
UNIT 9 TESTING AND DEBUGGING

- 9.0 Introduction
- 9.1 Objectives
- 9.2 What is Testing?
- 9.3 How to test Android application?
- 9.4 Unit Testing
 - 9.4.1 Check Your Progress
- 9.5 How to set up your Testing Environment?
 - 9.5.1 Video – V9: Android Unit Testing
 - 9.5.2 Check Your Progress
- 9.6 What is Debugging?
 - 9.6.1 Check Your Progress
- 9.7 What is Logcat?
- 9.8 Summary
- 9.9 Further readings

9.0 INTRODUCTION

This unit emphasizes the importance of the application testing and debugging. It provides you an opportunity to identify errors and faults in a developed application. It also introduces how to use testing tools and techniques to test Android Application and how to apply measures to rectify identified errors and faults.

9.1 OBJECTIVES

After studying this unit, you should be able to :



Outcomes

- differentiate testing and debugging an application
- set up the testing environment to develop Android applications
- write unit tests to test your Android programme
- perform debugging referring to log messages in Logcat



Terminology

- | | |
|------------------|--|
| error: | mistake in the program |
| fault: | usually a hardware problem happening at run time |
| emulator: | hardware or software that enables one computer system to behave like another |

9.2 WHAT IS TESTING?

Testing is one of the phases in software development life cycle that requires substantial amount of time before releasing software to users. To find the software bugs we use the process of executing a program or application. We have to test the software with test data to verify that a given set of input to a given function produces

the expected result. There are two testing approaches; static and dynamic testing. An introduction to static and dynamic testing is given in the next section

Static and Dynamic testing

Static testing is done basically to test the requirement specifications, test plan, user manual etc. They are not executed, but tested with the set of some tools and processes. Reviews, walkthroughs and inspections are some example processes. These processes are not discussed in this material.

Dynamic Testing is when execution is done on the software code as a technique to detect defects and to determine quality attributes of the code.

With dynamic testing methods software is executed using a set of inputs and its output and then compared with the expected results. There are various levels of dynamic testing techniques. Some of them are unit testing, integration testing, system testing and acceptance testing. Here we will be only focusing on how different dynamic testing techniques can be used in an Android application.

Now you know that techniques can be used in testing software. Let's learn how to test an Android application using above techniques.

9.3 HOW TO TEST ANDROID APPLICATION?

Testing an application on multiple physical devices at one place is not practical. Testing an application to run on different devices is referred to as device compatibility which is discussed in detail in a different unit. In order to avoid this practical barrier, Android SDK provides an emulator to test your application against all versions of Android on different devices. An emulator provides a virtual environment to test your application. It eliminates the requirement of a real physical device. This emulator uses an Android Virtual Device (AVD) to run an application. In order to do that it is required to create an AVD. Creating an AVD was discussed under an earlier section.

In addition, Amazon and various other companies maintain device farms to test applications with automation as well as manual testing.

An Android application should be tested for its functionality, user interfaces, performance etc. based on the generated test cases. It is application developers' responsibility to perform the unit tests and a separate testing team will be responsible of performing certain other types of testing such as functional testing, integration testing, acceptance testing etc.

9.4 UNIT TESTING

Unit tests are the fundamental tests in your app testing strategy. By creating and running unit tests against your code, you can easily verify that the logic of individual units is correct. Running unit tests after every build helps you to quickly catch and fix software regressions introduced by code changes to your app.

A unit test generally exercises the functionality of the smallest possible unit of code (which could be a method, class, or component) in a repeatable way. You should build unit tests when you need to verify the logic of specific code in your app. For example, if you are unit testing a class, your test might check that the class is in the right state.

Typically, the unit of code is tested in. After completing this section, you will be able to perform unit testing for your application.

Android unit tests are based on JUnit and to test Android apps, you typically create these types of automated unit tests:

- **Local tests:** Unit tests that run on your local machine only. These tests are compiled to run locally on the Java Virtual Machine (JVM) to minimize execution time. Use this approach to run unit tests that have no dependencies on the Android framework or have dependencies that can be filled by using mock objects.
- **Instrumented tests:** Unit tests that run on an Android device or emulator. These tests have access to instrumentation information, such as theContext for the app under test. Use this approach to run unit tests that have Android dependencies which cannot be easily filled by using mock objects.

You should write your unit or integration test class as a JUnit 4 test class in JUnit framework. This framework offers a convenient way to perform common setup, teardown, and assertion operations in your test.

9.4.1 Check Your Progress



Differentiate the use of local test from instrumented test when performing a unit test of an Android application.

