Lesson 3: Testing, debugging, and using support libraries

3.1: The Android Studio debugger

Contents:

In this chapter you'll learn about debugging your apps in Android Studio.

About debugging

Debugging is the process of finding and fixing errors (bugs) or unexpected behavior in your code. All code has bugs, from incorrect behavior in your app, to behavior that excessively consumes memory or network resources, to actual app freezing or crashing.

Bugs can result for many reasons:

- Errors in your design or implementation
- Android framework limitations (or bugs)
- Missing requirements or assumptions for how the app should work
- Device limitations (or bugs)

Use the debugging, testing, and profiling capabilities in Android Studio to help you reproduce, find, and resolve all of these problems. Those capabilities include:

- The **Logcat** pane for log messages
- The **Debugger** pane for viewing frames, threads, and variables
- Debug mode for running apps with breakpoints
- Test frameworks such as JUnit or Espresso
- Dalvik Debug Monitor Server (DDMS), to track resource usage

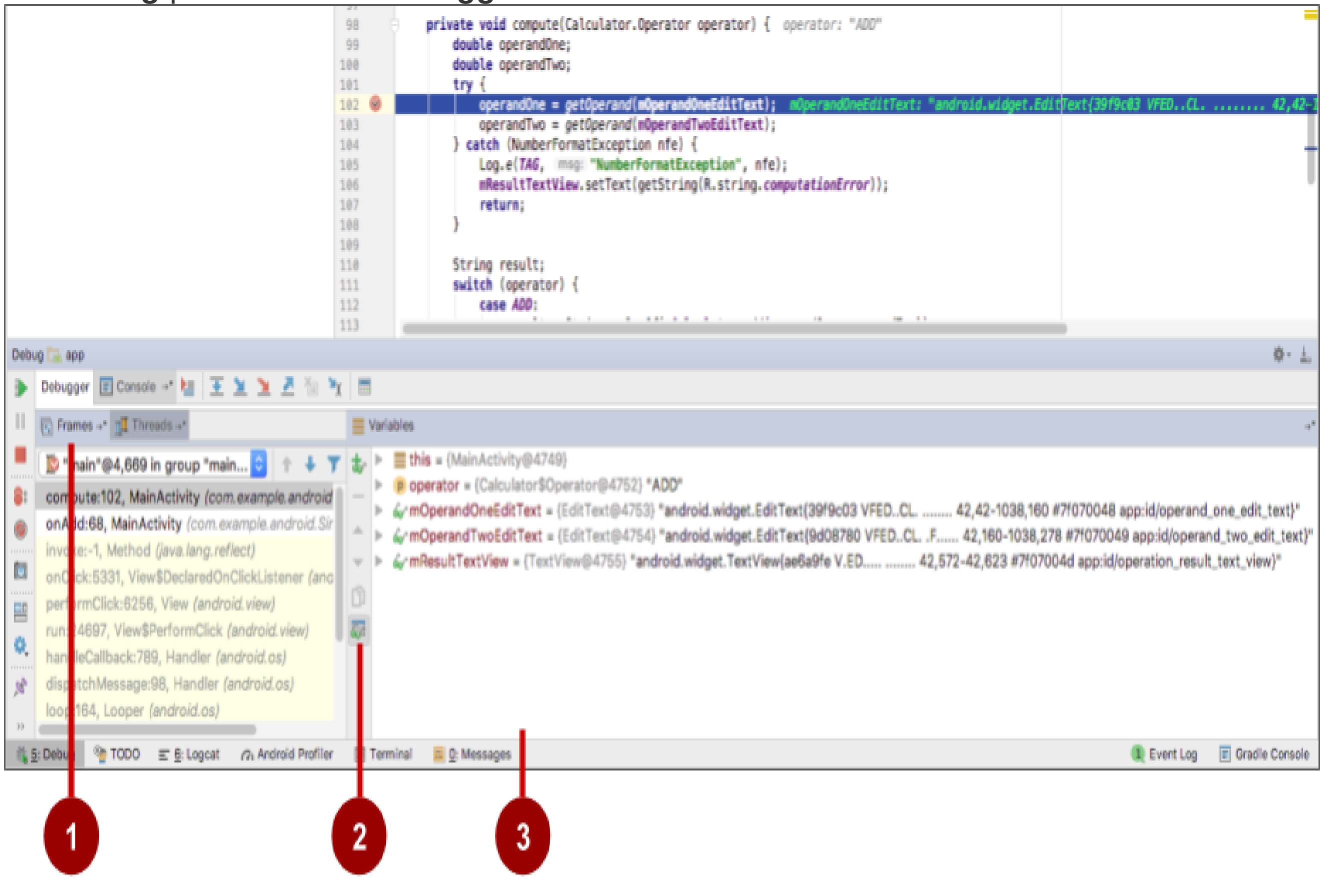
In this chapter you learn how to debug your app with the Android Studio debugger, set and view breakpoints, step through your code, and examine variables.

Running the debugger

Running an app in debug mode is similar to running the app. You can either run an app in debug mode, or attach the debugger to an already-running app.

Run your app in debug mode

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** pane with the **Debugger** and **Console** tabs.



```
private void compute(Calculator.Operator operator) {   operator: "ADD"
    double operandOne;
    double operandTwo;
    try {
        operandOne = getOperand(mOperandOneEditText);   mOperandOneEditText: "android.widget.EditText{39f9c03 VFED..CL. ........ 42,42-1
        operandTwo = getOperand(mOperandTwoEditText);
    } catch (NumberFormatException nfe) {
        Log.e(TAG, msg: "NumberFormatException", nfe);
        mResultTextView.setText(getString(R.string.computationError));
        return;
    }

    String result;
    switch (operator) {
        case ADD:
```

```
"main"@4,669 in group "main...
compute:102, MainActivity (com.example.android
onAdd:68, MainActivity (com.example.android.Sir
invoke:-1, Method (java.lang.reflect)
onClick:5331, View$DeclaredOnClickListener (anc
performClick:6256, View (android.view)
run:24697, View$PerformClick (android.view)
handleCallback:789, Handler (android.os)
dispatchMessage:98, Handler (android.os)
loop:164, Looper (android.os)
```

```
this = {MainActivity@4749}
operator = {Calculator$Operator@4752} "ADD"
mOperandOneEditText = {EditText@4753} "android.widget.EditText{39f9c03 VFED..CL. ........ 42,42-1038,160 #7f070048 app:id/operand_one_edit_text}"
mOperandTwoEditText = {EditText@4754} "android.widget.EditText{9d08780 VFED..CL. .F...... 42,160-1038,278 #7f070049 app:id/operand_two_edit_text}"
mResultTextView = {TextView@4755} "android.widget.TextView{ae6a9fe V.ED..... ........ 42,572-42,623 #7f07004d app:id/operation_result_text_view}"
```

The figure above shows the **Debug** pane with the **Debugger** and **Console** tabs. The **Debugger** tab is selected, showing the **Debugger** pane with the following features:

1. **Frames** tab**:** Click to show the **Frames** pane with the current execution stack frames for a given thread. The execution stack shows each class and method that have been called in your app and in the Android runtime, with the most recent method at the top.

Click the **Threads** tab to replace the **Frames** pane with the **Threads** pane.

2. **Watches** button**:** Click to show the **Watches** pane within the **Variables** pane, which shows the values for any variable watches you have set. Watches allow you to keep track of a specific variable in your program, and see how that variable changes as your program runs.

3. **Variables** pane**:** Shows the variables in the current scope and their values. Each variable in this pane has an expand icon to expand the list of object properties for the variable. Try expanding a variable to explore its properties.

Debug a running app

If your app is already running on a device or emulator, start debugging that app with these steps:

1. Select **Run > Attach debugger to Android process** or click the **Attach** icon in the toolbar.
2. In the **Choose Process** dialog, select the process to which you want to attach the debugger.

By default, the debugger shows the device and app process for the current project, as well as any connected hardware devices or virtual devices on your computer. Check the **Show all processes** option to show all processes on all devices.

3. Click **OK**. The **Debug** pane appears as before.

Resume or stop debugging

To resume executing an app after debugging it, select **Run > Resume Program** or click the Resume icon.

To stop debugging your app, select **Run > Stop** or click the Stop icon in the toolbar.

Using breakpoints

Android Studio supports several types of breakpoints that trigger different debugging actions. The most common type is a breakpoint that pauses the execution of your app at a specified line of code. While paused, you can examine variables, evaluate expressions, then continue execution line by line to determine the causes of runtime errors.

