

CHAPTER 04: DEBUGGING AND TESTING USING JUNIT ESPRESSO AND MOCKITO FRAMEWORKS

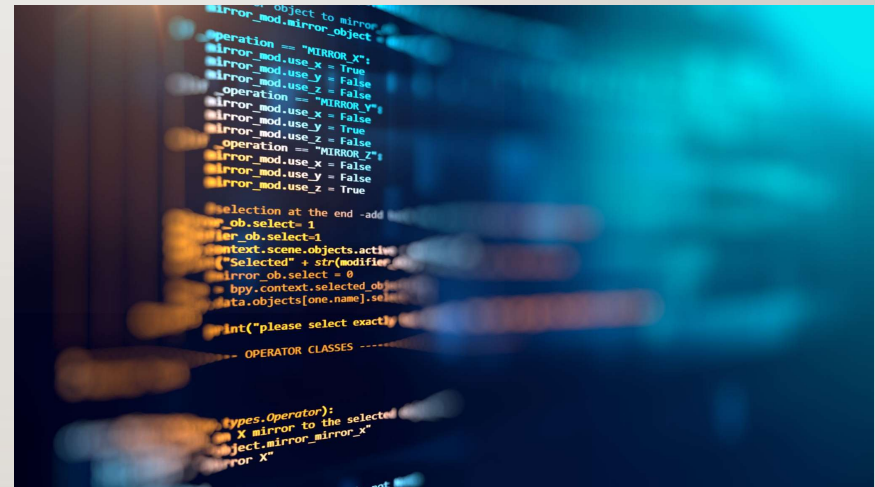


WHAT YOU WILL LEARN IN THIS CHAPTER

Debug	Debug code with the Android Studio Debugger
profiler	Run the Android Profiler to profile your code
Explorer	Use the Device File Explorer to manage your app files
Bridge	Use the Android Debug Bridge command-line tool to access Linux shell
Messages	Create Toast and Snackbars messages
Logcat	Use Log and the Logcat utility to view log messages from the Logcat window
Test	Use Junit , Espresso , and Mockito frameworks testing to test your code
Coverage	Apply test coverage using Android Studio
Reverse	Use a reverse engineering technique to generate a class diagram from the code

CHECK OUT THE DEMO PROJECT

- Download the demo app, **ch04.zip**
- How to run the code: unzip the code in a folder of your choice, then in Android Studio click File->import->Existing Android code into the workspace



4.1 INTRODUCTION

In this chapter, the focus will be put on fault detection using the Android Studio Debugger to locate faults and code errors, and to fix them

In addition to what Android Studio offers as an IDE to enable debugging and testing, we also study the Android classes and methods that facilitate testing

Different from the previous chapters, in this chapter, you need to practice what will be described



4.2 FAULT HANDLING METHODS

There are various software fault-handling methods and approaches. These include preventing a fault from happening in the first place, detecting the fault and fixing it, or applying mechanisms to tolerate fault.

These methods and techniques can be grouped and classified in various ways.

Fault Handling

- Fault Avoidance
- Fault detection
 - Debugging
 - Correcting code
 - App performance
 - Requirement testing
- Fault-tolerance
 - Component redundancy
 - N version programming

BUG TYPES YOU WATCH FOR

When developing Android apps, the types of bugs that you should watch out for include incorrect or unexpected results, wrong values, crashes, exceptions, freezes, memory leaks

Android Studio comes with various built-in tools and plugins for app debugging, i.e., finding and fixing errors in code, as well as testing app performance, and verifying/validating the non-functional requirements.

Android Studio enables you to debug apps running on the emulator and Android devices.

PART I: THE ANDROID STUDIO DEBUGGER

Tools and options that can be useful for testing and debugging include:

- Inspect and modify variable values during debugging
- Use the Android profiler tool to profile the performance of code
- Use the device file explorer to access and modify files on Android devices
- Use the Android Debug Bridge command-line tool to run Linux commands on an Android device

4.3 ENABLE DEBUGGER

To debug code, you need to enable the debugging option on your device. It is enabled for the emulator by default

On newer Android devices, you need to go to Settings → System → about → build number, then tap *build number* *seven times* or more to make developer options available

You also need to enable the USB debugging option.

The debugger also allows you to see the execution stack



4.2 STEP THROUGH YOUR CODE

Open the project in Android Studio. Select a hardware device, an emulator, or your connected device, for debugging code.

Click on the debug icon on the toolbar or click *run* → *debug* on the menu bar.

The app starts on the selected device and the debugging window opens.

Once you see the debugging window, it indicates that the debugger is attached to your development environment.

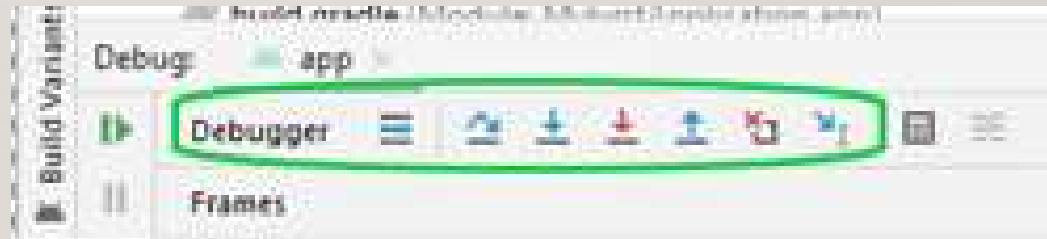
Now, you can step through your code, execute it line by line, using the debugging buttons

4.3 ENABLE DEBUGGER

While debugging, you can execute code by clicking on the step buttons or using the shortcuts such as F8 for Start Over and F7 for Step Into.

During debugging, you can highlight any variable or expression by right-clicking on it and pressing **Evaluate Expression** or

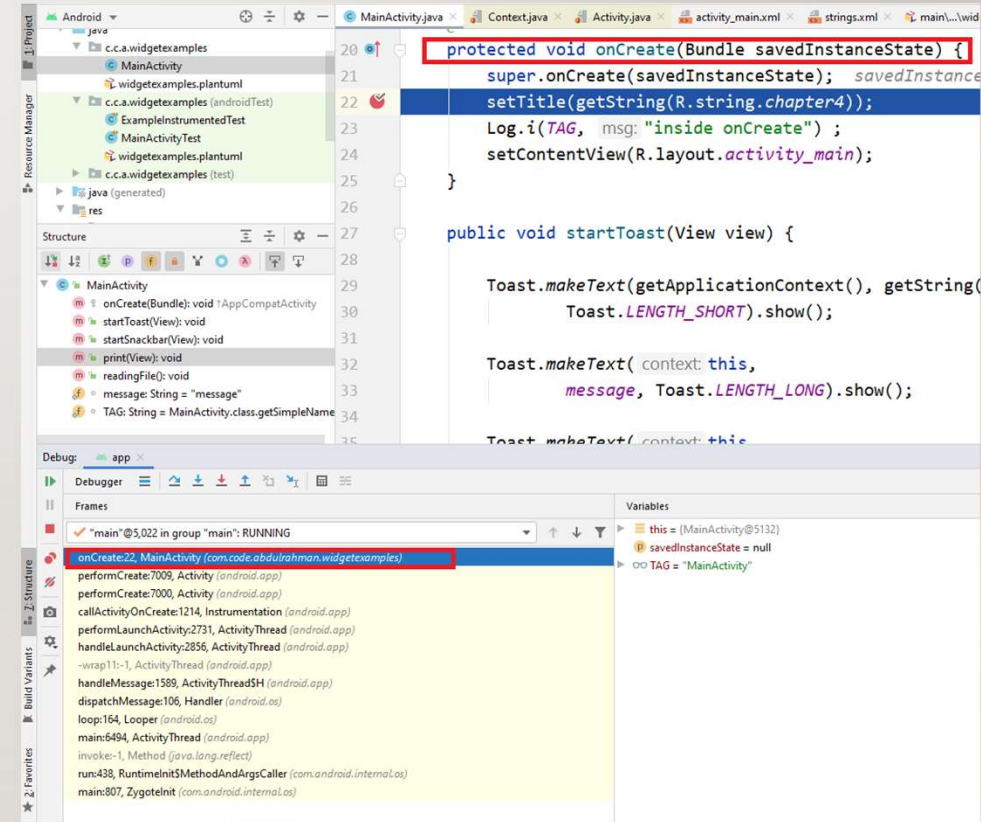
Add to **Watch** to find out the value of the variable and the value of the expression when it is executed.



