

CS304

Software Engineering

TAN, Shin Hwei

陈馨慧

Southern University of Science and Technology

Slides adapted from

Jadx: <https://github.com/skylot/jadx>

Jd-gui: <http://java-decompiler.github.io/> Apktool:

<https://ibotpeaches.github.io/Apktool/> Dex2jar

<https://github.com/pxb1988/dex2jar>

Axmlprinter2 <https://code.google.com/archive/p/android4me/downloads>

How to reverse engineer an Android app?

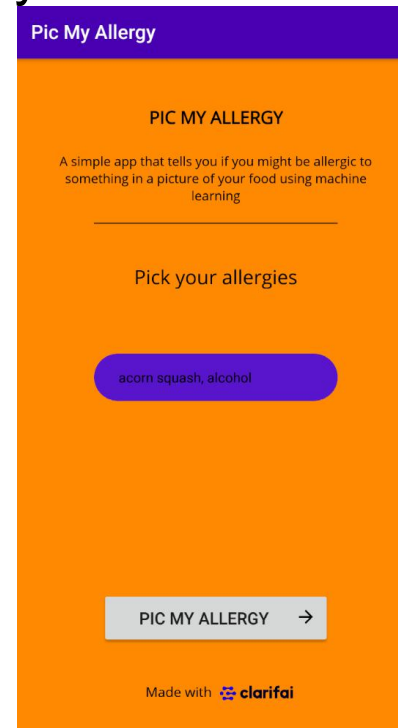
How to learn from other App?

When you are developing an Android App and do not know how to design your UI.

There is an App, you really like its UI, but you only have the APK.

What will you do?

Right, I want to look at its code!



Try to Reverse Engineer an app

Get the app from:

<https://classroom.github.com/a/4Dfhej1A>

Follow the steps in the following slides

Reverse Engineering for APK file

Reverse Engineering, also called Decompile.

It is the process by which a man-made object is deconstructed to reveal its designs, architecture, or to extract knowledge from the object.

In this lab, we will learn how to get the source code from an APK file

The components of APK

First, rename your .apk file to .zip

Then try to extract the zip file

You will get

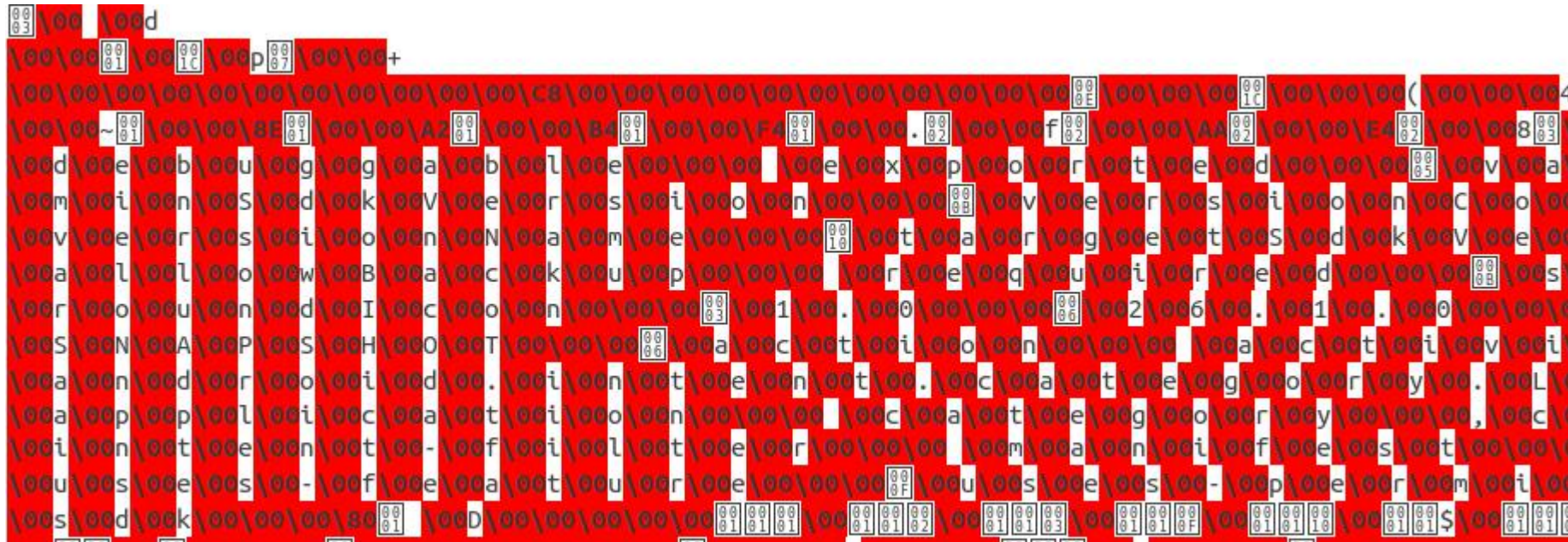
Name	Size	Type	Modified
assets	33.8 kB	Folder	
META-INF	145.3 kB	Folder	
res	1.2 MB	Folder	
AndroidManifest.xml	3.4 kB	XML docum...	30 十一月 1979, 00:00
classes.dex	4.0 MB	unknown	30 十一月 1979, 00:00
resources.arsc	268.6 kB	unknown	30 十一月 1979, 00:00

