

## 3.1: The Android Studio Debugger

### Contents:

- About debugging
- Running the debugger
- Using Breakpoints
- Stepping through code
- Viewing execution stack frames
- Inspecting and modifying variables
- Setting watches
- Evaluating expressions
- More tools for debugging
- Trace logging and the Android manifest

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

### About debugging

Finding and repairing mistakes (bugs) or unexpected behavior in your code is the process of debugging. All software contains defects, which can range from improper behavior in your app to excessive memory or network resource usage to actual program stalling or crashing.

Bugs can appear for a variety of causes:

- 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 Android monitor (logcat)
- The Android Studio debugger
- Testing frameworks such as JUnit or Espresso
- Dalvik Debug Monitor Server (DDMS), to track resource usage

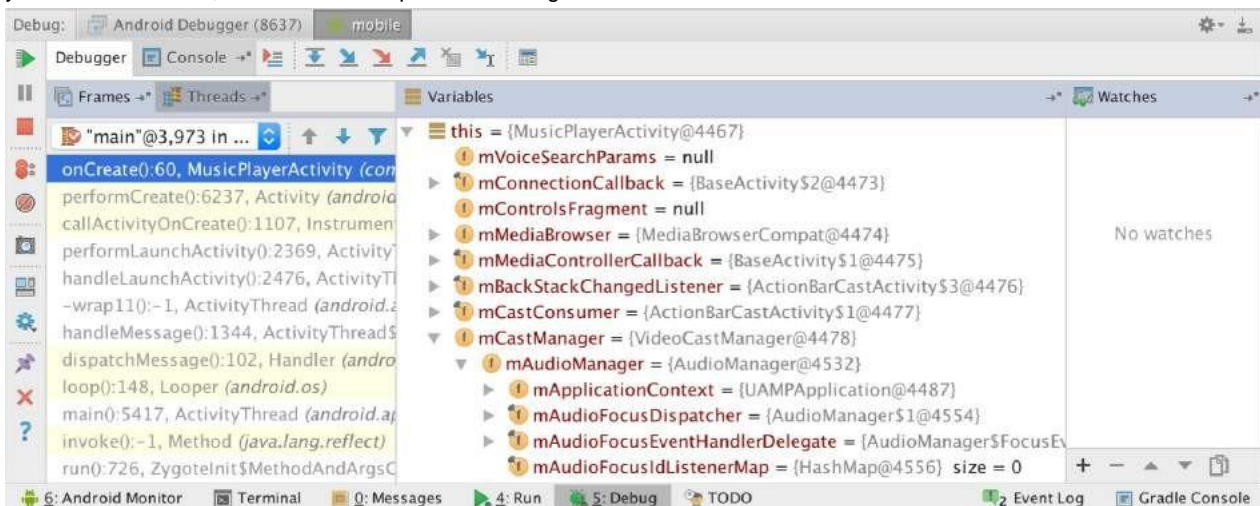
This chapter teaches you how to use the Android Studio debugger to debug your app, how to establish and view breakpoints, how to step through your code, and how to look at variables.

### Running the debugger

Debug mode is identical to the default mode of executing an app. Alternatively, you can attach the debugger to an application that is currently running and run it in debug mode.


### 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 window.



## 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.

The debugger by default displays any connected hardware devices or virtual devices on your computer, along with the device and app processes for the current project. To display all processes across all devices, choose **Show all processes**.

3. Click **OK**. The Debug window 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

A breakpoint is a place in your code where you want to pause the normal execution of your app to perform other actions such as examining variables or evaluating expressions, or executing your code line by line to determine the causes of runtime errors.

### 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 window at that line, next to the line numbers. A red dot appears at that line, indicating a breakpoint.