4.3 THE EXECUTION STACK

The debugger also allows you to see the execution stack. This is shown in the figure

The execution stack and the frame that responds to the current breakpoint are shown



4.4 INSPECTING AND MODIFYING VARIABLE VALUES

While you are running your code in the debugging mode with **breakpoints** set, if you execute your code one step at a time using the step buttons, you can see the state of each class variable and object in the Debug window

When your app is ready to be published you need to **remove** the logging statements and any calls to Toasts and Snackbar debugging messages

This cleanup is needed to avoid app hanging and crashes

4.5 ANDROID PROFILER

Quality Assurance requirements, such as performance, usability, robustness, security, privacy, and other non-functional requirements, are essential parts of almost all apps

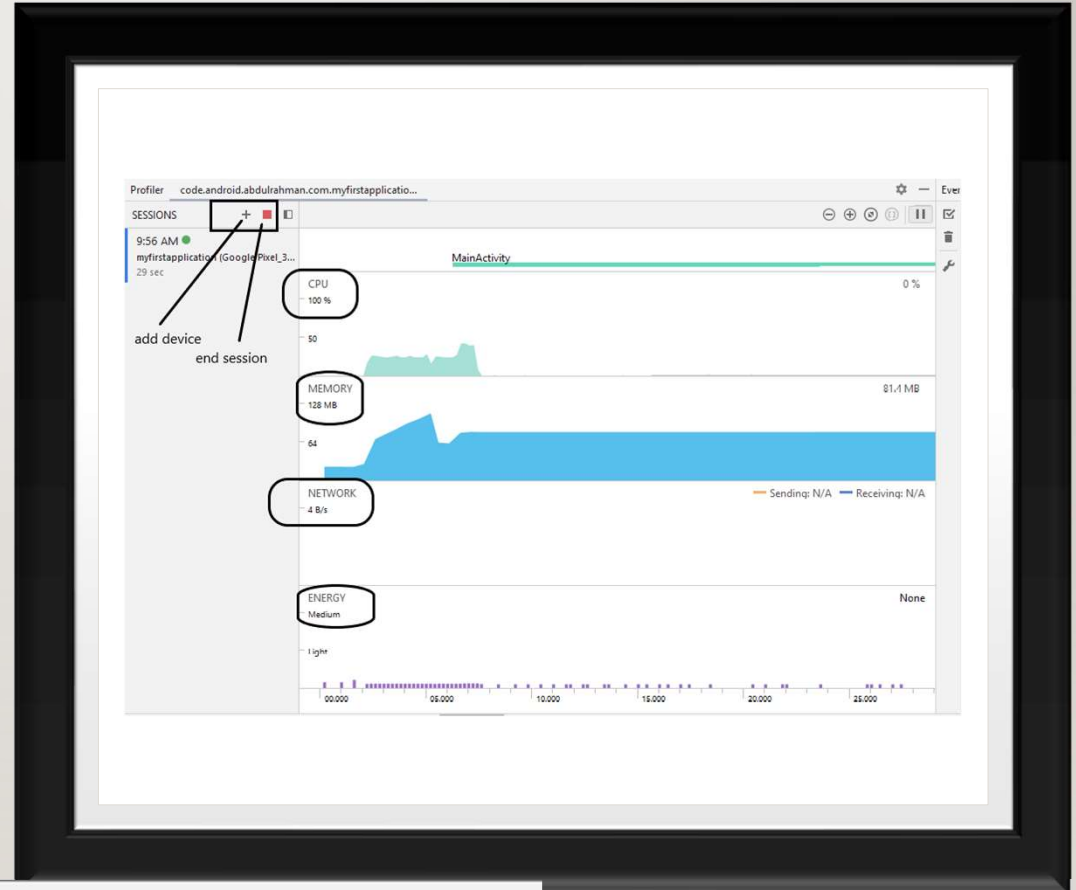
The Android Profiler provides you with the ability to profile your code, i.e., verify/validate the non-functional requirement such as the performance and usability of your app

Code profiling provides you with real-time data on how much your app uses CPU, memory, network, and battery resources

you have the option to choose between two different profiling modes: "Low Overhead" and "Complete Data."

4.5 ANDROID PROFILER

- To open the Profiler window, on the Android Studio's menu bar click *View* → *Tool Windows* → *Profiler* or click on the Profile icon



4.6 DEVICE FILE EXPLORER

The Device File Explorer enables you to view, edit, copy, and delete files on an Android device

To use the Device File Explorer on the Android Studio's menubar Click View → Tool Windows → Device File Explorer or click the Device File Explorer button if it is visible in the window toolbar

4.6 DEVICE FILE EXPLORER

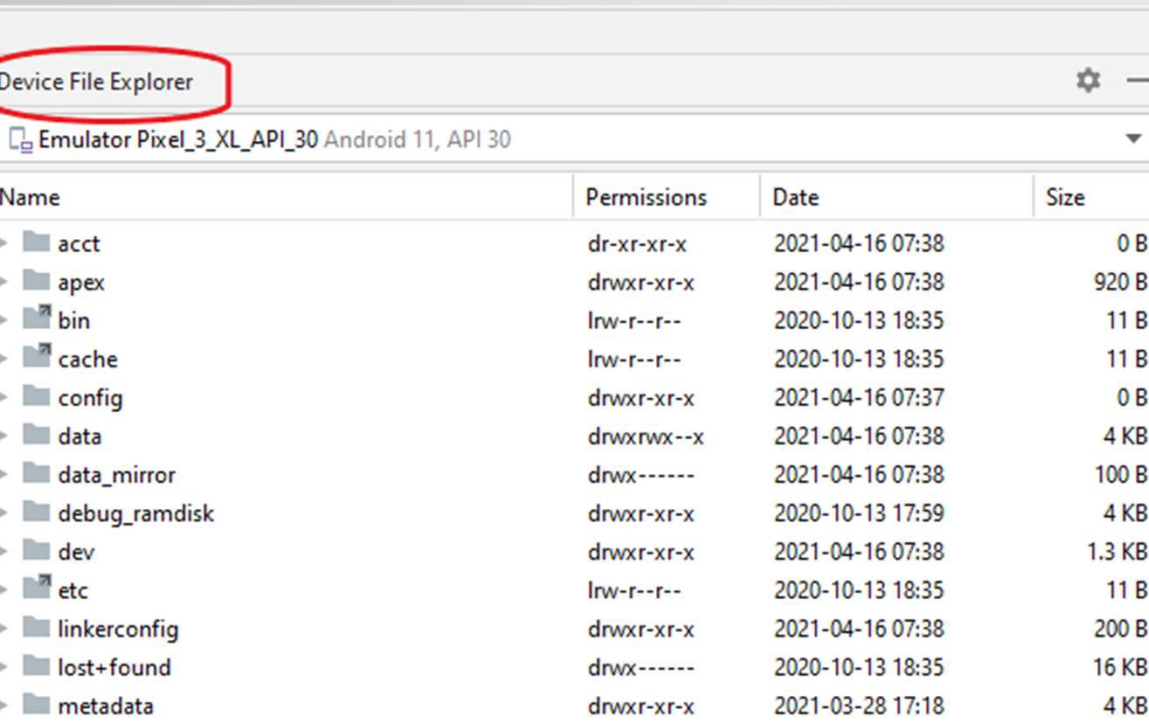
Using the Device File Explorer, you can move and create files and directories and check to see whether the file your app is supposed to create has actually been created.

It is important to be able to access the data directory of your emulator/device and be able to, for example, delete some of the apps/files you no longer use to free up space.

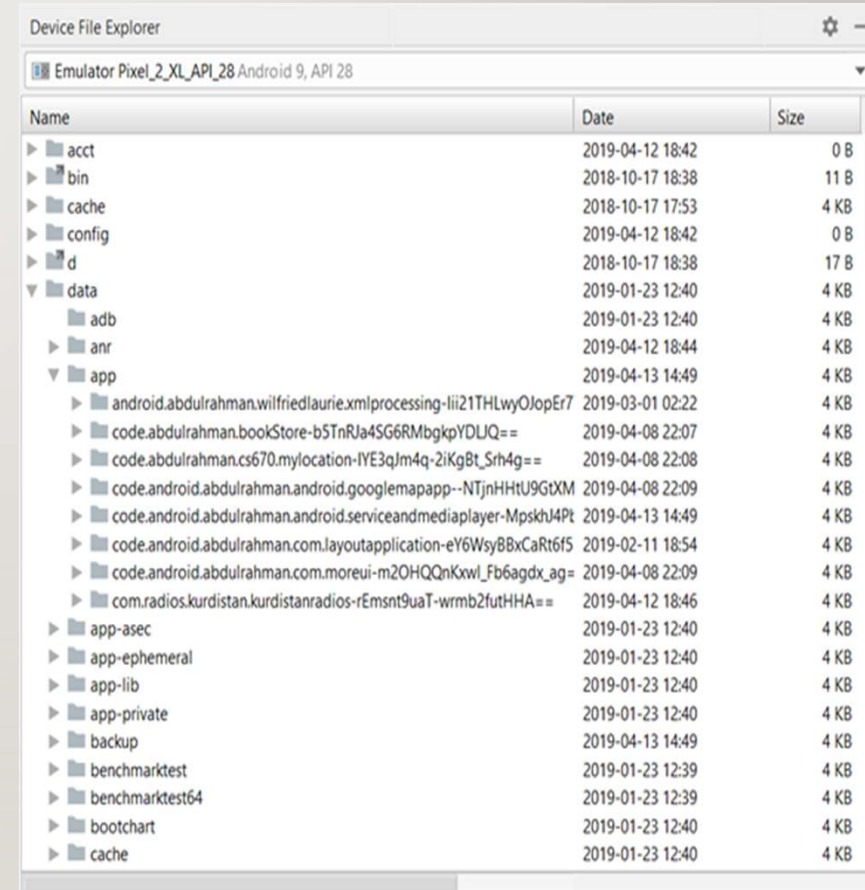
To practice with the Device File Explorer, run your app and select a device, and open the Device File Explorer window to see the files stored on the device

4.6 DEVICE FILE EXPLORER

- device's directories



Name	Permissions	Date	Size
acct	dr-xr-xr-x	2021-04-16 07:38	0 B
apex	drwxr-xr-x	2021-04-16 07:38	920 B
bin	lrw-r--r--	2020-10-13 18:35	11 B
cache	lrw-r--r--	2020-10-13 18:35	11 B
config	drwxr-xr-x	2021-04-16 07:37	0 B
data	drwxrwx--x	2021-04-16 07:38	4 KB
data_mirror	drwx-----	2021-04-16 07:38	100 B
debug_ramdisk	drwxr-xr-x	2020-10-13 17:59	4 KB
dev	drwxr-xr-x	2021-04-16 07:38	1.3 KB
etc	lrw-r--r--	2020-10-13 18:35	11 B
linkerconfig	drwxr-xr-x	2021-04-16 07:38	200 B
lost+found	drwx-----	2020-10-13 18:35	16 KB
metadata	drwxr-xr-x	2021-03-28 17:18	4 KB



