Kyle Mew

Mastering Android Studio 3

Build Dynamic and Robust Android applications

Packt

Mastering Android Studio 3

Build Dynamic and Robust Android applications

Kyle Mew

Packt

BIRMINGHAM - MUMBAI

Mastering Android Studio 3

First published: August 2017

Production reference: 1230817

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78646-744-7

www.packtpub.com

Credits

Author
Kyle Mew

Reviewer
Jessica Thornsby

Commissioning Editor
Wilson D'souza

Acquisition Editor
Shweta Pant

Content Development Editor
Aditi Gour

Technical Editor
Akansha Bathija

Copy Editor
Dhanya Baburaj

Project Coordinator
Ritika Manoj

Proofreader
Safis Editing

Indexer
Mariammal Chettiyar

Graphics
Jason Monteiro

Production Coordinator
Shantanu Zagade

About the Author

Kyle Mew has been programming since the early '80s and has written for several technology websites. Also, he has written three radio plays and four other books on Android development.

About the Reviewer

Jessica Thornsby studied poetry, prose, and scriptwriting at Bolton University before discovering the world of open source and technical writing, and has never looked back since. Today, she is a freelance technical writer and full-time Android enthusiast; She is also the author of *Android UI Design* and the co-author of *iWork: The Missing Manual*.

Customer Feedback

Table of Contents

Preface

Welcome to *Mastering Android Studio 3*, a comprehensive guide to the latest and most sophisticated Android development environment. Whether you are new to IDEs or want to migrate from another, such as Eclipse, this book uses practical examples to demonstrate how Android Studio facilitates each stage of development.

Beginning with an introduction to the workspace itself, the book then explores the wide variety of UI design tools the IDE provides, including the powerful visual layout editor, automatic constraint layout tools and animated material icons.

Once the IDE's design tools have been mastered, the book continues by exploring code development with Android Studio and its many helpful and innovative programming tools, such as code completion, template customization, and most importantly, the fantastic testing and profiling tools that are provided with Android Studio 3.

Android Studio is not just a great tool for vanilla coding; it provides all manner of additions and extensions in the form of plugins and native language support for languages such as C++ and Kotlin. It is this extensibility of the native SDK that makes Mastering Android Studio 3 such an essential skill-set for any mobile developer, and the most useful and popular of these are covered in detail to give the reader mastery of what is without doubt one of today's most exciting development tools.

What this book covers

`Chapter 1`, *Workspace Structure*, an introduction the overall workspace. It covers the major features and will be of great use to those who are brand new to the IDE.

`Chapter 2`, *UI Design*, introduces the subject of UI design and development, looking at the automating and time saving features of the layout editor.

`Chapter 3`, *UI Development*, remaining with UI development tools, explores more sophisticated layouts and how these can be easily implemented using code libraries that come packaged in the support repository.

`Chapter 4`, *Device Development*, extends the previous work and looks at developing for physical devices and form factors, covering topics such as screen rotation and shape-aware layouts for wearable devices.

`Chapter 5`, *Assets and Resources*, looks at resource management, in particular Android's use of material icons and vector assets. It demonstrates how Android Studio provides great time-saving features for this aspect of development.

`Chapter 6`, *Templates and Plugins,* is the first of two chapters on extending Android Studio beyond vanilla usage. Here, we look at ready-made and freely available code samples, provided not only within the IDE but also via third-party plugins.

`Chapter 7`, *Language Support*, continues the theme of the previous chapter. Here, we look at how to include C++ and Kotlin code seamlessly.

`Chapter 8`, *Testing and profiling*, explores the powerful testing and profiling tools provided by the IDE and how to use them to test and fine-tune our work.

`Chapter 9`, *Packaging and Distribution*, covers the final aspects of the development cycle. This involves taking a close look at Gradle and covers monetization technologies.

What you need for this book

Both Android Studio SDK are open source and can be downloaded from `developer.android.com`.

Various third-party plugins are mentioned throughout the book-along with their download locations where relevant.

Who this book is for

This book is for Android developers of any experience level who are looking to migrate to or simply master Android Studio 3.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning. Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "In the previous examples, we used `app:srcCompat` as opposed to `android:src`. "

A block of code is set as follows:

```
public class ExampleUnitTest
    {
      @Test
        public void addition_isCorrect() throws Exception {
               assertEquals(4, 2 + 2);
   }
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
buildTypes {
release {
         . . .
         }
    }
    productFlavors {
        flavorDimensions "partial", "full"
```

Any command-line input or output is written as follows:

```
gradlew clean
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this:
"Besides **MakeBuild** and **Analyze**, the **Build** menu has other useful entries, for example, the **Clean Project** item, which removes build artifacts from the build directory "

Warnings or important notes appear like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book-what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of. To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message. If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from `https://www.packtpub.com/sites/default/files/downloads/MasteringAndroidStudio3_ColorImages.pdf`.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books-maybe a mistake in the text or the code-we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title. To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy. Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material. We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

1
Workspace Structure

Android Studio is a powerful and sophisticated development environment, designed with the specific purpose of developing, testing, and packaging Android applications. It can be downloaded, along with the Android SDK, as a single package, but as we shall see throughout the course of this book, it is, in reality, a collection of tools and components, many of which are installed and updated independently of each other.

Android Studio is not the only way to develop Android apps; there are other IDEs, such as Eclipse and NetBeans, and it is even possible to develop a complete app using nothing more than Notepad and the command line, although this last method would be very slow and cumbersome.

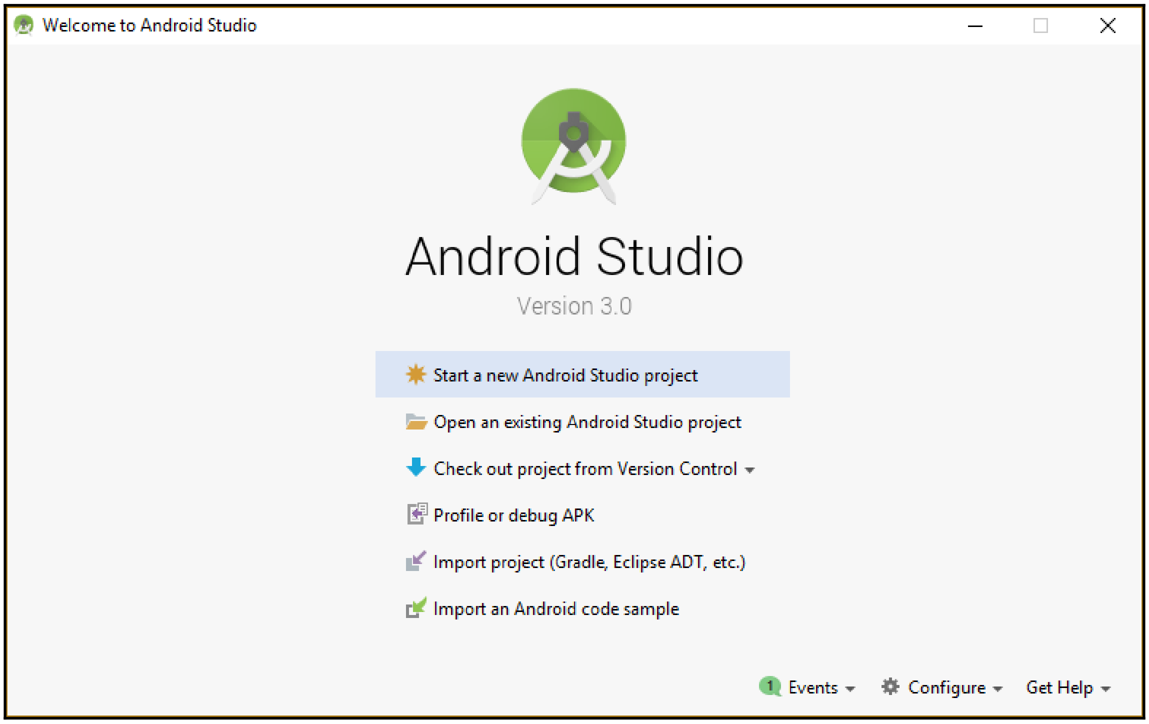
Whether you are migrating from another IDE or just want to get the most from Android Studio, this book will take you through all of its most useful features in the order that these would be encountered during the course of developing an application, starting with UI development and progressing through coding and testing to building and distribution. Android Studio provides some useful and smart tools to assist us at every step of this journey.

Built for a purpose, Android Studio has attracted a growing number of third-party plugins that provide a large array of valuable functions, not available directly via the IDE. These include plugins to speed up build times, debug a project over Wi-Fi, and many more. The most useful and popular of these will be covered in the relevant sections. Throughout the book, we will be finding ways to speed up tedious and difficult tasks using these plugins and Android Studio's built in components.

In this chapter, you will be engaged with the following topics:

- Exploring the differences between Studio and other IDEs
- Taking a brief guided tour
- Learning how the workspace is structured

- Exploring editor windows
- Creating a Material theme
- Understanding Tools windows
- Exploring device filesystems
- Using Instant Run to speed up the build process
- Exploring the SDK manager
- Introducing the virtual device manager
- Importing a project from another IDE


Welcome to Android Studio
Android Studio
Version 3.0
Start a new Android Studio project
Open an existing Android Studio project
Check out project from Version Control
Profile or debug APK
Import project (Gradle, Eclipse ADT, etc.)
Import an Android code sample
Events
Configure
Get Help

Android Studio

If you are already familiar with Android Studio, then you may wish to skip some sections of this introductory chapter, as it is written more with those migrating from other IDEs in mind. However, there are a number of handy tips that you may not have come across before.

Despite being arguably a superior tool, there are some very good reasons for having stuck with another IDE, such as Eclipse. Many developers develop for multiple platforms, which makes Eclipse a good choice of tool. Every developer has deadlines to meet, and getting to grips with unfamiliar software can slow them down considerably at first. This book will speed up this transition so that migrating developers can begin to take advantage of the added features provided by Android Studio with as little interruption as possible.

How Android Studio differs

There are many ways that Android Studio differs from other IDEs and development tools. Some of these differences are quite subtle, such as the way support libraries are installed, and others, for instance the build process and the UI design, are profoundly different.

Before taking a closer look at the IDE itself, it is a good idea to first understand what some of these important differences are. The major ones are listed here:

- **UI development**: The most significant difference between Studio and other IDEs is its layout editor, which is far superior to any of its rivals, offering text, design, and blueprint views, and most importantly, constraint layout tools for every activity or fragment, an easy-to-use theme and style editors, and a drag-and-drop design function. The layout editor also provides many tools unavailable elsewhere, such as a comprehensive preview function for viewing layouts on a multitude of devices and simple-to-use theme and translation editors.
- **Project structure**: Although the underlying directory structure remains the same, the way Android Studio organizes each project differs considerably from its predecessors. Rather than using workspaces as in Eclipse, Studio employs modules that can more easily be worked on together without having to switch workspaces.

TIP

What is called a workspace in Eclipse is called a project in Studio, and what is called a project in Eclipse is a module in Studio.

This difference in structure may seem unusual at first, but any Eclipse user will soon see how much time it can save once it becomes familiar.

- **Code completion and refactoring:** The way that Android Studio intelligently completes code as you type makes it a delight to use. It regularly anticipates what you are about to type, and often a whole line of code can be entered with no more than two or three keystrokes. Refactoring too, is easier and more far-reaching than alternative IDEs, such as Eclipse and NetBeans. Almost anything can be renamed, from local variables to entire packages.
- **Emulation:** Studio comes equipped with a flexible virtual device editor, allowing developers to create device emulators to model any number of real-world devices. These emulators are highly customizable, both in terms of form factor and hardware configurations, and virtual devices can be downloaded from many manufacturers. Users of other IDEs will be familiar with Android AVDs already, although they will certainly appreciate the preview features found in the Design tab.
- **Build tools:** Android Studio employs the Gradle build system, which performs the same functions as the Apache Ant system that many Java developers will be familiar with. It does, however, offer a lot more flexibility and allows for customized builds, enabling developers to create APKs that can be uploaded to TestFlight, or to produce demo versions of an app, with ease. It is also the Gradle system that allows for the modular nature discussed previously. Rather than each library or a third-party SDK being compiled as a JAR file, Studio builds each of these using Gradle.

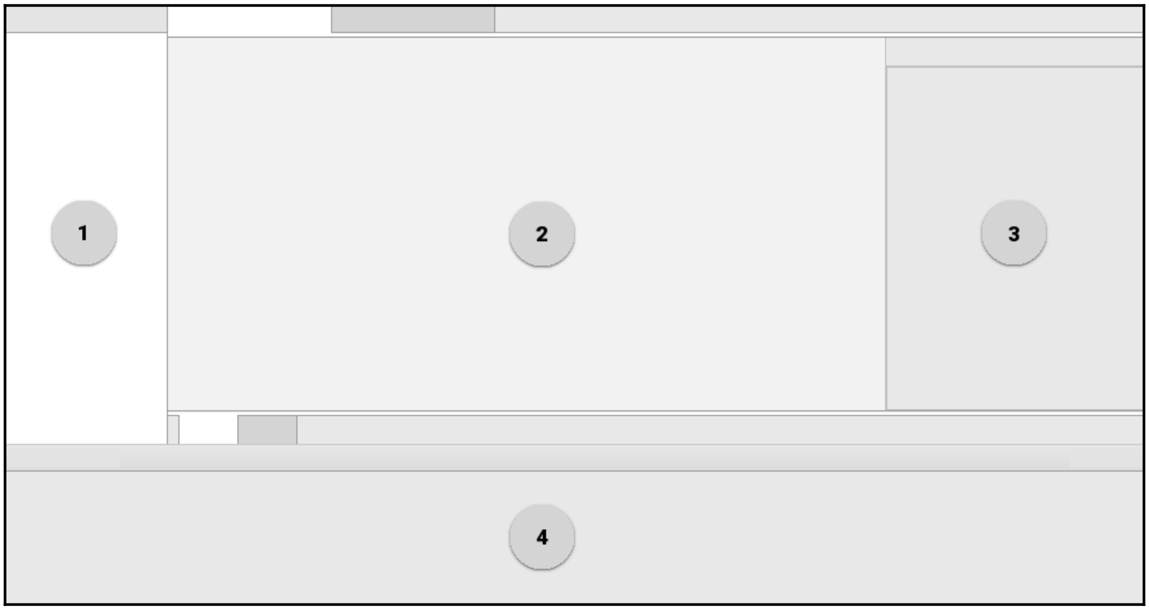
These are the most far-reaching differences between Android Studio and other IDEs, but there are more as well as many features that are unique. Studio provides the powerful JUnit test facility and allows for cloud platform support and even Wi-Fi debugging. It is also considerably faster than Eclipse, which, to be fair, has to cater for a wider range of development needs, as opposed to just one, and it can be run on less powerful machines.

Android Studio also provides an amazing time-saving device in the form of Instant Run. This feature cleverly only builds the part of a project that has been edited, meaning that developers can test small changes to code without having to wait for a complete build to be performed for each test. This feature can bring these waiting times down from minutes to almost zero.

Whether you are new to Android Studio or want to gain more from it, the first step is to take a broad look at its most prominent structures.

Workspace structure

The overall structure of Android Studio is not dissimilar to other IDEs. There are windows for editing text and screen components, others for navigating project structures, and others still for monitoring and debugging. The IDE is highly flexible and can be configured to suit many specific needs and preferences. A typical layout might look like this:



A typical workspace layout

Although these windows can be arranged in any way we please, generally speaking, in the previous screenshot, the four panes might have the following functions:

1. Navigating a project, module, or library
2. Editing text and designing layouts
3. Defining component properties or screen previews
4. Monitoring and debugging

There are times when a large number of open panes can be distracting; for these times, Studio has a Distraction Free Mode, which displays only the current editor window and can be entered from the **View** menu.

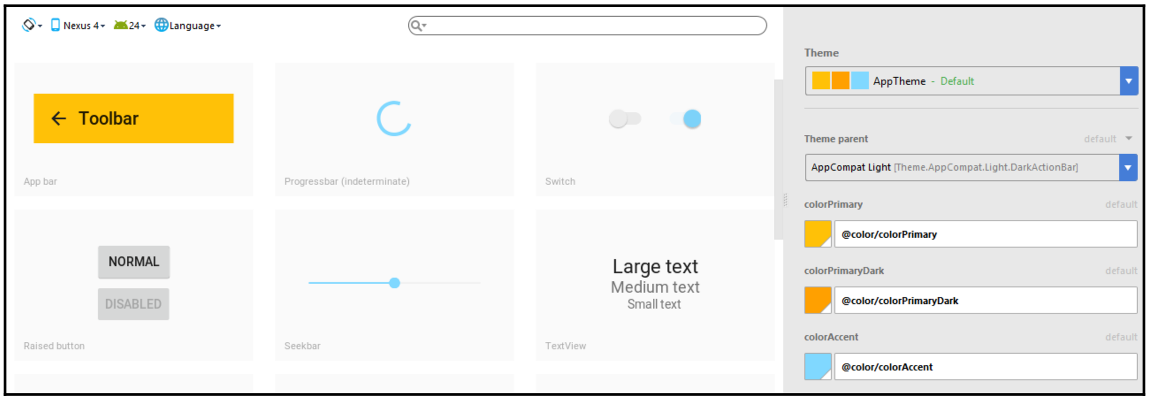
There are many different perspectives we can take on our projects and many ways to organize them. The best way to see how is to take a look at each in turn.

Editor windows

Naturally, the most important of all the windows in an IDE are those where we create and modify the code that underlies all our apps. Not only do we use editors for our XML and Java, there are, among others, editors for simplifying other resources, such as translations and themes. However graphical the editor may be, all Android resources end up as XML files in the `res` directory.

It is quite possible to create most Android resources without ever having to write any code at all. Themes can be created with the corresponding editor with nothing more than a few clicks of a mouse. Nevertheless, if we are to consider ourselves as experts, it is important that we have a good understanding of the underlying code and how and where Studio stores these resources. The following example demonstrates how to create a new Android theme using the theme editor:

1. Start or open an Android Studio project.
2. Open the theme editor from **Tools** | **Android** | **Theme Editor**.



Theme
AppTheme - Default

Theme parent
AppCompat Light [Theme.AppCompat.Light.DarkActionBar]

colorPrimary
@color/colorPrimary

colorPrimaryDark
@color/colorPrimaryDark

colorAccent
@color/colorAccent

The theme editor

3. From the **Theme** drop-down in the top right corner of the editor, select **Create New Theme** and enter a name in the **New Theme** dialog.
4. Leave the **Theme parent** field as-is.
5. Click on the **colorPrimary** thumbnail.
6. Choose a color you like from the resultant swatch with a weight of `500`.
7. In the same manner, select the same color with a weight of `700` for the secondary color.
8. Select a color with a weight of `100` that contrasts nicely with your primary colors for the accent.
9. Open a preview or the design editor to view these changes.

In the preceding example, we created a new theme that will be automatically applied throughout the application. We could have simply edited the default `AppTheme`, but this approach will simplify matters if we later decide to employ more than one theme. The IDE applies these changes straightaway by adding something like the following line to the `res/values/styles.xml` file:

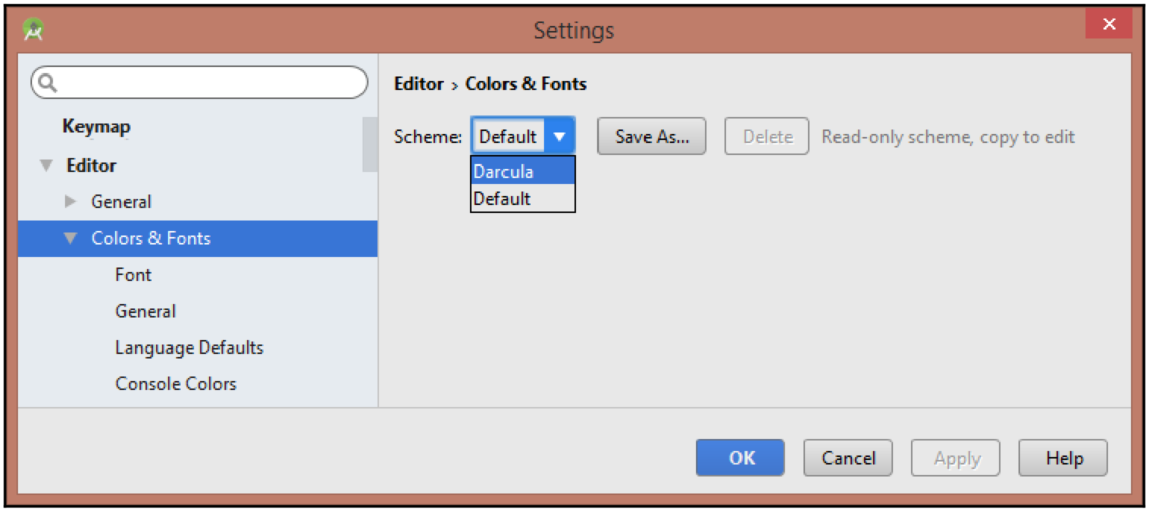
```
<style name="MyTheme" parent="AppTheme" />
```

The actual color changes can be found in the `res/values/colors.xml` file.

The theme editor demonstrates rather nicely how Studio editors can create and modify code after little more than a few mouse clicks from us.

All editors can be maximized with *Ctrl* + *Shift* +*F12*. Use the same keys to return to your original layout.

It is also possible to change the theme of the IDE itself by selecting **Settings | Editor | Colors and Fonts** from the **File** menu, as displayed in the following image:



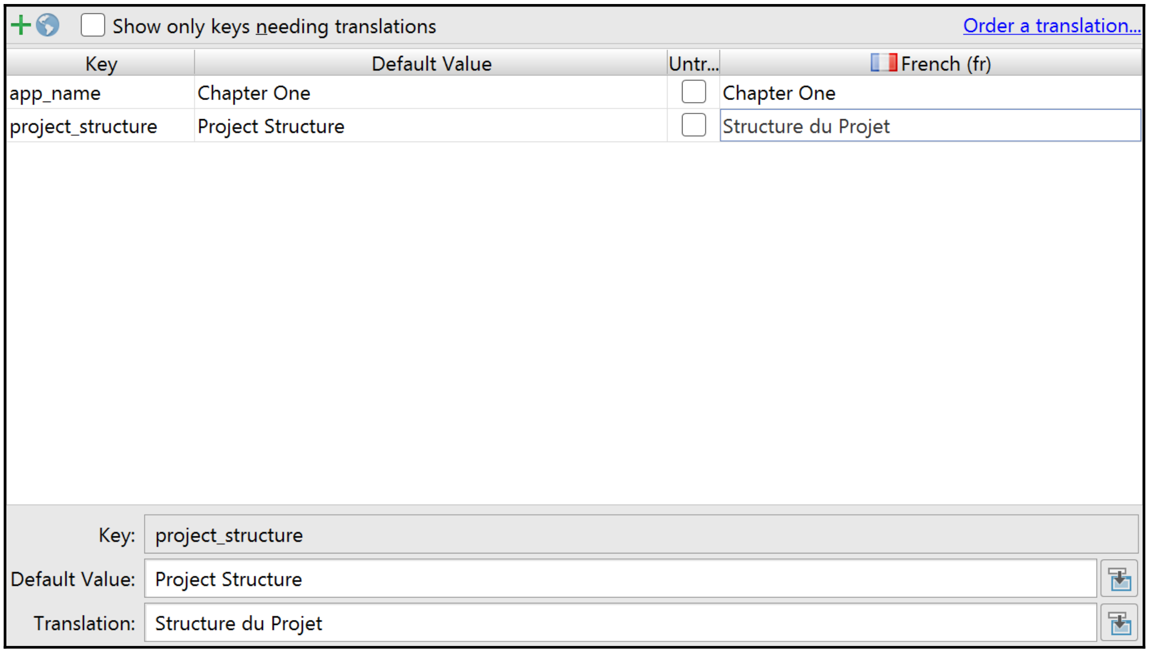
The Studio theme dialog

Android Studio comes equipped with just one alternative color scheme, **Darcula**. This theme presents light text on a dark background and, as such, is far easier on the eye than the default settings, especially for those long, late night development sessions. There are other schemes available online and it can be a lot of fun to design one's own. However, for the purposes of producing printed material, we will stick with the default IDE theme here.

Another good example of a subsidiary editor is the Translations editor, which is also a good way to demonstrate how the project structure differs from other IDEs. The following steps show how this is achieved:

1. Open the Translations editor by right-clicking on the `res/values/strings.xml` file and selecting it from the menu. This can also be found from the **Language** drop-down in the design XML editor.
2. Click on the globe icon near the top left corner of the editor and select a language from the list.

3. Select the string you want to translate in the top pane and enter the value in the lower pane, as shown:



Show only keys needing translations — Order a translation...

Key	Default Value	Untr...	French (fr)
app_name	Chapter One	☐	Chapter One
project_structure	Project Structure	☐	Structure du Projet

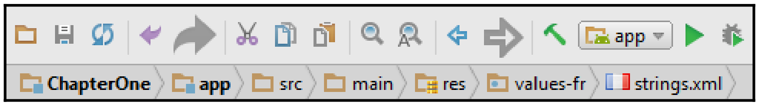
Key:	project_structure
Default Value:	Project Structure
Translation:	Structure du Projet

The Translations editor

This is a remarkably simple exercise, the point of it being to demonstrate how Android Studio stores such resources and how it displays them. The editor has created a new `strings.xml` file, identical to our original in every way apart from the string values of our translated text. This file will be referred to automatically by any device that has that language set as the default by the user.

Judging by the Project Explorer, one might think that there was a project directory called `strings.xml` in the values directory and that it contained two `strings.xml` files. This, in fact, is presented this way only to help us organize our resources. An examination of the `project` folder on disk will show that there are in fact two (or more) folders inside the `res` directory named `values` and `values-fr`. Not only does this help organize our work but it also helps minimize the space our apps take up on a device, as only the resource folders that are needed are installed on an end device.

The actual folder hierarchy can always be determined from the navigation bar directly under the main toolbar.

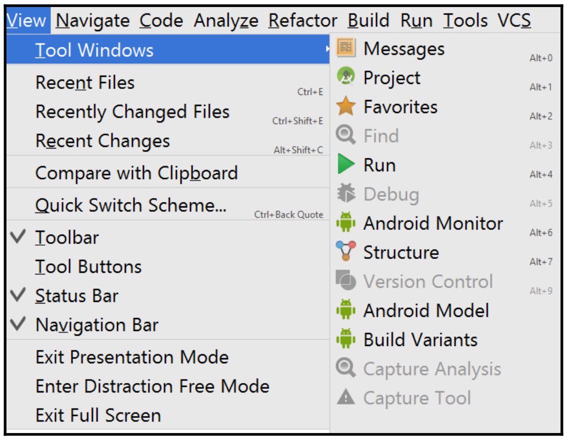


The navigation bar

Themes and Translations are two of the least significant editors but make a good introduction to how Android Studio manages app resources. The majority of a developer's time is spent using code editors and these will, of course, be covered in depth throughout the book. However, although editors make up the core of the IDE, there are many other useful, and even vital, tools available to us, and the most commonly used of these are available from the tools margin.

Tool windows

There are at least a dozen tool windows available to us, more if you have installed plugins. They can be accessed via the **View | Tools Windows** menu, the tools icon on the far left-hand side of the status bar at the bottom of the workspace, or by pressing *Alt* and the corresponding number key to open a specific tool window.

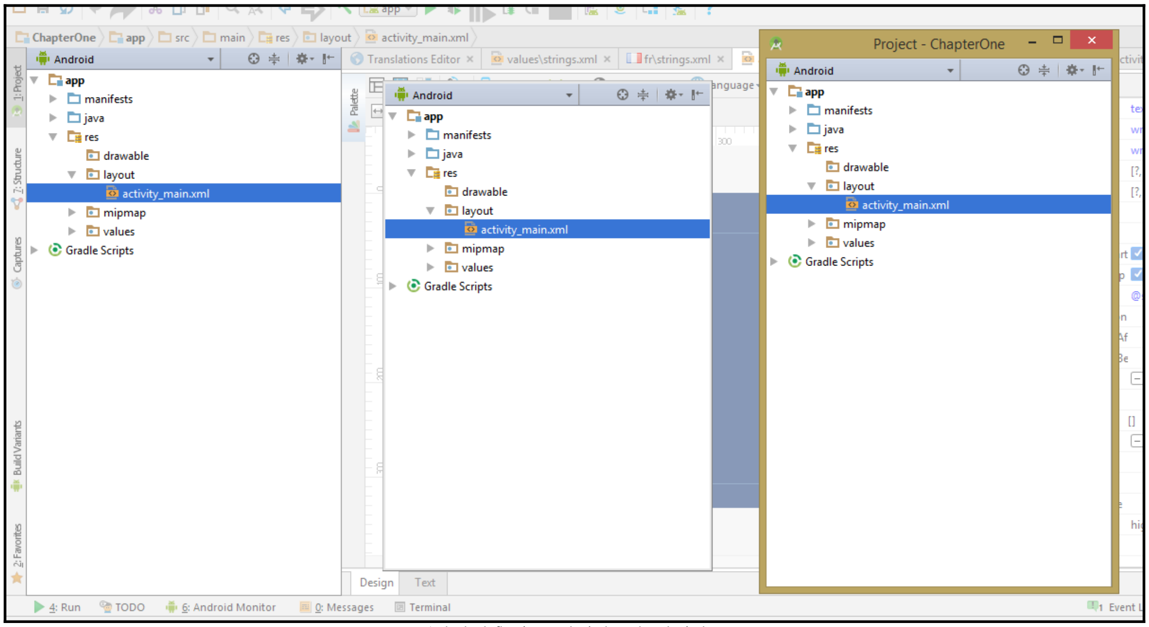


The Tool Windows menu

Tool Windows are highly configurable, and each window can be set as docked, floating, or contained in its own window.

The Tool Windows icon on the status bar can be used to hide and reveal tool window tabs around the border of the workspace.

This is particularly useful when working with more than one screen, as follows:



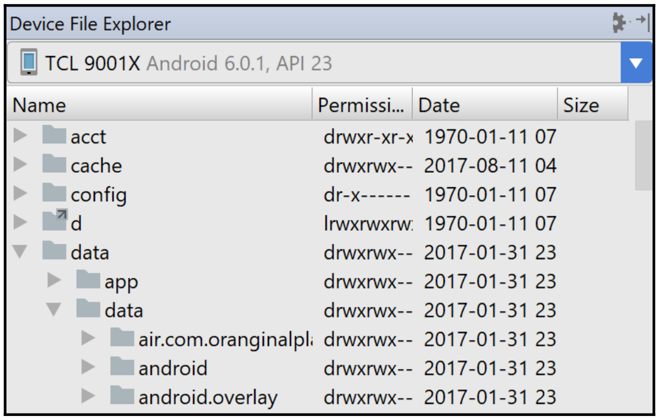
A docked, floating, and windowed tool window

We will be covering all these tools in depth throughout the course of the book. For now though, the following is a brief introduction to the most commonly used tools:

- **Messages**: *Alt* + *0*. This tool produces a paired-down version of the Gradle build process. A more detailed output can be found within the Gradle Console.
- **Project**: *Alt* + *1*. Usually docked to the left of the workspace, this tool is our main navigational tool.
- **Favorites**: *Alt* + *2*. This is a very handy organizational tool, providing quick access to commonly used classes and components. To add any file to the Favorites list, simply right-click on it in the project window and select Add to Favorites from the drop-down menu.

- **Run**: *Alt* + *3*. A powerful diagnostic tool that becomes available when an application is running on a device or emulator.
- **Android**: *Alt* + *4*. This is Studio's main debugging window and is used to monitor log output from a running application and take screenshots.
- **Memory Monitor**: *Alt* + *5*. This incredibly useful tool produces a live graph of memory usage as an application is running.
- **Structure**: *Alt* + *6*. This tool produces detailed information about the current editor, showing a hierarchical view of classes, variables, and other components contained in that particular file.

One of the most useful Tool Windows is the **Device File Explorer** tool. This allows us to browse the filesystem of any connected device or emulator.



Name	Permissi...	Date	Size
acct	drwxr-xr-x	1970-01-11 07	
cache	drwxrwx--	2017-08-11 04	
config	dr-x------	1970-01-11 07	
d	lrwxrwxrw:	1970-01-11 07	
data	drwxrwx--	2017-01-31 23	
app	drwxrwx--	2017-01-31 23	
data	drwxrwx--	2017-01-31 23	
air.com.oranginalpl:	drwxrwx--	2017-01-31 23	
android	drwxrwx--	2017-01-31 23	
android.overlay	drwxrwx--	2017-01-31 23	

The Device File Explorer tool.

TIP

All application files can be found in `data/data`.

Tool windows are fantastically useful and enable us to configure the IDE to suit the particular task we are working on. Being able to select appropriate tools like this is one of Android Studio's most useful features. Of course, Android Studio is nothing more than a frontend interface that allows us to connect with the real power behind Android, the SDK.

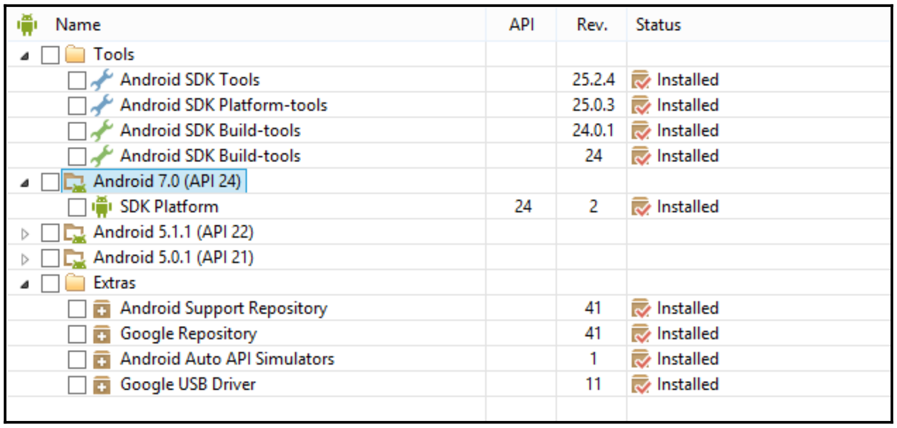
The Android SDK

Technically speaking, it is quite possible to describe the **Software Development Kit**(**SDK**) as not being a part of Android Studio, as it is used by other IDEs. However, the IDE would be useless without it, and now is as good a time as any to take a quick look at it and its manager.

The Android SDK is a huge collection of APIs, consisting of Java classes and interfaces organized into complex but logical hierarchies along with other utilities, such as USB drivers and hardware accelerators.

The SDK and its components update far more frequently than the operating system itself, a setup that users should be blissfully unaware of. Android users think in terms of Lollipop or Honeycomb; as developers, we see the Android world in terms of SDK levels.

The SDK is controlled with **SDK Manager**, which can be accessed via the main toolbar or from **Settings | Appearance & Behavior | System Settings | Android SDK** from the File menu. There is also a standalone SDK Manager, which can be run without Android Studio. This can be found in the following directory: `\AppData\Local\Android\sdk`.



Name	API	Rev.	Status
Tools			
Android SDK Tools		25.2.4	Installed
Android SDK Platform-tools		25.0.3	Installed
Android SDK Build-tools		24.0.1	Installed
Android SDK Build-tools		24	Installed
Android 7.0 (API 24)			
SDK Platform	24	2	Installed
Android 5.1.1 (API 22)			
Android 5.0.1 (API 21)			
Extras			
Android Support Repository		41	Installed
Google Repository		41	Installed
Android Auto API Simulators		1	Installed
Google USB Driver		11	Installed

The Android SDK standalone manager

There are three sections to the SDK manager: tools, platforms, and extras. At the very minimum, you will need to install the latest SDK tools, platform tools, and build tools. You will also need to install the most recent platform and any other platform you intend to target directly. You will also need system images for any virtual devices you wish to create as well as the **Google USB driver** and **HAXM hardware accelerator**.

If you have been using Eclipse to develop Android apps, you will be acquainted with the Android support libraries. When using Android Studio, it is the Support Repository that should be installed instead.

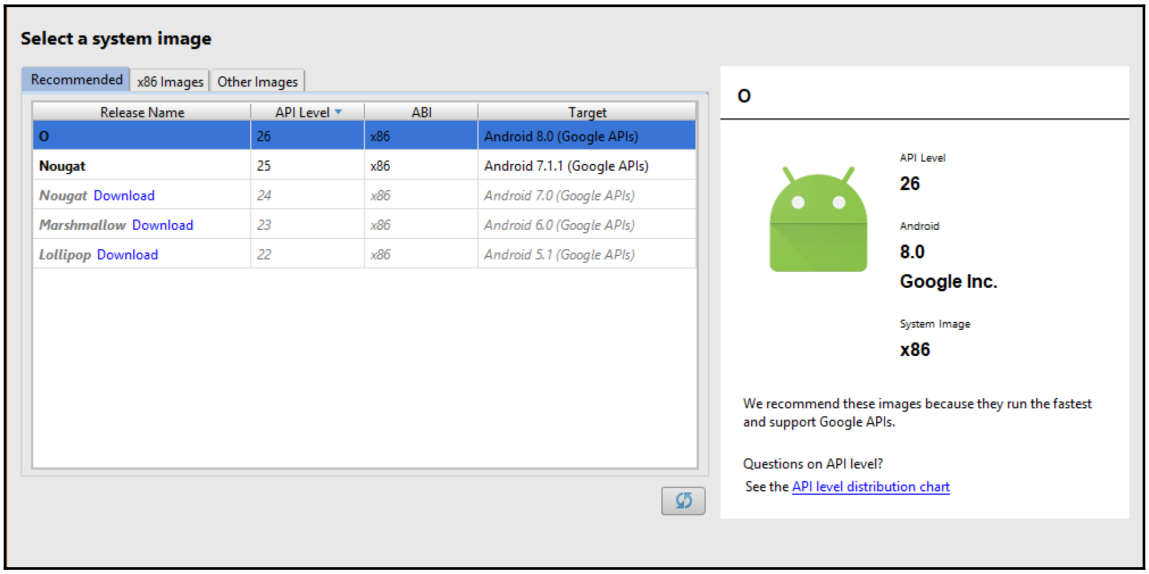
The easiest way to manage the various updates is to set them to be installed automatically, and this can be done from the **Settings** dialog (*Ctrl* + *Alt* + *S*) under **Appearance and Behavior** | **System Settings** | **Updates**.

The SDK forms the backbone of our development environment, but however well we master it, we still need some way to test our creations, and in the absence of a large number of real devices, this depends on creating virtual devices with the Android device emulator.

Virtual devices

There are so many Android devices available on the market that it would be an impossibility to thoroughly test our apps on very many real devices . It is for this reason that the system allows us to create emulated devices using the virtual device manager.

The AVD Manager allows us to create both form factor and hardware profiles from scratch and to provide several ready-made virtual devices and system images that can be downloaded from various manufacturers' websites.



Select a system image

Release Name	API Level	ABI	Target
O	26	x86	Android 8.0 (Google APIs)
Nougat	25	x86	Android 7.1.1 (Google APIs)
Nougat Download	*24*	*x86*	*Android 7.0 (Google APIs)*
Marshmallow Download	*23*	*x86*	*Android 6.0 (Google APIs)*
Lollipop Download	*22*	*x86*	*Android 5.1 (Google APIs)*

O

API Level
26

Android
8.0
Google Inc.

System Image
x86

We recommend these images because they run the fastest and support Google APIs.

Questions on API level?
See the API level distribution chart

AVD configuration screen

Android emulators can be notoriously slow, even on very powerful machines, and this is to be expected, as creating a fully functioning virtual device is a remarkably complex task. There are, however, a few things that can be done to speed things up a little by designing each virtual device to match the particular tasks of the app we are developing. For example, if your app does not make use of the device camera, then do not include it in the configuration. Likewise, do not allocate much more memory than the app itself requires.

Android virtual devices are not the only option available to us, and there are a small but growing number of third-party emulators. Many of these are designed with gamers rather than developers in mind; although Genymotion is specifically a development tool, it contains more functions and is generally faster than the native emulators. Its only drawbacks are that is only free for personal use and only provides system images for phones and tablets and not wearables or large screen devices, such as TVs.

Real-world devices naturally respond far faster than any emulator and, when it comes to testing basic functionality, using our own devices will provide swifter results. This approach is great for testing the fundamentals of an app but provide little to no feedback on just how our apps will look on the wide variety of screen sizes, shapes, and densities that Android devices can have.

Using real devices is a fast way to test application logic but developing apps for specific models or even generic size and shapes will inevitably require the creation of virtual devices. Fortunately, Android Studio comes equipped with an accelerated build process: Instant Run.

Instant Run

In earlier versions of Android Studio, each time a project was run on any kind of device, a full build had to be performed. Even if we made only tiny changes to our code, we would still have to wait for the entire app to be rebuilt and reinstalled. This could prove very time-consuming, especially on less powerful machines. This slowness often resulted in having to test several modifications at once, leading to a more complex debugging process than is ideal.

Instant Run attempts to build only those classes or activities that have been changed since the last build, and providing the manifest file has not been edited, the app is not even reinstalled, and in some cases, the launch activity is not even restarted.

As Instant Run is a recent innovation, it is unfortunately not available on all versions of Android and, to take full advantage of it, you will need to set the minimum SDK level to API 21 or higher, although elements of it will work with API level 15 and higher. In Android Studio, this level is set from the `build.gradle (Module: app)` file, as follows:

```
android {
    compileSdkVersion 25
    buildToolsVersion "25.0.1"
    defaultConfig {
        applicationId "com.mew.kyle.chapterone"
        minSdkVersion 21
        targetSdkVersion 25
        versionCode 1
        versionName "1.0"
        testInstrumentationRunner
"android.support.test.runner.AndroidJUnitRunner"
    }
```

More often than not, we try to make our apps as backward-compatible as possible and developing an app that only works with API level 21 or higher would seriously limit the number of users we can reach. However, the time Instant Run saves us makes it worthwhile to test and debug an app API 21 or higher and then, later, reassemble it to match the versions we wish to target.

When deciding which Android versions to target, a useful dashboard displays the up-to-date usage data of platforms and screens. It can be found at `developer.android.com/about/dashboards/index.html`.

Moving from another IDE to Android Studio need not be a difficult transition and will prove invaluable once complete. However, it may be that you have projects developed in other IDEs that you wish to continue developing using Studio. Fortunately, this is a simple task, as the following section demonstrates.

Importing projects into Android Studio

Eclipse is, without question, one of the finest development tools around and, after 15 years, many of us have become very familiar with it. When it comes to developing for a variety of platforms, Eclipse is a fantastic tool but cannot compete with Android Studio when it comes to developing Android applications.

If you are migrating from Eclipse, you will more than likely have projects you have been working on that you wish to import into Studio. The following steps demonstrate how this is done:

1. First ensure that your Eclipse ADT root directory contains both the `src` and `res` directories and the `AndroidManifest.xml` file.
2. Make a note of any Eclipse third-party plugins you have used as you will need to install equivalents into Studio.
3. Open Android Studio and select **Import Project** from the welcome screen or from **File** | **New** | **Import Project**.
4. Select the folder that contains the manifest and prepare a destination folder, then follow the prompts to complete the import.

The import process makes a complete copy of the project, leaving the original untouched, meaning it can still be worked on in Eclipse if you wish. Unfortunately, it is not possible to import third-party plugins but a large and growing number of plugins are available for Studio, and it is more than likely that you will be able to find equivalents. These can be browsed from **File** | **Settings** | **Plugins**.

If you have several Eclipse projects in the same workspace, then you should import one as a project and the rest as modules.

We will look again at this process when we come to project configuration, but otherwise, from here on in, we will be assuming that all projects are begun in Android Studio.

Summary

This chapter has served as a brief but complete introduction to Android Studio for those readers who are unfamiliar with it. We explored how the workspace is structured and the various flavors of editor available to us. This exploration led us to create a Material Design theme, use tool windows to perform a variety of useful tasks, as well as apply Instant Run to speed up the otherwise time-consuming build process.

The chapter concluded with a quick look at virtual devices and how we can import our projects from other IDEs. With this introduction complete, the following chapters will delve into the layout editor itself, as we see how to design application interfaces that work across the widest number of form factors.

2
UI Design

The one feature that stands out in Android Studio above all others, including the Gradle build system, is the powerful **User Interface**(**UI**) development tools. The IDE provides a variety of views of our designs, allowing us to combine drag and drop construction and hard code in the development of a UI. Android Studio also comes equipped with a comprehensive preview system, which allows us to test our designs on any manner of device before running the project on an actual device. Along with these features, Android Studio also includes useful support libraries, such as the design library for creating material design layouts and the Percent Support Library for simplifying complex, proportional designs.

This chapter is the first of four, covering UI development. In it, we take a closer look at Studio's Layout Editors and tools. We will be building working interfaces using each of the most useful Layout/ViewGroup classes and designing for and managing screen rotation. The chapter continues by exploring Studio's preview system and how XML layout resources are stored and applied. The chapter concludes by returning to themes, Material Design, and the design support library.

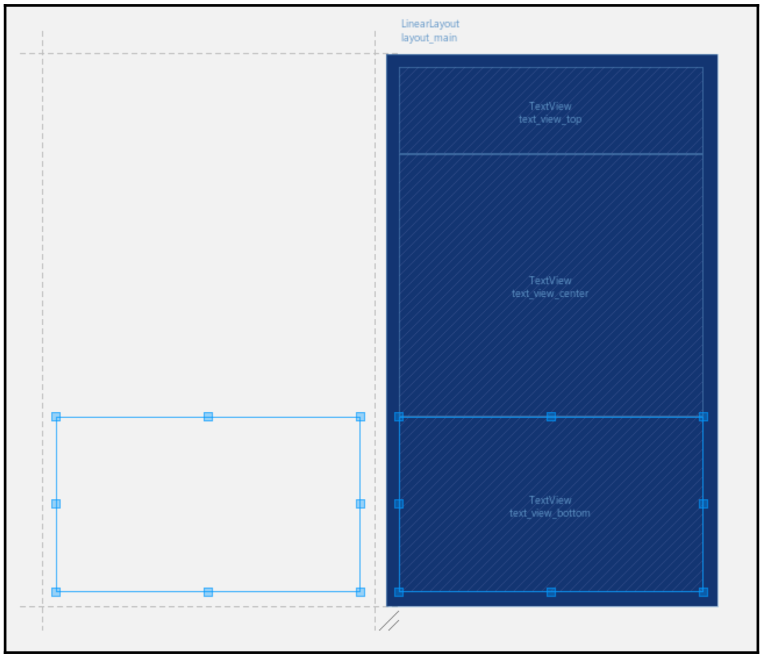
In this chapter, you will learn how to:

- Explore the Layout Editor
- Apply linear and relative layouts
- Install the constraint library
- Create a `ConstraintLayout`
- Apply constraints
- Use the graphical constraints editor
- Add constraint guidelines
- Align `TextView` baselines
- Apply bias

- Use auto-connect
- Build a hardware profile for a virtual device
- Create a virtual SD card

The Layout Editor

If there were only one reason to use Android Studio, it would be the Layout Editor and its associated tools and preview system. The differences are apparent as soon as you open a project. The difference between layout and blueprint view is also shown in the following figure:



The design and blueprint layout views

The **blueprint mode** is new to Android Studio 2.0 and portrays a simplified, outlined view of our UI. This is particularly useful when it comes to editing the spacing and proportions of complex layouts without the distraction of content. By default, the IDE displays both design and blueprint views side by side, but the editor's own toolbar allows us to view only one, and in most cases one would select the mode most suitable to the task in hand.

TIP

The *B* key can be used to toggle between design, blueprint, and combined views as an alternative to the toolbar icons.

It would be perfectly possible to generate every layout required for a project using these graphical views without any knowledge of the underlying code. This is not a very professional approach though, and a good understanding of the XML under the hood is essential for good testing and debugging and, if we know what we are doing, very often tweaking the code is faster than dragging and dropping objects.

The XML responsible for the previous layout is as follows:

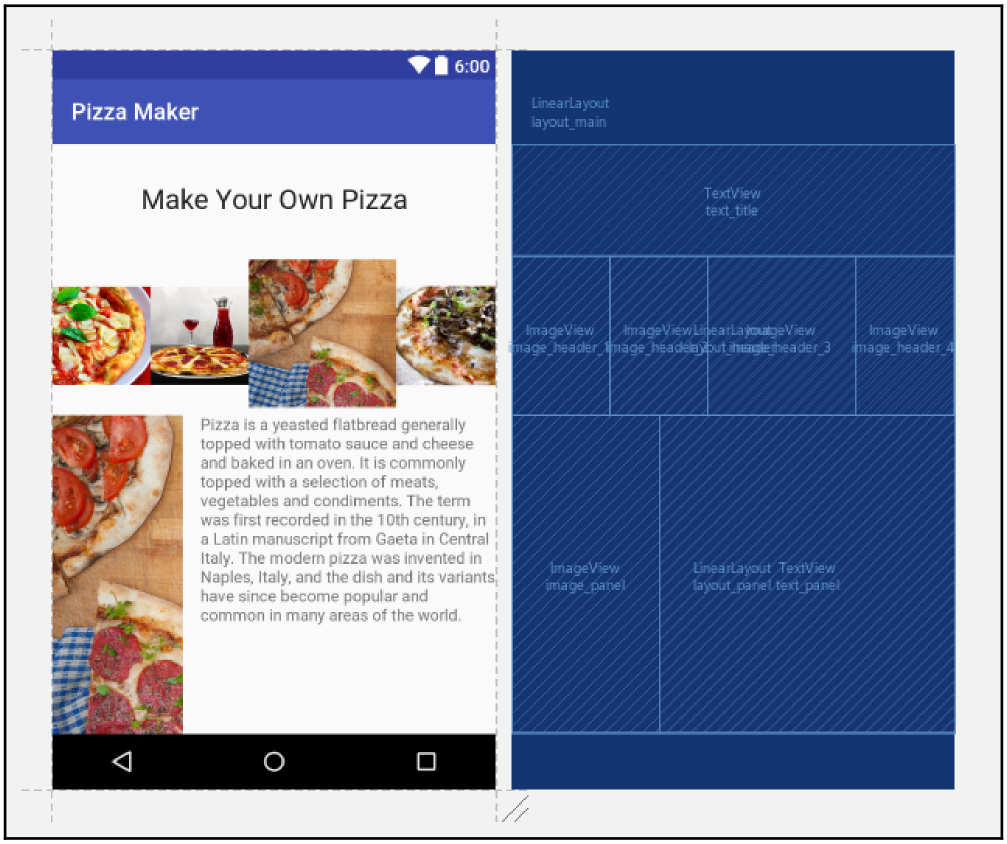
```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/layout_main"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <TextView
        android:id="@+id/text_view_top"
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1" />