9.5 HOW TO SET UP YOUR TESTING ENVIRONMENT?

In your Android Studio project, you must store the source files for instrumented tests at *module-name/src/androidTests/java/*. This directory already exists when you create a new project.

Before you begin, you should download the Android Testing Support Library Setup, which provides APIs that allow you to quickly build and run instrumented test code for your apps. The Testing Support Library includes a JUnit 4 test runner (AndroidJUnitRunner) and APIs for functional UI tests (Espresso and UI Automator).

You also need to configure the Android testing dependencies for your project to use the test runner and the rules APIs provided by the Testing Support Library. To simplify your test development, you should also include the Hamcrest library, which lets you create more flexible assertions using the Hamcrest matcher APIs.

In your app's top-level build.gradle file, you need to specify these libraries as dependencies:

```
dependencies {  
    androidTestCompile 'com.android.support:support-  
annotations:24.0.0'  
    androidTestCompile 'com.android.support.test:runner:0.5'  
    androidTestCompile 'com.android.support.test:rules:0.5'  
    // Optional -- Hamcrest library  
    androidTestCompile 'org.hamcrest:hamcrest-library:1.3'
```

```
// Optional -- UI testing with Espresso
androidTestCompile
'com.android.support.test.espresso:espresso-core:2.2.2'
// Optional -- UI testing with UI Automator
androidTestCompile
'com.android.support.test.uiautomator:uiautomator-v18:2.1.2'
}
```

To use JUnit 4 test classes, make sure to specify [AndroidJUnitRunner](#) as the default test instrumentation runner in your project by including the following setting in your app's module-level `build.gradle` file:

```
android {
    defaultConfig {
        testInstrumentationRunner
        "android.support.test.runner.AndroidJUnitRunner"
    }
}
```

A basic JUnit 4 test class is a Java class that contains one or more test methods. A test method begins with the `@Test` annotation and contains the code to exercise and verify a single functionality (that is, a logical unit) in the component that you want to test.

The code snippet below shows an example JUnit 4 integration test that uses the Espresso APIs to perform a click action on a UI element, and check whether an expected string is displayed.

```
@RunWith(AndroidJUnit4.class)
@LargeTest
public class MainActivityInstrumentationTest {

    @Rule
    public ActivityTestRule mActivityRule
    = new ActivityTestRule<>(MainActivity.class);

    @Test
    public void sayHello() {
        onView(withText("Sayhello!")).perform(click());

        onView(withId(R.id.textView)).check(matches(withText("Hello, World!")));
    }
}
```

In your JUnit 4 test class, you can call out sections in your test code for special processing by using the following annotations:

@Before: Use this annotation to specify a block of code that contains test setup operations. The test class invokes this code block before each test. You can have multiple `@Before` methods but the order in which the test class calls these methods is not guaranteed.

@After: This annotation specifies a block of code that contains test tear-down operations. The test class calls this code block after every test method. You can define

multiple `@After` operations in your test code. Use this annotation to release any resources from memory.

`@Test`: Use this annotation to mark a test method. A single test class can contain multiple test methods, each prefixed with this annotation.

`@Rule`: Rules allow you to flexibly add or redefine the behavior of each test method in a reusable way. In Android testing, use this annotation together with one of the test rule classes that the Android Testing Support Library provides, such as `ActivityTestRule` or `ServiceTestRule`.

`@BeforeClass`: Use this annotation to specify static methods for each test class to invoke only once. This testing step is useful for expensive operations such as connecting to a database.

`@AfterClass`: Use this annotation to specify static methods for the test class to invoke only after all tests in the class have run. This testing step is useful for releasing any resources allocated in the `@BeforeClass` block.

`@Test(timeout=)`: Some annotations support the ability to pass in elements for which you can set values. For example, you can specify a timeout period for the test. If the test starts but does not complete within the given timeout period, it automatically fails. You must specify the timeout period in milliseconds, for example:

`@Test(timeout=5000)`.

Instrumented unit tests

Unit tests that run on an Android device or emulator can take advantage of the Android framework APIs and supporting APIs, such as the Android Testing Support Library. You should create instrumented unit tests if your tests need access to instrumentation information (such as the target app's Context) or if they require the real implementation of an Android framework component (such as a Parcelable or SharedPreferences object). These tests have access to Instrumentation information, such as the Context of the app you are testing. Use these tests when your tests have Android dependencies that mock objects cannot satisfy.

Because instrumented tests are built into a stand-alone APK, they must have an AndroidManifest.xml file. However, Gradle automatically generates this file during the build so it is not visible in your project source set. You can add your own manifest file if necessary, such as to specify a different value for `minSdkVersion` or register run listeners just for your tests. When building your app, Gradle merges multiple manifest files into one manifest.