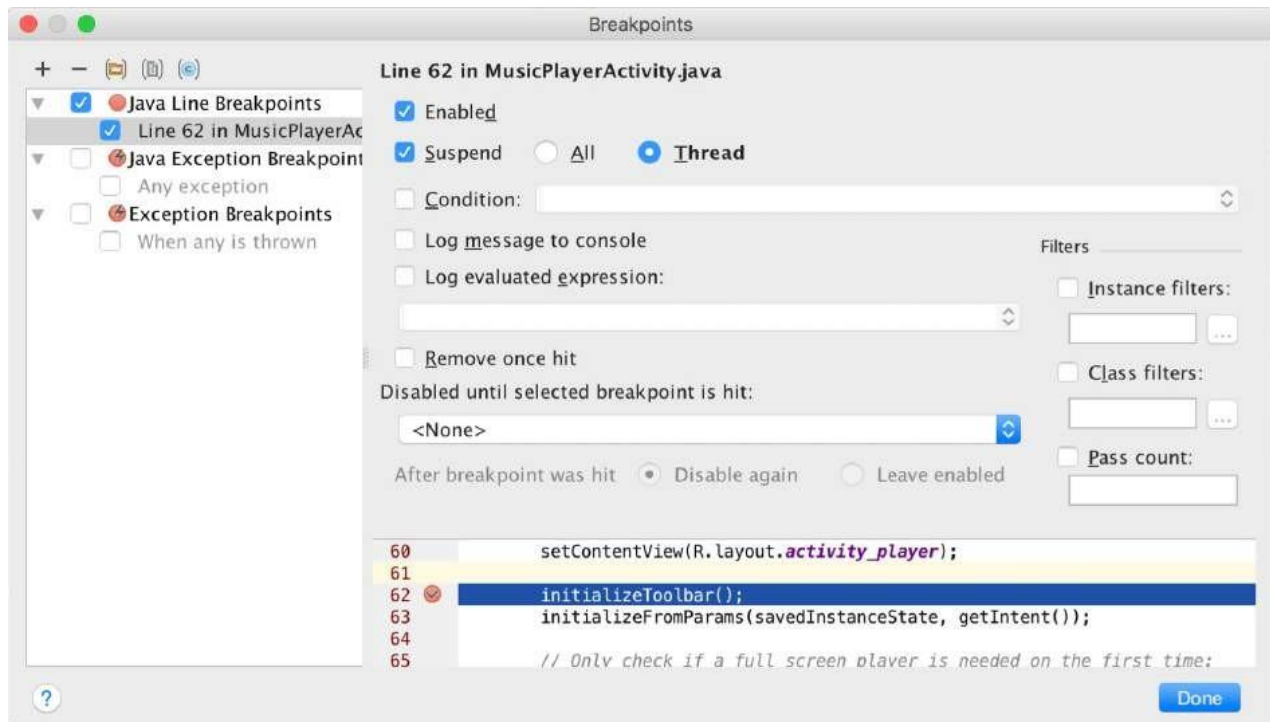
You can also use **Run > Toggle Line Breakpoint** or **Control-F8** (Command-F8 on OS X) to set a breakpoint at a line.

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

When your code execution reaches the breakpoint, Android Studio pauses execution of your app. You can then use the tools in the Android debugger 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 debugger window. The Breakpoints window appears.



In this window all the breakpoints you have set appear in the left pane, and you can enable or disable each breakpoint with the check boxes. 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

You can temporarily "mute" a breakpoint by disabling it rather than deleting it from your code. Disabling a breakpoint may be a better option because removing a breakpoint completely also means losing any conditions or other features you added for that breakpoint.