Name	Date	Size
acct	2019-04-12 18:42	0 B
bin	2018-10-17 18:38	11 B
cache	2018-10-17 17:53	4 KB
config	2019-04-12 18:42	0 B
d	2018-10-17 18:38	17 B
data	2019-01-23 12:40	4 KB
adb	2019-01-23 12:40	4 KB
anr	2019-04-12 18:44	4 KB
app	2019-04-13 14:49	4 KB
android.abdulahman.wilfriedlaurie.xmlprocessing-lili21THLWYOJopEr7	2019-03-01 02:22	4 KB
code.abdulahman.bookStore-b5TnRJa4SG6RMbgkpYDLUQ==	2019-04-08 22:07	4 KB
code.abdulahman.cs670.mylocation-IYE3qIm4q-2iKgBt_Srh4g==	2019-04-08 22:08	4 KB
code.android.abdulahman.android.googlemapapp--NTjnHHTU9GtXM	2019-04-08 22:09	4 KB
code.android.abdulahman.android.serviceandmediaplayer-MpskhJ4Pt	2019-04-13 14:49	4 KB
code.android.abdulahman.com.layoutapplication-eY6WsyBBxCaRt6f5	2019-02-11 18:54	4 KB
code.android.abdulahman.com.moreui-m2OHQQnKxwl_Fb6agdx_ag=	2019-04-08 22:09	4 KB
com.radios.kurdistan.kurdistanradios-rEmsnt9uaT-wrmb2futHHA==	2019-04-12 18:46	4 KB
app-asec	2019-01-23 12:40	4 KB
app-ephemeral	2019-01-23 12:40	4 KB
app-lib	2019-01-23 12:40	4 KB
app-private	2019-01-23 12:40	4 KB
backup	2019-04-13 14:49	4 KB
benchmarktest	2019-01-23 12:39	4 KB
benchmarktest64	2019-01-23 12:39	4 KB
bootchart	2019-01-23 12:40	4 KB
cache	2019-01-23 12:40	4 KB

4.7 ANDROID DEBUG BRIDGE



Android has a command-line tool, the Android Debug Bridge tool, for running Linux commands on the Android device



The executable command, adb.exe, is located inside the **platform-tools directory**



At the bottom of Android Studio, click on the Terminal button, or on Android Studio's menubar, click **View → Tool Windows → Terminal to open the Terminal window**



In the Device File Explorer section, we saw that the file structure of the Android device is the same, or very similar to, the Linux file structure

4.7 ANDROID DEBUG BRIDGE

In the *Terminal Window* change the directory, i.e., `cd`, to where `adb.exe` is located. It is usually located at:
`C:\Users\userName\AppData\Local\Android\Sdk\platform-tools\adb.exe`.

Replace 'userName' in the path above with *your* username

Once you change the directory to the `adb` directory, you can start `adb shell` and run Linux commands.

This tool shows that the Android operating system is a modified version of the Linux operating system.

To start type: `./adb shell`

`C:\Users\AbdulYunis\AppData\Local\Android\Sdk\platform-tools>adb shell` **make sure**
you are connected to emulator or a device

4.7 ANDROID DEBUG BRIDGE

C:\Users\yoursuename\AppData\Local\Android\Sdk\platform-tools> **./adb devices**. The result for example can be:

List of devices attached

emulator-5554 device

./adb shell

ls -l /sdcard

adb install /path/to/your/app.apk

adb uninstall package_name

adb reboot

4.8 DO IT YOURSELF

- Open one of the apps we have created for this course and practice the Android Debugger, the Device File Explorer, the Android Profiler, and the Android debugging bridge
- Run one of the apps developed in this course and use the Device File Explorer to find where the apps have been installed on the emulator or your device file system
- **Project Idea:**

Create an app that allows you to access all the apps installed on a device. To do so, you need to access the Data directory on the device



PART 2: TOAST AND SNAKBAR MESSAGES

In programming languages such as Java, `System.out.println` is heavily used by programmers for debugging

C/C++ developers use the “`printf`” and “`cout`” commands to print messages to the console

Developers use the `print` and `println` methods to print messages to the console and learn what their code is doing

PART 2: TOAST AND SNAKBAR MESSAGES

In Android, the Toast and snackbar classes are used to give users feedback messages on events that occurred

When the toast message is shown, it appears as a floating text over the screen.
It does not block user interaction with the device.

You can use toast/snackbar to check

1. when inserted data is correct/incorrect
 2. the file download is complete
 - 3, the background process is started
 4. a file is deleted
 5. etc
-

PART 2: TOAST AND SNAKBAR MESSAGES

You can use these classes to identify where in your code something went wrong as well

For example, you can create snackbar/toast messages inside the catch block of exception handling to find out where the exception happened and its type.

4.9 TOAST MESSAGES

Toast messages are usually created by calling one of the static methods of the Toast class, for example, **makeText**

The method signature of makeText is as follows:

```
public static Toast makeText (Context context, CharSequence text, int duration);
```

PUBLIC STATIC TOAST MAKETEXT

MakeText is a standard toast and takes three parameters:

- the object, or the view, that displays the toast message,
- the text of the message,
- the duration that the message stays displaying.

PUBLIC STATIC TOAST MAKETEXT

- The context parameter to the `MakeText` method is usually 'this'.
- The duration can have any of these constants:
 - `Toast.LENGTH_SHORT`
 - `Toast.LENGTH_LONG`

You need to call the **show** method on the `makeText()` method to display the actual Toast message



PUBLIC STATIC TOAST MAKETEXT

There is another `MakeText` method that takes the resource id as a second parameter instead of the `CharSequence` type for the text of the message, meaning that the text is from a resource folder

Android uses `CharSequence` in places where `String` is used in Java, but you can still use `String`



The signature for the second `MakeText` method is shown below:

```
public static Toast makeText(Context  
cxt, int resId, int duration);
```

you can use `getString(R.string.toastText)` for the ***resId***.

getString() method



The `getString(R.string.toastText)` method is a public method in the Activity class



`R.string.toastText` can be interpreted as follows:



the `R` class references its static field `res` (`res` directory),



`string` refers to the `strings.xml` file inside the `values` directory,



`toastText` is a key for a string element inside the resource tag. The steps of referencing the

4.9 TOAST MESSAGES



```
import android.widget.Toast;

public class MainActivity extends AppCompatActivity {

    ...

    public void startToast(View view) {

        Toast.makeText(getApplicationContext(), getString(R.string.app_name),
            Toast.LENGTH_SHORT).show();

        Toast.makeText(this, message, Toast.LENGTH_LONG).show();

        Toast.makeText(this, "some text", Toast.LENGTH_LONG).show();

        Toast.makeText(this, getString(R.string.app_name),
            Toast.LENGTH_SHORT).show();

    }
}
```

4.10 SNACKBAR MESSAGES



Like Toast messages, the Android **SDK** has another class called **Snackbar** in the package `android.support.design.widget` for displaying messages



It can be used to display a brief message to the user at the bottom of the phone and lower left on large devices



The message automatically goes away after a short period making it different from Toast which goes away only when the display time expires

4.10 SNACKBAR MESSAGES

- Displaying the Snackbar message in your app involves a few steps.
 - Create a Snackbar object with the message text by calling the static method `make()`
 - The Snackbar object creation is shown below.
`Snackbar snackbar = Snackbar.make(view, resouceID, duration);`
- Call the `show` method to show the message to the user.
For example,
`snackbar.show();`

4.10 SNACKBAR MESSAGES

You also need to import

android.support.design.widget.Snackbar to
your code

have the **com.android.support:design** support
library in your *build.gradle* file.

4.10 SNACKBAR MESSAGES

- Below is an example of a build.gradle file with a support library for the Snackbar.
- Depending on the API level used, your support library can be the same or different from the example below:

```
dependencies {  
    implementation fileTree(dir: 'libs', include: ['*.jar'])  
    implementation 'com.android.support:appcompat-v7:28.0.0'  
    implementation 'com.android.support:design:28.0.0'  
    implementation 'com.android.support.constraint:constraintlayout:1.1.3'  
    testImplementation 'junit:junit:4.12'  
    androidTestImplementation 'com.android.support.test:runner:1.0.2'  
    androidTestImplementation 'com.android.support.test.espresso:espresso-core:3.0.2'  
}
```

4.10 SNACKBAR MESSAGES



The parameters for the `Snackbar.make` are **View**, **resourceID**, and the **duration**.
`Snackbar.make (View view, int resId, int duration)`



The purpose and the description of the parameters are as follows:



View: The view that the Snackbar should be attached to.



The **view object** can be obtained in different ways:



`View parentLayout = findViewById(R.id.content)`. `R.id.content` returns the root of the current View.

4.10 SNACKBAR MESSAGES

The purpose and the description of the parameters continue ...

On some devices, `R.id.content` returns null. If you experienced a null return, use

- `getWindow().getDecorView().findViewById(android.R.id.content) ;`

You can always assign an id to the Activity layout and use the given id to access the View of an Activity. Here is an example:

- `View parentLayout = findViewById(R.id.myLayout);`
- where **myLayout** is an id of a screen layout that you would like to display the Snackbar message on.

