Lesson 2 - Exchange Rate App

Objectives

- Describe the static factory method and builder design pattern in Java
- Use the BigDecimal class for financial calculations
- From an EditText widget, extract data and specify input settings
- Use the logcat to display messages to track the behaviour of an app
- 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 Unit Testing and write code for unit tests using the JUnit framework

© Singapore University of Technology and Design

Introduction

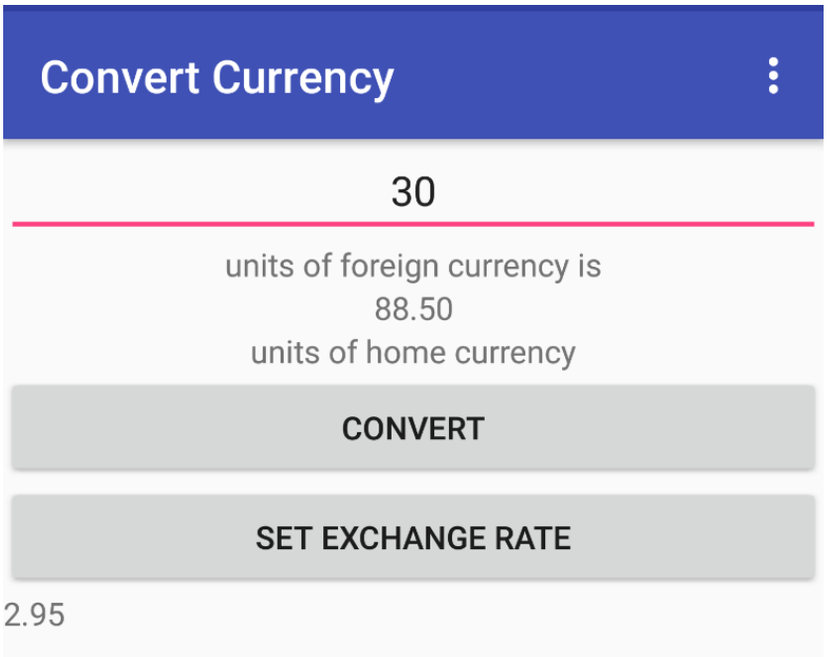
In this series of lessons, we will build an app that is handy for travelling. I often travel with people who want to know exactly how much an item would cost in their home currency.

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 foreign currency buys 2.95 units of home currency.

- 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.



