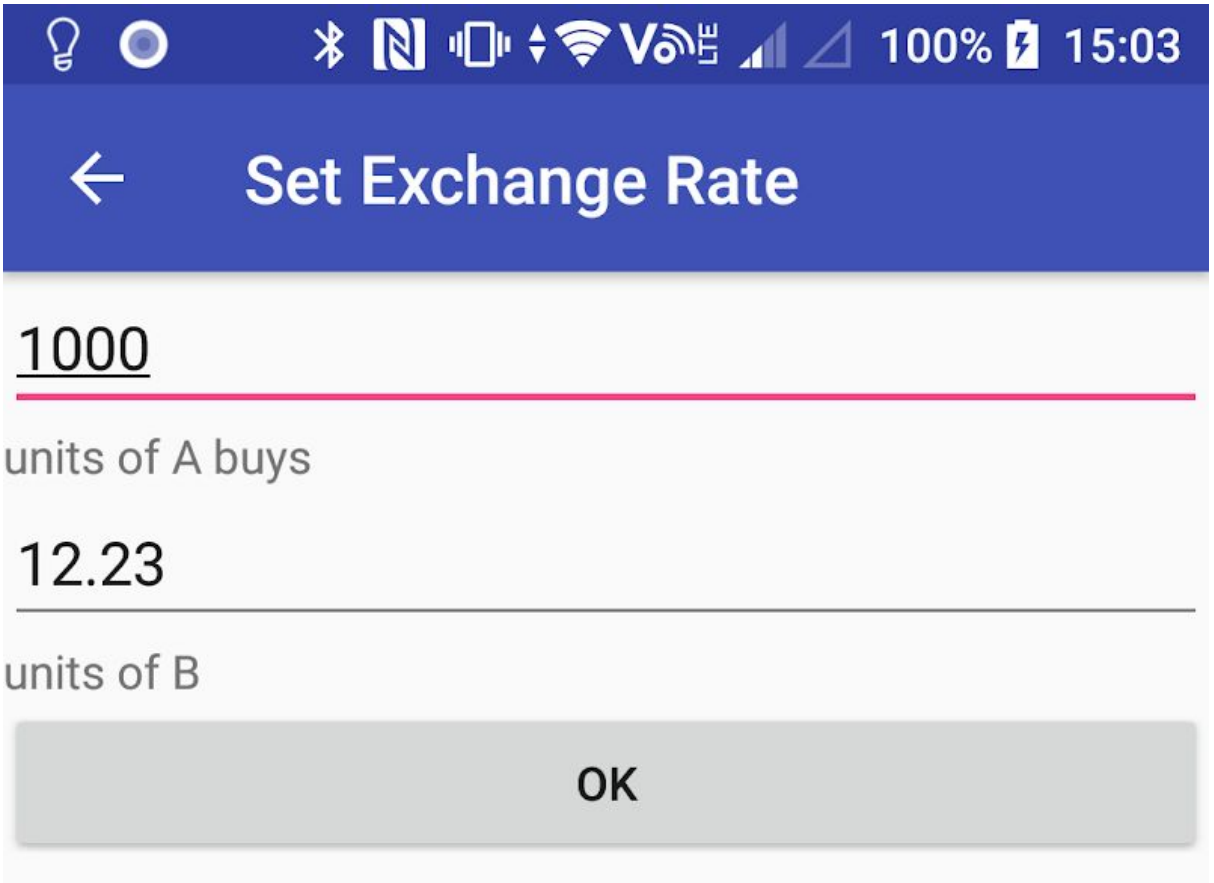
SubActivity

A screenshot of SubActivity is shown below.

- It allows the user to key in the exchange rate at which he/she bought the currency in any ratio.
- You can see the user has keyed in 1000 units of A buys 12.23 units of B.
- Clicking **OK** brings the user back to **MainActivity**.
- The user can also choose to click on the top-left-hand arrow to go back.

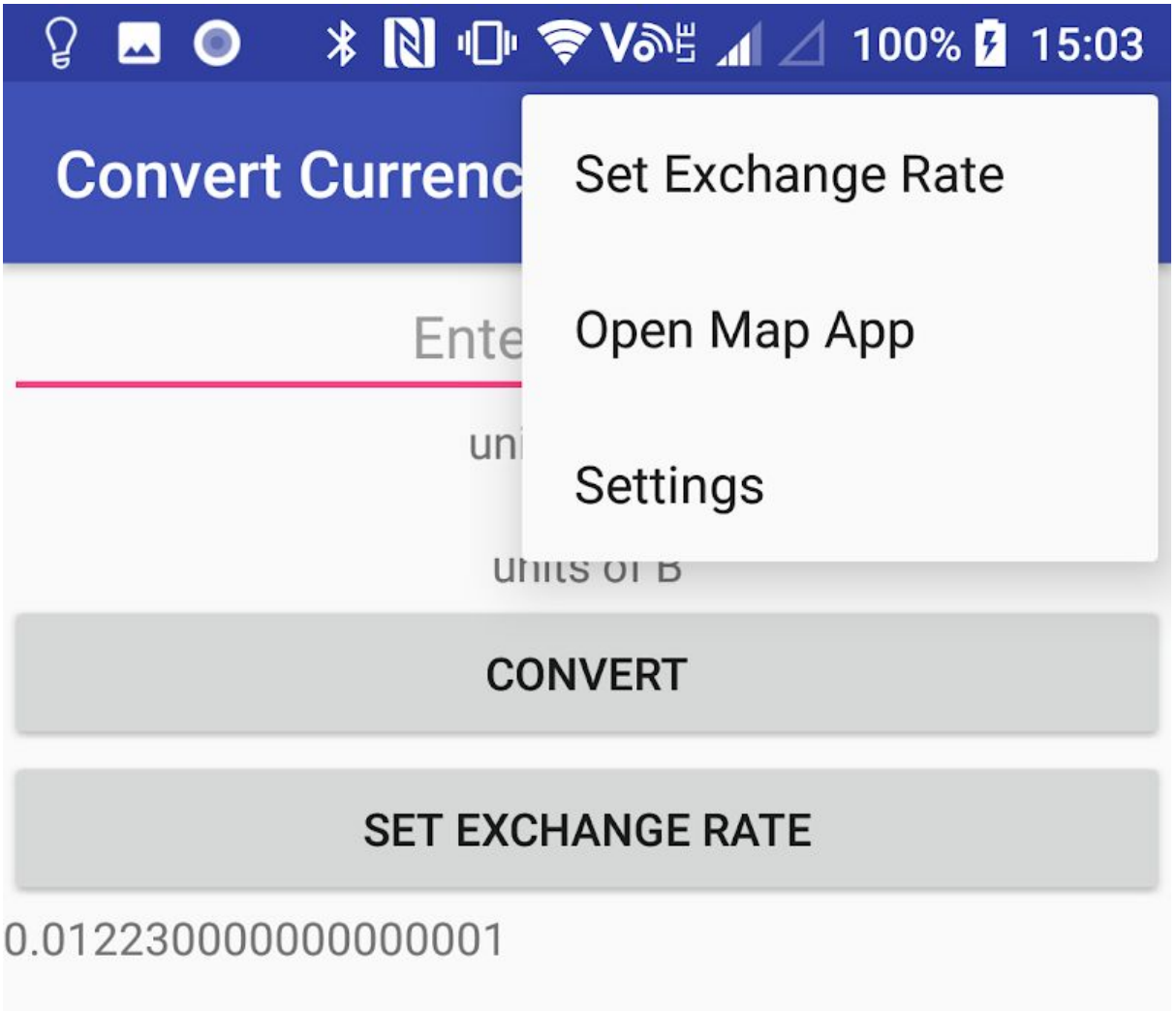
You would also have to deal with the situations:

- the user keys in non-numeric data
- 0 is entered for the units of A



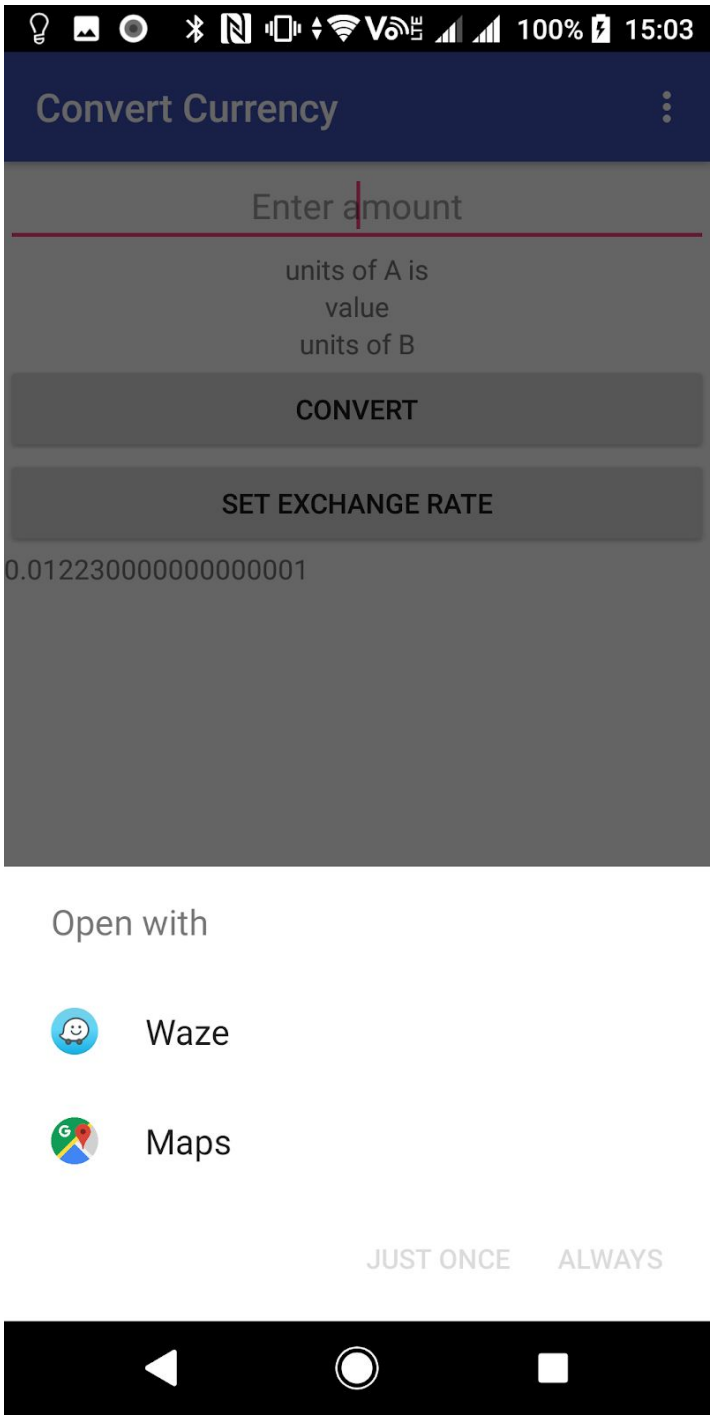
Back in Main Activity

- Once the exchange rate is set, the new exchange rate is reflected in **MainActivity**.
- Without some coding, closing and reopening the app erases the data entered. We would like the data to be retained, by using the **Shared Preferences** class.
- We would like to give the user more options to navigate around the app, by clicking on the three dots.
- This opens the **Options Menu**.
- The Options Menu has the following items (*Settings* is a dummy item)
    - *Set Exchange Rate*, which brings the user to SubActivity
    - *Open Map App*, which brings the user to the Map Apps residing on his/her phone



When Open Map App is clicked

After clicking on Open Map App, the following screen is shown. The user is allowed to pick from the available map apps on his/her phone.



Get Ready

1. Download the starter code from eDimension
2. Open the project in android studio.
3. Try to run the app on your phone or emulator.
   If you encounter problems, try **Build → Rebuild Project** or **Clean Project**
4. You'll find that the buttons are not responsive, but you can click on the three dots.
5. Examine the **res/layout** folder and note where the layout files are stored and how they are connected together.
6. Answer the following questions:
   a. How many activities does the app contain?
   b. Currently, how many activities can you see in the app?

What you need to know (Android/Java)

res/values folder

The res/values folder stores constants that appear throughout your app, such as

- Strings
- Style definitions
- Color definitions
- Dimensions

This is useful for several reasons:

- Different parts of your app can use the same constants
- Changes can be made in one place instead of trawling through your code

Modifying res/values/strings.xml

The **`strings.xml`** file is the one place to string constants that are used throughout your app.
For example:

- Name of your app and activities
- Text of buttons or other widgets
- Messages in Toasts

The **`strings.xml`** file might look like

```
<string name="app_name">Exchange Rate</string>
<string name="action_settings">Settings</string>
<string name="set_exchange_rate">Set Exchange Rate</string>
<string name="main_activity_name">Convert Currency</string>
```

As you code your app, you are free to add to or modify this list.
An example of how constants here are accessed by the R class is
`R.string.set_exchange_rate`. (Note how string is spelled)

Android manifest

The android manifest is found in **app/manifest** folder and is in a file named **`AndroidManifest.xml.`**

It stores all important information regarding your app:

- App name
- App icon
- Activities that your app contains
- Permissions of your app (e.g. to access the internet)

The **`android:label`** attribute under the application tag specifies your app name.
Notice how it uses a reference to the strings.xml file.

```
<application
   --deleted--
   android:label="@string/app_name"
```

Notice also that there is a similar attribute under the **`activity`** tag.
Specifying this attribute thus changes the title of your activity as displayed on the screen.

The **`android:parentActivityName`** attribute specifies the parent Activity. A back arrow appears in the title, which allows the user to click to go back to its parent.

styles

EditText widget

A typical xml tag of an EditText widget:

```
<EditText
   android:id="@+id/editTextValue"
   android:hint="Enter amount"
   android:gravity="center"
   android:inputType="numberDecimal"
   android:layout_width="match_parent"
   android:layout_height="wrap_content" />
```

- The **`android:hint`** attribute displays a faint grey text telling the user what to enter in the box.