It seems we can get the AndroidManifest.xml easily. Let's try to open it directly.

The components of APK

Oops, the file looks very strange.

Because currently it is binary code .



The components of APK

To translate the binary code to human readable format, we need AXMLPrinter2.jar

Terminal command:

```
Java -jar AXMLPrinter2.jar AndroidManifest.xml ->  
AndroidManifest.txt
```

Also, you can decompile layout xml within directory **res** by this method.

How can you get program code?

You may have a question now.

Where is the program code?

Because Android system runs on the Dalvik Virtual Machine, so we have .dex file, it just like .jar to JVM

Name	Size	Type	Modified
assets	33.8 kB	Folder	
META-INF	145.3 kB	Folder	
res	1.2 MB	Folder	
AndroidManifest.xml	3.4 kB	XML docum...	30 十一月 1979, 00:00
classes.dex	4.0 MB	unknown	30 十一月 1979, 00:00
resources.arsc	268.6 kB	unknown	30 十一月 1979, 00:00

How can you get program code?

By the tool, dex2jar.jar

We can translate .dex file to .class file

Terminal command: `./d2j-dex2jar.sh classes.dex`

Name	Size	Type	Modified
assets	33.8 kB	Folder	
META-INF	145.3 kB	Folder	
res	1.2 MB	Folder	
AndroidManifest.xml	3.4 kB	XML docum...	30 十一月 1979, 00:00
classes.dex	4.0 MB	unknown	30 十一月 1979, 00:00
resources.arsc	268.6 kB	unknown	30 十一月 1979, 00:00

I still cannot get what I want!

Now we get classes-dex2jar.jar (the default output name)

Name	Size	Type	Modified
android	5.0 MB	Folder	
butterknife	39.4 kB	Folder	
clarifai2	757.2 kB	Folder	
com	507.0 kB	Folder	
dagger	169.3 kB	Folder	
javax	20.3 kB	Folder	
okhttp3	497.8 kB	Folder	
okio	94.8 kB	Folder	
timber	11.5 kB	Folder	

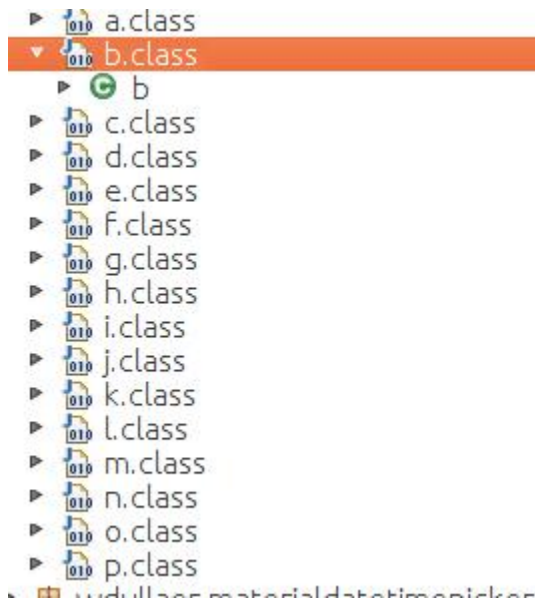
You may think that finally we can know how they implement some function. But... you will find that all these files end with .class

Use jd-gui(<http://java-decompiler.github.io/>) to browse for code

Do you think it always works?