© Singapore University of Technology and Design

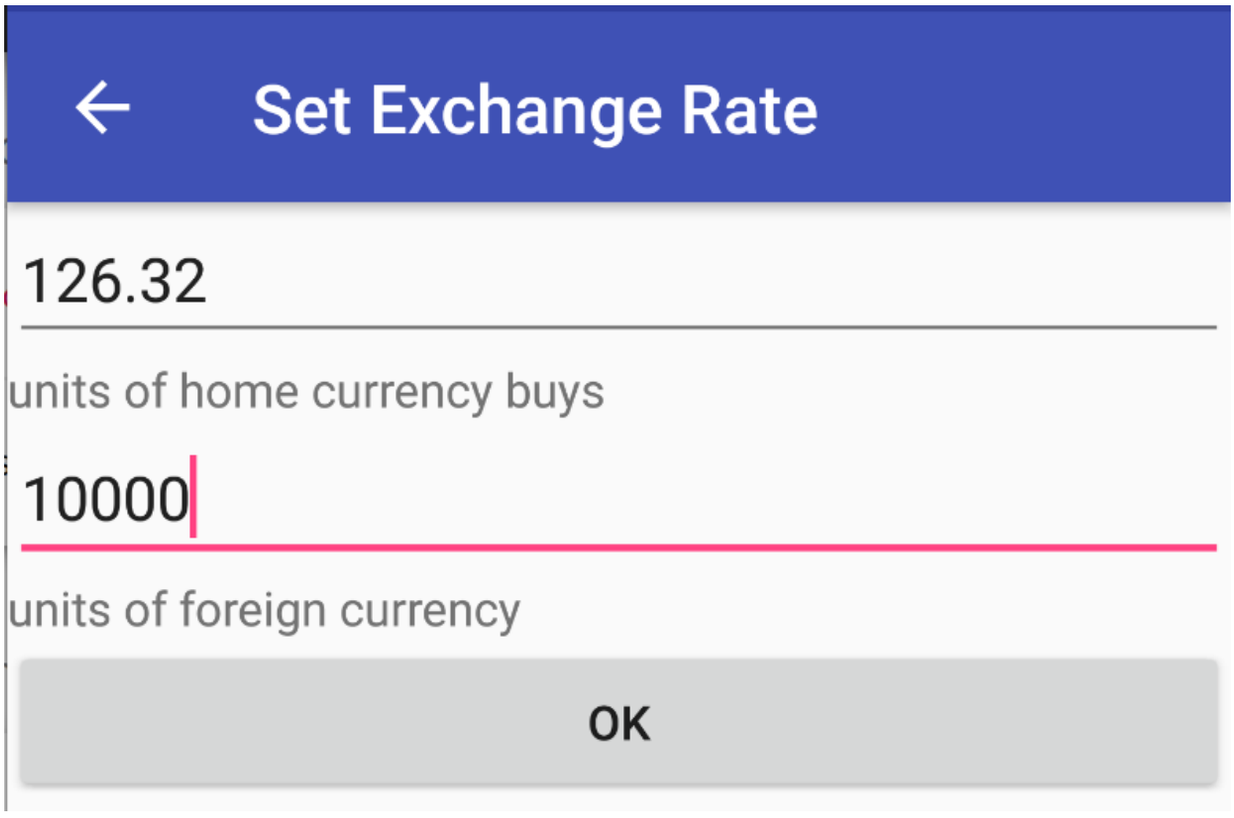
SubActivity

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 entered 126.32 units of home currency to buy 10000 units of foreign currency.
- 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 following situations:

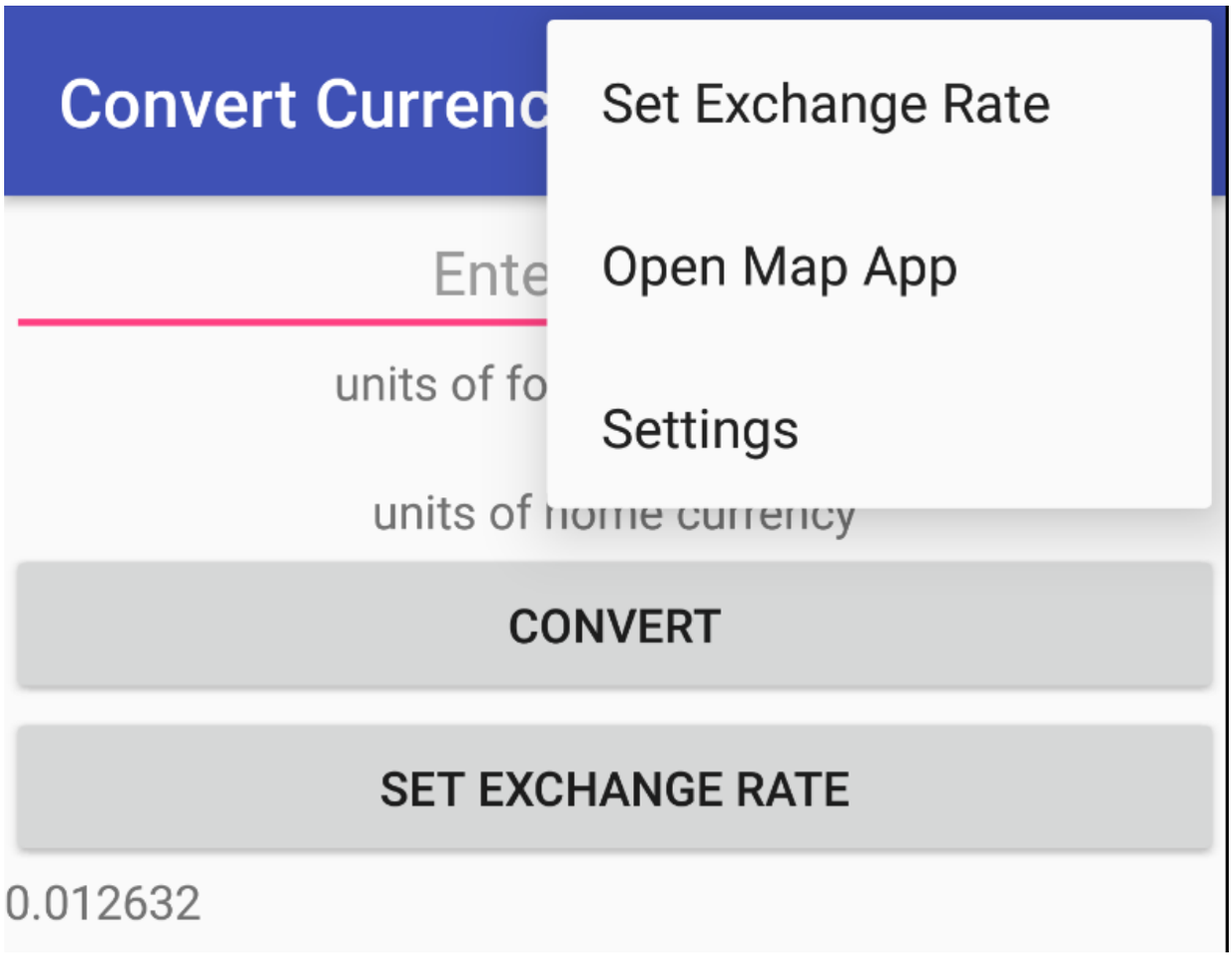
- the user enters non-numeric data
- 0 or a negative number is entered