4.10 SNACKBAR MESSAGES

The purpose and the description of the parameters continue ...

resourceID: The *resource ID* of the message you want to display

- The description of the resourceID is same as the resourceID we discussed for MakeText in the Toast class.
- It refers to the text that can be obtained from the strings.xml file inside the res/values folder.
- For example, you can pass R.string.**app_name** or any other string you would like to display as a method parameter.

duration

It can be any one of these three static constants:

- **LENGTH_LONG, LENGTH_SHORT, or LENGTH_INDEFINITE**

4.10 SNACKBAR MESSAGES



The code snippet shows how the Snackbar class can be used in your coding:

- `public class SnackbarActivity extends AppCompatActivity {`
- `@Override`
- `protected void onCreate(Bundle savedInstanceState) {`
- `super.onCreate(savedInstanceState);`
- `setContentView(R.layout.activity_main);`
- `View parentLayout = findViewById(R.id.content);`
- `Snackbar snackbar = Snackbar.make (parentLayout,`
- `R.string.app_name,Snackbar.LENGTH_INDEFINITE);`
- `snackbar.show();`
- `}`
- `}`

4.12 THE LOG CLASS AND LOGCAT WINDOW



We have introduced the Log class and Logcat tool in the previous chapter



The Log class is included in the `android.util` package and can be used to log messages at runtime



The Log class has multiple static methods that can be utilized to create and filter proper messages

4.12 THE LOG CLASS AND LOGCAT WINDOW

The methods are *Log.e*, *Log.w*, *Log.i*, *Log.d*, *Log.v* and *Log.wtf*. As a developer, you use these methods to log proper messages: *error*, *warning*, *information*, *debugging messages*, *verbose*, and *what a terrible failure*, respectively.

These methods take two parameters, the first one is the TAG and the second one is the message information.

4.12 LOGGING CLASS EXCEPTION

There is a second version of each logging method listed above that takes three parameters each

The second version of the methods logs the error messages and the exceptions that can be thrown

4.12 LOGGING CLASS EXCEPTION

Exceptions and exception handling are an essential part of programming in Java and any object-oriented programming

The second versions of the log methods recognize this role and provide you with the ability to log the exceptions, thus, helping you to recognize errors and exceptions in your code

4.12 LOGGING CLASS EXCEPTION

An example of the Log method signature with three parameters is as follows:

```
Log.e(String TAG, String message, Throwable exception);
```

When an exception is thrown, it will be caught inside the catch block. The log statement inside the catch block is then used to log the exception.

4.12 LOGGING CLASS EXCEPTION

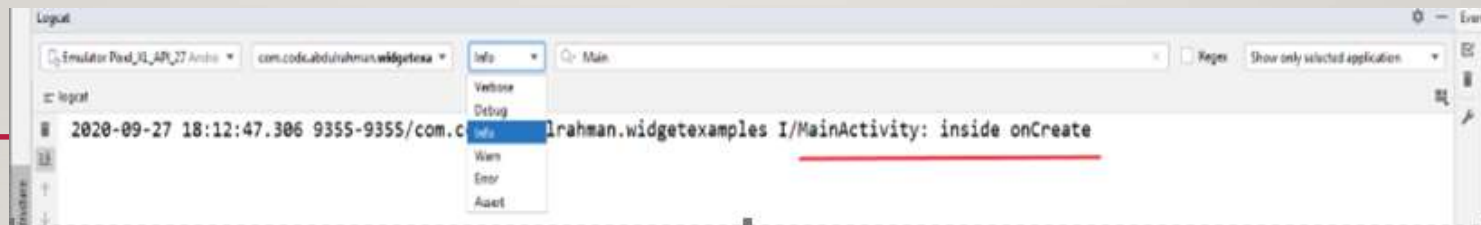
The code shows how the Log class can be used to log the exceptions to the Logcat window for error handling.

```
public void readingFile() {  
    final String logTAG = "MainActivity";  
    final String FILE_NAME = "fileName.txt";  
    try {  
        FileInputStream fis = openFileInput(FILE_NAME);  
        // do something with the file  
    } catch (FileNotFoundException fileNotFound) {  
        String errorMsg = FILE_NAME + " not found";  
        Log.e(logTAG, errorMsg, fileNotFound);  
        Toast toast = Toast.makeText(this, errorMsg, Toast.LENGTH_SHORT);  
        toast.show();  
    }  
}
```


4.12 THE LOG CLASS AND LOGCAT WINDOW

The Logcat window is shown in Fig. below.

The log message can be filtered based on message type (error, warning, verbose, etc.) or by the TAG specified in the log method calls.



4.12 USING ADB WITH LOG MESSAGES

- Log messages can also be viewed in the debugger by using the Android Debug Bridge (adb)
- The command `./ADB logcat` launches the Logcat window.
- You can redirect the output of the adb to a file using `>>` operator.
- For example, to redirect your log message to a text file in c:\temp directory
- cd to `Android\Sdk\platform-tools`
- Type:
- `adb logcat >> c:\temp\log.txt`
- The command line creates a text file with all log messages for further analysis.

PART 3: ANDROID APP TESTING

- Testing is a fundamental step in the software development process
- You need to test your code to make sure it works the way you expect
- There is a software development process that depends entirely on testing
- This framework, or methodology, is called Test-driven development (TDD)

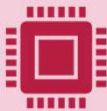
PART 3: ANDROID APP TESTING



When developing Android apps, various elements need to be tested. These include Intent, Activity lifecycles, event handlers, XML files, and null handling.



Depending on the application development stage, you will have a different type of testing.



For example, when different components developed by different teams are integrated together, you need to do integration testing. To deliver software to the user, you need to do user acceptance testing, etc.

PART 3: ANDROID APP TESTING

Unit testing is the testing type performed by the developers. While developers develop a solution, they must test their code.

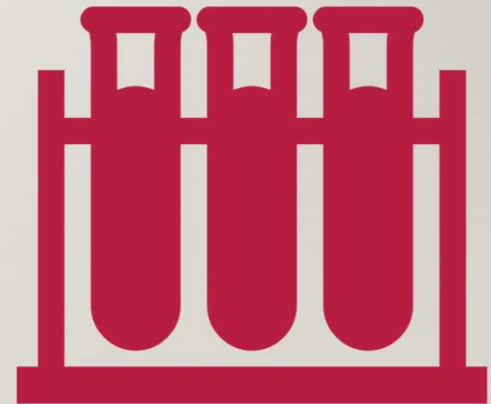
The piece of the code that is tested during a unit test is the smallest one.

It is either a method, part of a method, a class, or few classes.

PART 3: ANDROID APP TESTING

Junit framework enables you:

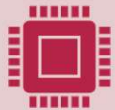
- Write unit tests
- Conduct automatic testing
- Re-run existing tests to make sure that previous tests are still working and that you are not breaking any previously tested code



PART 3: ANDROID APP TESTING



When you create a new Android project, Android Studio generates two folders for testing, *instrumental folder (androidTest)* and *unit test (test) folder*



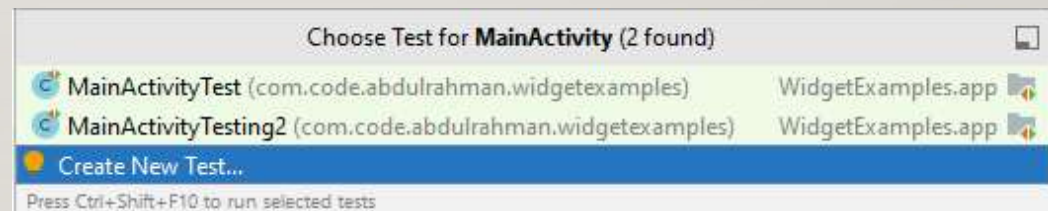
Instrumental testing is used to run tests that need to be run on Android devices or Android Virtual Devices