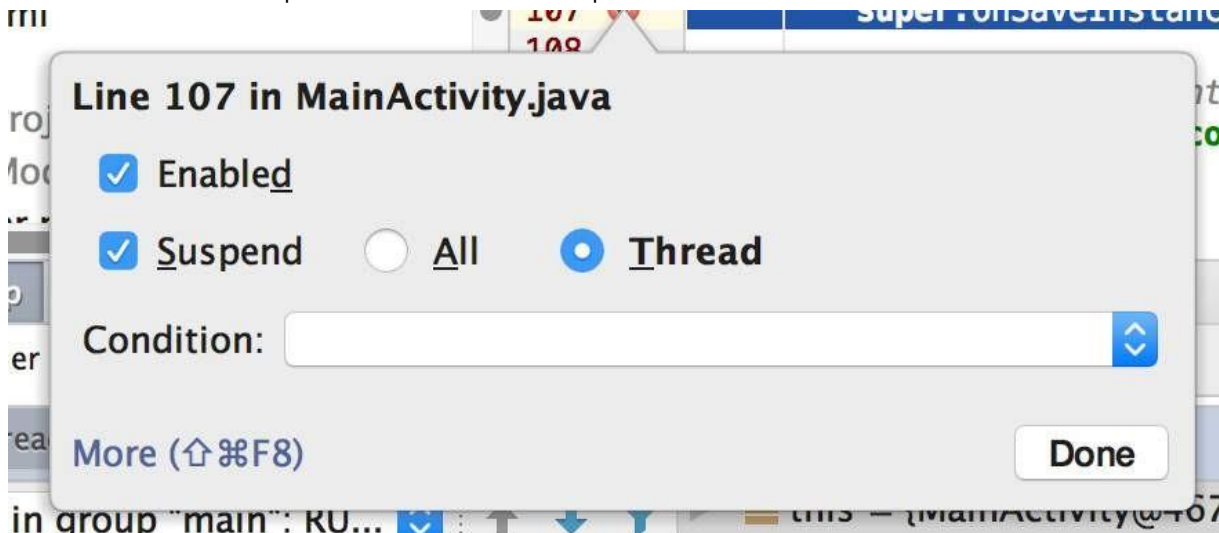
To mute all breakpoints, click the **Mute Breakpoints**  icon. Click the icon again to enable (unmute) all breakpoints.

## 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 on a breakpoint icon, and enter a test in the Condition field.

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



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.

- 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:

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

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

- Click the **Step Over**  icon, select **Run > Step Over**, or type 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.

- Click the **Step Into**  icon, select **Run > Step Into**, or type F7.

Instead of just running the method and staying on the same line, Step Into moves into the execution of a method call on the current line. The new stack frame (the new technique) is updated in the **Frames** view, which you'll learn about in the following section. The file for that class is opened and the current line in that file is highlighted if the method call is found in another class. You can either move deeper into other methods or keep stepping over lines in this new method call.

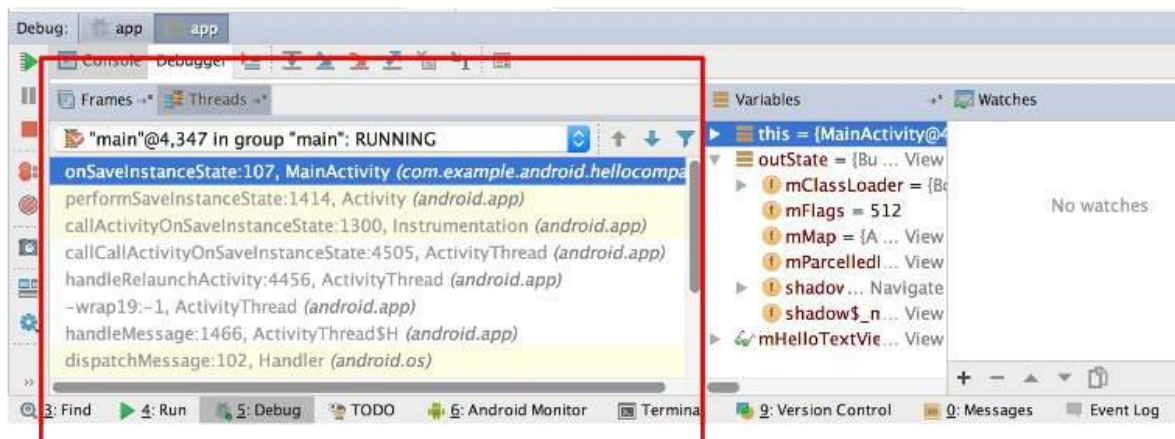
- Click the **Step Out**  icon, select **Run > Step Out**, or type Shift-F8.

Step Out finishes executing the current method and returns to the point where that method was called.