Create an Instrumented Unit Test Class

Your instrumented unit test class should be written as a JUnit 4 test class. To learn more about creating JUnit 4 test classes and using JUnit 4 assertions and annotations, see Create a Local Unit Test Class.

To create an instrumented JUnit 4 test class, add the `@RunWith(AndroidJUnit4.class)` annotation at the beginning of your test class definition. You also need to specify the `AndroidJUnitRunner` class provided in the

Android Testing Support Library as your default test runner. This step is described in more detail in [Getting Started with Testing](#).

The following example shows how you might write an instrumented unit test to test that the `Parcelable` interface is implemented correctly for the `LogHistory` class:

```
import android.os.Parcel;
import android.support.test.runner.AndroidJUnit4;
import android.util.Pair;
import org.junit.Test;
import org.junit.runner.RunWith;
import java.util.List;
import static org.hamcrest.Matchers.is;
import static org.junit.Assert.assertThat;

@RunWith(AndroidJUnit4.class)
@SmallTest
public class LogHistoryAndroidUnitTest {

    public static final String TEST_STRING = "This is a string";
    public static final long TEST_LONG = 12345678L;
    private LogHistory mLogHistory;

    @Before
    public void createLogHistory() {
        mLogHistory = new LogHistory();
    }

    @Test
    public void logHistory_ParcelableWriteRead() {
        // Set up the Parcelable object to send and receive.
        mLogHistory.addEntry(TEST_STRING, TEST_LONG);

        // Write the data.
        Parcel parcel = Parcel.obtain();
        mLogHistory.writeToParcel(parcel,
mLogHistory.describeContents());

        // After you're done with writing, you need to reset
the parcel for reading.
        parcel.setDataPosition(0);

        // Read the data.
        LogHistory createdFromParcel
=LogHistory.CREATOR.createFromParcel(parcel);
        List<Pair<String, Long>> createdFromParcelData =
createdFromParcel.getData();

        // Verify that the received data is correct.
        assertThat(createdFromParcelData.size(), is(1));
        assertThat(createdFromParcelData.get(0).first, is(TEST_S
TRING));
        assertThat(createdFromParcelData.get(0).second, is(TEST_
LONG));
    }
}
```

Create a test suite

To organize the execution of your instrumented unit tests, you can group a collection of test classes in a *test suite* class and run these tests together. Test suites can be nested. That is your test suite can group other test suites and run all their component test classes together.

A test suite is contained in a test package, similar to the main application package. By convention, the test suite package name usually ends with the suite suffix (e.g. com.example.android.testing.mysample.suite).

To create a test suite for your unit tests, import the JUnit `RunWith` and `Suite` classes. In your test suite, add the `@RunWith(Suite.class)` and the `@Suite.SuiteClasses()` annotations. In the `@Suite.SuiteClasses()` annotation, list the individual test classes or test suites as arguments.





The following example shows how you might implement a test suite called `UnitTestSuite` that groups and runs the `CalculatorInstrumentation-Test` and `CalculatorAddParameterizedTest` test classes together.

```
import
com.example.android.testing.mysample.CalculatorAddParameterized
Test;
import
com.example.android.testing.mysample.CalculatorInstrumentationT
est;
import org.junit.runner.RunWith;
import org.junit.runners.Suite;

// Runs all unit tests.
@RunWith(Suite.class)
@Suite.SuiteClasses({CalculatorInstrumentationTest.class,
    CalculatorAddParameterizedTest.class})
public class UnitTestSuite {}
```

Run Instrumented Unit Tests

To run your instrumented tests, follow these steps:

1. Be sure your project is synchronized with Gradle by clicking **Sync Project**  in the toolbar.
2. Run your test in one of the following ways:
 - o To run a single test, open the **Project** window, and then right-click a test and click **Run** .
 - o To test all methods in a class, right-click a class or method in the test file and click **Run** .
 - o To run all tests in a directory, right-click on the directory and select **Run tests** .


The Android Plugin for Gradle compiles the instrumented test code located in the default directory (`src/androidTest/java/`), builds a test APK and production APK, installs both APKs on the connected device or emulator, and runs the tests. Android Studio then displays the results of the instrumented test execution in the *Run* window.

Note: While running or debugging instrumented tests, Android Studio does not inject the additional methods required for Instant Run and turns the feature off.



By default, Android Studio sets up new projects to deploy to the Emulator or a physical device with just a few clicks. With Instant Run, you can push changes to methods and existing app resources to a running app without building a new APK, so code changes are visible almost instantly.