- The **`android:inputType`** attribute restricts the type of input into the box. Read the documentation on the allowable inputs.

Refactoring

Refactoring is the process of improving your code design without changing its behaviour.

One common task in refactoring is to change variable names. You can do so and update all references to them easily using Android studio.
Select any variable name, right-click → **Refactor** → **Rename**.

Look at the other options available in the menu.

Exceptions

An **exception object** is thrown during events that prevents execution from continuing normally.

You handle these exceptions by putting code in a **`try-catch`** block.

Run the code below and you will see that an **`ArithmeticException`** is caught.

`ArithmeticException` is

- A subclass of **`RuntimeException`**, such exceptions are usually thrown by the JVM
- An **unchecked exception** - the compiler *does not* force you to put the code in a try-catch block (another such exception is **`NumberFormatException`**)

```
public class ExceptionsExample {

   public static void main(String[] args){

       try{
           int a = quotientInt(5,0);
       }catch(ArithmeticException ex){
           ex.printStackTrace();
       }
   }

   public static int quotientInt(int a, int b){
       return a / b;
   }
   //write quotientDouble here later
}
```

Dividing a floating point number by zero does not cause an exception to be thrown.

- Add the following method to the class above
- Call it in the try-catch block with b = 0. Is the catch block activated?
- Modify it such that it throws an ArithmeticException if b = 0.

```
public static double quotientDouble(double a, double b){
    return a/b;
}
```

Static Factory Method

A **static factory method** is a static method in a class definition
that returns an instance of that class. (*Attention: this is not the factory design pattern*).

You can overload your constructor to initialize your class with different states,
but you are constrained by Java to have the same name for all constructors.

On the other hand, you can give your static factory method meaningful names to describe what you are doing.

The constructor can be declared private, in which case your class can only be instantiated by calling the static factory methods. Recall that in the singleton design pattern, there is one static method.

```
public class Tea {

   private boolean sugar;
   private boolean milk;

   Tea(boolean sugar, boolean milk){
       this.sugar = sugar;
       this.milk = milk;
   }

   public static Tea teh(){
       return new Tea(true, true);
   }

   public static Tea tehkosong(){
       return new Tea(false, true );
   }
}
```

Hence, you invoke the static factory method like this:

```
Tea tea = Tea.tehkosong();
```

Toasts

You might have seen a message on an Android phone that disappears after a while.
That is known as a **toast**.
Usually, toasts are displayed to notify users of an event occuring.

The code recipe of a toast is as follows.

- First, call the **static factory method** **`makeText()`** of the toast class and give its required inputs:
    - **`Context`** object.
      A **`Context`** is a super-class of **`AppCompatActivity`**
      (Refer to the AppCompatActivity docs)
      Specifying the context here just means to say on which Activity will your toast be seen.
    - Either a resource id or a String.
      Hard-coded strings are not recommended.
    - Duration of toast. You specify one of two static variables in this:
      `Toast.LENGTH_SHORT` or
      `Toast.LENGTH_LONG`

- **`makeText()`** returns a **`Toast`** object. You can then call the **`show()`** instance method to display the toast. (sometimes I forget this).

Here's an example.

```
Toast.makeText(MainActivity.this,
R.string.warning_blank_edit_text,Toast.LENGTH_LONG).show();
```

Note

If you read the documentation,

- The Toast class constructor is actually public. It is used when you want to customize the design of your toast.
- Most of the time, there is no need to, so **`makeText()`** gives you the standard Toast design.

Logcat

The **Logcat** tab of Android studio displays messages as your app runs.
You may display your own messages to the Logcat using the **`Log`** class.

Messages in the Logcat are divided into one of the following levels and you can filter messages by these levels

- **`d`** for debug
- **`w`** for warning
- **`e`** for error
- **`i`** for info

In addition, every message has a tag for added filtering.
Typically, the apps we do are small apps, so we just stick to one of these levels.

A typical statement to print a message as follows:

```
Log.i(TAG, "Empty String");
```

`TAG` is a String variable that is declared final and static. Here, we are specifying that the message uses the **`i`** level.
Having your app print messages to the Logcat is useful for

- viewing data without having to display it on the UI
- Checking and debugging your code

Question. **`i`** is a static method of the Log class. True/False

Explicit Intent

An **intent** is a message object that makes a request to the Android runtime system

- to start another specific activity (an **Explicit Intent**), or
- start some other general component in the phone

  e.g. a Map app (an **Implicit Intent**)

Using an intent, you are also able to pass data between the components.

This section is about **Explicit Intents**.

No Data being passed

If you are not passing data between activities, a typical explicit intent is written as follows using the Intent class.

```
Intent intent = new Intent(MainActivity.this, SubActivity.class);
startActivity(intent);
```

The constructor of the intent object takes in two inputs

- **`Context`** object specifying the current activity
- **`Class`** object specifying the activity to be started

The intent is then launched by invoking the **`startActivity()`** instance method.

You do not need to write any code in the receiving activity.

Data to pass

If there is data to be passed, we use the **`putExtra()`** method of the intent object as well. Data is stored as key-value pairs.

Step 1. In **MainActivity**:

```
Intent intent = new Intent(MainActivity.this, SubActivity.class);
intent.putExtra(KEY,value);
startActivity(intent);
```

The **`putExtra()`** method takes in two inputs

- A final string variable that acts as the key
- The data that is to be stored

Step 2. In **SubActivity**, you then need to obtain the intent object using **`getIntent()`** and retrieve the data using the key using one of the **`get`** methods attached to the intent object. Hence:

```
Intent intent = getIntent();
double value = intent.getDoubleExtra(MainActivity.KEY,
defaultValue);
```

You invoke the appropriate method according to the data type that you want to receive.

In this case, we want to receive a double object, so we invoke **`getDoubleExtra()`**

This method has two inputs

- The key to retrieve the value
- The default value when there is nothing to be retrieved

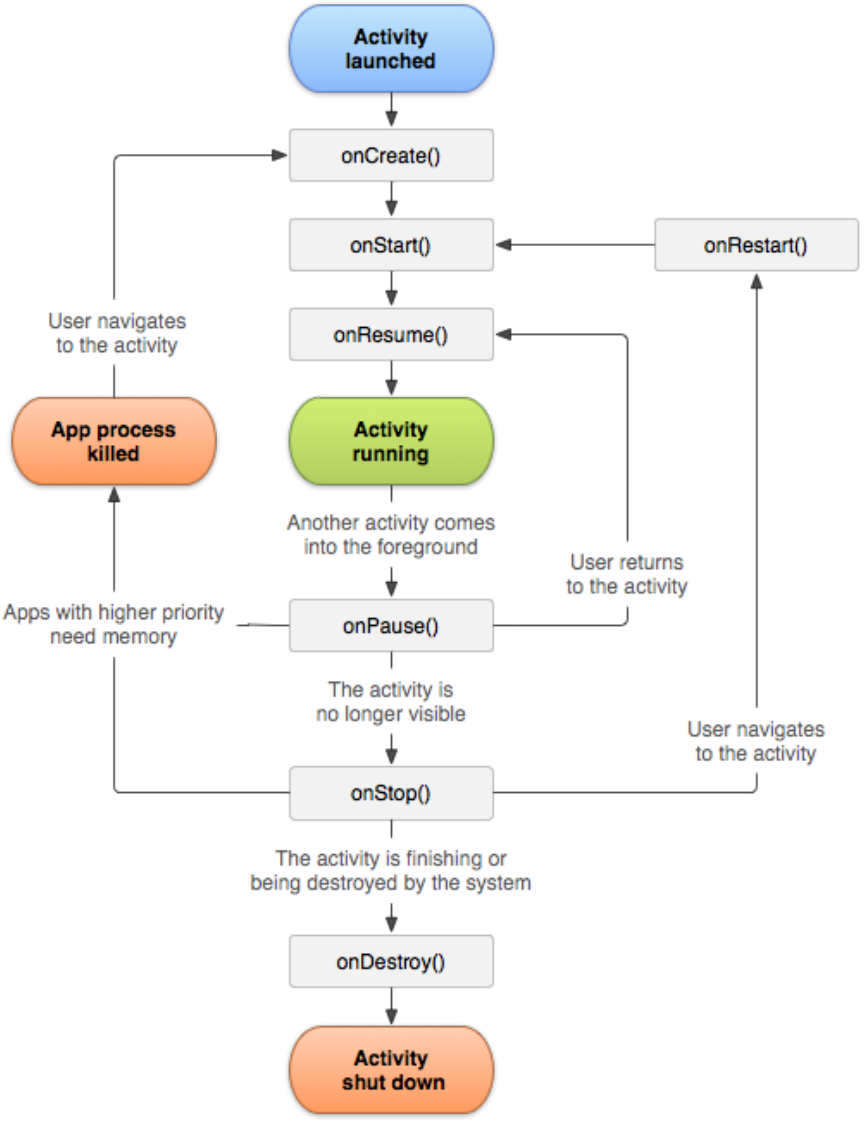
Android Activity Lifecycle

So far, we have been writing code in **`onCreate(),`** which is called when the activity starts. When interacting with your app, the user may find himself doing some of the following actions

- Access a different activity
- Rotate the screen
- Access a new app
- Closing the app

This changes the state of the current activity.

In doing so, other callbacks are executed. The callbacks are summarized in the following diagram of the **Android Activity Lifecycle**.



Implement the seven callbacks in your app, and write a logcat message in each of them.

- Which methods are called as the activity starts?
- Which methods are called as you navigate from one activity to another?
- Which methods are called as you rotate the screen?

The next callback that we will write code in is **`onPause()`**;
This is typically done when you want to carry out the following tasks before your activity is destroyed:

- Saving data
- Stopping a timer

Data Persistence with Shared Preferences

As your user interacts with your app, he or she may

- Close the app and restart it
- Rotate the screen (if you allow the screen to be rotated)

In such situations, data entered by your user is not stored.

There are many ways of storing data, which is called **Data Persistence**.
One way of enabling data persistence is through the **`SharedPreferences`** interface.
Information you would like to store is done using **key-value pair**s.
The code recipe is as follows.

1. Declare the filename of your **`SharedPreferences`** object as a final string instance variable. Also, declare a final string variable as a key.
2. In **`onCreate()`**, get an instance of the **`SharedPreferences`** object.
3. In **`onPause()`**, get an instance of the **`SharedPreferences.Editor`** object and store your key-value pairs. Commit your changes using apply.
4. In **`onCreate()`**, retrieve your data using the key. Don't forget to also assign a default value for the situation when no data is stored.

```
private final String sharedPrefFile =
"com.example.android.mainsharedprefs";
public static final String KEY = "MyKey";
SharedPreferences mPreferences;

@Override
protected void onCreate(Bundle savedInstanceState) {
   //other code not shown
   mPreferences = getSharedPreferences(sharedPrefFile,
MODE_PRIVATE);
   String Rate_text = mPreferences.getString(KEY,defaultValue);
}
@Override
protected void onPause() {
   super.onPause();
   SharedPreferences.Editor preferencesEditor =
mPreferences.edit();
   preferencesEditor.putString(KEY, value);
   preferencesEditor.apply();
}
```

Options Menu

When you start a new Android studio project, one option is the **Basic Activity** template. In this template, code for the following UI elements are automatically provided

- **Options menu**
- **Floating Action Bar** (not discussed in this lesson, will talk about it in Lesson 4)

The **Options menu** is a one-stop location for your user to navigate between the activities of the app.
You should see some additional xml tags in the xml files that specify your layout.
You should see the following lines of code in your MainActivity.java:

In **`onCreate()`**, the following code makes the toolbar containing the options menu appear. Don't delete this code.

```
Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
setSupportActionBar(toolbar);
```

The **`onCreateOptionsMenu()`** method is added to inflate the menu layout and make it appear in the toolbar. Usually, you do not need to add code to this method. The xml layout file is found in **res/menu/menu_main.xml .**

The **`onOptionsItemSelected()`** method is also added. This is where you need to add code to specify what happens when each menu item is clicked. You do this after modifying **menu_main.xml**.

Hence, here are the steps in customizing the options menu.

- add your menu items in **res/menu/menu_main.xml** and remember to give each item a unique ID
- Modify **`onOptionsItemSelected()`** to specify what happens when each menu item is clicked. Very often, you would need to write an intent to bring your user to the other activities in your app.

Builder Design Pattern

When we discussed the Tea class, we saw how static factory methods can be used.

Let's expand the tea class to four options.

```
public class TeaTwo {
   private boolean sugar;
   private boolean milk;
   private boolean ice;
   private boolean toGo;
   //code not shown
}
```

If you were to write a constructor or static factory method for each possible combination of options, how many constructors would you have to write?

We may solve this problem by introducing a **static nested class**, usually called a **builder class** that has

- methods to allow the user to specify the options one by one
- One method that returns the actual object

You could choose to make the constructor private, hence the only way to instantiate your object would be to use the builder class.

```
public class TeaTwo {
   private boolean sugar;
   private boolean milk;

   TeaTwo(TeaBuilder teaBuilder){
       this.sugar = teaBuilder.sugar;
       this.milk = teaBuilder.milk;
   }

   static class TeaBuilder{
       private boolean sugar;
       private boolean milk;

       TeaBuilder(){}

       public TeaBuilder setSugar(boolean sugar){
           this.sugar = sugar;
           return this; }

       public TeaBuilder setMilk(boolean milk){
           this.milk = milk;
           return this;}

       public TeaTwo build(){
           return new TeaTwo(this); }
   }
}
```

The builder is then used as follows:

```
TeaTwo teaTwo = new
TeaTwo.TeaBuilder().setSugar(true).setMilk(true).build();
```

Universal Resource Indicators

URI stands for Universal Resource Identifier, which is a string of characters used to identify a resource. Here are some examples:

Absolute URIs specify a scheme e.g.