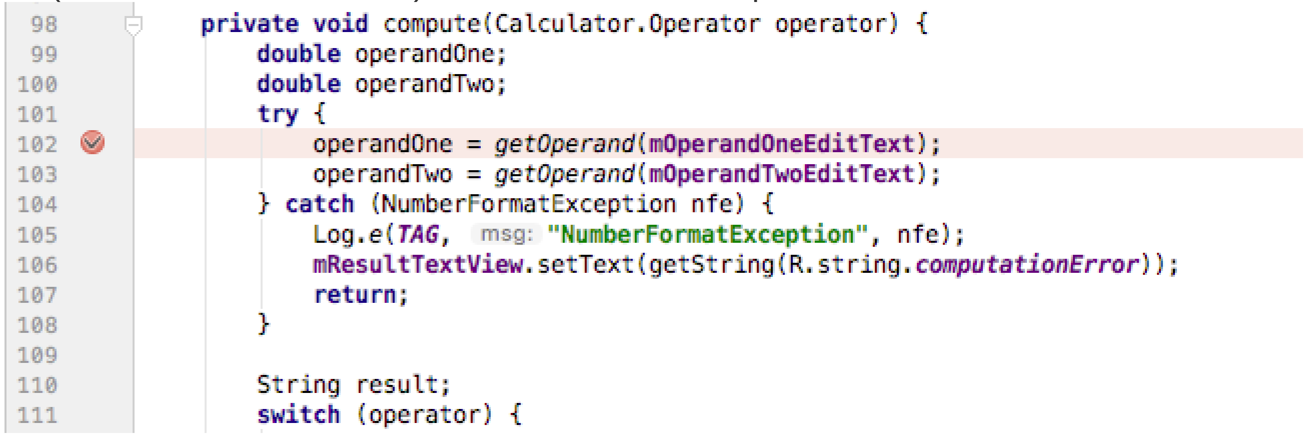
You can set a breakpoint on any executable line of code.

Add breakpoints

To add a breakpoint to a line in your code, use these steps:

1. Locate the line of code where you want to pause execution.
2. Click in the left gutter of the editor pane at that line, next to the line numbers. A red dot appears at that line, indicating a breakpoint. The red dot includes a check mark if the app is already running in debug mode.

As an alternative, you can choose **Run > Toggle Line Breakpoin**t or press **Control-F8** (**Command-F8** on a Mac) to set or clear a breakpoint at a line.



```
private void compute(Calculator.Operator operator) {
    double operandOne;
    double operandTwo;
    try {
        operandOne = getOperand(mOperandOneEditText);
        operandTwo = getOperand(mOperandTwoEditText);
    } catch (NumberFormatException nfe) {
        Log.e(TAG, msg: "NumberFormatException", nfe);
        mResultTextView.setText(getString(R.string.computationError));
        return;
    }

    String result;
    switch (operator) {
```

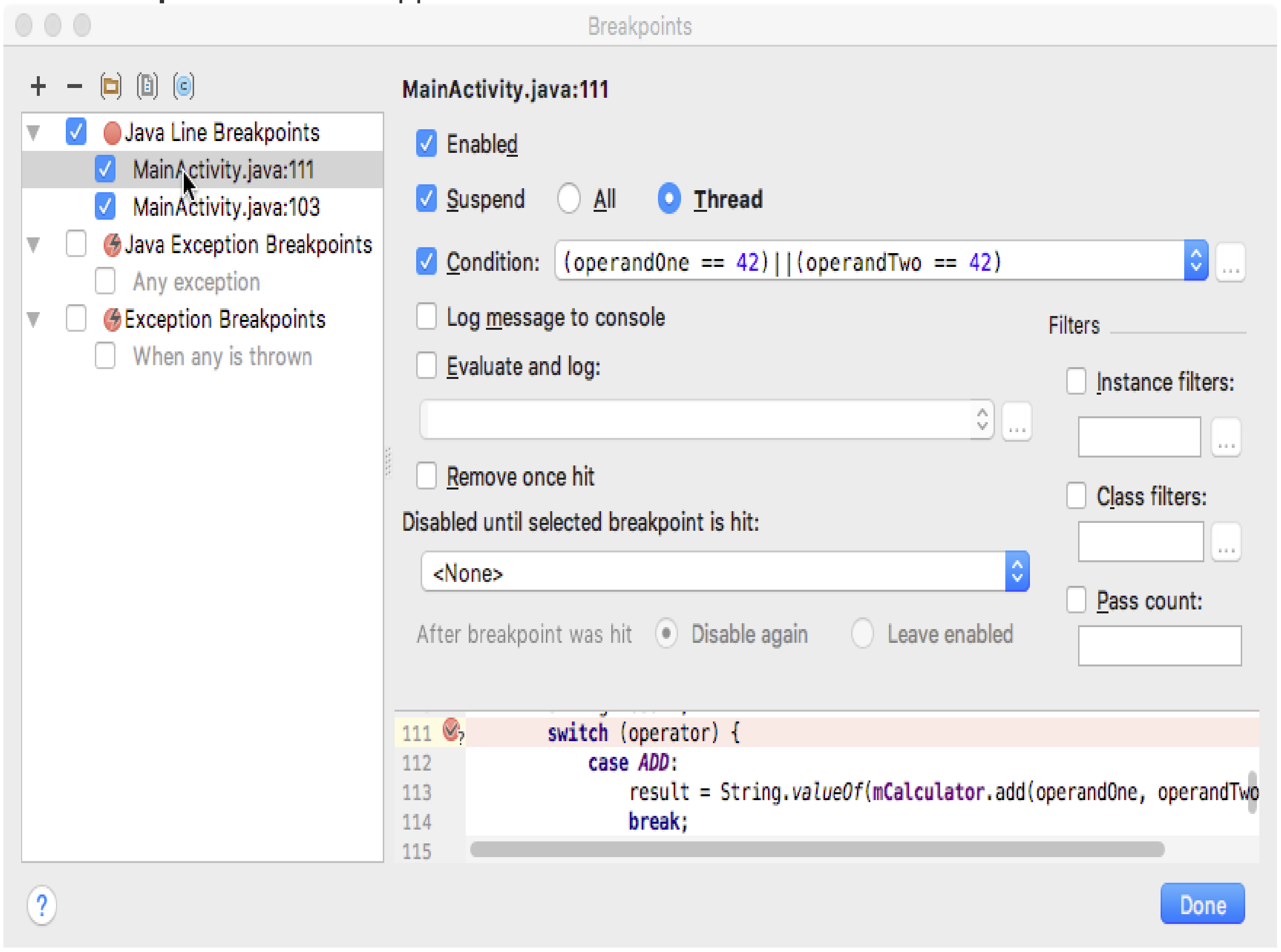
If your app is already running, you don't need to update it to add the breakpoint.

If you click a breakpoint by mistake, you can undo it by clicking the breakpoint. If you clicked a line of code that is not executable, the red dot includes an "**x**" and a warning appears that the line of code is not executable.

When your code execution reaches the breakpoint, Android Studio pauses execution of your app. You can then use the tools in the **Debug** pane to view the state of the app and debug that app as it runs.

View and configure breakpoints

To view all the breakpoints you've set and configure breakpoint settings, click the **View Breakpoints** icon on the left edge of the **Debug** pane.
The **Breakpoints** window appears.



MainActivity.java:111

- [✓] Enabled
- [✓] Suspend ( ) All (● Thread)
- [✓] Condition: `(operandOne == 42)||(operandTwo == 42)`
- [ ] Log message to console
- [ ] Evaluate and log:
- [ ] Remove once hit

Disabled until selected breakpoint is hit: <None>

After breakpoint was hit (● Disable again) ( ) Leave enabled

Filters
- [ ] Instance filters:
- [ ] Class filters:
- [ ] Pass count:

```
111        switch (operator) {
112            case ADD:
113                result = String.valueOf(mCalculator.add(operandOne, operandTwo
114                break;
115
```

In this window all the breakpoints you have set appear in the left pane, and you can enable or disable each breakpoint with the checkboxes. If a breakpoint is disabled, Android Studio does not pause your app when execution reaches that breakpoint.

Select a breakpoint from the list to configure its settings. You can configure a breakpoint to be disabled at first and have the system enable it after a different breakpoint is encountered. You can also configure whether a breakpoint should be disabled after it has been reached.

To set a breakpoint for any exception, select **Exception Breakpoints** in the list of breakpoints.