If it works, it's the best. But obviously not.

If the APK file is encrypted, you will get something like this.



???

```
return this._a;
}

public void a(int paramInt1, int paramInt2)
{
    if (paramInt1 == paramInt2) {
        return;
    }
    Collections.swap(this._a, paramInt2, paramInt1);
}

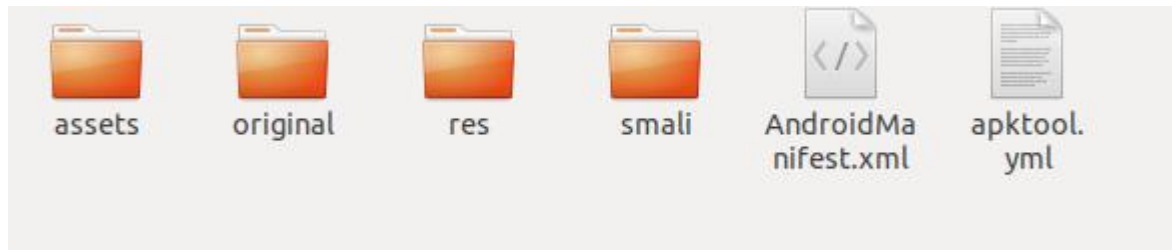
public void a(int paramInt, a parama)
{
    this._a.add(paramInt, parama);
}

public void a(int paramInt, Task paramTask)
{
    b(paramInt, new a(paramTask));
}
```

Do you think it is too complicated?

Try apktool.jar, it directly extracts readable layout file from .apk, and you will get a smali directory, which contains the code. If you are good at assemble language, you may modify .smali to change the behavior of the App!

Terminal cmd: `java -jar apktool.jar d xx.apk`



Do you think it is too complicated?

Or it is still too complex.

Try JADX(<https://github.com/skylot/jadx>), which gives all you want!

Terminal Command: `jadx/bin/jadx xx.apk`

You need to “cd”(Change directory) to right path

Try the steps in the reverse engineering video played during the lecture!

Given the app “[Pic My Allergy_v1.0_apkpure.com.apk](#)”,

Run the app to see how it works

For example, in the video: Enter two inputs and show “..**invalid**...”

Use the knowledge gained from the run

➤ For example, in the video: App A has part that process a string “..**invalid**...”

Decompile the application

Use code browser(Jadx/Jd-gui) to check for the code

Jadx: <https://github.com/skylot/jadx>

Jd-gui: <http://java-decompiler.github.io/>

Search for the string (for example, in App A in the video, the string is “invalid”)

Find the location where the string is at

What to submit for this part?

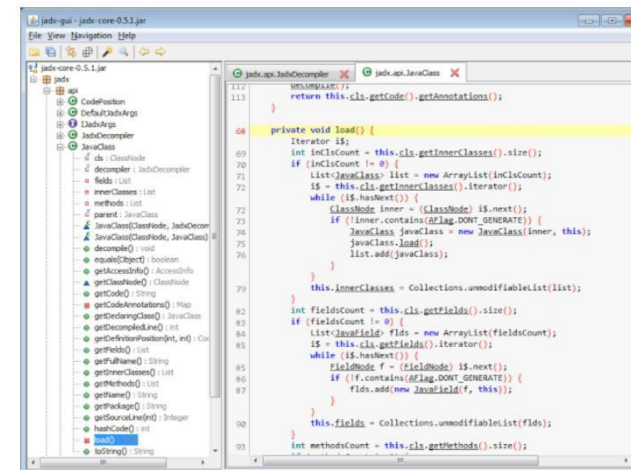
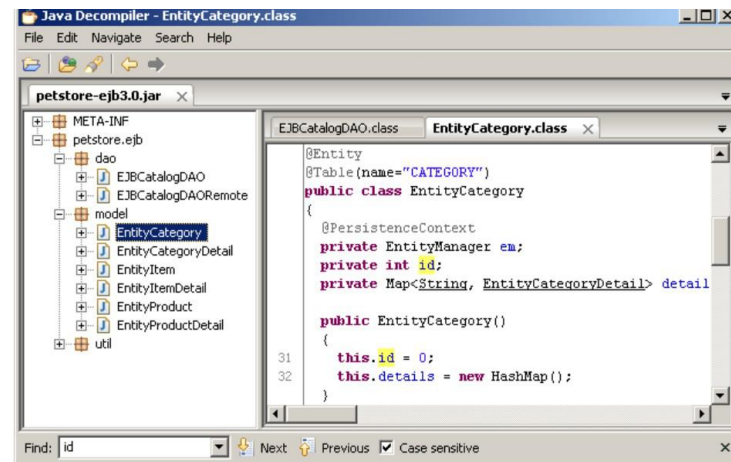
Remove the [Pic My Allergy_v1.0_apkpure.com.apk](http://PicMyAllergy_v1.0_apkpure.com.apk)