© Singapore University of Technology and Design

Back in Main Activity

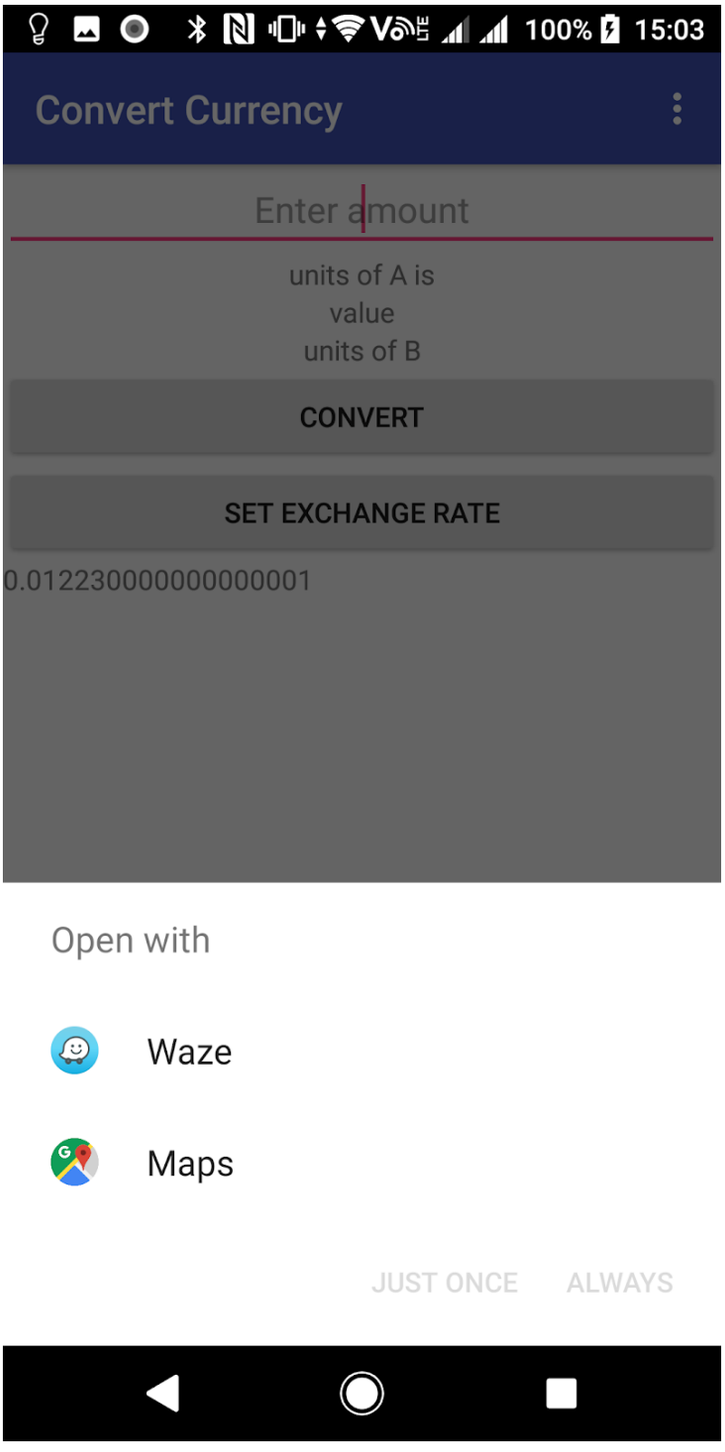
- Once the exchange rate is set, the new exchange rate is reflected in **MainActivity**.
- Without some coding, closing and reopening the app erases the data entered. We would like the data to be retained, by using the **Shared Preferences** class.
- We would like to give the user more options to navigate 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