- A document on the internet: **`http://www.google.com`**
- A file on your computer: **`file:/Users/Macintosh/Downloads/url.html`**
- A geographic location: **`geo:0.0?q=test`**
- An email: `mailto:` **`test@sutd.edu.sg`**

Hierachical URIs have a slash character after the scheme and can be parsed as follows:
[*scheme*`:`][authority*][*path*][*`?`*query*][*`#`*fragment*//*]

Opaque URIs do not have a slash characters and can be parsed as follows:
[scheme :][opaque part][? Query]

To show a location in a maps app on your phone, you would need to

- Specify the geo URI correctly
- Execute an implicit Intent

Implicit Intents

Recall that in an explicit intent, you specify exactly which activity your user is brought to.

A more general way of using intents is to use an Implicit Intent.
Rather than a specific Activity, In an implicit intent, you specify the type of action you want, provide just enough information and let the android run-time decide.

To illustrate, the code recipe for launching a Map App is given below.

Step 1. You build the URI to specify the location that you want your map app to be. Using this builder helps to avoid errors when hardcoding a URI string.

```
String location = getString(R.string.default_location);
Uri.Builder builder = new Uri.Builder();
builder.scheme("geo").opaquePart("0.0").appendQueryParameter("q",locatio
n);
Uri geoLocation = builder.build();
```

Step 2. You specify the implicit intent by:

- Specify the general action, in this case it is to view data, hence **`Intent.ACTION_VIEW`** is passed to the constructor
- Specify the data that you wish to view, in this case it is the location URI that you build in Step 1.

```
Intent intent = new Intent(Intent.ACTION_VIEW);
intent.setData(geoLocation);
```

Step 3. Check that the intent is able to be carried out before calling startActivity().

```
if( intent.resolveActivity(getPackageManager()) != null){
      startActivity(intent);
}
```

Other intents

Other common intents, eg. opening the camera, are specified here:
https://developer.android.com/guide/components/intents-common

Unit Testing with JUnit4

Testing your app is one of the phases in development. We'll describe two ways of testing:

- Unit testing
- Instrumented Testing (next section).

A **unit** is the smallest component that can be tested in your software, usually it is a method in a class. **Unit testing** thus ensures that each of these components behave as designed.

Why do unit testing?

- Unit testing validates that your software works,
  even in the face of continual changes in your code
- Writing code for unit testing also forces your program to be modular

In the context of an android app,

- Testing the parts that don't involve the UI **(easy)**: **unit testing**
- Testing the parts that involve the UI (hard): **instrumented testing**

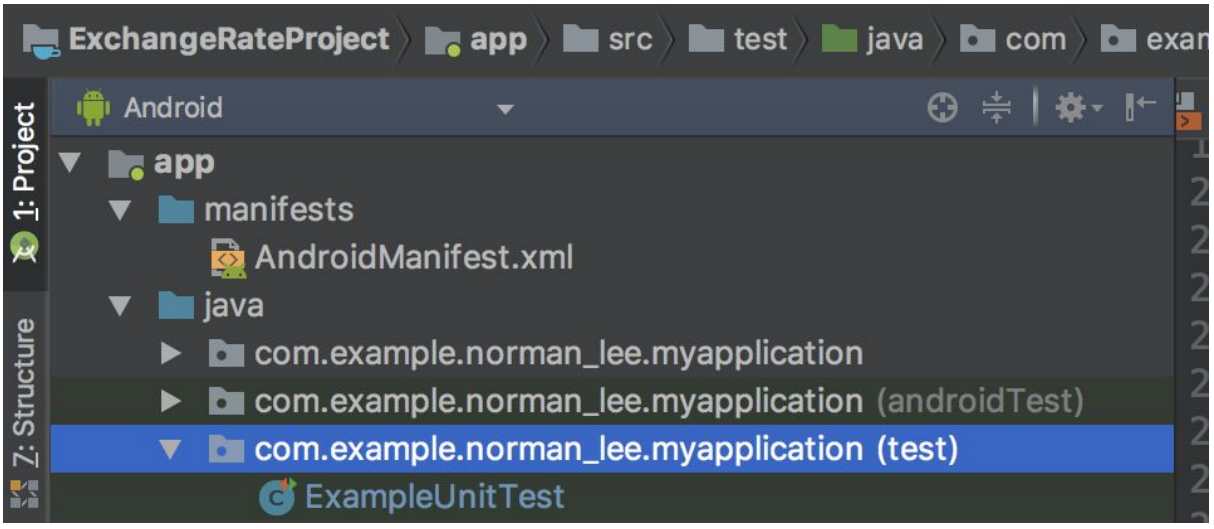
Hence, it is good programming practice to
separate the parts of your code that involve the UI and those that do not.

JUnit4 is a commonly-used open-source framework to conduct unit testing.
To conduct unit testing, you write your tests in a **test class**.

Android studio automatically generates the test class for you.

Where and how to run unit tests

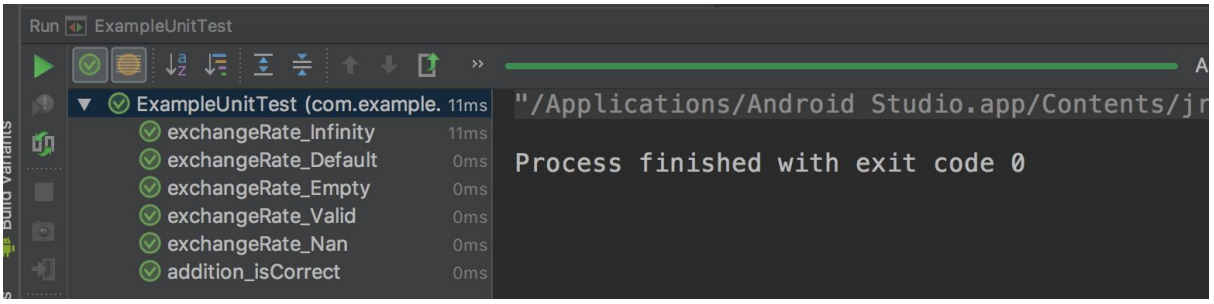
The test class is found in the folder marked **(test)**



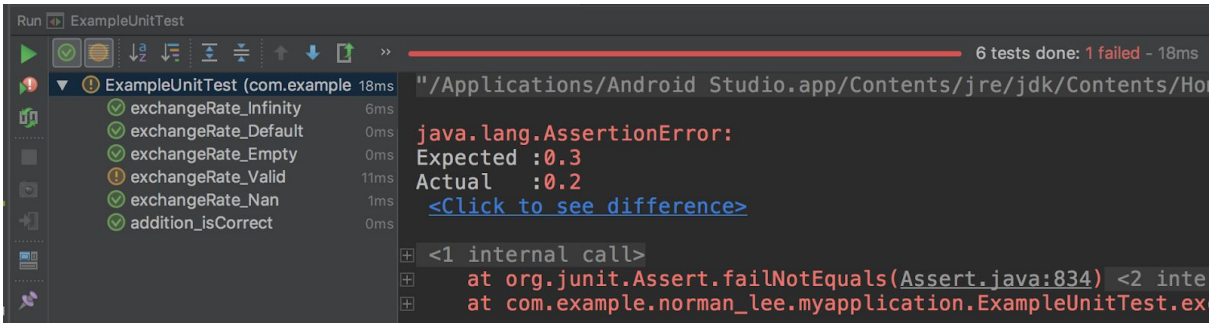
The default file generated contains one trivial unit test, and you can write more.

To run the tests that you have written, right click on **ExampleUnitTest** → **Run 'ExampleUnitTest**'.

If you have passed all the tests, you should see that all the tests you have written have a green tick beside it.



If you failed one of the tests, this is what you will see:



For each unit test,

- you write a method that returns void with the **`@Test`** annotation
- Within this method, use the **`assertEquals()`** method to compare the actual object and the expected object

```
@Test
public void addition_isCorrect() {
   assertEquals(expected, actual);
}
```

There are other assert methods that are available. Take the time to explore them.

Example 1. In the following unit test we want to compare if the function returns a double value of 2.95

```
@Test
public void exchangeRate_Default(){
   assertEquals(2.95, ExchangeRate.calculateExchangeRate(),0.001);
}
```

Example 2. In the following unit test, we want to see if the function throws an **`ArithmeticException`**

```
@Test(expected= ArithmeticException.class)
public void exchangeRate_Infinity() {
   ExchangeRate.calculateExchangeRate("0.0","1.0");
}
```

Instrumented Testing With Espresso

When your app is completed, you can test your app by conducting **user acceptance testing** by giving it to a user and asking him/or her to carry out specific tasks.
If done early and often enough, this can help you spot problems in your app.

In the course of your programming, it is often helpful to automate such testing. Changes in code within a large codebase mean that it is hard to ensure that your app behaves in the same way even with changes. This is where **instrumented testing** will help.

Instrumented testing automates the process of clicking through the parts of your UI and helps to ensure that the user-interaction is as what is intended.

The test class is found inside the (androidTest) folder. Connect your phone or get the emulator ready and you run the instrumented tests in the same way as the unit tests.

As an introduction, there are two kinds of instrumented tests.

- Ensure that your widgets have certain properties.
- Carry out some action on your widgets

Often you have to combine the two.

Ensure that your widgets have certain properties

```
onView ( ViewMatcher  ).check( ViewAssertion  )
```

ViewMatchers look for particular widgets in your UI.

There are two ways to do it

- Match a particular ID: **withID(...)**
- Match a particular text: **withText(*String*)**

ViewAssertions check your widget for certain properties.

Some possible ViewAssertions are

- Check that a certain property exists eg.: **matches(isDisplayed())**
- Check that a certain text exists : **matches(withText(String))**
- Check that a certain view does not exist: **doesNotExist()**

Example 1.

The following tests checks that the View with id editTextValue is displayed on the screen.

```
@Test
public void mainLayoutCorrect(){
 onView(withId(R.id.editTextValue)).check(matches(isDisplayed()));
}
```

Carry out some action on your widgets

```
onView ( ViewMatcher  ).perform( ViewAction  )
```

ViewActions check your widget for certain properties.

Some possible ViewActions are

- Click the widget.: **`click()`**
- Double-click the widget: **`doubleClick()`**
- Long-click the widget: **`longClick()`**
- Clear the text: **`clearText()`**
- Replace the text of the widget: **`replaceText(String)`**

Example 2.

```
@Test
public void exchangeRateZero(){
   onView(withId(R.id.buttonSetExchangeRate)).perform(click());
}
```

For reference, here are some links for further reading

- More on Android testing
  https://developer.android.com/training/testing/
- More on espresso testing
  https://developer.android.com/training/testing/ui-testing/espresso-testing

Part 1 - EditText, Logcat, Toasts

Objective Of Part 1

You will write code so that the exchange rate will be calculated using the default exchange rate of 2.95.

- The user will enter the value in the EditText widget.
- The code will retrieve the user input, perform the conversion and display the result.
- If the user input is blank and the button is clicked, then you would display a toast and a logcat message.

Get Ready (If you haven't done this before coming to class)

1. Download the starter code from eDimension
2. Open the project in android studio.
3. Try to run the app on your phone or emulator.
   If you encounter problems, try **Build → Rebuild Project** or **Clean Project**
4. You'll find that the buttons are not responsive, but you can click on the three dots.
5. Examine the **res/layout** folder and note where the layout files are stored and how they are connected together.
6. Answer the following questions:
   c. How many activities does the app contain?
   d. Currently, how many activities can you see in the app?

TODO 2.1 Use findViewById to get references to the widgets in the layout

You'll need references to widgets with the following IDs:

- **`editTextValue`**
- **`buttonConvert`**
- **`textViewExchangeRate`**

Using **`findViewById()`**, assign them to the instance variables with similar names.
Check that the EditText widget is constrained to numerical inputs only.
If not, add it in.

TODO 2.2 Assign a default exchange rate of 2.95 to the textView

We are going to allow the user to specify the exchange rate in the next lesson, but for now we will hardcode the exchange rate. How you would like to do this is up to you!

TODO 2.3 Set up setOnClickListener for the Convert Button

You would have done this in Lesson 1, so recall what you have done there.
Don't forget to use Android studio's code completion feature.

TODO 2.4 Display a Toast & Logcat message if the editTextValue widget contains an empty string

TODO 2.5 If not, calculate the units of B with the exchange rate and display

First you would have to retrieve the text of the edit text widget by calling its **`getText().toString()`** method. Then you are ready to do the checking using an if/else statement.

Part 2 - Android Manifest, Explicit Intents

Objective of Part 2

Mobile apps typically contain more than one Activity.
You will write code to allow the user to enter the exchange rate information in a different activity called **SubActivity**. This means that you will have to accomplish the following tasks:

- Specify that your app has more than one activity in the **Android manifest**
- Bring the user from **MainActivity** to **SubActivity** using an **Explicit Intent**
- When the user keys in the exchange rate, write code to handle bad or unintended inputs. My suggested implementation is to use **Exceptions** to recall this Java concept.
- Bring the user from **SubActivity** back to **MainActivity** and pass the data back

TODO 3.1 Modify the Android Manifest to specify that the parent of SubActivity is MainActivity

Ensure that you see the following tag inside the Android Manifest

```
<activity android:name=".SubActivity"
   android:parentActivityName=".MainActivity"
   android:label="@string/sub_activity_name"></activity>
```

In MainActivity …

TODO 3.2 - 3.4 When the user clicks Set Exchange Rate button, an Explicit Intent is carried out to bring the user to SubActivity

Start the process of allowing the user to key

- Get a reference to the Set Exchange Rate button
- Set up setOnClickListener
- Write an **Explicit Intent** to bring the user from MainActivity to SubActivity

In SubActivity …

TODO 3.5 - 3.10 get user inputs, calculate the exchange rate and give the data back to MainActivity via an Explicit Intent

By completing these TODOs, you achieve the task of getting the user inputs, calculating the exchange rate and bringing the user back to MainActivity

- Get references to the editText widgets
- Get a reference to the Back To Calculator Button
- Set up setOnClickListener
- Obtain the values stored in the editTextWidgets
- Calculate the exchange rate
- Set up an explicit intent and pass the exchange rate back to MainActivity

The code for all this is given in the explanation sections, you just need to modify accordingly.