Disable (mute) all breakpoints

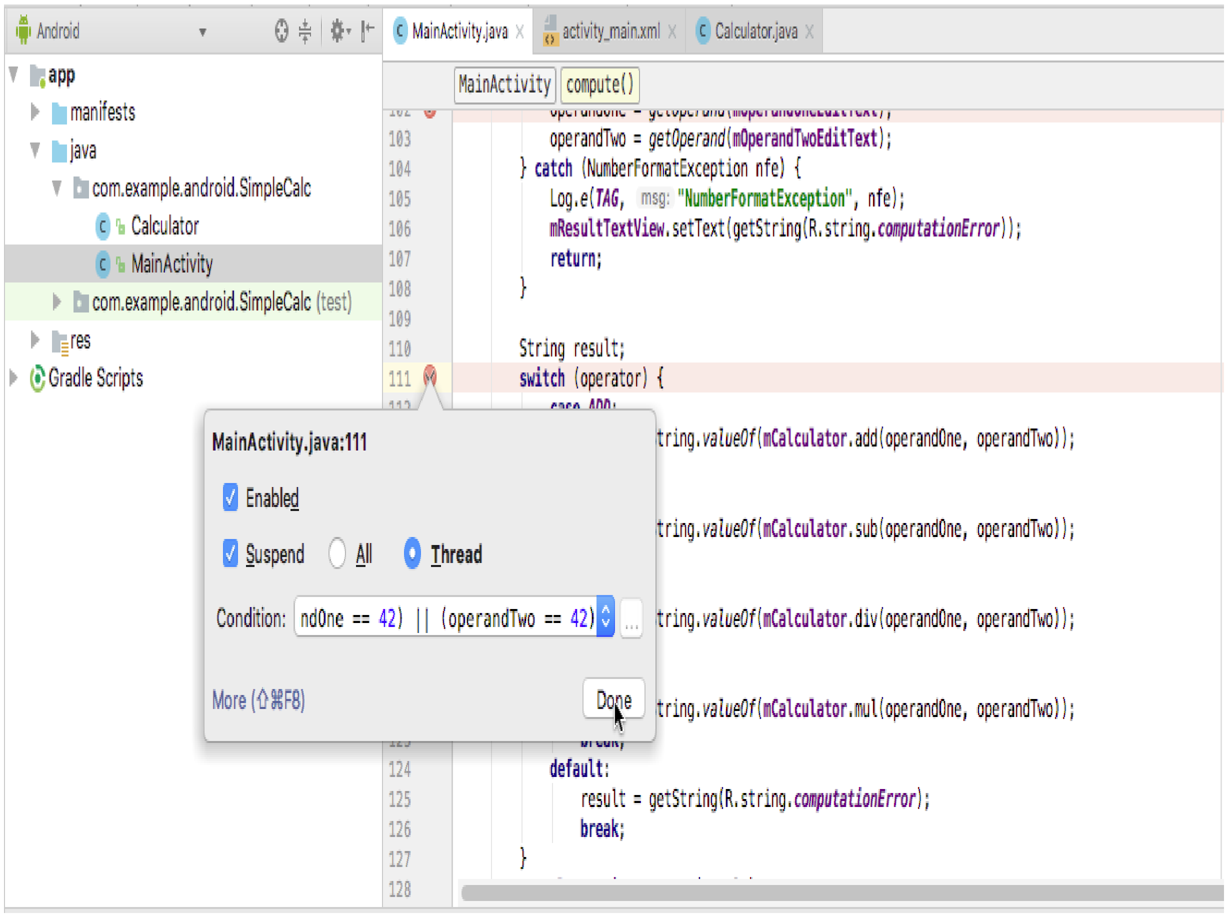
Disabling a breakpoint enables you to temporarily "mute" that breakpoint without removing it from your code. If you remove a breakpoint altogether you also lose any conditions or other features you created for that breakpoint, so disabling it can be a better choice.

To mute all breakpoints, click the **Mute Breakpoints** icon . Click the icon again to enable (unmute) all breakpoints.

Use conditional breakpoints

Conditional breakpoints are breakpoints that only stop execution of your app if the test in the condition is true. To define a test for a conditional breakpoint, use these steps:

1. **Right-click** (or **Control-click**) a breakpoint, and enter a test in the **Condition** field.



```
        operandTwo = getOperand(mOperandTwoEditText);
    } catch (NumberFormatException nfe) {
        Log.e(TAG, msg: "NumberFormatException", nfe);
        mResultTextView.setText(getString(R.string.computationError));
        return;
    }

    String result;
    switch (operator) {
            tring.valueOf(mCalculator.add(operandOne, operandTwo));
            tring.valueOf(mCalculator.sub(operandOne, operandTwo));
            tring.valueOf(mCalculator.div(operandOne, operandTwo));
            tring.valueOf(mCalculator.mul(operandOne, operandTwo));
        default:
            result = getString(R.string.computationError);
            break;
    }
```

MainActivity.java:111

☑ Enabled

☑ Suspend ○ All ● Thread

Condition: ndOne == 42) || (operandTwo == 42)

More (⇧⌘F8) Done

The test you enter in this field can be any Java expression as long as it returns a boolean value. You can use variable names from your app as part of the expression.

You can also use the **Breakpoints** window to enter a breakpoint condition.

2. Run your app in debug mode. Execution of your app stops at the conditional breakpoint, if the condition evaluates to true.

Stepping through code

After your app's execution has stopped because a breakpoint has been reached, you can execute your code from that point one line at a time with the **Step Over**, **Step Into**, and **Step Out** functions.

To use any of the step functions:

1. Begin debugging your app. Pause the execution of your app with a breakpoint.

   Your app's execution stops, and the **Debugger** pane shows the current state of the app. The current line is highlighted in your code.

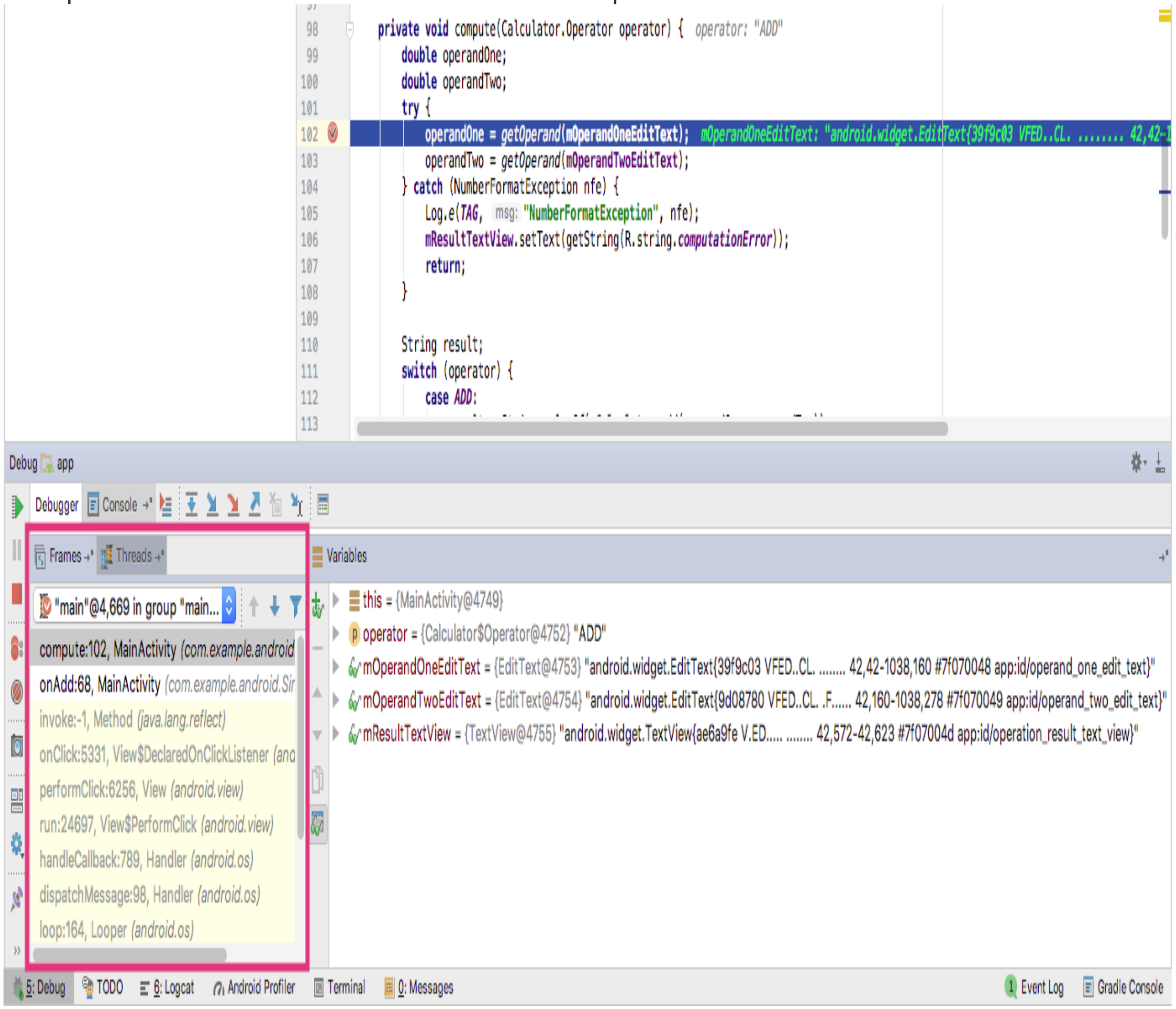
2. Click the **Step Over** icon , select **Run > Step Over,** or press **F8**. **Step Over** executes the next line of the code in the current class and method, executing all of the method calls on that line and remaining in the same file.

3. Click the **Step Into** icon , select **Run > Step Into,** or press **F7**. **Step Into** jumps into the execution of a method call on the current line (as compared to just executing that method and remaining on the same line). The **Frames** pane (which you'll learn about in the next section) updates to show the new stack frame (the new method). If the method call is contained in another class, the file for that class is opened and the current line in that file is highlighted. You can continue stepping over lines in this new method call, or step deeper into other methods.

4. Click the **Step Out** icon , select **Run > Step Out,** or press **Shift-F8**. **Step Out** finishes executing the current method and returns to the point where that method was called.
5. To resume normal execution of the app, select **Run > Resume Program** or click the Resume icon.