© Singapore University of Technology and Design

When Open Map App is clicked

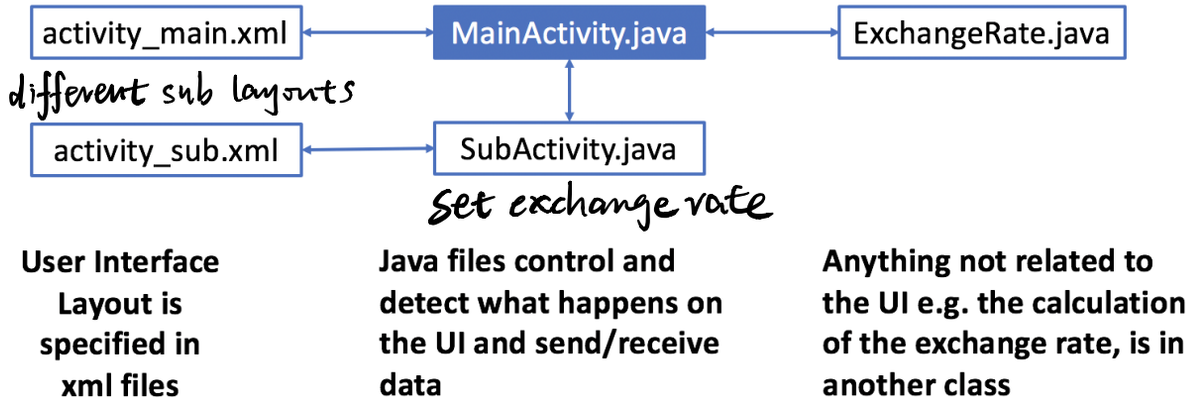
After clicking on Open Map App, the following screen is shown. The user is allowed to pick from the available map apps on his/her phone.



© Singapore University of Technology and Design

Design and Structure of the app

The diagram (which is overly simplified) below shows how the classes interact with each other. This shows how each file has its own responsibilities.



The calculation of an exchange rate could be easily integrated into MainActivity.java. However, I choose to encapsulate it in another class because of the **Single Responsibility Principle** - A class should have only one reason to change. Hence, applying this principle leads me to separate the job of calculating the exchange rate, from the job of interacting with the UI.

package all the needed thing in one package (file)

© Singapore University of Technology and Design

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?

© Singapore University of Technology and Design

What you need to know (Android/Java)

Use the BigDecimal class for financial calculations

Types such as double and float are not suitable for financial calculations because often, accuracy is demanded, but you will encounter floating point errors with these types.

```
System.out.println(0.7 + 0.1); //do you get 0.8?
```

0.79999...

2 base system

Why?

In computer every decimal value is represented in two parts,

1. The integer value **i**
2. The **c** represents the scale.
3. Every decimal value can be encoded as **i * b**$^{-c}$ where **b** is the *base*, can be 10, 2 etc.

E.g. if b is 10, 0.1 is represented as 1 * 10^{-1} and 0.25 is represented as 25 * 10^{-2} (or normalized as 2*10^{-1} + 5 * 10^{-2}. If b is 3, ⅓ is represented as 1 * 3^{-1}. Note that ⅓ in 10-base representation is an approximation. In most of the computer run-time, the base is 2 (binary), hence 0.5 is 1*2^{-1}, 0.25 is 1*2^{-2} and 0.75 is 3*2^{-2} or 1*2^{-1} + 1*2^{-2}. Unfortunately in a 2-base system, 0.1 has to be approximated, just like ⅓ has to be approximated in a 10-base system, but perfectly accurate in a 3-base system.

The **`BigDecimal`** class provides a natural representation of decimal by representing numbers in a **10-base** system with two parts

- *Unscaled Value* - an integer of arbitrary precision, that's the **i.**
- *Scale* - the number of digits after the decimal point, that's the **c.**

b=10

The **`BigDecimal`** class can be initialized with different types, but we shall stick to using **`String`**. If a **`String`** passed to the constructor does not have a recognizable number e.g. an empty string or a string containing text, a **`NumberFormatException`** is thrown.

Objects of the **BigDecimal** class have methods *add*, *subtract*, *multiply* and *divide*. As **BigDecimal** is immutable, these methods return a new object.

```
BigDecimal a = new BigDecimal("1");
```


© Singapore University of Technology and Design

```
BigDecimal b = new BigDecimal("4");
BigDecimal c = a.divide(b);
System.out.println(c);
System.out.println("Scale:" + c.scale() );
System.out.println("Unscaled value:" + c.unscaledValue());
```

$c \rightarrow 25 * 10^{-2}$

0.25

2

25

If any operation results in an infinite number of decimal places e.g. ⅓ , a **`MathContext`** object is needed, which encapsulates the number of significant figures and the mode of rounding. The following code fragment illustrates.

```
BigDecimal up = new BigDecimal("10");
BigDecimal down = new BigDecimal("7");
int sigFigures = 5;
MathContext mc = new MathContext(sigFigures,RoundingMode.HALF_UP);
BigDecimal result = up.divide(down,mc);
System.out.println(result);
```

5.s.f

above 0.5 round up

There are different rounding modes, which you can read about.