TODO 3.11 - 3.12 Consider bad inputs

You do not want your app to crash and restart because the user has given bad inputs. This will annoy the user. Hence you should

- Decide how you are going to handle a divide-by-zero situation
- Decide how you are going to handle a situation when the editText widgets are empty

My own implementation is to have a separate class called **`ExchangeRate`** that contains a static method **`calculateExchangeRate()`**.
It does the calculation and throws

- **`ArithmeticException`** in a divide-by-zero situation
- **`NumberFormatException`** if the editText widgets have an empty string

Here's the method stump.

```
public class ExchangeRate {

   public static double calculateExchangeRate(String A, String B)
           throws NumberFormatException, ArithmeticException {
   }
}
```

Please follow this way as I'm using it to illustrate unit testing in Part 4. Of course, since these are unchecked exceptions, you could handle these situations using if/else statements.

Back in MainActivity …

TODO 3.13 Get the intent and retrieve the exchange rate passed to it

Part 3 - Options Menu, Activity Lifecycle, SharedPreferences

Objective of Part 3

User Navigation. If your app has more than one Activity, you will need a way for the user to navigate among them.
You have many options, some of which require advanced concepts. One easy way is to implement the **Options Menu**.

Activity Lifecycle. For each activity, the android runtime actually makes use of several callbacks in what is known as the **Activity Lifecycle**. In short, when a user navigates to another activity, the current activity is actually destroyed.

Data Persistence. Because of the activity lifecycle, data entered by the user is not automatically saved. Such data includes user input and user preferences. Hence, you need to deliberately store the app's data, which is called **Data Persistence**. There are several ways, one easy way is to use the `SharedPreferences` class.

TODO 4.1 - 4.2 Implement a new menu item in the options menu

You are reminded that an options menu can be generated by using the **Basic Activity** template.

In this set of tasks, you are going to have a menu item that brings your user to SubActivity.

- Got to **res/menu/menu_main.xml** and add a menu item *Set Exchange Rate*
- Go to **MainActivity** and in `onOptionsItemSelected()`, add a new if-statement and code accordingly

TODO 4.3 - 4.4 Get acquainted with the methods in the Activity lifecycle

For this set of tasks

- override the methods in the Android Activity Lifecycle in **`MainActivity`**
- for each of them, write a suitable string to display in the Logcat

Try out the following actions on the phone

- Rotating the phone
- Closing the app and opening it again

Answer the following question:

- What is the sequence of Lifecycle methods calls?
- What does this tell you about what the Android runtime does when your phone screen is rotated?
- Is the information entered by the user retained?

TODO 4.5 - 4.8 Store the exchange rate previously calculated

It would be annoying for the user to have to key in the exchange rate everytime he/she starts the app.

We use the **`SharedPreferences`** class together with an understanding of the Activity lifecycle callbacks.

In **`onPause()`**:

- get a reference to the **`SharedPreferences.Editor`** object
- store the exchange rate using the putString method with a key

Questions for you to consider:

- What type of class is the **`SharedPreferences.Editor`** class?
- Why do we write this code in the **`onPause()`** callback?

In **`onCreate()`**:

- Get a reference to the sharedPreferences object
- Retrieve the value using the key, and set a default when there is none

TODO 4.9 - 4.10 [Implement yourself] Implement saving to shared preferences for the contents of the EditText widget

Part 4 - Implicit Intents, Instrumented Testing

Objective of Part 4

Implicit Intents. You may want your app to start a service or another app within the phone. One common use case is for your app to access the phone's camera or the image gallery. This is done via an **Implicit Intent**.

Unit Testing. **Unit Test**s written in your code validates that your code performs as it should. You will write some simple unit tests for the calculateExchangeRate() method.

Instrumented Testing. Tests written in your code validates that your app performs as designed. While it is good to test with actual users, automated testing can help to speed up the development lifecycle. This is done with **Instrumented Testing** using the Espresso framework.

TODO 5.1 - 5.3 Add an Implicit intent to a map app on the phone in the Options menu

- Add a new menu item by modifying **res/menu/menu_main.xml**
- In **`onOptionsItemSelected()`**, add a new if-statement
- code the Uri object and set up the implicit intent

Compare the constructor for an Explicit Intent and an Implicit Intent. What's the difference?

TODO 5.4 Write the following unit tests in the test folder

Call **`calculateExchangeRate()`** with the following unit tests. You may add your own.

Input A	Input B	Expected Outcome
“5”	“1”	Returns 0.2
“100”	“4.45”	Returns 0.0445
“” (Empty string)	“0.0”	NumberFormatException is thrown
“0.0”	“0.0”	ArithmeticException is thrown
“0.0”	“1.0”	ArithmeticException is thrown

TODO 5.5 - 5.9 Write the following instrumented tests in the androidTest folder

- Check that the layout of **`MainActivity`** has the required widgets
- For 5 units of A buys 1 unit of B, enter these values in **`SubActivity`** and check that the result 0.2 is displayed in **`MainActivity`**
- If 0 is entered in **`editTextSubValueA`** in **`SubActivity`** check that **`textViewExchangeRate`** is not displayed when button is clicked (this is one way of checking that the explicit intent is not carried out)
- Write a class called **`CheckEditText`** to check that an editText widget is numberInput. Check that the EditText Widgets in this app accept numerical inputs only
- [On your own] check that if the EditText widgets in SubActivity are blank and button is clicked, MainActivity is not displayed.

Lesson 3 - Comic App

Objectives

- Describe how the AsyncTasks class is used
- Query an API using the AsyncTasks class
- State the limitations of the AsyncTasks class
- Modify the android Manifest to set permissions and fix the orientation
- Build an URL object
- Download JSON data given a URL of an API call
- Parse JSON data using the JSONObject class
- Download an Image file given a URL
- Display the information in the UI

Introduction

In this lesson, we are going to download data from the xkcd.com web API and display the comic image in our app.

The Android/Java that you need to know

Abstract Classes

A class declared as **`abstract`** cannot be instantiated.

Typically, **abstract classes** have **abstract methods**, which you will recall is a **method signature** that includes the keyword abstract.

Why do we do this?

- **What is to be done** is specified by the abstract methods (and interfaces too)
- **How it is to be done** is specified by their implementation

Because abstract classes can behave as a datatype, it promotes **code reuse**.
You may write methods that take in an abstract class as an input, and you are guaranteed that objects passed to it will have the abstract method implemented.
(This remark also applies to interfaces.)

To use an abstract class, sub-class it and implement any abstract methods.

In the example below, complete the class **`Tiger`**.

```
public class TestAbstract {

    public static void main(String[] args){
        Feline tora = new Tiger("Tiger","Sumatran Tiger");
        makeSound(tora);
    }

    public static void makeSound(Feline feline){
        feline.sound();
    }
}

abstract class Feline {

    private String name;
    private String breed;

    public Feline(String name, String breed) {
        this.name = name;
        this.breed = breed;
    }

    public String getName() {
        return name;
    }

    public String getBreed() {
        return breed;
    }

    public abstract void sound();
}

class Tiger extends Feline{

     //write the constructor and complete the class
}
```

Template Method Design Pattern

One application of an abstract class is in the **template method design pattern**.
In this design pattern, there is an algorithm with a fixed structure, but the implementation of some steps are left to the subclasses. The following example is taken from "Heads First Design Patterns - A brain-friendly guide".

We have a fixed way of brewing caffeine beverages (Coffee, Tea etc), but you'll agree that how you brew it and what condiments to add depends on the beverage.
In the example below,

- The algorithm to make the caffeine beverage, **`prepareRecipe()`** is declared **`final`** to prevent subclasses from altering the algorithm.
- The steps in the algorithm that are common to all beverages are implemented.
- The steps that can vary are declared **`abstract`**.

```
public abstract class CaffeineBeverage {

    final void prepareRecipe(){
        boilWater();
        brew();
        addCondiments();
        pourInCup();
    }

    abstract void brew();

    abstract void addCondiments();

    void boilWater(){
        System.out.println("Boiling Water");
    }

    void pourInCup(){
        System.out.println("Pouring in Cup");
    }
}
```

We may then have our sub-classes of **`CaffeineBeverage`**.

```
class GourmetCoffee extends CaffeineBeverage{
    @Override void brew() {
        System.out.println("Put in Coffee Maker");
    }

    @Override void addCondiments() {
        System.out.println("Adding nothing, because GourmetCoffee");
    }
}
```

We may then brew our coffee:

```
CaffeineBeverage caffeineBeverage = new GourmetCoffee();
caffeineBeverage.prepareRecipe();
```

Generic Classes & Interfaces

You would have used the ArrayList class in the following way::

```
ArrayList<Integer> arrayList = new ArrayList<>();
```

Recall that this design is for **type safety** - an operation cannot be performed on an object unless it is valid for that object.
This means that the code below would cause a compile-time error.

```
arrayList.add("abc");
```

You would also have worked with the Comparable interface:

```
public class Octagon implements Comparable<Octagon>{
    //code not shown
    @Override
    public int compareTo(Octagon octagon) {
        //code not shown
    }
}
```

The **`ArrayList`** class is an example of a **Generic class** where a class takes in an object or objects as a parameter and operates on it.
Similarly, the **`Comparable<T>`** interface is an example of a **Generic interface**.

AsyncTasks Abstract Class allows tasks to run in the background

In this app, we are going to query a web API to retrieve data.

The length of time this task will take depends

- on the internet connection,
- file size, etc

While the data is being retrieved, you want the UI to remain responsive and inform the user that the download is still in progress.

This means that the **download should be done on a separate thread from the UI.**

Android provides an abstract class called **`AsyncTask`** to carry out this task.

AsyncTasks is an abstract class with three generic types

```
AsyncTask<Params, Progress, Result >
```

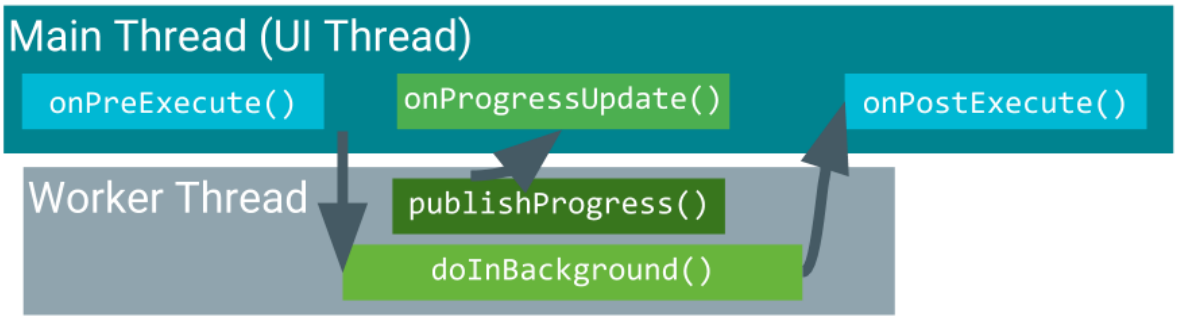
Params - this is the type of the object sent to the background task to begin execution
Progress - this is the type of the object that is reported to the UI while the background task executes
Result - this is the type of the object that is returned when the background task completes

Example. Suppose you want to download an image given a URL, and not report the download progress to the UI, then you would subclass `AsyncTask` as follows.

```
class MyBackgroundTask extends AsyncTask<URL, Void, Bitmap>{
}
```

AsyncTasks has four methods you can override

The diagram below is taken from the Android developer fundamentals.



The four methods that you can override are

`onPreExecute()` - seldom-used. Execute any jobs before the background task begins.

`doInBackground()` - always used. Execute the background task on the worker thread.

`onPostExecute()` - sometimes-used. Execute any jobs once the background task ends.

`onProgressUpdate()` - this method puts information on the UI that is provided by **`publishProgress()`**. This is useful if you have a long download and want to tell your user the progress.

Hence, you override this if

- you have specified the second generic type,
- this allows you to call **`publishProgress()`** in **`doInBackground()`**

How to use AsyncTasks

Step 0. Prevent your activity from changing orientation.

- AsyncTask is easy to use but has a major flaw.
- In the Android Activity Lifecycle, recall that the app layout is destroyed and re-created if the screen is rotated.
- If AsyncTask is running in the background during this process, it will not be able to display the result on the re-created activity.
- Hence, you need to constrain your activity so that it is always in your desired orientation.
- This is done in the android manifest.

Step 1. Write an inner class in MainActivity that extends AsyncTask.

Make the following decisions to help you decide the generic types:

- What information launches the background task?
- What information do I want to give the user as the task proceeds?
- What information does the background task provide?

Step 2. Decide what jobs are to be run in each of the four methods:

- **`doInBackground()`** always carries out the background task e.g. downloading the image data from a URL.
- Decide if you need **`onPreExecute()`** - I have not found a reason to use it so far.
- **`onPostExecute()`** carries out the job to be done after the background task is complete. Suppose the task is to download an image given a URL, this is where you write code to display the image on the UI.
- If you use **`publishProgress()`** within **`doInBackground()`**, decide how you want this information to be displayed on the UI using **`onProgressUpdate()`**

Step 3. Subclass AsyncTask with the parameters and implement the methods.

One possible code stump is shown below. In this code stump, **`onPreExecute`** is not needed. **Look at the method signatures and observe how the generic types are used.**

```
class GetComic extends AsyncTask<URL,String, Bitmap>{

   @Override
   protected Bitmap doInBackground(URL... urls) {

       Bitmap bitmap = null;
       return bitmap;
   }

   @Override
   protected void onProgressUpdate(String... values) {
       super.onProgressUpdate(values);
   }

   @Override
   protected void onPostExecute(Bitmap bitmap) {
       super.onPostExecute(bitmap);
   }
}
```

Step 4. Decide how the user is to execute the background task.