Viewing execution stack frames

The **Frames** pane of the **Debug** pane allows you to inspect the execution stack and the specific frame that caused the current breakpoint to be reached.



```
private void compute(Calculator.Operator operator) {  operator: "ADD"
    double operandOne;
    double operandTwo;
    try {
        operandOne = getOperand(mOperandOneEditText);  mOperandOneEditText: "android.widget.EditText{39f9c03 VFED..CL. ........ 42,42-1
        operandTwo = getOperand(mOperandTwoEditText);
    } catch (NumberFormatException nfe) {
        Log.e(TAG,  msg: "NumberFormatException", nfe);
        mResultTextView.setText(getString(R.string.computationError));
        return;
    }

    String result;
    switch (operator) {
        case ADD:
```

```
"main"@4,669 in group "main...
compute:102, MainActivity (com.example.android
onAdd:68, MainActivity (com.example.android.Sir
invoke:-1, Method (java.lang.reflect)
onClick:5331, View$DeclaredOnClickListener (anc
performClick:6256, View (android.view)
run:24697, View$PerformClick (android.view)
handleCallback:789, Handler (android.os)
dispatchMessage:98, Handler (android.os)
loop:164, Looper (android.os)
```

```
this = {MainActivity@4749}
operator = {Calculator$Operator@4752} "ADD"
mOperandOneEditText = {EditText@4753} "android.widget.EditText{39f9c03 VFED..CL. ........ 42,42-1038,160 #7f070048 app:id/operand_one_edit_text}"
mOperandTwoEditText = {EditText@4754} "android.widget.EditText{9d08780 VFED..CL. .F...... 42,160-1038,278 #7f070049 app:id/operand_two_edit_text}"
mResultTextView = {TextView@4755} "android.widget.TextView{ae6a9fe V.ED..... ........ 42,572-42,623 #7f07004d app:id/operation_result_text_view}"
```

The execution stack shows all the classes and methods (frames) that are being executed up to this point in the app, in reverse order (most recent frame first). As execution of a particular frame finishes, that frame is popped from the stack and execution returns to the next frame.

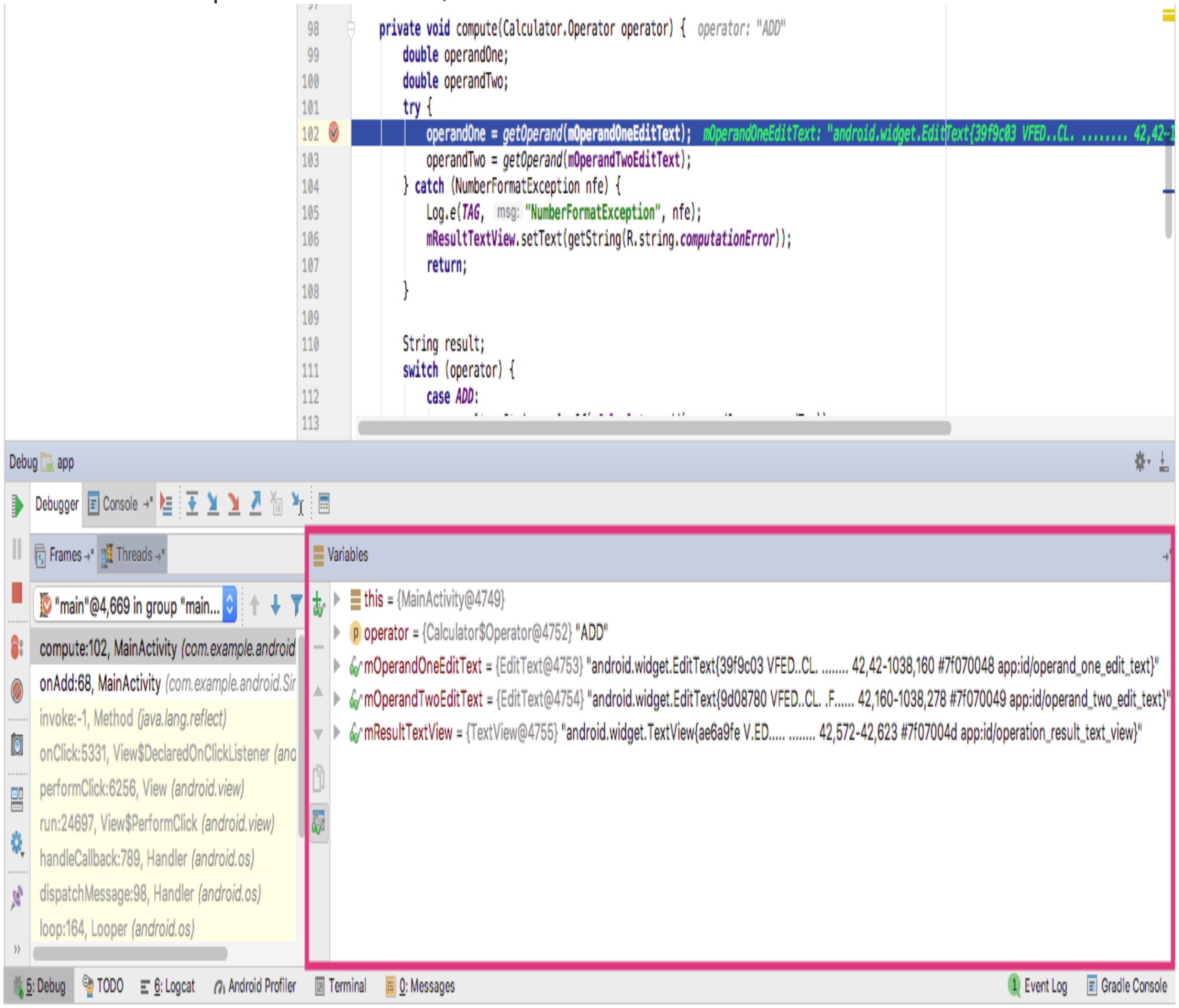
Clicking a line for a frame in the **Frames** pane opens the associated source in the editor and highlights the line where that frame was initially executed.
The **Variables** and **Watches** panes also update to reflect the state of the execution environment when that frame was last entered.

Inspecting and modifying variables

The **Variables** pane of the **Debugger** pane allows you to inspect the variables available at the current stack frame when the system stops your app on a breakpoint. Variables that hold objects or collections such as arrays can be expanded to view their components.

The **Variables** pane also allows you to evaluate expressions on the fly using static methods or variables available within the selected frame.

If the **Variables** pane is not visible, click the **Restore Variables View** icon .



```
private void compute(Calculator.Operator operator) {  operator: "ADD"
    double operandOne;
    double operandTwo;
    try {
        operandOne = getOperand(mOperandOneEditText);  mOperandOneEditText: "android.widget.EditText{39f9c03 VFED..CL. ........ 42,42-1
        operandTwo = getOperand(mOperandTwoEditText);
    } catch (NumberFormatException nfe) {
        Log.e(TAG, msg: "NumberFormatException", nfe);
        mResultTextView.setText(getString(R.string.computationError));
        return;
    }

    String result;
    switch (operator) {
        case ADD:
```

```
"main"@4,669 in group "main...
compute:102, MainActivity (com.example.android
onAdd:68, MainActivity (com.example.android.Sir
invoke:-1, Method (java.lang.reflect)
onClick:5331, View$DeclaredOnClickListener (and
performClick:6256, View (android.view)
run:24697, View$PerformClick (android.view)
handleCallback:789, Handler (android.os)
dispatchMessage:98, Handler (android.os)
loop:164, Looper (android.os)
```

```
Variables
this = {MainActivity@4749}
operator = {Calculator$Operator@4752} "ADD"
mOperandOneEditText = {EditText@4753} "android.widget.EditText{39f9c03 VFED..CL. ........ 42,42-1038,160 #7f070048 app:id/operand_one_edit_text}"
mOperandTwoEditText = {EditText@4754} "android.widget.EditText{9d08780 VFED..CL. .F...... 42,160-1038,278 #7f070049 app:id/operand_two_edit_text}"
mResultTextView = {TextView@4755} "android.widget.TextView{ae6a9fe V.ED..... ........ 42,572-42,623 #7f07004d app:id/operation_result_text_view}"
```

To modify variables in your app as it runs:

1. **Right-click** (or **Control-click**) any variable in the **Variables** pane, and select **Set Value**. You can also press **F2**.
2. Enter a new value for the variable, and press **Return**.

The value you enter must be of the appropriate type for that variable, or Android Studio returns a "type mismatch" error.

Setting watches

The **Watches** pane provides similar functionality to the **Variables** pane except that expressions added to the **Watches** pane persist between debugging sessions. Add watches for variables and fields that you access frequently or that provide state that is helpful for the current debugging session.

To use watches:

1. Begin debugging your app.
2. Click the **Show Watches** icon . The **Watches** pane appears next to the **Variables** pane.
3. In the **Watches** pane, click the plus (**+**) button. In the text box that appears, type the name of the variable or expression you want to watch and then press **Enter**.

Remove an item from the Watches list by selecting the item and then clicking the minus (**–**) button.

Change the order of the elements in the **Watches** pane list by selecting an item and then clicking the up or down icons.

Evaluating expressions

Use **Evaluate Expression** to explore the state of variables and objects in your app, including calling methods on those objects. To evaluate an expression:

1. Click the **Evaluate Expression** icon , or select **Run > Evaluate Expression**.

The **Evaluate Code Fragment** window appears. You can also right-click on any variable and choose **Evaluate Expression**.

2. Enter any Java expression into the top field of the **Evaluate Code Fragment** window, and click **Evaluate**.

The Result field shows the result of that expression. Note that the result you get from evaluating an expression is based on the app's current state. Depending on the values of the variables in your app at the time you evaluate expressions, you may get different results. Changing the values of variables in your expressions also changes the current running state of the app.

More tools for debugging

Android Studio and the Android SDK include a number of other tools to help you find and correct issues in your code.