- To resume normal execution of the app, select **Run > Resume Program** or click the Resume  icon.

## Viewing execution stack frames

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



The execution stack displays, in reverse chronological sequence (most recently executed frame first), all classes and methods (frames) that have been called so far in the app. When a frame's execution is complete, that frame is removed from the stack and processing moves on to the following frame.

When a frame's line in the Frames view is clicked, the related source opens in the editor and the line where the frame was first run is highlighted. The status of the execution environment at the time that frame was last accessed is updated in both the Variables and Watches views.

## Inspecting and modifying variables

When the system stops your program on a breakpoint, you may investigate the variables available at the current stack frame in the **Variables** view of the debugger window. Expandable variables, like arrays, allow you to see the individual elements included within an object or collection.

Using static methods and/or variables present in the chosen frame, the **Variables** pane also enables you to instantly evaluate expressions.

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



To modify variables in your app as it runs:

1. Right-click any variable in the Variables view, and select **Set Value**. You can also use F2.
2. Enter a new value for the variable, and type Return.



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

## Setting watches

With the exception of the fact that expressions added to the **Watches** pane survive between debugging sessions, the Watches view and the Variables view offer comparable functionality. You should add watches for variables and fields that you regularly visit or that offer information that is useful for the current debugging session.

Using watches

1. Start testing your app for bugs.
2. Click the plus (+) sign to expand the Watches window.


Enter the name of the variable or expression you want to watch into the text box that displays after you input it.

Select the item, then click the minus (-) button to remove it from the Watches list.

By selecting an item and then clicking the up or down icons, you can rearrange the items in the Watches list.

## Evaluating expressions

To examine the current state of variables and objects in your app, including calling methods on those objects, use the Evaluate Expression tool. To assess a statement:

1. Click the Evaluate Expression  icon, or select **Run > Evaluate Expression**. You can also right-click on any variable and choose Evaluate Expression.

The Evaluate Expression window appears.

2. Enter any expression into the Expression window and click **Evaluate**.

The outcome of the execution is updated in the Evaluate Expression window. It should be noted that the outcome of evaluating an expression depends on the app's current state. You might obtain various outcomes depending on the values of the variables in your app at the time of expression evaluation. Your expressions' variable values also affect how the program is currently functioning.

## 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. These tools include:

- System log (logcat). As you've learned in previous lessons, you can use the Log class to send messages to the Android system log, and view those messages in Android Studio.
  - To write log messages in your code, use the Log class. As you work with your app, log messages collect the system debug output to assist you understand the execution flow. You can learn what component of your program failed from log messages. For additional details on 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 your activity's onCreate() method, and call stopMethodTracing() in that activity's onDestroy() method.

- The Android Debug Bridge (ADB). ADB is a command-line tool that lets you communicate with an emulator instance or connected Android-powered device.
- Dalvik Debug Monitor Server (DDMS). The DDMS tool provides port-forwarding services, screen capture, thread and heap information, logcat, process, and radio state information, incoming call and SMS spoofing, location data spoofing, and more.
- CPU and memory monitors. Android Studio includes a number of monitors to help you visualize the behavior and performance of your app.
- Screenshot and video capture.

## Trace logging and the Android manifest

Beyond setting breakpoints and stepping through code, you have access to a variety of debugging techniques. In order to detect problems with your code, you can also utilize logging and tracing. You can load trace log files in Traceview, which shows the log data in two panels, when you have a trace log file (either by DDMS or by adding tracing code to your application:

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

Likewise, you can set `android:debuggable` in the `<application>` tag of the Android Manifest to `"true"`, which sets whether or not the application can be debugged, even when running on a device in user mode. By default, this value is set to `"false"`.

The module-level `build.gradle` file located inside the `android` block allows you to create and define several build types. Android Studio automatically produces the debug and release build types for you when you create a new module. Android Studio configures the debug build type with `debuggable true` even if it doesn't appear in the build configuration file. This sets up APK signing with a generic debug keystore and enables you to debug the program on safe Android devices. If you want to add or update certain settings, you can add the debug build type to your configuration.