Let's say your user will download the image with a button click. Then you would put the following code in the anonymous class within **`setOnClickListener()`**

```
GetComic getComic = new GetComic();
getComic.execute(url);
```

If you recall the **template method design pattern**, what we have done is

- Override the steps of the algorithm
- Execute the algorithm using the **`execute()`** method

Using the xkcd.com web API

Many websites provide an API for you to access the data. In this lesson, we are going to use the xkcd.com API to access the comics.

This is all the documentation that is provided, in the **About** section:

> **Is there an interface for automated systems to access comics and metadata?**
>
> Yes. You can get comics through the JSON interface, at URLs like http://xkcd.com/info.0.json (current comic) and http://xkcd.com/614/info.0.json (comic #614).

Hence, this is the easiest web API that I could find and we are going to use it.

Building a URL

A URL is a URI that refers to a particular website. The parts of a URL are as follows

Component	Example
Scheme	`https`
Authority	`xkcd.com`
Path	`info.0.json` or `614/info.0.json`

To prevent parsing errors you may use the **`Uri`** builder before creating a **`URL`** object.

Using such a builder helps to eliminate coding errors.

The string constants may be placed in the **`strings.xml`** file instead.

After you create the **`Uri`** object, use it to make a **`URL`** object.

The constructor of the **`URL`** class throws a **`MalformedURLException`**.

As this is a **checked exception**, you have to put the code in a **`try-catch`** block.

```
final String scheme = "https";
final String authority = "xkcd.com";
final String back = "info.0.json";
URL url = null;

Uri.Builder builder = new Uri.Builder();
builder.scheme(scheme)
       .authority(authority)
       .appendPath(back);

Uri uri = builder.build();

try{
   url = new URL(uri.toString());
}catch(MalformedURLException ex) {
   Log.i(TAG, "malformed URL: " + url.toString());
}
```

Connecting to the internet

With a URL object, you are ready to download data from the internet.
In the starter code, you are provided with a **`Utils`** class.
There are three static methods for you to use.
You need not worry about how they are implemented.
You should see that some of these methods print data to the logcat.

The following methods take in a URL object and return their respective objects:

```
public static String getJson(URL url)
public static Bitmap getBitmap(URL url)
```

The following method takes in a Context object and checks if a network connection is available. True is returned if a network connection is available, False if is not.
`Context` is the superclass of **`AppCompatActivity`**.

```
public static boolean isNetworkAvailable(Context context)
```

If you use the android Emulator, you may experience difficulties with checking for an internet connection. Please borrow a phone.

Android Manifest

In order to

- Constrain the activity orientation
- Access the internet

The android manifest needs some additional information.

They are shown below.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.norman_lee.comicapp">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="ComicApp"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity"
            android:screenOrientation="portrait"
            android:configChanges="orientation|keyboardHidden">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />

</manifest>
```

The JSON format

JSON stands for **J**ava**S**cript **O**bject **N**otation and stores data using key-value pairs.

A sample response from the xkcd API is

```
{"month": "11", "num": 2068, "link": "", "year": "2018", "news": "",
"safe_title": "Election Night", "transcript": "", "alt": "\"Even the
blind\u00e2\u0080\u0094those who are anxious to hear, but are not able
to see\u00e2\u0080\u0094will be taken care of. Immense megaphones have
been constructed and will be in use at The Tribune office and in the
Coliseum. The one at the Coliseum will be operated by a gentleman who
draws $60 a week from Barnum & Bailey's circus for the use of his
voice.\"", "img": "https://imgs.xkcd.com/comics/election_night.png",
"title": "Election Night", "day": "5"}
```

Before you make use of this data, you would have to make sense of it. Online JSON viewers are available to help you.

Parsing JSON data using the JSONObject class

Once you are able to make sense of this data, you are ready to parse it. One way is to use the **`JSONObject`** class.

The **`JSONObject`** constructor takes in a string variable that contains the JSON data.

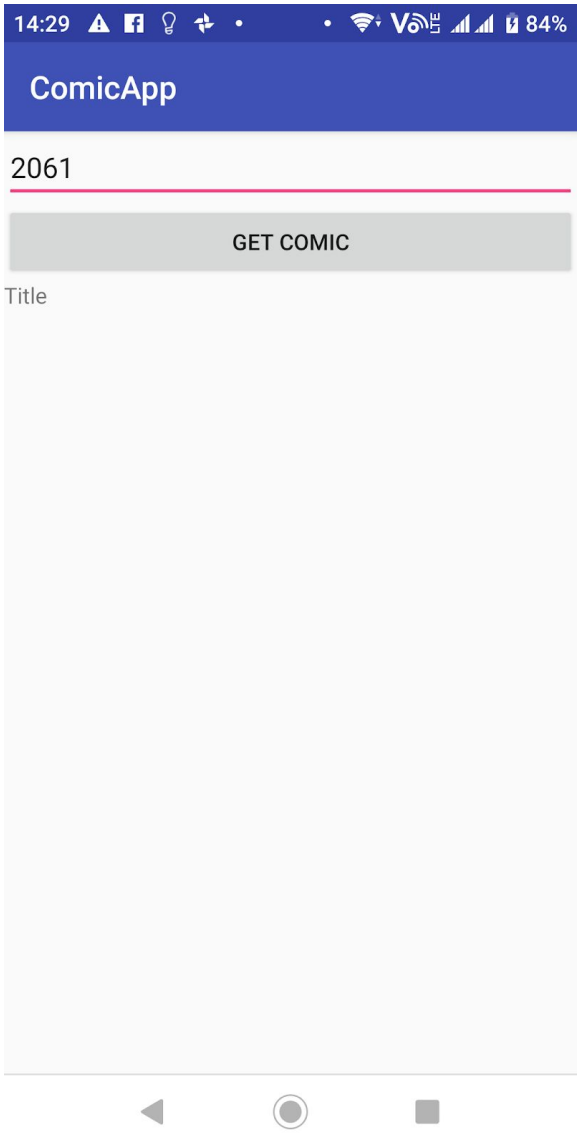
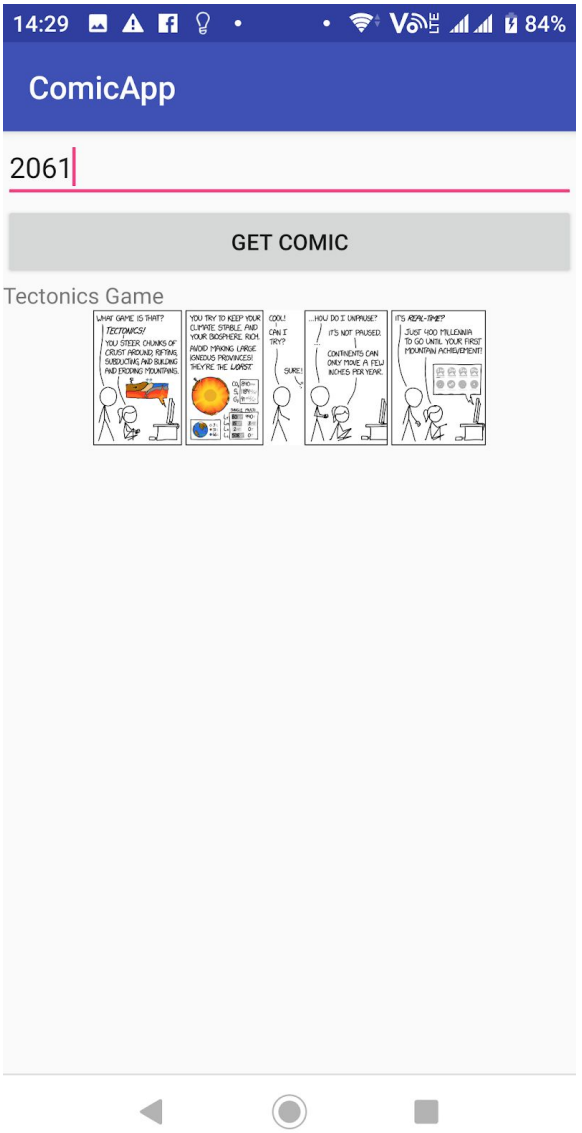
You then retrieve the value using the key and one of the appropriate get methods.

```
JSONObject jsonObject = new JSONObject(json);
String safe_title = jsonObject.getString("safe_title");
```

Another alternative is the GSON library, which many programmers find useful in parsing JSON data. I will leave you to learn it on your own and it won't be tested on it.

Completing Your App

What the app should do

Step 1. A user enters a number from 1 to the latest xkcd comic number and clicks **Get Comic**.	**Step 2.** The comic and its title is displayed.
	

BEFORE YOU BEGIN, download the starter code and compile your app. You may need to Build → Clean Project.

TODO 6.1 Ensure that Android Manifest has permissions for internet and has the orientation fixed

TODO 6.2 - 6.5 When button is clicked, Get the user input and make the xkcd API url

- Get references to widgets
- Set up setOnClickListener for the button
- Retrieve the user input from the EditText
- Set up the xkcd url by completing buildURL

TODO 6.6 - 6.13 Write the GetComic AsyncTasks class to download the JSON file

- In publish progress, write code to update textViewTitle with a string
- In doInBackground, get the JSON Response
- If the JSON response is null, then call publishProgress with an appropriate message and return null
- Create a new JSONObject wtih a try/catch block
- Using getString, Extract the value with key "safe_title" and display it on textViewTitle
- Extract the image url with key "img”
- Create a URL object and put another catch block
- Get the image with the url
- Complete onPostExecute to assign the Bitmap downloaded to imageView

TODO 6.14 Back in setOnClickListener, If network is active, instantiate your AsyncTask class and call the execute method

Lesson 4 - Characters App (Parts 1 & 2)

Objectives of Part 1 & 2

- Describe SQL commands to create a SQLite database table, insert rows into a database table, query a database table and delete a row
- Write a helper class that extends SQLiteOpenHelper to carry out the tasks of creating, inserting, querying and deleting for a SQLite database
- Make use of the helper class to display a random image
- Use explicit intents and implicit intents to write a UI for the user to provide data to insert a row into the database

The Android/Java that you need to know

Recall Lesson 2 & 3

We might get bogged down with specific details in the code, but we should also gain an overview of the Android framework.
For each feature described, state the part of the android framework that you can implement.

Feature	**Framework component**
Bring the user from one Activity to another Activity	
To validate that your app behaves as it should when a user interacts with it	
Bring the user from the app to another app with a specific function	
The series of callbacks as an Activity is created and/or destroyed	
To run long tasks in another thread	
Store data to make it available at another point in time	
To validate that code components (e.g. methods) in your app behaves as it should	

Static Nested Classes

Recall that a class definition can contain **nested classes**.

```
public class OuterClass {
   // code not shown

   class InnerClass{
       //code not shown
   }

}
```

This is typically done when you have classes that logically depend on the outer class and are used together with the outer class.

By declaring a nested class as static, it is known as a **static nested class.**

- It can only access static variables and methods in the outer class.
- It can be instantiated without an instance of the outer class.

A static nested class behaves like a top-level class and is a way to organize classes that are used only by some other classes.
One reason for having a static nested class is to have a model class to store data.

Recall the Singleton design pattern

Recall that the singleton design pattern allows only one instance of a class to exist.

This is done by

- Making the constructor private
- The sole instance is stored in a private static variable
- Using a static factory method to return an instance

```
public class Singleton{

    private static Singleton singleton;

    private Singleton(){
         //any tasks you need to do here
    }

    public Singleton getInstance(){

        if(singleton == null){
            singleton = new Singleton();
        }

        return singleton;
     }

     //other methods in your class

}
```

Getting Used to SQL Commands

A **relational database** (“**SQL database**”) has its data stored in tables. The closest thing you might be familiar with is having data in a spreadsheet.

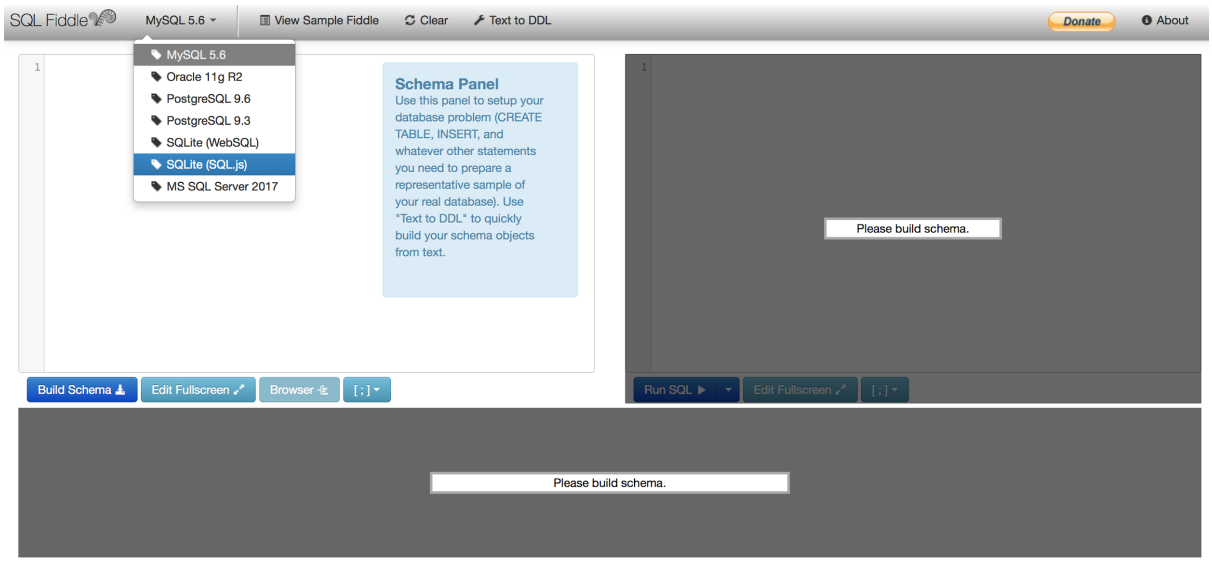
A **non-relational database** (“**noSQL database**”) has its data stored in other ways. One common way is using key-value pairs, which you would have encountered in Google firebase.

In this section, you are going to be introduced to the SQL language to query a **relational database**. We will be writing an android app to query a local SQLite database, which is available for all apps to use.

A relational database will have more than one table. However, to illustrate the essential concepts to you, we will just create one table and query it.

Step 1. Access http://sqlfiddle.com .
If you are a Mac user, you may check if your terminal has sqlite by typing in **`sqlite3`**.

Step 2. Click on **MySQL5.6** and from the dropdown menu, select **SQLite (SQL.js)**.
Ensure that the box beside “Browser” shows a semi-colon.



Step 3. Put in *all* the following commands in the left hand side box and click **Build Schema**. The response from the website might be slow, wait for a while and try again if you are unsuccessful.
The first line creates a table with three columns (what are the column names?) and the subsequent three lines insert rows into the table.