System log (**Logcat** pane)

As you've learned in another chapter, you can use the `Log` class to send messages to the Android system log, and view those messages in Android Studio in the **Logcat** pane.
To write log messages in your code, use the `Log` class. `Log` messages help you understand the execution flow by collecting the system debug output while you interact with your app. `Log` messages can tell you what part of your app failed. For more information about logging, see Reading and Writing Logs.

Tracing and logging

Analyzing traces allows you to see how much time is spent in certain methods, and which ones are taking the longest times.

To create the trace files, include the `Debug` class and call one of the `startMethodTracing()` methods. In the call, you specify a base name for the trace files that the system generates.
To stop tracing, call `stopMethodTracing()`. These methods start and stop method tracing across the entire virtual machine. For example, you could call `startMethodTracing()` in the `onCreate()` method of your `Activity`, and call `stopMethodTracing()` in the onDestroy() method of that `Activity`.

Other tools

- The Android Debug Bridge (ADB) is a command-line tool that lets you communicate with an emulator instance or connected Android-powered device.
- The Android Profiler provides real-time data for your app's CPU, memory, and network activity. You can perform sample-based method tracing to time your code execution, capture heap dumps, view memory allocations, and inspect the details of network-transmitted files.
- The CPU Profiler helps you inspect your app's CPU usage and thread activity in real-time, and record method traces, so you can optimize and debug your app's code.
- The Network Profiler displays real-time network activity on a timeline, showing data sent and received, as well as the current number of connections. This lets you examine how and when your app transfers data, and optimize the underlying code appropriately.

Trace logging and the AndroidManifest.xml file

There are multiple types of debugging available to you beyond setting breakpoints and stepping through code. You can also use logging and tracing to find issues with your code. When you have a trace log file (generated by adding tracing code to your app), you can load the log files in Traceview, which displays the log data in two panes:

- A timeline pane describes when each thread and method started and stopped.
- A profile pane provides a summary of what happened inside a method.

Likewise, to make the app debuggable even when the app is running on a device in user mode, you can set `android:debuggable` in the `<application>` tag of the `AndroidManifest.xml` file to `"true"`. By default, the `debuggable` value is set to `"false"`.
You can create and configure build types in the module-level `build.gradle` file inside the `android {}` block. When you create a new module, Android Studio creates the debug and release build types for you. Although the debug build type doesn't appear in the build configuration file, Android Studio configures it with debuggable true. This allows you to debug the app on secure Android-powered devices and configures APK signing with a generic debug keystore. If you want to add or change certain settings, add the debug build type to your configuration.
When you prepare your app for release, you must remove all the extra code in your source files that you wrote for testing purposes.

In addition to prepping the code itself, there are a few other tasks you need to complete in order to get your app ready to publish. These include:

- Removing logging statements.
- Remove any calls to show `Toast` messages.
- Disable debugging in the `AndroidManifest.xml` file by either removing `android:debuggable` attribute from the `<application>` tag, or setting `android:debuggable` to `"false"`.
- Remove all debug tracing calls from your source code files such as `startMethodTracing()` and `stopMethodTracing()`.

All these changes made for debugging must be removed from your code before release because they can impact the execution and performance production code.

Related practical

The related practical is 3.1: The debugger.

Learn more

Android Studio documentation:

- Android Studio User Guide
- Debug Your App
- Write and View Logs
- Analyze a Stack Trace
- Android Debug Bridge
- Android Profiler
- Network Profiler
- CPU Profiler
- Traceview
- Create and Edit Run/Debug Configurations

Other:

- Video: Debugging and Testing in Android Studio

3.2: App testing

Contents:

In this chapter you get an overview of Android testing, and you learn about creating and running local unit tests in Android Studio with JUnit.

About testing

Even though you have an app that compiles and runs and looks the way you want it to on different devices, you must make sure that your app will behave the way you expect it to in every situation, especially as your app grows and changes. Even if you try to manually test your app every time you make a change—a tedious prospect at best—you might miss something or not anticipate what end users might do with your app to cause it to fail.

Writing and running tests is a critical part of the software development process. Test-driven development (TDD) is a popular software development philosophy that places tests at the core of all software development for an app or service.This does not negate the need for further testing, it merely gives you a solid baseline to work with.

Testing your code can help you catch issues early in development—when they are the least expensive to address—and improve the robustness of your code as your app gets larger and more complex. With tests in your code, you can exercise small portions of your app in isolation, and in an automatable and repeatable manner for more efficient testing.

The code you write to test your app doesn't end up in the production version of your app; it lives only on your development machine, alongside your app's code in Android Studio.

Types of tests

Android supports several different kinds of tests and testing frameworks. Two basic forms of testing Android Studio supports are local unit tests and instrumented tests.

Local unit tests are tests that are compiled and run entirely on your local machine with the Java Virtual Machine (JVM). Use local unit tests to test the parts of your app (such as the internal logic) that do not need access to the Android framework or an Android-powered device or emulator, or those for which you can create fake ("mock" or stub) objects that pretend to behave like the framework equivalents.

Instrumented tests are tests that run on an Android-powered device or emulator. These tests have access to the Android framework and to `Instrumentation` information such as the app's `Context`. You can use instrumented tests for unit testing, user interface (UI) testing, or integration testing, making sure that the components of your app interact correctly with other apps. Most commonly, you use instrumented tests for UI testing, which allows you to test that your app behaves correctly when a user interacts with your app or enters a specific input. For most forms of user interface testing, you use the Espresso framework, which allows you to write automated UI tests. You'll learn about instrumented tests and Espresso in another chapter.

Unit Testing

Unit tests should be the fundamental tests in your app testing strategy. By creating and running unit tests against your code, you can verify that the logic of individual functional code areas or units is correct. Running unit tests after every build helps you catch and fix problems 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. Create unit tests when you need to verify the logic of specific code in your app. For example, if you unit test a class, your test might check that the class is in the right state. For a method, you might test its behavior for different values of its parameters, especially `null`.
Typically, you test the unit of code in isolation, and your test monitors changes only to that unit. You can use a mocking framework such as Mockito to isolate your unit from its dependencies.You can also write your unit tests for Android in JUnit 4, a common unit testing framework for Java code.

The Android Testing Support Library

The Android Testing Support Library provides the infrastructure and APIs for testing Android apps, including support for JUnit 4. With the testing support library you can build and run test code for your apps.

You may already have the Android Testing Support Library installed with Android Studio. To check for the Android Support Repository, follow these steps:

1. In Android Studio choose **Tools > Android > SDK Manager**.
2. Click the **SDK Tools** tab, and look for the Support Repository.
3. If necessary, update or install the library.

The Android Testing Support Library classes are located under the `android.support.test` package. There are also older testing APIs in `android.test`. You should use the support libraries first, when given a choice between the support libraries and the older APIs, as the support libraries help build and distribute tests in a cleaner and more reliable fashion than directly coding against the API itself.

Setting up testing

To prepare your project for testing in Android Studio, you need to:

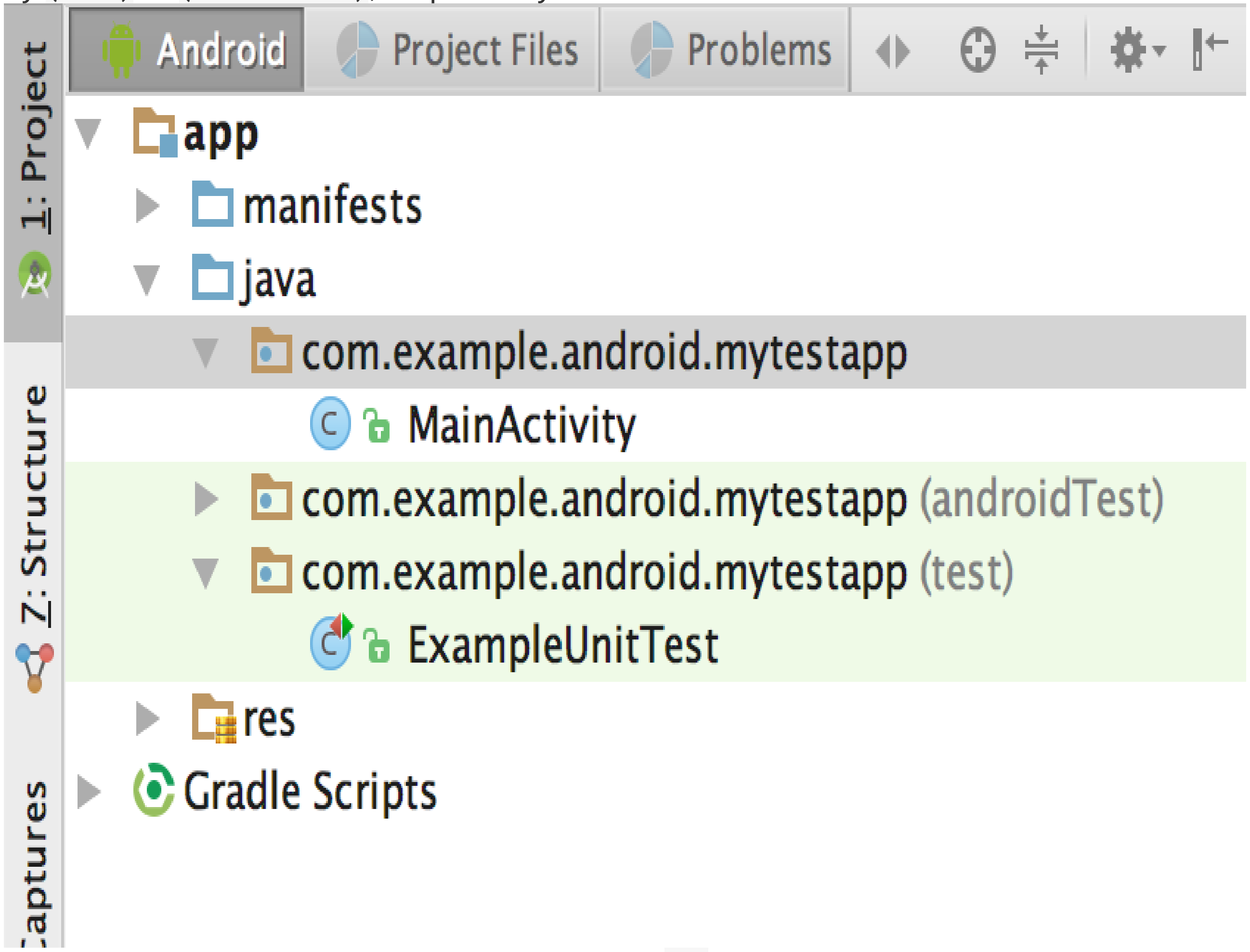
- Organize your tests in a *source set*.
- Configure your project's Gradle dependencies to include testing-related APIs.