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

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 Toasts
- Disable debugging in the Android manifest by either:
  - Removing `android:debuggable` attribute from `<application>` tag Or
  - setting `android:debuggable` attribute to false

Remove all debug tracing calls from your source code files such as `startMethodTracing()` and `stopMethodTracing()`.

## 3.2: Testing your App

### Contents:

- Introduction
- About testing
- Setting up testing
- Creating and running unit tests

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

### About testing

Even if your app builds, runs, and appears the way you want it to on various devices, you still need to make sure that it will function as you anticipate it to in every circumstance, especially as it develops and evolves. Even if you attempt to manually test your app after each change—a laborious task at best—you can miss something or fail to foresee what end users would do to make it malfunction.

A crucial step in the software development process is the writing and execution of tests. This does not eliminate the necessity for additional testing; rather, it only provides you with a strong foundation from which to build. "Test-Driven Development" (TDD) is a well-known software development methodology that places tests at the center of all software development for an application or service.

As your program grows bigger and more complicated, testing your code can help you find problems early on, when fixing them is cheapest. It can also make your code more robust. You can test isolated, repeatable, and automatable tiny chunks of your app using tests built into your code. Due to the fact that the code you create to test your app only exists on your development machine together with the code for your app 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 those that the Java Virtual Machine (JVM) compiles and executes solely on your personal computer. Use local unit tests to test the components of your app (like the internal logic) that do not require access to the Android framework, a device running Android, or an emulator, or for which you may make fictitious ("mock" or "stub") objects that mimic their behavior.

*Instrumented tests* are tests that run on an Android device or emulator. These tests have access to the Android framework and to Instrumentation information such as the app's Context. When checking that your app's components work properly with other apps, you can utilize instrumented tests for integration, UI, and unit testing. For UI testing, instrumented tests are most frequently used, allowing you to verify that your app responds appropriately when a user interacts with its features or submits a particular input.

The Espresso framework, which enables you to create automated UI tests, is used for the majority of user interface testing types. In a subsequent chapter, you'll read more about Espresso and instrumented tests.



## Unit Testing

The cornerstone tests in your app testing plan should be unit tests. You can check the logic of certain functional code regions or units by designing and executing unit tests against your code. Running unit tests after each build enables you to identify and address issues brought on by app code changes.

A unit test, which could be a method, class, or component, often tests the functionality of the smallest possible unit of code in a repeatable manner. When you need to check the logic of a certain piece of code in your application, create unit tests. If you are unit testing a class, for instance, your test might determine whether the class is in the appropriate state. You could test a method's behavior under various parameter values, including null. Usually, a unit of code is tested independently; your test only keeps track of changes to that unit. You can separate your unit from its dependencies by using a mocking framework like Mockito. 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:

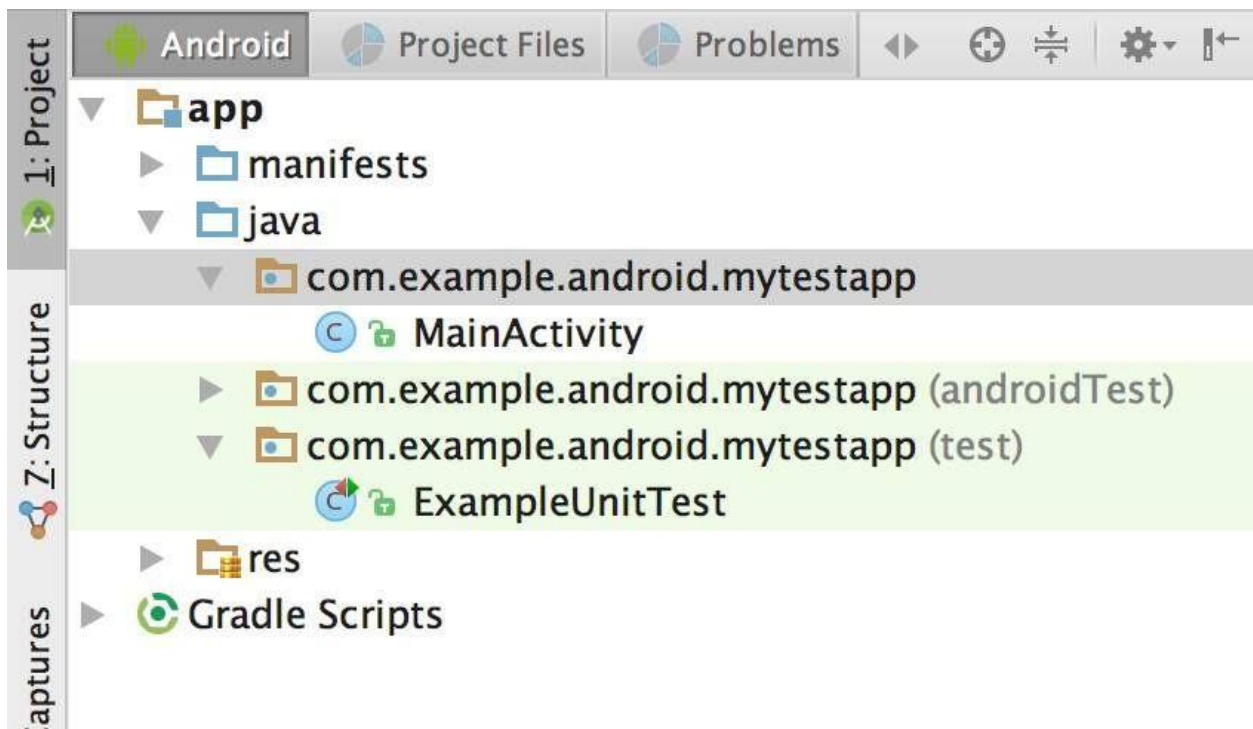
- Organize your tests in a *source set*.
- Configure your project's gradle dependencies to include testing-related APIs.

## Android Studio source sets

*Source sets* are a collection of related 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 **androidTest** source set, for Android instrumented tests.

Source sets appear in the Android Studio Android view 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.

## Configure Gradle for test dependencies

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

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

```
dependencies {  
    // Required -- JUnit 4 framework  
    testCompile 'junit:junit:4.12'  
    // Optional -- hamcrest matchers  
    testCompile 'org.hamcrest:hamcrest-library:1.3'  
    // Optional -- Mockito framework  
    testCompile 'org.mockito:mockito-core:1.10.19'  
}
```

After you add dependencies to your build.gradle file you may have to sync your project to continue. Click **Sync Now** in Android Studio when prompted.

## Configure 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 only demonstrates the default runner for unit tests. To set `AndroidJUnitRunner` as the default test runner in your Gradle project, add the following dependency to your build.gradle file. There may already be dependencies in the defaultConfig section. If so, add the `testInstrumentationRunner` line to that section.

```
android {  
    defaultConfig {  
        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`.

### Create 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, use 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 on the test source set folder and select **New > Java Class**.
3. Name the file and click **OK**.

### Write 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 set up for the test.
- The `@Test` annotation marks a method as an actual test.

For more information on JUnit Annotations, see the [JUnit Reference documentation](#).