```
CREATE TABLE SpendingRecord (
  _ID INTEGER PRIMARY KEY AUTOINCREMENT,
Amount INTEGER NOT NULL,
Remarks TEXT NOT NULL );

INSERT INTO SpendingRecord (Amount, Remarks)
VALUES (10000, 'Man');

INSERT INTO SpendingRecord (Amount, Remarks)
VALUES (20000, 'Norman');

INSERT INTO SpendingRecord (Amount, Remarks)
VALUES (30000, 'Eric');
```

Let's unpack the command to create a table.

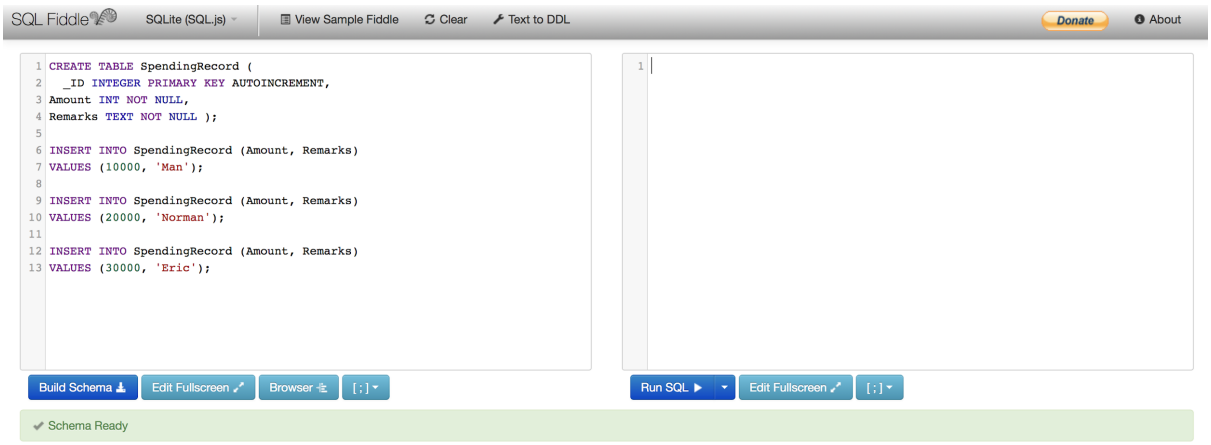
The table name is called **SpendingRecord**. There are three columns in this table, called **_ID**, **Amount** and **Remarks**.

The column called **_ID** is specified as a **PRIMARY KEY** and is of **INTEGER** (integer) data type. A **primary key** is a unique entry that identifies each row in the table. Your student ID is a primary key. The primary key need not be an integer.
AUTOINCREMENT specifies that as each row is added, the primary key increases by one.

Subsequently, the column called **Amount** is specified to be of Integer type and **NOT NULL** specifies that the rows in the column are not allowed to be empty.
Finally, the column called **Remarks** is specified to be of Text data type i.e. it accepts strings.

It is possible for the table to store images. The datatype is called **BLOB**.

Step 4. If you are successful, you should see the following screen. You are ready to key in commands to query your database on the right-hand panel.


SQL Fiddle
SQLite (SQL.js)
View Sample Fiddle
Clear
Text to DDL
Donate
About
CREATE TABLE SpendingRecord (
_ID INTEGER PRIMARY KEY AUTOINCREMENT,
Amount INT NOT NULL,
Remarks TEXT NOT NULL);
INSERT INTO SpendingRecord (Amount, Remarks)
VALUES (10000, 'Man');
INSERT INTO SpendingRecord (Amount, Remarks)
VALUES (20000, 'Norman');
INSERT INTO SpendingRecord (Amount, Remarks)
VALUES (30000, 'Eric');
Build Schema
Edit Fullscreen
Browser
[;]
Run SQL
Edit Fullscreen
[;]
Schema Ready

Step 5. Run some SQL queries.

Here is the command to query the entire table. Put it in the right-hand box and click **Run SQL**.

```
SELECT * FROM SpendingRecord;
```

You should see this appear below:

If you wish to see only a particular column, specify the column name after **SELECT**. :

```
SELECT Remarks FROM SpendingRecord;
```

You may use a “WHERE clause” to query subsets of your table. For example:

Access the row where Remarks contains the text Eric:

```
SELECT * FROM SpendingRecord WHERE Remarks = 'Eric';
```

Access the rows where amount > 15000

```
SELECT * FROM SpendingRecord WHERE Amount > 15000;
```

Access a random row:

```
SELECT * FROM SpendingRecord ORDER BY RANDOM() LIMIT 1;
```

Further reading

Android Developer Fundamentals v2, Concept Reference, Section 10.0

https://google-developer-training.github.io/android-developer-fundamentals-course-concepts-v2/unit-4-saving-user-data/lesson-10-storing-data-with-room/10-0-c-sqlite-primer/10-0-c-sqlite-primer.html

Local SQLite Database & Introduction to SQLiteOpenHelper

We learnt about Data Persistence using **`SharedPreferences`**.
Data is stored using **key-value pairs**.
Hence, if there is a lot of data, using **`SharedPreferences`** will be tedious.
An alternative is a local **SQLite database** on the phone within an android app.

To create and manage a local SQLite database in your android app,
write a class that subclasses **`SQLiteOpenHelper`**.

This class will perform the following tasks

- Create the database - override **`onCreate()`**
- Upgrade the database - override **`onUpgrade()`**

This class can also include methods for the database CRUD queries

- **C**reating records
- **R**eading or Querying records
- **U**pdating records
- **D**eleting records

How to use SQLiteOpenHelper

Constructor

Your constructor should take in a Context object.

Then, call the superclass constructor with

- The same context object
- The table name
- null for the CursorFactory
- The database version (explained below).

An extract from the documentation is shown below.

Summary

Public constructors
`SQLiteOpenHelper(Context context, String name, SQLiteDatabase.CursorFactory factory, int version)` Create a helper object to create, open, and/or manage a database.
`SQLiteOpenHelper(Context context, String name, SQLiteDatabase.CursorFactory factory, int version, DatabaseErrorHandler errorHandler)` Create a helper object to create, open, and/or manage a database.
`SQLiteOpenHelper(Context context, String name, int version, SQLiteDatabase.OpenParams openParams)` Create a helper object to create, open, and/or manage a database.

The **singleton design pattern** should also be used with this class,
as only one instance of the helper should be querying the database at any point in time.

Database Version

Included in the superclass constructor is an integer variable **`version`**
that is usually initialized to 1. This is the database version.
Usually, this is declared as a static variable in the class and passed to the constructor.

You manually increment this number when you make changes to the database e.g.

- After adding/removing columns
- After fixing a mistake in your SQL commands

Incrementing this number triggers a call to **`onUpgrade()`**.

onCreate()

In onCreate, an instance of a **SQLiteDatabase** object is passed to it.

Execute its **execSQL** method by passing to it the SQL command for creating a table.

You may also write code

- to populate this database with initial entries
- to display logcat messages to see when it is executed.

SQL_CREATE_TABLE is a static string variable that stores an SQL command to create the table in the schema that you desire.

```
@Override
public void onCreate(SQLiteDatabase sqLiteDatabase) {
   sqLiteDatabase.execSQL(SQL_CREATE_TABLE);
   fillTable(sqLiteDatabase);
}
```

onUpgrade()

This method is triggered when the database version is incremented.

- The SQL command to delete the table is executed.
- onCreate is then invoked

SQL_DROP_TABLE is a static string variable that stores an SQL command to delete the table.

```
@Override
public void onUpgrade(SQLiteDatabase sqLiteDatabase, int i, int
i1) {
   sqLiteDatabase.execSQL(SQL_DROP_TABLE);
   onCreate(sqLiteDatabase);
}
```

SQLiteDatabase Operations - Querying

You may write methods to query the SQLite database in the helper.

How to start

Begin by calling **`getReadableDatabase()`**, which returns an **`SQLiteDatabase`** object.

Usually the result is stored in an instance variable.

Then execute an SQL query using the **`rawQuery`** method.

The raw query method returns a **`Cursor`** object with the result

```
if (readableDb == null){
   readableDb = getReadableDatabase();
}
String SQLCommand = ; //command is written here
Cursor cursor = readableDb.rawQuery(SQLCommand, null);
```

Using the cursor object

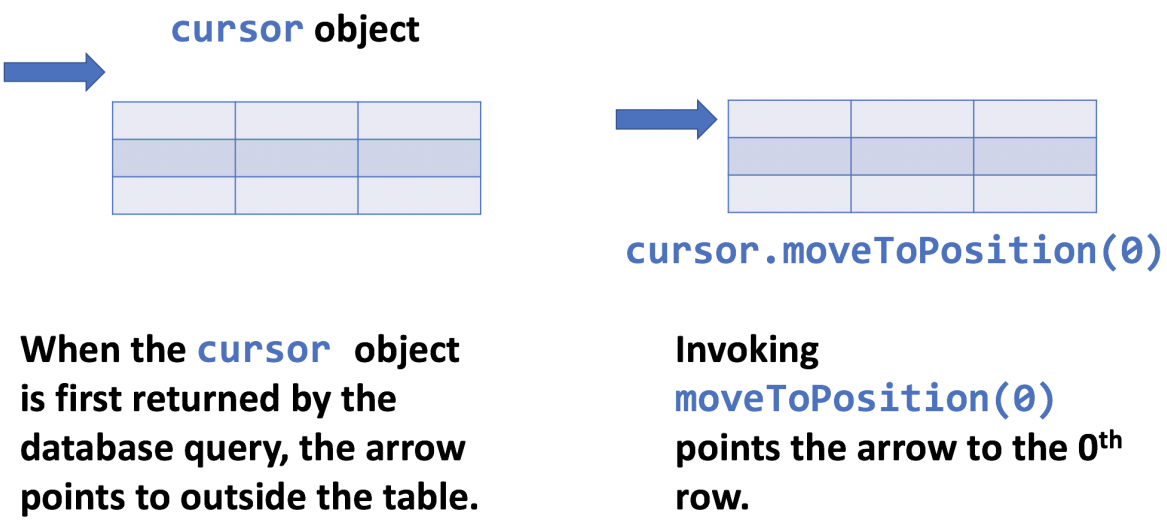
Depending on the SQL query, the number of columns returned could vary. There could also be more than one row.

- The cursor object will point initially to outside of the table,
  you need to move the pointer to the row you want.
  Use the **`moveToPosition()`** method. The first row is position = 0.
- Obtain the column index in the cursor object using the column name.
  In the example below, a static string variable **`COL_NAME`** is used for this.
- Use the column index to retrieve the data.

Repeat the code for each column that you wish to retrieve.

Regardless of the number of columns, **`moveToPosition()`** needs to be called only once per row.

```
cursor.moveToPosition(position);
int nameIndex = cursor.getColumnIndex(COL_NAME);
String name = cursor.getString(nameIndex);
```



SQLiteDatabase Operations - Creating New Rows

To create a new row, you need not run any SQL query.

Android has the following solution:

- Begin by calling **`getWritableDatabase()`**, which returns an **`SQLiteDatabase`** object. Usually the result is stored in an instance variable.
- Create a new **`ContentValues`** object.
- Use the **`put`** method on this object to put the data, with the column name as a key.
- Finally, call the **`insert`** method on the SQLiteDatabase object and provide the table name and the ContentValues object as inputs.

The code sample below shows you how to insert data into a table with one column.

```
 if (writeableDb == null){
   writeableDb = getWritableDatabase();
}
ContentValues cv = new ContentValues();
cv.put(COL_NAME, nameData);
writeableDb.insert(TABLE_NAME,null,cv);
```

SQLiteDatabase Operations - Deleting a Row

To delete a row, you only need the **where-clause** part of the SQL query.

Android has the following solution:

- Begin by calling **`getWritableDatabase()`**, which returns an **`SQLiteDatabase`** object. Usually the result is stored in an instance variable.
- Write your where-clause as a string. Notice what is placed after the equal sign.
- Arguments that replace the question mark are stored in a string array.
- Finally, call the **`delete`** method on the SQLiteDatabase object and provide the table name, where-clause and where-args. This method returns the number of rows deleted.

```
if (writeableDb == null){
   writeableDb = getWritableDatabase();
}

String WHERE_CLAUSE = COL_NAME + " = ?";
String[] WHERE_ARGS = {name};
int rowsDeleted =
writeableDb.delete(TABLE_NAME,WHERE_CLAUSE,WHERE_ARGS);
```

https://guides.codepath.com/android/local-databases-with-sqliteopenhelper

Utility Classes

You will realize that there are several **string constants** that are used throughout the code. These include

- table name and column names
- SQL commands

Placing such information in **static variables**
within **static inner classes declared final** can help to organize it.

As the function of such classes is just to store data,
instantiation ought to be prevented by making the constructor private.

Lastly, **having static inner classes groups related classes together.**

An *excerpt* (due to space constraints) of the such a class in this lesson is shown below.