Unit testing tests that require only a Java virtual machine to run

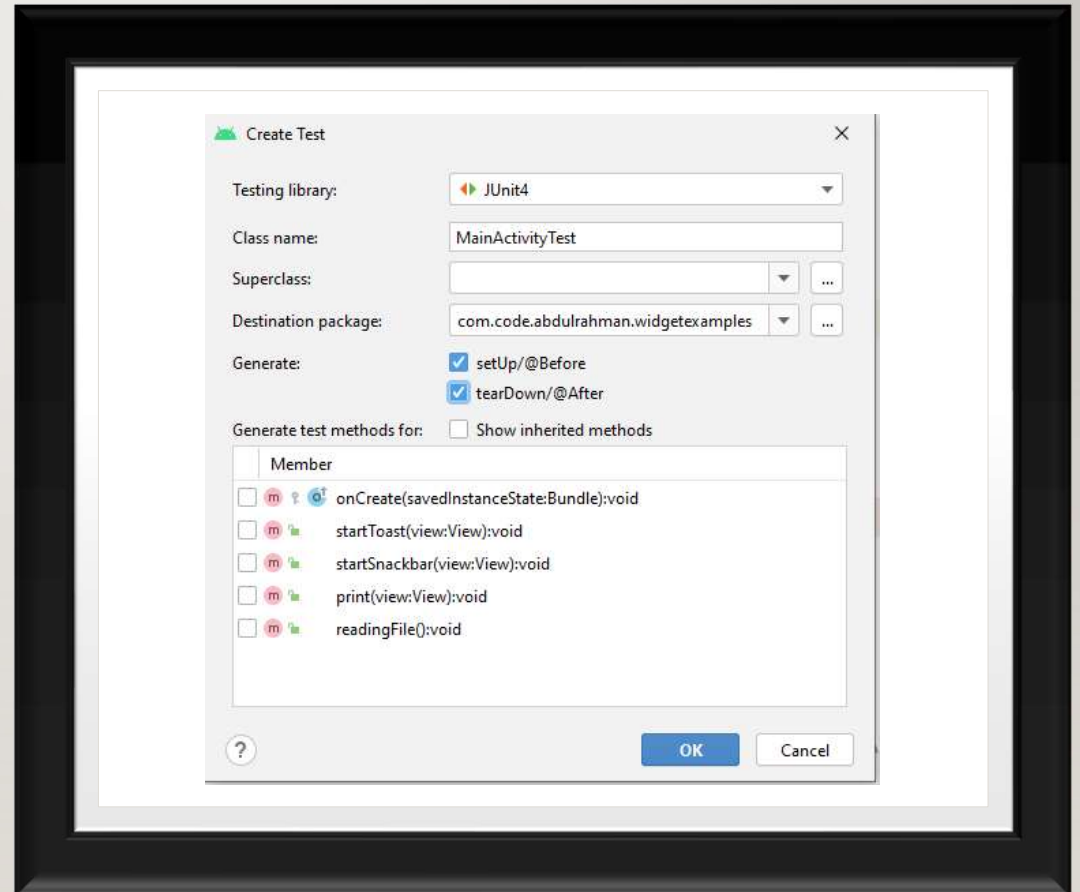
4.13 CREATE A TEST CLASS

- It is easy to create test cases and test your code in the Android Studio
- Once the Java class you want to unit test has a focus in the editor, on computers with the Windows operating system, click anywhere inside the class you want to test and **press Ctrl+Shift+T**, a window pops up to help you with the creation of a new test
- In the menu that appears click *Create New Test*, a test window will pop up for creating a test class,



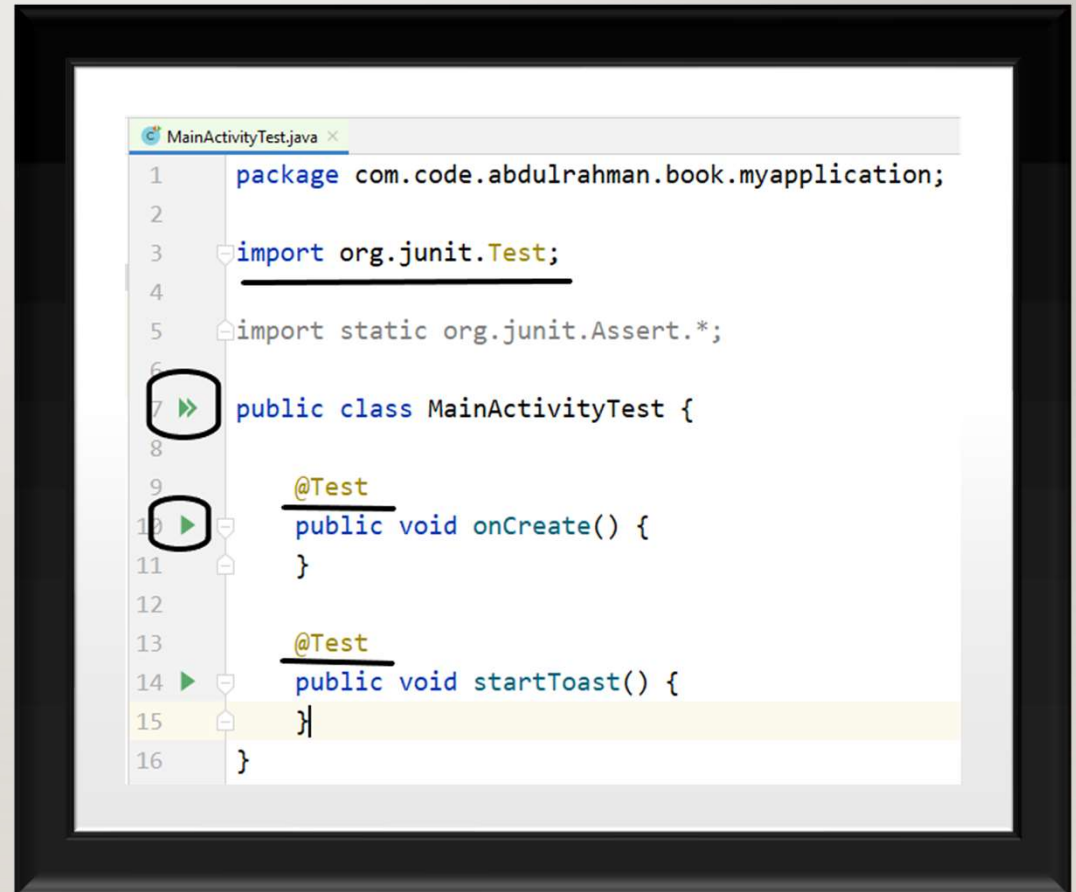
4.13 CREATE A TEST CLASS

- In the menu that appears select the methods you want to test and press OK.



4.13 CREATE A TEST CLASS

- The previous steps will generate a test class inside the instrumental directory.
- A snapshot of the generated class is listed below, it is a skeleton class that needs to be completed.



PART 3: ANDROID APP TESTING

You can also create a test class following a regular way of creating classes

Right-click on the directory where you want to create your testing class and create a class

Inside your testing class, declare your test methods

The method must be **public with a void return type**

Each test method should **start with annotation @test**

PART 3: ANDROID APP TESTING

- An example of a Unit test is shown. It includes the necessary libraries and classes needed for running unit tests.
- All the unique features and requirements for the test class are underlined.

```
MainActivity.java x ExampleInstrumentedTest.java x MainActivityTest.java x
1 package com.code.abdulrahman.widgetexamples;
2
3 import org.junit.Test;
4 import static org.junit.Assert.assertTrue;
5
6 public class MainActivityTest {
7     @Test
8     public void onCreate() {
9
10         assertTrue( condition: MainActivity.message.compareTo(
11             "Hello World") != 0);
12         assertTrue( condition: MainActivity.class.getName().compareTo(
13             "com.code.abdulrahman.widgetexamples.MainActivity") == 0);
14     }
15     @Test
16     public void startToast() {
17         assertTrue( condition: MainActivity.message.compareTo(
18             "message") == 0);
19     }
}
```

4.14 ASSERT METHODS



The assert methods compare the expected results, that is, what the programmer thinks the results will be, with the actual outcome of the program.



For example, a programmer can expect that adding two positive numbers will result in a number that is bigger than zero. So, it is possible to write a test case to add two numbers and assert the expected result to be positive.



In this case, the assert statement can be: **assertTrue (x+y > 0)**

4.14 ASSERT METHODS

- In the examples, we used `assertTrue()`, but that is not the only assert method that Android supports.
- The `org.junit.Assert` package provides assert methods for all primitive types, objects, and arrays.

`assertTrue(condition) or assertTrue(string message, condition)`

`assertFalse(condition) or assertFalse(string message, condition)`

`assertNull(object)`

`assertNotNull(object)`

`assertEquals(string message, object expected, object actual)`

`assertSame(object expected, object actual):`

`assertNotSame(object expected, object actual)`

`fail(condition)`

`assertEquals(double expected, double actual, double delta).`

4.15 HAMCREST ASSERT METHODS



To test your Android app, you can use a newer version of the assert methods using third-parties' libraries



The newer forms of the assert methods provide more clarity as to what the methods are supposed to do, and they improve the performance



For most, you don't have to do any additional tasks; these libraries are downloaded with the Android SDK or Android Studio.



You just need to learn the new syntax of the methods to apply them, and probably check to see if the library is included in the Gradle build file.

4.15 HAMCREST ASSERT METHODS

In next example, we show how to use `assertThat` from the Hamcrest library.

The first parameter to the *assertThat* method is the *addingTwoInteger()* method that we would like to test.

The second parameter uses methods from the ***org.hamcrest.Matchers*** class to enable successful testing.

The Matcher classes provide a large number of methods that can be utilized for your testing.

example, we show how
to use `assertThat` from
the Hamcrest library

```
package com.code.abdulrahman.widgetexamples;
import org.hamcrest.Matchers;
import org.junit.Test;
import static org.hamcrest.CoreMatchers.both;
import static org.hamcrest.CoreMatchers.is;
import static org.hamcrest.MatcherAssert.assertThat;

public class UsingAssertThatTest {
    @Test
    public void addingInteger() {
        assertThat(com.code.abdulrahman.widgetexamples.
            •      DataOperation.addingTwoInteger(3, 4), Matchers.is(7));
    }
    @Test
    public void usingGreaterThan() {
        // greaterThan and lessThan.
        assertThat(com.code.abdulrahman.widgetexamples.DataOperation.
            addingTwoInteger(10, 20), Matchers.greaterThan(25));
    }
    @Test
    public void UsinglessThan() {
        assertThat(com.code.abdulrahman.widgetexamples.DataOperation.
            addingTwoInteger(10, 20), Matchers.lessThan(30));
    }
    @Test
    public void range() {
        assertThat(com.code.abdulrahman.widgetexamples.DataOperation.
            addingTwoInteger(10, 20),
            both(Matchers.greaterThan(25)).and (Matchers.lessThan(30)));
    }
}
```

4.16 ESPRESSO TESTING

The third-party software Espresso can be used to test the app's user interface and generate test cases

To do testing, you need to connect the IDE to an Android device or Android virtual device, which means it is instrumental testing

**You can start recording Espresso testing by clicking on run
→ record Espresso test**

4.16 ESPRESSO TESTING

Run the app you want to test. For example, insert some data into an EditText, and click a button to display the content of the editText screen.