```
@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);  
        assertEquals(2d, resultAdd);  
    }  
}
```

The `addTwoNumbers()` method is the only 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. 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 assertions, see the JUnit reference documentation for the Assert class. For more information on the hamcrest framework, see the Hamcrest Tutorial.

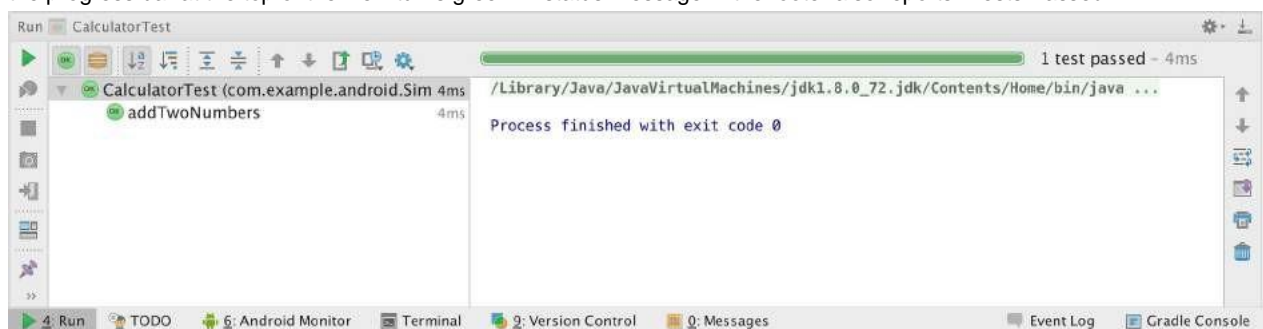
Note that the `addTwoNumbers()` method in this example includes only one assertion. The general rule for unit tests is to provide a separate test method for every individual assertion. Grouping more than one assertion into a single method can make your tests harder to debug if only one assertion fails,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

## Run your tests

To run your local unit tests, use these steps:

- To run a single test, right-click that test method and select **Run**.
- To test all the methods in a test class, right-click the test file in the project view and select **Run**.
- To run all tests in a directory, right-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."



## 3.3: The Android Support Library

### Contents:

- Introduction
- About the Android Support Library
- Support libraries and features
- Setting up and using the Android Support Library

In this chapter you'll explore the Android Support Library, part of the Android SDK tools. You can use the Android Support Library for backward-compatible versions of new Android features and for additional UI elements and features not included in the standard Android framework.

## About the Android Support Library

The Android SDK tools include a number of 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 a previous 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, action bars, or Material Design elements. The support library provides access to those features on older devices.
- Additional layout and user interface 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 a later 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

Apps operating on previous Android platforms can support functionality made accessible on later platforms thanks to support libraries. For instance, a Material Design element cannot be displayed in an app that runs on an Android version lower than 5.0 (API level 21), as that version of the Android framework does not support Material Design. But if the app uses the v7 appcompat library, it will have access to many of the API level 21 capabilities, including Material Design support. Your app will be able to provide a more consistent experience on a wider range of platform versions as a consequence.



The compatibility layer between various versions of the framework APIs is also provided by the support library APIs. This layer of compatibility intercepts API requests invisibly and modifies the arguments supplied, handles the operation directly, or redirects the request. By utilizing the methods of the compatibility layer in the case of the support libraries, you can guarantee compatibility between earlier and more recent Android releases. Every time Android is updated, new classes and methods are added, and maybe some older classes and methods are deprecated. The support libraries include compatibility classes that can be used for backward compatibility. You can identify these classes by their name as their names include "Compat" (such as ActivityCompat).

The behavior of a method that an app calls from the support class depends on the Android version being used. The support library uses the framework if the device has the required framework capability. The support library makes an effort to offer similar compatible behavior with the APIs it has access to if the device is running an older version of Android.