    <TextView
        android:id="@+id/text_view_center"
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="3" />

    <TextView
        android:id="@+id/text_view_bottom"
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="2" />

</LinearLayout>
```

Hopefully, the terms used in the preceding code will be familiar to you. The use of `layout_weight` is frequently used with linear layouts to assign proportion, a great time saver when developing for screens with slightly different aspect ratios.

Until recently, the only choice we had for creating more complex UIs were the linear and relative layouts. Both of these are less than ideal, being either unnecessarily expensive or fiddly. Android Studio 2 introduced the constraint layout, which offers an elegant solution to these problems. To best appreciate its value, it makes sense here to first take a look at the older classes, which still have their place in many simpler designs.

Linear and relative layout classes

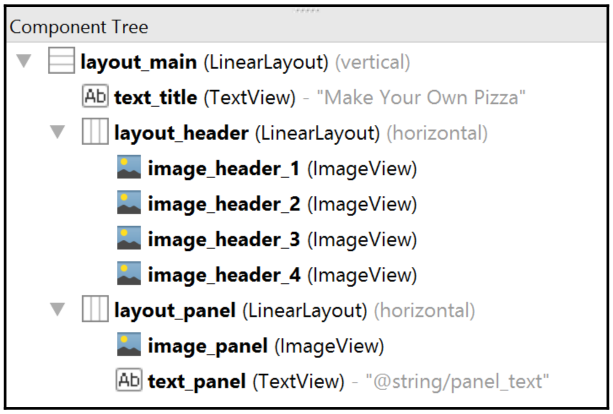
The linear layout is relatively lightweight and very useful for layouts based on single rows or columns. However, more complex layouts require nesting layouts inside each other and this very quickly becomes resource hungry. Take a look at the following layout:



Pizza is a yeasted flatbread generally topped with tomato sauce and cheese and baked in an oven. It is commonly topped with a selection of meats, vegetables and condiments. The term was first recorded in the 10th century, in a Latin manuscript from Gaeta in Central Italy. The modern pizza was invented in Naples, Italy, and the dish and its variants have since become popular and common in many areas of the world.

Nested linear layouts

The preceding layout was built using only linear layouts, as can be seen from the following **Component Tree**:



Component Tree
- **layout_main** (LinearLayout) (vertical)
  - **text_title** (TextView) - "Make Your Own Pizza"
  - **layout_header** (LinearLayout) (horizontal)
    - **image_header_1** (ImageView)
    - **image_header_2** (ImageView)
    - **image_header_3** (ImageView)
    - **image_header_4** (ImageView)
  - **layout_panel** (LinearLayout) (horizontal)
    - **image_panel** (ImageView)
    - **text_panel** (TextView) - "@string/panel_text"

The component tree

Although perfectly workable and easy to understand, this layout is not as efficient as it could be. Even a single extra layer of layout nesting will have an impact on performance. Prior to the constraint layout, this problem was solved with the relative layout.

As the name suggests, the relative layout allows us to place screen components in relation to each other, using markup such as `layout_toStartOf` or `layout_below`. This allows us to flatten view hierarchies and the preceding layout could be recreated with just one single relative, root viewgroup. The following code demonstrates how the row of images in the previous layout can be generated without nesting any new layouts:

```
<ImageView
    android:id="@+id/image_header_1"
    android:layout_width="128dp"
    android:layout_height="128dp"
    android:layout_alignParentStart="true"
    android:layout_below="@+id/text_title"
    app:srcCompat="@drawable/pizza_01" />

<ImageView
    android:id="@+id/image_header_2"
    android:layout_width="128dp"
    android:layout_height="128dp"
    android:layout_below="@+id/text_title"
    android:layout_toEndOf="@+id/image_header_1"
    app:srcCompat="@drawable/pizza_02" />
```

```
<ImageView
    android:id="@+id/image_header_3"
    android:layout_width="128dp"
    android:layout_height="128dp"
    android:layout_alignParentEnd="true"
    android:layout_below="@+id/text_title"
    app:srcCompat="@drawable/pizza_03" />

<ImageView
    android:id="@+id/image_header_4"
    android:layout_width="128dp"
    android:layout_height="128dp"
    android:layout_alignParentStart="true"
    android:layout_below="@+id/text_title"
    app:srcCompat="@drawable/pizza_04" />
```

Even if you are new to Android Studio, it is assumed that you will be familiar with linear and relative layouts. It is less likely that you will have encountered the constraint layout, which has been especially developed for Studio to alleviate the shortcomings of these older approaches.

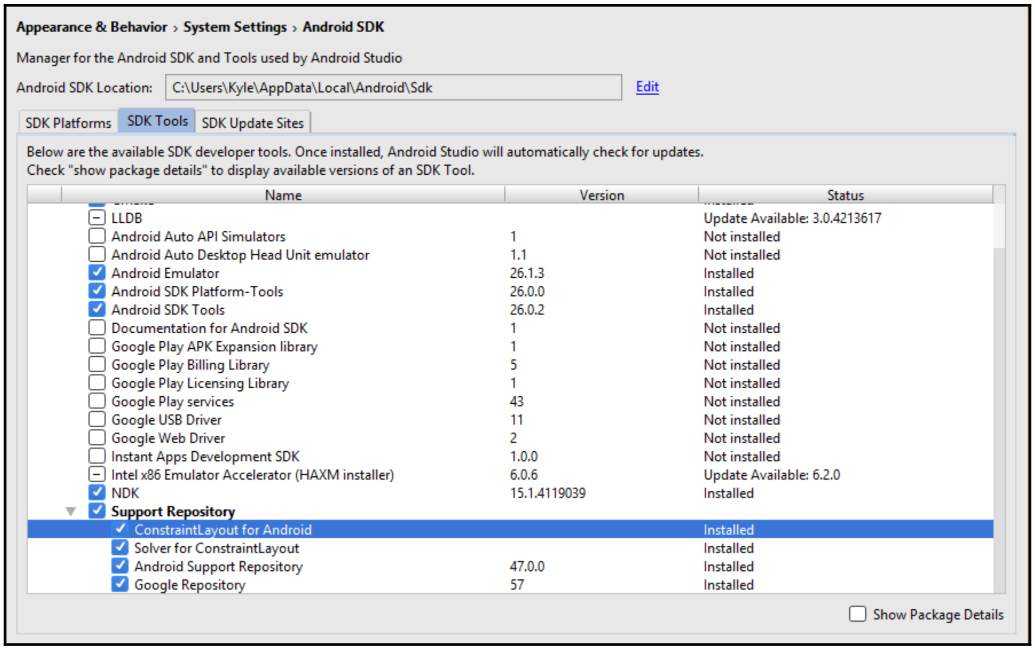
In the previous examples, we used `app:srcCompat` as opposed to `android:src`. This is not strictly required here, but if we wished to apply any tinting to the image and hope to distribute the app for older Android versions, this choice will enable that.

The constraint layout

The constraint layout is similar to the relative layout, in that it allows us to generate complex layouts without having to create memory sapping, view group hierarchies. Android Studio makes creating such layouts far easier, because it provides a visual editor that enables us to drag and drop not only screen components, but also their connections. Being able to experiment with layout structures so easily provides us with a great sandbox environment to develop new layouts.

The following exercise will take you through the process of installing the constraint library so that you can begin experimenting yourself.

1. As of Android Studio 3.0 the `ConstraintLayout` is downloaded by default, but if you want to update an earlier project, you will need to open the SDK manager. The **ConstraintLayout** and constraint solver can both be found under the **SDK Tools** tab, as follows:



Appearance & Behavior › System Settings › Android SDK

Manager for the Android SDK and Tools used by Android Studio

Android SDK Location: C:\Users\Kyle\AppData\Local\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
− LLDB		Update Available: 3.0.4213617
☐ Android Auto API Simulators	1	Not installed
☐ Android Auto Desktop Head Unit emulator	1.1	Not installed
✓ Android Emulator	26.1.3	Installed
✓ Android SDK Platform-Tools	26.0.0	Installed
✓ Android SDK Tools	26.0.2	Installed
☐ Documentation for Android SDK	1	Not installed
☐ Google Play APK Expansion library	1	Not installed
☐ Google Play Billing Library	5	Not installed
☐ Google Play Licensing Library	1	Not installed
☐ Google Play services	43	Not installed
☐ Google USB Driver	11	Not installed
☐ Google Web Driver	2	Not installed
☐ Instant Apps Development SDK	1.0.0	Not installed
− Intel x86 Emulator Accelerator (HAXM installer)	6.0.6	Update Available: 6.2.0
✓ NDK	15.1.4119039	Installed
✓ **Support Repository**		
✓ ConstraintLayout for Android		Installed
✓ Solver for ConstraintLayout		Installed
✓ Android Support Repository	47.0.0	Installed
✓ Google Repository	57	Installed

☐ Show Package Details

The constraint layout API

2. Check the **Show Package Details** box and make a note of the version number as this will be required shortly.
3. Next, add the `ConstraintLayout` library to our dependencies. The simplest way to do this is selecting your module and then the **Dependencies** tab of the **Project Structure** dialog, which can be accessed from the **File** menu.
4. The constraint library can be found by clicking on the **+** button and then **1 Library dependency** and selecting it from the list.
5. Finally, synchronize your project from the toolbar, the **build** menu, or *Ctrl* + *Alt* + *Y*.

This is the simplest way to add module dependencies, but it is always good as developers to understand what is going on under the hood. In this case, we could have added the library manually by opening the module level `build.gradle` file and adding the following, highlighted text to the `dependencies` node:

```
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    androidTestCompile('com.android.support.test.espresso:espresso-
                        core:2.2.2', {
        exclude group: 'com.android.support', module: 'support-annotations'
    })
    compile 'com.android.support:appcompat-v7:25.1.0'
    compile 'com.android.support.constraint:constraint-layout:1.0.0-beta4'
    testCompile 'junit:junit:4.12'
```

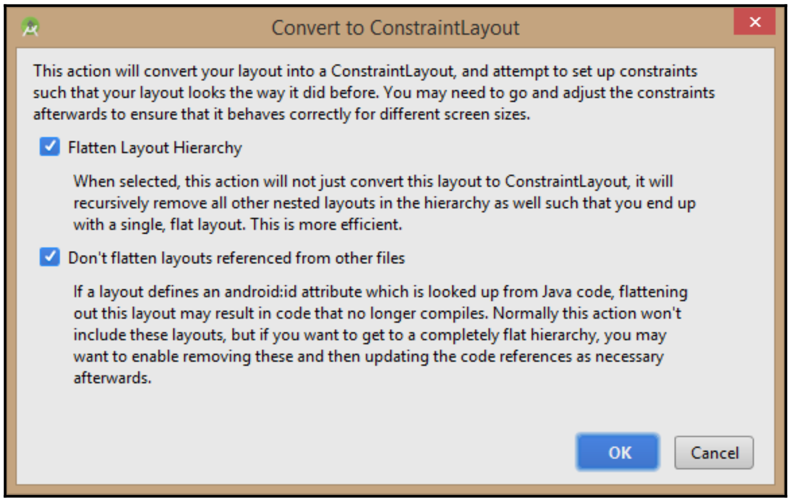
Those of you that have developed with the relative layout will be familiar with commands such as `layout_toRightOf` or `layout_toTopOf`. These attributes can still be applied to a `ConstraintLayout`, but there are more. In particular, the `ConstraintLayout` allows us to position views based on individual sides, for example `layout_constraintTop_toBottomOf`, which aligns the top of our view to the bottom of the one specified.

TIP

Useful documentation on these attributes can be found at: `developer.android.com/reference/android/widget/RelativeLayout.LayoutParams.html`.

Creating a ConstraintLayout

There are two ways to create a ConstraintLayout. The first is to convert an existing layout to a ConstraintLayout, which can be done by right-clicking on the layout in either the component tree or the graphical editor and selecting the **convert** option. You will be presented with the following dialog:



This action will convert your layout into a ConstraintLayout, and attempt to set up constraints such that your layout looks the way it did before. You may need to go and adjust the constraints afterwards to ensure that it behaves correctly for different screen sizes.

☑ Flatten Layout Hierarchy

When selected, this action will not just convert this layout to ConstraintLayout, it will recursively remove all other nested layouts in the hierarchy as well such that you end up with a single, flat layout. This is more efficient.

☑ Don't flatten layouts referenced from other files

If a layout defines an android:id attribute which is looked up from Java code, flattening out this layout may result in code that no longer compiles. Normally this action won't include these layouts, but if you want to get to a completely flat hierarchy, you may want to enable removing these and then updating the code references as necessary afterwards.

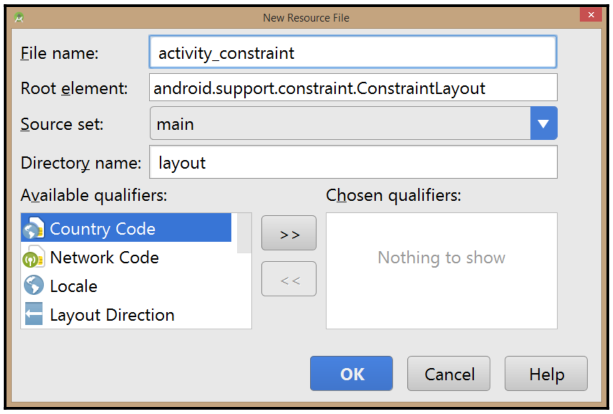
The **Convert to ConstraintLayout** dialog

It is usually best to check both these options, but it is worth noting that these conversions will not always produce the desired results and often view dimensions will require a little tweaking to faithfully reproduce the original.

When it works, the previous approach provides a fast solution, but if we are to master the topic, we need to know how to create constraint layouts from scratch. This is particularly important as once we have become familiar with the working of the constraint layout; we will see that it is by far the easiest and most flexible way to design our interfaces.

The `ConstraintLayout` is so well incorporated with the Layout Editor, that it is perfectly possible to design any layout we choose without ever having to write any XML. However, here we will be looking closely at both the graphical and textual perspectives, so as to develop a deeper understanding of the technology.

You can create a fresh `ConstraintLayout` from the `res/layout` directory in project explorer's context-sensitive menu as a **New | Layout resource file** with the following **root element**:



File name: activity_constraint
Root element: android.support.constraint.ConstraintLayout
Source set: main
Directory name: layout
Available qualifiers: Country Code, Network Code, Locale, Layout Direction
Chosen qualifiers: Nothing to show

Adding a new ConstraintLayout

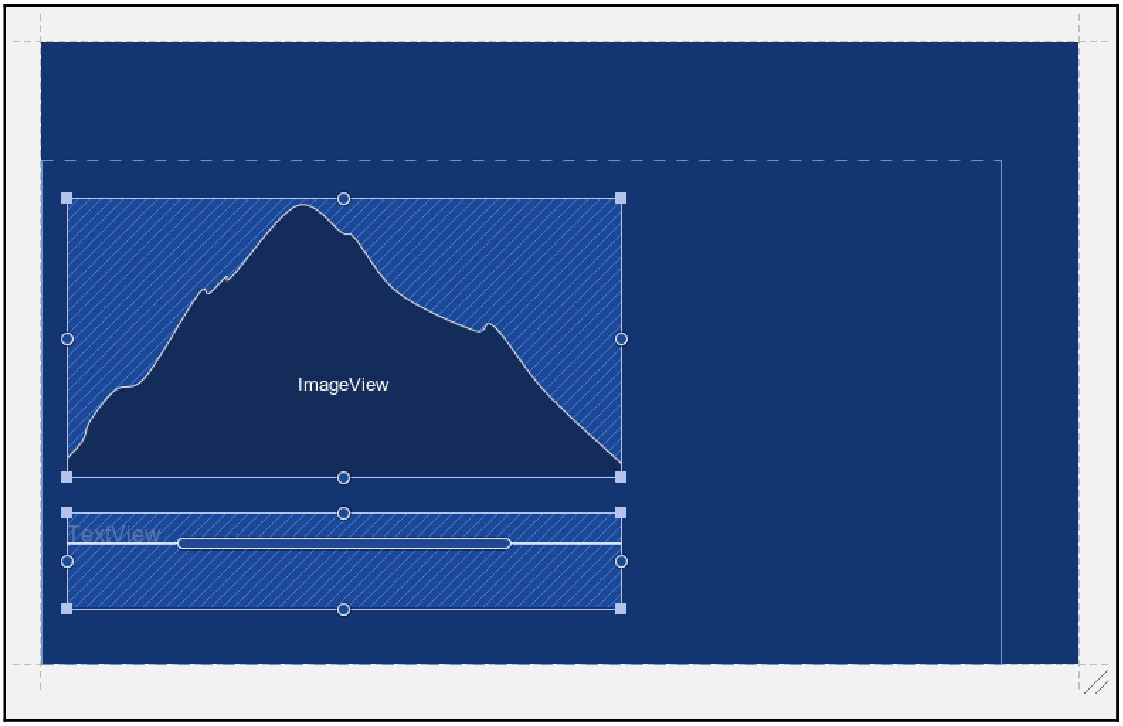
This will produce the XML shown here:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

</android.support.constraint.ConstraintLayout>
```

As with other layout types, the constraint layer provides mechanisms for positioning and aligning the views and widgets within it. This is done largely through the use of handles that can be positioned graphically to size and align our views.

Applying constraints

The best way to see how this works is to try it, as these few simple steps will demonstrate. Create a `ConstraintLayout` as described previously and drag and drop one or two views or widgets from the palette onto the blueprint screen, similar to following figure:



Constraint handles

Each view has constraint handles on its corners and its sides. Those on the corners are for simply resizing a view in the way that we are all familiar with. Those along the sides, however, are used to create constraints. These position views are relative to their parents or each other in a way that is not dissimilar to the relative layout.

As this is largely a graphical form of editing, it is best demonstrated by action. Drag the left side anchor point of one of your views toward the left side of the layout and release the mouse button as prompted to create a parent constraint. This being a layout that contains others will be a parent constraint.

As you experiment with constraints, you will notice how the margins automatically stick to those values recommended by the creative design guidelines.

If you now open the text editor, you will see the constraint realized like so:

```
app:layout_constraintLeft_toLeftOf="parent"
```

You will also notice from the code that an error is generated by this view. This is because we need both vertical and horizontal constraints for each view. This is achieved in exactly the same way, with something like:

```
app:layout_constraintTop_toTopOf="parent"
```

Constraints can be created between child views as well, using the same drag and drop technique or:

```
app:layout_constraintTop_toBottomOf="@+id/image_view"
```

Setting a constraint on all four sides of a view will center it in its container.

Constraints can be used to align sibling views as well as joining two adjacent sides, which generates code along these lines:

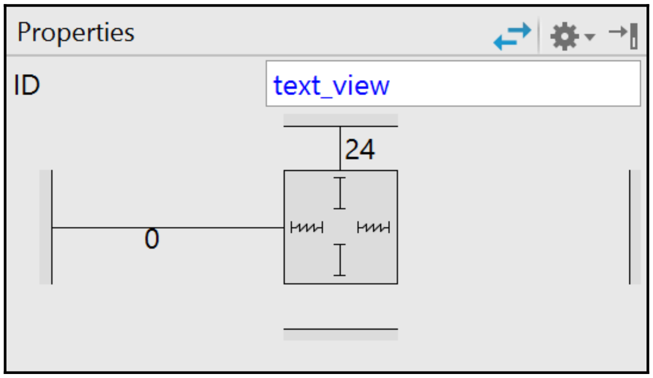
```
app:layout_constraintLeft_toLeftOf="@+id/image_view"
```

A constraint can be deleted very simply, in either editor mode, by clicking on its originating handle.

This drag and drop method is not peculiar to Android Studio, but there is one editing tool that is unique to Android Studio by providing a editable schematic perspective in the Properties tool.

Graphic Properties tool

You will no doubt have noticed the diagrammatic representation of a view that pops up in the **Properties** window, when a ConstraintLayout view is selected as follows:



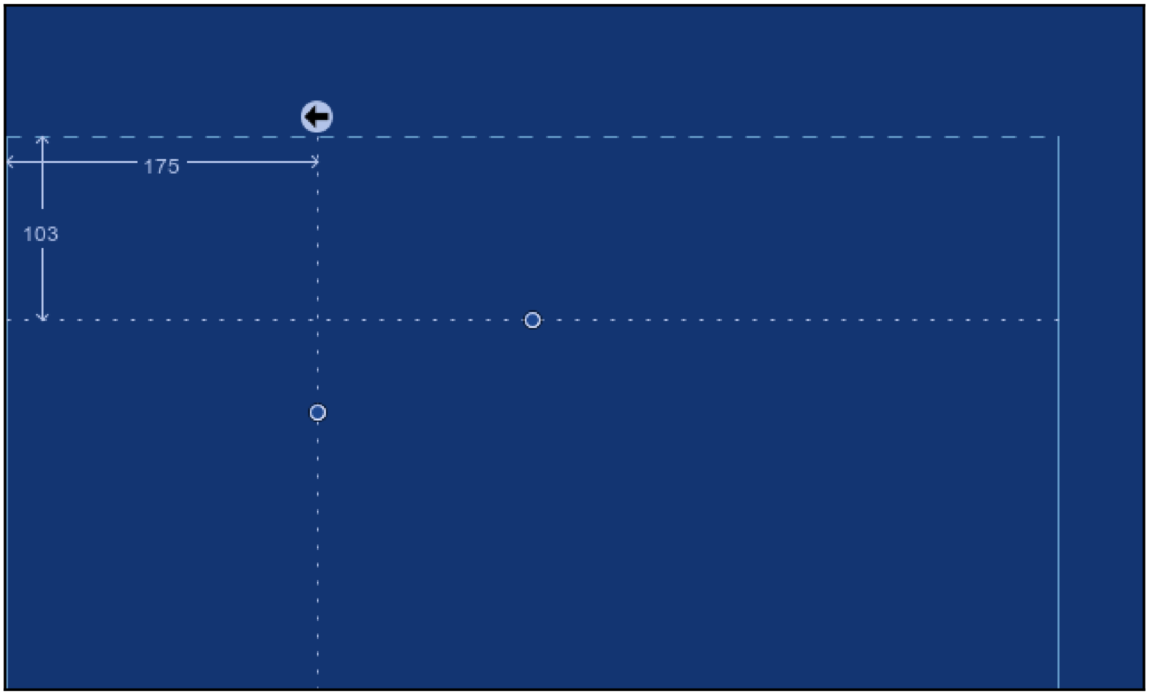
The Properties schematic.

This tool allows size and position properties to be edited with a single click and for the output to be instantly understood in a simple schematic form. It only takes seconds to learn and can speed up interface design considerably, particularly when experimenting with different layouts.

Inside the central square, representing our view, there are four lines, which when clicked on cycle through the following three states:

- **Unbroken line**: The view is an exact width, for example, `240dp`
- **Fuzzy line**: The view can be any size (dependent on bias), `match_parent`
- **Directed line**: The view matches its own content, `wrap_content`

Often, we will not want to constrain a view to the very edge of its container. For example, we may wish to divide the layout into two or more sections and have views organized within them. Guidelines allow us to divide our screens into sections and can be used exactly like parent edges. Take the following example:

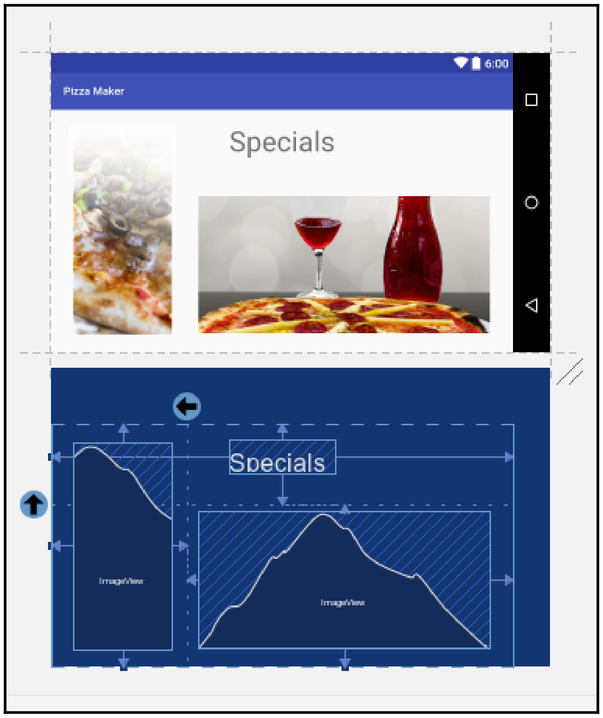


Constraint guidelines

Guidelines like this are most easily added from the constraint toolbar at the top of the design editor. Guidelines are added as XML components and look like this:

```
<android.support.constraint.Guideline
    android:id="@+id/gl_vertical"
    android:layout_width="wrap_content"
    android:layout_height="311dp"
    android:orientation="vertical"
    app:layout_constraintGuide_begin="175dp" />
```

We can now use these guidelines to center elements according to the whole layout or any of the four panes we have created, and all without nesting a single layout. In the following screenshot, we have a centered header and side panel with another view contained in a single pane, and we can of course apply bias to any of these sections:



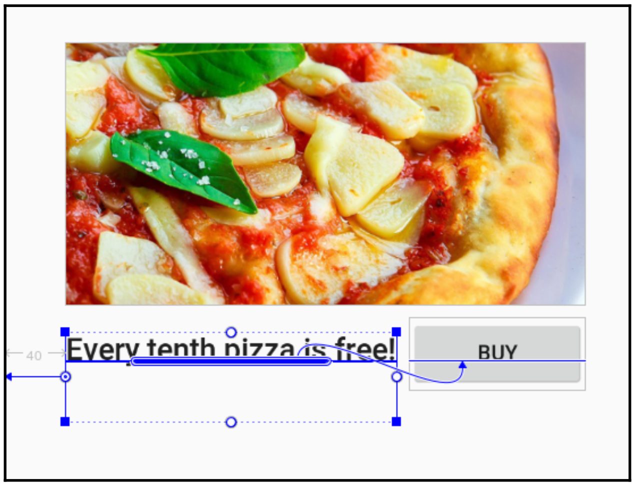
Applying constraint guidelines

If this system did not already provide enough advantages, there are more. Firstly, it proves very useful when aligning text as well as a more enhanced positioning technique referred to as bias, which performs a similar function to the weight attribute, but is better when it comes to designing for multiple screens. We will take a look first at text alignment constraints.

Baseline alignment

Aligning text across multiple views using their baselines can be fiddly, especially when text is of different sizes. Fortunately, constraint layouts offer a simple and easy way to achieve this.

Any constrained view or widget designed to contain text, will contain a bar across its center. Hover over this for a moment, until it flashes, and then drag it to the view whose text you wish to align it with, as seen here:



Baseline alignment.

You will probably be familiar with the gravity attributes that the relative layout class uses to control positioning.

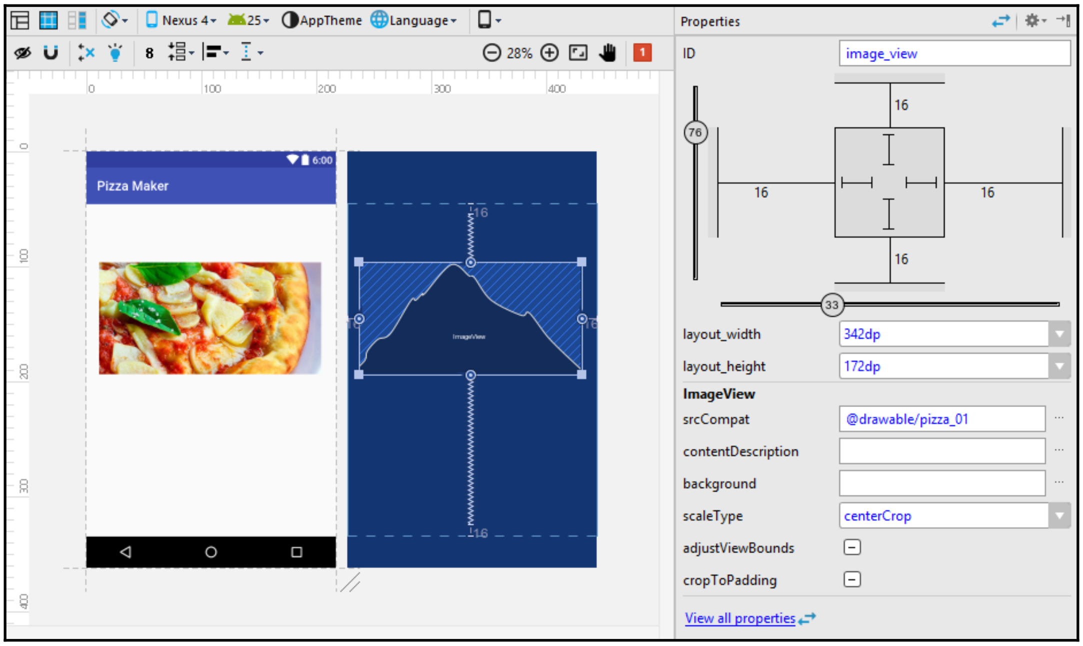
Baseline constraints can only be connected to other baselines.

The constraint layout introduces a new approach, allowing us to control the relative distances either side of the view.

Controlling position with bias

Bias can be best understood here as a percentage value, but rather than position the view according to its center or a corner, it is the percentage of space either side of it. So if the bias towards the top is 33%, then the margin below will be twice the size of the one below it.

Setting bias is even easier than understanding it, as once a constraint is set on any opposing sides of a view, an associated slider will appear in the Properties graphic Properties editor:



Properties

ID image_view

layout_width 342dp

layout_height 172dp

ImageView

srcCompat @drawable/pizza_01

contentDescription

background

scaleType centerCrop

adjustViewBounds

cropToPadding

Applying bias with the GUI

A quick glance at the generated code shows the format for this attribute as follows:

```
app:layout_constraintHorizontal_bias="0.33"
```

The value of using bias to position screen elements lies partly in the simplistic approach, but its true value comes when it comes to developing for multiple screens. There are so many models available and they all seem to have slightly different proportions. This can make design layouts that look fantastic on all of them, very time consuming and even shapes as similar as 720 x 1280 and 768 x 1280 can produce undesirable results when tested with the same layout. The use of the bias attribute goes a long way to solving these issues, and we shall see more of this later when we take a look at layout previews and the percent library.

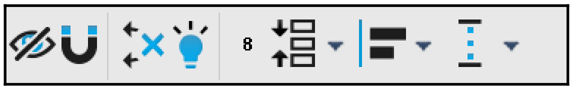
TIP

The **Design** and **Text** modes of the editor can be switched between using *Alt* + Left or Right.

As if all this didn't make designing a layout easy enough, constraint layouts have two other extremely handy functions that almost automate UI design: auto-connect and inference.

The constraints toolbar

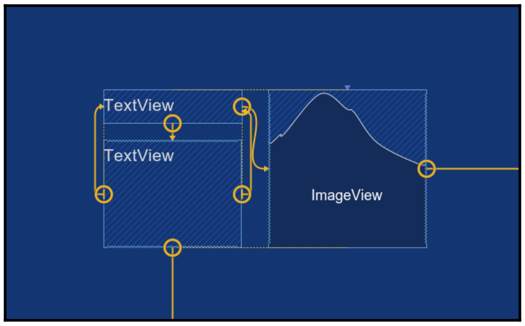
Although we will always want to spend a lot of time perfecting our final designs, a lot of the development cycle will be taken up with experimentation and trying out new ideas. We want to test these individual designs as quickly as possible, and this is where auto-connect and inference come in. These functions can be accessed through the constraints toolbar, which contains other useful tools and is worth looking at in detail.



The constraints toolbar

From left to right, the toolbar breaks down as follows.

- **Show Constraints**: Displays all constraints, not just those of the selected components.
- **Auto-connect**: When this is enabled, new views and widgets will have their constraints set automatically according to where they are placed.
- **Clear All Constraints**: As its title suggests, a one click solution to starting again. This can cause some unexpected results, so it should be used with a little caution.
- **Infer Constraints**: Apply this once you have designed your layout. It will automatically apply constraint in a similar fashion to auto-connect, but it will do so to all views in a single pass.



The infer process

- **Default Margins**: Sets the margins for the whole layout.
- **Pack**: This offers a series of distribution patterns, which help to evenly expand or shrink the area used by the selected items.
- **Align**: This drop-down offers the most commonly used group alignment options.
- **Guidelines:** Allows the quick insertion of guidelines.

Both auto-connect and infer offer intelligent and fast methods of building constraint layouts, and although they make fantastic tools for testing out ideas, they are far from perfect. Very often these automations will include unnecessary constraints that will need removing. Also, if you examine the XML after you have employed these techniques, you will notice that some values are hardcoded, and as you will know this is somewhat less than best practice.

As you will have hopefully have seen in this section, Android Studio and the ConstraintLayout are literally made for each other. This is not to say that it should replace the linear and relative layouts in all cases. When it comes to simple lists, the linear layout is still the most efficient. And for layouts with only two or three children, a relative layout is often cheaper too.

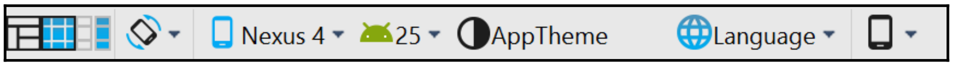
There is still more to the `ConstraintLayout` class, such as distribution chaining and runtime constraint modification, and we will return to the subject frequently throughout the book, but for now we will take a look at another of Android Studio's unique and powerful tools, device preview and emulation.

Multiple screen previewing

One of the most interesting challenges an Android developer faces is the bewildering number of devices that employ it. Everything from wristwatches to widescreen televisions. It is rare that we would want to develop a single application to run across such a range, but even developing layouts for all mobile phones is still a daunting task.

Fortunately, this process is aided by the way the SDK allows us to categorize features such as screen shape, size, and density into broader groups. Android Studio adds another powerful UI development tool, in the form of a complex preview system. This can be used to preview many popular device configurations as well as allowing us to create custom configurations.

In the previous section, we took a look at the **ConstraintLayout** toolbar, but as you will have noticed, there is a more generic design editor toolbar:



Design editor toolbar

Most of these tools are self explanatory, and you will have used many of them. One or two, however, are worth taking a closer look at, especially if you are new to Android Studio.

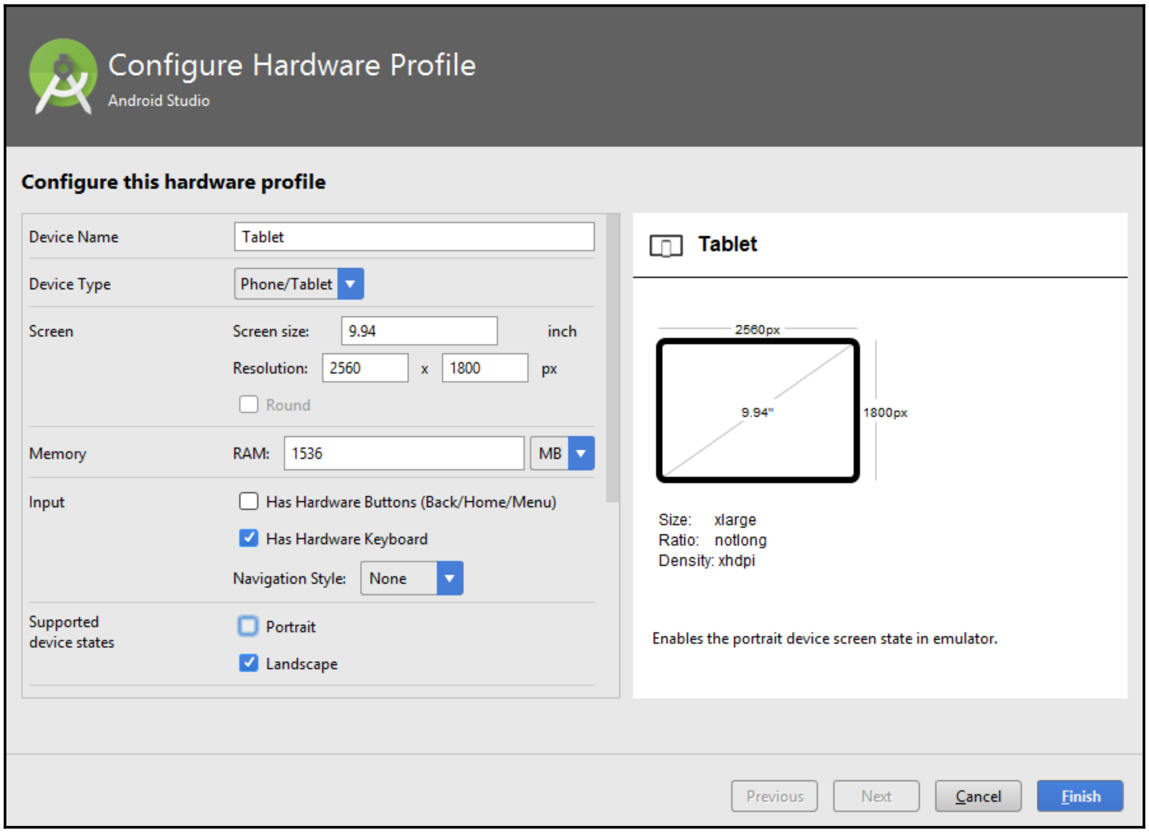
By far, one of the most useful design tools available to us is the **Device in Editor** tool, displaying in the preceding figure as **Nexus 4**. This allows us to preview our layouts as they would appear in any number of devices, without having to compile the project. The drop-down provides a selection of generic and real-world profiles, any AVD we might have created, and the option to add our own device definition. It is this option that we shall look at now.

Hardware profiles

Selecting **Add Device Definition...** from the **Device in Editor** drop-down will open the AVD manager. To create a new hardware profile click on the **Create Virtual Device...** button. The **Select Hardware** dialog allows us to install and edit all the device profiles listed in the previous drop-down as well as the option to create or import a definition.

A standalone version of the AVD manager can be run from `user\AppData\Local\Android\sdk\`. This can be useful on lower end machines, as AVDs can be booted up without Studio running.

It is usually easier to take an existing definition and adapt it to our needs, but to gain further insight into the operation, here we will create one from scratch by clicking on the **New Hardware Profile** button from the **Select Hardware** dialog. This will take you to the **Configure Hardware Profile** dialog where you can select hardware emulators such as cameras and sensors as well as defining internal and external storage options.


Configure Hardware Profile
Android Studio

Configure this hardware profile

Device Name	Tablet
Device Type	Phone/Tablet
Screen	Screen size: 9.94 inch
	Resolution: 2560 x 1800 px
	☐ Round
Memory	RAM: 1536 MB
Input	☐ Has Hardware Buttons (Back/Home/Menu)
	✓ Has Hardware Keyboard
	Navigation Style: None
Supported device states	☐ Portrait
	✓ Landscape

Tablet

Size: xlarge
Ratio: notlong
Density: xhdpi

Enables the portrait device screen state in emulator.

Hardware configuration

Once you have your profile and click on **Finish**, you will be returned to the hardware selection screen where your profile will now have been added to the list. Before moving on, however, we should take a quick look at how we can emulate storage hardware.

Virtual storage

Each profile contains an SD card disk image to emulate external storage, and obviously this is a useful feature. However, it would be even more useful if we could remove these cards and share them with other devices. Fortunately, Android Studio has some very handy command-line tools, which we will be encountering throughout this book. The command that interests us here though is `mksdcard`.

The `mksdcard` executable can be found in `sdk/tools/` and the format for creating a virtual SD card is:

```
mksdcard <label> <size> <file name>
```

For example:

```
mksdcard -l sharedSdCard 1024M sharedSdCard.img
```

When testing an application on a large number of virtual devices, it can save a lot of time to be able to share external memory, and of course such images can be stored on actual SD cards, which not only makes them more portable, but can reduce the load on a hard drive.

Our profile is now ready to be combined with a system image to form an AVD, but first we will export it to get a better look at how it is put together. This is saved as an XML file and can be achieved by right-clicking on your profile in the main table of the hardware selection screen. Not only does this provide an insight as well as a handy way to share devices across networks, but it is also very quick and simple to edit itself.

The configuration itself can be quite long, so the following is a sample node to provide an idea:

```
<d:screen>
    <d:screen-size>xlarge</d:screen-size>
    <d:diagonal-length>9.94</d:diagonal-length>
    <d:pixel-density>xhdpi</d:pixel-density>
    <d:screen-ratio>notlong</d:screen-ratio>
    <d:dimensions>
        <d:x-dimension>2560</d:x-dimension>
        <d:y-dimension>1800</d:y-dimension>
    </d:dimensions>
    <d:xdpi>314.84</d:xdpi>
    <d:ydpi>314.84</d:ydpi>
    <d:touch>
        <d:multitouch>jazz-hands</d:multitouch>
        <d:mechanism>finger</d:mechanism>
        <d:screen-type>capacitive</d:screen-type>
    </d:touch>
    </d:screen>
```

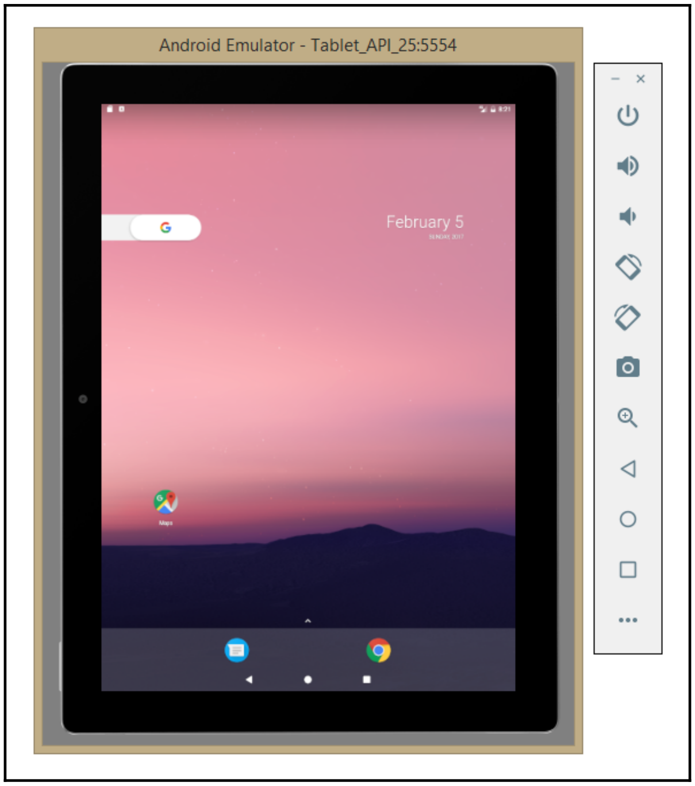
Looking at the way a screen is defined here, provides a useful window into the features and definitions we need to consider when developing for multiple devices.

To see our profile in action we need to connect it to a system image and run it on the emulator. This is done by selecting the profile and clicking on **Next**.

TIP

To test an app thoroughly, it is usually best to create an AVD for each API level, screen density, and hardware configuration you intend to publish the app on.

Having selected an image, you will be given the opportunity to tweak your hardware profile and then create the AVD:



An Android AVD

Emulating the latest mobile devices is an impressive task, even for the toughest of computers and, even with HAXM hardware acceleration, it can be frustratingly slow, although the addition of Instant Run has considerably speeded this process up. There is little in the way of alternatives, with the exception of Genymotion, which provides faster virtual devices and several features unavailable on the native emulator. These include the drag and drop installation, real-time window resizing, working network connection, and one click mock location setting. The only drawbacks being that there are no system images for Android Wear, TV, or Auto and it is only free for personal use.

This section shows how we can preview our layouts across a large number of form factors and how we can build a virtual device to match the exact specifications of any target device, but this is only part of the story. In the next chapter, we will see how to create layout files for all our target devices.

Summary

In this chapter, we covered the fundamentals of interface development, and this is largely a matter of using and understanding the various layout types. A lot of the chapter was devoted to the constraint layout as this is the latest and most flexible of these viewgroups and is catered for fully in Android Studio with intuitive visual tools.

The chapter concluded by seeing how we take the completed layouts and view them on an emulator using a customized hardware profile.

In the following chapter, we will look more deeply into these layouts and see how the coordinator layout is used to coordinate a number of child components to work together with very little coding required from us.

3
UI Development

In the previous chapter, we saw how Android Studio provides many invaluable tools for designing layouts quickly and simply. However, we only concerned ourselves with the design of static UIs. This, of course, is an essential first step, but our interfaces can, and should, be dynamic. And, according to material design guidelines, user interactions should be illustrated graphically using movement and color to intuitively demonstrate the action being performed, such as the ripple animations that result from tapping on a button.

To see how this is done, we need to look at a practical example and start building a simple, but functional, application. But first, we will examine one or two more ways of applying the look and feel we want, and Android users expect, to our designs. This process is largely assisted by the use of support libraries, in particular, the AppCompat and Design libraries.

We will begin the chapter by looking at how Android Studio facilitates the implementation of Material design with the material-based visual theme editor as well as the design support library.

In this chapter, you will learn how to do the following:

- Generate material styles and themes
- Use XML fonts
- Create XML font families
- Use basic code completion
- Apply a coordinator layout
- Coordinate design components
- Create a collapsing app bar
- Deploy raw resources
- Use the percent support library

We saw in the previous chapter that, when sizing and moving the screen elements of a constraint layout using the design editor, our views tend to stick to a particular set of dimensions. These are selected according to Material design guidelines. In case you are unaware, Material is a design language, stipulated by Google and based on traditional design and animation techniques, intended to clean up user interfaces by communicating the process through movement and position.

Material design

Although Material design is by no means essential, and can be ignored entirely if you are developing full-screen apps such as games, which often come with their own design rules, it is nevertheless an elegant design paradigm and is widely recognized and understood by the user base.

One very good reason for implementing Material is that many of its features, such as card views and sliding drawers, can be applied with great ease, thanks to the associated support libraries.

One of the first design decisions we need to take is what color scheme, or theme, we want to apply to our app. There are one or two material guidelines regarding the shade and contrast of our themes. Fortunately, Android Studio's Theme Editor makes generating material-compliant themes very simple indeed.