Android Studio source sets

Source sets are collections of code in your project that are for different build targets or other "flavors" of your app. When Android Studio creates your project, it creates three source sets for you:

- The *main* source set, for your app's code and resources.
- The `(test)` source set, for your app's local unit tests. The source set shows `(test)` after the package name.
- The `(androidTest)` source set, for Android instrumented tests. The source set shows `(androidTest)` after the package name.

Source sets appear in the Android Studio **Project > Android** pane under the package name for your app. The main source set includes just the package name. The `test` and `androidTest` source sets have the package name followed by `(test)` or `(androidTest)`, respectively.



These source sets correspond to folders in the `src` directory for your project. For example, the files for the test source set are located in `src/test/java`.

Configuring Gradle for test dependencies

To use the unit testing APIs, you may need to configure the dependencies for your project. The default Gradle build file, provided by `Activity` templates such as the Empty Activity template, includes some of these dependencies, but you may need to add more dependencies for additional testing features such as matching or mocking frameworks.

In your app project's `build.gradle (Module: app)` file, the following dependency should already be included (if not, you should add it):

```
testImplementation 'junit:junit:4.12'
```

The `androidTestImplementation` dependencies are required for UI testing with Espresso, described in another chapter:

```
androidTestImplementation 'com.android.support.test:runner:1.0.1'
androidTestImplementation
                    'com.android.support.test.espresso:espresso-core:3.0.1'
```

Note that the version numbers for these libraries may have changed. If Android Studio reports a newer library, update the number to reflect the current version.

You may also want to add dependencies for the optional Hamcrest matchers and the Mokito framework:

```
testCompile 'org.hamcrest:hamcrest-library:1.3'
testCompile 'org.mockito:mockito-core:1.10.19'
```

After editing the `build.gradle (Module: app)` file, you must sync your project to continue. Click **Sync Now** in Android Studio when prompted.

Configuring a test runner

A test runner is a library or set of tools that enables testing to occur and the results to be printed to a log. Your Android project has access to a basic JUnit test runner as part of the JUnit4 APIs. The Android test support library includes a test runner for instrumented and Espresso tests, `AndroidJUnitRunner`, which also supports JUnit 3 and 4.
This chapter demonstrates the default runner for unit tests, which is supplied by `Activity` templates such as the Empty Activity template. In your app project's `build.gradle (Module: app)` file, the following test runner should already be included in the `defaultConfig` section (if not, you should add it):

```
testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
```

Creating and running unit tests

Create your unit tests as a generic Java file using the JUnit 4 APIs, and store those tests in the `(test)` source set. Each Android Studio project template includes this source set and a sample Java test file called `ExampleUnitTest`.

Creating a new test class

To create a new test class file, add a Java file to the `(test)` source set for your project. Test class files for unit testing are typically named for the class in your app that you are testing, with "Test" appended. For example, if you have a class called `Calculator` in your app, the class for your unit tests would be `CalculatorTest`.
To add a new test class file, follow these steps:

1. Expand the **java** folder and the folder for your app's test source set. The existing unit test class files are shown.
2. **Right-click** (or **Control-click**) on the test source set folder and select **New > Java Class.**
3. Name the file and click **OK**.

Writing your tests

Use JUnit 4 syntax and annotations to write your tests. For example, the test class shown below includes the following annotations:

- The `@RunWith` annotation indicates the test runner that should be used for the tests in this class.
- The `@SmallTest` annotation indicates that this is a small (and fast) test.
- The `@Before` annotation marks a method as being the setup for the test.
- The `@Test` annotation marks a method as an actual test.

For more information on JUnit annotations, see JUnit 4 API Reference.

```
@RunWith(JUnit4.class)
@SmallTest
public class CalculatorTest {
   private Calculator mCalculator;
   // Set up the environment for testing
   @Before
   public void setUp() {
      mCalculator = new Calculator();
   }

   // test for simple addition
   @Test
   public void addTwoNumbers() {
       double resultAdd = mCalculator.add(1d, 1d);
       assertThat(resultAdd, is(equalTo(2d)));
   }
}
```

The `addTwoNumbers()` method is the actual test. The key part of a unit test is the assertion, which is defined here by the `assertThat()` method. Assertions are expressions that must evaluate and result in a value of true for the test to pass. For more information on assertions, see the JUnit reference documentation for the `Assert` class.

JUnit 4 provides a number of assertion methods, but `assertThat()` is the most flexible, as it allows for general-purpose comparison methods called *matchers*. The Hamcrest framework is commonly used for matchers ("Hamcrest" is an anagram for matchers). Hamcrest includes a large number of comparison methods as well as enabling you to write your own. For more information on the Hamcrest framework, see the Java Hamcrest homepage.

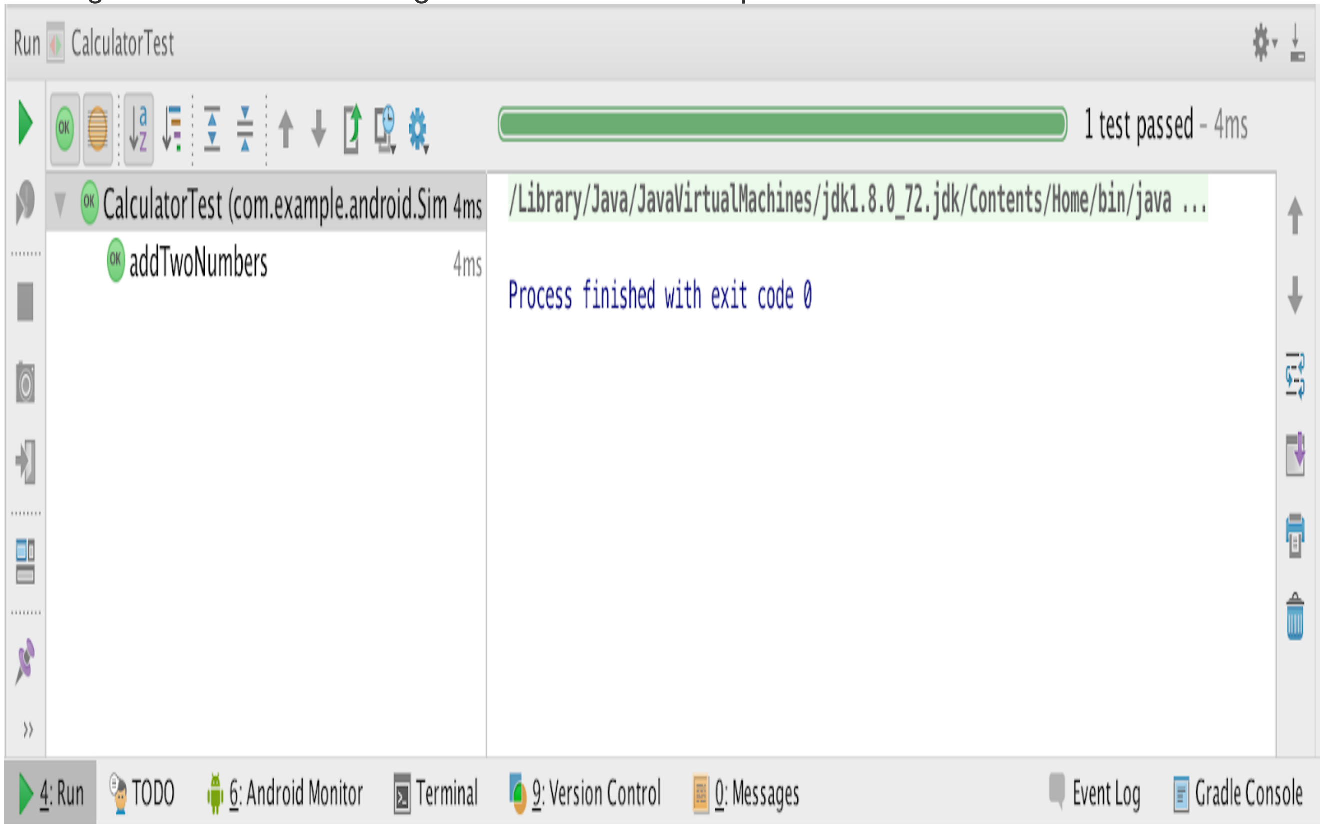
Note that the `addTwoNumbers()` method in this example includes only one assertion. The general rule for unit tests is to provide a separate test method for every individual assertion. Grouping more than one assertion into a single method can make your tests harder to debug if only one assertion fails, and obscures the tests that do succeed.