Instant Run

Introduced in Android Studio 2.0, Instant Run is a behavior for the Run  and

Debug  commands that significantly reduces the time between updates to your app. Although your first build may take longer to complete, Instant Run pushes subsequent updates to your app without building a new APK, so changes are visible much more quickly.

Instant Run is supported only when you deploy the debug build variant, use Android Plugin for Gradle version 2.0.0 or higher, and set `minSdkVersion` to 15 or higher in your app's module-level `build.gradle` file. For the best performance, set `minSdkVersion` to 21 or higher.

After deploying an app, a small, yellow thunderbolt icon appears within the Run  button (or Debug  button), indicating that Instant Run is ready to push updates the next time you click the button. Instead of building a new APK, it pushes just those new changes and, in some cases, the app doesn't even need to restart but immediately shows the effect of those code changes.

Instant Run pushes updated code and resources to your connected device or emulator by performing a hot swap, warm swap, or cold swap. It automatically determines the type of swap to perform based on the type of change you made.

9.5.1 Video – V9: Android Unit Testing



In this video you will be shown how to setup testing environment and how to write a unit test. You may watch this video and do activity 9.2.

URL: <https://tinyurl.com/yaatacnv>



9.5.2 Check Your Progress



Create an Android application “MyApp” with a class “ConversionUtil” to perform the given two functionalities.

- To convert centimeters into inches [write a method ConvertCmtoInch()]
- To convert inches into centimeters [write a method ConvertInchtoCm()]


Then write local unit tests to check whether the written functionalities provide the expected output. Use the values given as inputs and expected output to test the method.

Functionality to test	Input	Output
Convert centimeters into inches	10 centimeters	3.93701 inches
Convert inches into centimeters	10 inches	25.4 centimeters

9.6 WHAT IS DEBUGGING?

It is the procedure of finding defects in a source code and removing them. Android Studio includes a debugger that allows you to debug apps running on the Android Emulator or a connected Android device. With the Android Studio debugger, you can:

- Select a device to debug your app on.
- Set breakpoints in your code.
- Examine variables and evaluate expressions at runtime.
- Capture screenshots and videos of your app.

To start debugging, click **Debug**  in the toolbar. Android Studio builds an APK, signs it with a debug key, installs it on your selected device, then runs it and opens the **Debug** window.

If no devices appear in the **Select Deployment Target** window after you click **Debug**, then you need to either connect a device or click **Create New Emulator** to setup the Android Emulator.

9.6.1 Check Your Progress



How to enable USB debugging in your device?

9.7 WHAT IS LOGCAT?

Logcat is a command-line tool that dumps a log of system messages, including stack traces when the device throws an error and messages that you have written from your app with the Log class.

This page is about the command-line logcat tool, but you can also view log messages from the Logcat window in Android Studio. For information about viewing and filtering logs from Android Studio

You can run logcat as an adb command or directly in a shell prompt of your emulator or connected device. To view log output using adb, navigate to your SDK platform-tools/ directory and execute:

```
$ adb logcat
```

You can create a shell connection to a device and execute:

```
$ adb shell  
# logcat
```

How to write Log Messages?

The Log class allows you to create log messages that appear in the logcat window. Generally, you should use the following log methods, listed in order from the highest to lowest priority (or, least to most verbose):

- Log.e → (error)
- Log.w → (warning)
- Log.i → (information)
- Log.d → (debug)
- Log.v → (verbose)

You should never compile verbose logs into your app, except during development. Debug logs are compiled in but stripped at runtime, while error, warning and info logs are always kept.

For each log method, the first parameter should be a unique tag and the second parameter is the message. The tag of a system log message is a short string indicating the system component from which the message originates (for example, ActivityManager). Your tag can be any string that you find helpful, such as the name of the current class.

A good convention is to declare a TAG constant in your class to use in the first parameter. For example, you might create an information log message as follows:

```
private static final String TAG = "MyActivity";  
...  
Log.i(TAG, "MyClass.getView() — get item number " + position);
```

Note: Tag names greater than 23 characters are truncated in the logcat output.

9.8 SUMMARY



Android Studio is designed to make testing simple. This unit explained how to set up a JUnit test that runs on the local JVM or an instrumented test that runs on a device.

9.9 FURTHER READINGS

<https://developer.android.com/studio/test?authuser=1>

<https://developer.android.com/training/testing>

<https://www.softwaretestinghelp.com/android-app-testing/>



ignou
THE PEOPLE'S
UNIVERSITY