Android styles

Graphical properties, such as background color, text size, and elevation, can all be set individually on any UI component. It often makes sense to group properties together into a style. Android stores such styles in the values directory as XML in the `styles.xml` file. An example of this is as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <style name="TextStyle" parent="TextAppearance.AppCompat">
        <item name="android:textColor">#8000</item>
        <item name="android:textSize">48sp</item>
    </style>
</resources>
```

Styles, such as this, can be applied to views and widgets simply and without having to specify each property, as follows:

```
<TextView
    . . .
    android:textAppearance="@style/TextStyle"
    . . . />
```

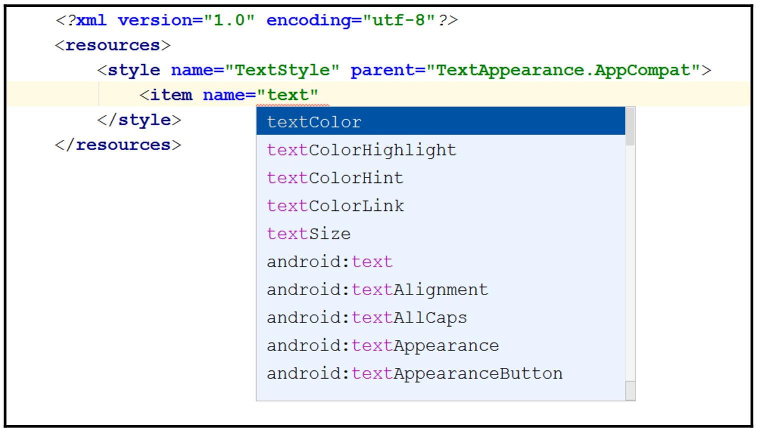
It is quite possible to create any style from scratch by defining all its properties, but it is far more practical to take an existing style and modify only the properties we wish to change. This is done by setting the `parent` property, which can be seen in the preceding example.

We can also inherit from our own styles and without having to set a parent property, for example:

```
<style name="TextStyle.Face">
    <item name="android:typeface">monospace</item>
</style>
```

Previously, we created a new resource file, but we could just as easily add a new `<style>` node to the existing `styles.xml` file.

If you are new to Android Studio, you will have noticed the code completion drop-downs that appear as you are typing, as shown in the following screenshot:



```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <style name="TextStyle" parent="TextAppearance.AppCompat">
        <item name="text"
    </style>
</resources>
```

```
textColor
textColorHighlight
textColorHint
textColorLink
textSize
android:text
android:textAlignment
android:textAllCaps
android:textAppearance
android:textAppearanceButton
```

Code completion

This is an invaluable tool and we will look at it in more detail later on. For now, it is useful to know that code completion exists in three levels, as outlined briefly here:

- **Basic**: *Ctrl* + Space; Displays possibilities for the next word.
- **Smart**: *Ctrl* + *Shift* + Space; Context-sensitive suggestions.
- **Statement**: *Ctrl* + *Shift* + *Enter*; Completes whole statements.

Calling basic and smart code completion twice in a row will broaden the scope of suggestions.

Creating and applying styles like this is a great way of fine-tuning an app's appearance without having to do a great deal of extra coding, but there are also times when we want to apply a look and feel to an entire application, and for this, we use themes.

Material themes

When creating an overall theme for an app, we have two opposing goals. On the one hand, we want our app to stand out from the others and be easily recognized; on the other, we want it to comply with the user's expectations of the platform and we want them to find the controls familiar and simple to use. The Theme Editor strikes a good compromise between individuality and conformity.

At their simplest, material themes take two or three colors and apply these throughout an application to give it a consistent feel and this is probably the major benefit of using themes. The color selected as an accent will be used to color checkboxes and highlight text and is generally chosen to stand out and draw attention. Primary colors, on the other hand, will be applied to toolbars and, unlike the earlier versions of Android, to the status bar and navigation bar. For example:

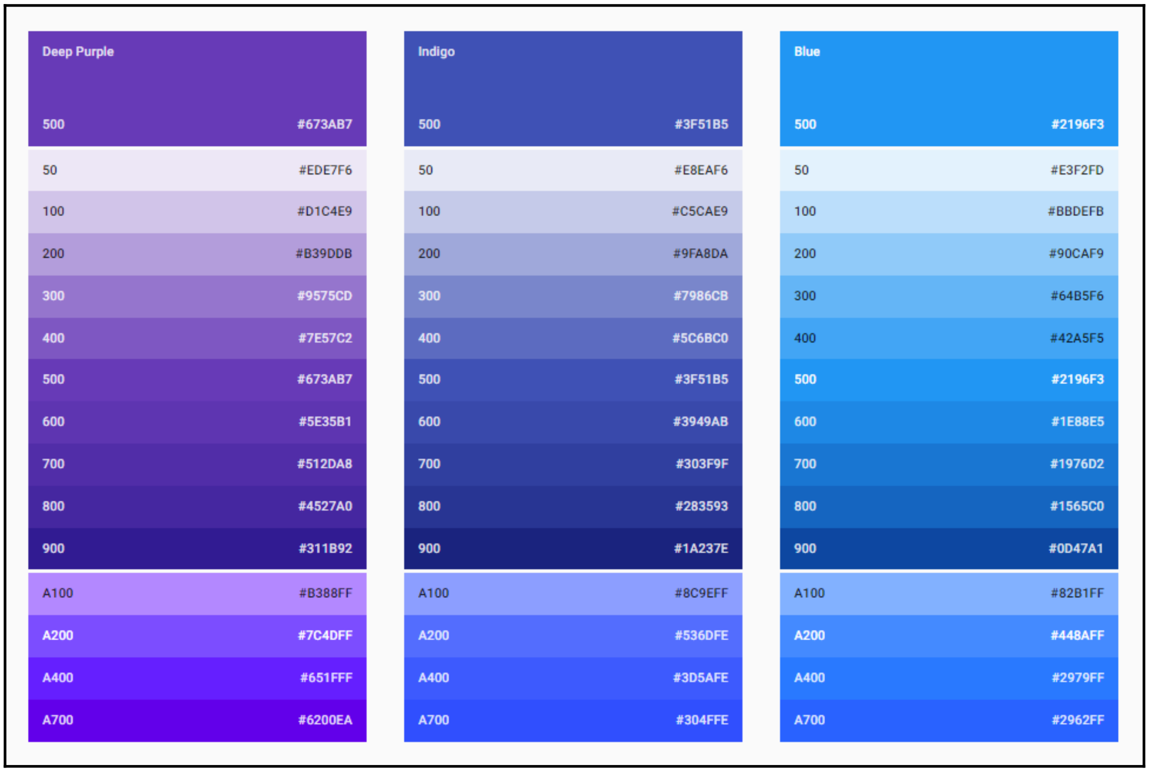
```
<color name="colorPrimary">#ffc107</color>
<color name="colorPrimaryDark">#ffa000</color>
<color name="colorAccent">#80d8ff</color>
```

This enables us to control the color scheme of the entire screen when our app is running, avoiding ugly clashes with any native controls.

The general rule of thumb for selecting these colors is to pick two shades of the same color for the primary values and a contrasting, but complementary, color for the accent. Google is a little more precise about which shades and which colors to use and there are no hard and fast rules to decide which colors contrast nicely with others. There are, however, some useful guidelines to help us, but first we will take a look at Google's material palette.

The Theme Editor

Google prescribes a series of 20 different hues to be used in Android apps and Material design web pages. Each hue comes in ten shades, as seen in the following examples:



Deep Purple		Indigo		Blue	
500	#673AB7	500	#3F51B5	500	#2196F3
50	#EDE7F6	50	#E8EAF6	50	#E3F2FD
100	#D1C4E9	100	#C5CAE9	100	#BBDEFB
200	#B39DDB	200	#9FA8DA	200	#90CAF9
300	#9575CD	300	#7986CB	300	#64B5F6
400	#7E57C2	400	#5C6BC0	400	#42A5F5
500	#673AB7	500	#3F51B5	500	#2196F3
600	#5E35B1	600	#3949AB	600	#1E88E5
700	#512DA8	700	#303F9F	700	#1976D2
800	#4527A0	800	#283593	800	#1565C0
900	#311B92	900	#1A237E	900	#0D47A1
A100	#B388FF	A100	#8C9EFF	A100	#82B1FF
A200	#7C4DFF	A200	#536DFE	A200	#448AFF
A400	#651FFF	A400	#3D5AFE	A400	#2979FF
A700	#6200EA	A700	#304FFE	A700	#2962FF

Material palettes

The full palette along with downloadable swatches can be found at: `material.io/guidelines/style/color.html#color-color-palette`.

Material guidelines recommend that we use shades of **500** and **700** for our primary and dark primary colors and 100 for the accent. Fortunately, we do not have to concern ourselves overly with these numbers, as there are tools to help us.

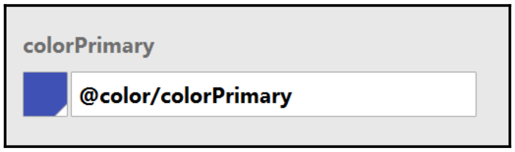
The most useful of these tools is the Theme Editor. This is another graphical editor and can be accessed from the main menu with **Tools | Android | Theme Editor**.

Once you open the Theme Editor, you will see that it is divided into two sections. To the right, we have a list of color properties, and to the left, a panel displaying the effect these selections have on various interface components, giving us a handy preview and the opportunity to try out all the manner of combinations quickly and intuitively.

As you can see, there are more than just two primary colors and an accent. There are in fact 12, covering text and background colors, as well as alternatives for both dark and light themes. These are set, by default, to those of the parent theme declared in the `styles.xml` file.

To quickly set up a customized material theme, follow the steps:

1. Start a new Studio project or open one that you wish to apply a theme to.
2. Open the Theme Editor from the **Tools | Android** menu.
3. Select the solid color block to the left of the **colorPrimary** field.


colorPrimary

@color/colorPrimary

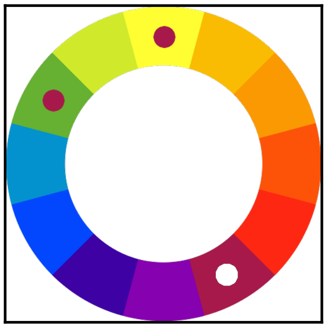
4. Select one of the solid color blocks in the lower right corner of the **Resources** dialog and click **OK**.
5. Open the dialog for **colorPrimaryDark** and select the only suggested block on the right-hand side below the color selection options. It will be the same hue with a shade of 700.
6. Select the **Accent** property and then choose from one of the suggested colors.

These choices can be immediately seen in the preview pane on the left-hand side of the editor and also from the Layout Editor.

As you can see, these colors are not declared directly but rather are references to the values specified in the `values/colors.xml` file.

Not only does the editor help create material themes by suggesting acceptable colors, it also helps if we want to select a color of our own choice. Clicking anywhere on the color table in the **Select Resource** window will produce a prompt to select **closest material color**.

When it comes to selecting appropriate colors for the accent, there are several schools of thought. According to color theory, there are several ways to create a harmonic complement to any color using a color wheel, such as the following one:


A RYB color wheel displaying harmonic complements

The simplest way to calculate an harmonic color is to take the one opposite to ours on the wheel (known as the direct complement). However, those of artistic vision consider this somewhat obvious and lacking in subtlety, and prefer what is known as a split complement. This equates to selecting from those colors that closely neighbor the direct complement, as shown previously.

The Theme Editor suggests several split complements beneath the color selector when the accent color is being selected. However, it also suggests analogous harmonies. These are colors that lie close to the original, and although they look nice together, these are not good choices for an accent, as there is little contrast and the user may well miss important hints.

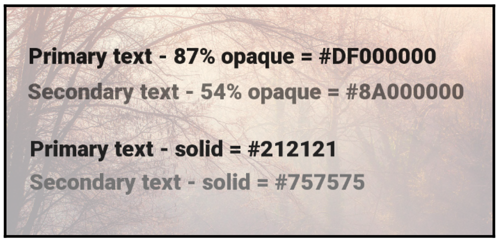
There is a very pleasing JetBrains plugin available that will apply a material theme to the IDE itself. It can be found at: `plugins.jetbrains.com/androidstudio/plugin/8006-material-theme-ui`.

As we have just seen, the Theme Editor is very helpful when it comes to generating material themes. There is also a growing number of online tools that generate complete XML material themes with a few clicks. MaterialUps can be found at: `www.materialpalette.com`.

This will produce the following `colors.xml` file:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <color name="primary">#673AB7</color>
    <color name="primary_dark">#512DA8</color>
    <color name="primary_light">#D1C4E9</color>
    <color name="accent">#FFEB3B</color>
    <color name="primary_text">#212121</color>
    <color name="secondary_text">#757575</color>
    <color name="icons">#FFFFFF</color>
    <color name="divider">#BDBDBD</color>
</resources>
```

At first sight, this looks like a quick way to select many of a theme's properties, but if you look at the text colors, you will see that they are shades of gray. According to Material design guidelines, this is incorrect, and the alpha channel should be used to create shades using transparency. This makes no difference when the text is on a plain background, but when placed over imagery grayscale text can be harder to read particularly the lighter shades, as demonstrated here:


Primary text - 87% opaque = #DF000000
Secondary text - 54% opaque = #8A000000
Primary text - solid = #212121
Secondary text - solid = #757575

Grayscale vs transparency

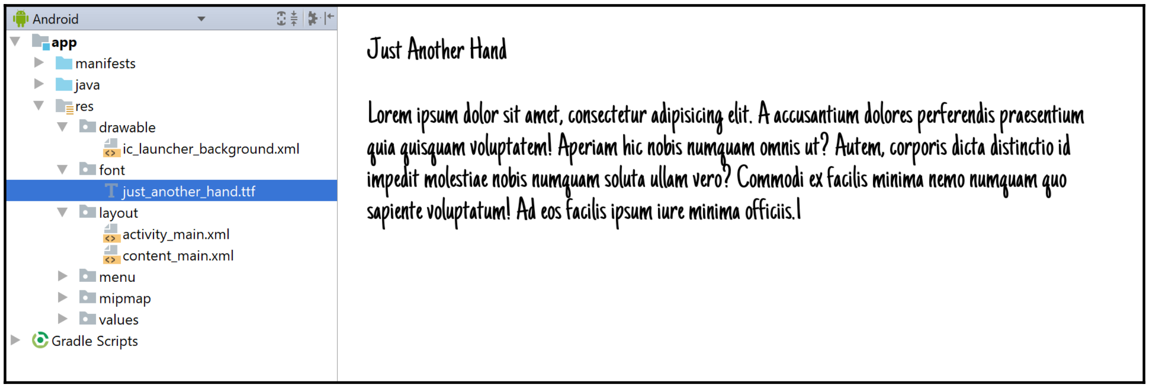
Android themes allow us to define the appearance of our applications in terms of colors, but often we will want to do more than just customize the color of our text, and being able to include fonts in a similar way to other resources is a recent and extremely useful addition.

XML fonts

As of API level 26, it has been possible to include fonts in a project as XML resources in the `res` directory. This feature has simplified the task of using nondefault fonts in an application as well as bringing the process into line with other resource management.

Adding XML fonts is remarkably simple, as the following exercise demonstrates:

1. Right-click on the `res` directory and select **New** | **Android resource directory**.
2. Select **font** from the **Resource type** drop-down and click on **OK**.
3. Right-click on the newly created **font** folder and select **Show in Explorer**.
4. Rename your font files so that they only contain permissible characters. For example, `times_new_roman.ttf` not `TimesNewRoman.ttf`.
5. Place your selected fonts in the font directory.
6. These can now be previewed directly from the editor.



Just Another Hand

Lorem ipsum dolor sit amet, consectetur adipisicing elit. A accusantium dolores perferendis praesentium quia quisquam voluptatem! Aperiam hic nobis numquam omnis ut? Autem, corporis dicta distinctio id impedit molestiae nobis numquam soluta ullam vero? Commodi ex facilis minima nemo numquam quo sapiente voluptatum! Ad eos facilis ipsum iure minima officiis.

XML fonts.

Using these fonts in a layout is even simpler than adding them as resources. Simply use the `fontFamily` attribute, as follows:

```
<TextView
        . . .
        android:fontFamily="@font/just_another_hand"
        . . . />
```

When working with fonts, it is common to want to emphasize words in various ways, such as using a bolder typeface or italicizing text. Rather than relying on a separate font for each of these versions, it is more convenient to be able to refer to font groups, or families. Simply right-click on your `font` folder and select **New** | **Font resource file**. This will create an empty font family file, which can then be filled out along the following lines:

```
<?xml version="1.0" encoding="utf-8"?>
<font-family xmlns:android="http://schemas.android.com/apk/res/android">
    <font
        android:fontStyle="bold"
```

```
            android:fontWeight="400"
            android:font="@font/some_font_bold" />

        <font
            android:fontStyle="italic"
            android:fontWeight="400"
            android:font="@font/some_font_italic" />
    </font-family>
```

There is, of course, a lot more to a design language than selecting the right colors and fonts. There are conventions regarding spacing and proportion and, usually, a number of specially designed screen components. In the case of material, these components take the form of widgets and layouts, such as FABs and sliding drawers. These do not come as part of the native SDK but are included in the design support library.

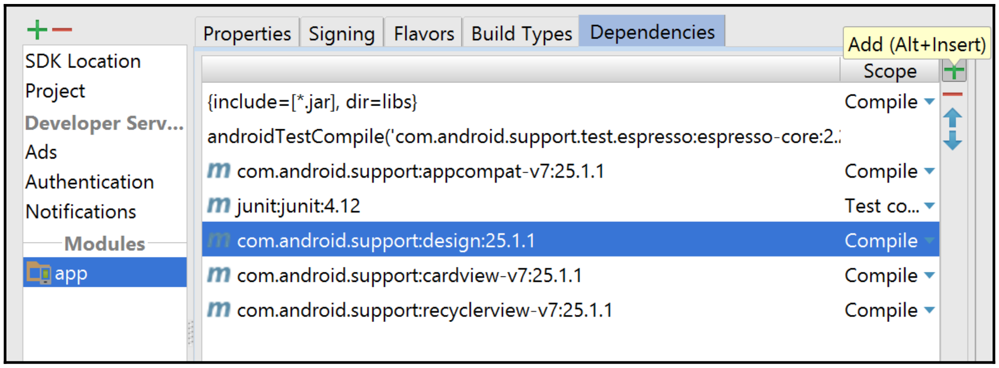
The design library

As mentioned, the design support library provides widgets and views commonly found in material apps.

As you will know, the design library, like other support libraries, needs to be included as a gradle dependency in the module level `build.gradle` file, as follows:

```
dependencies {
    compile fileTree(include: ['*.jar'], dir: 'libs')
    androidTestCompile('com.android.support.test.espresso:espresso-
      core:2.2.2', {
          exclude group: 'com.android.support', module: 'support-
                annotations'
    })
    compile 'com.android.support:appcompat-v7:25.1.1'
    testCompile 'junit:junit:4.12'
    compile 'com.android.support:design:25.1.1'
}
```

Although it is always useful to understand how things are done, there is, in fact, a great shortcut for adding a support library as a project dependency. Open the **Project Structure** dialog from the **File** menu, and select your module and the **Dependencies** tab.


SDK Location
Project
Developer Serv...
Ads
Authentication
Notifications
Modules
app

Properties | Signing | Flavors | Build Types | Dependencies

Add (Alt+Insert)

	Scope
{include=[*.jar], dir=libs}	Compile
androidTestCompile('com.android.support.test.espresso:espresso-core:2.:	
com.android.support:appcompat-v7:25.1.1	Compile
junit:junit:4.12	Test co...
com.android.support:design:25.1.1	Compile
com.android.support:cardview-v7:25.1.1	Compile
com.android.support:recyclerview-v7:25.1.1	Compile

The project structure dialog

You can select the library you are after by clicking on the **Add** icon in the top right corner and selecting **Library dependency** from the drop-down.

The project structure dialog can be summoned with the *Ctrl* + *Alt* + *Shift* + *S* keys.

There are two other advantages to using this method. Firstly, the IDE will automatically rebuild the project, and secondly, it will always import the most recent revision.

Many developers preempt future revisions by using a plus sign, as follows: `compile 'com.android.support:design:25.1.+'`. This has the effect of applying future minor revisions. However, it is not always guaranteed to work and can cause crashes, and it is wiser to keep versions up-to-date manually, even if this means publishing more updates.

As well as importing the design library, if you are planning on developing a material app, you will most likely want the `CardView` and `RecyclerView` libraries as well.

The best way to find your way around the IDE is to work through a practical example. Here, we will put together a simple weather app. It won't be at all complex, but it will take us through each stage of application development and it will adhere to the Material design guidelines.

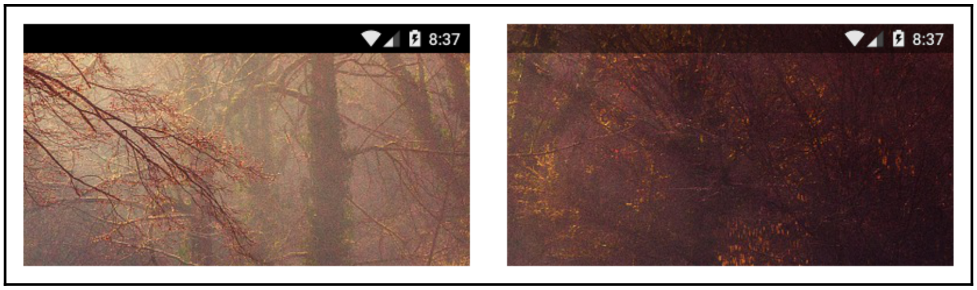
The coordinator layout

The design library provides three layout classes. There is one for designing tabular activities and one for toolbars, but the most important layout is `CoordinatorLayout`, which acts as a material-aware container, automatically performing many material tricks, such as expanding headers when a user scrolls to the top of a list or ensuring a FABs slides out of the way when a snack bar pops up over it.

The coordinator layout should be placed as the root layout of an activity and would typically look like the following lines:

```
<android.support.design.widget.CoordinatorLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/coordinator_layout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:fitsSystemWindows="true">

    . . .

</android.support.design.widget.CoordinatorLayout>
```

The property `fitsSystemWindows` is particularly useful, as it sets the status bar to partially transparent. This allows our design to dominate native controls without hiding them completely as well as avoiding any clashes with system colors.



Drawing behind the status bar

It is also possible to combine `fitsSystemWindows` with our own choice of color using `colorPrimaryDark` to assign the status bar color.

The navigation bar's color can also be changed with the `navigationBarColor` attribute but this is not advisable, as devices with soft navigation controls are becoming rarer.

CoordinatorLayout is very similar to FrameLayout with one important exception. The coordinator layout can take direct control of its children using the CoordinatorLayout.Behavior class. The best way to see how this works is with an example.

Snackbars and Floating Action Buttons

Snackbars and **Floating Action Buttons**(**FABs**) are two of the most recognizable material widgets. Although not entirely replacing the toast widget, snackbars offer a more sophisticated form of activity notification, allowing controls and media rather than just text, which was the case for toasts. FABs perform the same function as conventional buttons but use their position to indicate their function.

Without a coordinator layout to control behavior, Snackbar rising from the bottom of the screen would obscure any views or widgets behind it. It would be far preferable if the widget were to slide gracefully out of the way, something you will have seen often in well designed material apps. The following exercise explains how this is done:

1. Start a new project in Android Studio.
2. Replace the root layout of the main activity with CoordinatorLayout here:

```
<android.support.design.widget.CoordinatorLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/coordinator_layout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:fitsSystemWindows="true">
```

3. Add the following button:

```
<Button
    android:id="@+id/button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="top|start"
    android:layout_marginStart=
            "@dimen/activity_horizontal_margin"
    android:layout_marginTop=
            "@dimen/activity_vertical_margin"
    android:text="Download" />
```

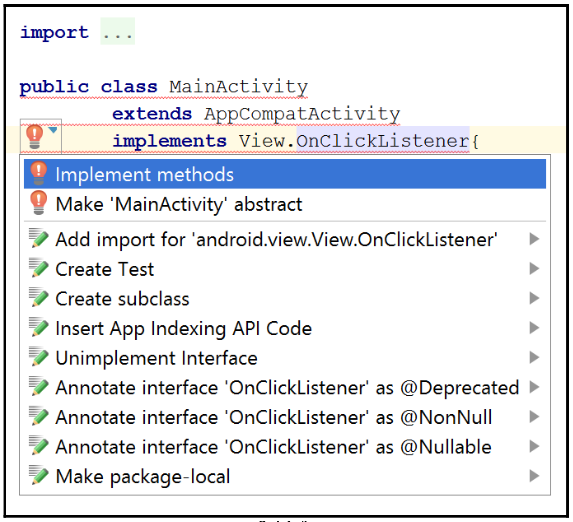
4. This is followed by `Snackbar`:

```
<android.support.design.widget.FloatingActionButton
    android:id="@+id/fab"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="bottom|end"
    android:layout_marginBottom=
            "@dimen/activity_vertical_margin"
    android:layout_marginEnd=
            "@dimen/activity_horizontal_margin"
    app:srcCompat="@android:drawable/stat_sys_download" />
```

5. Open the main activity Java file and extend the class declaration to implement a click listener, as follows:

```
public class MainActivity
    extends AppCompatActivity
    implements View.OnClickListener
```

6. This will generate an error and a red light bulb (known as a quick fix) will appear.



```
import ...

public class MainActivity
        extends AppCompatActivity
        implements View.OnClickListener{
```

Implement methods
Make 'MainActivity' abstract
Add import for 'android.view.View.OnClickListener'
Create Test
Create subclass
Insert App Indexing API Code
Unimplement Interface
Annotate interface 'OnClickListener' as @Deprecated
Annotate interface 'OnClickListener' as @NonNull
Annotate interface 'OnClickListener' as @Nullable
Make package-local

Quick fix

7. Select **Implement methods** to add `OnClickListener`.

8. Add the following fields to the class:

```
private Button button;
private CoordinatorLayout coordinatorLayout;
```

9. Create references for these components in the `onCreate()` method:

```
coordinatorLayout = (CoordinatorLayout)
    findViewById(R.id.coordinator_layout);
button = (Button)
    findViewById(R.id.button);
```

10. Associate the button with the listener, as follows:

```
button.setOnClickListener(this);
```

11. Then complete the listener method, as follows:

```
@Override
public void onClick(View v) {
    Snackbar.make(coordinatorLayout,
            "Download complete",
            Snackbar.LENGTH_LONG).show();
    }
}
```

This code can now be tested on an emulator or a real device. Clicking the button will display `Snackbar` temporarily, sliding the FAB out of the way as it does so.

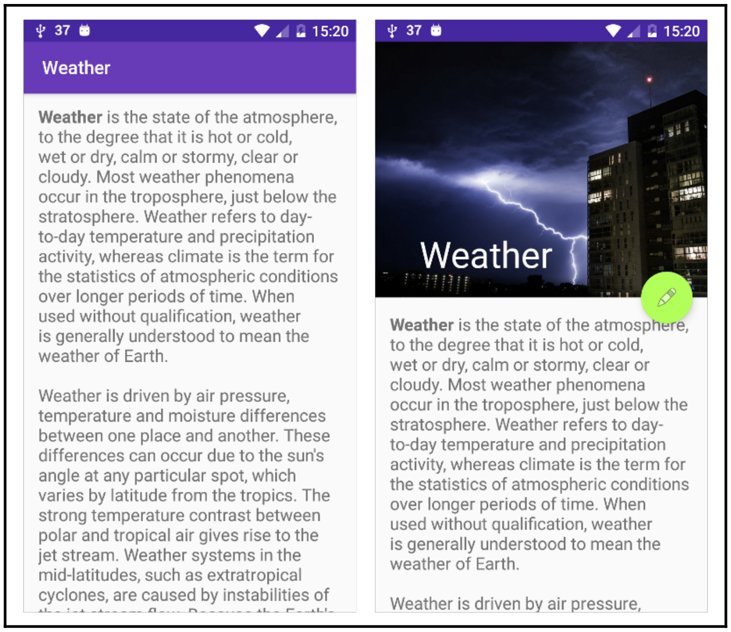
`Snackbar`, in the previous demonstration, behaves exactly like a toast, but `Snackbar` is `ViewGroup` and not a view like the toast; as a layout, it can act as a container. To see how this is done, replace the previous listener method with the one here:

```
@Override
public void onClick(View v) {
    Snackbar.make(coordinatorLayout,
            "Download complete",
            Snackbar.LENGTH_LONG)
            .setAction("Open", new View.OnClickListener() {

                @Override
                public void onClick(View v) {
                    // Perform action here
                }

            }).show();
    }
    }
```

The way the FAB moves out of the way of `Snackbar` is handled automatically by the parent coordinator layout, and this is the case for all design library widgets and ViewGroups. We will see shortly how we have to define our own behaviors when including native views, such as text views and images. We can also customize design component behavior, but first, we will take a look at the other design library components.

Collapsing app bars

Another well recognized Material design feature is the collapsing toolbar. This generally contains a relevant image and a title. This type of toolbar will fill a large proportion of the screen when the user scrolls to the top of the content, it handily tucks itself out of the way when the user wishes to view more content and scrolls down. This component serves a useful purpose, as it provides a great branding opportunity and a chance for our app to stand out visually, but it does so without using up valuable screen real estate.



Weather is the state of the atmosphere, to the degree that it is hot or cold, wet or dry, calm or stormy, clear or cloudy. Most weather phenomena occur in the troposphere, just below the stratosphere. Weather refers to day-to-day temperature and precipitation activity, whereas climate is the term for the statistics of atmospheric conditions over longer periods of time. When used without qualification, weather is generally understood to mean the weather of Earth.

Weather is driven by air pressure, temperature and moisture differences between one place and another. These differences can occur due to the sun's angle at any particular spot, which varies by latitude from the tropics. The strong temperature contrast between polar and tropical air gives rise to the jet stream. Weather systems in the mid-latitudes, such as extratropical cyclones, are caused by instabilities of

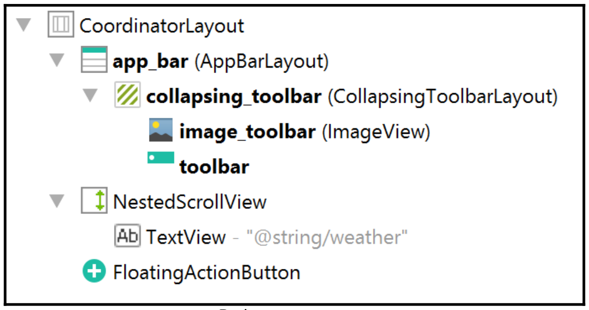
Weather is the state of the atmosphere, to the degree that it is hot or cold, wet or dry, calm or stormy, clear or cloudy. Most weather phenomena occur in the troposphere, just below the stratosphere. Weather refers to day-to-day temperature and precipitation activity, whereas climate is the term for the statistics of atmospheric conditions over longer periods of time. When used without qualification, weather is generally understood to mean the weather of Earth.

Weather is driven by air pressure,

A collapsing app bar

The best way to see how this is constructed is to examine the XML code behind it. Follow the steps to recreate it:

1. Start a new Android Studio project. We will be creating the following layout:



```
CoordinatorLayout
  app_bar (AppBarLayout)
    collapsing_toolbar (CollapsingToolbarLayout)
      image_toolbar (ImageView)
      toolbar
  NestedScrollView
    TextView - "@string/weather"
  FloatingActionButton
```

Project component tree

2. First open the `styles.xml` file.
3. Ensure that the parent theme speculates no action bar, as follows:

```
<style name="AppTheme"
    parent="Theme.AppCompat.Light.NoActionBar">
```

4. Add the following line if you want a translucent status bar:

```
<item name="android:windowTranslucentStatus">true</item>
```

5. As before, create a layout file with `CoordinatorLayout` at its root.
6. Next, nest the following `AppBarLayout`:

```
<android.support.design.widget.AppBarLayout
    android:id="@+id/app_bar"
    android:layout_width="match_parent"
    android:layout_height="300dp"
    android:fitsSystemWindows="true"
    android:theme="@style/ThemeOverlay
        .AppCompat
        .Dark
        .ActionBar">
```

7. Inside this, add `CollapsingToolbarLayout`:

```
<android.support.design.widget.CollapsingToolbarLayout
    android:id="@+id/collapsing_toolbar"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
```

```
        android:fitsSystemWindows="true"
        app:contentScrim="?attr/colorPrimary"
        app:expandedTitleMarginEnd="64dp"
        app:expandedTitleMarginStart="48dp"
        app:layout_scrollFlags="scroll|exitUntilCollapsed"
        app:>
```

8. Inside the toolbar, add these two widgets:

```
    <ImageView
        android:id="@+id/image_toolbar"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:fitsSystemWindows="true"
        android:scaleType="centerCrop"
        app:layout_collapseMode="parallax"
        app:srcCompat="@drawable/some_image" />

    <android.support.v7.widget.Toolbar
        android:id="@+id/toolbar"
        android:layout_width="match_parent"
        android:layout_height="?attr/actionBarSize"
        app:layout_collapseMode="pin"
        app:popupTheme="@style/ThemeOverlay.AppCompat.Light" />
```

9. Beneath `AppBarLayout`, place `NestedScrollView` and `TextView`:

```
    <android.support.v4.widget.NestedScrollView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        app:layout_behavior=
            "@string/appbar_scrolling_view_behavior">

        <TextView
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:padding="@dimen/activity_horizontal_margin"
            android:text="@string/some_string"
            android:textSize="16sp" />

    </android.support.v4.widget.NestedScrollView>
```

10. Finally add an FAB:

```
    <android.support.design.widget.FloatingActionButton
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_margin="@dimen/activity_horizontal_margin"
```

```
                    app:layout_anchor="@id/app_bar"
                    app:layout_anchorGravity="bottom|end"
                    app:srcCompat="@android:drawable/ic_menu_edit" />
```

If you now test this on a device or emulator, you will see that the toolbar collapses and expands automatically, without any programming, and this is the beauty of the design library. To code this type of behavior without it would be a lengthy and often difficult process.

Most of the preceding XML is self-explanatory, but there are one or two points that are worth glancing at.

Raw text resources

To demonstrate the scrolling behavior, the string used in the preceding text view was a lengthy one. This was placed in the `strings.xml` file, and although this works perfectly well, it is not an elegant way to manage lengthy texts. This kind of text is better dealt with as a text file resource that can be read at runtime.

The following steps demonstrate how this is done:

1. Prepare a plain text file.
2. Create a directory named `raw` inside the `res` directory of your project by right-clicking on the `res` folder in the project explorer and selecting `New | Directory`.
3. Add the text file to this directory.

Project directories can be opened quickly from the explorer context menu.

4. Open the java activity with the text view you wish to populate and add this function:

```
private StringBuilder loadText(Context context) throws
IOException {
    final Resources resources = this.getResources();
    InputStream stream = resources
        .openRawResource(R.raw.weather);
    BufferedReader reader =
        new BufferedReader(
        new InputStreamReader(stream));
```

```
            StringBuilder stringBuilder = new StringBuilder();
            String text;

            while ((text = reader.readLine()) != null) {
                stringBuilder.append(text);
            }

            reader.close();
            return stringBuilder;
        }
```

5. Finally, add this code to the `onCreate()` method:

```
        TextView textView = (TextView)
            findViewById(R.id.text_view);

        StringBuilder builder = null;

        try {
            builder = loadText(this);
        } catch (IOException e) {
            e.printStackTrace();
        }

        textView.setText(builder);
```

The other point in the preceding demonstration is the use of a hardcoded value for the height of the expanded toolbar, that is, `android:layout_height="300dp"`. This works just fine on the model it was tested on but to achieve the same effect on all screen types could involve creating a large number of alternative layouts. One simpler solution is to only recreate the `dimens` folder, which can be simply copied and pasted, for example, `dimens-hdpi`, and edit only the appropriate value. One could even create a separate file containing just this value. Another way to combat this is with a support library designed for just this kind of situation.

The percent library

The percent support library offers just two layout classes `PercentRelativeLayout` and `PercentFrameLayout`. It needs to be added to the gradle build file as a dependency, as follows:

```
compile 'com.android.support:percent:25.1.1'
```

To recreate the layout in the previous section, we need to place `AppBarLayout` inside `PercentRelativeLayout`. We can then use a percentage value to set the maximum height of our app bar, as follows:

```
<android.support.percent.PercentRelativeLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

    <android.support.design.widget.AppBarLayout
        android:id="@+id/app_bar"
        android:layout_width="match_parent"
        android:layout_height="30%"
        android:fitsSystemWindows="true"
        android:theme="@style/ThemeOverlay
            .AppCompat
            .Dark
            .ActionBar">

        . . .

    </android.support.design.widget.AppBarLayout>

</android.support.percent.PercentRelativeLayout>
```

This method saves us having to create a large number of alternative layouts to reproduce the same effect across numerous devices, although it will always be necessary to generate more than one.

Another effective way of achieving this uniformity is to create our image drawables so that they are the exact height we require in dp and set the layout height in XML as `wrap_content`. All we need to do then is create an image for each desired designated resource directory, something we were very likely to do anyway.

Together, the preceding tools make designing material interfaces simple and intuitive, and also provide ways to cut down on the amount of time it takes to prepare layouts for the bewildering array of devices available to the user.

Summary

In this chapter, we built on the work of the previous chapter and explored how more sophisticated layouts can be easily constructed using the coordinator layout and its associated library, which does a great deal of the work for us such as automating collapsing toolbars and preventing overlapping widgets.

We concluded the chapter by exploring another invaluable design library, the percent library, which can solve a multitude of design problem when developing for very different screen sizes and shapes.

The next chapter will expand on this one by exploring more dynamic elements for interface development, such as screen rotation, developing for wearables, and reading sensors.

4
Device Development

Android Studio provides some very powerful layout tools, enabling us to quickly and easily experiment with and develop user interfaces. However, perhaps the biggest challenge any Android developer faces is the bewildering number of form factors their applications could run on.

We saw in previous chapters how classes, for example the constraint layout and libraries such as the percent library, help us design uniform and consistent layouts. However, these techniques only provide general solutions, and we will all have stumbled across apps that do not really seem to have been designed with our device in mind. With a little knowledge and effort, these design faults can easily be avoided.

In this chapter, you will learn to:

- Create alternative layout files
- Extract string resources
- Manage screen rotation
- Configure resources
- Create wearable UIs
- Build shape-aware layouts
- Read sensor data
- Use virtual sensors
- Apply Studio's templates
- Create a debug filter
- Monitor devices

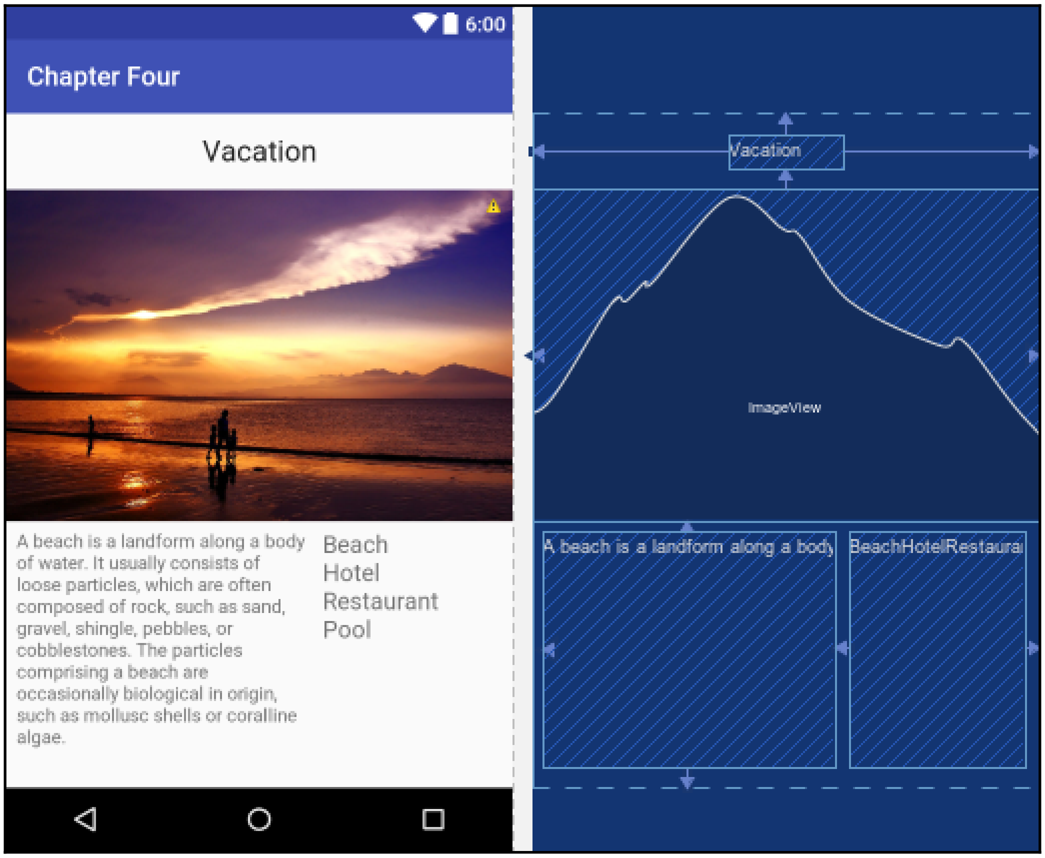
Before looking into how to develop our UIs so that they look great on all our users' devices, we need to explore the most significant layout situation we will encounter: the rotation of a screen between portrait and landscape mode.

Screen orientation

A large percentage of Android apps designed for phones and tablets are designed to work in both landscape and portrait mode and generally switch between these automatically. Many activities, such as videos, are best viewed in landscape mode, while lists are usually easier to scan in portrait; however, and there are some activities, and even entire apps, where the orientation is fixed.

There are some layouts that look equally good, whichever way they are viewed, but this is not often the case; most of the time, we will want to design one for each orientation. Android Studio simplifies and speeds up this process by saving us the task of developing an alternative layout from scratch.

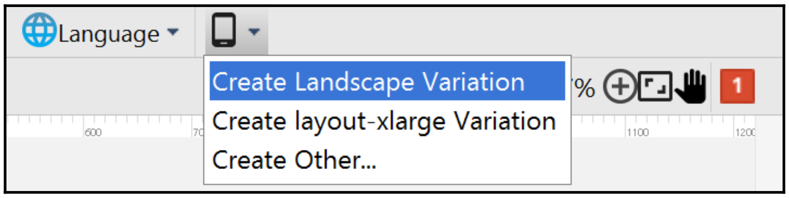
Take a simple layout like the one here:



A beach is a landform along a body of water. It usually consists of loose particles, which are often composed of rock, such as sand, gravel, shingle, pebbles, or cobblestones. The particles comprising a beach are occasionally biological in origin, such as mollusc shells or coralline algae.

A portrait layout

A landscape variant can be created with a single click from the **Layout Variant** tool at the top of the design editor, as follows:

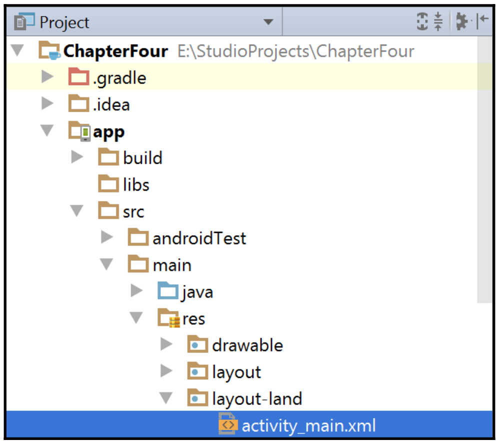


The Layout Variant tool.

If you recreate this exercise or create an equivalent of your own, you will soon see that a layout like this does not look good when just rotated, and you will have to reposition the views to best suit this aspect ratio. If you try this with a constraint layout, you will of discovered one of its few weaknesses, and the resultant layout can be quite a mess.

Just how you recreate these layouts is down to your own artistic and design skills, but what is worth noting is the manner in which Android Studio stores and presents these files as this can be a little confusing, particularly if you are migrating from Eclipse, which manages this differently.

If you open the project you just created in the Project Explorer, under Android you will find the landscape variant as `activity_main.xml (land)`, apparently inside the `activity_main.xml` directory. Studio presents it like this because it is convenient to have all our layouts in one place, but this is not how they are stored. Switching the Project Explorer to the **Project** view will display the actual file structure, as follows:



Project structure.

This structure can also be determined from the navigation bar at the top of the IDE.

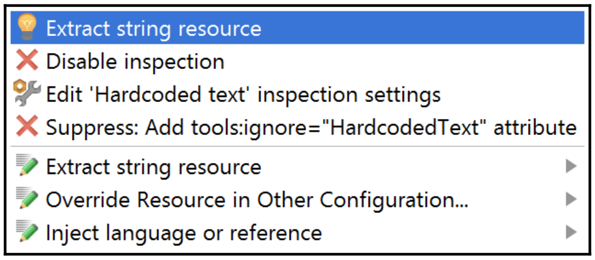
If you create layout variants like this, moving views into a more pleasing configuration, and give both versions the same ID, these will automatically animate between their two states when a user rotates their device. We will later see how to construct our own custom animations, but more often than not, the default animations are the best choice as they help promote a uniform user experience.

If you recreated the example above, you may have noticed a rather neat trick that the IDE performs to speed up the process of providing text resources.

You will already know that the use of hard-coded strings is strongly deprecated. Like many programming paradigms, Android development is designed so that data and code are created and worked on separately. Hard-coded strings also make translations nearly impossible.

We saw previously how the quick-fix function allows us to automatically implement methods. Here, we can use it to create string resources without ever even having to open the `strings.xml` file.

Simply enter a hard-coded string in your layout file and follow the quick-fix prompt to extract it as a string resource.


Extract string resource
Disable inspection
Edit 'Hardcoded text' inspection settings
Suppress: Add tools:ignore="HardcodedText" attribute
Extract string resource
Override Resource in Other Configuration...
Inject language or reference

String resource extraction.

The Layout Editor provides two ready made variants, landscape and extra large, but we can create variants of our own to suit any form factor we choose.

Now that we have started to add some dynamic elements such as screen rotation, the Layout Editor is not enough and we need to run our apps on a device or emulator.

Virtual devices

For a long time, **Android Virtual Devices** (**AVDs**) had the reputation of being buggy and horrendously slow. The introduction of hardware acceleration has made a big difference, but a powerful computer is still advised, especially if you want to run more than one at a time, which is very often the case.

The biggest change to Android emulation is not hardware acceleration, but rather the appearance of alternative emulators. As we shall see shortly, some of these offer distinct advantages over the native emulator, but AVDs should not be written off. Despite the drawbacks, Android emulators are the only emulators that run on all Android versions, including the most recent, developer-only, versions. Not only this, but Android emulators are the most customizable and any possible hardware or software configuration can be recreated with a little effort.

Early on in the development process, it is important to be able to test out our ideas quickly and using one or two real-world devices is probably the best choice for this level of testing; however, sooner or later we are going to need to make sure that our layouts look great on all possible devices.

Layout and image qualification

There are two issues that we need to consider here: screen density and aspect ratio. If you have done any Android development before, you will be aware of DPI and screen size groupings. These designated folders provide handy shortcuts to suit the enormous variety of available form factors, but we will all have experienced apps with layouts that don't quite work on our devices. This is something that is entirely avoidable, and although countering it will take some effort on our part, it will result in avoiding those poor ratings that can so damage a revenue stream.

It is very tempting to create an app that will work on as many form factors as possible, and Android Studio will occasionally encourage you to think that way. In reality we have to think about when and where devices are used. If we are waiting for a bus, then we probably want a game that is easily switched on and off and where tasks can be completed quickly. And although there are exceptions, these are not the same games that people choose to play on large screens for long periods. Picking the right platform is essential, and although it may sound counter-intuitive, it is often wiser to exclude a platform than just assume it may earn a little more revenue.

Bearing this in mind, we will consider an app designed only for phones and tablets; however as well as looking at familiar features such as screen size and density, we will see how we can provide customized resources for many other configuration issues.

The two most commonly used resource designations are screen size and density. Android provides the following four size designations.

- `layout-small`: from two to four inches, 320 x 420dp or larger
- `layout-normal`: from three to five inches, 320 x 480dp or larger
- `layout-large`: from four to seven inches, 480 x 640dp or larger
- `layout-xlarge`: from seven to 10 inches, 720 x 960dp or larger

If you are developing for Android 3.0 (API level 11) or below, devices at the lower end of this scale will often be categorized incorrectly. The only solution to this is to configure for individual devices or avoid developing for such devices at all.

Generally speaking, we will need to produce a layout for each of the above sizes.

The use of **density-independent pixels** (**dp** or **dip**) means that we do not need to design a new layout for each density setting, but we do have to provide a separate drawable for each density class, which are as follows.