Include a README.md with your **name** and **id**

Include the answer for the following question in the README.md

What is the string that you have searched for in the app?

Can you show a screenshot where you searched for the location of the string? Add the screenshot to the README.md



Include a "*.gitignore" to ignore irrelevant files for Android app

Unit Testing in Android

Question from students: How do I write unit tests in Android for my project?

Adapted from <https://developer.android.com/studio/test>

Types of tests

Local unit tests

Located at **module-name**/src/test/java/.

These are tests that **run on your machine's local Java Virtual Machine (JVM)**. Use these tests to minimize execution time when your tests have **no Android framework dependencies** or when you can mock the Android framework dependencies.

At runtime, these tests are executed against a modified version of android.jar where all final modifiers have been stripped off. This lets you use popular mocking libraries, like Mockito.

What is mock objects in testing?

Mock objects are simulated objects that mimic the behavior of real objects in controlled ways, most often as part of a software testing initiative

Instrumented tests

Located at **module-name**/src/androidTest/java/.

These are tests that **run on a hardware device or emulator**. These tests have access to Instrumentation APIs, give you access to information such as the Context of the app you are testing, and let you control the app under test from your test code. Use these tests when writing integration and functional UI tests to automate user interaction, or when your tests have Android dependencies that mock objects cannot satisfy.

Because instrumented tests are built into an APK (separate from your app APK), they must have their own 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.

Unit Tests v.s. Instrumented Test

When you create a new project or add an app module, Android Studio creates the test source sets listed above and includes an example test file in each. You can see them in the **Project** window as shown in figure 1.