Running your tests

To run your local unit tests, follow these steps:

- To run a single test, **right-click** (or **Control-click**) that test method and select **Run**.
- To test all the methods in a test class, **right-click** (or **Control-click**) the test file in the **Android > Project** pane, and select **Run**.
- To run all tests in a directory, **right-click** (or **Control-click**) on the directory and select **Run tests**.

The project builds, if necessary, and the testing view appears at the bottom of the screen. If all the tests you ran are successful, the progress bar at the top of the view turns green. A status message in the footer also reports "Tests Passed."



```
/Library/Java/JavaVirtualMachines/jdk1.8.0_72.jdk/Contents/Home/bin/java ...

Process finished with exit code 0
```

Related practical

The related practical is 3.2: Unit tests.

Learn more

Android Studio documentation:

- Android Studio User Guide
- Write and View Logs

Android developer documentation:

- Best Practices for Testing
- Getting Started with Testing
- Building Local Unit Tests

Other:

- JUnit 4 Home Page
- JUnit 4 API Reference
- `java.lang.Math`
- Java Hamcrest
- Mockito Home Page
- Video: Android Testing Support - Testing Patterns
- Android Testing Codelab
- Android Tools Protip: Test Size Annotations
- The Benefits of Using assertThat over other Assert Methods in Unit Tests

3.3: The Android Support Library

Contents:

In this chapter you explore the Android Support Library, which is part of the Android SDK tools. You can use the Android Support Library to get backward-compatible versions of new Android features, plus user interface (UI) elements not included in the standard Android framework.

About the Android Support Library

The Android SDK tools include a number libraries collectively called the *Android Support Library*. This package of libraries provides several features that are not built into the standard Android framework, and provides backward compatibility for older devices. Include any of these libraries in your app to incorporate that library's functionality.

Note: The Android Support library is a different package from the Android **Testing** Support library you learned about in another chapter. The testing support library provides tools and APIs just for testing, whereas the more general support library provides features of all kinds (but no testing).

Features

The features of the Android Support Library include:

- Backward-compatible versions of framework components. These compatibility libraries allow you to use features and components available on newer versions of the Android platform even when your app is running on an older platform version. For example, older devices may not have access to newer features such as fragments, app bars (also known as action bars), or Material Design elements. The support library provides access to those features on older devices.
- Additional layout and UI elements. The support library includes views and layouts that can be useful for your app, but are not included in the standard Android framework. For example, the `RecyclerView` view that you will use in another chapter is part of the support library.
- Support for different device form factors, such as TV or wearables: For example, the Leanback library includes components specific to app development on TV devices.
- Design support: The design support library includes components to support Material Design elements in your app, including floating action buttons (FAB). You'll learn more about Material Design in a later chapter.
- Various other features such as palette support, annotations, percentage-based layout dimensions, and preferences.

Backward Compatibility

Support libraries allow apps running on older versions of the Android platform to support features made available on newer versions of the platform. For example, an app running on a version of Android lower than 5.0 (API level 21) that relies on framework classes cannot display Material Design elements, as that version of the Android framework doesn't support Material Design. However, if the app incorporates the v7 appcompat library, that app has access to many of the features available in API level 21, including support for Material Design. As a result, your app can deliver a more consistent experience across a broader range of platform versions.

The support library APIs also provide a compatibility layer between different versions of the framework APIs. This compatibility layer transparently intercepts API calls and changes either the arguments passed, handles the operation itself, or redirects the operation. In the case of the support libraries, by using the compatibility layer's methods, you can ensure interoperability between older and newer Android releases. Each new release of Android adds new classes and methods, and possibly deprecates some older classes and methods. The support libraries include compatibility classes that can be used for backward compatibility.You can identify these classes by their names, which include "Compat" (such as `ActivityCompat`).

When an app calls one of the support class's methods, the behavior of that method depends on the underlying Android version. If the device includes the necessary framework functionality, the support library uses the framework. If the device is running an older version of Android, the support library makes an attempt to implement similar compatible behavior with the APIs it has available.

For most cases you do not need to write complex code that checks the version of Android and performs different operations based on that version. You can rely on the support library to do those checks and choose appropriate behavior.

When in doubt, choose a support library compatibility class over the framework class.

Versions

Each package in the support library has a version number in three parts (*x.y.z*) that corresponds to an Android API level, and to a particular revision of that library. For example, a support library version number of *22.3.4* is version 3.4 of the support library for API 22.

As a general rule, use the most recent version of the support library for the API your app is compiled and targeted for, or a newer version. For example, if your app targets API 26, use the version 26.*x.x* of the support library.

You can always use a newer support library than the one for your targeted API. For example, if your app targets API 22 you can use version 25 or higher of the support library. The reverse is not true—you cannot use an older support library with a newer API. As a general rule, you should try to use the most up-to-date API and support libraries in your app.

API Levels

In addition to the actual version number, the name of the support library itself indicates the API level that the library is backward-compatible with. You cannot use a support library in your app for an API higher than the minimum API your app supports. For example, if the minimum API your app supports is 10, you cannot use the v13 support library or v14 preferences support library in your app. If your app uses multiple support libraries, your minimum API must be higher than the largest number—that is, if you include support libraries for v7, v13, and v14 your minimum API must be at least 14.

All of the support libraries, including the v4 and v7 libraries, require a minimum SDK of API 9.

Support libraries and features

This section describes the important features provided by the libraries in the Android Support Library. You'll learn about many of the features described in this section in other chapters.

v4 support library

The v4 support libraries include the largest set of APIs compared to the other libraries, including support for app components, user interface features, accessibility, data handling, network connectivity, and programming utilities.

The v4 support libraries include these specific components:

- v4 compat library: Compatibility wrappers (classes that include the word "Compat") for a number of core framework APIs.
- v4 core-utils library: Provides a number of utility classes
- v4 core-ui library: Implements a variety of UI-related components.
- v4 media-compat library: Backports portions of the media framework from API 21.
- v4 fragment library: Adds support for Android fragments.

v7 support library

The v7 support library includes both compatibility libraries and additional features.

The v7 support library includes all the v4 support libraries, so you don't have to add those separately. A dependency on the v7 support library is included in every new Android Studio project, and new activities in your project extend from `AppCompatActivity`.
The v7 support libraries include these specific components:

- v7 appcompat library: Adds support for the app bar UI design pattern and support for Material Design UI implementations.
- v7 cardview library: Provides the `CardView` class, a view that lets you show information inside cards.
- v7 gridlayout library: Includes the `GridLayout` class, which allows you to arrange UI elements using a grid of rectangular cells
- v7 mediarouter library: Provides `MediaRouter` and related media classes that support Google Cast.
- v7 palette library: Implements the `Palette` class, which lets you extract prominent colors from an image.
- v7 recyclerview library: Provides the `RecyclerView` class, a view for efficiently displaying large data sets by providing a limited window of data items.
- v7 preference library: Provides APIs to support preference objects in app settings.

Other libraries

- v8 renderscript library: Adds support for the RenderScript, a framework for running computationally intensive tasks at high performance.
- v13 support library: Provides support for using a `Fragment` with the `FragmentCompat` class and additional `Fragment` support classes.
- v14 preference support library, and v17 preference support library for TV: provides APIs to add support for preference interfaces on mobile devices and TV.
- v17 leanback library: Provides APIs to support building UIs on TV devices.
- Annotations support library: Contains APIs to support adding annotation metadata to your apps.
- Design support library: Adds support for various Material Design components and patterns such as navigation drawers, floating action buttons (FAB), snackbars, and tabs.
- Custom Tabs support library: Adds support for adding and managing custom tabs in your apps.
- Percent support library: Enables you to add and manage percentage based dimensions in your app.
- App recommendation support library for TV: Provides APIs to support adding content recommendations in your app running on TV devices.