While you do these steps, your actions are recorded.

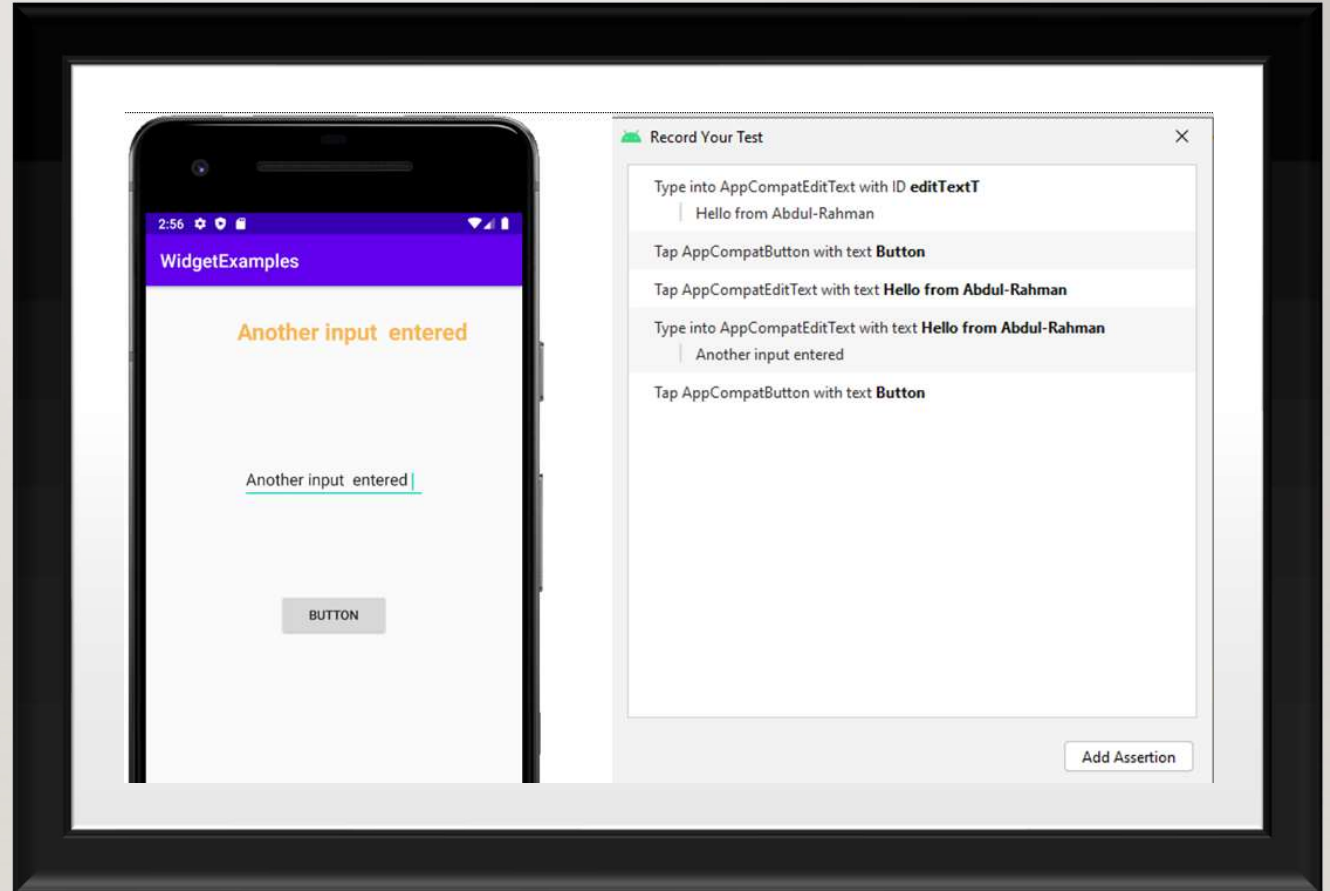
When you press the Ok button on the window recording screen, you will be prompted to give a name to your recording and a test class will be generated.

The test class will have all the actions you performed.

You can re-run the generated tests as many times as you want.

You can also modify the generated code to test different aspects of your GUI elements.

4.16 ESPRESSO TESTING



4.16 ESPRESSO TESTING

- Espresso uses Junit classes and has a small API
- To use Espresso, you need to use methods like *withId*, *onView*, *perform* and *check*
- You can use *withId* to match the View you would like to interact with, e.g., **withId(R.id.aButton)**. This is like the `findViewById` method that we have used in our apps
- To test a GUI component, you start with the `onView()` or `onData()` method.

For example, you can use `onView` and `withId` like this:

`onView(withId(R.id.myButton));`



4.16 TO USE ESPRESSO TESTING



Identify the type of action that you would like to have performed on the View object you are testing



For example, if you want to see a button clicked, then you pass `click()` to the `perform()` method from the **ViewInteraction** class



In this code statement `onView(withId(R.id.my_view)).perform(click())`



the `onView(withId (R.id.my_view))` method returns the **ViewInteraction** object.



The returned object is used to call the *perform* method with the `click()` method to simulate a click on a View

4.16 ESPRESSO TESTING

The `ViewInteraction` class has another method called `check()` that you can use to check the state of the `View` object.

For example, you can check to see if a `View` is being displayed

```
onView(withId(R.id.my_view)).perform(click())
```

```
.check(matches(isDisplayed()));
```



The code Listing shows how you can use all the constructs described above to do UI testing

The UI components used for testing are EditText and TextView

```
package com.code.abdulrahman.widgetexamples;
import androidx.test.ext.junit.rules.ActivityScenarioRule;
import org.junit.Rule;
import org.junit.Test;
import static androidx.test.espresso.Espresso.onView;
import static androidx.test.espresso.action.ViewActions.click;
import static androidx.test.espresso.action.ViewActions.closeSoftKeyboard;
import static androidx.test.espresso.action.ViewActions.typeText;
import static androidx.test.espresso.assertion.ViewAssertions.matches;
import static androidx.test.espresso.matcher.ViewMatchers.withId;
import static androidx.test.espresso.matcher.ViewMatchers.withText;
public class EspressoMainActivityTest {
    String inputText = "Hello Abdul-Rahman" ;
    @Rule
    public ActivityScenarioRule<EspressoMainActivity> activityScenarioRule
        = new ActivityScenarioRule<>(EspressoMainActivity.class);
    @Test
    public void onClick() {
        // Type text and then press the button.
        onView(withId(R.id.editTextT)).perform(typeText(inputText), closeSoftKeyboard());
        onView(withId(R.id.buttonT)).perform(click());
        // Check that the text was changed.
        onView(withId(R.id.textViewT)).check(matches(withText(inputText)));
    }
}
```

FOR MORE ON ESPRESSO TRAINING

- <https://developer.android.com/training/testing/espresso/>
- <https://developer.android.com/training/testing/espresso/cheat-sheet>

4.17 UNIT TESTING IN ANDROID STUDIO



To run tests **without a device or emulator** you use the second directory, the unit test directory in the Android project structure



Android uses the Junit framework for unit testing



To use Android classes and packages inside unit testing, you must make sure that you have Android libraries as a dependency on your project



add the following two lines of code into the dependencies section of your Gradle build file.

```
dependencies {  
    ...  
    androidTestImplementation 'com.android.support.test:rules:1.0.2'  
    androidTestImplementation 'com.android.support.test:runner:1.0.2'  
    ...  
}
```

<https://junit.org/junit4/javadoc/4.12/overview-tree.html>

4.17 UNIT TESTING IN ANDROD STUDIO

```
public class DataOperation {  
    public static int x, y;  
    public static int addingTwoInteger(int a, int b) {  
        x = a; y = b;  
        return (x + y);  
    }  
    public static int subtractingTwoInteger(int a, int b) {  
        x = a; y = b;  
        return (x - y);  
    }  
    public static int multiplyingTwoInteger(int a, int b) {  
        x = a; y = b;  
        return (x * y);  
    }  
}
```

4.17 UNIT TESTING IN ANDROID STUDIO

```
import org.junit.Test;
import static org.junit.Assert.*;
public class DataOperatoinTests {
    @Test
    public void addingtwoIntegers() {
        assertEquals(20, DataOperation.addingTwoInteger( 10, 10));
        assertTrue(DataOperation.x==10);
        assertTrue(DataOperation.y==10);
    }
    @Test
    public void subtractingIntegers() {
        assertEquals(5, DataOperation.subtractingTwoInteger( 50, 45));
        assertTrue(DataOperation.x==50);
        assertTrue(DataOperation.y==45);
    }
    @Test
    public void multiplyingAndAddingNumbers() {
        assertTrue((DataOperation.mltiplyingTwoInteger( 50, 45))!=5);
        assertTrue(DataOperation.x==50);
        assertTrue(DataOperation.y==45);
        assertTrue(DataOperation.y + DataOperation.x > 0);
    }
}
```

4.18 UNIT TESTING WITH MOCKITO

Mockito is a mocking framework for Java that allows you to create mock objects for testing purposes

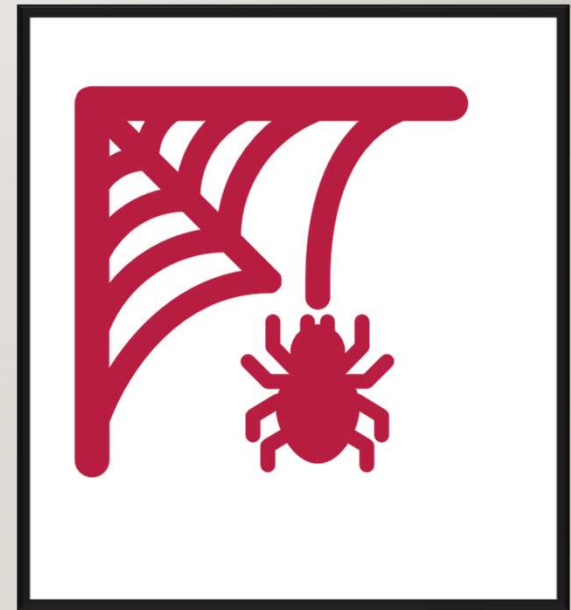
Whether you use Mockito objects or your Stub objects, mock objects are important for testing when the actual object you would like to test is not available yet.