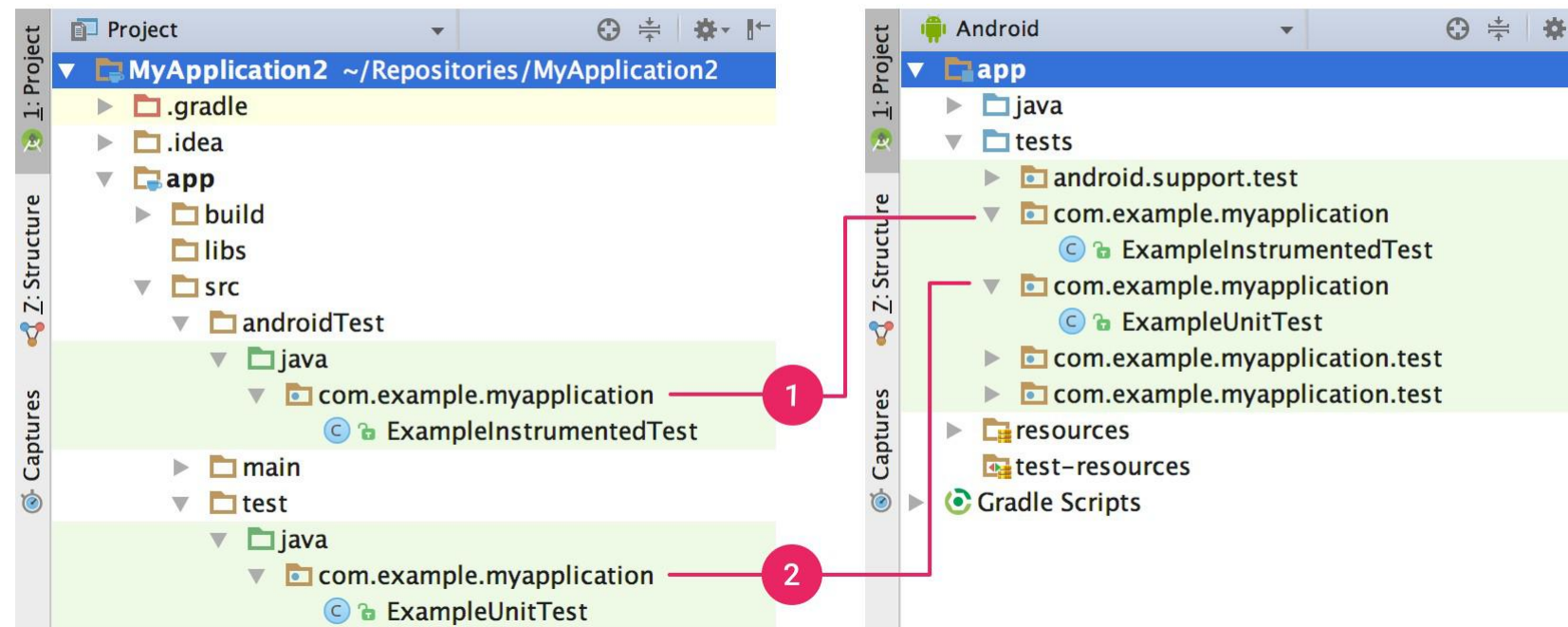


Figure 1. Your project's (1) instrumented tests and (2) local JVM tests are visible in either the **Project** view (left) or **Android** view (right).

How to add a new Test in Android Studio?

Add a new test

To create either a local unit test or an instrumented test, you can create a new test for a specific class or method by following these steps:

1. Open the Java file containing the code you want to test.
2. Click the class or method you want to test, then press **Ctrl+Shift+T** (⇧⌘T).
3. In the menu that appears, click **Create New Test**.
4. In the **Create Test** dialog, edit any fields and select any methods to generate, and then click **OK**.
5. In the **Choose Destination Directory** dialog, click the source set corresponding to the type of test you want to create: **androidTest** for an instrumented test or **test** for a local unit test. Then click **OK**.

Alternatively, you can create a generic Java file in the appropriate test source set as follows:

1. In the **Project** window on the left, click the drop-down menu and select the **Project** view.
2. Expand the appropriate module folder and the nested **src** folder. To add a local unit test, expand the **test** folder and the nested **java** folder; to add an instrumented test, expand the **androidTest** folder and the nested **java** folder.
3. Right-click on the Java package directory and select **New > Java Class**.
4. Name the file and then click **OK**.

Add test library dependencies





Also be sure you specify the test library dependencies in your app module's `build.gradle` file:

```
dependencies {  
    // Required for local unit tests (JUnit 4 framework)  
    testImplementation 'junit:junit:4.12'  
  
    // Required for instrumented tests  
    androidTestImplementation 'com.android.support:support-annotations:24.0.0'  
    androidTestImplementation 'com.android.support.test:runner:0.5'  
}
```

How to Run a test?

Run a test

To run a test, proceed as follows:

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:
 - In the **Project** window, right-click a test and click **Run** .
 - In the Code Editor, right-click a class or method in the test file and click **Run**  to test all methods in the class.
 - To run all tests, right-click on the test directory and click **Run tests** .

By default, your test runs using Android Studio's default run configuration. If you'd like to change some run settings such as the instrumentation runner and deployment options, you can edit the run configuration in the **Run/Debug Configurations** dialog (click **Run > Edit Configurations**).

How to view test coverage?

View test coverage

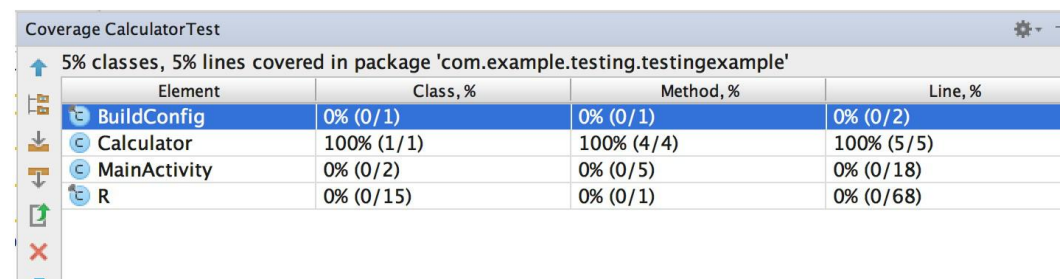
The test coverage tool is available for local unit tests to track the percentage and areas of your app code that your unit tests have covered. Use the test coverage tool to determine whether you have adequately tested the elements, classes, methods, and lines of code that make up your app.