© Singapore University of Technology and Design

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)

© Singapore University of Technology and Design

Android manifest

The android manifest is found in the **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.

sub activity pointing to the main activity

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.

© Singapore University of Technology and Design

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.

© Singapore University of Technology and Design

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;
}
```

double c = a / b; if (Double.isInfinite(c)) { throw new ArithmeticException(); } return c;

© Singapore University of Technology and Design

Static Factory Method

Create a specific object instance with initialization state

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 );
   }
}
```

we don't know what does it contain

create tea with m and s

create tea without s

Hence, you invoke the static factory method like this:

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


© Singapore University of Technology and Design

Toasts

You might have seen a message on an android app 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.

© Singapore University of Technology and Design

Logcat

like print

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");
```

Log.i("MainActivity.onCreat", "hello")

`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

© Singapore University of Technology and Design

Explicit Intent

a change of activity

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

target

- 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**) don't know which activity we goma switch over to

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.

transfer

cannot use type as arg need to use value

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

class name.class

value representation of the class

The constructor of the intent object takes in two inputs

instance of the

- **`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.

after that ask main activity to pause
then call subactivity.onCreat
main activity stop

© Singapore University of Technology and Design

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);
```

If the KEY doesn't exist in this activity, we use

assuming KEY is a Static attribute defined in the main activity class

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

In this case, we want to receive a double value, 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

© Singapore University of Technology and Design

Android Activity Lifecycle

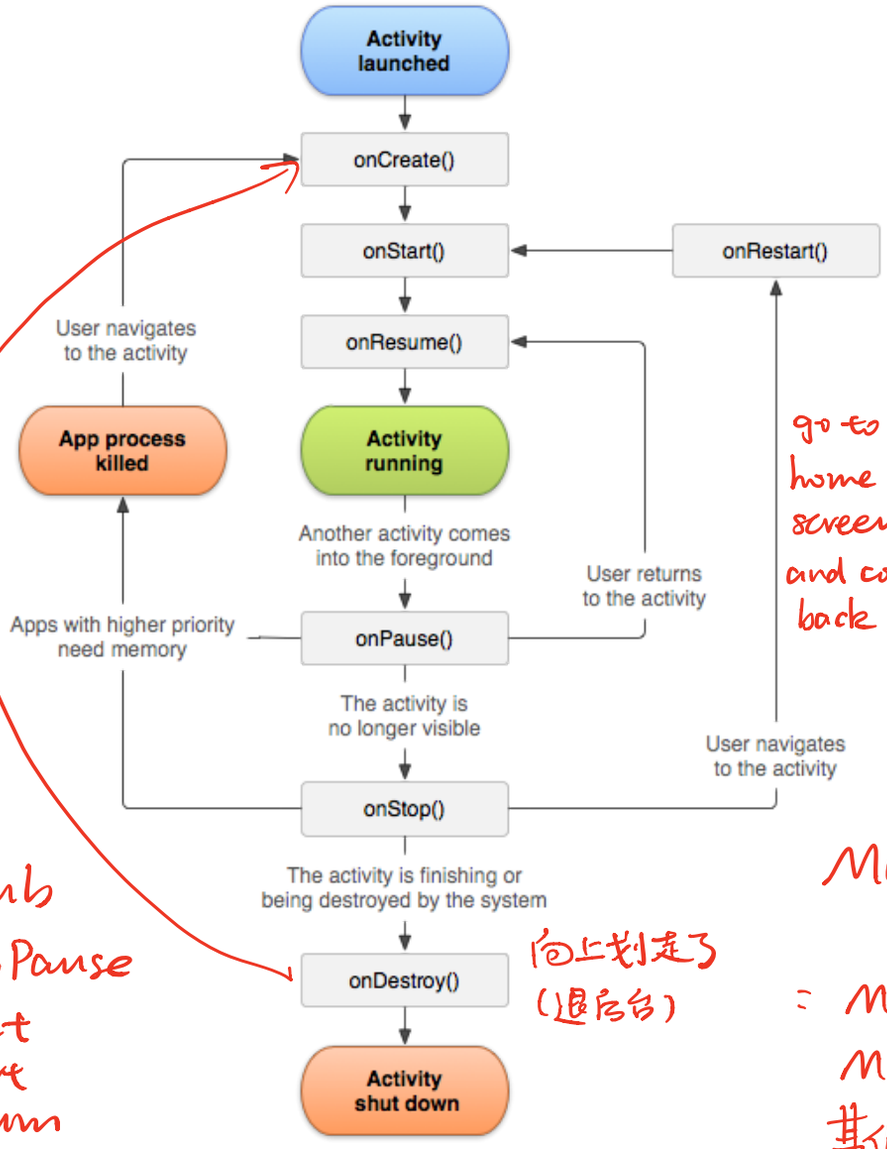
So far, we have been writing code in **`onCreate()`**, which is called when the activity starts.