```
public class CharaContract {

   public static final class CharaEntry implements BaseColumns {
       public static final String TABLE_NAME = "Chara";
       public static final String COL_NAME = "name";
       public static final String COL_DESCRIPTION = "description";
       public static final String COL_FILE = "file";
   }

   public static final class CharaSql {

       public static String SQL_DROP_TABLE = "DROP TABLE IF EXISTS
" + CharaEntry.TABLE_NAME;
       public static String SQL_QUERY_ALL_ROWS = "SELECT * FROM "
+ CharaEntry.TABLE_NAME;
   }
}
```

Examples of how variables are accessed as follows. Notice how the use of inner classes can add to making the names meaningful.

```
CharaContract.CharaEntry.TABLE_NAME
CharaContract.CharaSql.SQL_DROP_TABLE
```

Static inner classes can also be used to model data

An inner class declared in the database helper class mimics the structure of each row of the table, and helps in passing data around.
Bearing in mind that such static classes cannot access non-static variables of the enclosing class.

```
public class CharaDbHelper extends SQLiteOpenHelper {

//code not shown --

   static class CharaData{

       private String name;
       private String description;
       private String file;
       private Bitmap bitmap;

       //constructors and getters not shown

   }

}
```

Intents - startActivityForResult()

Recall that

- **Explict intents** bring you from one activity to another. Data can be passed during this process.
- **Implicit intents** bring you from one activity to a component in your app. You specify the kind of component you want, and the android runtime fetches what is available.

In both cases, you launched the “destination” by invoking startActivity().

By invoking **`startActivityForResult()`**,
you expect the destination component to return a result.

Explicit Intent - startActivityForResult()

Step 1. Declare your request code, a final static integer variable that contains a unique integer that identifies your particular intent.

This is necessary as your activity could have more than one call to **`startActivityForResult()`**

```
public static int REQUEST_CODE_FAB = 1000;
```

Step 2. Declare an explicit intent in the usual way.

Then invoke **`startActivityForResult()`** with two arguments

- The intent
- The request code

```
Intent intent = new Intent(MainActivity.this, DataEntryActivity.class);
startActivityForResult(intent, REQUEST_CODE_FAB);
```

Step 3. In the destination activity, the user should interact with it.

Step 4. A user action (e.g. clicking a button) brings the user back to the origin activity.

The following code initiates this process.

```
Intent returnIntent = new Intent();
returnIntent.putExtra(KEY, value); //optional
setResult(Activity.RESULT_OK, returnIntent);
finish();
```

The first argument of **`setResult()`** is either

- **`Activity.RESULT_OK`** if the user has successfully completed the tasks
- **`Activity.RESULT_CANCELED`** if the user has somehow backed out

Hence, this code may be written twice, one for each scenario described above.

If you have data to transfer, you are reminded that you can use the **`putExtra()`** method above. (Recall Lesson 2).

Step 5. Back in the origin activity, override the callback `onActivityResult()` to listen out for the result and carry out the next task. Note the sequence of if-statements.

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent
data) {

   if (requestCode == REQUEST_CODE_FAB) {
       if(resultCode == Activity.RESULT_OK){

           //if you use putExtra in Step 4, then you need this step
           double value = data.getDoubleExtra(DataEntryActivity.KEY,
defaultValue);
           Toast.makeText(this,"Message",Toast.LENGTH_LONG).show();
       }
       if (resultCode == Activity.RESULT_CANCELED) {
           //Write your code if there's no result
       }
   }
}
```

If you have data transferred from step 4, you may retrieve this data on the intent object passed to this callback using the **`getDoubleExtra()`** method or other suitable methods(Recall Lesson 2).

Further reading

- https://google-developer-training.github.io/android-developer-fundamentals-course-concepts-v2/unit-1-get-started/lesson-2-activities-and-intents/2-1-c-activities-and-intents/2-1-c-activities-and-intents.html#gettingdatabackfromactivity
- https://developer.android.com/training/basics/intents/result

Implicit Intents - opening the image gallery

Attention if you are using the android emulator. To have some images for you to select, one easy way is to use the emulator to take some pictures first. I leave you to discover the best solution for yourself.

The implicit intents to write can be found at the following **Common Intents** reference
https://developer.android.com/guide/components/intents-common#java
To get the example code to retrieve an image, go to the **File Storage → Retrieve a Specific Type of File** section.

In case the website is not accessible, here is the sample code,

Example intent to get a photo:

KOTLIN JAVA

```java
static final int REQUEST_IMAGE_GET = 1;

public void selectImage() {
    Intent intent = new Intent(Intent.ACTION_GET_CONTENT);
    intent.setType("image/*");
    if (intent.resolveActivity(getPackageManager()) != null) {
        startActivityForResult(intent, REQUEST_IMAGE_GET);
    }
}

@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    if (requestCode == REQUEST_IMAGE_GET && resultCode == RESULT_OK) {
        Bitmap thumbnail = data.getParcelable("data");
        Uri fullPhotoUri = data.getData();
        // Do work with photo saved at fullPhotoUri
        ...
    }
}
```

In order to convert a URI object to a bitmap object, here's the code:
A static method is provided in the Utils class for converting an InputStream object to a Bitmap object.

```
InputStream inputStream =
this.getContentResolver().openInputStream(data.getData());
bitmapSelected = Utils.convertStreamToBitmap(inputStream);
```

Building Your App (Part 1)

Objective For Part 1

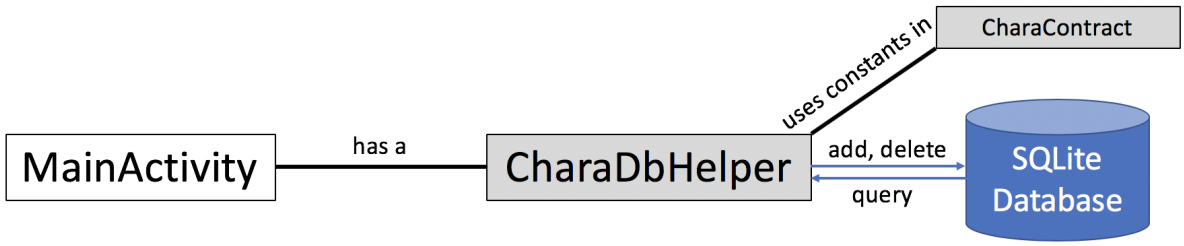
We will build CharaDbHelper with methods to

- Create the database
- Query the database
- Insert one row
- Delete one row

In this part, we will **not** be working with the UI.

All information will be printed to the Logcat.

The architecture of your app will be:



TODO 7.1 & 7.2 Examine the CharaContract Class and prevent instantiation of this class

The **`CharaEntry`** inner class stores the column names of the table.

The **`CharaSql`** inner class stores the SQL commands needed.

You may write logcat statements in **`MainActivity`** to display these strings.

TODO 7.3 Complete CharaData inner class with getters

The **`CharaData`** inner class is used in two ways:

- To store the name, description and filename as read from the pictures.json file in the raw folder
- To store the name, description and bitmap file

TODO 7.4 Make CharaDbHelper a singleton class

The constructor for **`CharaDbHelper`** is given to you. Implement the singleton design pattern with it.

TODO 7.5 & 7.6 Override and Implement onCreate() and onUpgrade()

TODO 7.7 Complete queryNumRows() to return the number of rows in the database

TODO 7.8 - 7.11 Complete the methods to conduct database operations

- getDataFromCursor()
- queryOneRowRandom()
- queryOneRow()
- insertOneRow()
- deleteOneRow()

TODO 7.12 & 7.13 In MainActivity, instantiate CharaDbHelper and test the methods written in TODO 7.7 - 7.11

Use the methods in TestCharaDbHelper to verify that your methods above do what they should.

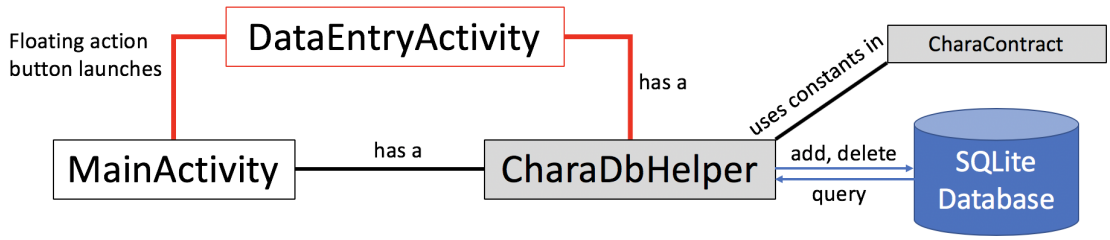
Building Your App (Part 2)

Objective For Part 2

Assuming that you have a working CharaDbHelper, then you are ready to make use of it in the user interface.

- Display a random image on the UI
- Obtain an image from your phone's image gallery and add it to your database

The architecture of your app will be:



TODO 8.1 & 8.2 When the button is clicked, query the database for a random image and display it on the screen

TODO 8.3 When the floating action button is clicked, bring the user to DataEntryActivity using startActivityForResult()

In DataEntryActivity ….

TODO 8.4 & 8.5 Get references to the widgets on the layout and CharaDbHelper

TODO 8.6 & 8.7 When the Select Image button is clicked, set up an implicit intent to the image gallery and complete onActivityResult() to receive the data

TODO 8.8 When the OK button is clicked, the data is added to the database and the user is brought back to MainActivity

In MainActivity …

TODO 8.9 Complete onActivityResult() to display a toast

Lesson 4 (Part 3)

Objectives

- Explain the components needed to set up a RecyclerView widget
- Display data in a local SQLite database in a RecyclerView widget by writing a RecyclerView adapter class
- Write code to Delete a RecyclerView ViewHolder by means of swiping

The Android/Java you need to know

Gradle and Configuration Files

Gradle is the software component that manages the build process for an android app.

The build process begins from the source code and ends with the APK file.

The APK file can then be installed on any Android phone.

You may obtain the APK file in Android studio using

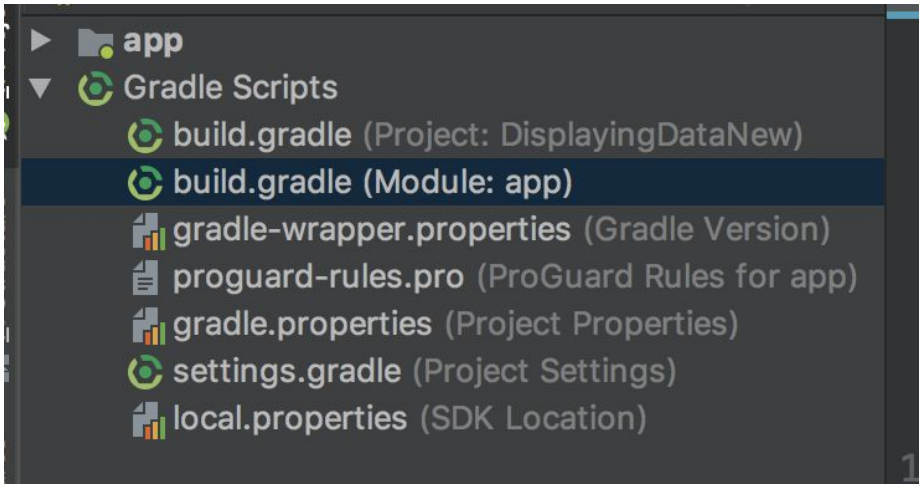
Build → Build APK(s)

More information on the build process here:

https://developer.android.com/studio/build/

Settings for the build process are stored in two build.gradle files:

- the project-level file
- the module-level file



Gradle module-level settings

Often the project-level default settings are sufficient.

Hence, we usually have to modify the module-level settings only.

The first part (Lines 1 - 20) shows information such as

- Minimum API level
- Compile API level
- Target API level

You may adjust these settings if you are aiming for certain API levels.

```
1   apply plugin: 'com.android.application'
2
3   android {
4       compileSdkVersion 28
5       defaultConfig {
6           applicationId "com.example.norman_lee.myapplication"
7           minSdkVersion 15
8           targetSdkVersion 28
9           versionCode 1
10          versionName "1.0"
11          testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
12      }
13      buildTypes {
14          release {
15              minifyEnabled false
16              proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-ru
17          }
18      }
19  }
20
```

The following part (Line 21 onwards) shows the dependencies that your app has.

```
21  dependencies {
22      implementation fileTree(dir: 'libs', include: ['*.jar'])
23      implementation 'com.android.support:appcompat-v7:28.0.0-rc02'
24      implementation 'com.android.support.constraint:constraint-layout:1.1.3'
25      implementation 'com.android.support:design:28.0.0-rc02'
26      implementation 'com.android.support:recyclerview-v7:28.*'
27      implementation 'com.android.support:cardview-v7:28.0.0'
28      testImplementation 'junit:junit:4.12'
29      androidTestImplementation 'com.android.support.test:runner:1.0.2'
30      androidTestImplementation 'com.android.support.test.espresso:espresso-core:3.0.2'
31  }
```

Lines 26 and 27 are not part of the default dependencies generated from a fresh project. They were added in, and adding them in downloads the packages if you are doing it for a first time.

Strategy Design Pattern

In the strategy design pattern, parts of the behaviours of an object is handed over to other objects. This is known as **delegation**.

This provides flexibility at run-time as you can change those behaviours.

```
public abstract class Duck {

    private FlyBehavior flyBehavior;
    private QuackBehavior quackBehavior;
    String name;

    public Duck(){
    }

    public Duck(String name){
        this.name = name;
    }

    public void setFlyBehavior(FlyBehavior flyBehavior) {
        this.flyBehavior = flyBehavior;
    }

    public void setQuackBehavior(QuackBehavior quackBehavior) {
        this.quackBehavior = quackBehavior;
    }

    public void performFly(){
        flyBehavior.fly();
    }

    public void performQuack(){
        quackBehavior.quack();
    }

    public abstract void display();
}
```

In the abstract class above, the delegation happens as follows

- The flying behaviour is delegated to a **`FlyBehavior`** object
- The quacking behaviour is delegated to a **`QuackBehavior`** object

For the FlyBehavior, we implement different objects that represent different behaviour.

```
interface FlyBehavior {
   void fly();
}
```

```
class FlapWings implements FlyBehavior {
   @Override
   public void fly() {
       System.out.println("Flapping my Wings");
   }
}
```

Implement a class **`CannotFly`** that implements **`FlyBehavior`**.