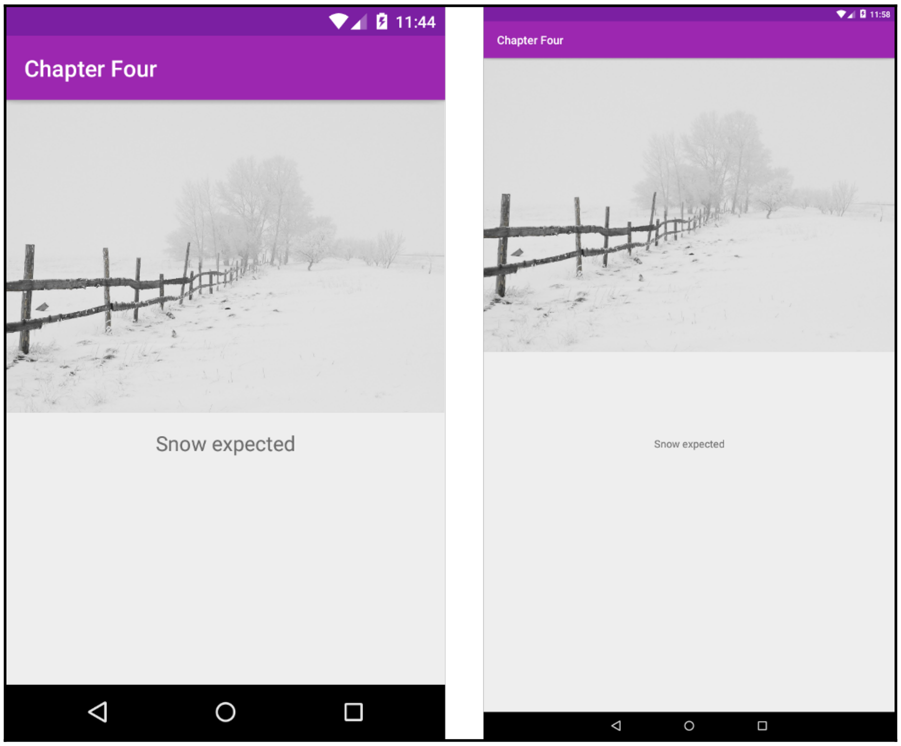
- `drawable-ldpi` ~ 120dpi
- `drawable-mdpi` ~160dpi
- `drawable-hdpi` ~240dpi
- `drawable-xhdpi` ~320dpi
- `drawable-xxhdpi` ~480dpi
- `drawable-xxxhdpi` ~640dpi

The dpi values in the preceding list inform us of the relative size in pixels that our resources need to be. For example bitmaps in the `drawable-xhdpi` directory need to be twice the size of their equivalent in the `drawable-mdpi` folder.

It is not really possible to create exactly the same output on every device, and this is not even desirable. People buy high-end devices because they want stunning imagery and fine detail, and we should endeavor to provide this level of quality. On the other hand many people buy small and less expensive devices for reasons of convenience and budget and we should reflect these choices in our designs. Rather than try to reproduce exactly the same experience on all devices, we should think about the reasons why people choose their devices and what it is they want from them.

The following short exercise demonstrates how these differences manifest themselves across different screen configurations. This will give readers the opportunity to see how to best exploit the user's choice of device, using their own artistic and design acumen.

1. Select any high-resolution image, ideally a photograph.
2. Using whatever tools you choose, create a copy that has half the width and height of the original.
3. Open a new Android Studio project.
4. From the Project Explorer, create two new folders inside the res directory, called `drawable-mdpi` and `drawable-hdpi`.
5. Place the prepared images in these folders.
6. Build a simple layout with an image view and some text.
7. Create two virtual devices, one with `mdpi` density and one `hdpi`.
8. Finally, run the app on each device to observe the differences.



Devices with mdpi and hdpi densities.

These are not actually the only density qualifiers we can use. Apps designed for televisions often use the `tvdpi` qualifier. This has a value between `mdpi` and `hdpi`. There are also the `nodpi` qualifier, which is used when we want exact pixel mapping, and `anydpi`, which is used when all artwork is vector drawables.

There are a lot of other qualifiers and a full list can be found at:

`developer.android.com/guide/topics/resources/providing-resources.html`

It is worth taking a look at some of the more useful ones now.

Scale and platform

Generalized qualifiers such as those discussed earlier are very useful, suit most purposes, and save us a lot of time. However, there are times when we want more exact information about the device our app is running on.

One of the most important features we want to have information about is screen size. We have already encountered qualifiers such as small, normal, and large, but we can also configure for more precise dimensions. The simplest of these is the available width and available height. For example, layouts in `res/layout/w720dp` will only inflate when there is a minimum of 720dp available and height and `res/layout/h1024dp` will inflate when the screen height is equal to or greater than 1024dp.

Another very handy feature to configure resources for is the platform version number. This operates on the API level. So one would use a qualifier of `v16` for resources to be used when running on Android Jelly Bean devices.

Being able to select and prepare resources for such a wide range of hardware means we can provide lavish resources for those devices that are capable of displaying them and simpler resources for devices that have reduced capacity. Whether we are developing for budget phones or high-end tablets we still need some way to test our apps. We have already seen how flexible AVDs can be, but it is well worth taking a quick look at some of the alternatives.

Alternative emulators

One of the very best alternative emulators is probably Genymotion. This unfortunately is not free and is not as up-to-date as the native AVDs but it is fast and supports drag-and-drop file installation and mobile network functionality. It can be found at:

`www.genymotion.com`

Another fast and easy to use emulator is Manymo. This is is a browser-based emulator and its primary purpose is to test web apps, but it works perfectly well for mobile apps. It too is not free but it does have a wide selection of ready-made form factors. It can be found at:

`www.manymo.com`

In a very similar vein there is Appetize, which is located at:

`appetize.io`

There are a growing number of such emulators but those mentioned above are probably the most functional from a development perspective. The following list directs the reader to some of the others:

- `www.andyroid.net`
- `www.bluestacks.com/app-player.html`
- `www.droid4x.com`
- `drive.google.com/file/d/0B728YkPxkCL8Wlh5dGdiVXdIS0k/edit`

There is one scenario where none of these alternatives are suitable and we are forced to use the AVD manager and that is when we want to develop for wearables, such as smart watches, which is what we will look at next.

Android Wear

Wearable devices have become very popular of late and Android Wear is fully incorporated into the Android SDK. The setting up of a Wear project is slightly more involved than other projects as wearable devices really act as a companion device with the apps themselves running from a mobile device.

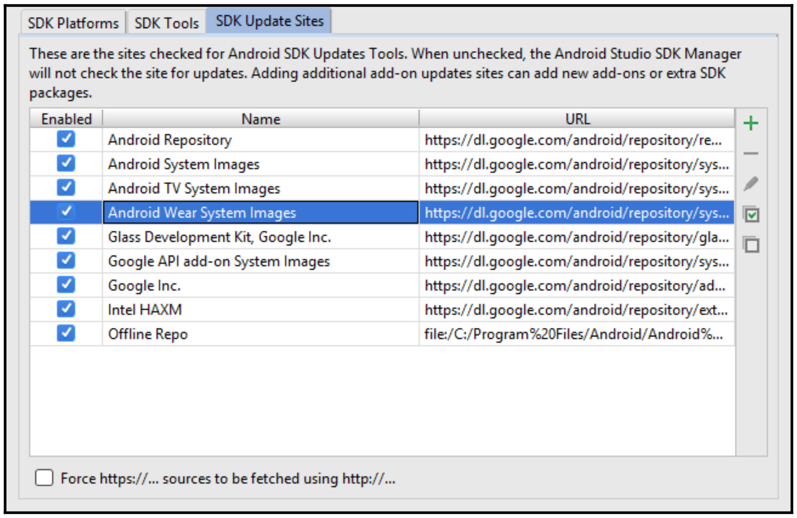
Despite this minor level of complication, developing for wearables can be a lot of fun, not least because they often offer us access to some cool sensors, such as the heart rate monitor.

Connecting to a wearable AVD

It may well be that you have access to a wearable device, but here we will be using emulators in the following exercise. This is because these devices come in two flavors: square and round.

When it comes to pairing one of these emulators with a phone or tablet, this can be done with either a real device or with another emulator, with a real device being preferable as this puts less strain on the computer. These two approaches are slightly different. The following exercise assumes you are pairing a wearable emulator with a real device, with an explanation of how to pair with an emulated mobile device at the end.

1. Before doing anything else, open the **SDK Manager** and check that you have downloaded **Android Wear System Images:**



These are the sites checked for Android SDK Updates Tools. When unchecked, the Android Studio SDK Manager will not check the site for updates. Adding additional add-on updates sites can add new add-ons or extra SDK packages.

Enabled	Name	URL
✓	Android Repository	https://dl.google.com/android/repository/re...
✓	Android System Images	https://dl.google.com/android/repository/sys...
✓	Android TV System Images	https://dl.google.com/android/repository/sys...
✓	Android Wear System Images	https://dl.google.com/android/repository/sys...
✓	Glass Development Kit, Google Inc.	https://dl.google.com/android/repository/gla...
✓	Google API add-on System Images	https://dl.google.com/android/repository/sys...
✓	Google Inc.	https://dl.google.com/android/repository/ad...
✓	Intel HAXM	https://dl.google.com/android/repository/ext...
✓	Offline Repo	file:/C:/Program%20Files/Android/Android%...

☐ Force https://... sources to be fetched using http://...

2. Open the AVD Manager and create two AVDs, one round and one square.
3. Install the Android Wear app from the Play store on your handset and connect it to the computer.
4. Locate and open the directory containing the `adb.exe` file. This can be found in `\AppData\Local\Android\Sdk\platform-tools\`.
5. Issue the following command:

```
adb -d forward tcp:5601 tcp:5601
```

6. Launch the companion app on your handset and follow the on-screen instructions to pair the devices.

You will need to execute the port forwarding command each time you reconnect the handset.

If you are going to pair the wearable with a virtual handset, the process is very similar, the only difference being the manner in which the companion app is installed. Follow the steps to achieve this:

1. Start or create an AVD that targets Google APIs.
2. Download `com.google.android.wearable.app-2.apk`. There are many places online where the file can be found, such as www.file-upload.net/download.
3. Put the file in your platform-tools folder and install it with:

    ```
    adb install com.google.android.wearable.app-2.apk
    ```

4. Start the wearable AVD and enter `adb devices` at the command prompt (or Terminal if you are on a Mac) to check that both devices are visible.
5. Enter `adb telnet localhost 5554`, where `5554` is the phone emulator.
6. Finally enter `adb redir add tcp:5601:5601`. You can now use the wear app on the emulated phone in the same way as the previous exercise to pair the devices.

Although it is added for us automatically, it is still important to understand that Android Wear apps require a support library. This can be seen by examining the module level in the `build.gradle` file.

```
compile 'com.google.android.gms:play-services-wearable:10.2.0'
```

With our devices now paired, we can begin to actually develop and design our wearable app.

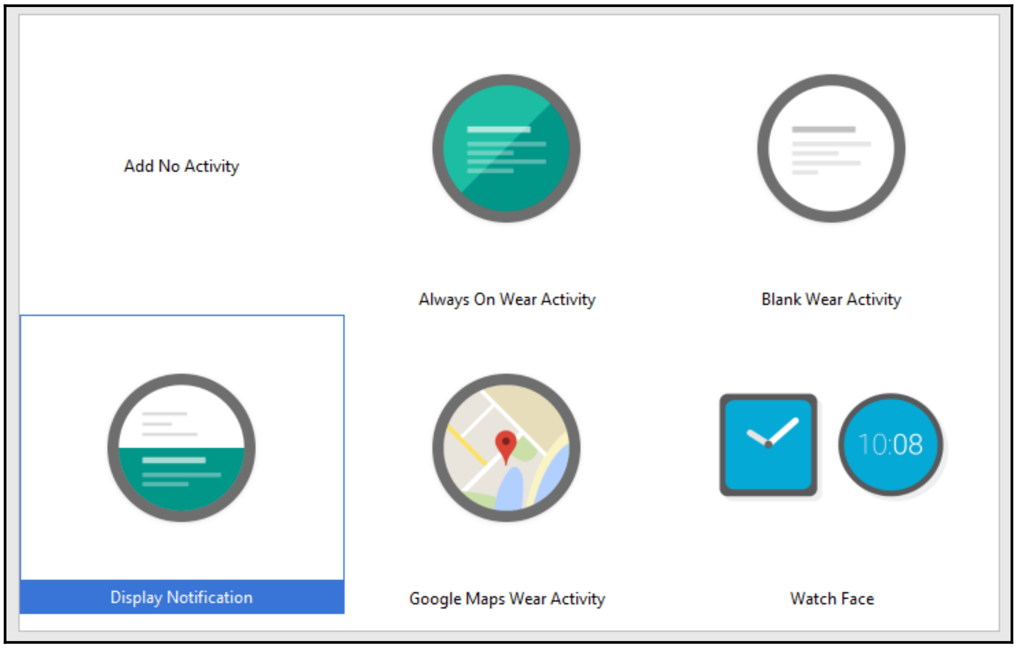
Wearable layouts

One of the most interesting challenge when it comes to Android Wear UI development, is the two different shapes these smart watches come in. There are two ways we can approach this.

One is similar to the way we have managed things previously and involves designing a layout for each form factor whilst the other technique uses a method that produces a layout that will work for either shape.

On top of these techniques the wearable support library comes equipped with some very handy widgets suited to curved and round layouts and lists.

One of the most useful and instructive features of Android Studio is the project templates that are offered when a project is first set up. There is a good selection of these and they provide good starting points to most projects, in particular Wear apps.



Wear templates

Starting a project this way can be helpful and revealing and even the blank activity template sets up both XML and Java files, creating a very creditable starting point.

If you start a project from Blank Wear Activity, the first thing you will notice is that, where we previously had only one module (called **app** by default), we now have two modules, one called **mobile** that replaces **app** and another named **wear**. Both these modules have the same structures as those we have encountered before, containing a manifest, resource directory and Java activities.

The WatchViewStub class

The blank Wear activity template applies the first technique we discussed earlier for managing different device shapes. This takes the form of the `WatchViewStub` class, which can be found in the `wear/src/main/res/layout` folder.

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.wearable.view.WatchViewStub
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/watch_view_stub"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:rectLayout="@layout/rect_activity_main"
    app:roundLayout="@layout/round_activity_main"
    tools:context="com.mew.kyle.wearable.MainActivity"
    tools:deviceIds="wear" />
```

As can be seen in the preceding example, the main activity directs the system to one or other of the two shaped layouts, which the template also provides.

As you can see, this is not the way we have selected the correct layout previously and that is because `WatchViewStub` operates differently and requires a specialized listener that inflates our layouts once `WatchViewStub` has detected the watch face type. This code too is provided by the template in the main activity Java file:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    final WatchViewStub stub = (WatchViewStub)
            findViewById(R.id.watch_view_stub);

    stub.setOnLayoutInflatedListener(new
WatchViewStub.OnLayoutInflatedListener() {

        @Override
        public void onLayoutInflated(WatchViewStub stub) {
            mTextView = (TextView) stub.findViewById(R.id.text);
        }

    });
}
```

It is tempting to think that `WatchViewStub` is all we need to design our wearable layouts. It allows us to design for both faces independently, which is precisely what we want to do. However, Wear layouts are generally very simple and indeed complex designs are strongly discouraged. Thus with a simple design of little more than an image and a button, it is simply a matter of convenience to have a `shape-aware` class that distributes its contents according to the shape of the device it finds itself being inflated on. This is how the `BoxInsetLayout` class works.

Shape - aware layouts

The `BoxInsetLayout` class is part of the Wear UI library and allows us to design just one layout that will optimize itself for both square and round watch faces. It does this by inflating the largest possible square within any round frame. This is a simple solution but what the BoxInsetLayout also does very nicely is ensure that any background image we choose always fills all available space. As we shall see in a moment, if you position components horizontally across the screen, the `BoxInsetLayout` class automatically distributes them to cause a best fit.

One of the very first things you will want to do, when developing for these unusual form factors when working in Android Studio, is take advantage of the powerful preview system provided by the Layout Editor. This provides previews of each type of wearable device as well as any AVDs you may have created. This saves a great deal of time when testing a layout as we can view this directly from the IDE without having to launch an AVD.

The preview tool can be accessed from the `View | Tool Windows` menu; or, if the **layout Text** editor is open; it can be found, by default, in the right hand margin.

Unlike `WatchViewStubs`, the `BoxInsetLayout` class is not provided by any of the templates and must be coded manually. Follow the short steps below to construct a dynamic Wear UI using the `BoxInsetLayout` class .

1. Create the following `BoxInsetLayout` as the root container of the main XML activity in the wear module:

```
<android.support.wearable.view.BoxInsetLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
```

```
        android:background="@drawable/snow"
        android:padding="15dp">

    </android.support.wearable.view.BoxInsetLayout>
```

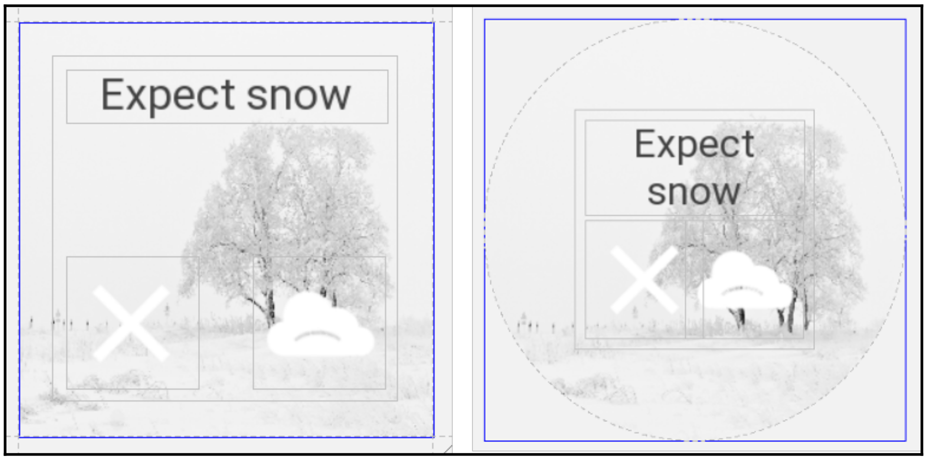
2. Place this `FrameLayout` inside the `BoxInsetLayout` class:

```
    <FrameLayout
        android:id="@+id/wearable_layout"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:padding="5dp"
        app:layout_box="all">

    </FrameLayout>
```

3. Inside `FrameLayout` include these widgets (or those of your own choosing):

```
    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:gravity="center"
        android:text="@string/weather_warning"
        android:textAppearance=
            "@style/TextAppearance.WearDiag.Title"
        tools:textColor="@color/primary_text_light" />

    <ImageView
        android:layout_width="60dp"
        android:layout_height="60dp"
        android:layout_gravity="bottom|start"
        android:contentDescription=
            "@string/generic_cancel"
        android:src="@drawable/ic_full_cancel" />

    <ImageView
        android:layout_width="60dp"
        android:layout_height="60dp"
        android:layout_gravity="bottom|end"
        android:contentDescription=
            "@string/buttons_rect_right_bottom"
        android:src="@drawable/ic_full_sad" />
```

4. Finally, run the demonstration on both a round and a square emulator:



The BoxInsetLayout

The `BoxInsetLayout` class is wonderfully easy to use. Not only does it save us time, it also keeps the memory footprint of our app down, as even the simplest layout has some cost. It may seem that it is somewhat wasteful of space in the round view, but Wear UIs should be bare and stripped down and empty space is not something to be avoided; a well designed wearable UI should be quickly grasped by the user.

One of the most frequently used features of Android Wear is the heart rate monitor and, as we are working with wearables, now would be a good time to look at how we access sensor data.

Accessing sensors

Devices worn on the wrist are ideal for fitness apps and the inclusion of a heart rate monitor in many models makes them perfect for such tasks. The way that the SDK manages all sensors is almost identical, so seeing how one works applies to the others.

The following exercise demonstrates how to read the heart rate sensor on a wearable device:

1. Open an Android Wear project with both mobile and wear modules.
2. Create a layout of your choosing, ensuring you include a `TextView` to display the output.

3. Open the `Manifest` file in the wear module and add the following permission:

```
<uses-permission
    android:name="android.permission.BODY_SENSORS" />
```

4. Open the `MainActivity.java` file in the wear module and add the following fields:

```
private TextView textView;
private SensorManager sensorManager;
private Sensor sensor;
```

5. Have the `Activity` implement a sensor event listener, like so:

```
public class MainActivity extends Activity implements
SensorEventListener {
```

6. Implement the methods required by this.
7. Edit the `onCreate()` method as follows:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    textView = (TextView) findViewById(R.id.text_view);

    sensorManager = ((SensorManager)
getSystemService(SENSOR_SERVICE));
    sensor = sensorManager
        .getDefaultSensor(Sensor
        .TYPE_HEART_RATE);
}
```

8. Add the `onResume()` method to register the listener when the activity starts or restarts:

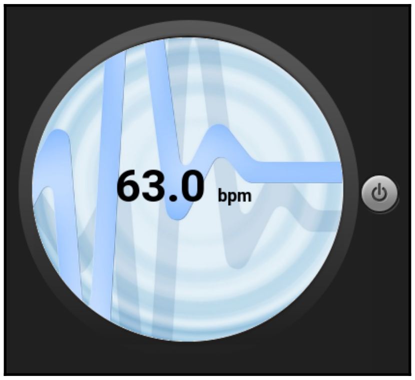
```
@Override
protected void onResume() {
    super.onResume();

sensorManager.registerListener(this, this.sensor, 3);
}
```

9. Then add the `onPause()` method to ensure the listener is switched off when not required:

```
@Override
protected void onPause() {
    super.onPause()

sensorManager.unregisterListener(this);
}
```

10. Finally, edit the `onSensorChanged()` callback, as follows:

```
@Override
public void onSensorChanged(SensorEvent event) {
   textView.setText("" + (int) event.values[0] + "bpm");
}
```



As mentioned earlier, all sensors can be accessed in the same fashion, although of course the values they output differ according to their purpose. Full documentation of this can be found at:

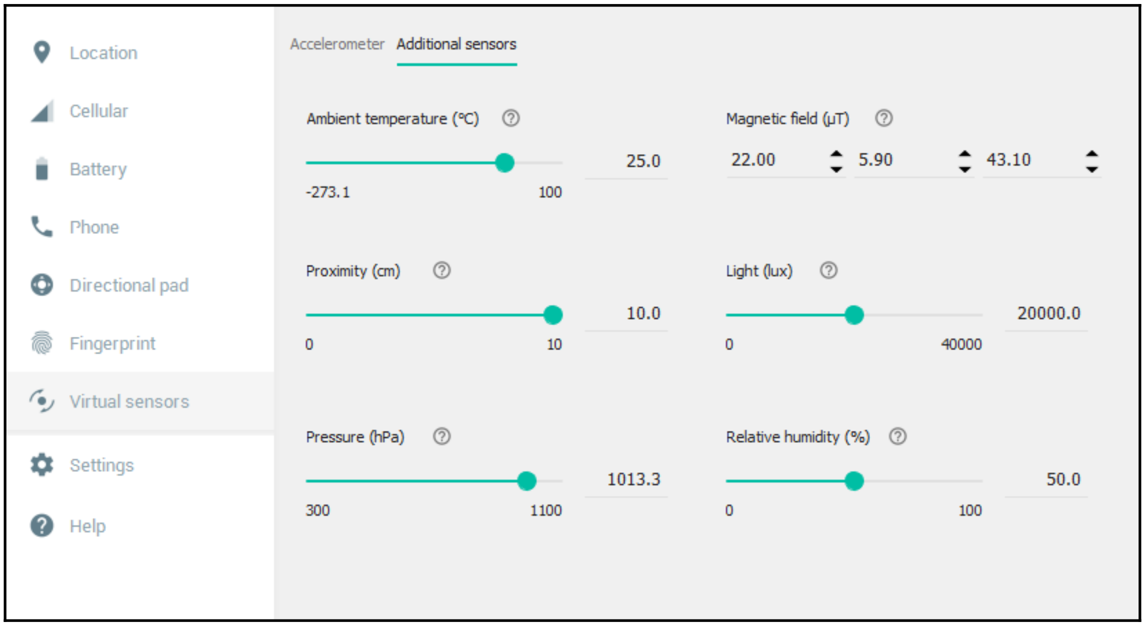
`developer.android.com/reference/android/hardware/Sensor.html`

Now, of course, the reader will be thinking this exercise is pointless without an actual device with an actual sensor. Fortunately, there is more than one way to make up for this lack of hardware in an emulator.

Sensor emulation

If you have not used the Android emulators for some time or are new to them, you may well have missed the extended controls that each AVD has. These can be accessed from the bottom of the emulator's toolbar.

These extended controls offer a host of useful functions such as the ability to easily set mock locations and alternative input methods. Of interest to us here are **virtual sensors**. These allow us to simulate a variety of sensors and input values directly:



Ambient temperature (°C): 25.0 (range -273.1 to 100)
Magnetic field (μT): 22.00, 5.90, 43.10
Proximity (cm): 10.0 (range 0 to 10)
Light (lux): 20000.0 (range 0 to 40000)
Pressure (hPa): 1013.3 (range 300 to 1100)
Relative humidity (%): 50.0 (range 0 to 100)

Virtual sensors

There are several other routes to running sensors on emulated devices. Most of these depend on connecting real devices and using their hardware. These SDK controller sensors can be download from the Play store. There are also some great sensor simulators on GitHub, my personal favorite being:

`github.com/openintents/sensorsimulator`

Now that we are beginning to develop more than just static layouts, we can start to take advantage of some of Studio's more powerful monitoring tools.

Device monitoring

Very often, simply running an app on a device or emulator is enough to tell us if what we have designed works and what, if anything, we need to change. However it is always great to see what is going on under the hood, and Android Studio has some fantastic tools when it comes to the live monitoring of an app's behavior.

We will cover debugging in detail in the next module, but it is never too soon to play with the **Android Debug Bridge** (**ADB**) and Android Studio's Device Monitor tool is one of the most significant benefits of choosing the IDE over the alternatives.

This section also offers a good opportunity to take a closer look at project templates, another fantastic feature of Android Studio.

Project templates

Android Studio comes packed with many useful project templates. These are designed for a series of typical project types, such as full screen apps or Google Maps projects. Templates are partially completed projects with code, layouts, and resources that can be used as a starting point for our own creations. The growing presence of material design has made the `Navigation Drawer Activity` template one of the most used templates and the one we will use to examine the Device Monitor tool.

The `Navigation Drawer Activity` template is interesting and useful in several ways. Firstly, note that there are four layout files including the `activity_main.xml` file that we are familiar with. Examining this code, you will note the following node:

```
<include
    layout="@layout/app_bar_main"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

The purpose of this node is simple to understand and the `app_bar_main.xml` file that it refers to contains the coordinator layout and other views that we covered earlier in the book. The use of the `<include>` tag is by no means necessary but it is very useful if we ever want to reuse that code in another activity, and of course it produces far cleaner looking code.

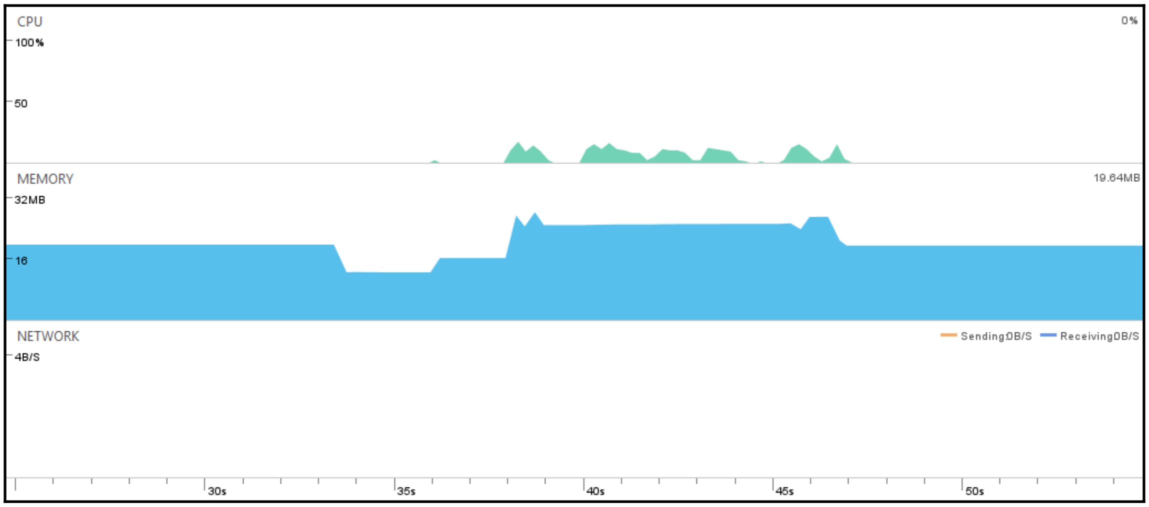
Another point of interest in this template is the use of vector graphics in the drawable directory. We will be looking at these in detail in the next chapter but for now it is enough to know that they provide a fantastic way to manage the problem of having to provide separate imagery for each screen density grouping as they will scale to any screen.

Before we take a look at how we can monitor an app's behavior, take a quick look at the main activity Java code. This shows very nicely how the various features are, and should be, coded. It is unlikely the example features will match those we want, but they can be replaced and edited very easily to suit our purposes, and an entire application can be built from this starting point.

Monitoring and profiling

One tool that all developers want is the ability to monitor an application during runtime. Watching the live impact user actions have on hardware components such as memory and processors is a fantastic way to identify possible bottlenecks and other problems. Android Studio has a sophisticated set of profiling tools, which will be examined thoroughly in the next module. However, the Android Profiler is useful for UI develeopment as well as coding and certainly worth looking at briefly here.

The Android Profiler can be opened from the **View | Tool Windows** menu, the tools gutter, or by pressing *Alt* + *6*. It appears at the bottom of the IDE by default, but this can be customized using the Settings icon to suit individual preferences:



The Android Profiler

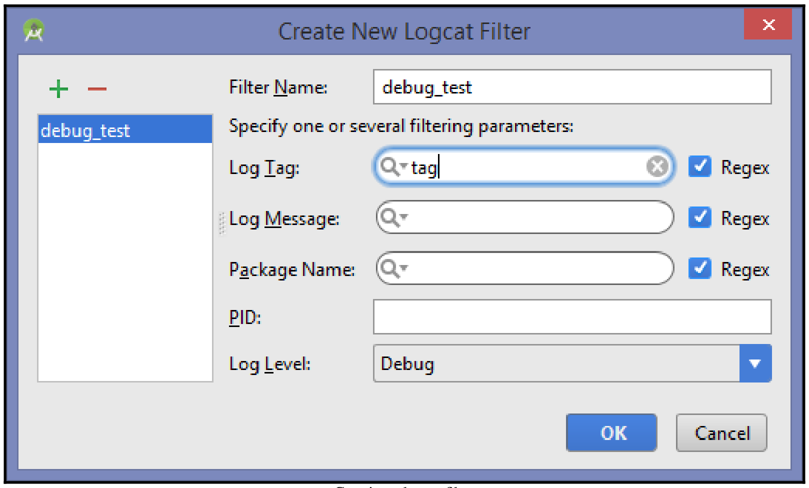
An advanced profiling option is available via the **Run configuration dialog**; this will be covered in the next module. For now there is another simple debugging/monitoring tool that can be very handy for UI design and development.

The visual feedback provided by the profiler provides a lot of useful information, but this information is fleeting and, although advanced profiling allows us to record very detailed examination, often all we need is to confirm that a particular event took place, or the order in which certain events took place.

For this, we can use another tool window, the logcat, and when all we need is to get some basic textual feedback on how and what our app is doing we can create a logcat filter for this purpose.

Perform the following steps to do this:

1. Open the **logcat** tool window via the **View** | **Tool Windows** menu or from the margin.
2. Select **Edit Filter Configuration** from the filter drop-down, on the right.
3. Complete the dialog, as follows:



debug_test

Filter Name: debug_test

Specify one or several filtering parameters:

Log Tag: tag ✓ Regex

Log Message: ✓ Regex

Package Name: ✓ Regex

PID:

Log Level: Debug

Creating a logcat filter

4. Add the following field to your `main` activity:

```
private static final String DEBUG_TAG = "tag";
```

5. Include the following import:

```
import android.util.Log;
```

6. Finally, add the highlighted line in the following code:

```
FloatingActionButton fab = (FloatingActionButton)
findViewById(R.id.fab);
fab.setOnClickListener(new View.OnClickListener() {

    @Override
    public void onClick(View view) {
        Snackbar.make(view, "Replace with your own action",
            Snackbar.LENGTH_LONG)
                .setAction("Action", null)
                .show();
        Log.d(DEBUG_TAG, "FAB clicked");
    }

});
```

7. Running the app with logcat open and tapping the FAB, will produce the following output.

```
...com.mew.kyle.devicemonitoringdemo D/tag: FAB clicked
```

Although this example is simple, the power of this technique is obvious and this form of debugging is the fastest and simplest way to check simple UI behavior, program flow, and the activity life cycle state.

Summary

This chapter has covered a lot of ground; we have taken the work done on Android layouts in an earlier chapter and begun to explore how these can be taken from static graphics to more dynamic structures. We have seen how Android provides classes and libraries that make developing for different screens easier than with other IDEs and how the emulator can be used to produce all possible form factors, including the most recent platforms.

There is only one more chapter in this module on layout and design before we move onto coding; in it we will cover how the numerous resources available to us are managed and how Android Studio assists us in this.

5
Assets and Resources

So far, in this book, we have covered layouts, design, and the libraries and tools that support them. We then went on to explore developing for different screen sizes, shapes, and densities, as well as other form factors. This is the last chapter in the UI development module where we will look at how Android Studio manages various assets and resources, such as icons and other drawables.

Android Studio is very accommodating when it comes to including drawables in our projects and, particularly, when it comes to vector graphics, which are invaluable to an Android developer, as they scale nicely across different screen sizes and densities, and this is catered for with a very valuable tool, the vector asset studio. Along with this, there is an asset studio to generate and configure bitmap images.

Vector drawables are widely used for in-app icons and in components such as menus, tabs, and the notification area and are also very flexible when it comes to animating icons and transforming them from one icon to another icon, a very useful space-saving function on small screens.

In this chapter, you will learn to do the following:

- Creating icons with asset studios
- Building adaptive icons
- Creating material launcher icons
- Using a material icon plugin
- Creating vector assets
- Importing vector assets
- Animating icons
- Viewing dynamic layouts with plugins
- Extracting prominent colors from an image

Asset Studio

There are very few, if any, apps that do not employ some forms of icons and even if these are only launcher and, action icons, the correct choices and design make the difference between a successful UI and a confusing one.

Although it is not essential, Google is very keen that we use material design icons. This is an attempt to create a uniform user experience across the platform to counter the perception that iOS offers a more consistent feel. This is unsurprising, as iOS is a closed system that places a lot of restrictions on the developer. Google, on the other hand, prefers to offer a far more creative freedom to developers. In the past, this has led Apple devices to gain a reputation for being generally slicker than Android and, to counter this, Google introduced material design guidelines, which have gone on to far exceed original expectations and can now be found on many other platforms, including iOS.

As would be expected, Android Studio provides tools to assist us in incorporating these design features and drawables. This comes in the form of Asset Studio. This facilitates the creation and configuration of all manner of icons, from brightly colored detailed launcher icons to fully customized and scalable vector graphic action and notification icons. Along with API level 26, Android introduced Adaptive Icons that can display as different shapes on different devices and perform simple animations.

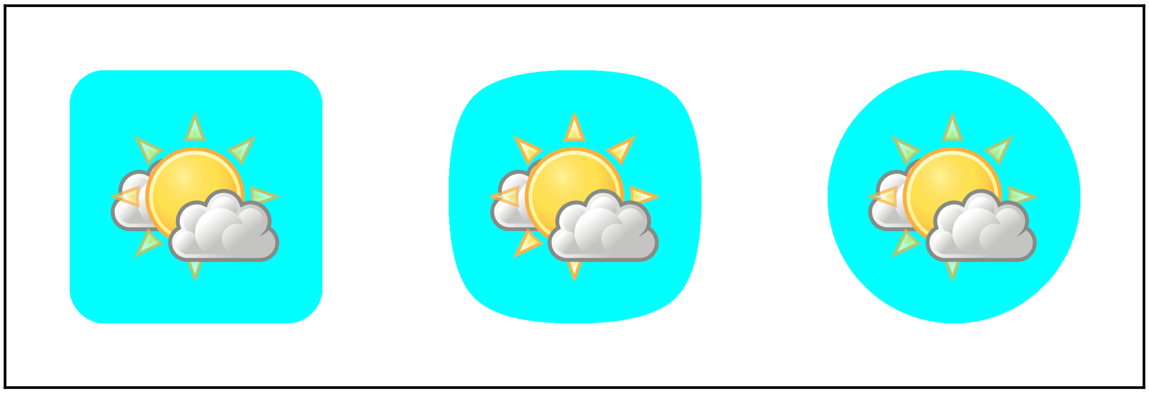
Asset Studio comes with two separate interfaces: one for general images and one for vector graphics. We will look at the first of these in the next section.

Image Asset Studio

When creating images for different screen configurations, we often have to create several versions of the same image and this is usually not a great deal of work. When it comes to icons, on the other hand, we may have several individual icons and dozens of versions, making resizing and scaling them a tedious process. Fortunately, Android Studio provides a neat solution to this in the form of Image Asset Studio.

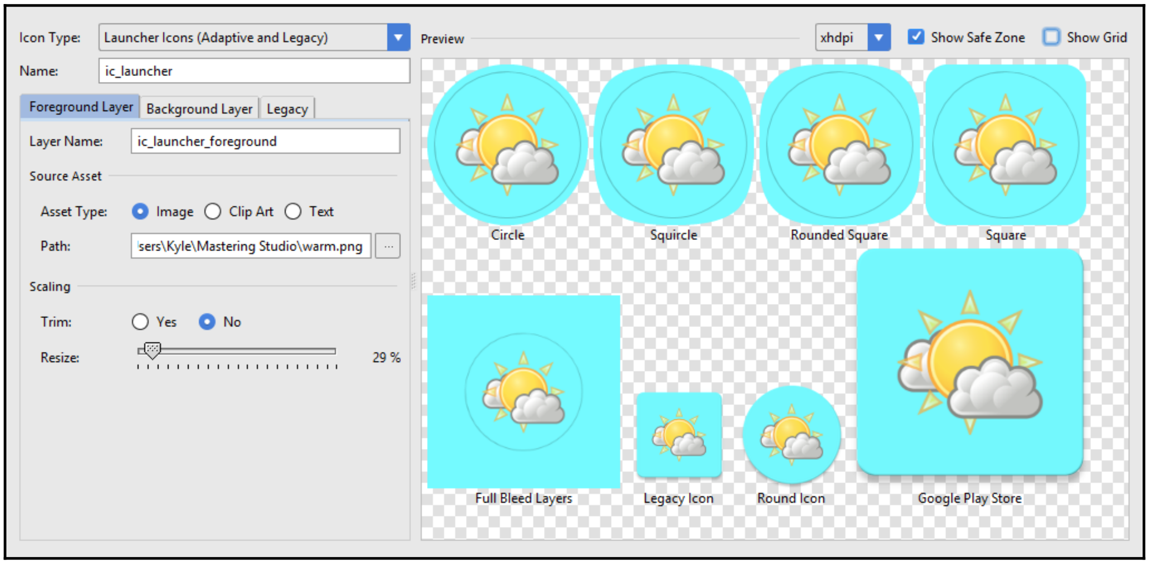
Device manufacturers are perhaps even more concerned with creating a consistent look and feel across their models. This is particularly apparent when it comes to the way launcher icons are displayed on their home screens. An ideal situation would be if developers could design a single icon and manufacturers could then fit that into a uniform shape, such as a square or circle, depending on its location on a device and the manufacturer's own design ideals.

Image Asset Studio achieves this by creating a two-layered icon that uses our original imagery and a plain background layer that a mask can be applied to in order to create the desired overall shape, often one of the three following images:



Adaptive icons

The Image Asset Studio can be opened by selecting **New** | **Image Asset** from your project's **drawable** context menu:



Icon Type: Launcher Icons (Adaptive and Legacy)

Name: ic_launcher

Foreground Layer | Background Layer | Legacy

Layer Name: ic_launcher_foreground

Source Asset

Asset Type: [✓ Image] Clip Art Text

Path: sers\Kyle\Mastering Studio\warm.png

Scaling

Trim: Yes [✓ No]

Resize: 29 %

Preview xhdpi [✓ Show Safe Zone] Show Grid

Asset Studio

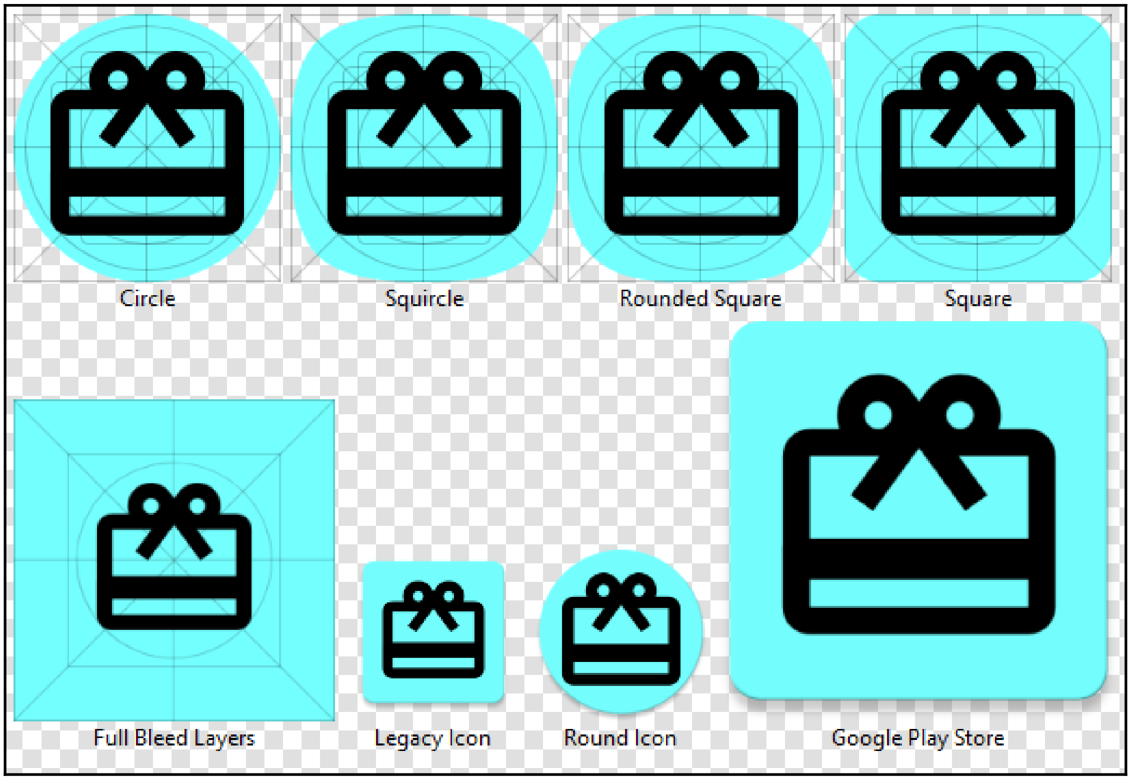
There are several stages to create icons that will work across the widest range of devices and API levels, and these stages are represented by the following three tabs in the wizard: **Foreground Layer**, **Background Layer**, and **Legacy**. There are some valuable features included in each of these tabs, which will be outlined in the next section.

Layered icons

The foreground layer is where we apply our imagery. This can be our own artwork, in the case of a launcher icon, or clip art/text, if we are creating action icons. The wizard automatically generates an icon for each possible usage, including a Play Store icon, which involves creating an entirely new asset. The **Show Safe Zone** feature is, without doubt, the most useful of the preview features, as it displays a bounding circle that our asset should not extend beyond if our icon is to display correctly on all devices and platforms. The **Resize:** control allows us to quickly ensure that none of our icons extend beyond this zone.

Selecting **Trim:** as a scaling option will remove any excess pixels before creating the finished icon, meaning that redundant transparent pixels are removed from the top layer, often significantly reducing its file size.

The background layer of an adaptive icon needs to be large enough to allow any trimming required to create the shapes and sizes shown in the preceding image. The default `ic_launcher_background.xml` produces a vector graphic describing a grid. This is very helpful when it comes to positioning and sizing our artwork, but it is not intended for use in a completed application. Google recommends that you use plain backgrounds with no borders or external shadows and, although Material guidelines allow some internal shading, the simplest solution is to use a color rather than an image for the background layer. This also allows us to select a prominent color from our theme, further promoting our brand.

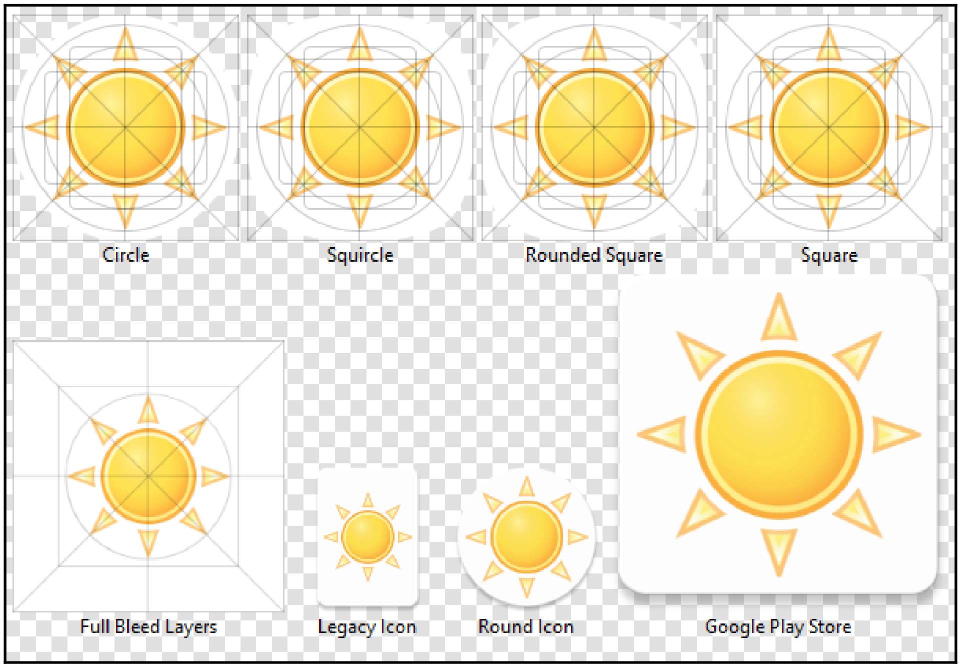


Asset background selection

The preceding image uses an icon from the clip art selection, which demonstrates nicely the purpose of the guidelines when it comes to designing our own.

The source image can only be selected when editing the foreground layer, regardless of the tab you are working on.

The legacy tab allows us to ensure that our icons will still work on devices that run API level 25 and lower and provides us with all the design features that devices running these earlier versions need, such as the elongated rectangular icon that suited many of these devices.



Editing legacy icons

Many developers are also accomplished artists and will be more than comfortable to design launcher icons from scratch. For these readers, it is important to know that the specified dimensions of launcher icons have changed since the inception of API level 26. Although icons had been designed for a `48 x 48 px` grid, they must now be `108 x 108 px`, with the central `72 x 72 px` representing the portion that must remain visible at all times. However, there is no guarantee what manufacturers in the future may do with these guidelines and, as always, it is advisable to test all assets against as many real devices as possible.

TIP

The guidelines given here are not only useful to ensure our imagery is not unnecessarily clipped, but also to cater for the pulse and jiggle animations now included by many manufacturers. These are often used to indicate the success or failure of an attempted user interaction.