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**.



switch activity

go to home screen and come back

Main → Sub

: Main = onPause

Sub : creat
start
resum

Main : onStop

向上划走了
(退后台)

(home screen)

Main → 其他地方
(eg. map)

: Main = onPause

Main = onStop

其他(map) → Main

: M : onRst
onS
onRe

© Singapore University of Technology and Design

Sub → main

: sub = onPause

main destory
onC
onS → sub onstop
onDestory

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

- Which methods are called as the activity starts? onCreat → onStart → onResume
- Which methods are called as you navigate from one activity to another?
- Which methods are called as you rotate the screen? Main: onP onStop onD onC onStart onResume

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

off the sreen: onP onS
↓
start sreen: onRestart onStart onResume

© Singapore University of Technology and Design

can also use database

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.

in the file within app

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();
}
```

nested class / interface

return editer interface

© Singapore University of Technology and Design

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.

© Singapore University of Technology and Design

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 have many attributes needed to be initialised

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?

© Singapore University of Technology and Design

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

If the constructor is (1) private and (2) takes in an instance of the builder class, the only way to instantiate your object would be to use the builder class.

The code below illustrates this point.

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

    private 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); }
    }
}
```

innerclass

within the outer class TeaTwo

same attri

to have this inner class we aim to create an object of the outer class

access constructor of the outer class

actual object

why static? for non-static inner class it can only be constructed with the presence of an object of the outer class

为什么static？需要他来creat outer class instance 但只有有了outer class instance才能用他(进了loop里面了)

The builder is then used as follows:

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

give object of TeaBuilder innerclass

same { TeaBuilder builder = new TeaTwo.TeaBuilder();
TeaTwo teaTwo = builder.setSugar .....

© Singapore University of Technology and Design

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.

HURL
- A document on the internet: **`http://www.google.com`**
- A file on your computer: **`file:/Users/Macintosh/Downloads/url.html`**

OURL
- A geographic location: **`geo:0.0?q=test`**
- An email: `mailto:` **`test@sutd.edu.sg`**

分层的

Hierarchical URIs have a slash character after the scheme and can be parsed as follows:

[*scheme*:][authority][path][?query][#fragment//] 有'/'的

不透明

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

© Singapore University of Technology and Design

Implicit Intents

to construct, first need to construct a URI

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. ☆ use a builder design pattern

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.

outer

builder

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

inner

setter

obtain a URI obj

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 I set
- 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);
```

Map app

(看地理位置)

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

② app doesn't have the right (promission) to access the method

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

① with the current android device we are running, doesn't have the right app to support this kind of action with this kind of data

Other intents

Other common intents, eg. opening the camera, are specified here:

https://developer.android.com/guide/components/intents-common

© Singapore University of Technology and Design

Implicit Intents in Android SDK API ver. 30+

The following declarations in AndroidManifest.xml are required for the implicit intent to work.

declaring the right