Most of the time, you won't need to create complicated code to verify the Android version and carry out various actions based on that version. The support library will do those checks and help you make the right decision.

When in doubt, pick a compatibility class for a support library over a framework class.

## Versions

Each package in the support library has a three-part version number (X.Y.Z) that corresponds to a specific revision of the library and an Android API level. The support library for API 22 is version 3.4, for instance, and has the version number 22.3.4. Use the most recent, or an updated version of, the support library for the API that your app is built and aimed for. For instance, if your application targets API 25, use the support library's version 25.X.X.

You can always use a support library that is more recent than the one for your desired API. For instance, you can utilize version 25 or higher of the support library if your app targets API 22.

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

The name of the support library itself reveals the API level that the library is backward-compatible with in addition to the actual version number. A support library for an API greater than the lowest API your app supports cannot be used in that app. For instance, you cannot utilize the v13 support library or the v14 preferences support library in your app if the minimum API your app supports is 10. Your minimum API must be higher than the greatest number if your app requires several support libraries; for example, if your app uses support libraries for versions 7, 13, and 14, your minimum API must be at least 14. A minimum SDK of API 9 is required for all support libraries, including the versions 4 and 7.

## 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 a later chapter.

### v4 support library

The v4 support libraries include the largest set of APIs compared to the other libraries, including support for application 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 Action Bar user interface design pattern and support for material design user interface 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 user interface 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 fragments 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 user interfaces 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 your build.gradle file.

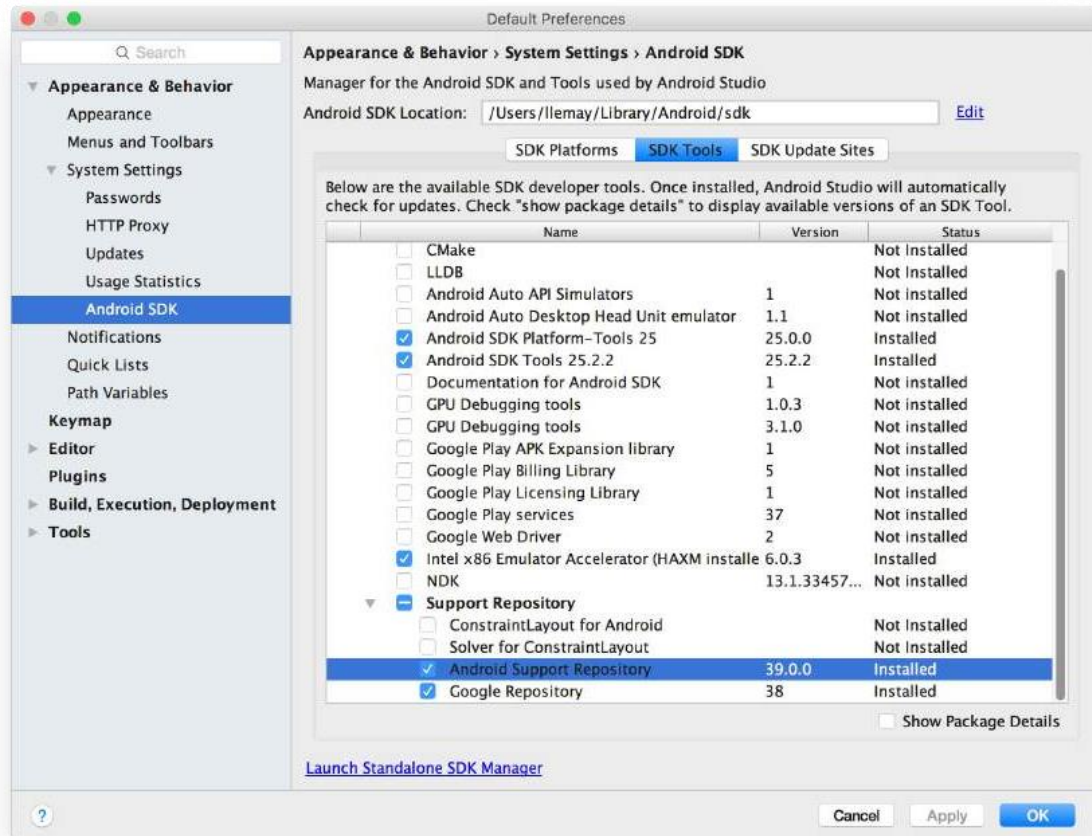
## Download the support library

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.
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](https://developer.android.com/support).
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:23.3.0
```

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** pane is open and the Android tab is clicked.
2. Expand **Gradle Scripts**, if necessary, 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 of build.gradle, near the end of the file.

The dependencies section for a new project may already include dependencies 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:23.3.0'
```

5. Update the version number, if necessary.

If the version number you specified is lower than the currently available library version number, Android Studio will warn 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 type Shift+Enter and choose "Change to XX.X.X" where XX.X.X is the updated version number.

6. 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 views, you must always use the fully-qualified name of that view 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>
```

**Note:** You'll learn about `CoordinatorLayout` in a later 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.

```
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 }  
}
```

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.