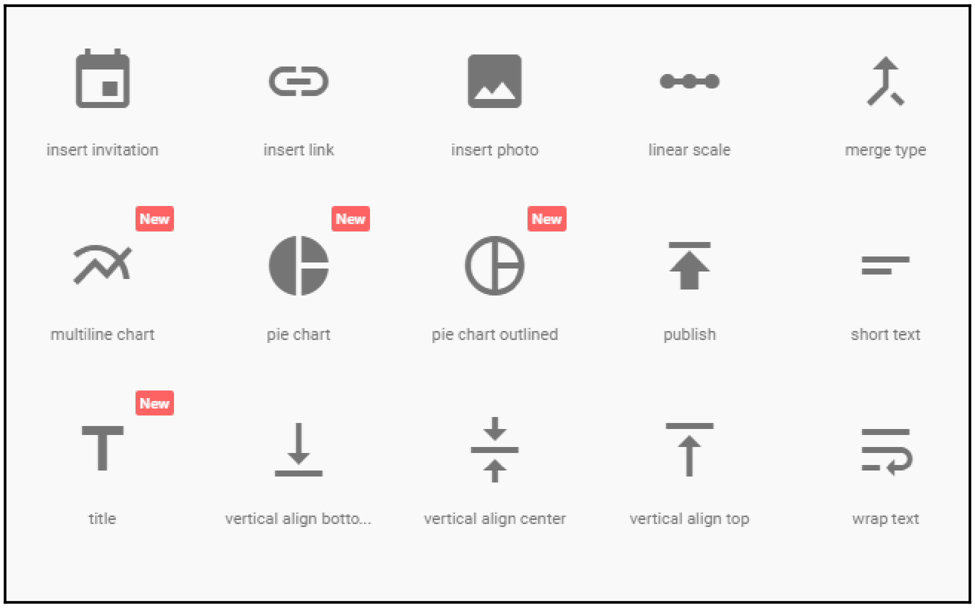
It is not, of course, strictly necessary to use the built-in asset studio to create adaptive icons, and once the basics are grasped, we can, of course, design and include our own directly as an XML. This can be done using the `android:roundIcon` identifier in a manifest file, as follows:

```
<application
 . . .
     android:roundIcon="@mipmap/ic_launcher_round"
 . . . >
</application>
```

Adaptive icons can then be added to any XML layout using the `adaptive-icon` attribute, as follows:

```
<adaptive-icon>
     <background android:drawable="@color/ic_some_background"/>
     <foreground android:drawable="@mipmap/ic_some_foreground"/>
</adaptive-icon>
```

Although the set of included action icons is comprehensive, it is always good to have as much choice as possible, and a much larger and constantly updated collection can be found at `material.io/icons/`.



Material icons

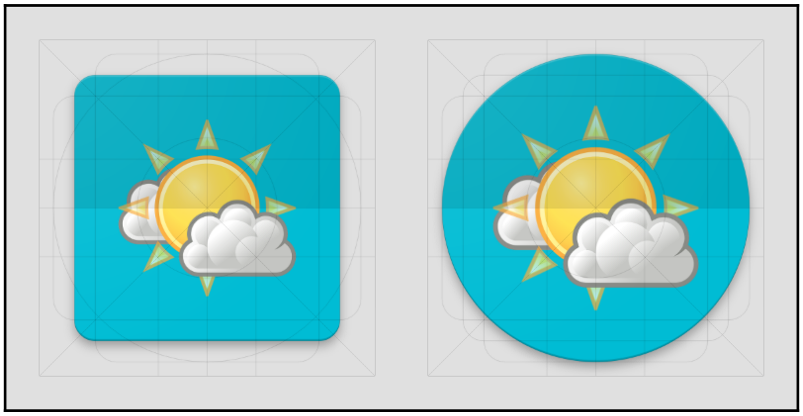
The Image Asset Studio is great for generating the small, in-app icons that we use on tabs, action bars, and so on, but it is limited when it comes to launcher icons, which should be bright and colorful and, in material terms, 3D. For this reason, launcher icons deserve a small section of their own.

Launcher icon tools

Generally speaking, launcher icons are created using an external editor and, as we shall see, there are Studio plugins to assist us in creating stylish Android icons. One of the best tools is an online, alternative, and enhanced version of Asset Studio itself. It was created by the Google designer Roman Nurik and can be found on GitHub at `romannurik.github.io/AndroidAssetStudio`.

This online version offers over half a dozen different icon generators, including features not included in the native version as well as a neat icon animator. The launcher icon generator is of interest here, as it allows us to set material features not offered in the IDE such as elevation, shadow, and scoring.

One of the best things about this editor is the way it displays the material design icon keylines.



Launcher icon keylines

The design of what Google call *product* icons is beyond the scope of this book, but Google has some very interesting guidelines on the matter, which can be found at `material.io/guidelines/style/icons`.

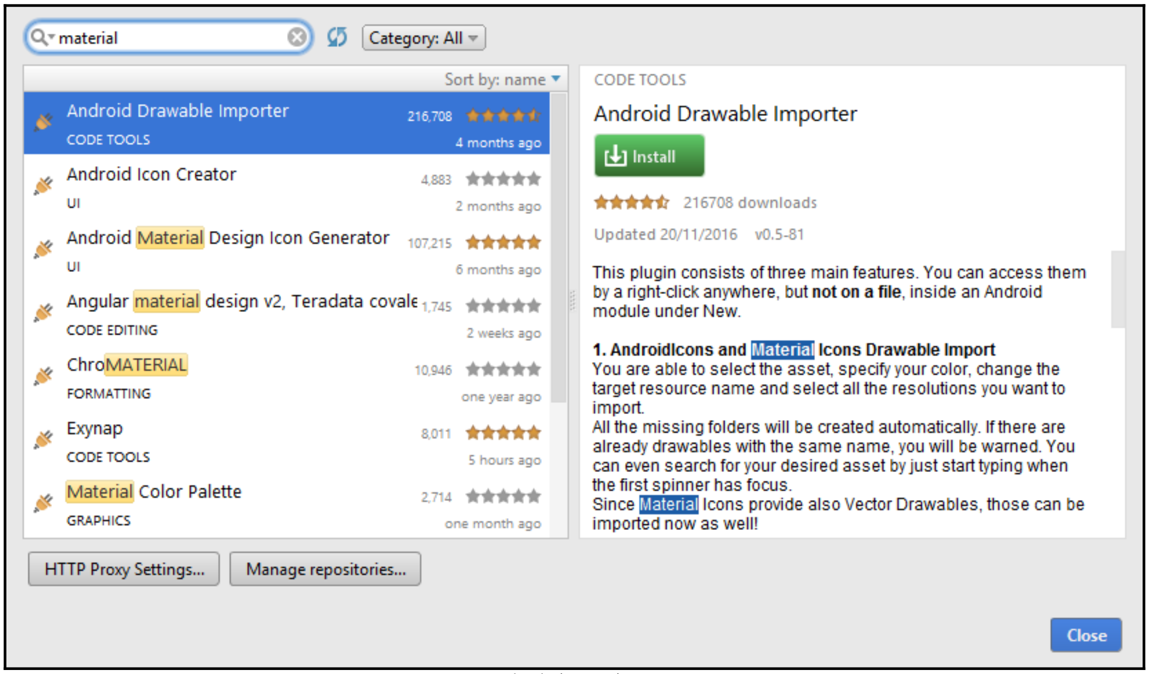
However, when you configure your launcher icons, you will need some kind of external graphics editor at some point. There are some tools that can help us integrate Android Studio with these editors.

The Android Material Design Icon Generator is a fantastic plugin from JetBrains and does precisely what its title suggests. It does not need to be downloaded, as it can be found in the plugin repository. If you want to use it with another IDE, it can be downloaded from the following URL:

`github.com/konifar/android-material-design-icon-generator-plugin`

If you are new to Android Studio plugins, perform the following simple steps:

1. Open the settings dialog from **File | Settings...**.
2. Open the **Plugins** dialog and click on **Browse repositories...**.
3. Type **Material** into the search box and select and install the plugin.



Android Drawable Importer	CODE TOOLS	216,708	4 months ago
Android Icon Creator	UI	4,883	2 months ago
Android Material Design Icon Generator	UI	107,215	6 months ago
Angular material design v2, Teradata covale	CODE EDITING	1,745	2 weeks ago
ChroMATERIAL	FORMATTING	10,946	one year ago
Exynap	CODE TOOLS	8,011	5 hours ago
Material Color Palette	GRAPHICS	2,714	one month ago

CODE TOOLS

Android Drawable Importer

216708 downloads

Updated 20/11/2016 v0.5-81

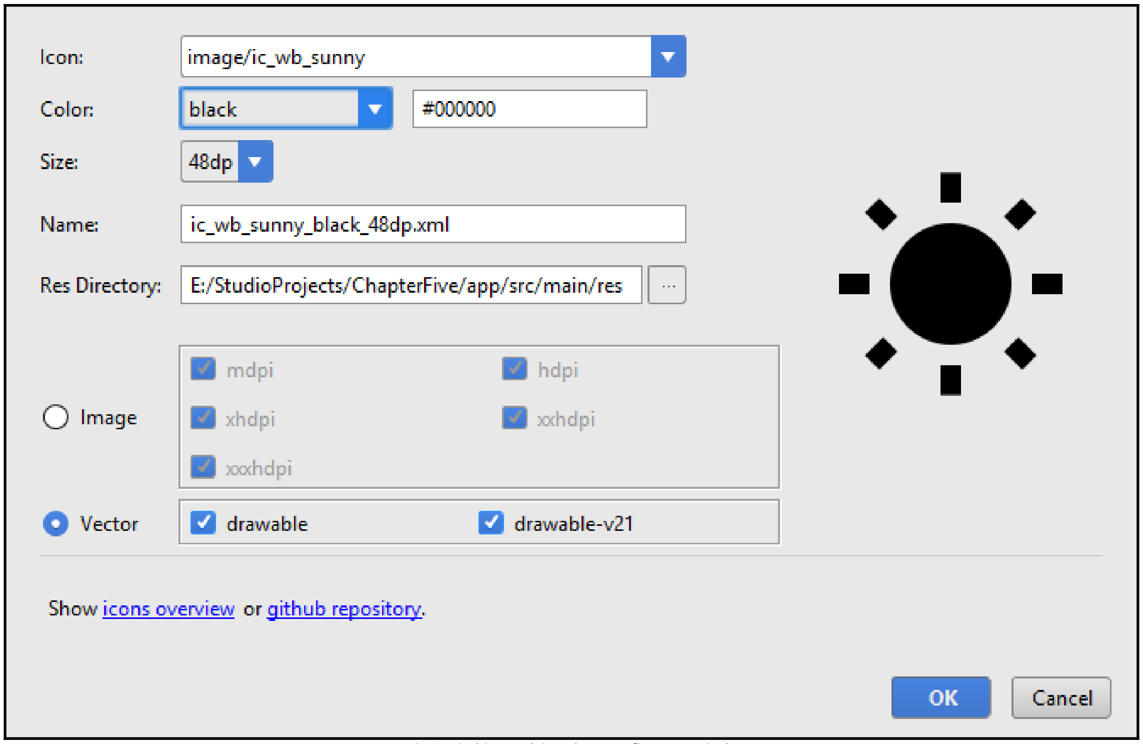
This plugin consists of three main features. You can access them by a right-click anywhere, but **not on a file**, inside an Android module under New.

1. AndroidIcons and Material Icons Drawable Import
You are able to select the asset, specify your color, change the target resource name and select all the resolutions you want to import.
All the missing folders will be created automatically. If there are already drawables with the same name, you will be warned. You can even search for your desired asset by just start typing when the first spinner has focus.
Since Material Icons provide also Vector Drawables, those can be imported now as well!

The plugins repository

4. Restart Android Studio.

The plugin can now be opened from most **New...** submenus or with *Ctrl* + *Alt* + *M*. The icon generator is simple, but offers all the important functions, such as being able to create both bitmap and vector images and a choice of all density groupings, as well as color and size selectors.



Icon: image/ic_wb_sunny
Color: black #000000
Size: 48dp
Name: ic_wb_sunny_black_48dp.xml
Res Directory: E:/StudioProjects/ChapterFive/app/src/main/res

Image: mdpi ✓, hdpi ✓, xhdpi ✓, xxhdpi ✓, xxxhdpi ✓
[✓ Vector]: drawable ✓, drawable-v21 ✓

Show icons overview or github repository.

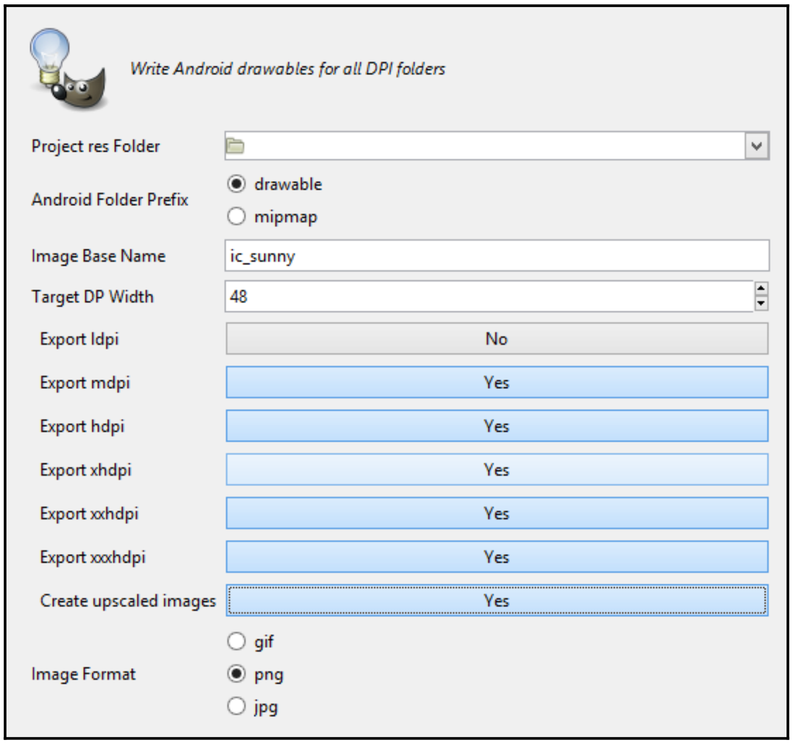
The Android Material Design Icon Generator plugin

The icon generator also has a handy link to the ever growing GitHub material design icon repository.

Sympli is a sophisticated, but expensive, design tool that works with the graphics editor of your choice and Android Studio via a Studio plugin. It generates icons and other assets automatically and is designed to be used among teams. It can be found at `sympli.io`.

Although not a Studio plugin as such, there is a handy Python script on GitHub that GIMP users can find at `github.com/ncornette/gimp-android-xdpi`.

Simply download the script and save it in your GIMP `plug-ins` folder as `gimpfu_android_xdpi.py`. It can then be accessed from the image's Filter menu.



Write Android drawables for all DPI folders

Project res Folder	
Android Folder Prefix	● drawable ○ mipmap
Image Base Name	ic_sunny
Target DP Width	48
Export ldpi	No
Export mdpi	Yes
Export hdpi	Yes
Export xhdpi	Yes
Export xxhdpi	Yes
Export xxxhdpi	Yes
Create upscaled images	Yes
Image Format	○ gif ● png ○ jpg

Automatic icon generation

As you can see in the preceding screenshot, this plugin provides all the main choices we need to make when converting a single image into a set of icons.

Being able to create and configure icons using these tools is useful and time saving, but there are many times when we will not use bitmaps for our icons at all, and instead use vector graphics, which only require an image for all densities.

TIP

Vector graphics load more slowly than raster images, but, once loaded, are a little faster. Very large vector images load slowly, so they should be avoided.

Vector drawables are cached as correctly sized bitmaps at runtime. If you want to display the same drawable at different sizes, create a vector graphic for each.

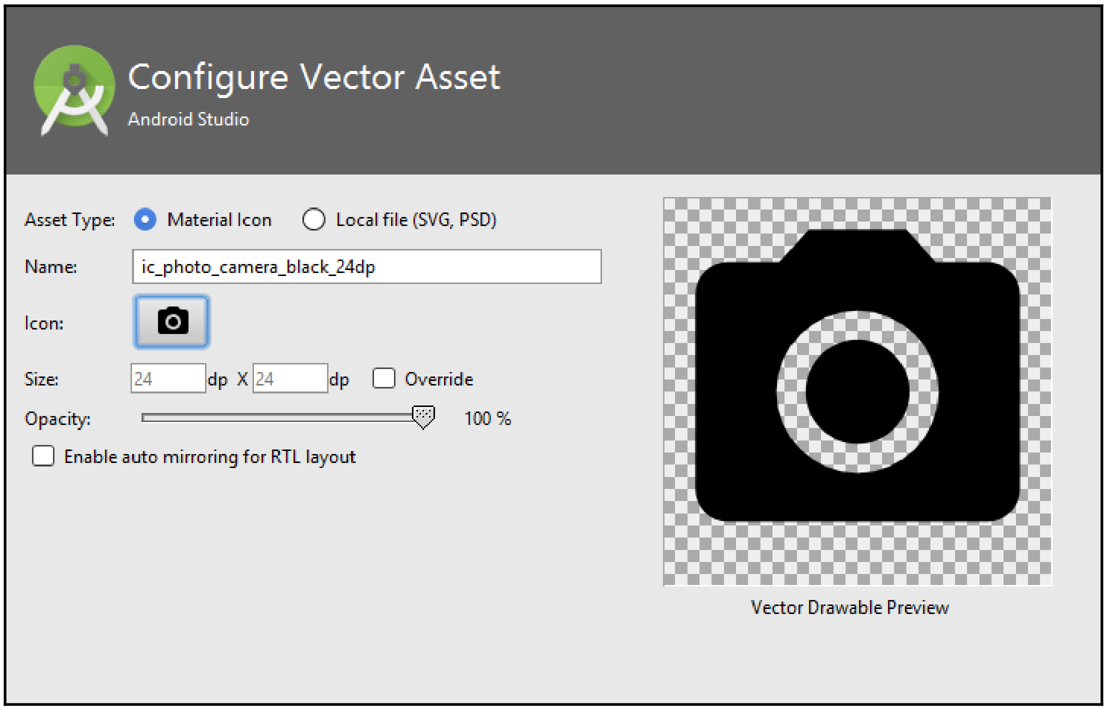
For those who fancy creating vector images from scratch, there are some very useful free tools.

Method Draw is an online **Scaleable Vector Graphics** (**SVG**) editor that offers a simple, but very functional, set of tools to generate simple vector images, such as those we want for our action and notification icons. Creations can be downloaded as `.svg` files and imported directly into Studio. It can be found at `editor.method.ac`.

If you want a more sophisticated tool, Boxy SVG Editor is available on the Chrome Web Store, but it works offline and offers features similar to packages such as Inkscape or Sketch.

Vector Asset Studio

The vector graphics asset studio performs the same function as the raster graphics version, but it is more fun to work with. When dealing with preset icons, it is even simpler to use a sibling that requires nothing more than the selection of the material icon.



Configure Vector Asset
Android Studio

Asset Type: [● Material Icon] ○ Local file (SVG, PSD)
Name: ic_photo_camera_black_24dp
Icon:
Size: 24 dp X 24 dp ☐ Override
Opacity: 100 %
☐ Enable auto mirroring for RTL layout

Vector Asset Studio

Once created, an asset like this is saved in XML as a `VectorDrawable` class:

```
<vector xmlns:android="http://schemas.android.com/apk/res/android"
    android:width="24dp"
    android:height="24dp"
    android:viewportHeight="24.0"
    android:viewportWidth="24.0">

    <path
        android:fillColor="#FF000000"
        android:pathData="M19,13h-6v6h-2v-6H5v-2h6V5h2v6h6v2z" />

</vector>
```

Android vector drawables are a similar, and somewhat simplified, version of the SVG format, familiarly associated with `.svg` files. As with raster assets, it is very easy to use existing icons. Only when we want to modify these or create our own does it become interesting.

Of course, it is not necessary to learn SVG or even understand the `pathData` of a `VectorDrawable`, but it is good to understand a little of the process and some of the tools at our disposal.

Vector drawables

The vector studio allows us to import SVG files and convert them into VectorDrawables. There are many ways to obtain vector graphics, and many graphic editors can convert from other formats. There are also some very good online tools to convert other formats to SVG:

`image.online-convert.com/convert-to-svg`

And JetBrains plugin is also available from:

`plugins.jetbrains.com/plugin/8103-svg2vectordrawable`

It is unlikely that you will do much when you write your own SVG objects, but it is useful to see how the process operates, as these steps demonstrate:

1. Save the following code as a `.svg` file:

```
<svg
    height="210"
    width="210">
```

```
<polygon
    points="100,10 40,198 190,78 10,78 160,198"
    style="fill:black;fill-rule:nonzero;"/>

</svg>
```

2. Open an Android Studio project and then navigate to the vector studio.
3. Select **Local File** and then the SVG file created in the preceding code.
4. Click on **Next** and **Finish** to convert to the following `VectorDrawable`:

```
<vector
xmlns:android="http://schemas.android.com/apk/res/android"
    android:width="24dp"
    android:height="24dp"
    android:viewportHeight="210"
    android:viewportWidth="210">

    <path
        android:fillColor="#000000"
        android:pathData="M100,10l-60,188l150,
                -120l-180,0l150,120z" />

</vector>
```

It is usually a good idea to color vector icons black and color them using the `tint` property. This way, one icon can be reused with different themes.

The SVG `<polygon>` is easy to understand, as it is a simple list of points defining the corners of the shape. The `android:pathData` string, on the other hand, is a little more cryptic. It is most easily explained as follows:

- `M` is move
  `100,10`
- `l` line to
  `-60,188`
- `l` line to
  `150,-120`
- `l` line to
  `-180,0`
- `l` line to
  `150,120 z`
  `(end path)`

The preceding format uses caps to indicate absolute positions and lowercase to indicate relative ones. We can also create vertical and horizontal lines with `V(v)` and `H(h)`, respectively.

It is actually not necessary to include the final coordinate if the path end qualifier, z, is provided. Also, a character can be omitted if it is the same as the one before, as is the case with the `line-to` command previousl; consider the following string:

```
M100,10l-60,188l150,-120l-180,0l150,120z
```

The preceding string could be written as follows:

```
M100,10 l-60,188 150,-120 -180,0z
```

Note that there are two sets of image dimensions, as you might expect-- `viewportWidth` and `viewportHeight`; that refer to the canvas size of the original SVG image.

It may seem unnecessary to concern ourselves with the vector data itself, as this is generated by the asset studio; but, as we shall see next, when it comes to animated icons (as well as other animated vector graphics), an understanding of the inner structure of a vector drawable can be very useful.

Animated icons

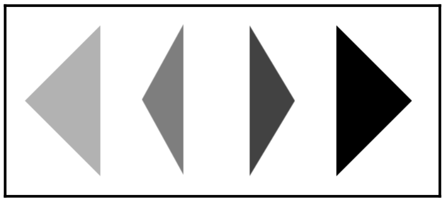
Everyone with an Android device will be familiar with the animated icon. Perhaps the best known example is the way the hamburger icon transforms into an arrow and vice versa when a navigation drawer is opened and closed. The use of vector graphics makes this process remarkably simple. Provided that both the icons have the same number of points, any icon can be transformed into any other.

Using space efficiently is essential on a mobile device, and animating action icons not only looks good, but also saves space and, if applied intelligently, will convey meaning to the user as well.

Vector images are easily transformed from one to another by mapping points on the original image onto the target image. This is done with the `AnimatedVectorDrawable` class.

There are several methods of animating these drawables. Firstly, we can apply a number of predefined animations, such as rotation and translation. We can also use built-in interpolation techniques to *morph* from one drawable to another, regardless of the number of points. We will take a look at both of these techniques. However, first, we will examine how to use the image paths to control the animation, as this gives us the most control.

The following image represents an arrow icon animating from pointing left, to pointing right:



An animated arrow icon.

The following steps demonstrate how to create such an animated vector drawable.

1. Begin by storing the paths to the two arrows as strings, as follows:

```
<!-- Spaces added for clarity only -->
<string name="arrow_right">
    M50,10 l40,40 l-40,40 l0,-80z
</string>
<string name="arrow_left">
    M50,10 l-40,40 l40,40 l0,-80z
</string>
```

2. As both paths are recorded as strings, we only need to define one vector drawable--call it `ic_arrow_left.xml`:

```
<vector
xmlns:android="http://schemas.android.com/apk/res/android"
    android:width="24dp"
    android:height="24dp"
    android:viewportHeight="100.0"
    android:viewportWidth="100.0">

    <path
        android:name="path_left"
        android:fillColor="#000000"
        android:pathData="@string/arrow_left" />

</vector>
```

3. Create the `res/animator` folder and the `arrow_animation.xml` file, inside it:

```
<?xml version="1.0" encoding="utf-8"?>
<set
xmlns:android="http://schemas.android.com/apk/res/android">

    <objectAnimator
        android:duration="5000"
        android:propertyName="pathData"
        android:repeatCount="-1"
        android:repeatMode="reverse"
        android:valueFrom="@string/arrow_left"
        android:valueTo="@string/arrow_right"
        android:valueType="pathType" />

</set>
```

4. We can use this to create our animated drawable, `ic_arrow_animated.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<animated-vector
xmlns:android="http://schemas.android.com/apk/res/android"
    android:drawable="@drawable/ic_arrow_left">

    <target
        android:name="path_left"
        android:animation="@animator/arrow_animation" />

</animated-vector>
```

5. To see this in action, use the following Java snippet:

```
ImageView imageView = (ImageView)
findViewById(R.id.image_arrow);
Drawable drawable = imageView.getDrawable();

if (drawable instanceof Animatable) {
    ((Animatable) drawable).start();
}
```

TIP

By animating a vector's path, we can easily create new animations by reordering our points.

The key to this process is the `ObjectAnimator` class in the `arrow_animation` file. This class is far more powerful than it might seem here. In this example, we selected the `pathData` property to animate, but we could have animated almost any property we choose. In fact, any numerical property, including colors, can be animated this way.

The object animator provides an opportunity to create imaginative new animations, but only for extant properties. However, what if we want to animate a value that we defined or, perhaps, a variable, reflecting some app-specific data? In these circumstances, we can take advantage of the ValueAnimator, from which the ObjectAnimator is descended.

Roman Nurik's online asset studio also has a powerful and easy-to-use animated icon generator , which can be found at:
`romannurik.github.io/AndroidIconAnimator`

Using path data, this way offers a very flexible animation framework, particularly when we want to morph one icon into another, as it changes its function, as is often seen with toggle action such as play/pause. However, this is not our only option, as there are ready-made animations that we can apply to our vector assets and ways to transform icons into others that do not share the same number of points.

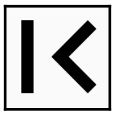
Other animations

Morphing path data is one of the the most fun ways to animate icons (and other drawables), but sometimes we just need a simple symmetrical motion, such as rotation and translation.

The following example demonstrates how to apply one of these animation types:

1. Select a vector drawable of your choice and save its `pathData` as a string. Here, we have taken the data from the asset studio using the `ic_first_page_black_24dp` icon:

```
<string name="first_page">
    M18.41,16.59 L13.82,12 l4.59,-4.59 L17,6 l-6,6 6,6 z
            M6,6 h2 v12 H6 z
</string>
```



the ic_first_page_black_24dp icon

2. As before, create an XML asset for this; here, we will call it `ic_first_page.xml`:

```
<vector
xmlns:android="http://schemas.android.com/apk/res/android"
    android:height="24dp"
    android:width="24dp"
    android:viewportHeight="24"
    android:viewportWidth="24" >
    <group
        android:name="rotation_group"
        android:pivotX="12.0"
        android:pivotY="12.0" >
        <path
            android:name="page"
            android:fillColor="#000000"
            android:pathData="@string/first_page" />
    </group>
</vector>
```

3. Once again, create an object animator, call it `rotation.xml` this time, and complete it as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<set
xmlns:android="http://schemas.android.com/apk/res/android">

    <objectAnimator
        android:duration="5000"
        android:propertyName="rotation"
        android:repeatCount="-1"
        android:valueFrom="0"
        android:valueTo="180" />

</set>
```

4. Now, we can create the animated version of the icon, as we did before, by setting a target. Here, the file is called `ic_animated_page.xml`, and it looks like this:

```
<?xml version="1.0" encoding="utf-8"?>
<animated-vector
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:drawable="@drawable/ic_first_page">

    <target
        android:name="rotation_group"
        android:animation="@animator/rotation" />
```

```
</animated-vector>
```

5. The animation can be called by first adding it to our layout, as we would do with any other icon, and calling it from code like this:

```
ImageView imagePage = (ImageView)
findViewById(R.id.image_page);
Drawable page_drawable = imagePage.getDrawable();

if (page_drawable instanceof Animatable) {
    ((Animatable) page_drawable).start();
}
```

The biggest difference here, apart from the animation type, is the inclusion of our `<path>` within a `<group>`. This is normally used for when there is more than one target, but, in this case, it is because it allows us to set a pivot point for the rotation with `vectorX/Y`. It also has equivalent settings for `scaleX/Y`, `translateX/Y`, and `rotate`.

TIP

To change an icon's transparency, set `alpha` in `<vector>`.

Having to build a project to test simple graphical features, such as these animated icons, can be very time-consuming. Jimu Mirror is a layout preview plugin that displays animations and other moving components. It connects via a device or emulator, and through a sophisticated hot-swapping process, layouts can be edited and retested within seconds. Jimu is not open source, but is not overly expensive and is available on a free trial. It can be downloaded from `www.jimumirror.com`.

The focus of this chapter is primarily to examine how Android Studio and associated tools can facilitate the generation of application icons. This has led us to take a look at Android drawables in general, both bitmaps and vector graphics. We explored other drawables briefly, earlier in the book, and now that we looked more deeply into the matter, now is a good time to revisit these drawables.

General drawables

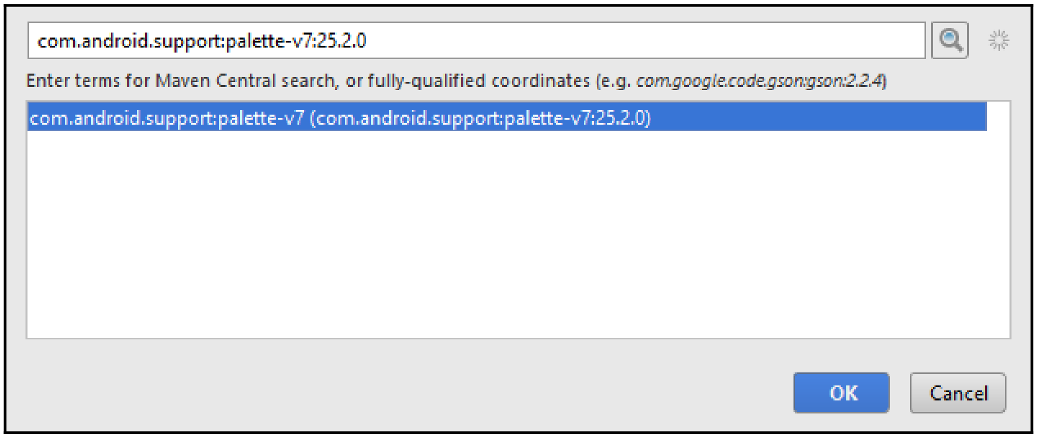
We saw earlier how to convert a black icon into a color to match our app or current activity using tinting. With other images, there are times when they take up a considerable amount of the screen, and we want to apply the reverse and have our icons colored so that they match our graphics. Fortunately, Android provides a support library to extract prominent and dominant colors from any bitmap.

The palette library

Applying our own themes to our apps can produce very stylish-looking interfaces, especially when we are dealing with text, icons, and images we created ourselves to suit the app. Many apps incorporate the user's own images and, in these cases, there is no way of knowing in advance how to select a pleasing design. The **palette support library** provides us with this functionality, allowing fine control over text, icon, and background coloring.

The following steps demonstrate how to extract the prominent colors from a bitmap drawable:

1. Start a new Android Studio project and open the **Project Structure** dialog from the `File` menu or *Ctrl* + *Alt* + *Shift* + *S*.
2. Open the **Dependency** tab from your **app Module** and add a **Library dependency** from the **+** icon in the top-right corner, using the search tool to find the library.


com.android.support:palette-v7:25.2.0

Enter terms for Maven Central search, or fully-qualified coordinates (e.g. *com.google.code.gson:gson:2.2.4*)

com.android.support:palette-v7 (com.android.support:palette-v7:25.2.0)

Library dependency selector

3. This will add the following line to your `build.gradle` file:

```
compile 'com.android.support:palette-v7:25.2.0'
```

4. Create a layout with a large image view and at least two text views. Call these text views `text_view_vibrant` and `text_view_muted`.
5. Open your main Java activity and add the following fields:

```
private Palette palette;
private Bitmap bmp;
private TextView textViewVibrant;
private TextView textViewMuted;
```

6. Associate the preceding `TextViews` with their XML counterparts, as follows:

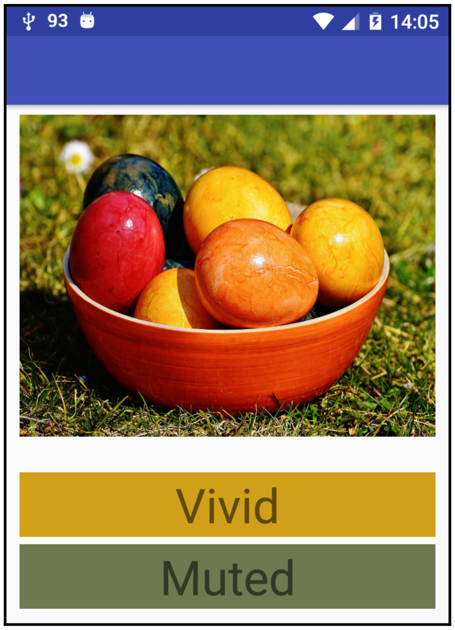
```
textViewVibrant = (TextView)
        findViewById(R.id.text_view_vibrant);

textViewMuted = (TextView)
        findViewById(R.id.text_view_muted);
```

7. Assign the bitmap declared in step 5:

```
 bmp = BitmapFactory.decodeResource(getResources(),
        R.drawable.color_photo);
```

8. Finally, add the following clause to extract prominent vivid and muted colors from the image:

```
// Make sure object exists.
if (bmp != null && !bmp.isRecycled()) {
    palette = Palette.from(bmp).generate();

    // Select default color (black) for failed scans.
    int default_color=000000;

    // Assign colors if found.
    int vibrant = palette.getVibrantColor(default_color);
    int muted = palette.getMutedColor(default_color);

    // Apply colors.
    textViewVibrant.setBackgroundColor(vibrant);
    textViewMuted.setBackgroundColor(muted);
}
```



Extracted colors

The preceding method outlined is effective but crude. There is a lot more that can be done with the palette library, and we need to know quite a few things to be able to take best advantage of it.

The use of a `default_color` by the palette is needed, as the extraction of these colors is not always possible and sometimes fails. This often happens with *washed out* images with very few colors and also with highly irregular images with little definition. Somewhat ironically, the scan can also fail when presented with over-saturated graphics with many colors and with very regular patterns where no color, if any, dominates.

One very important point when extracting these palettes is that working with large bitmaps can present a serious drain on device resources and all work with bitmaps should not, where possible, be performed on the current thread. The preceding example took no account of this, but the library has a listener class that allows us to perform these tasks asynchronously.

Consider the following example:

```
Palette palette = Palette.from(bmp).generate();
```

Use the following listener, instead of the preceding one, to react once the bitmap is generated:

```
Palette.from(bmp).generate(new PaletteAsyncListener() {

    public void onGenerated(Palette p) {
        // Retrieve palette here.

    }

});
```

In the preceding example, we extracted just two colors, using `Palette.getVibrantColor()` and `Palette.getMutedColor()`. These often suit our purposes very well, but if they do not, there are lighter and darker versions of each, and these can be accessed using getters, such as `getDarkVibrantColor()` or `getLightMutedColor()`.

The palette library has more features than we have space for here, such as being able to select text coloring to match analyzed images, and as it is not exclusive to Android Studio, it is likely that readers switching from other IDEs will already be familiar with it.

The Studio features we have covered in this book show how useful the IDE is when it comes to developing layouts and UIs, but, of course, this is just half the story. No matter how well put together our layouts are, they are as good as useless without logic behind them, and this is where Android Studio really starts to come into its own.

Summary

Not only in this chapter, but also in the previous three chapters, we saw how Android Studio makes the designing and testing of our graphical layouts over a wide range of devices and factors both simple and intuitive. Having been specifically designed for Android's eccentricities, Studio is also the first to integrate new design features, such as the constraint layout, which has revolutionized designing of visual activities.

The chapters done till now have covered all the fundamental design considerations catered for by the IDE and hopefully introduced the reader to the wealth of features that simplify and clarify this often complex process.

In the next chapter, we will begin the process of bringing these designs to life as we see how Android Studio facilitates the often complex processes of coding, testing, and debugging our applications. These essential processes often overlap and most developers will find themselves having to revisit each as they fine-tune their work. Android Studio guides developers through and around this course, enabling them to track and evaluate as they go.

Android Studio has helped you to turn your ideas into delightful layouts. The next step is to bring these layouts to life with your logic. As one might imagine, the IDE is as helpful when it comes to logic as it has been when it was applied to design.

6
Templates and Plugins

As a development environment, Android Studio provides facilities to design and develop all aspects of any Android app we can imagine. In the previous chapters we saw how it acts as a visual design tool, along with a dynamic layout editor, emulators, and XML structures. From now onward, we will delve under the bonnet and take a look at how the IDE facilitates, simplifies, and speeds up the process of coding, testing, and fine-tuning our work.

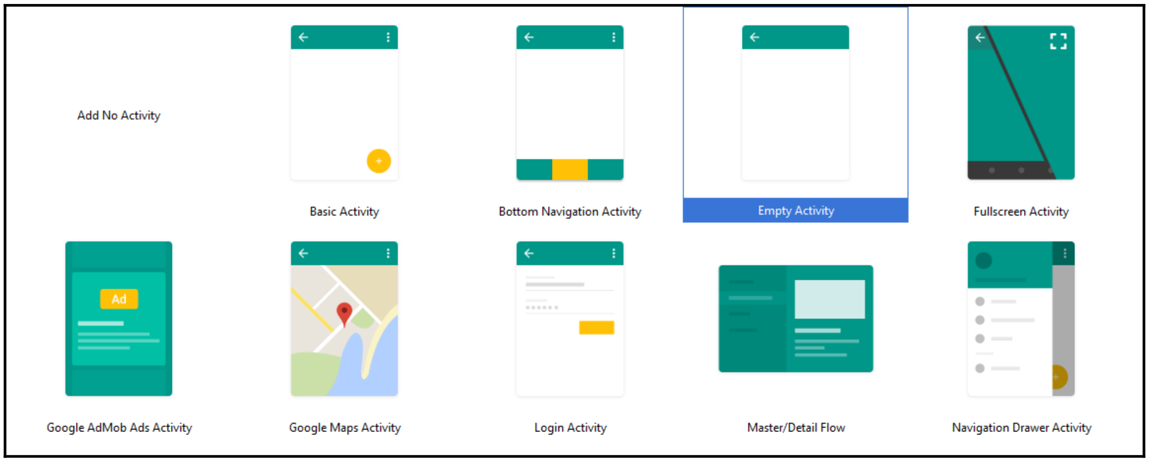
Most readers will already be expert coders and require no help with this. Consequently, it is the way Android Studio improves this experience that we will globally explore in the coming chapters. In this chapter, we will look at various examples of ready-made code that comes with the IDE in the form of activity templates and API samples. These are useful as ways to explore and learn how various components are coded and to speed up the process of coding by providing an already existing starting point. Moreover as we shall see, if the bundled selection of templates is not enough, Android Studio allows us to create our own.

In this chapter, you will learn the following:

- Understanding built-in project templates
- Accessing the Structure Tool window
- Installing and using UML plugins
- Applying simple refactoring
- Applying code templates
- Creating a custom template
- Using project samples

Project templates

Most readers will have already encountered project templates whenever they start a new **Android Studio project** from the IDE's Welcome screen.

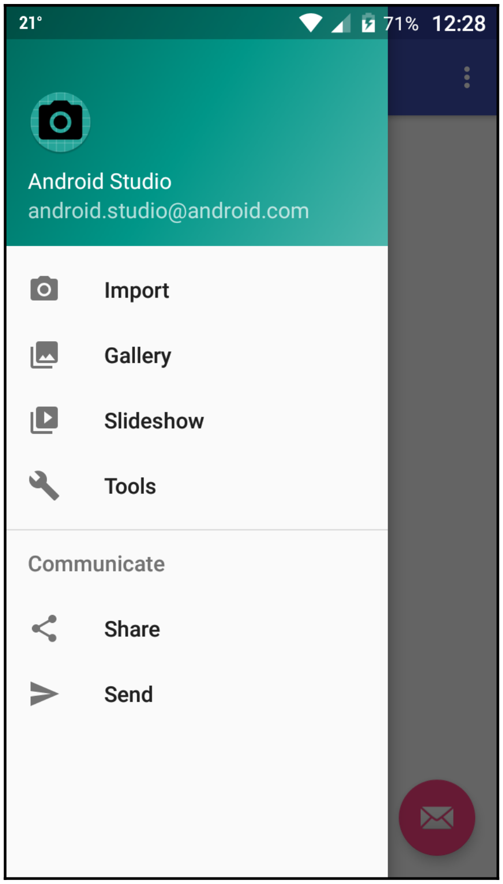


Project templates

Even the **Empty Activity** template provides files and a little code that are essential for almost all applications and, as you saw in the preceding screenshot, there is a growing collection of project templates designed to suit many common application structures and purposes.

The Navigation Drawer template

It will not be necessary to examine all of these templates here, but there are one or two that can be very instructive and provide further insight into the workings of Android Studio. The first, and perhaps the most useful, is the **Navigation Drawer Activity** template, which when run as-is produces the following output:



The Navigation Drawer template

As can be seen in the preceding screenshot, this template provides many common, and recommended, UI components, including icon drawables, menus, and action buttons. The most interesting aspect of these templates is the code that they create and the file structures they use, both of which are nice examples of best practices.