```
<queries>
     <!-- Browser -->
     <intent>
         <action android:name="android.intent.action.VIEW" />
         <data android:scheme="http" />
     </intent>
     <!-- Camera -->
     <intent>
         <action android:name="android.media.action.IMAGE_CAPTURE"
/>
     </intent>
     <!-- Gallery -->
     <intent>
         <action android:name="android.intent.action.GET_CONTENT"
/>
     </intent>
     <!-- Map -->
     <intent>
         <action android:name="android.intent.action.VIEW"/>
         <data android:scheme="geo"/>
     </intent>
 </queries>
```

For details, refer to these documentation.

https://developer.android.com/about/versions/11/privacy/package-visibility

https://medium.com/androiddevelopers/package-visibility-in-android-11-cc857f221cd9

© Singapore University of Technology and Design

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)~~ **NOT TAUGHT IN 2022 Spring, but you are encouraged to read it up by yourself**

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**.

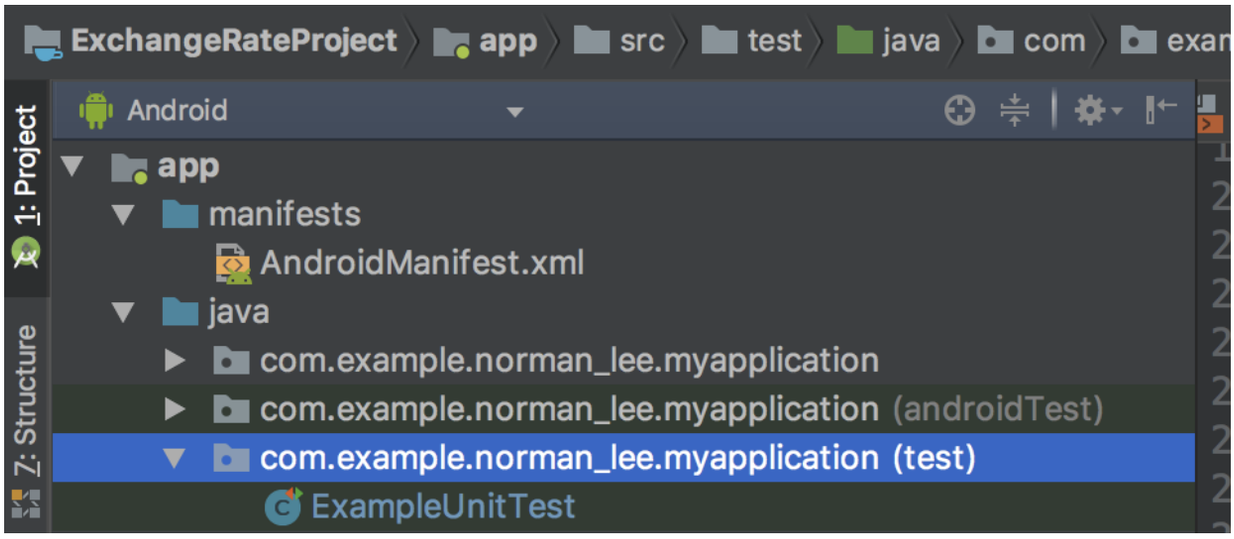
Android studio automatically generates the test class for you.

Question. Since Java comes with a compiler that tries to detect all kinds of anomalies during run-time? Why do we still need to perform testing?

© Singapore University of Technology and Design

Where and how to run unit tests

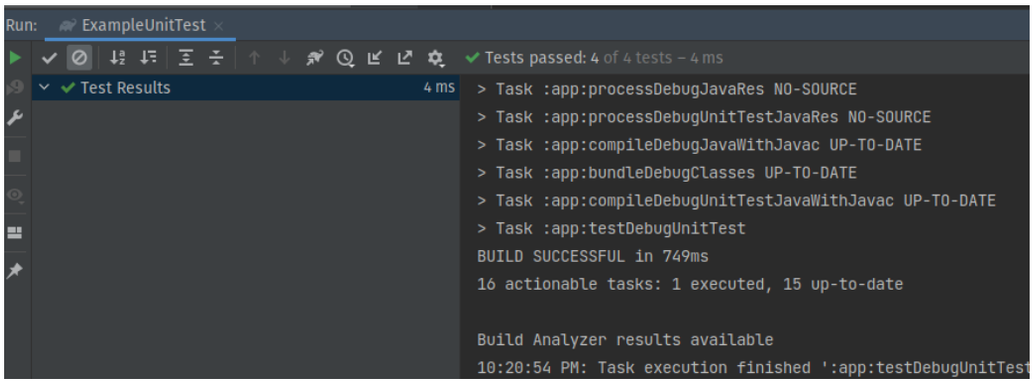
The test class is found in the folder marked **(test)**



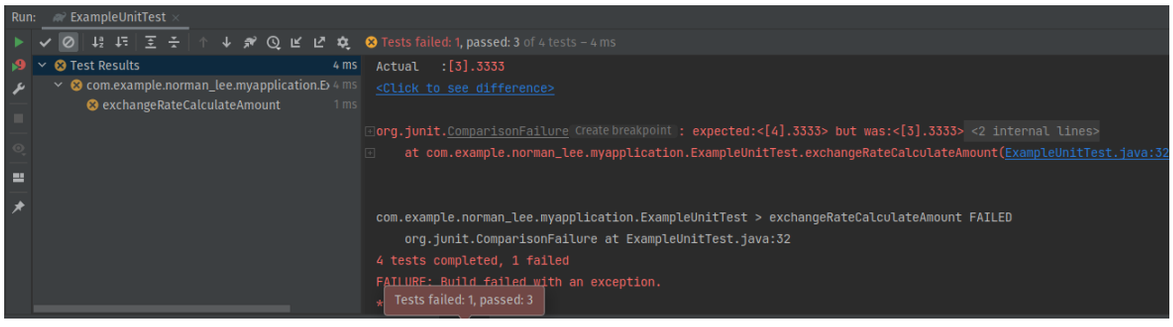
The default file generated contains one trivial unit test, and you can write more.

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:



© Singapore University of Technology and Design

How to write a unit test

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.

In the following unit tests in Examples 1 to 3, suppose that there is a class **`A`** that has a method called **`someFunction()`** that is overloaded and has the following specifications:

- If no parameters are passed to it, it returns a double value of 2.95
- If a string containing a number is passed to it, it returns a **`BigDecimal`** object initialized with that string
- If a string is passed to it and it does not contain a number, it throws an **`IllegalArgumentException`**

Example 1. In the following unit test, we want to write a test to ensure that **`someFunction()`** returns a double value of **`2.95`**. As double objects could have floating point error, the third argument is a tolerance: if abs(*expected* - *actual*) < *tolerance*, then *expected* is treated to be equal to *actual*.