There are a number of ways to run a unit test, and they are described on the IntelliJ [Running with Coverage](#) page. The following procedure shows how to run a unit test inline from the editor:

1. Double-click the unit test you want to run.
2. In the editor, place your cursor in the line you want to run with coverage.
 - If you place your cursor in the class declaration, all test methods in that class run.
 - If you place your cursor in a method declaration, all code in that method runs.
 - If you place your cursor on a particular line within a method, only that line runs.
3. Right-click the line where you placed your cursor.
4. In the context menu, choose **Run test-name with coverage**.

The [coverage tool window](#) appears.

Figure 2 shows the coverage tool window for a calculator unit test that tests for addition, subtraction, multiplication, and division.



Element	Class, %	Method, %	Line, %
BuildConfig	0% (0/1)	0% (0/1)	0% (0/2)
Calculator	100% (1/1)	100% (4/4)	100% (5/5)
MainActivity	0% (0/2)	0% (0/5)	0% (0/18)
R	0% (0/15)	0% (0/1)	0% (0/68)

How to see the test results?

View the test results

When you run a JUnit or **instrumented** test, the results appear in the **Run** window. A green bar means all tests succeeded and a red bar means at least one test failed. Figure 3 shows a successful test run.

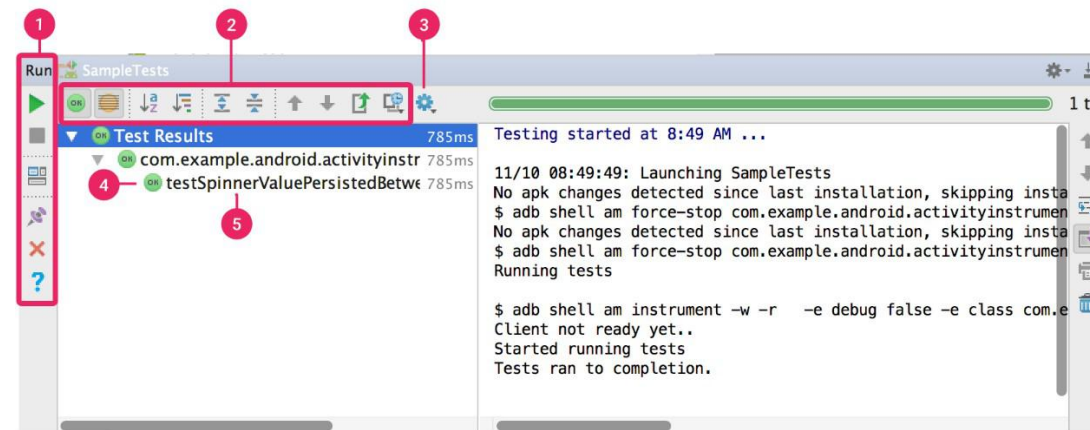



Figure 3. Test results appear in the Run window.

The **Run** window displays the tests in a tree view on the left, and the results and messages for the current test suite in the output pane on the right. Use the toolbars, context menus, and status icons to manage the test results, as follows:

- 1 Use the **run toolbar** to rerun the current test, stop the current test, rerun failed tests (not shown because it is available for unit tests only), pause output, and dump threads.
- 2 Use the **testing toolbar** to filter and sort test results. You can also expand or collapse nodes, show test coverage, and import or export test results.
- 3 Click the **context menu**  to track the running test, show inline statistics, scroll to the stacktrace, open the source code at an exception, auto scroll to the source, and select the first failed test when the test run completes.
- 4 **Test status icons** indicate whether a test has an error, was ignored, failed, is in progress, has passed, is paused, was terminated, or was not run.
- 5 Right-click a line in the tree view to display a context menu that lets you run the tests in debug mode, open the test source code file, or jump to the line in the source code being tested.

Check Progress Report in Sakai!

Discuss with your group members and start implementing the features following the coding standard!