If we start a new project with this template and start to look around, one of the first things we will observe is that the `activity_main.xml` file is different from those we have seen earlier, in that it uses `android.support.v4.widget.DrawerLayout` as its root. Inside this, we find `CoordinatorLayout` and `NavigationView` from the design library. As in the previous example, the coordinator layout contains a toolbar and an FAB; here, though these components are created in a separate file and included with the `include` tag, as follows:

```
<include
    layout="@layout/app_bar_main"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

This structure can help keep our code easier to keep track of and make future modifications easier. As can be seen, this approach is also used to define the navigation bar header and main content.

Most of the template-generated XML will be familiar to the reader, but there are one or two snippets that might require explanation. For example, the `content_main.xml` file contains the following line:

```
app:layout_behavior="@string/appbar_scrolling_view_behavior"
```

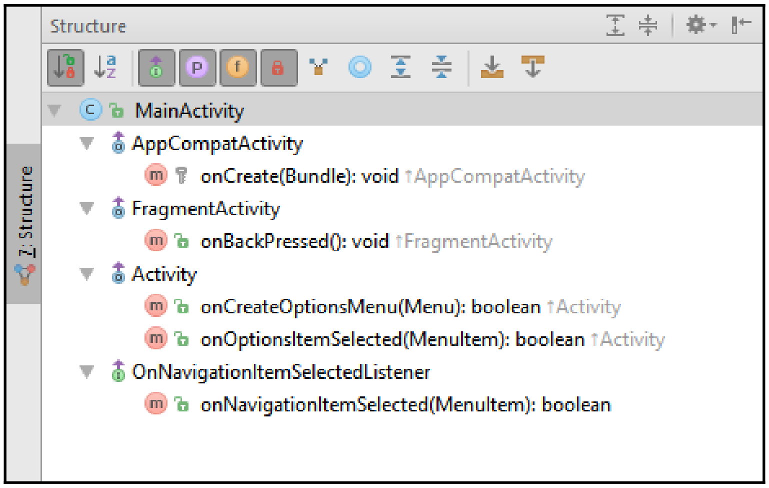
The referenced string will not be found in the `strings.xml` file, as it is system-provided and points to the `AppBarLayout.ScrollingViewBehavior` class, ensuring that our toolbar is an instance of it.

The line `tools:showIn="@layout/app_bar_main"` may also be puzzling. This is one of many useful features found in the `tools` namespace and is used here so that the navigation drawer is visible in the preview editor, saving the developer from having to rebuild a project every time they want to view a graphical change.

The XML resources generated by this and other templates only tell half the story, as many of them produce a sizable amount of Java code as well, and this is just as interesting, if not more so, than the XML. Just a quick look through the `MainActivity.Java` code will demonstrate how the template has set up methods to handle basic navigation, menus, and button clicks. None of the code is complex to understand, but it is very handy as all of these methods would have had to be coded by us at some point or another, and the comments and placeholders make it very simple to substitute our own resources and add our own code.

The Structure Explorer

It is nice when we have time to browse through code this way, but few developers have this luxury and naturally want a way to look quickly at a class's structure and contents. Android Studio provides a neat, schematic window onto a Java class's inner workings in the form of the **Structure Tool window**, which can be selected from the project explorer bar or by pressing *Alt + 7:*



```
MainActivity
  AppCompatActivity
    onCreate(Bundle): void ↑AppCompatActivity
  FragmentActivity
    onBackPressed(): void ↑FragmentActivity
  Activity
    onCreateOptionsMenu(Menu): boolean ↑Activity
    onOptionsItemSelected(MenuItem): boolean ↑Activity
  OnNavigationItemSelectedListener
    onNavigationItemSelected(MenuItem): boolean
```

The **Structure** (*Alt + 7*) Tool window

When dealing with projects having many classes, the structure tool is a very useful way to maintain an overview, and when dealing with lengthy classes, selecting any item will highlight the corresponding text in the code editor.

TIP

Pressing *F4* when an item in the structure explorer is selected will cause the code editor to jump to that location in the text.

The toolbar at the top of the **Structure** pane allows for some very handy filters so that our methods can be displayed according to their defining type (as shown in the preceding figure) and as other levels of detail, such as whether to display properties and fields.

Despite the usefulness of being able to view any class from such a perspective, there are often times when we would like to view class structures from even more points of view, and there are of course plugins that allow deeper inspection.

Class inspection plugins

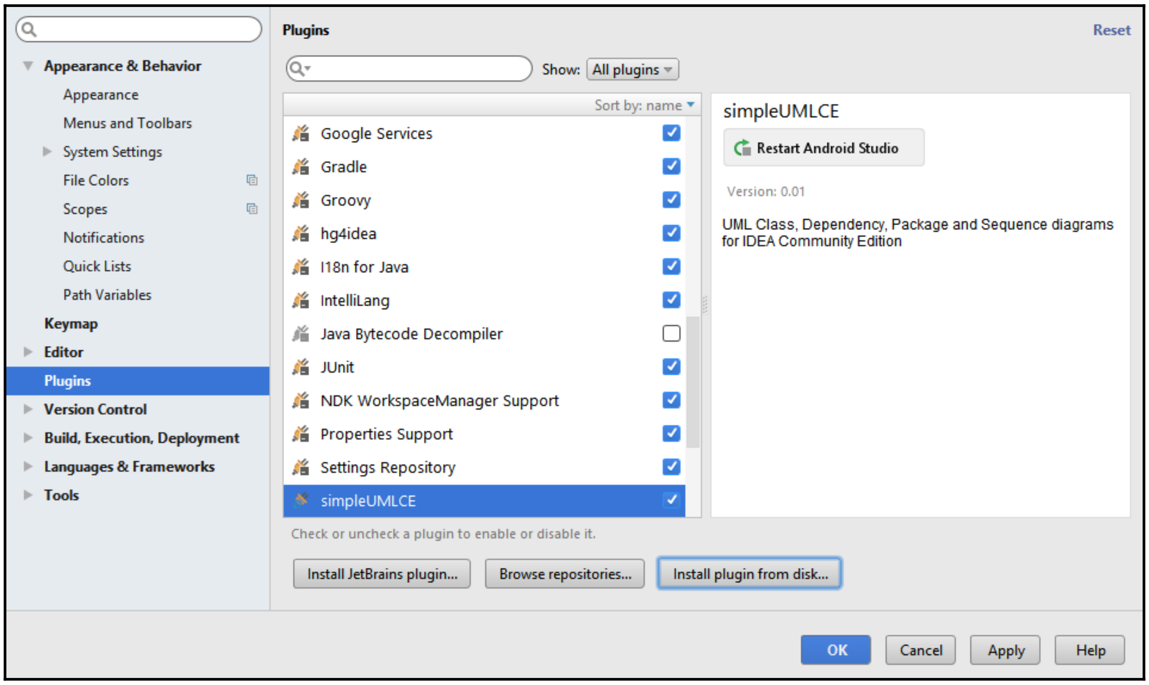
There are many ways to visualize classes and groups of classes, making them easier to follow or to highlight certain features or properties. A tried and tested visual programming tool is the **Universal Modeling Language** (**UML**). These are of particular use if the developer is working with design patterns.

There are several UML class diagram plugins available to us, ranging from basic to sophisticated. If you just want a simple UML tool, then the JetBrains plugin, simpleUMLCE, can be downloaded from the following link:

`plugins.jetbrains.com/plugin/4946-simpleumlce`

If you are new to plugins, follow the quick steps to install and use the simple UML plugin:

1. Download the plugin from the previous link.
2. Open the **Plugins** dialog from Android Studio's **File** | **Settings** menu.
3. Use the **Install plugin from disk...** button to locate and install the plugin.



Google Services	✓
Gradle	✓
Groovy	✓
hg4idea	✓
I18n for Java	✓
IntelliLang	✓
Java Bytecode Decompiler	☐
JUnit	✓
NDK WorkspaceManager Support	✓
Properties Support	✓
Settings Repository	✓
simpleUMLCE	✓

Check or uncheck a plugin to enable or disable it.

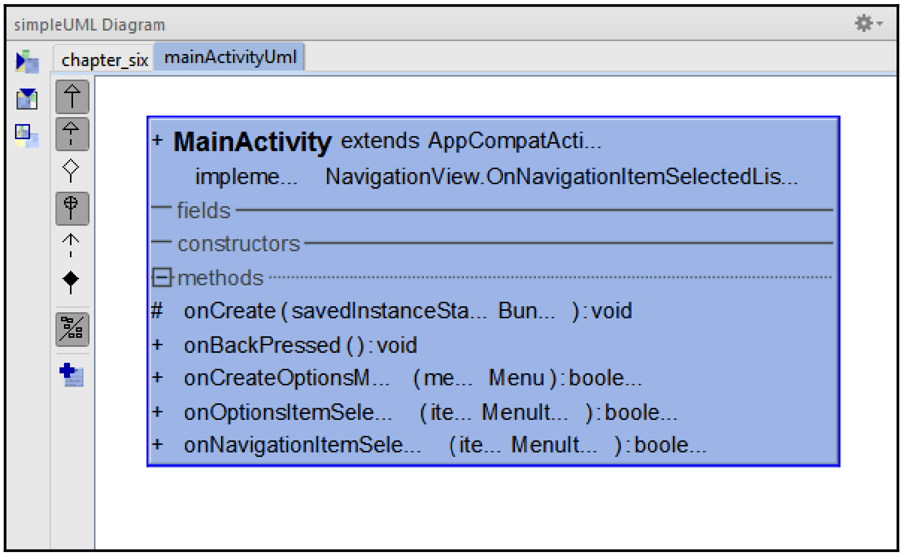
simpleUMLCE

Version: 0.01

UML Class, Dependency, Package and Sequence diagrams for IDEA Community Edition

The Plugins dialog

You will need to restart the IDE before you can access the plugin. Once you have done so, right-click on a package or class from the project explorer and select **Add to simpleUML Diagram** | **New Diagram...** from the menu. The plugin will add a tab to the left gutter, which will open the tool area and display the following screenshot:



```
+ MainActivity extends AppCompatActi...
    impleme...   NavigationView.OnNavigationItemSelectedLis...
— fields ———————————————————————————————
— constructors ——————————————————————————
⊟ methods ································
#  onCreate ( savedInstanceSta...  Bun...  ) : void
+  onBackPressed ( ) : void
+  onCreateOptionsM...   ( me...  Menu ) : boole...
+  onOptionsItemSele...   ( ite...  MenuIt...  ) : boole...
+  onNavigationItemSele...   ( ite...  MenuIt...  ) : boole...
```

The **simpleUML Diagram** tool

The activity class used in the navigation drawer example that we used in this chapter is a little too simple to really show off the advantages of taking a diagrammatic view of our code, but if the reader applies this tool to a more complex class, with several fields and dependencies, its value will soon become apparent.

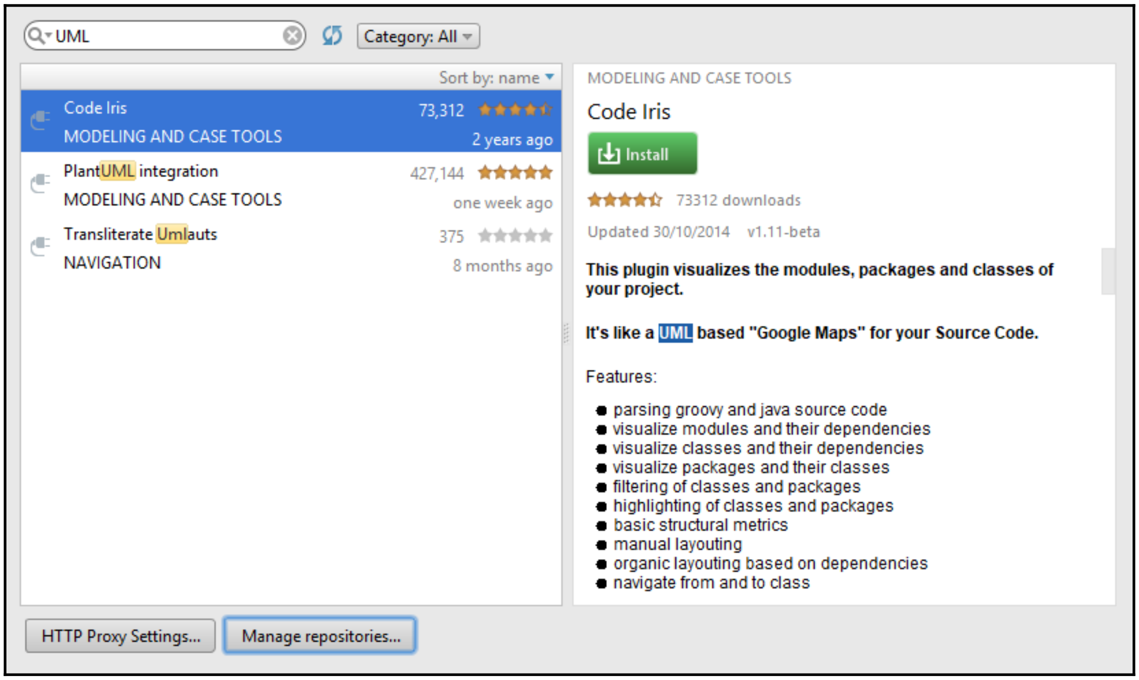
Diagrams can be saved as images and a variety of perspectives are available from the window's own toolbar.

TIP

Most Studio plugins will add a tab to the guttering. This is normally placed at the top of the left-hand gutter. This can often interfere with our own workspace preferences. Fortunately, these tabs can be rearranged by simply dragging and dropping them into a preferred position.

For most developers, a simple UML tool like this is enough, but if you prefer a more sophisticated tool, then **Code Iris** may be the plugin for you.

Code Iris does not need to be downloaded, as it can be found by browsing the plugins repository. Although, you will still need to restart the IDE. The repository can be accessed from the same **Plugins** window of the **Settings** dialog as the previous plugin but by clicking the **Browse repositories...** button.



This plugin visualizes the modules, packages and classes of your project.

It's like a UML based "Google Maps" for your Source Code.

Features:

- parsing groovy and java source code
- visualize modules and their dependencies
- visualize classes and their dependencies
- visualize packages and their classes
- filtering of classes and packages
- highlighting of classes and packages
- basic structural metrics
- manual layouting
- organic layouting based on dependencies
- navigate from and to class

The Browse Repositories dialog

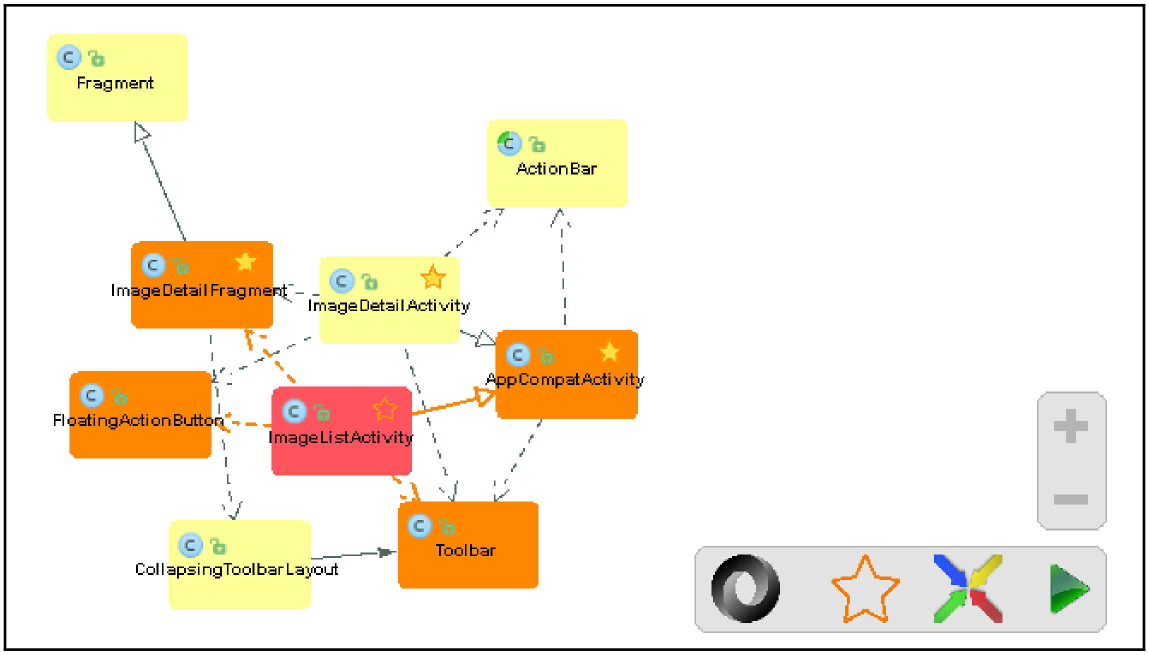
A quick look at the description of the project web page , which can be found at `plugins.jetbrains.com/plugin/7324-code-iris`, will show that Code Iris can do a great deal more than create class diagrams and should be thought of more as a more general visualization tool. The toolis described as a UML-based Google Maps for your source code, making it not only a useful development tool for individuals but also a great communication tool between teams, and it is better suited to graphing whole projects than equivalent tools.

TIP

Any third-party plugin repository can be made available in the repository browser, by clicking on the **Manage repositories...** button and pasting the relevant URL into the resultant dialog.

Code visualizations can be generated by either by opening the tool window and selecting the **Create / Update Diagram** button or from the project explorer from an individual module, package, or class's entry.

Code Iris' strength lies in its ability to visualize a project on any scale and quickly switch between these using the view slider. Used in conjunction with filters and an automatic arrangement algorithm called **organic layouting**, we can quickly generate appropriate and easily understood visualizations.



Code Iris visualization

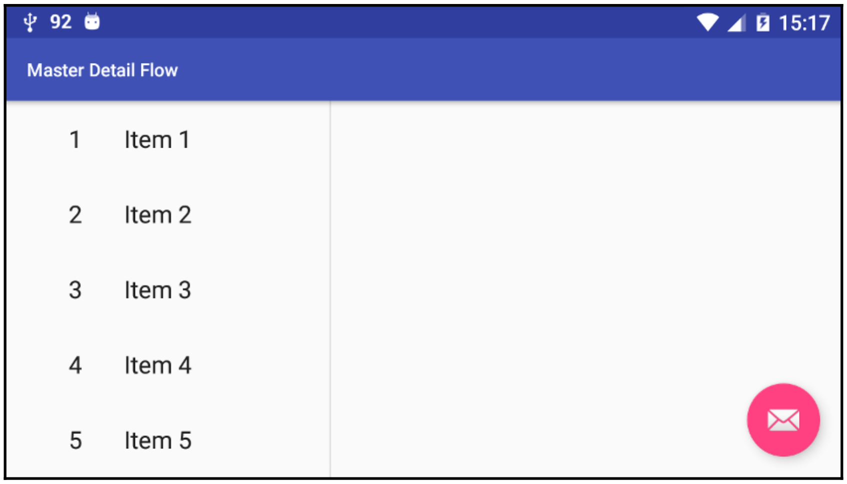
Despite its pretensions toward biology and the poor use of English, the organic layouting tool is actually remarkably clever, useful, and good-looking. When switched on (using the **play** button), diagrams will arrange themselves dynamically, according to our focus, which can be directed simply by clicking on the class of interest. If you have ever found yourself in the unenviable position of having to work with poorly documented code, this tool can save you many hours of head scratching.

The two plugins covered here are by no means the only inspection and visualization tools available to us, and a quick search on the internet will uncover many more. The two we selected here were chosen as they are indicative of the types of tool around.

The navigation drawer template we have explored here is remarkably useful as it contains several, almost ubiquitous UI components. It is also quite straightforward to understand. Another very handy project template is the **Master/Detail Flow** template.

The Master/Detail Flow template

Master/Detail UIs are found frequently on mobile devices, as they maximize the use of space very nicely by displaying a list and each of its items separately or side by side depending on the current width of the screen. This results in phone in a portrait orientation displaying two single panes; however, but in landscape mode or on a larger device, such as a tablet, two panes will be displayed side by side:



A two-pane view

The preceding screenshot is taken from the unmodified **Master/Detail Flow** project template viewed on a device in landscape mode so that both panes are displayed. We said previously that the two-pane view was visible on phones in landscape mode. If you have tried this on many phones, you will have found that this is not necessarily true. By default the template only displays two panes on screens whose longest side is 900dp or more. This can be deduced from the presence of the `res/layout-w900dp` directory.

To enable a two-pane view on smaller devices, we need only change the name of this folder. This, of course, can be done directly from our file explorers, but Android Studio has a powerful refactoring system with a sophisticated preview window. Although it is not needed in this case, it is perhaps most useful as it searches for references to it and renames them too.

The **Rename** dialog can be opened directly by pressing *Shift* + *F6*.

Of course, if you try to access the `layout-w900dp` folder from the navigation bar or project explorer, you will not be able to do so. To do this, switch from the **Android** tab to the **Project** tab in the explorer, as this presents the project exactly as it is on disk.

A quick examination of the code will reveal a Java class called DummyContent. As you will see, there is a TODO notice, pointing out that this class needs to be removed before publication, although, of course, it is quite possible simply to refactor it. The purpose of this file is to demonstrate how content can be defined. All that is required from us is to replace the placeholder array applied in the template with our own. This can, of course, take any form we choose, such as video or a web view.

Being able to start a project with ready-made code is very useful and can save us a great deal of time. However, there are many times when we might want to begin a project with a structure of our own that suits our purpose, and this may well not be one of those available via the IDE. Such a situation does not, however, prevent us from doing this, as code templates are available to us at any point and, as we shall see, it is even possible to create our own.

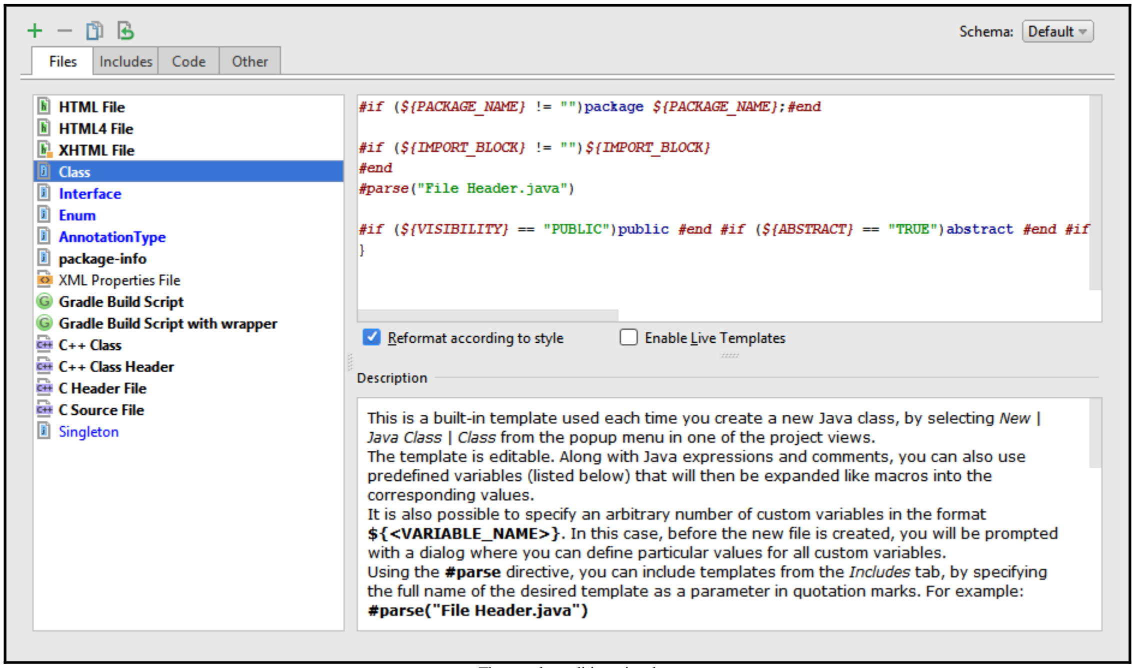
Custom templates

Imagine that you are developing a project that uses one of the templates we have already examined, but you would also like a login activity. Fortunately, this is easily managed using an already started project from within the IDE.

It is not obviously apparent that the project templates screen that we are presented with when starting a new project is available to us at any point. Simply select **New | Activity | Gallery...** from the project explorer's context-sensitive menu, and then choose the activity of your choice. You will then be presented with a customization screen, similar to those you have seen before, but with options to declare the parent and package, enabling us to use as many templates as we wish.

If you have visited the activity gallery, you will also have noted that these activities can also be selected directly and without having to open the gallery.

This is not the whole story, as the same menu allows us to create and save our own templates. Opening the context-sensitive menu of a source code folder and selecting **New** | **Edit File Templates...** will open the following dialog:



```
#if (${PACKAGE_NAME} != "")package ${PACKAGE_NAME};#end

#if (${IMPORT_BLOCK} != "")${IMPORT_BLOCK}
#end
#parse("File Header.java")

#if (${VISIBILITY} == "PUBLIC")public #end #if (${ABSTRACT} == "TRUE")abstract #end #if
}
```

☑ Reformat according to style ☐ Enable Live Templates

This is a built-in template used each time you create a new Java class, by selecting *New* | *Java Class* | *Class* from the popup menu in one of the project views.
The template is editable. Along with Java expressions and comments, you can also use predefined variables (listed below) that will then be expanded like macros into the corresponding values.
It is also possible to specify an arbitrary number of custom variables in the format **${<VARIABLE_NAME>}**. In this case, before the new file is created, you will be prompted with a dialog where you can define particular values for all custom variables.
Using the **#parse** directive, you can include templates from the *Includes* tab, by specifying the full name of the desired template as a parameter in quotation marks. For example:
#parse("File Header.java")

The template editing wizard

As you can see from the preceding screenshot, there are a good number of file templates available, as well as three other tabs. The **Includes** tab provides file headers and the **Code** tab contains smaller code units, many of which are useful during testing. The **Other** tab is of particular use and provides templates for larger application components, such as activities, fragments, manifests, layouts, and resources.

The **+** icon in the top left-hand corner allows us to create our own templates (but only from the first two tabs). At its simplest, one can simply paste code directly into the window provided. Once named and saved, this will then appear in the **File** | **New** menu or from project explorer directories directly.

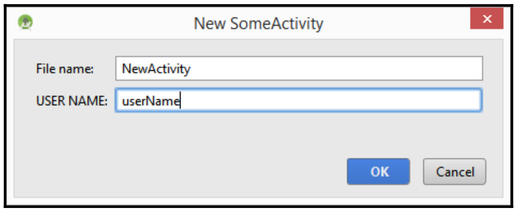
Taking just a quick look at some of the built-in templates will immediately reveal the use of a placeholder variable in the form of `${VARIABLE}`. It is these placeholders that make customized template such a useful and flexible tool.

The easiest way to see how these variables work is to use one of the existing templates to take a look at how these placeholders are implemented; this is outlined in the following exercise:

1. Open the **Edit File Templates** wizard, as described previously.
2. From the **Other** tab, copy the code from the `Activity.java` entry.
3. Create a new template with the **+** button, name it, and paste the code into the space provided.
4. Edit the code according to what you want, ensuring to include a custom variable, such as in the following code:

```
package ${PACKAGE_NAME};

import android.app.Activity;
import android.os.Bundle;

#parse("File Header.java")
public class ${NAME} extends Activity {

public String ${USER_NAME}

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
    }
}
```

5. Click on **OK** to save the template, which will now appear in **New** menus.
6. Now, whenever you create an instance of this template, Android Studio will prompt you to enter any variables defined in the template placeholders:


New SomeActivity

File name: NewActivity

USER NAME: userName

Generating a class from a template

All developers rely on code that they can use again and again, and although other IDEs provide code templates, the number of built-in structures and the ease of creation makes this Android Studio's biggest time-saving feature.

Third-party templates

Before moving on, we need to take a quick look at another way to access ready-made templates: third-party templates. There are many of these across the web, as a quick search will reveal. Many of the best of these are unfortunately not free, although many do offer free trials.

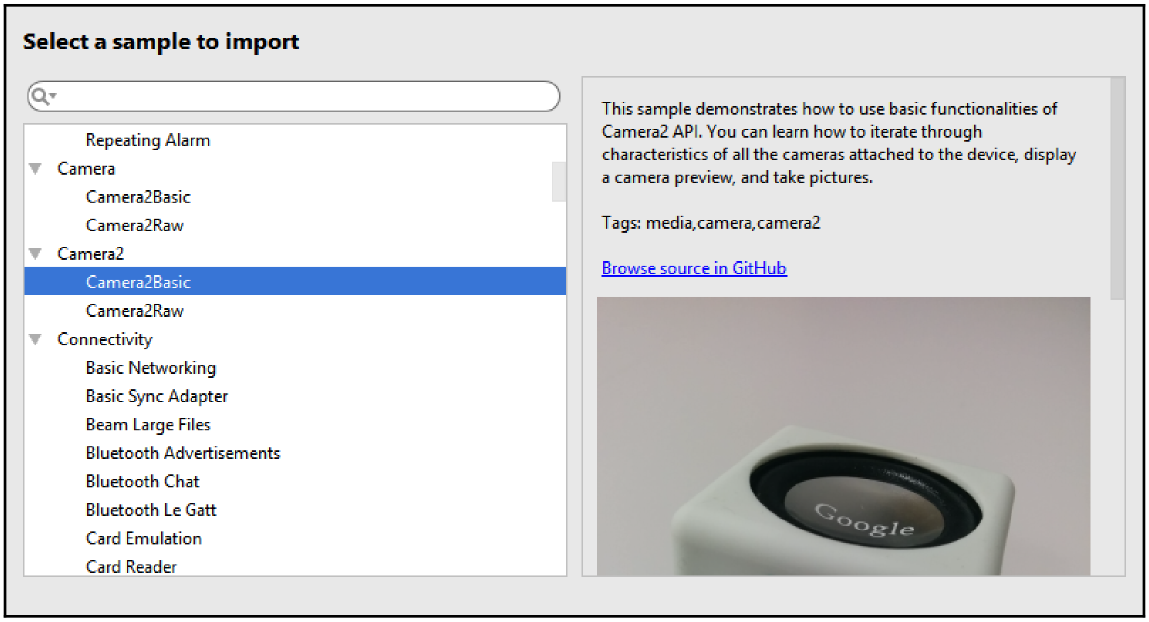
Sites such as Softstribe and Envato offer highly developed templates for a large number of app types, such as radio streaming, restaurant booking, and city guides. These templates are mostly fully developed apps and require little more than configuring and customizing. Such an approach may not suit the seasoned developer, but if speed is your top priority and you have a budget for such things, these services offer powerful shortcuts to project completion.

Templates are not the only time-saving feature of Android Studio when it comes to providing ready-made code, and you will no doubt have noticed the many sample projects available through the IDE.

Project samples

Although samples can be accessed from within the IDE from the sample browser, it is more usual to open one of these samples at the beginning of a project. This can be done from the welcome screen under **Import an Android code sample**. There are hundreds of these samples (and many more can be found online), and these are nicely grouped into categories in the sample browser.

Like templates, samples can be used as a starting point for a bigger project, but they are also an education in, and of, themselves, as they are written by extremely knowledgeable developers and are wonderful examples of best practices:



This sample demonstrates how to use basic functionalities of Camera2 API. You can learn how to iterate through characteristics of all the cameras attached to the device, display a camera preview, and take pictures.

Tags: media,camera,camera2

The sample browser

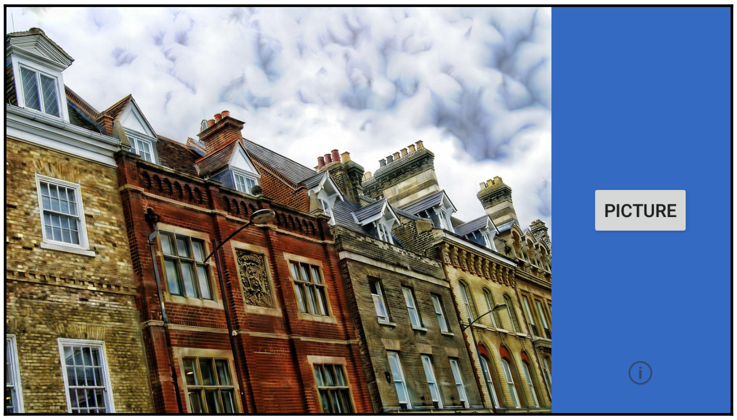
As can be seen from the preceding sample browser, samples can not only be downloaded from the browser but can also be viewed in GitHub. As the reader will know, GitHub is a fantastic code repository of immense use to developers of all kinds, and it houses all the samples found in the samples browser; there are thousands of other Android projects, code libraries, and plugins. From a samples point of view, this resource is a great time saver, as it is much quicker to take a look at the code before deciding whether to download and build.

There are so many samples, and they are all equally useful and instructive, making it difficult to select any one to examine here, although the **Camera2** samples can be helpful to explore as this is one API that many developers may not have examined before. This is down to the following two factors:

- Very often, a camera functionality can be accessed simply by calling the native app (or one the user has installed) from within our own app.
- The Camera2 API is incompatible with devices running on API level 20 and lower. Unlike many APIs, there is no handy support library to make Camera2 backward-compatible beyond this.

Despite these drawbacks, if you are planning an app that focuses on capturing images and video, then you will need to build in all the functionalities yourself. This is where these samples can come in remarkably handy. The **Camera2Basic** sample is probably the best one to look at first.

The sample contains just three classes and a simple layout structure catering to both landscape and portrait orientations. The classes include a basic startup activity, an extended TextureView that resizes the area to be captured depending on the device it is running on, and a Fragment that does most of the work. All these classes are nicely commented and are pretty much self-explanatory, and a quick examination of the code is all that is required to understand its workings. Like other samples, **Camera2Basic** can be built and run without any further modifications:



The **Camera2Basic** sample

All of the samples in the repository are equally useful, depending on your choice of project, and all of them are equally well written and instructive.

One of GitHub's most useful properties is the large number of third-party libraries available there, and when we take a closer look at the Gradle build process, we will see how these libraries can be included as dependencies from within Android Studio.

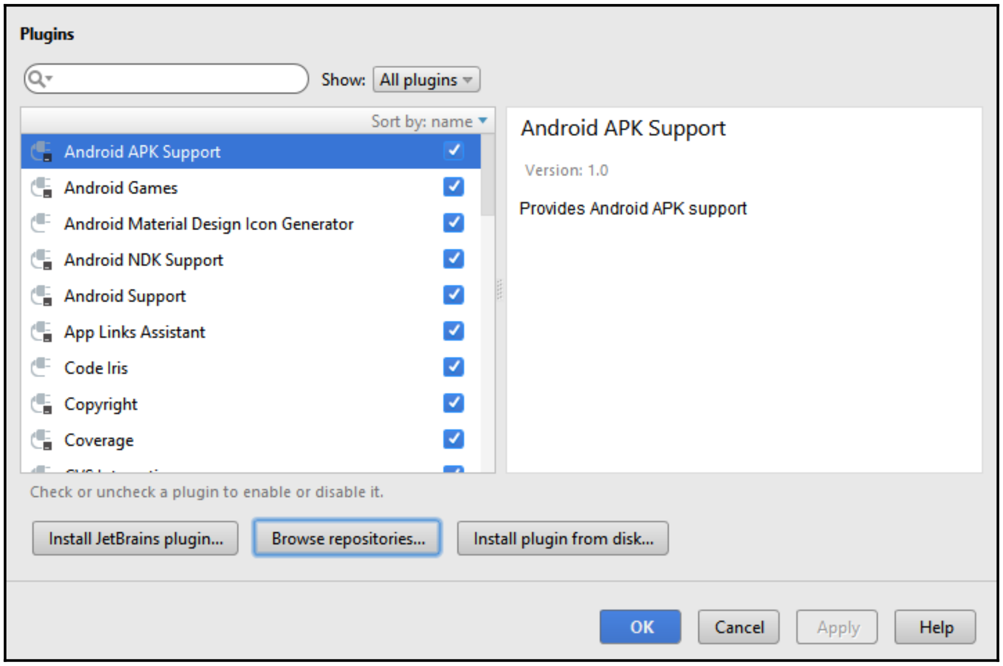
These are not the only useful tools available on GitHub, and GitHub is by no means the only source of such tools. As we have moved from UI development to coding, now is as good a time as any to take another look at some of the third-party plugins available for Android Studio.

Third-party plugins

There is a large and growing number of third-party plugins available, making it difficult to select a fair sample. The following sections covers a small collection of plugins selected primarily for their general-purpose usefulness.

ADB Wi-Fi

This is a JetBrains plugin and so can be found using the **Browse Repositories...** button of the **Plugins** screen of the **Settings** dialog:


Plugins

Show: All plugins

Sort by: name

- Android APK Support ✓
- Android Games ✓
- Android Material Design Icon Generator ✓
- Android NDK Support ✓
- Android Support ✓
- App Links Assistant ✓
- Code Iris ✓
- Copyright ✓
- Coverage ✓

Android APK Support

Version: 1.0

Provides Android APK support

Check or uncheck a plugin to enable or disable it.

The plugins dialog

This plugin simply allows us to debug our apps using a shared Wi-Fi connection. Despite its simplicity, this app is more than a cable-saving convenience, as it allows live debugging of many of the device's sensors that would be greatly restricted by being tethered to the host machine. Fitness and augmented reality apps can be tested with far more ease this way.

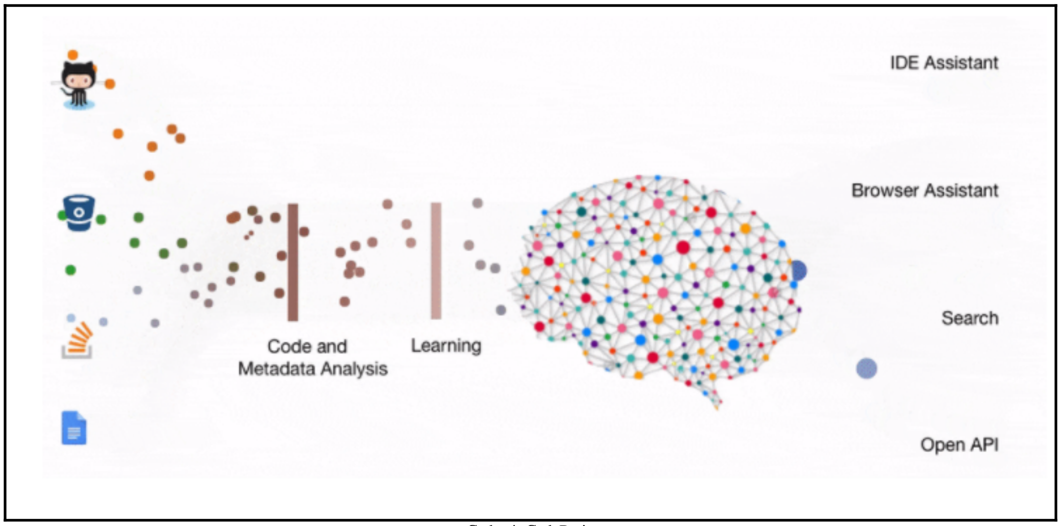
ADB Wi-Fi is very simple to set up and use, especially if your host and device already share a Wi-Fi connection and the device has been used for debugging before.

Most plugins are far more complex than this one, and many employ some very sophisticated technology, such as AI.

Codota

A great deal is being made recently of advances in Artificial Intelligence, and while many such claims are overblown and pretentious, nevertheless there are some striking examples of a truly helpful AI available, and Codota is one such project.

Many developers will already be familiar with Codota as an online search tool or browser extension. As smart and useful that these tools are, Codota really comes into its own as an IDE plugin. It uses a smart and evolving AI system named **CodeBrain**. This code is based on a form of emergent programming known as **Example-Centric Programming**.



Codota's CodeBrain

Technically, the Codota plugin is not really a plugin, as it has to be installed separately and runs in a window not connected from the IDE. This is because it works with the Java IDE, not individual environments. This has several advantages, and one of them is the fact that the plugin is a Java plugin and not Android-specific, meaning that it will run on any version of Studio and (unlike some plugins) doesn't require waiting for updates.

The Android Studio Codota plugin runs as a separate application, which is something of an advantage. When turned on, it is seriously clever (and according to its algorithmic philosophy, getting cleverer by the second). The software claims to provide an intelligent coding assistant, and very often it does just that, and I would challenge anyone to not be impressed by its acumen. It has some flaws but more often than not, finds examples from a wide variety of online sources, including stackoverflow and GitHub, that are almost always useful.

Once downloaded and opened, simply open the IDE click on some code of interest, and Codota will, most likely, provide some quite lovely answers to any questions you might have. Imagine having an assistant that, although not very bright, is very knowledgeable and can scan all pertinent online code within seconds. This is Codota, and whether you use it as a browser extension, a search tool, or an IDE plugin, it is one of the better coding assistants available.

Summary

In this chapter, we have looked at two approaches to assisting the process of code development: ready-made code, and assistant plugins and add-ons. Templates are often a great way to get a project off the ground quickly, and class inspection plugins allow us to easily understand larger templates, without having to pore over reams of code.

The other plugins we looked at in this chapter offered some different ways to make the task of app development easier and more fun. There are, of course, very many fantastic tools out there, and coding is continually becoming less monotonous and more creative.

This chapter has focused on programming in Java, but as any developer knows this is by no means the only language available. Android Studio provides support for both C++ and Kotlin (the latter can even be included alongside Java code).

In the next chapter we will explore how to support other languages as well as taking a look at Android Things, which, although not another language, does require skill sets that many traditional developer may be unfamiliar with. Fortunately Android Studio provides tools that make developing for single-board computers very similar to developing other Android applications.

7
Language Support

For an IDE to be considered truly essential, it has to do more than just offer the basics. In particular, it has to be accessible to developers from all kinds of backgrounds, using all kinds of languages and philosophies. For example, many developers prefer to take an object-oriented approach, whereas others prefer a more function-based philosophy, and many potential projects will lend themselves more easily to one or the other of these paradigms.

Android Studio 3 provides complete language support for both C++ and Kotlin, giving the developer the chance to focus on speed or programmability depending on the needs of the project in hand.

In addition to providing this language support, Android Studio also facilitates the development of apps for a wide variety of form factors. The reader will already be familiar with Android Wear and Android Auto, and recently the IDE has included support for Android Things.

In this chapter, we will take a look at these language support systems and the exciting new form factors that comprise the **Internet of Things** (**IoT**).

In this chapter, you will learn the following:

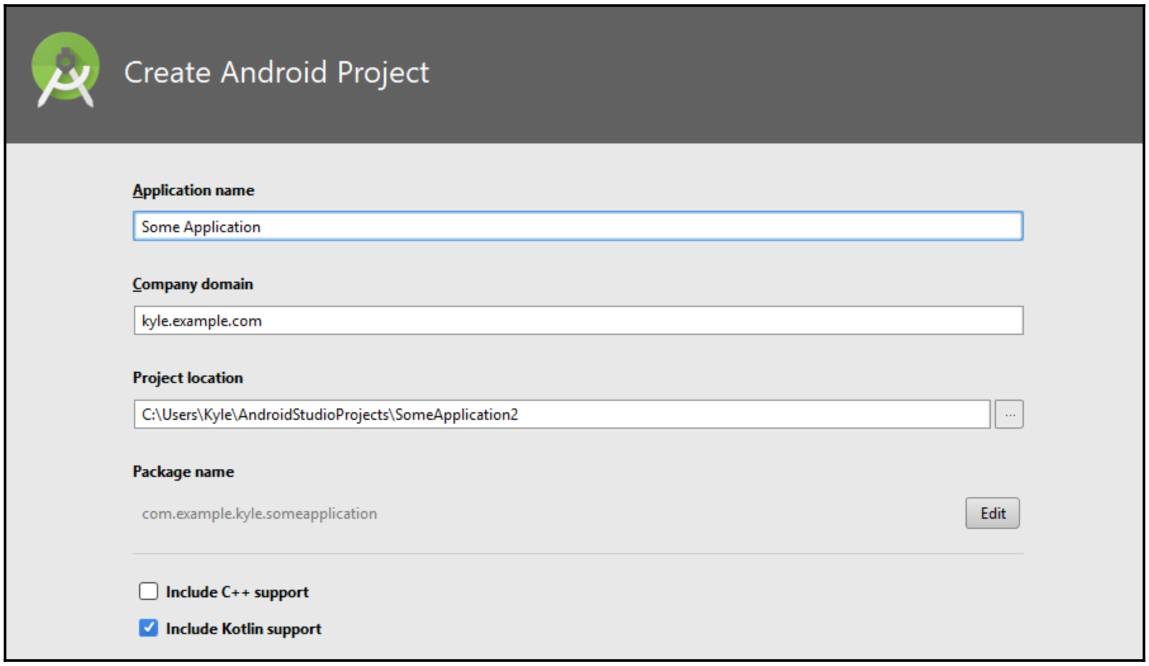
- Including Kotlin language support
- Integrating Kotlin with Java
- Applying Kotlin extensions
- Setting up native components
- Including C/C++ code within a project
- Creating an Android Things project

Kotlin support

Since the dawn of the mobile app, software development has undergone more than one revolution, and the Android framework has been no stranger to these changes. Many developers prefer Java as it is relatively easy to work with, but there will always be times when we want the raw speed of C++, and Java predates mobile devices by decades. Wouldn't it be nice if there ware a high-level language, such as Java, that had been designed with mobile development largely in mind.

Fortunately, a JetBrains team in Russia created Kotlin, which works alongside Java and even runs on the Java Virtual Machine, to create a language that better suits the needs of Android developers. It is also 100 percent interoperable with Java, so you can use Java and Kotlin files in the same project, and everything will still compile. You can also continue to use all existing Java frameworks and libraries with Kotlin.

Kotlin has been available for some time to developers as a plugin, but since the inception of Android Studio 3.0, Kotlin is now fully integrated into the IDE and is officially supported as a development language by Google. Both working samples written in Kotlin and wizard templates are included in the IDE.



Application name: Some Application

Company domain: kyle.example.com

Project location: C:\Users\Kyle\AndroidStudioProjects\SomeApplication2

Package name: com.example.kyle.someapplication

☐ Include C++ support

☑ Include Kotlin support

Including Kotlin support in the project setup wizard

Learning a new programming language is rarely a great deal of fun, and Kotlin is no exception. What takes the hardwork out of using it is that we do not have to make the leap from one language to another in a single jump, we can gradually introduce Kotlin as and when we choose.

Many of us will have worked with Java for a very long time and will see no real reason for changing. After all, Java works perfectly well, and years of experience can lead to some very speedy work practices. On top of this, the internet is awash with high-quality, open source, code repositories, making research and learning new skills very appealing to the Java coder.

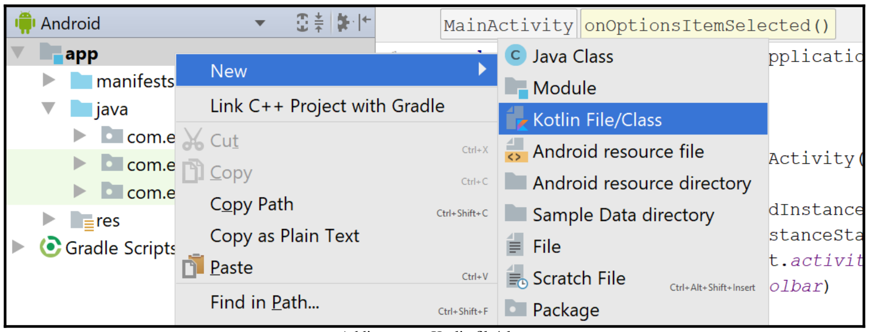
It will never be strictly necessary to learn and use Kotlin in Android development, but it is certainly worth taking a look at why so many developers think it represents the future of Android app development.

The advantages of Kotlin

Along with Google's own endorsements, there are a number of good reasons for developers to consider Kotlin. One of the reasons might be the end to null pointer exceptions, which the compiler acheives by not allowing a null value to be assigned to any object reference. Other exciting features of the language include favoring composition over inheritance, smart casting, and the ability to create data classes.

The best way to see how much of an advantage such innovations provide is to take a look at these.

As the screenshot in the preceding section demonstrates, Kotlin can be included into a project directly from the template wizard, but we can also include Kotlin classes from an already open project in precisely the same way that we add any other class or file, from the module's **New** menu.



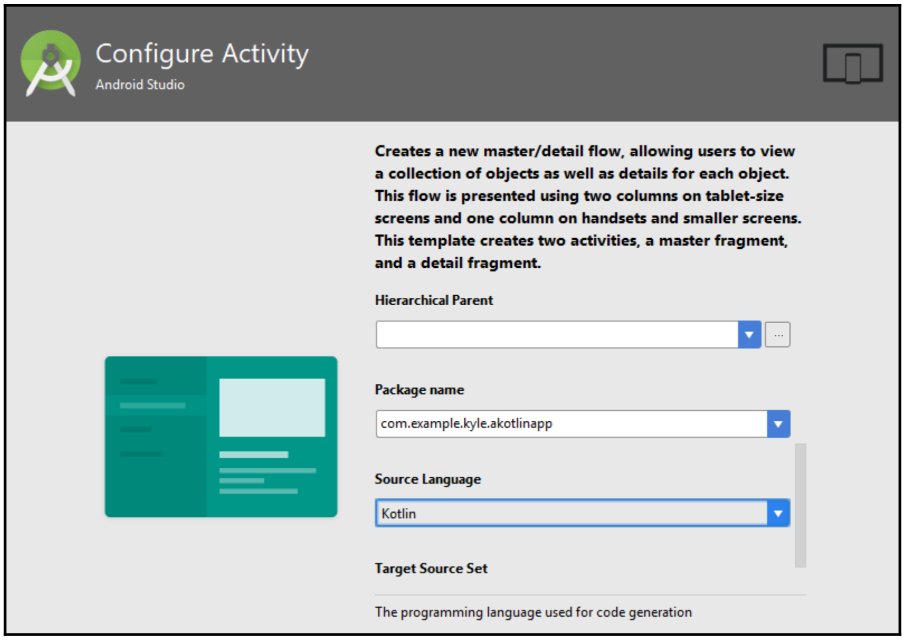
Adding a new Kotlin file/class

Including a Kotlin class in this manner will prompt the IDE to automatically configure Kotlin with Gradle. It does this by modifying the top-level `build.gradle` file as follows:

```
buildscript {
    ext.kotlin_version = '1.1.3-2'

    repositories {
        google()
        jcenter()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:3.0.0-alpha9'
        classpath "org.jetbrains.kotlin:kotlin-gradle-
                      plugin:$kotlin_version"
    }
}

allprojects {
    repositories {
        google()
        jcenter()
    }
}

task clean(type: Delete) {
    delete rootProject.buildDir
}
```

Using Kotlin in an Android application will incur almost no extra overheads; moreover, once it is compiled, Kotlin code will run no more slowly than its Java equivalent, nor will it occupy any more memory.

Including a new Kotlin class or file like this is very useful, but what about creating a Kotlin activity or fragment from a template? As Java developers, we are used to simple configuration dialogs to set these up. Fortunately, Kotlin activities are no different, and the **Configure Activity** dialog allows us to select our source language appropriately.


Configure Activity
Android Studio

Creates a new master/detail flow, allowing users to view a collection of objects as well as details for each object. This flow is presented using two columns on tablet-size screens and one column on handsets and smaller screens. This template creates two activities, a master fragment, and a detail fragment.

Hierarchical Parent

Package name: com.example.kyle.akotlinapp

Source Language: Kotlin

Target Source Set

The programming language used for code generation

Selecting the source language

As before, it is well worth taking a look at the resultant code to see just how much more concise and readable it is compared to the tradition Java activity/fragment templates:

```
class ItemDetailFragment : Fragment() {

    private var mItem: DummyContent.DummyItem? = null

    public override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        if (getArguments().containsKey(ARG_ITEM_ID)) {
            mItem = DummyContent.ITEM_MAP.
                     get(getArguments().getString(ARG_ITEM_ID))

            val activity = this.getActivity()
            val appBarLayout = activity.findViewById<View>
                     (R.id.toolbar_layout) as CollapsingToolbarLayout
            if (appBarLayout != null) {
                appBarLayout!!.setTitle(mItem!!.content)
            }
        }
    }
```

```
        public override fun onCreateView(inflater: LayoutInflater?,
             container: ViewGroup?, savedInstanceState: Bundle?): View? {
            val rootView = inflater!!.inflate(R.layout.item_detail,
                                   container, false)

            if (mItem != null) {
                (rootView.findViewById(R.id.item_detail) as
                             TextView).setText(mItem!!.details)
            }

            return rootView
        }

        companion object {
            val ARG_ITEM_ID = "item_id"
        }
    }
```

TIP This code can be made even more concise by removing calls to `findViewById()` using the Kotlin extension, as explained in the following section.

Although mixing-and-matching languages this way can be very useful to adapt and update existing apps, Kotlin really comes into its own when applied across an entire project. Perhaps its most appealing feature is its conciseness, and this can be easily seen by starting two projects from scratch and comparing their code. The following is the `onCreate()` listing from the Navigation Drawer template code:

```
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
        setSupportActionBar(toolbar);

        FloatingActionButton fab = (FloatingActionButton)
                     findViewById(R.id.fab);
        fab.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                Snackbar.make(view, "Replace with your own action",
                       Snackbar.LENGTH_LONG)
                        .setAction("Action", null).show();
            }
        });
        DrawerLayout drawer = (DrawerLayout)
                  findViewById(R.id.drawer_layout);
        ActionBarDrawerToggle toggle = new ActionBarDrawerToggle(
                this, drawer, toolbar, R.string.navigation_drawer_open,
                      R.string.navigation_drawer_close);
                      drawer.addDrawerListener(toggle);
                      toggle.syncState();

        NavigationView navigationView = (NavigationView)
                  findViewById(R.id.nav_view);
        navigationView.setNavigationItemSelectedListener(this);
    }
```

Here is its Kotlin equivalent:

```
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        setSupportActionBar(toolbar)

        fab.setOnClickListener { view ->
            Snackbar.make(view, "Replace with your own action",
                      Snackbar.LENGTH_LONG)
                    .setAction("Action", null).show()
        }

        val toggle = ActionBarDrawerToggle(
                this, drawer_layout, toolbar,
                R.string.navigation_drawer_open,
                R.string.navigation_drawer_close)
        drawer_layout.addDrawerListener(toggle)
        toggle.syncState()

        nav_view.setNavigationItemSelectedListener(this)
    }
```

The increased simplicity of this syntax is something all developers will welcome, and this is by no means the only advantage to using Kotlin.

Extending Kotlin

As one would expect with any powerful programming paradigm, it can be extended with a plugin to increase its usefulness even further.

Every Android developer will have lost count of the number of times they have typed `findViewById()`. They will also be aware of how error-prone such static typing can be.

The Kotlin extension is included, by default, when Kotlin support is enabled during project setup, as can be seen in the module-level `build.gradle` file:

```
apply plugin: 'com.android.application'

apply plugin: 'kotlin-android'

apply plugin: 'kotlin-android-extensions'
```

Using an extension also requires that it be imported into the appropriate class, usually an activity or fragment. It is more than likely that the reader will have set up automatic imports through system settings. All that is needed then is to create a view using XML in the usual fashion, like so:

```
<TextView
       android:id="@+id/text_view"
      . . . />
```

Now, all that is needed to set values on that widget is something along the lines of the following:

```
text_view.setText("Some text")
```

Regardless of whether you have imports set to be included automatically, it is useful to know the format these imports take. Consider the following example:

```
import kotlinx.android.synthetic.main.some_layout.*
```

It is also possible to import a reference to a specific view rather than the entire layout file with the following:
`import kotlinx.android.synthetic.main.some_layout.text_view`

1. In this case, `some_layout.xml` would be the file containing our `text_view`.
2. If you are in the healthy habit of using `<include>` to reference content XML from your activity XML, then you will need two imports, along these lines:

```
import kotlinx.android.synthetic.main.some_activity.*
import kotlinx.android.synthetic.main.some_content.*
```

It is not only the text we can set here; any function we like can be called in the same way without ever having to reference a view by searching for its ID.

Note that the use of a semicolon as a statement delineation is entirely optional in Kotlin.

Hopefully, by now, the reader will be convinced of the advantages of coding in Kotlin, but before we move on, one Kotlin feature is hidden away at the bottom of the main `Code` menu. This is **Convert Java File to Kotlin File**. This does exactly what it says and can even find and solve most conversion issues, making it a great time-saver and a fun way to learn the differences between the two languages.

Automatically convert Java into Kotlin with *Ctrl* + *Alt* + *Shift* + *K*.

Although Kotlin may be one of the newest additions to Android Studio, it is not our only choice for an alternative language, and the speed and low-level memory access that C++ provides have made it many developers' first choice. In the following section we will see how this powerful language can be easily supported by the IDE.

C/C++ support

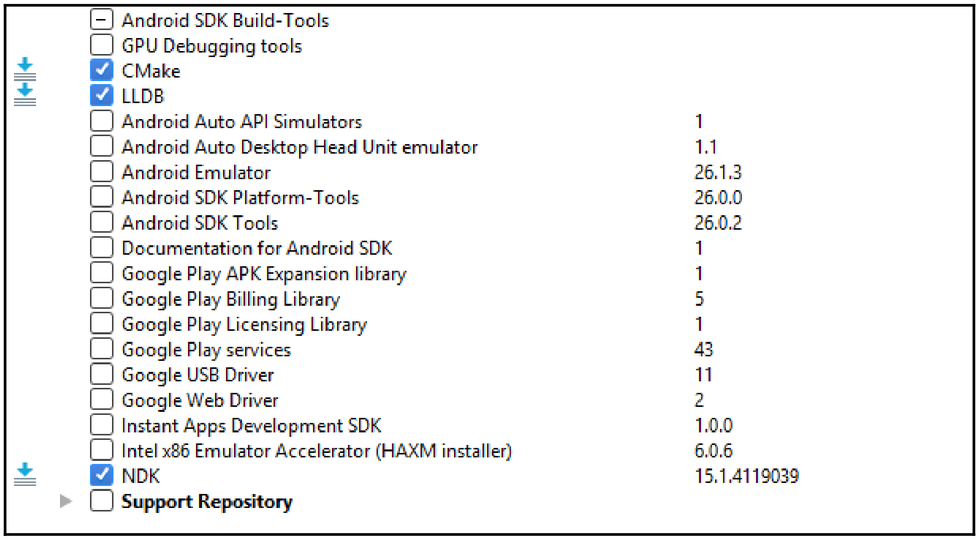
As we have seen so far, there are pros and cons to all programming languages. C and C++ probably take a little more discipline to master, but this is often more than made up for by the low-level control the language provides us with.

When it comes to Android Studio, a slightly different subset of development tools is required. This includes the **Native Development Kit** (**NDK**) and the **Java Native Interface** (**JNI**), along with other ways of debugging and building. As with most processes in Android Studio, setting up these tools is quite straightforward.

The NDK

As mentioned in the preceding section, native programming requires a slightly different set of tools from those we have been using up until now. As one might expect, everything we need can be found in the SDK Manager.

It is most likely that you will need to install the components highlighted in the following screenshot; you will need at least **NDK**, **CMake**, and **LLDB**:



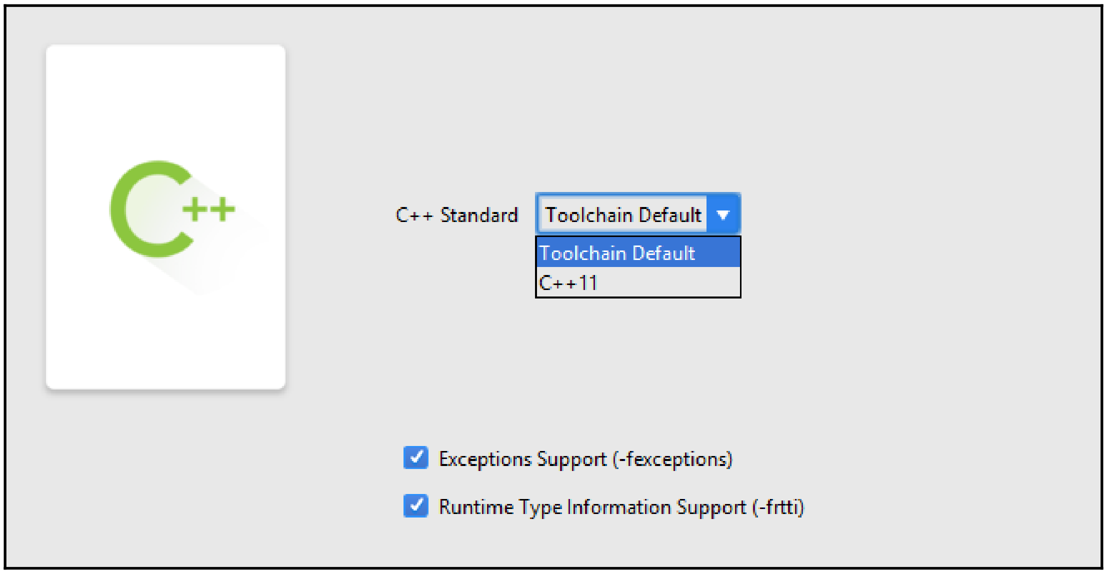
☐ Android SDK Build-Tools	
☐ GPU Debugging tools	
✓ CMake	
✓ LLDB	
☐ Android Auto API Simulators	1
☐ Android Auto Desktop Head Unit emulator	1.1
☐ Android Emulator	26.1.3
☐ Android SDK Platform-Tools	26.0.0
☐ Android SDK Tools	26.0.2
☐ Documentation for Android SDK	1
☐ Google Play APK Expansion library	1
☐ Google Play Billing Library	5
☐ Google Play Licensing Library	1
☐ Google Play services	43
☐ Google USB Driver	11
☐ Google Web Driver	2
☐ Instant Apps Development SDK	1.0.0
☐ Intel x86 Emulator Accelerator (HAXM installer)	6.0.6
✓ NDK	15.1.4119039
☐ **Support Repository**	

Native development components

- **CMake**: This a multiplatform test and build tool works alongside Gradle. For comprehensive documentation, visit `cmake.org`.
- **LLDB**: This is a powerful, open source debugging tool designed specifically to work with multithreaded applications. Its detailed usage is beyond the scope of this book, but interested users can visit `lldb.llvm.org`.

With the correct software installed, native coding is incorporated into Android Studio projects very smoothly, alongside our Java/Kotlin classes and files. As with Kotlin support, all that is needed is to check the appropriate box during setup.

Once this option is selected, you will be given the opportunity to configure C++ support, as follows:


C++
C++ Standard
Toolchain Default
Toolchain Default
C++11
✓ Exceptions Support (-fexceptions)
✓ Runtime Type Information Support (-frtti)

The C++ support customization dialog

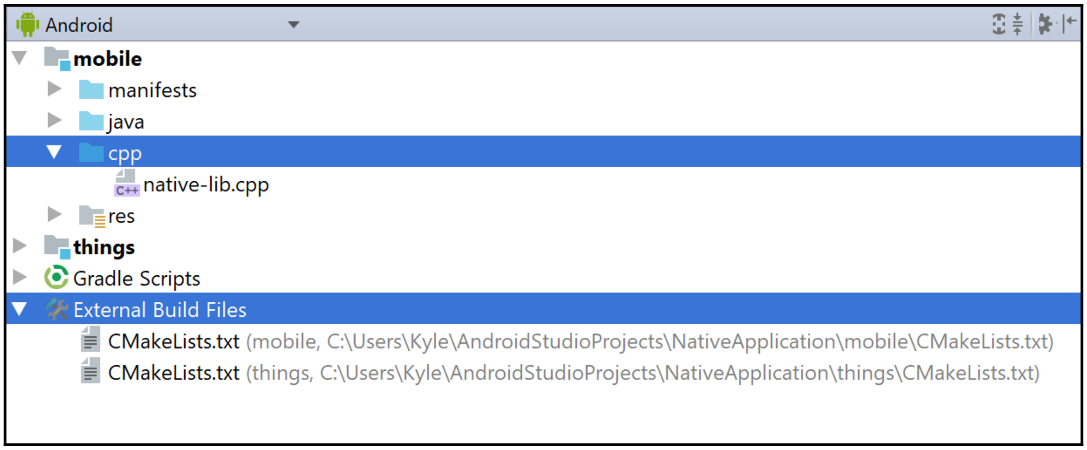
Selecting **Toolchain Default** as **Standard** is your best option if you are working with **CMake**, and both **Exceptions Support** and **Runtime Type Information Support** are probably worth checking on most occasions. Their inclusion can be most clearly seen by examining the module-level `build.gradle` file:

```
DefaultConfig { . . . externalNativeBuild { cmake { cppFlags "-frtti -
fexceptions" } } }
```

As is often the case, one of the best ways to get a good look under the hood without having to get our hands too dirty is with the ready-made Android samples. There are not a huge number of these, but they are all good, and there is a growing community project on GitHub at `github.com/googlesamples/android-ndk`.

What all these samples show is which code structures are added and where; like the project structures we've encountered before, the actual file structure is not reflected by the project file explorer, which organizes files according to their type, not location.

The obvious additions are the `main/cpp` directory, containing source code, and the external build files used by CMake.



CMakeLists.txt (mobile, C:\Users\Kyle\AndroidStudioProjects\NativeApplication\mobile\CMakeLists.txt)
CMakeLists.txt (things, C:\Users\Kyle\AndroidStudioProjects\NativeApplication\things\CMakeLists.txt)

Native code structures

C++ is not everyone's cup of tea, and going into greater detail is beyond the scope of this book. From an Android Studio point of view, those who want to take further advantage of the NDK will find that the way that CMake seamlessly integrates with Gradle makes testing and building apps calling on native libraries a fantastic time-saver.

One of the beauties of the Android OS, for users, manufacturers, and developers alike, is the enormous variety of devices it can, and does, run on. At first, this phenomenon appeared on our wrist watches, television sets, and in our cars. More recently, the development of the IoT has led to the need for sophisticated operating systems in any number of electronic devices. This has led Google to develop Android Things.

Android Things

The IoT has already made an impact on the consumer with the introduction of smart household appliances, such as kettles and washing machines. In addition to this, many municipal authorities uses the technology to manage things such as traffic and utility usage.

In reality, absolutely any device could constitute a Thing. A device does not even need to have a screen or any buttons, provided it has an IP address and can communicate with other devices. One can even get a toothbrush with an IP address, although the advantages of that sadly escape me.

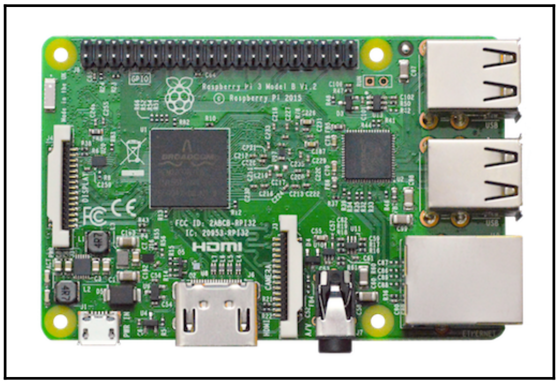
From a developer's perspective, the IoT is incredibly exciting, and the inclusion of APIs in the SDK opens up almost unlimited new worlds. Naturally, these APIs have been sewn neatly into Android Studio, making Android Thing development as simple and as much fun as any other branch of Android app creation.

Development kits

Perhaps the most significant differences between Android Things and other forms of Android development is the hardware. It is tempting to think that an expertise in embedded circuitry is needed, and although a little knowledge in this area is useful, it is by no means necessary, as Google works with **System on a Chip** (**SoC**) manufacturers, such as Intel, NXP, and Raspberry Pi, to produce developer kits that allow us to quickly produce and test prototypes.

As always, Android Studio has been designed to assist us in as many ways as possible, and of course, there are a support library, system images, and a growing collection of working samples to help us on our way. Unfortunately, there is no way to simply emulate Android Things, and although some functions can be emulated on some mobile AVDs, some form of physical development kit is required.

It is perfectly possible to create one's own development board with a little expertise and a soldering iron, but the low prices of boards, such as the Intel Edison and the Raspberry Pi, along with free system images from Android make this a time-consuming process. If what you have is an idea and you want to quickly test and develop it into a finished project, the way to go is with an approved development kit, such as the Raspberry Pi 3, as shown in the following image:



Raspberry Pi 3

Information about which single-board computers are available for Things can be found at `developer.android.com/things/hardware/developer-kits.html`. There are also peripheral kits available for each of the boards, and these can be found on the same page.

Once you have chosen a kit, system images can be found at `developer.android.com/things/preview/download.html` and also on your Things developer console at `partner.android.com/things/console/`.

Once you have your kit and peripherals, you are ready to develop your first Things application, the basics of which are outlined in the next section.

Creating a Things project

The APIs used by Android Things are not included in the standard SDK, and so a support library is required. At the very least, you will need the following dependency:

```
dependencies {
 ...
 provided 'com.google.android.things:androidthings:0.5-devpreview'
}
```

Along with this entry in your manifest:

```
<application ...>
 <uses-library android:name="com.google.android.things"/>
 ...
</application>
```

Most Things projects will need more than this, depending on which peripherals are used and whether the project will be tested using Firebase. Taking a look at the provided samples is a good way to see what dependencies are needed; the following snippet is taken from the Things Doorbell sample:

```
dependencies { provided 'com.google.android.things:androidthings:0.4-
devpreview'
  compile 'com.google.firebase:firebase-core:9.4.0'
  compile 'com.google.firebase:firebase-database:9.4.0'
  compile 'com.google.android.things.contrib:driver-button:0.3'
  compile 'com.google.apis: google-api-services-vision:v1-rev22-1.22.0'
  compile 'com.google.api-client: google-api-client-android:1.22.0' exclude
module: 'httpclient'
  compile 'com.google.http-client: google-http-client-gson:1.22.0' exclude
module: 'httpclient' }
```

The next major difference in setting up an Android Things project can be seen in the manifest file, and adding the highlighted `<intent-filter>` in the following code will enable the project to run successfully when testing and debugging:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
package="com.example.androidthings.doorbell">

  <uses-permission android:name="android.permission.CAMERA" />
  <uses-permission android:name="android.permission.INTERNET" />
  <uses-permission android:name="com.google.android.things.permission
.MANAGE_INPUT_DRIVERS" />

  <application android:allowBackup="true"
android:icon="@android:drawable/sym_def_app_icon"
android:label="@string/app_name">
    <uses-library android:name="com.google.android.things" />
    <activity android:name=".DoorbellActivity">

    <intent-filter>
      <action android: name="android.intent.action.MAIN" />
      <category android: name="android.intent.category.LAUNCHER" />
    </intent-filter>

    <intent-filter>
      <action android: name="android.intent.action.MAIN" />
      <category android: name="android.intent.category.IOT_LAUNCHER" />
      <category android: name="android.intent.category.DEFAULT" />
    </intent-filter>

    </activity>
  </application>
</manifest>
```

These are really the only differences when it comes to setting up an Android Things project. The other differences will be based more around which peripherals and sensors are being used. As is so often the case, one of the best ways to explore Things further is through the provided samples. Not many samples are available, but the number is growing, and they have all been written to assist our learning.

Developing for Android Things can appear daunting for many developers, but the way that Android Studio facilitates this through its system images, support libraries, and code samples means that any developer with a great idea can cheaply and quickly develop, test, and produce such products.

Summary

Throughout this chapter, we explored some of the more exotic ways in which Android Studio can assist the developer. Google's immense influence in the digital world has provided alternative technologies such as the Kotlin language and encouraged manufacturers to develop technologies that appeal to Android developers in a way that makes cutting-edge technology available to anyone with skills and ideas.

Android Studio is not alone in offering the opportunity to code in different languages or for different form factors, but Android Studio does make it simpler and easier for developers to learn new skills.

In the next chapter, we will take a look at one of the final development stages; testing. This will give us a great chance to explore one of Android Studio's most innovative and useful tools: the device monitor and profiler.

8 Testing and Profiling

If there was only one reason to choose Android Studio over other IDEs, it could easily be claimed that it was due to its powerful debugging and testing tools. These tools range from a simple Logcat reporting to complex testing mechanisms based on the JUnit framework. In addition to tools to help us identify bugs in our code, Android Studio also has a host of very smart, performance monitoring tools that allow developers to fine-tune projects and maximize their efficiency.

This chapter will explore each of these processes in turn, beginning with simple inline debug calls and then moving on to the different types of JUnit tests and concluding by taking a look at how to monitor the performance of our apps under a variety of conditions.

In this chapter, you will learn how to:

- Configure a Logcat debug filter
- Create local unit tests
- Build instrumented tests
- Record Espresso tests
- Test UIs
- Perform remote testing
- Stress testapps
- Enable advanced profiling
- Record method traces
- Record memory allocation
- Inspect Java heap dumps
- Inspect network traffic

Logcat filters

One of the simplest, but nevertheless the most useful debugging, techniques is the simple inclusion of a Logcat filter. This can be used to report variable values or simply keep track of which methods are being called. This is of particular use when keeping track of processes that are not visibly apparent, such as services, broadcast receivers and callbacks that have no obvious impact on the UI.

Perhaps the very simplest debug tool available, and useful when we are in a hurry and just want to check for a single value or event, is to include a line like:

```
System.out.println("Something happened here");
```

This is only an on the fly solution as the output will be buried in among the rest of the Logcat text. Far easier to manage is to configure a Logcat filter. The following short exercise demonstrates how this is done:

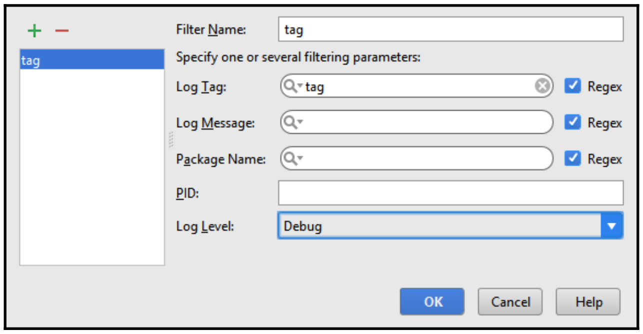
1. Start a new project, or open a new one.
2. Select an activity or fragment and include the following field:

    ```
    private static final String DEBUG_TAG = "tag";
    ```

3. Select a method that you wish to examine, and add a line resembling the one here:

    ```
    Log.d(DEBUG_TAG, "Some method called");
    ```

4. Open the Logcat tool using *Alt* + *6*.
5. Select **Edit Filter Configuration** from the dropdown in the top-right corner, and complete the resultant dialog, as follows:


tag

Filter Name: tag

Specify one or several filtering parameters:

Log Tag: tag ✓ Regex

Log Message: ✓ Regex

Package Name: ✓ Regex

PID:

Log Level: Debug

Filter configuration

6. Run the app.
7. The Logcat tool can now be used to track any value, activity, or event in the same fashion.

This is, by far, the least sophisticated manner to interrogate code as it is running, but it has its uses: it can be applied at any time and very quickly. This approach is fine for combating individual errors; once we have working code, we will need to test them under some well-defined conditions. This is where Android Studio's JUnit-based testing system comes into its own.

JUnit testing

No development project is complete until it has been thoroughly and rigorously tested, and Android Studio incorporates JUnit testing directly into the workspace. As the name suggests, the framework allows the testing of individual units of code. These are often individual modules but can just as likely be a single class or method.

The Android Studio JUnit test framework provides for two distinct types of test. They are as follows:

- Local unit tests are used to test business logic in an isolated environment that is not dependent on Android components or other code, although it is possible to mock some dependencies. These tests run on the local Java virtual machine and are consequently considerably faster than testing on a hardware device or emulator.
- Instrumented tests are used when we want to test elements of the Android framework itself, such as how our UIs behave. These tests generate an APK file and are, therefore, slower to build.

During the course of most development life cycles, we will need to employ both these techniques, and we will take a look at each in turn next.

For nearly all projects, we can expect to spend around twice the time testing code stability than we will testing a functionality, and we will take a look at both of these in the next section.

Local unit tests

If you have created an Android Studio project using the project wizard, then the basic test case for both test types will have been created automatically. The wizard will also include the necessary Gradle dependencies. If you are using a project created any other way, you will need to create a test directory structure and include Gradle dependencies by hand. These steps are outlined as follows:

1. Inside your `module/src` directory, create a new folder alongside `src/main` called `src/test`.
2. Inside this `test` directory, recreate the folder structure inside your `main` directory, for example:

   ```
   main/java/com/packt/chapterseven
   ```

3. This directory is where you will place your test classes, and it will now be accessible from the IDE's project explorer.
4. Finally, add the following dependency to your `build.gradle` file if it is not already included:

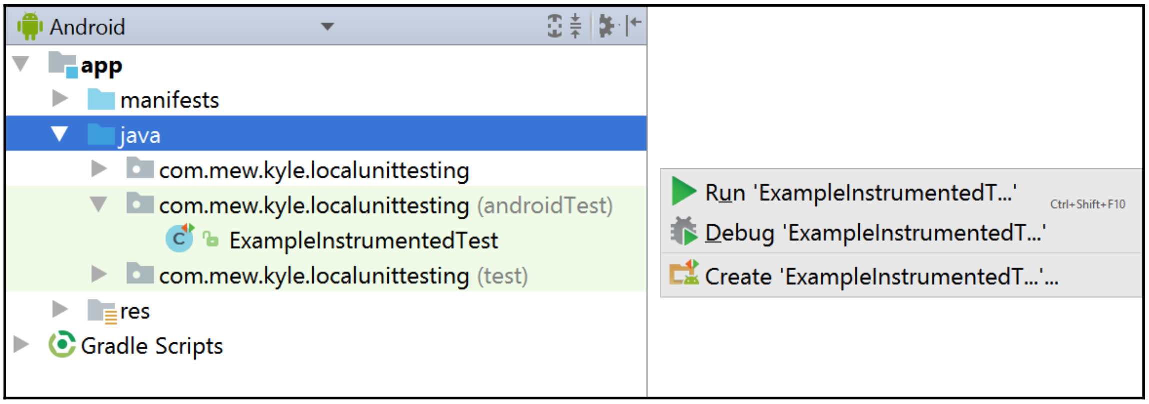
   ```
   testImplementation 'junit:junit:4.12'
   ```

If you have created your project using the wizard, then it will have included an example test class, `ExampleUnitTest.Java`. This class contains a single method for testing arithmetic:

```
public class ExampleUnitTest {

    @Test
    public void addition_isCorrect() throws Exception {
        assertEquals(4, 2 + 2);
    }
}
```

This is a very simple example, but it is nevertheless a good way to take a first look at how unit testing works in this setting. The best way to see this, is to create a project using the project setup wizard or open one that was created that way, so that it contains the test class.

Despite their actual location on disk, test modules can be found alongside your regular Java modules in the IDE's project explorer.

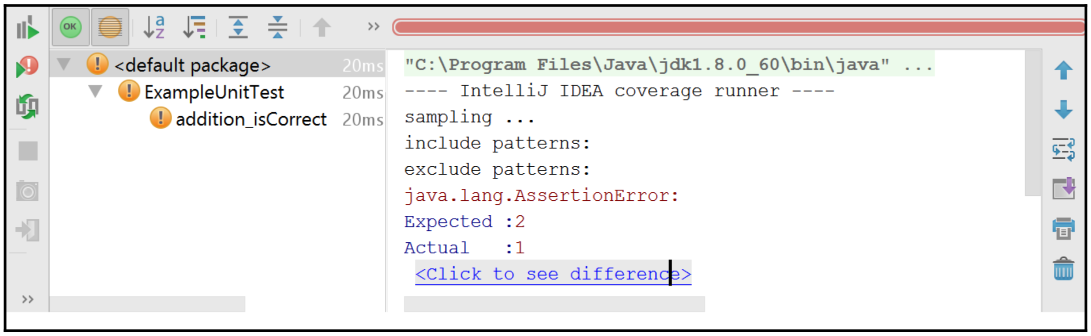

Android
app
manifests
java
com.mew.kyle.localunittesting
com.mew.kyle.localunittesting (androidTest)
ExampleInstrumentedTest
com.mew.kyle.localunittesting (test)
res
Gradle Scripts
Run 'ExampleInstrumentedT...' Ctrl+Shift+F10
Debug 'ExampleInstrumentedT...'
Create 'ExampleInstrumentedT...'...

Accessing tests from the IDE

The simplest way to see these tests in action and explore other test features is to doctor the `addition_isCorrect()` method so that it fails. The `assertEquals()` method simply compares two expressions and can be set up to fail, as follows:

```
public class ExampleUnitTest {
    int valueA;
    int valueB;
    int valueC;

    @Test
    public void addition_isCorrect() throws Exception {
        valueA = 2;
        valueB = 2;
        valueC = 5;

        assertEquals("failure - A <> B + C", valueA, valueB + ValueC);
    }
}
```

This produces the predictable output shown here:



```
"C:\Program Files\Java\jdk1.8.0_60\bin\java" ...
---- IntelliJ IDEA coverage runner ----
sampling ...
include patterns:
exclude patterns:
java.lang.AssertionError:
Expected :2
Actual   :1
 <Click to see difference>
```

Unit test output

The preceding run tool shown has a number of useful features that can be found in the toolbars. In particular, the third icon down on the left allows us to automatically rerun a test whenever we make any changes. The main toolbar allows us to filter and sort passed and ignored tests as well as import and export results, which can be saved as HTML, XML, or a custom format.

The **Click to see differences** link will open a failure comparison table that is very useful when multiple tests fail.

Tests can be run just as other code, mostly simply with the run icon in the main toolbar, but the **Run** menu and the **Run Test** icons in the left gutter of the code editor include the option to debug and also to display a class coverage window. These editor icons are particularly useful, as they can be used to run individual methods.

The example provided uses the JUnit `assertEquals()` assertion. There are many similar JUnit assertions and other structures available to us, and the complete documentation is available at `junit.org`.

The preceding example is self-contained and tells us nothing about how to use such classes to test our application code. The following example demonstrates how this can be done:

1. Create a Java class in the default package, with a single function, like the one here:

```
public class PriceList {

    public int CalculateTotal(int item1, int item2) {
```

```
                int total;
                total = (item1 + item2);

                return total;
            }
        }
```

2. Create a new class in the `test` package along these lines:

```
        public class PriceListTest {

            @Test
            public void testCalculateTotal(){
                PriceList priceList = new PriceList();
                int result = priceList.CalculateTotal(199, 250);

                assertEquals(449,result);
            }
        }
```

Unlike the first example, the preceding code demonstrates how we can incorporate our business logic within test code.

Once we find ourselves with several tests, it can become useful to have some control over the order in which these tests run, and particularly if we wish to run preparatory code at the beginning of every test run. This can be achieved with a series of JUnit annotations, as follows:

```
@BeforeClass
@Test(timeout=50)
public void testSomeMethod() {
...
```

The preceding configuration annotation will cause the method to run once only, before all the other methods in the class are called as well as failing after 50 ms. `@Before` can be used to cause a method to execute before every other test, and there are equivalent `@After` and `@AfterClass` annotations.

There are many other assertions and other classes available in the `org.junit` package and full documentation can be found at the following link:

`junit.sourceforge.net/javadoc/org/junit/package-summary.html#package_description`

Often, you will want to run the same group of test classes together. Rather than running each of these separately each time, a suite of tests can be recreated and run as one, with code similar to the following one:

```
@RunWith(Suite.class)
@SuiteClasses({
        someClassTest.class,
          someOtherClassTest.class })
```

It is not always possible, or desirable, to test every unit in complete isolation. Often, we will need to test a unit's interaction with Android and other Java interfaces and classes. This is generally achieved by creating mock dependencies.

As the reader will know, there are many ways to create mock objects and classes, from the painstaking task of building them from scratch to using ready-made third-party frameworks. In most cases, this second option is preferable, perhaps the only exception being some full-screen games that completely redefine the UI. Otherwise, the easiest and, probably, the best option for Android Studio users would be Mockito.

Mockito is a powerful Java framework, and although it is easily incorporated into Android Studio, it is by no means particular to it, and many readers will already be familiar with it from other IDEs. There is a great deal that could be covered on the subject, but this would be beyond the scope of this book. Naturally, Mockito needs to be declared as a dependency in our `build.gradle` files, and this is done as follows:

```
testImplementation 'org.mockito:mockito-core:2.8.9'
```

Fortunately, it is not necessary to create mock dependencies to be able to make calls to the Android API. If the default return values from any `android.jar` method are sufficient, then we can instruct Gradle to do this by adding the following snippet to theAndroid section of the `build.gradle` file:

```
testOptions {
  unitTests.returnDefaultValues = true
}
```

Mockito provides structures to mock up most of the Java classes we might need to test our business logic, but, at the end of the day, we are developing an Android application and will need to test it on real devices and emulators. Once we are satisfied that our model works well in isolation, we need to see how it performs in the real world.

Testing a UI

Although considered separately here, instrumented tests can also be unit tests. There are many non-UI Android classes that we need to test against, and although these can be mocked up, this can be a time-consuming process, particularly when we know these classes are sitting there, already fully implemented on our devices and emulators. If we are prepared to sacrifice the fast build times of mock testing, then we might as well plug in our devices and boot up our emulators.

One aspect of development that is difficult to mock is UI simulation and interaction, and generally speaking, when we want to test our layouts against physical gestures. Fortunately, there are some very handy tools and features at our disposal that help test and optimize our designs.

Testing views

At the heart of instrumented UI testing lies the Android Testing Support Library. This includes the JUnit APIs, a UI Automator and the Espresso testing framework. There is virtually nothing involved in setting Espresso up on Android Studio, as it is included as a dependency by default if you are working on a project generated by the project setup wizard. If not, you will need to add the following to your `build.gradle` file:

```
androidTestImplementation('com.android.support.test.espresso:espresso-
core:2.2.2', {

    exclude group: 'com.android.support',
          module: 'support-annotations'

})
```

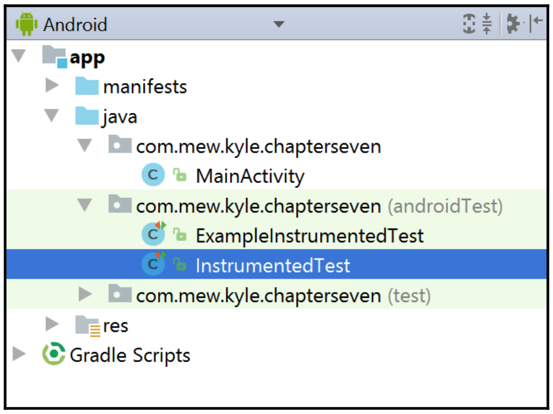
If you have developer animation options, such as **window** and **transition animation scales** set on your test device, you will need to disable them for the duration of your tests for Espresso to work smoothly.

Put simply, Espresso allows us to perform three essential tasks:

1. Identify and access views and other UI elements.
2. Perform an activity, such as clicks and swipes.
3. Validate assertions to test code.

The best way to see how this works is with a simple example. Similarly to unit tests, instrumented tests need to be placed in the correct disk location to be recognized by Android Studio, as follows:

```
\SomeApp\app\src\androidTest
```


Android
app
manifests
java
com.mew.kyle.chapterseven
MainActivity
com.mew.kyle.chapterseven (androidTest)
ExampleInstrumentedTest
InstrumentedTest
com.mew.kyle.chapterseven (test)
res
Gradle Scripts

Instrumented test location

The following steps demonstrate how we can carry out the three tasks referred to a moment ago:

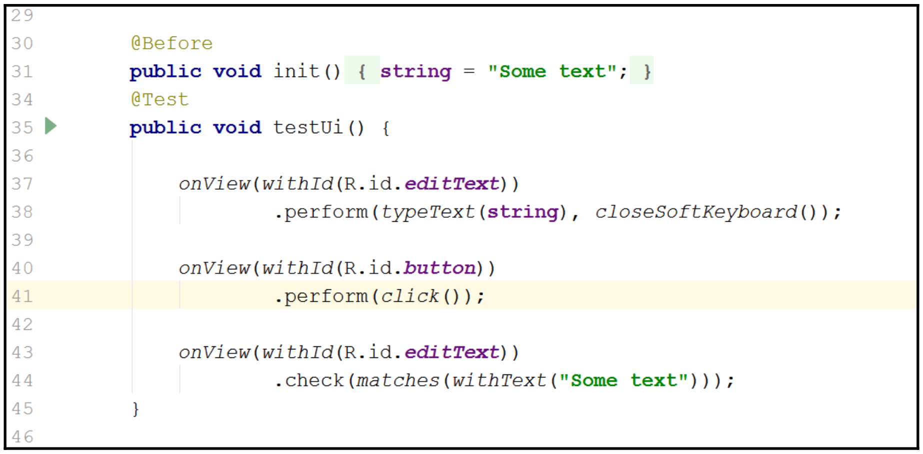
1. Create two views, as shown in the following code, in an activity; here, the main activity is used:

```
<EditText
    android:id="@+id/editText"
    . . .
    />

<Button
    android:id="@+id/button"
    . . .
    />
```

2. Create a test class in the `androidTest` directory along these lines:

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

    private String string;
    @Rule
    public ActivityTestRule<MainActivity> testRule = new
ActivityTestRule<>(
            MainActivity.class);

    @Before
    public void init() {
        string = "Some text";
    }

    @Test
    public void testUi() {

        onView(withId(R.id.editText))
                .perform(typeText(string),
                        closeSoftKeyboard());

        onView(withId(R.id.button))
                .perform(click());

        onView(withId(R.id.editText))
                .check(matches(withText("Some text")));
    }
}
```

3. Note that the IDE identifies Espresso term in italics:



```
    @Before
    public void init() { string = "Some text"; }
    @Test
    public void testUi() {

        onView(withId(R.id.editText))
                .perform(typeText(string), closeSoftKeyboard());

        onView(withId(R.id.button))
                .perform(click());

        onView(withId(R.id.editText))
                .check(matches(withText("Some text")));
    }

```

Italicized Espresso terms

4. Run the test, either from the editor's left gutter or the **Run** menu.
5. The application will open on the test device, `string` will be typed into the edit box, the button will be clicked, and the activity will be finished and closed.
6. The test results can then be viewed in the IDE.

There are one or two items that could do with pointing out in the preceding code, especially if one is new to Espresso. `ActivityTestRule` is used to access the widgets in our activity, and the call to `closeSoftKeyboard()`; the latter is not strictly necessary, but, as you will see if you run the test, it does precisely as one might imagine and closes the soft keyboard.

When running instrumented tests, the platform makes use of a test manifest, which, if you have created your project from a template or are working on a sample, will already be included. This will be located in the following directory on disk: `\SomeApplication\app\build\intermediates\manifest\androidTest\debug`

Nearly all of the libraries used in these tests will need to be imported, and although the code editor is good at picking up on missing imports, it is also good to know which libraries are needed. The following is a list of those required for the preceding test:

```
android.support.test.filters.LargeTest;
android.support.test.rule.ActivityTestRule;
android.support.test.runner.AndroidJUnit4;

org.junit.Before;
org.junit.Rule;
org.junit.Test;
org.junit.runner.RunWith;

android.support.test.espresso.Espresso.onView;
android.support.test.espresso.action.ViewActions.click;
android.support.test.espresso
        .action.ViewActions.closeSoftKeyboard;
android.support.test.espresso.action.ViewActions.typeText;
android.support.test.espresso.assertion.ViewAssertions.matches;
android.support.test.espresso.matcher.ViewMatchers.withId;
android.support.test.espresso.matcher.ViewMatchers.withText;
```

Hamcrest assertion matchers can be included in JUnit tests by including the following dependency in the `build.gradle` file:
`Implementation 'org.hamcrest:hamcrest-library:1.3'`

Espresso provides many other actions along with typing and clicking, such as scrolling and clearing text. Comprehensive documentation on Espresso can be found at the following link:

`google.github.io/android-testing-support-library/docs/`

Testing lists and data

The preceding example uses `onView()` to identify the views we want to test using their ID, and this is fine for components we have already named; however, items in lists cannot be identified so explicitly, and, for this, we will need another approach. When dealing with lists, such as recycler views and spinners, Espresso provides the `onData()` method to identify list items.

To see this in action, add a spinner as shown in the following one of your app activities:

```
public class SomeActivity extends AppCompatActivity {

    ArrayList<String> levelList = new ArrayList<String>();
    TextView textView;

    @Override
    protected void onCreate(Bundle savedInstanceState) {

        . . .

        Spinner spinner = (Spinner) findViewById(R.id.spinner);

        levelList.add("Easy");
        levelList.add("Medium");
        levelList.add("Hard");
        levelList.add("Impossible");

        ArrayAdapter<String> adapter = new ArrayAdapter
                <String>
                (MainActivity.this,
                        android.R.layout.simple_spinner_item,
                        levelList);
        spinner.setAdapter(adapter);
        spinner.setOnItemSelectedListener
                (new AdapterView.OnItemSelectedListener() {

            @Override
            public void onItemSelected(AdapterView<?>
                    parent, View view, int position, long id) {

                Snackbar.make(view, "You selected the"
                        + levelList.get(position)
                        + " level ", Snackbar.LENGTH_LONG)
                        .setAction("Action", null)
                        .show();
            }

            @Override
            public void onNothingSelected(AdapterView<?> parent) {

                Snackbar.make(view, "Nothing selected"
                        ,Snackbar.LENGTH_LONG)
                        .setAction("Action", null).show();

            }
        });
    }
```

We can now write a test using `onData()` to interrogate the widget:

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

    private String string;

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

    @Before
    public void init() {

        string = "Medium";
    }

    @Test
    public void testSpinner() {

        onView(withId(R.id.spinner))
                .perform(click());
        onData(allOf(is(instanceOf(String.class)), is(string)))
                .perform(click());

        onView(withId(R.id.spinner))
                .check(matches(withText
                (containsString("Medium"))));
    }
}
```

TIP

Even though you have included Hamcrest as a Gradle dependency, the studio's quick-fix feature will not kick in, and the following imports will need to be included in the test code:

```
import static org.hamcrest.Matchers.allOf;
import static org.hamcrest.Matchers.containsString;
import static org.hamcrest.Matchers.instanceOf;
import static org.hamcrest.Matchers.is;
```

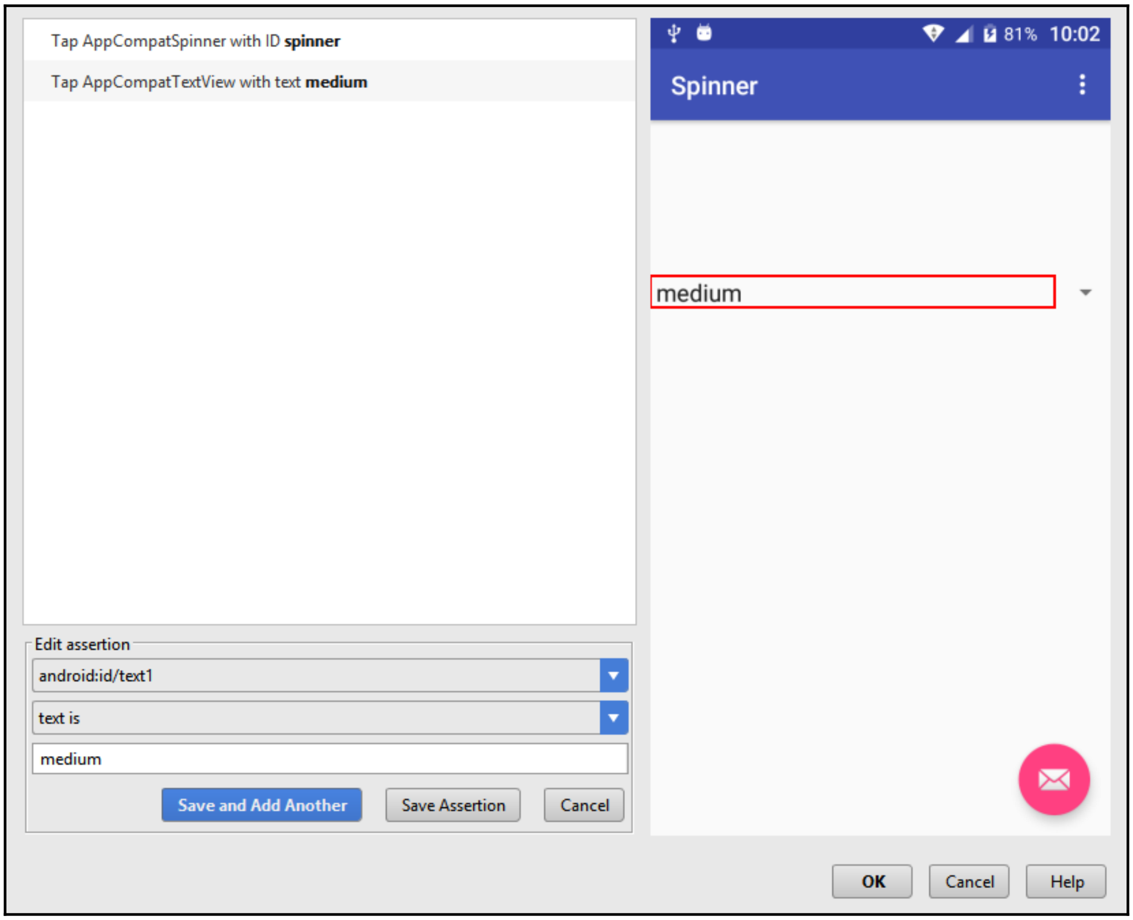
Recording tests

In the preceding section we saw how Android Studio provides a comprehensive set of tools for testing our code, but writing these tests is time consuming, and with anything other than the most trivial of projects will require many individual tests. Fortunately, Android Studio provides a semi-automated way to construct tests, using our own UI interactions to create, identify, and perform code elements of the test.

The following simple exercise shows how this is done to perform the preceding test we just wrote by hand:

1. Open the project with the spinner we created in the previous exercise, or create a new one.
2. Select **Record Espresso Test** from the **Run** menu.
3. Select an item from the spinner. This will be reflected in the **Record Your Test** dialog.
4. Click on the **Add Assertion** button.

5. Select the spinner by clicking on it, and complete the dialog, as shown here:



Tap AppCompatSpinner with ID **spinner**
Tap AppCompatTextView with text **medium**

Edit assertion
android:id/text1
text is
medium

The **Record Your Test** dialog

6. Save and run the test.

   As you can see, the IDE has taken our screen gestures and converted them into code:

```
@Test
public void someTest() {
    ViewInteraction appCompatSpinner = onView(
      allOf(withId(R.id.spinner),
        childAtPosition(
          childAtPosition(
withClassName(is("android.support.design.widget.CoordinatorLayo
ut")),
            1),
          0),
        isDisplayed()));
    appCompatSpinner.perform(click());

    DataInteraction appCompatTextView = onData(anything())
      .inAdapterView(childAtPosition(
withClassName(is("android.widget.PopupWindow$PopupBackgroundVie
w")),
           0))
      .atPosition(1);
    appCompatTextView.perform(click());

    ViewInteraction textView = onView(
      allOf(withId(android.R.id.text1), withText("medium"),
        childAtPosition(
          allOf(withId(R.id.spinner),
            childAtPosition(
IsInstanceOf.<View>instanceOf(android.view.ViewGroup.class),
            0)),
         0),
      isDisplayed()));

    textView.check(matches(withText("medium")));
}
```

This code is perhaps not as efficient or as user friendly as it might be, but the time saved is probably worth it, and, at the end of the day, all our tests are temporary and will be done away with once we are happy with our code.

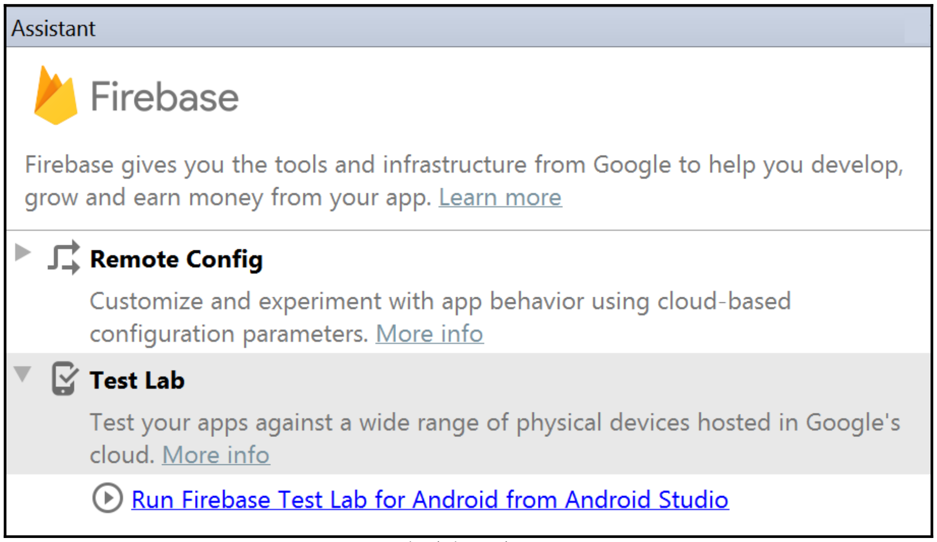
It will not have escaped the reader's notice that when running tests from the **Select Deployment Target** dialog, there is also a **Cloud Testing** tab. This feature allows us access to the Firebase Test Lab directly from the IDE.

Remote testing

When developing an Android application for a general release, it is desirable to test it on as many different device configurations and platform versions as possible. Testing on a large number of real devices is impractical, and it would seem that virtual devices offer the only other option. Fortunately, Firebase provides a cloud-based test lab that allows us to test our apps on a wide range of real devices and emulators across all platform versions.

Firebase is a powerful and fully formed, cloud-based application development suite with many useful features, such as file hosting and real-time crash reporting. For the purpose of this chapter, we will focus on just one Firebase product, the test lab.

Firebase is well catered for within the IDE, and the simplest way to get started is the Firebase **Assistant**, which can be found in the **Tools** menu:


Assistant

Firebase

Firebase gives you the tools and infrastructure from Google to help you develop, grow and earn money from your app. Learn more

Remote Config

Customize and experiment with app behavior using cloud-based configuration parameters. More info

Test Lab

Test your apps against a wide range of physical devices hosted in Google's cloud. More info

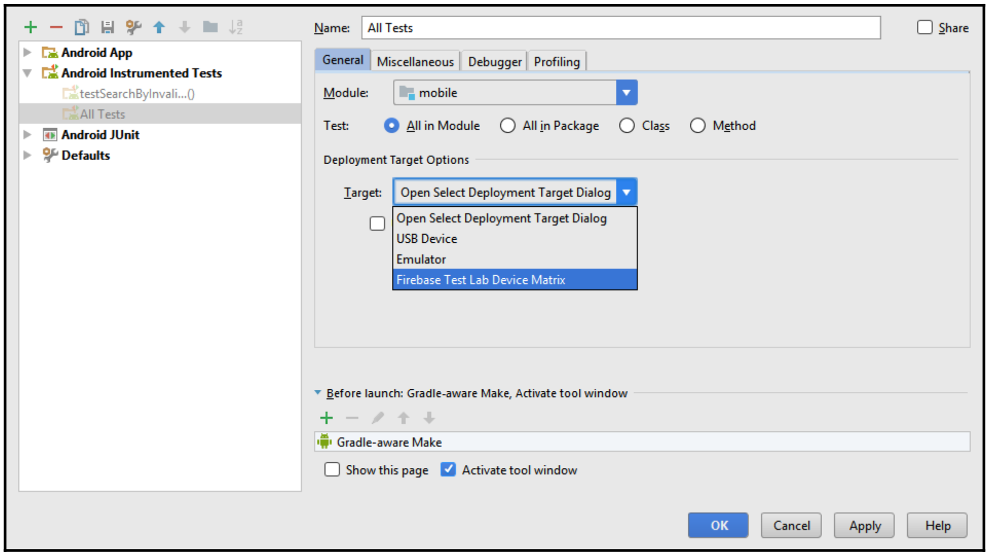
Run Firebase Test Lab for Android from Android Studio

The Firebase assistant

Before connecting Android Studio to Firebase, first log in using your Google account at `https://firebase.google.com/`.