```
@Test
public void someFunctionTest1(){
   assertEquals(2.95, new A().someFunction(),0.001);
}
```

Example 2. In the following unit test, we want to write a test to ensure that **`someFunction()`** returns a **`BigDecimal`** object when a string is passed to it.

```
@Test
public void someFunctionTest2(){
   assertEquals(new BigDecimal("2.95"), new
A().someFunction("2.95") );
}
```


© Singapore University of Technology and Design

Example 3. In the following unit test, we want to see if the function throws an **`IllegalArgumentException`**

```
@Test(expected= IllegalArgumentException.class)
public void someFunctionTest3() {
   new A().someFunction("apple");
}
```

This section on Instrumented Testing will not be discussed (hence, not tested) in 2022.

The material is retained here for your information.

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.

© Singapore University of Technology and Design

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()**

© Singapore University of Technology and Design

- 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

Discussion. Commercial software applications tend to have a huge code base in size. With highly frequent releases, tight schedules, and limited resources. Software testing is always costly. Suppose you are a manager, how do you make software testing cost-effective without sacrificing the quality of the software?

© Singapore University of Technology and Design

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.

© Singapore University of Technology and Design

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! One option is to use the **`ExchangeRate`** class provided 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

For TODO 2.4 - 2.5, there are three steps.

Step 1. First you would have to retrieve the text of the edit text widget by calling its **`getText().toString()`** method.

Step 2. If it is an empty string, you could handle it in two ways, either with

- an if/else block, or
- Try-catch block, making use of the behaviour of the BigDecimal class

Step 3. In the ExchangeRate class, if **`calculateAmount()`** is not completed, you should complete it as well.

© Singapore University of Technology and Design

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>
```


© Singapore University of Technology and Design

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 in the exchange rate information

- 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 EditText widgets
- Check that these values are valid (please see next section)
- Set up an explicit intent and pass the values back to MainActivity

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

© Singapore University of Technology and Design

TODO 3.9, 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. In this app, you should:

- Decide how you are going to handle a divide-by-zero situation or when negative numbers are entered
- 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 **`Utils`** that contains a static method **`checkInvalidInputs()`**.
It throws

- **`IllegalArgumentException`** if 0 or a negative number is entered. You have to write code to throw this exception.
- **`NumberFormatException`** if the EditText widgets have an empty string. This is thrown by using the **`valueOf`** static method of the **`Double`** class.

Here's the method stump.

```
public class Utils {

   static void checkInvalidInputs (String value){

   }
}
```

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, retrieve the values passed to it, and instantiate the ExchangeRate class

- Instantiate the **`ExchangeRate`** class based on the data passed to the intent.
  To do so, complete the two-argument constructor in **`ExchangeRate`**.
- You also have to think about the situation when the app is launched for the first time i.e. how would you get the exchange rate if the user has not keyed it in yet?

© Singapore University of Technology and Design

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

© Singapore University of Technology and Design

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?

© Singapore University of Technology and Design

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 (i.e. when the app is started from the first time)

You have to think about the instantiation of the **`ExchangeRate`** class. There are two situations now:

- The exchange rate is retrieved from the SharedPreferences
- The exchange rate is calculated based on values passed via the intent from SubActivity

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

TODO 4.11 [Implement yourself] In MainActivity, when the home amount is calculated, it should be rounded to two decimal places.

You'll need to modify **`ExchangeRate`** for this.

© Singapore University of Technology and Design

Part 4 - Implicit Intents, Unit 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 **ExchangeRate** class.

~~**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?

© Singapore University of Technology and Design

TODO 5.4 Write the following unit tests in the test folder

Write the following unit tests

1. The no-arg constructor gives a default exchange rate of 2.95000, represented by a **`BigDecimal`** object. Complete the following example:

```
@Test
 public void exchangeRateDefaultRate(){
     String defaultExchangeRate = "2.95000";
     assertEquals( ?? , new ExchangeRateNew().getExchangeRate() );
```

2. Write similar tests for the other constructors.

3. **`calculateAmount()`** should return the correct answer as a **`BigDecimal`** object.

4. An empty string passed to a constructor argument results in a **`NumberFormatException`** thrown.

5. Can you think of other tests to write?

© Singapore University of Technology and Design