You create mock objects when, for example,

- the database is not ready for your testing
- the server is not set up yet
- components that are developed by other groups or team members are not available yet
- etc

4.18 UNIT TESTING WITH MOCKITO

- Remember, object mocking is done during the development phase and in the testing environment
- It is part of the unit testing, and you should not fake anything in the production environment
- All mock objects should be removed from your production code



4.18 UNIT TESTING WITH MOCKITO

To use Mockito objects in your project, there are few steps you need to follow:

- you need to add the line of code below to the dependency section of your Gradle build file.

testImplementation "org.mockito:mockito-core:3.5.13"

- Add the `@RunWith (MockitoJUnitRunner.class)` annotation to the beginning of the test class:

@RunWith(MockitoJUnitRunner.class)
public class MockitoExample {...}

This will instruct the Mockito test runner to validate the proper usage of the framework's syntax and semantics. It will also help mock object initialization

4.18 UNIT TESTING WITH MOCKITO

- To create a **mock object** for an actual object or component that has not been written yet, add **@mock** annotation before the declaration of the variable. See the examples below:

@Mock

MyServer server; or

@Mock

SQLiteOpenHelper myDatabase ;

- If you want to **mock a method** and not an entire class or an object, use **@spy** instead of **@Mock** annotation like this:

@Spy List<String> alist = new ArrayList<String>();

- In Mockito, methods like *when()* and *thenReturn()* are used to enable object mocking.

4.18 UNIT TESTING WITH MOCKITO

In next example, we show how to use the Mockito object.



We created a class called `ClassUnderTesting` which has one field of type `AppCompatActivity` and one method called `getPath()`.



It uses its `getString()` method to get information about a file path from its internal object field and return it.



It uses `AppCompatActivity` to return some results, not necessarily the correct results.

4.18 UNIT TESTING WITH MOCKITO

// class under test is **appCompatActivity** that has not been written yet.
// we are able to test a class that has not been written yet.

```
import androidx.appcompat.app.AppCompatActivity;
public class ClassUnderTesting {
    AppCompatActivity appCompatActivity ;

    public ClassUnderTesting(AppCompatActivity aca) {
        this.appCompatActivity = aca;
    }
    public String getString () {
        System.out.println(ClassUnderTesting.class.getName() +
            appCompatActivity.getString(R.string.filePath));
        return appCompatActivity.getString(R.string.filePath) ;
    }
}
```

4.18 UNIT TESTING WITH MOCKITO

A test class called **MockitoExample** is created to test the `ClassUnderTesting`

A **mock object** is created to enable `ClassUnderTesting` to return a file path which can be any `String` value

The mock object is created using this statement:

```
ClassUnderTesting testingObject = new ClassUnderTesting(futureObject);
```

4.18 UNIT TESTING WITH MOCKITO

The **assertThat** statement in the MockitoExample class checks whether ClassUnderTesting can return the file path

The assert statement is like this:

```
String filePath = testingObject.getString();  
assertThat(filePath, is(fakePath));
```

4.18 Unit Testing with Mockito

```
import androidx.appcompat.app.AppCompatActivity;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.mockito.Mock;
import org.mockito.Mockito;
import org.mockito.junit.MockitoJUnitRunner;
import java.util.ArrayList;
import java.util.List;
import static org.hamcrest.MatcherAssert.assertThat;
import static org.hamcrest.Matchers.is;
import static org.junit.Assert.assertEquals;
import static org.mockito.Mockito.when;
```

@RunWith(MockitoJUnitRunner.class)

```
public class MockitoExample {
```

```
    private static final String fakePath = "fakePath";
```

```
    @Mock
```

```
    protected AppCompatActivity futureObject;
```

```
    @Test
```

```
    public void gettingStringFromSecondActivity() {
```

```
        // get filepath from the mock object
```

```
        when(futureObject.getString(R.string.filePath))
```

```
            .thenReturn(fakePath);
```

```
        ClassUnderTest TestingObject = new ClassUnderTest(futureObject);
```

```
        // checking that mock object is returning correct file path
```

```
        String filePath = TestingObject.getString();
```

```
        assertThat(filePath, is(fakePath));
```

```
    }
```

```
    @Test
```

```
    public void testMockMethod(){
```

```
        List mockList = Mockito.mock(ArrayList.class);
```

```
        mockList.add("Hello class");
```

```
        Mockito.verify(mockList).add("Hello class");
```

```
        assertEquals(0, mockList.size());
```

```
    }
```

```
}
```


4.18 Unit Testing with Mockito



Different from the listing above where an object is mocked, in the code snippet below, the *spy* method from the Mockito framework is used to mock a method instead of an object.



This approach is useful if you want to mock a method only

```
@Test
public void testSpyMethod() {
    ArrayList myArrayList = Mockito.spy(new
    ArrayList());
    myArrayList.add("Hello class");
    Mockito.verify(myArrayList).add("Hello class");
    assertEquals(1, myArrayList.size());
}
```

<https://javadoc.io/doc/org.mockito/mockito-core/latest/org/mockito/Mockito.html>

4.19 CODE COVERAGE



Code coverage is a useful metric or measurement for getting information on how well your project is tested



It is used during software development and when you have access to the source code



It allows you to estimate the relevant parts of the source code that have never been executed by your test cases, thus, facilitating further testing and improvement of test cases



Android Studio has a built-in capability that allows you to run tests with code coverage

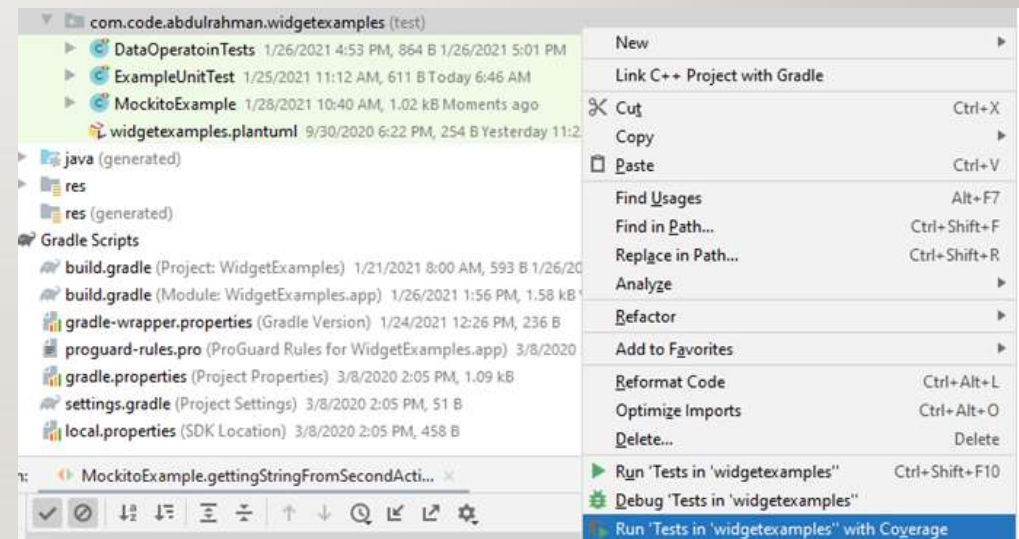


The results will be displayed in a window showing the percentage of the classes, methods, and code lines that have been tested by your test cases