The **`fly()`** method prints outs "**`I cannot fly :(`**"

Similarly, for **`QuackBehavior`** objects:

```
public interface QuackBehavior {
   void quack();
}
```

```
public class LoudQuack implements QuackBehavior {
   @Override
   public void quack() {
       System.out.println("QUACK");
   }
}
```

Finally, we subclass Duck with our own object.

```
public class MallardDuck extends Duck {

   MallardDuck(String name){
       super(name);
   }

   @Override
   public void display() {
       System.out.println("I am " + name + ", the Mallard Duck");
   }
}
```

And we can run our **`MallardDuck`** object and set their behaviours at run-time:

```
public class TestDuck {

   public static void main(String[] args){

       Duck duck = new MallardDuck("Donald");
       duck.setFlyBehavior(new FlapWings());
       duck.setQuackBehavior(new LoudQuack());
       duck.display();
       duck.performFly();
       duck.performQuack();
   }
}
```

We see here the **flexibility of composition over inheritance**.

We develop our duck behaviours independent of the type of duck.

The behaviour of the duck is delegated to separate objects.

You assemble your specific duck at run-time,

which gives you the flexibility to change its behaviour if needed.

Adapter Design Pattern

The word **interface** is an overloaded word

- in Java terminology it would mean a type of class with method signatures only
- it could also mean the set of methods that a class allows you to access
  (think of ‘user interface’)

An adapter design pattern converts the interface of one class into another that a client class expects.

Adapter Design Pattern Example

You have an interface **`Duck`** and a class **`MallardDuck`** that implements this interface.

```
public interface Duck {
   void quack();
   void fly();
}
```

```
public class MallardDuck implements Duck {
   @Override
   public void quack() {
       System.out.println("Mallard Duck says Quack");
   }

   @Override
   public void fly() {
       System.out.println("Mallard Duck is flying");
   }
}
```

Then you have a client that loops through all ducks and make them fly and quack.

```
import java.util.ArrayList;
public class DuckClient {

   static ArrayList<Duck> myDucks;

   public static void main(String[] args){
       myDucks = new ArrayList<>();
       myDucks.add( new MallardDuck());
       makeDucksFlyQuack();
   }

   static void makeDucksFlyQuack(){
      for(Duck duck: myDucks){
               duck.fly();
               duck.quack();
      }
   }
}
```

Now you have a **Turkey** interface.

```
public interface Turkey {

   public void gobble();
   public void fly();
}
```

How might we allow **Turkey** objects to be used by the same client?

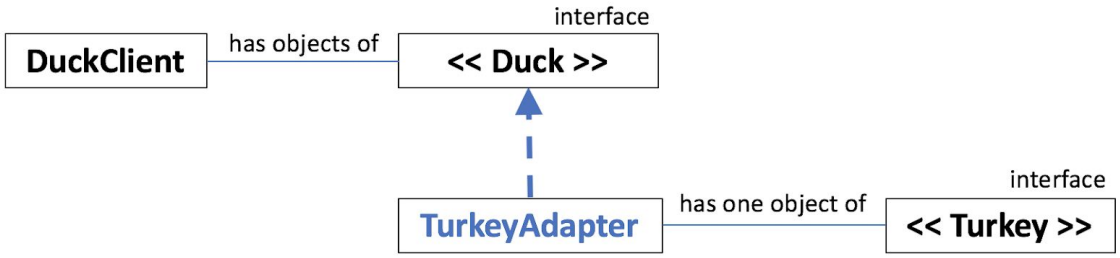
We write an adapter class that

- has the same Duck interface and
- takes in a Turkey object

```
public class TurkeyAdapter implements Duck {

   Turkey turkey;

   TurkeyAdapter(Turkey turkey){
       this.turkey = turkey;

   }
   @Override
   public void quack() {
       //implement this
   }

   @Override
   public void fly() {
       //implement this
   }
}
```

This material was taken from “HeadFirst-Design Patterns”

Explaining the Duck/Turkey Adapter Example



What is a RecyclerView?

Suppose you have a collection of similar data e.g.

- Images with descriptions
- Chat messages with sender's name

… and you want to display them in your app.

The **RecyclerView** widget allows the user to scroll through the data.
This is done by loading each data item onto its own item in RecyclerView.
A typical RecyclerView display is shown below.

CardView

A useful widget that can display data in RecyclerView is the **CardView** widget.

To use CardView, ensure that you have the following dependency in your module-level gradle file:

```
implementation 'com.android.support:cardview-v7:28.0.0'
```

CardView gives the “card look” to each item.

- You can change attributes to tweak the look of the cards.
- You then specify the layout of widgets within a CardView.

The following is an overview of an XML file specifying a CardView and the layout within. Details have been removed from most widgets.

```
<android.support.v7.widget.CardView
   android:id="@+id/cardViewItem"
   app:cardPreventCornerOverlap="false"
   cardCornerRadius="5dp"
   cardMaxElevation="1dp"
   cardElevation="1dp"
   cardUseCompatPadding="true"
   android:layout_width="match_parent"
   android:layout_height="100dp"
   android:layout_margin="16dp">

   <RelativeLayout
       android:id="@+id/ard"
       android:layout_width="match_parent"
       android:layout_height="match_parent">

       <ImageView />
       <TextView />
       <TextView />
       <TextView />
   </RelativeLayout>

</android.support.v7.widget.CardView>
```

How to implement RecyclerView

To use RecyclerView in your app,

Step 1. ensure that you have the following dependency in your module-level gradle file

```
implementation 'com.android.support:recyclerview-v7:28.*'
```

Step 2. Include the following widget tag in the Activity layout where you want to have the recyclerView.

```
<android.support.v7.widget.RecyclerView
   android:id="@+id/charaRecyclerView"
   android:layout_width="match_parent"
   android:layout_height="match_parent" />
```

Step 3. Assuming each data item is stored in a CardView, design the layout of each data item.

Step 4. Decide the source of your data:

- Stored in the res folder
- SQLiteDatabase
- Cloud Database
- etc

This lesson shows you how to use data from a local SQLiteDatabase.

You would have written a Database Helper class.

Step 5. Write an Adapter class that extends the **`RecyclerView.Adapter<VH>`** class.

This class takes in your data source and is called by the Android runtime to display the data on the RecyclerView widget. This class also references the data item that you designed in step 3.

This will be explained in the next section.

Step 6 continues next page …

Step 6. In the java file for your activity, write code for the following

- Get a reference to the recyclerView widget using findViewById()
- Get an instance of an object that points to your dataSource
- Instantiate your Adapter
- Attach the adapter to your recyclerView widget
- Attach a Layout manager to your recyclerView widget. A LayoutManager governs how your widgets are going to be displayed. Since we are scrolling up and down, we will just need a LinearLayoutManager.

The sample code is here. You will need to adapt the code a little.

```
recyclerView = findViewById(R.id.charaRecyclerView);
dataSource = ??? ;
charaAdapter = new CharaAdapter(this, dataSource );
recyclerView.setAdapter(charaAdapter);
recyclerView.setLayoutManager(new LinearLayoutManager(this));
```

This way of coding shows you how **delegation** is performed.

Delegation is the transferring of tasks from one object to a related object.

The **`RecyclerView`** object delegates

- the role of retrieving data to the **`RecyclerView.Adapter`** object.
- the role of managing the layout to the **`LinearLayoutManager`** object

Thus the RecyclerView object makes use of the Strategy Design Pattern.

Writing The RecyclerView Adapter - Static Inner Class

The Recycler View adapter class is the adapter class between the RecyclerView widget and the object containing your source of data.

Your RecyclerView Adapter should extend the **`RecyclerView.Adapter<VH>`** class.
`VH` is a generic class that subclasses **`RecyclerView.ViewHolder`**.
This is an **abstract class** without abstract methods.
Hence, Android is forcing you to subclass this class to use its methods.

This class is meant to hold references to the widgets in each data item layout.
Typically, we will write such a class as an inner class within the recyclerView adapter.

Hence, the classes are declared in the following way.

```
public class CharaAdapter extends
RecyclerView.Adapter<CharaAdapter.CharaViewHolder>{

   //code not shown
   static class CharaViewHolder extends RecyclerView.ViewHolder{
        //code not shown
   }
}
```

Having designed your CardView layout for each data item, **`CharaViewHolder`** will contain instance variables that are meant to hold references to the widgets on the layout.
The references are obtained by calling **`findViewById()`** within the constructor.

Writing the RecyclerView Adapter - write the constructor and override three methods

The **constructor** should take in

- a Context object
- Object for your data source

The context object is used to get a layout inflator object to be used in **`onCreateViewHolder()`**.

`RecyclerView.Adapter<VH>` is an abstract class and you have to override three abstract methods.

`onCreateViewholder()` is called by the run-time each time a new data item is added.
In the code recipe below:

- The CardView layout is inflated
- A reference to the layout in memory is returned **`itemView`**
- This reference **`itemView`** is passed to the constructor of **`CharaViewHolder`**
- **`CharaViewHolder`** uses this reference to get references to the individual widgets in the layout

Here's a typical recipe:

```
public CharaViewHolder onCreateViewHolder(@NonNull ViewGroup
viewGroup, int i) {
   View itemView = mInflater.inflate(R.layout.layout, viewGroup,
false);
   return new CharaViewHolder(itemView);
}
```

`onBindViewHolder()` is meant to

- get the appropriate data from your data source
- attach it to the widgets on each data item, according to the adapter position.

Hence, the data on row 0 of a table goes on position 0 on the adapter and so on.

`getItemCount()` is meant to return the total number of data items. Hence, if you return 0, nothing can be seen on the RecyclerView.

Seeing the connection

The Recycler View adapter class is the adapter class between the RecyclerView widget and the object containing your source of data.

Compare the RecyclerView implementation with the Duck/Turkey example

Example	**RecyclerView component**
Duck and DuckClient	
TurkeyAdapter	
Turkey	

Getting Each Item to Respond to Clicks

This is not in the list of TODOs, but it would be instructive to think about how it can be done.

Since we extend **`RecyclerView.ViewHolder`**, we have access to the parent class' methods. One method is **`getAdapterPosition()`**, which displays the ViewHolder's position on the Recycler View.
Use this method to display a toast when each ViewHolder is clicked.

Option 1. Since a reference to the CardView layout is passed to the ViewHolder class, then you may call **`setOnClickListener`** on this reference within the constructor, and pass to it an anonymous class in the usual way.

Option 2. **`CharaViewHolder`** class can implement the **`View.OnClickListener`** interface. Then **`onClick`** has to be implemented as an instance method.
You still need to call **`setOnClickListener`** on the reference to the CardView layout.
What object do you pass to **`setOnClickListener`**?

Swiping To Delete

We are able to write code to delete a particular ViewHolder when it is swiped left/right.
The code recipe is to create an instance of **`ItemTouchHelper`**
and attach the RecyclerView instance to it.

```
ItemTouchHelper itemTouchHelper
              = new ItemTouchHelper(simpleCallback);
itemTouchHelper.attach(recyclerView);
```

The constructor takes in an object that extends the **`ItemTouchHelper.SimpleCallback`** abstract class.
To use this class,

- Pass the direction of swiping that you want to detect to its constructor
- Override **`onSwipe()`**

From the documentation, the directions are specified via constants.
As you are going to use this object only once,
an acceptable practice is to use an anonymous abstract class.
As we are coding for swiping, we do not write any other code in **`onMove()`**.

```
ItemTouchHelper.SimpleCallback simpleCallback = new
ItemTouchHelper.SimpleCallback(0, ItemTouchHelper.LEFT |
ItemTouchHelper.RIGHT ) {
   @Override
   public boolean onMove(@NonNull RecyclerView recyclerView,
@NonNull RecyclerView.ViewHolder viewHolder, @NonNull
RecyclerView.ViewHolder viewHolder1) {
       return false;
   }

   @Override
   public void onSwiped(@NonNull RecyclerView.ViewHolder
viewHolder, int i) {

   }
}
```

Two parameters are passed to **`onSwiped()`**:

- an instance of the ViewHolder that is currently being swiped
- the direction (change the variable name of the autogenerated code …)

The tasks are

- **Downcast** the ViewHolder object so that you can use the instance variables or methods that you have defined
- Call your database helper with the required information to delete the particular row in the database
- Display any other UI message e.g. a toast message saying a deletion has been happening
- Notify the RecyclerView adapter that the database has an item removed
  (Where did the **`getAdapterPosition()`** method come from?)

```
CharaAdapter.CharaViewHolder charaViewHolder =
(CharaAdapter.CharaViewHolder) viewHolder;

String name = charaViewHolder.textViewName.getText().toString();
charaDbHelper.deleteOneRow(name);

Toast.makeText(RecylerViewActivity.this, "Deleting " +
       name, Toast.LENGTH_LONG).show();

charaAdapter.notifyItemRemoved(viewHolder.getAdapterPosition());
```

Building Your App

TODO 9.1 - In MainActivity, add an Options Menu Item to bring your users to the RecyclerView activity.

When this is done, you should see an empty screen. You may add an additional text view widget to the RecyclerView activity layout to gain the confidence that you have done this correctly.

TODO 9.2 - 9.6 - Complete CharaAdapter

To do this, you should

- Complete the inner class CharaViewHolder
- Complete the constructor
- Complete onCreateViewHolder
- Complete onBindViewHolder
- Complete getItemCount to return the size of the data

You may add logcat statements to see when onCreateViewHolder and onBindViewHolder are invoked

TODO 9.7 Complete RecyclerViewActivity

Add the code so that your RecyclerView widget has an adapter and a linearLayout manager. At this stage, if you have added logcat statements above, add sufficient data so that your screen is filled. You may scroll up and down and observe the logcat statements to see how the adapter callbacks are invoked.

TODO 9.8 Complete the code to delete by swiping left and right

Explained above.