Setting up and using the Android Support Library

The Android Support Library package is part of the Android SDK, and available to download in the Android SDK manager. To set up your project to use any of the support libraries, use these steps:

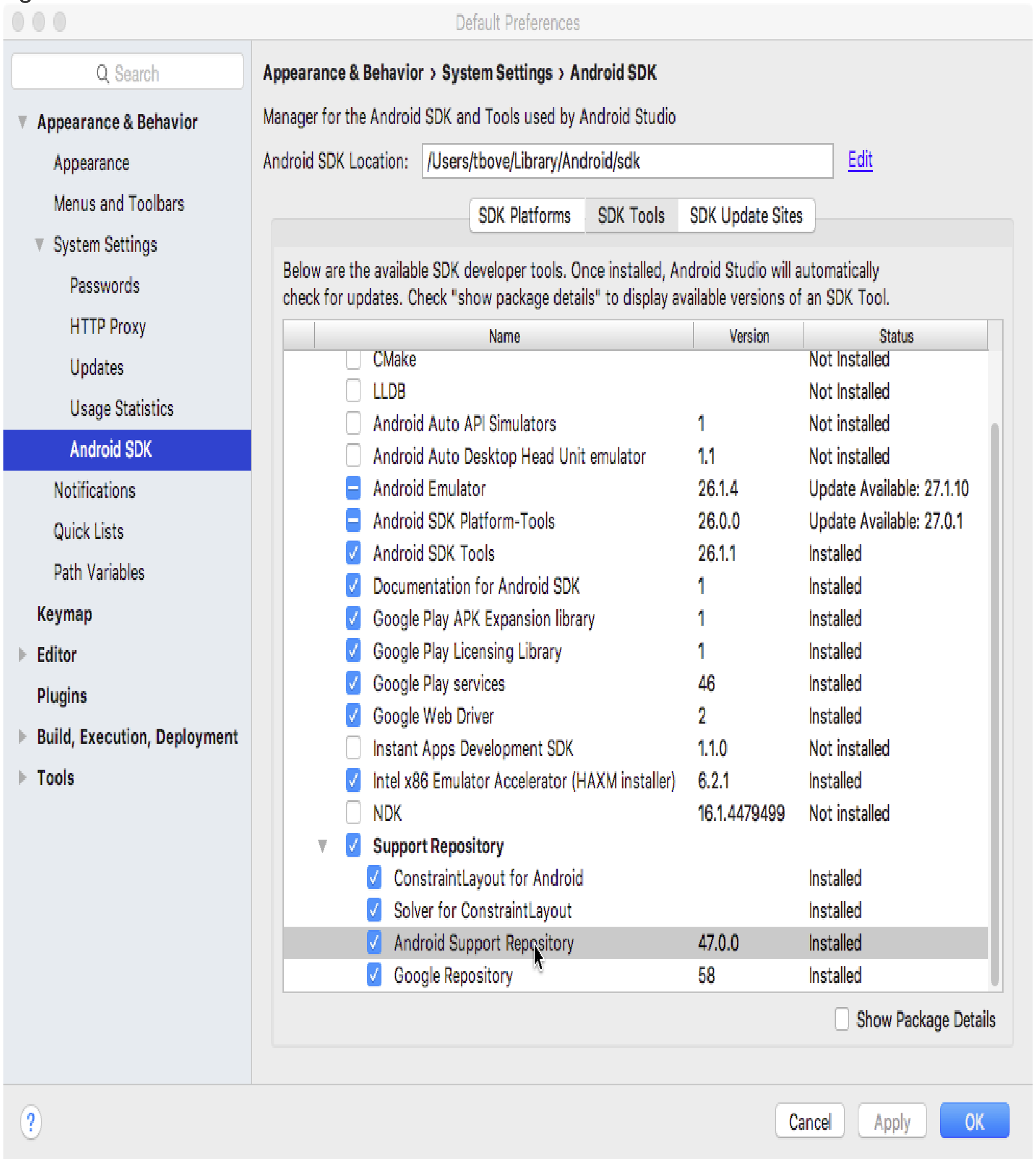
1. Download the support library with the Android SDK manager, or verify that the support libraries are already available.
2. Find the library dependency statement for the support library you're interested in.
3. Add that dependency statement to the `dependencies` section of your `build.gradle (Module: app)` file.

Download the support library

In Android Studio, you'll use the Android Support Repository—the repository in the SDK manager for all support libraries—to get access to the library from within your project.

You may already have the Android support libraries downloaded and installed with Android Studio. To verify that you have the support libraries available, follow these steps:

1. In Android Studio, select **Tools > Android > SDK Manager**, or click the SDK Manager icon. The SDK Manager preference pane appears.
2. Click the **SDK Tools** tab and expand **Support Repository**, as shown in the figure below.



Default Preferences

Appearance & Behavior › System Settings › Android SDK

Manager for the Android SDK and Tools used by Android Studio

Android SDK Location: /Users/tbove/Library/Android/sdk Edit

SDK Platforms | SDK Tools | SDK Update Sites

Below are the available SDK developer tools. Once installed, Android Studio will automatically check for updates. Check "show package details" to display available versions of an SDK Tool.

	Name	Version	Status
☐	CMake		Not Installed
☐	LLDB		Not Installed
☐	Android Auto API Simulators	1	Not installed
☐	Android Auto Desktop Head Unit emulator	1.1	Not installed
–	Android Emulator	26.1.4	Update Available: 27.1.10
–	Android SDK Platform-Tools	26.0.0	Update Available: 27.0.1
✓	Android SDK Tools	26.1.1	Installed
✓	Documentation for Android SDK	1	Installed
✓	Google Play APK Expansion library	1	Installed
✓	Google Play Licensing Library	1	Installed
✓	Google Play services	46	Installed
✓	Google Web Driver	2	Installed
☐	Instant Apps Development SDK	1.1.0	Not installed
✓	Intel x86 Emulator Accelerator (HAXM installer)	6.2.1	Installed
☐	NDK	16.1.4479499	Not installed
✓	**Support Repository**		
✓	ConstraintLayout for Android		Installed
✓	Solver for ConstraintLayout		Installed
✓	Android Support Repository	47.0.0	Installed
✓	Google Repository	58	Installed

☐ Show Package Details

3. Look for **Android Support Repository** in the list. If **Installed** appears in the Status column, you're all set. Click **Cancel**.

   If **Not installed** or **Update Available** appears, click the checkbox next to **Android Support Repository**. A download icon should appear next to the checkbox. Click **OK.**

4. Click **OK** again, and then **Finish** when the support repository has been installed.

Find a library dependency statement

To provide access to a support library from your project, you add that library to your Gradle build file as a dependency. Dependency statements have a specific format that includes the name and version number of the library.

1. Visit the Support Library Features page on developer.android.com.
2. Find the library you're interested in on that page, for example, the Design Support Library for Material Design support.
3. Copy the dependency statement shown at the end of the section. For example, the dependency for the design support library looks like this:

```
com.android.support:design:26.1.0
```

The version number at the end of the line may vary from the one shown above. You will update the version number when you add the dependency to the build.gradle file in the next step.

Add the dependency to your build.gradle file

The Gradle scripts for your project manage how your app is built, including specifying the dependencies your app has on other libraries. To add a support library to your project, modify your Gradle build files to include the dependency to that library you found in the previous section.

1. In Android Studio, make sure the **Project > Android** pane is open.
2. Expand **Gradle Scripts** and open the **build.gradle (Module: app)** file. Note that `build.gradle` for the overall project (`build.gradle (Project:` *app_name* `)`) is a different file from the `build.gradle` for the app module.
3. Locate the `dependencies` section near the end of the file. The `dependencies` section for a new project already includes dependencies for several other libraries.
4. Add a dependency for the support library that includes the statement you copied in the previous task. For example, a complete dependency on the design support library looks like this:

```
compile 'com.android.support:design:26.1.0'
```

1. Update the version number, if necessary.

If the version number you specified is lower than the currently available library version number, Android Studio warns you that an updated version is available. ("a newer version of `com.android.support:design` is available"). Edit the version number to the updated version, or enter **Alt-Enter** (**Option-Return** on a Mac) and select **Change to *xx.xx.x*** from the menu, where *xx.xx.x* is the most up-to-date version available.

2. Click **Sync Now** to sync your updated gradle files with the project, if prompted.

Using the support library APIs

All the support library classes are contained in the android.support packages. For example, `android.support.v7.app.AppCompatActivity` is the fully qualified name for the `AppCompatActivity` class, from which all of your activities extend.
Support Library classes that provide support for existing framework APIs typically have the same name as framework class but are located in the android.support class packages. Make sure that when you import those classes you use the right package name for the class you're interested in. For example, when applying the `ActionBar` class, use one of:

- `android.support.v7.app.ActionBar` when using the Support Library.
- `android.app.ActionBar` when developing only for API level 11 or higher.

The support library also includes several `View` classes used in XML layout files. In the case of the `View` elements (such as `CoordinatorLayout`), you must always use the fully qualified name in the XML element for that `View`:

```
<android.support.design.widget.CoordinatorLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">
</android.support.design.widget.CoordinatorLayout>
```

You'll learn about `CoordinatorLayout` in another chapter.

Checking system versions

Although the support library can help you implement single apps that work across Android platform versions, there may be times when you need to check for the version of Android your app is running on, and provide the correct code for that version.

Android provides a unique code for each platform version in the `Build` constants class. Use these codes within your app to test for the version and to ensure that the code that depends on higher API levels is executed only when those APIs are available on the system.

```
private void setUpActionBar() {
    // Make sure we're running on Honeycomb or higher to use ActionBar APIs
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB) {
        ActionBar actionBar = getActionBar();
        actionBar.setDisplayHomeAsUpEnabled(true);
    } else { // do something else }
}
```

Related practical

The related practical is 3.3: Support libraries.

Learn more

Android Studio documentation:

- Android Studio User Guide

Android developer documentation:

- Android Support Library (introduction)
- Support Library Setup
- Support Library Features
- Supporting Different Platform Versions
- Package Index (all API packages that start with android.support)

Other:

- Picking your compileSdkVersion, minSdkVersion, and targetSdkVersion
- Understanding the Android Support Library
- All the Things Compat