4.19 CODE COVERAGE

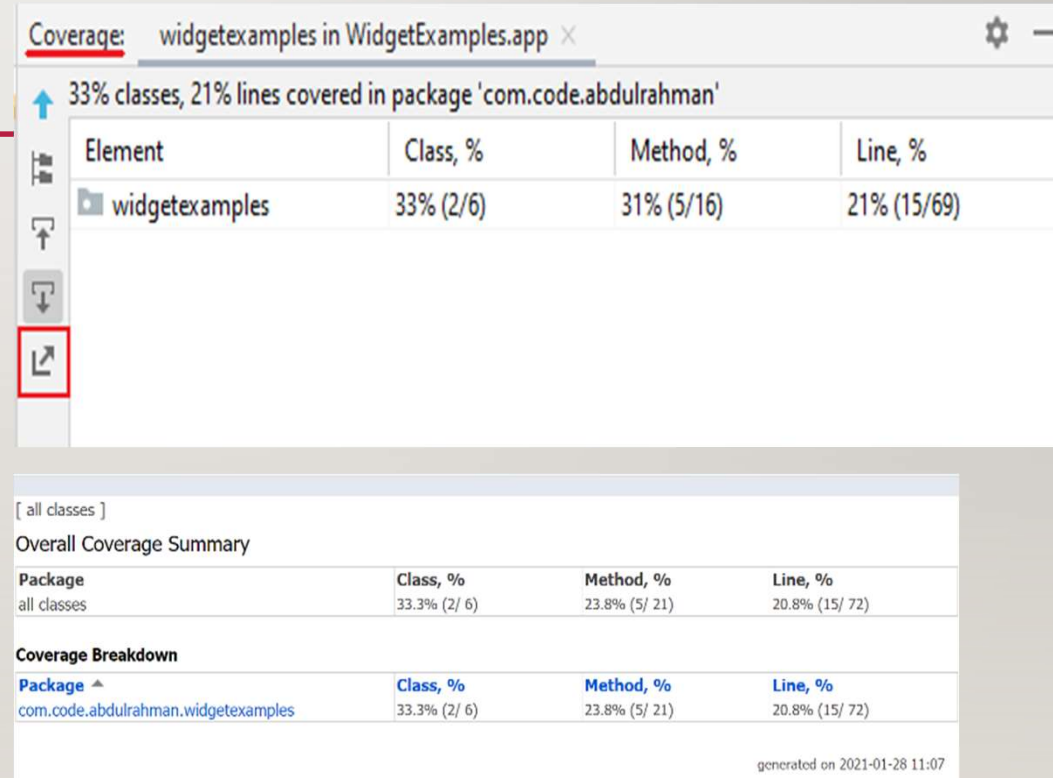
Right-click on the unit test folder and select *Run tests with coverage*

The results will be displayed in a window showing the percentage of the classes, methods, and code lines that have been tested by your test cases.



4.19 CODE COVERAGE

- You can send the coverage result to an HTML file using the output button on the coverage window
- This is a useful option for documenting your test results.
- The output button is squared in red in the Figure above



4.19 CODE COVERAGE

- Using code coverage as the main testing metric, however, does not give a definitive answer to how well your code is tested.
- For example, if you covered 95% of the test cases but the most important tests are within the 5% that you did not test, then you have not done a good job testing even though the coverage rate is at a very high percentage.
- Code coverage to find out about the parts of your code that are untested can still be useful.



4.20 CODE INSPECTION AND REFACTORING

Code Refracting is an important approach to improve the quality of your code

It refers to changing your code during development

rename your classes, methods, and class attributes,

re-organize the structure of your classes

moving classes to different packages and move methods to different classes

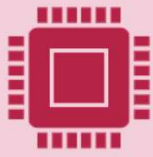
4.20 CODE INSPECTION AND REFACTORING

changes are done to

- improve the quality of code
- improve the readability of the code
- reduce coupling
- improve cohesion performance



4.20 CODE INSPECTION AND REFACTORING



An empirical study on how often code refactoring is conducted during Android app development shows that it is very popular among developers, and on average an app undergoes 47.79 refactoring operations



Refactoring is easily done when using Android Studio.

For example, right-click on the class that you would like to rename and type the new name for your class. The change will be applied everywhere in your project.

4.20 CODE INSPECTION AND REFACTORING



Similar to code refactoring, static analysis of code is an important technique to improve the quality of your code by detecting potential bugs and errors in the code



To analyze your code, on the Android Studio menu bar click on *Analyze* → *inspect code*. An inspection window will pop up.



Select the scope of the inspection:




whole project,
custom scope

4.20 CODE INSPECTION AND REFACTORING

An inspection result window shows all kinds of shortcomings in your code if any:

- Hardcoding
- unused resources
- redundant variables
- empty methods
- unused imports
- typos
- performance issues
- etc.

Resolving all or as many as you can result in better coding and better quality of your app

- > **Android Lint: Internationalization** 4 warnings
- > **Android Lint: Performance** 5 warnings
- > **Android Lint: Usability** 4 warnings
- > **Gradle** 1 warning
- > **HTML** 4 errors 10 warnings
- ✓ **Java** 44 warnings 1 weak warning
 - > **Code maturity** 6 warnings 1 weak warning
 - > **Compiler issues** 4 warnings
 - > **Data flow** 1 warning
 - ✓ **Declaration redundancy** 13 warnings
 - > Declaration can have 'final' modifier 6 warnings
 - > Empty method 4 warnings
 - ✓ **Unused declaration** 3 warnings
 - > + Entry Points
 - >  EspressoMainActivity 1 warning
 - >  MainActivity 1 warning
 - >  SecondActivity 1 warning
 - > **Java language level migration aids** 3 warnings
 - > **Naming conventions** 5 warnings
 - > **Probable bugs** 3 warnings
 - > **Resource management** 1 warning
 - > **Test frameworks** 8 warnings
 - > **Proofreading** 227 typos

4.2| REVERSE ENGINEERING



The last topic of this lesson is about using a reverse engineering technique to find out the anomalies in your code structure and design



For small projects, design and implementation most likely go hand in hand; but typically, you have to develop app design first then code it

4.2I REVERSE ENGINEERING



Different from the typical software development process, reverse engineering is about decomposing existing codes and programs to understand how it works, its parts, and the structure

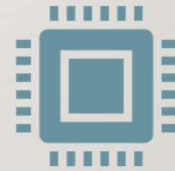


If you find that your code has performance issues, memory leaks, or usability issues and you don't have the proper design documents, generating the class diagram of your code, or parts of your code, will be a useful way to understand the code

4.2| REVERSE ENGINEERING



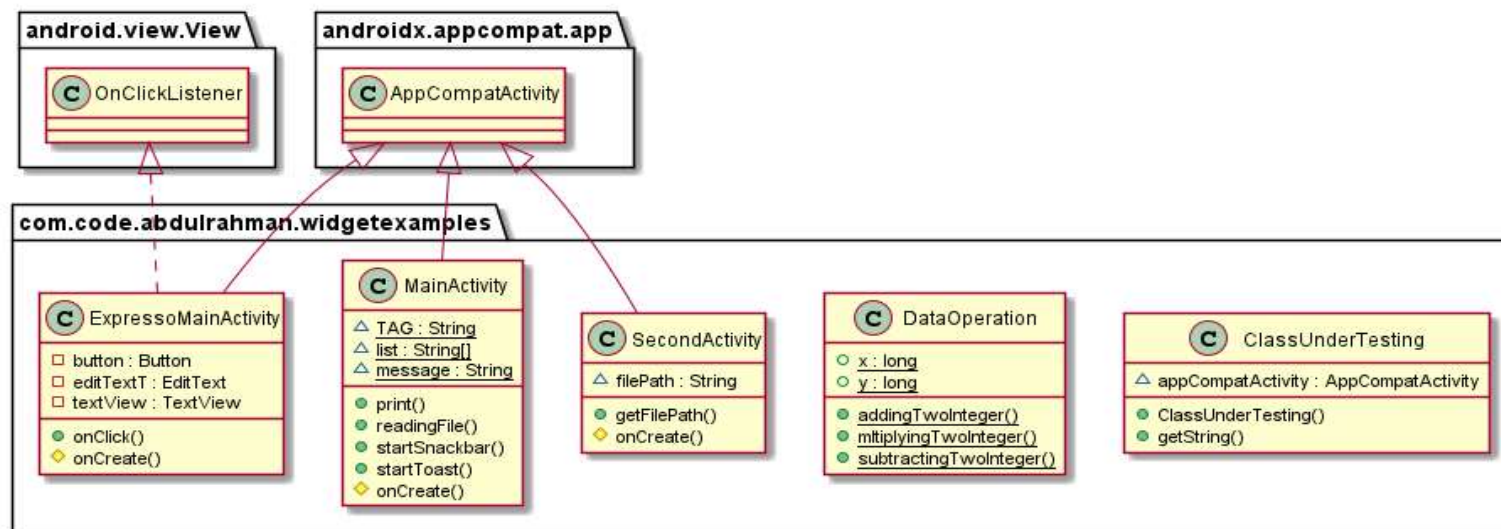
Generating the UML class diagram for your code, or parts of your code will help you to understand the relationship between the classes and the structure of your app



Android Studio allows you to generate UML class diagrams from code using plugins

4.2| REVERSE ENGINEERING

An Example of Class Diagram generated by SketchUML from PlantUML file



4.4 CHAPTER SUMMARY

In this chapter, we described various Android Studio tools and Android classes to debug the app, i.e., find and fixing errors in code, and to profile your code to find out the performance and usability issues of the app

We studied Android Studio Debugger, Android Profiler, Device File Explorer, Android Debug Bridge, Toast and Snackbars classes, the Log class and the LogCat utility

We also studied unit testing using Junit and Espresso, and Mockito tools for testing UI and creating mock objects, as well as code coverage, reverse engineering, code refactoring, and code inspection

CHECK YOUR KNOWLEDGE

Below are some of the fundamental concepts and vocabularies that have been covered in this chapter. To test your knowledge and your understating of this chapter you should be able to describe each of the below concepts in one or two sentences.

**Android
Debug
Bridge**

**Code
Coverage**

**Code
Inspection**

Debugging

**Device File
Explorer**

Espresso

File Explorer

GraphVis

**Instrumental
testing**

**Integration
testing**

CHECK YOUR KNOWLEDGE

Junit

Log

Logcat

Mockito

PlantUML

Profiler

Refactoring

**Reverse
engineering**

Sketch It

Snakbar

Toast

Unit Testing

**User
acceptance
testing**

FURTHER READING



For more information about the topics covered in this chapter, we suggest that you refer to the online resources listed at the end of the chapter



The links are mostly maintained by Google and are a part of the Android API specification



The resource titles convey which section/subsection of the chapter the resource is related to