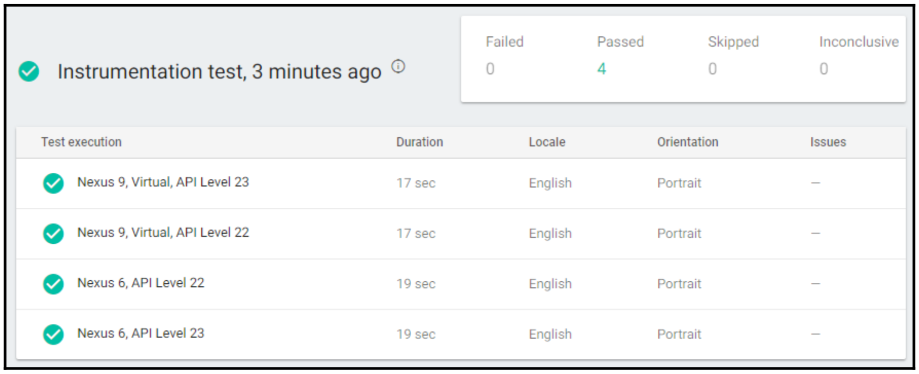
Clicking on the **Learn more** link will allow you to connect to Firebase directly from the IDE. This will take you through a quick wizard/tutorial, culminating in the clicking of a **Connect to Firebase** button.

We can now configure our cloud-based test by opening the **Run/Debug Configurations...** dialog from the **Run** | **Edit Configurations...** menu:


Name: All Tests
Share
Android App
Android Instrumented Tests
testSearchByInvali...()
All Tests
Android JUnit
Defaults
General
Miscellaneous
Debugger
Profiling
Module: mobile
Test: All in Module, All in Package, Class, Method
Deployment Target Options
Target: Open Select Deployment Target Dialog
Open Select Deployment Target Dialog
USB Device
Emulator
Firebase Test Lab Device Matrix
Before launch: Gradle-aware Make, Activate tool window
Gradle-aware Make
Show this page
Activate tool window
OK
Cancel
Apply
Help

Test configuration

These tests can now be started in the same way as any other project, with the Run icon or menu item, and you will see from the test output a link to view an HTML version of the results, as follows:


Instrumentation test, 3 minutes ago

Failed	Passed	Skipped	Inconclusive
0	4	0	0

Test execution	Duration	Locale	Orientation	Issues
Nexus 9, Virtual, API Level 23	17 sec	English	Portrait	–
Nexus 9, Virtual, API Level 22	17 sec	English	Portrait	–
Nexus 6, API Level 22	19 sec	English	Portrait	–
Nexus 6, API Level 23	19 sec	English	Portrait	–

The Firebase output

It is worth noting at this point that, although many prefer it, Firebase is not the only cloud available device to test Android apps, and interested readers should look up **Amazon Web Service** (**AWS**) Device Farm, Xamarin Test Cloud, Sauce Labs, Perfecto, and several others.

The preceding methods outlined demonstrate a variety of testing techniques that we can apply to our code and ways that Android Studio can speed up and automate much of this essential, but often unexciting, aspect of development. Before moving on to more interesting topics, there is one other form of testing that needs a little explanation, and although not strictly a part of the IDE, the Application Exerciser Monkey is nevertheless a very useful little tool.

Stress testing

The Android Application Exerciser Monkey is a handy command line, application stress tester. It works by performing (or injecting) a stream of random input actions, such as clicking, typing, and swiping. It is akin to handing your app to a toddler and seeing if they can break it. All developers understand that the user can, and will, attempt to do utterly ridiculous and unpredictable things with their app, and short of sitting there trying to replicate every possible combination of gestures, the exerciser Monkey is as close as we can get to predict the unpredictable.

The Monkey is very simple to run: simply open the command prompt in your `sdk/platform-tools` directory and enter the following command:

```
adb shell Monkey -p com.your.package -v 5000
```

Where 5000 is the number of random actions you want carried out, the output will resemble the following snippet:

```
. . .
:Sending Touch (ACTION_DOWN): 0:(72.0,1072.0)
:Sending Touch (ACTION_UP): 0:(70.79976,1060.0197)
:Sending Touch (ACTION_DOWN): 0:(270.0,1237.0)
:Sending Touch (ACTION_UP): 0:(284.45987,1237.01)
:Sending Touch (ACTION_DOWN): 0:(294.0,681.0)
:Sending Touch (ACTION_UP): 0:(301.62982,588.92365)
:Sending Trackball (ACTION_MOVE): 0:(-3.0,-1.0)
. . .
```

A table with all Monkey command-line options can be found at `https://developer.android.com/studio/test/monkey.html.`

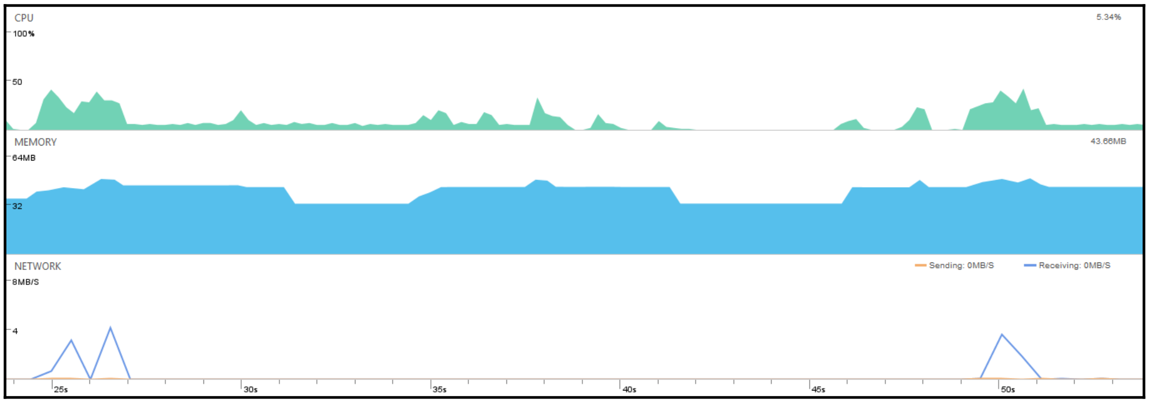
Testing our business logic, how it incorporates itself with the rest of the system, and how it behaves on a wide range of devices under numerous conditions, is a vital part of any development life cycle. Once we are certain, however, that our code behaves as we intend, we can move on and interrogate how well it performs these tasks. We need to ask how efficient our work is, whether it contains memory or resource bottlenecks, or drains the battery unnecessarily. To do this, we will need to turn to the Android Profiler.

Performance monitoring

We may have ironed out all the glitches in our code, but there is still plenty of fine-tuning to do, and one of Android Studio's most innovative features, the Android Profiler, allows us to do just that.

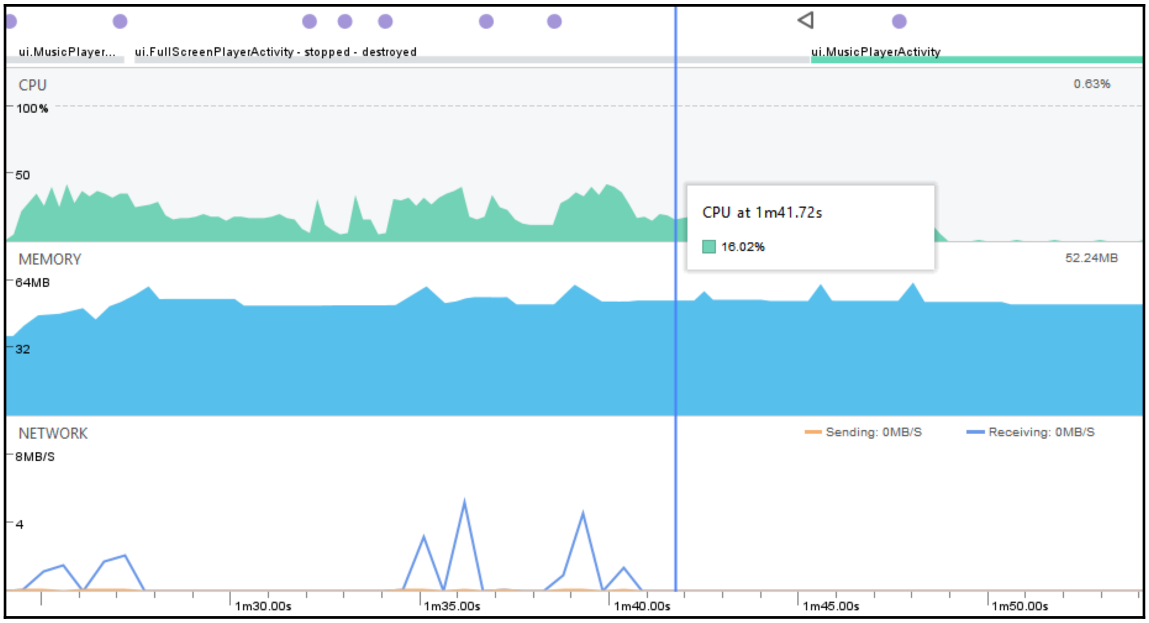
The Android Profiler is not available for modules developed using C++.

The Android Profiler was introduced in Android Studio 3.0 and replaced the previous Android Monitor. At the most basic level, it monitors live CPU, memory, and network usage. This allows us to test our app under different conditions and configurations and improve its performance. It can be accessed from the **View | Tool Windows** menu or the tool window bar.



Performance monitoring

This basic monitoring is no different from the Android Monitor of previous incarnations. This is because features such as method tracing and memory allocation inspection have a negative impact on build times. Advanced profiling can easily be enabled from the **Run/Debug Configurations** dialog, found via the **Run | Edit Configurations...** menu.



Advanced performance monitoring

The profiler now displays a specific event information, along with a host of other features that we will explore now.

CPU profiling

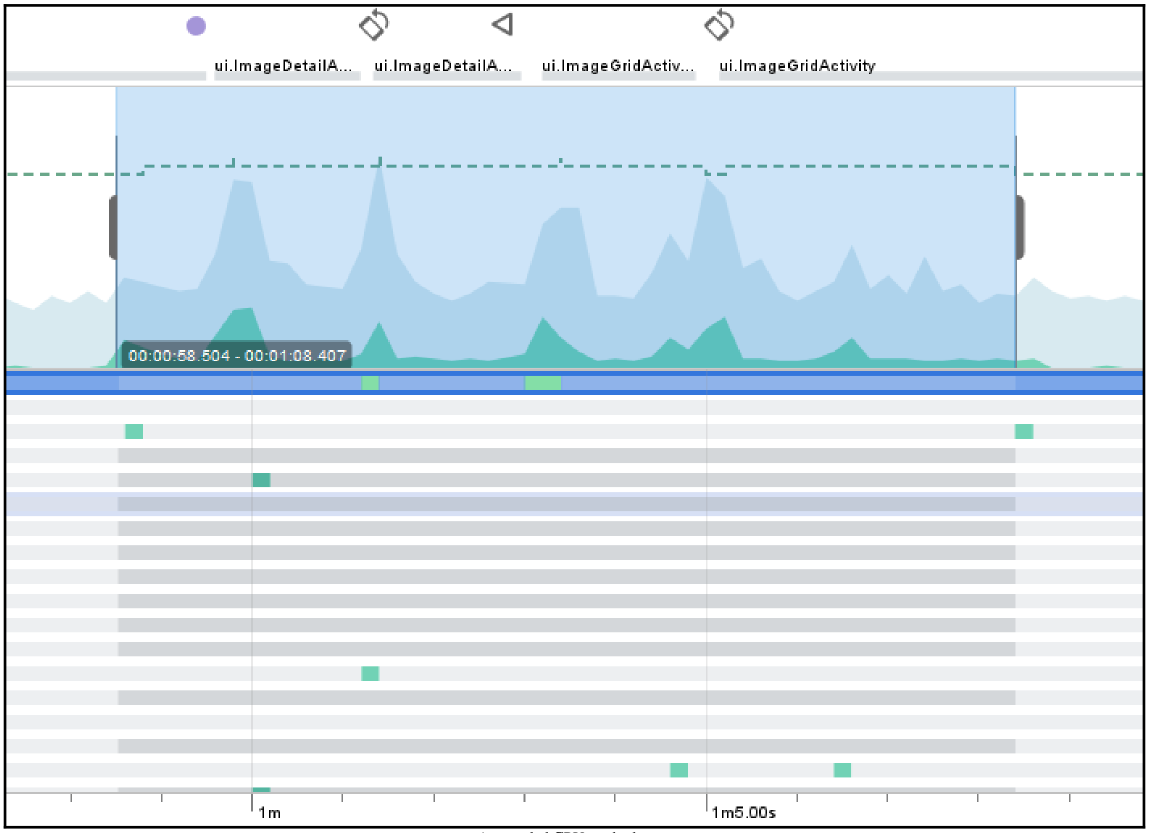
The Android Profiler provides far deeper inspection than its predecessor, the Android Monitor, and allows detailed inspection of thread activity, UI events, and individual method performance. The CPU profiler also allows us to record method traces along with some sophisticated inspection tools to help us make our programs more efficient.

The CPU advanced profiling features can be seen by clicking anywhere in the CPU timeline. This will then display the thread activity timeline in the lower portion of the display.

Observing our app's behavior live like this can be very revealing, but, to best see what is going on, we need to record a period of activity. This way, we can inspect individual threads.

The following short exercise demonstrates how to record such a method trace:

1. Click anywhere in the CPU timeline to open the advanced CPU profiler.
2. Decide which actions you want to record.
3. There are two new dropdowns at the top of this pane. Select **Instrumented** over **Sampled** and leave the other as is.
4. If you are planning a long recording, zoom out.
5. Click on the record icon and perform your planned actions.
6. Click on the same icon again to stop.



A recorded CPU method trace

The ears on either side of the recorded sample can be dragged to adjust the length of the recording.

As the exercise demonstrates, there are two kinds of recording, instrumented and sampled; their difference is as follows:

- Instrumented recordings take exact timings from when methods are called.
- Sampled recordings take samples of memory usage at regular intervals.

As you will see, there is a choice of how to represent these data, in the four tabs at the bottom of the tool window. **Call chart** and **Flame chart** present a graphical view of the method hierarchy, whereas **Top down** and **Bottom up** display this information as lists.

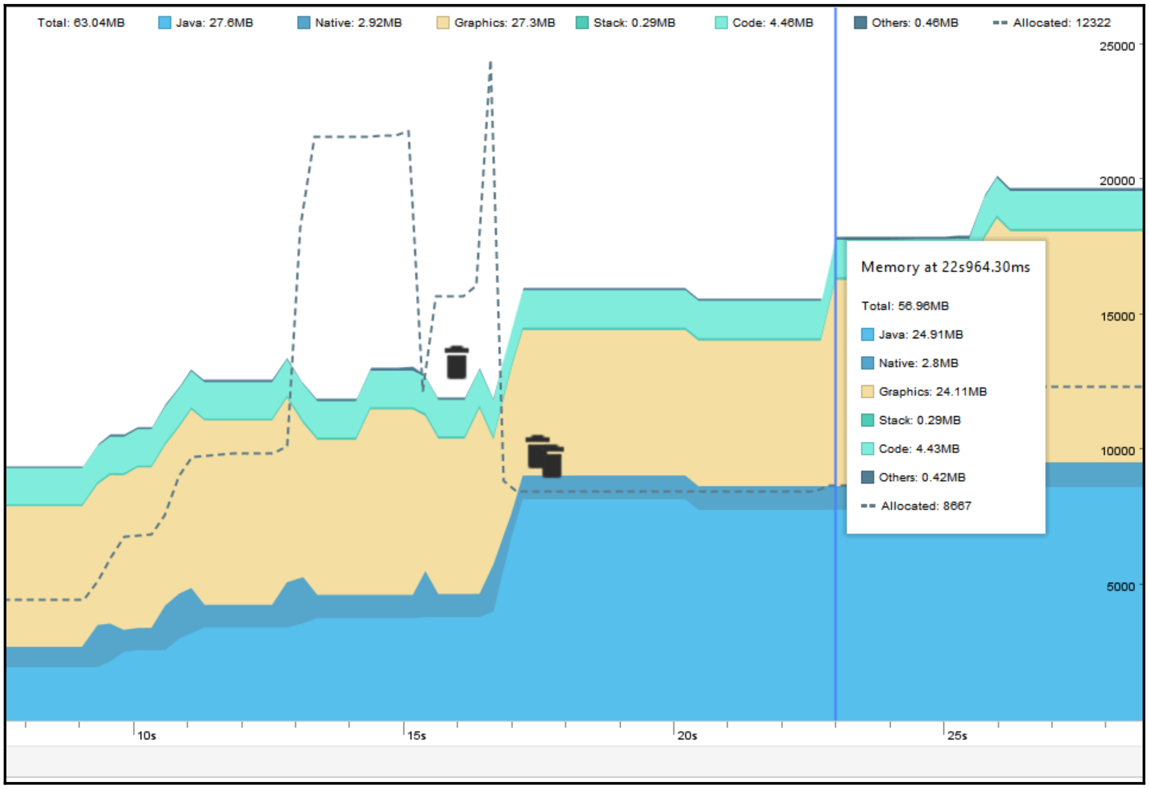
Clicking on any method in these charts will open the source code for that method.

Being able to inspect program flow in detail like this is immensely helpful and can save a lot of unnecessary debugging, but it is not just processor time we need to consider; we also need to keep a close eye on just how much memory our apps are consuming.

Memory profiler

Fully understanding the impact our app has on a device's CPU is just one consideration. As developers, we have to create apps without knowing what the memory capabilities of our target devices are, and, furthermore, we have no way of knowing what other uses these devices are making of their memory at the time our app is running.

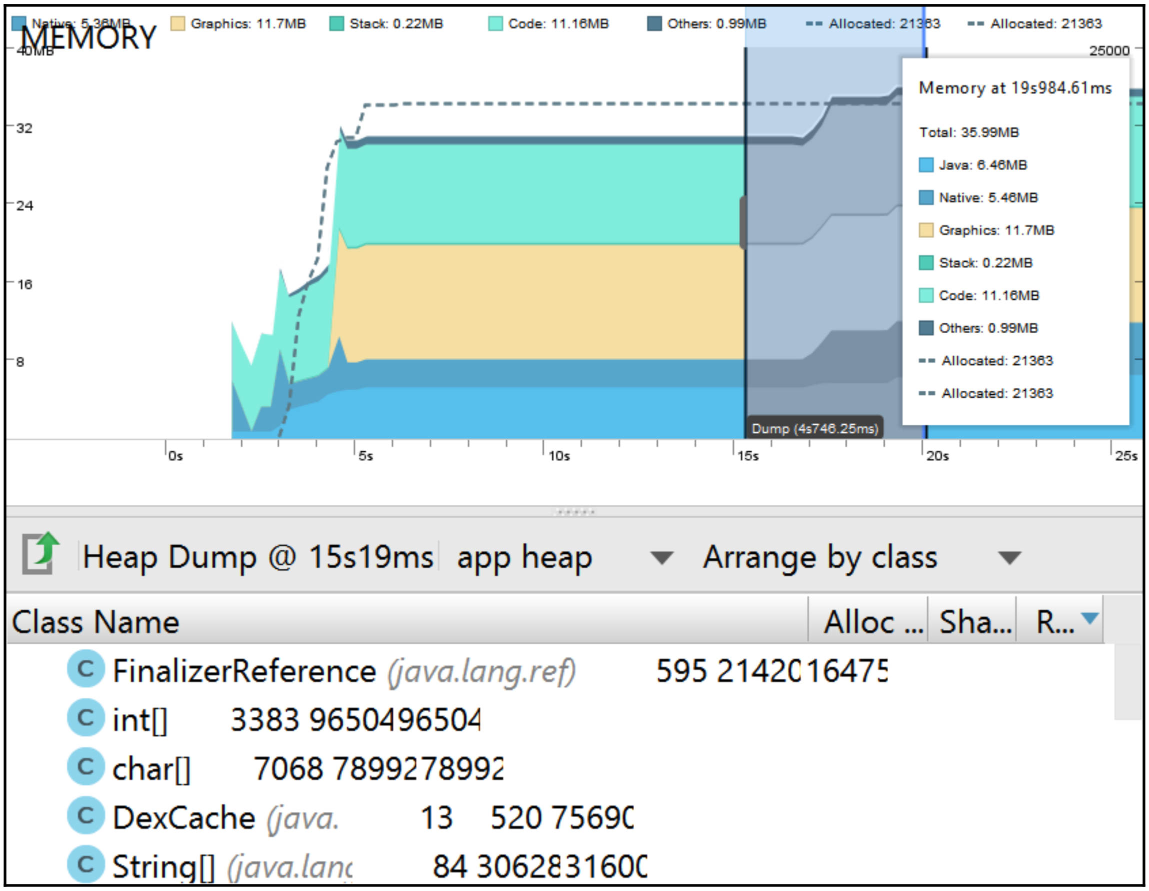
To assist us in planning memory use and avoiding leaks, Android Studio comes equipped with a powerful memory profiler. This allows us to view the Java heap and record memory allocation. Provided you have advanced profiling enabled, the advanced memory profiler can be opened in the same fashion as the processor profiler by clicking anywhere on the live timeline.



The advanced memory profiler

As the preceding image shows, the profiler also displays automatic garbage collections. Such clearing up can also be performed manually with the bin icon in the profiler's toolbar. This also contains buttons for recording memory allocation and capturing a Java heap dump (the **download** icon).

Obtaining a memory dump is as simple as clicking the icon and waiting a moment for the data to be collected. A heap dump displays the objects in use at the time the heap was dumped and is a great way to identify memory leaks. The best time to explore a heap dump is after an extended UI test by looking for objects that should have been discarded that still take up memory.



```
Memory at 19s984.61ms

Total: 35.99MB
Java: 6.46MB
Native: 5.46MB
Graphics: 11.7MB
Stack: 0.22MB
Code: 11.16MB
Others: 0.99MB
Allocated: 21363
Allocated: 21363
```

Heap Dump @ 15s19ms | app heap | Arrange by class

```
Class Name                                Alloc ... Sha... R...
FinalizerReference (java.lang.ref)        595 2142C16475
int[]      3383 9650496504
char[]      7068 7899278992
DexCache (java.        13    520 7569C
String[] (java.lanc        84 30628316OC
```

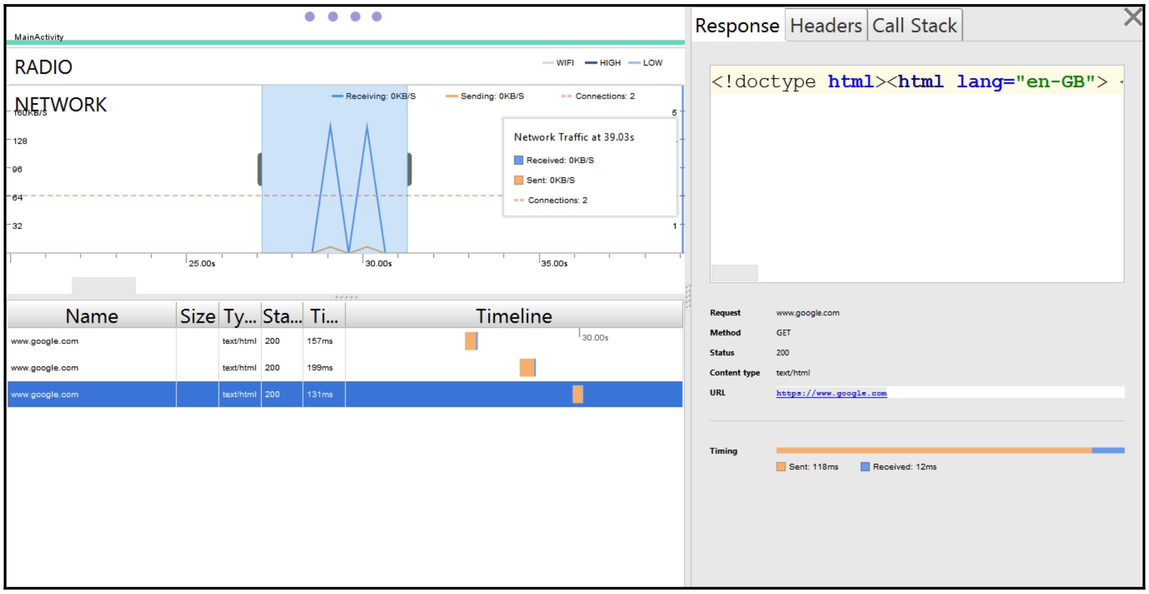
A Java heap dump

Clicking on classes in the dump list will open the appropriate source code in the editor.

Memory dumps like this are very useful for observing how much memory our objects are consuming, but they do not tell us what they are doing with this memory. To see that, we need to record memory allocation. This is accomplished in the same way that CPU recordings were taken, that is, by clicking the record icon. This handy memory inspection tool needs little more explanation, leading us to the third and final profiling tool, the network profile.

Network profiler

There is not much in the way of a difference between the way that this profiler and the previous two operate. Rather than record network activity, simply click and drag over the area of the timeline that interests you. The files involved are then listed in the following pane, with details available when you select them:



```
<!doctype html><html lang="en-GB">
```

Name	Size	Ty...	Sta...	Ti...	Timeline
www.google.com		text/html	200	157ms	
www.google.com		text/html	200	199ms	
www.google.com		text/html	200	131ms	

Request	www.google.com
Method	GET
Status	200
Content type	text/html
URL	https://www.google.com
Timing	🟧 Sent: 118ms 🟦 Received: 12ms

The advanced network profiler

The advanced network profiler provides a great way to identify inefficient network usage. Situations where the network controller has to switch the radios on and off often for small files should be avoided in preference of downloading several small files at once.

The network profiler along with the other two profilers are great examples of a time-saving tool that makes Android Studio such a good choice for developing mobile apps. Thorough testing and fine-tuning of an application can often make all the difference between a mediocre app and a successful one.

Summary

In this chapter, we took a look at the process of testing and profiling our apps. Not only did we see how to take advantage of JUnit integration to test the integrity of our own business logic, but also how to incorporate tools such as Mockito and Espresso to test the platform itself, and resources such as Firebase to test on a wider range of devices.

Besides testing our code and UIs, we need a way of testing our apps, hardware performance and whether there are issues with CPU, memory, or network usage. This is where Android Studio's built-in profiler, which allows us to inspect and record our app performance in great detail, comes in handy.

With our apps now running smoothly and fine-tuned for performance, we can take a look at the final stages of development, building, packaging, and deployment. Android Studio allows us to use the Gradle build system to simply create signed APKs, including those of different flavors as well as simplifying signing and security.

9
Packaging and Distribution

Compiling and building APKs is something we do many times during the course of an application's development, and other than including various dependencies, we have taken our build automation system, Gradle, pretty much for granted. Despite this, it will not have escaped the reader's attention that what Gradle actually does is really quite sophisticated and complex.

One of the reasons that we can take Gradle for granted is the way that it configures each build using a process known as **convention over configuration**. This ensures that, in nearly all cases, Gradle selects the most sensible configuration options for each project. It is when we override these settings that Gradle becomes interesting and useful. For example, we can use it to build mobile and tablet versions of an app from the same Studio project.

Producing a compiled APK file is by no means the final step in our journey, as there is still plenty of testing and analysis that we can do. These processes are assisted greatly by the presence of Android Studio's APK Analyzer.

Once our testing is complete and we are satisfied with our product, we will enter the final stage of the journey by generating signed APK files, ready for release. This step is not an involved process and Android Studio helps the developer every step of the way.

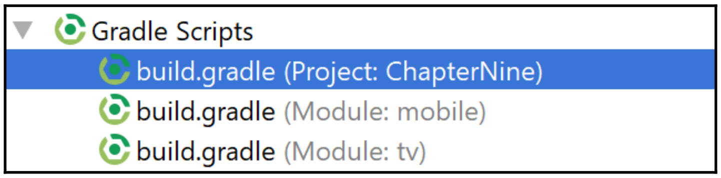
In this chapter, you will learn about:

- Understanding the build process
- Creating product flavors
- Importing Gradle builds from Eclipse
- Analyzing APK files
- Cleaning projects

- Generating a signed APK
- Enrolling for Google Play app signing
- Configuring automatic signing

Gradle build configurations

As the reader will have seen, Gradle scripts generally have a single project (or root) file and one or more module level files:


Gradle Scripts
build.gradle (Project: ChapterNine)
build.gradle (Module: mobile)
build.gradle (Module: tv)

Gradle scripts

We are told not to edit this file in comments in the root script; unless we have configuration options common to all modules, this is best left as-is.

Module-level scripts are of far more interest to us and the following is a breakdown of a typical one.

The first line simply declares the use of the Gradle plugin:

```
apply plugin: 'com.android.application'
```

Next, Android-targeted API level and build tools versions are declared:

```
android {
    compileSdkVersion 27
    buildToolsVersion "27.0.0"
```

The default configuration settings define elements of the Android manifest file, and editing them here will be automatically reflected in the manifest after the next build or sync, as follows:

```
defaultConfig {
     applicationId "com.example.someapp"
     minSdkVersion 21
     targetSdkVersion 27
     versionCode 1
     versionName "1.0"
```

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

The build types section configures **ProGuard**, which is a tool used to minimize and obfuscate our code.

The difference Proguard makes to the size of your APK can often be minimal, but the effects of obfuscation cannot be underestimated and ProGuard can make it as good as impossible for our APKs to be reverse-engineered.

The `buildTypes` section contains instructions on whether and how to run ProGuard on the APK file when a release version of the application is built:

```
buildTypes {
     release {
         minifyEnabled false
         proguardFiles
             getDefaultProguardFile('proguard-android.txt'),
             'proguard-rules.pro'
        }
    }
}
```

ProGuard rules can be edited from the `proguard.rules.pro` file, which overrides the default rules and can be found in `sdk\tools\proguard`.

Note that `minifyEnabled` is set by default to `false`. This is an extremely useful set of functions that strips our code of redundancies, often resulting in far smaller APK files, and should generally be set to `true`.
It is also a good idea to add shrinkResources `true`, which performs a similar action on our resource files.

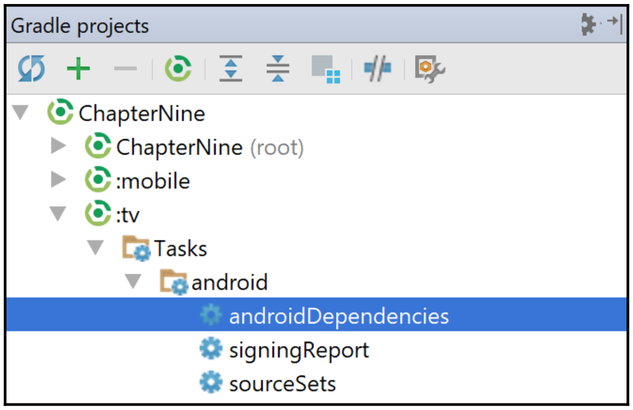
Finally, we will focus on the dependencies section, with which we are already quite familiar. Here, any `.jar` files in the module's `lib` directory are included:

```
dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
    androidTestImplementation('com.android.support
        .test.espresso:espresso-core:2.2.2', {
        exclude group: 'com.android.support',
            module: 'support-annotations'
    })
    implementation 'com.android.support:
        appcompat-v7:26.0.0-beta2'
    testImplementation 'junit:
        junit:4.12'
    implementation 'com.android.support.constraint:
        constraint-layout:1.0.2'
    implementation 'com.android.support:
        design:26.0.0-beta2'
}
```

Command-line options

Many readers who have migrated from other IDEs may well have run Gradle scripts from the command line, and this, of course, is possible with Android Studio, although Studio incorporates this nicely within the workspace so that it is not necessary to exit the IDE to execute commands this way.

There are two handy tool windows to assist us in this task, the **Gradle tool window** and the **Gradle console**, and both are available from the **View** | **Tool Windows** menu.



The Gradle tool window

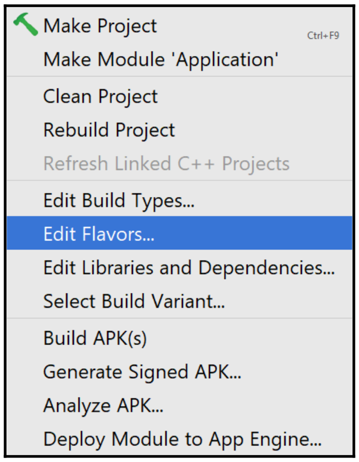
The breakdown in the preceding screenshot provides a nice overview of the role Gradle plays, but to really get to grips with it we will need to work through a simple example, and the following section demonstrates how to configure Gradle to produce different product flavors from a single project.

Product flavors

Generally speaking, there are two reasons for wanting to create customized versions, or flavors, of our applications:

- When we are creating versions for different form factors, such as a mobile phone and a tablet
- When we want two different versions of our app to be available in the Play store, such as paid and free versions

Both of these situations can be catered for by configuring our build files, and this can be done for both `debug` and `release` APKs. New flavors and build types can both be configured using their respective dialogs, which can be found under the **Build** menu:



Build options

As always, it is best to see how these processes operate at first hand. In the following example, we will create two product flavors to represent a free and a paid version of an app. The steps to be followed are:

1. Start a new project in Android Studio with just a single module.
2. Create two **New** | **Directories** in your `values` folder, called `paid` and `free`.
3. These will not be visible in the Explorer under the **Android** label, but can be found by switching to the **Project** view.

TIP

A shortcut to this can be achieved by selecting `values` in the navigation toolbar and selecting `paid` or `free` from the drop-down. This will automatically open the project view and expand it to display our new folders.

4. Create two appropriate `strings.xml` files along these lines:

```
<resources>
    <string name="app_name">Product Flavors Pro</string>
    <string name="version">Pro</string>
</resources>

<resources>
    <string name="app_name">Product Flavors Free</string>
    <string name="version">Free</string>
</resources>
```

5. Open the `build.gradle` file, and complete it like so:

```
apply plugin: 'com.android.application'

android {

    . . .

    defaultConfig {

        . . .

        flavorSelection 'full', 'paid'
        flavorSelection 'partial', 'free'
    }
    buildTypes {
        release {
            . . .

        }
    }

    productFlavors {
        flavorDimensions "partial", "full"

        paid {
            applicationId = "com.example.someapp.paid"
            versionName = "1.0-paid"
            dimension "full"
        }
```

```
                free {
                    applicationId = "com.example.someapp.free"
                    versionName = "1.0-free"
                    dimension "partial"
                }
            }
        }

        dependencies {

            . . .

        }
```

Now use the **Build Variants** tool window to select which of the two flavors is subsequently built.

Many reader will have migrated from the Eclipse IDE to Android Studio. Importing the Eclipse project is relatively straightforward, but importing Gradle build files is less so. This can be achieved with the following `build.gradle` root file:

```
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:3.0.0'
    }
}
apply plugin: 'com.android.application'


android {
     lintOptions {
         abortOnError false
      }

    compileSdkVersion 27
    buildToolsVersion "27.0.0"

        defaultConfig {
            targetSdkVersion 27
        }

    sourceSets {
        main {
            manifest.srcFile 'AndroidManifest.xml'
            java.srcDirs = ['src']
```

```
                resources.srcDirs = ['src']
                aidl.srcDirs = ['src']
                renderscript.srcDirs = ['src']
                res.srcDirs = ['res']
                assets.srcDirs = ['assets']
            }

            debug.setRoot('build-types/debug')
            release.setRoot('build-types/release')
        }
    }
```

Being able to create distinct versions of an APK without having to create separate projects is a great time-saver, and Gradle makes this remarkably simple. Nevertheless, for most of the time, we can just let Gradle get on with its work.

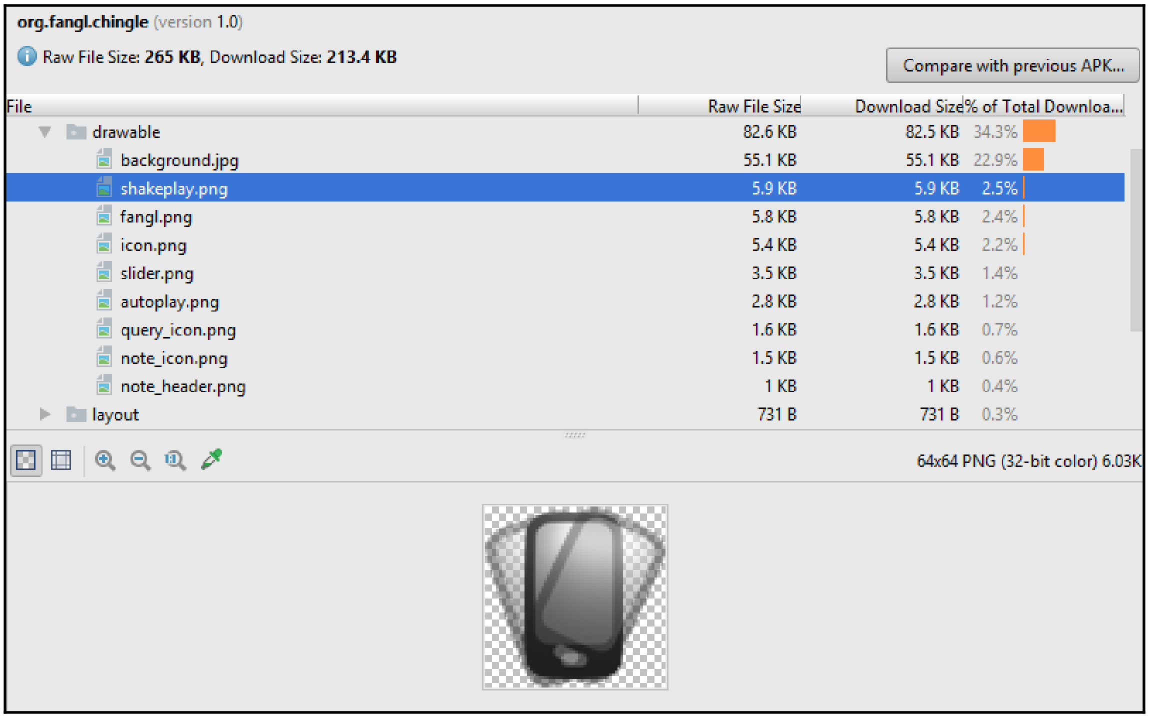
It would be tempting to think that the testing process will be complete with the production of an APK. However, Android Studio provides a marvelous tool that allows us to analyze completed APKs.

An APK analysis

APK Analyzer is one of the handiest features of Android Studio; as its name suggests, it allows us to analyze APK files themselves, even performing a certain amount of reverse engineering by extracting resources and XML and allowing us to compare different versions.

The APK Analyzer can also be found in the **Build** menu, under **Analyze APK...**. Every time we run a project on a device or emulator, a debug APK is generated. This can be found in your project directories under; `\SomeProject\App\build\outputs\apk\debug`.

The Analyzer displays its output, as follows:


org.fangl.chingle (version 1.0)

Raw File Size: **265 KB**, Download Size: **213.4 KB**

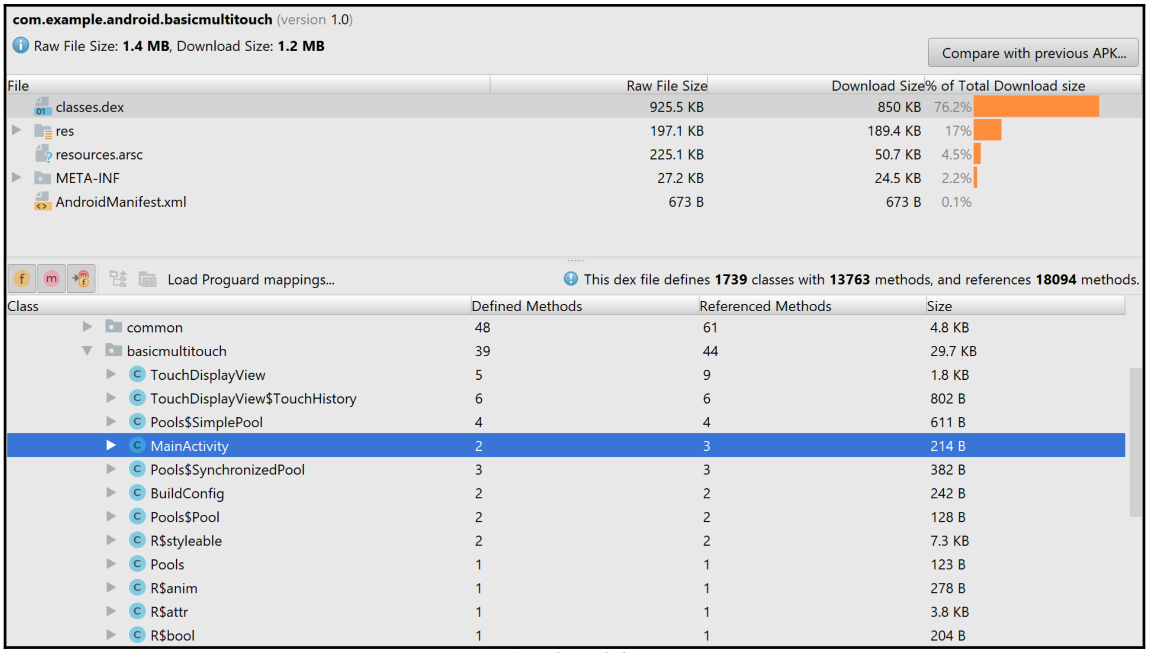
File	Raw File Size	Download Size	% of Total Downloa...
drawable	82.6 KB	82.5 KB	34.3%
background.jpg	55.1 KB	55.1 KB	22.9%
shakeplay.png	5.9 KB	5.9 KB	2.5%
fangl.png	5.8 KB	5.8 KB	2.4%
icon.png	5.4 KB	5.4 KB	2.2%
slider.png	3.5 KB	3.5 KB	1.4%
autoplay.png	2.8 KB	2.8 KB	1.2%
query_icon.png	1.6 KB	1.6 KB	0.7%
note_icon.png	1.5 KB	1.5 KB	0.6%
note_header.png	1 KB	1 KB	0.4%
layout	731 B	731 B	0.3%

64x64 PNG (32-bit color) 6.03K

APK analysis

The Analyzer's output contains a wealth of information, beginning with its size and its compressed Play store size. It is possible to see at a glance which resources take up the most room and rectify this where possible, for example by using vectors instead of bitmaps.

The `classes.dex` file allows us to explore the memory consumed by our classes and imported libraries.



com.example.android.basicmultitouch (version 1.0)

Raw File Size: **1.4 MB**, Download Size: **1.2 MB**

Compare with previous APK...

File	Raw File Size	Download Size	% of Total Download size
classes.dex	925.5 KB	850 KB	76.2%
res	197.1 KB	189.4 KB	17%
resources.arsc	225.1 KB	50.7 KB	4.5%
META-INF	27.2 KB	24.5 KB	2.2%
AndroidManifest.xml	673 B	673 B	0.1%

Load Proguard mappings...

This dex file defines **1739** classes with **13763** methods, and references **18094** methods.

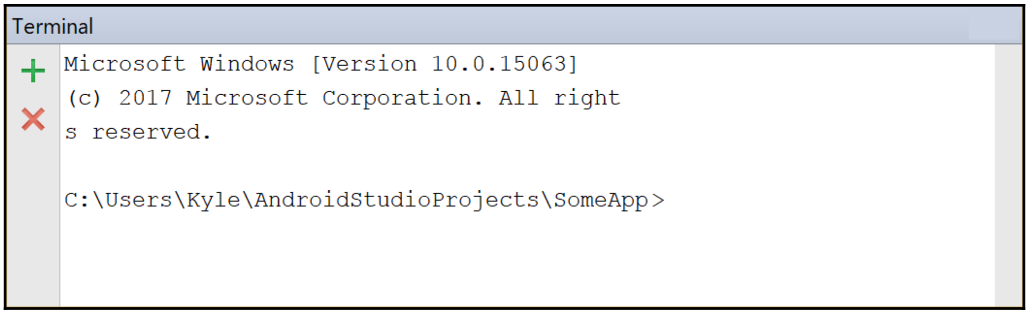
Class	Defined Methods	Referenced Methods	Size
common	48	61	4.8 KB
basicmultitouch	39	44	29.7 KB
TouchDisplayView	5	9	1.8 KB
TouchDisplayView$TouchHistory	6	6	802 B
Pools$SimplePool	4	4	611 B
MainActivity	2	3	214 B
Pools$SynchronizedPool	3	3	382 B
BuildConfig	2	2	242 B
Pools$Pool	2	2	128 B
R$styleable	2	2	7.3 KB
Pools	1	1	123 B
R$anim	1	1	278 B
R$attr	1	1	3.8 KB
R$bool	1	1	204 B

APK class analysis.

One of the most useful features of the Analyzer is the ability to compare two APKs side by side, which can be achieved using the button in the top-right corner of the window.

If the APK Analyzer is not enough, then there is the **Profile or Debug APK...** entry in the main **File** menu. This opens up a new project and disassembles the APK so that it can be fully explored and even debugged.

Besides **MakeBuild** and **Analyze**, the **Build** menu has other useful entries, for example, the **Clean Project** item removes build artifacts from the build directory if we want to share them with colleagues and collaborators across the internet. For a deeper clean, open the command prompt in your project folder using your native file explorer or select **Terminal** from the **File | Tools Window** menu.


Terminal

```
Microsoft Windows [Version 10.0.15063]
(c) 2017 Microsoft Corporation. All right
s reserved.

C:\Users\Kyle\AndroidStudioProjects\SomeApp>
```

The following command will clean your project:

```
gradlew clean
```

The amount of space this operation can save is often very impressive. Of course, the next time you build the project, it will take as long as it did the first time.

For the vast majority of the development cycle, we are only concerned with debug versions of our APKs, but sooner or later we will need to produce an APK fit for release.

Publishing applications

Developing a mobile application, even a relatively simple one, is a lengthy process, and once we have tested all our code, ironed out any bumps, and polished our UI, we want to be able to get our product on the shelf as quickly and simply as possible. Android Studio has all these processes incorporated into the workspace.

As the reader will know, the first step toward publication is generating a signed APK.

Generating a signed APK

All Android applications require a digital certificate before they can be installed on a user's device. These certificates follow the usual pattern of including a public key with every download that corresponds with our own private key. This process guarantees authenticity for the user and prevents anyone else producing updates of other developers' work.

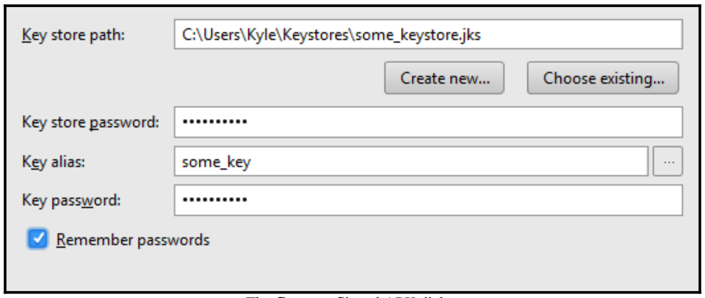
During development, the IDE automatically generates a debug certificate for us, for use only during development. These certificates can be found in: `\SomeApp\build\outputs\apk\debug`

There are two ways to create these identity certificates: we can manage our own keystore, or we can use Google Play App Signing. We will take a look at both of these techniques now, beginning with self-management.

Managing keystores

Whether we are managing our own keystore or Google does it for us, the process begins in the same way, as the following steps demonstrate:

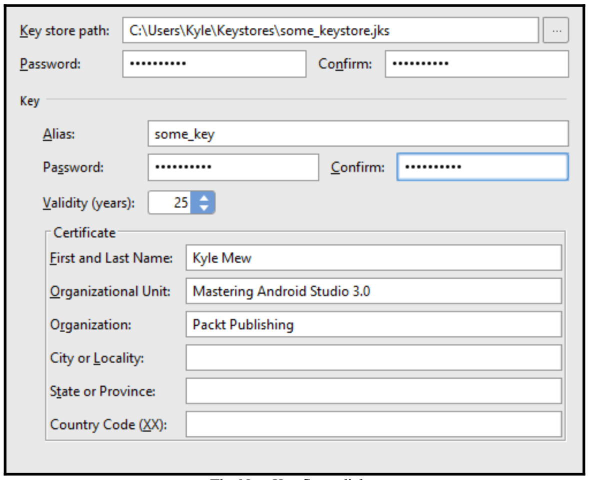
1. Click on the **Generate Signed APK...** entry in the **Build** menu.
2. Complete the following dialog, using very strong passwords.



Key store path:	C:\Users\Kyle\Keystores\some_keystore.jks
Key store password:	••••••••••
Key alias:	some_key
Key password:	••••••••••
✓	Remember passwords

The Generate Signed APK dialog

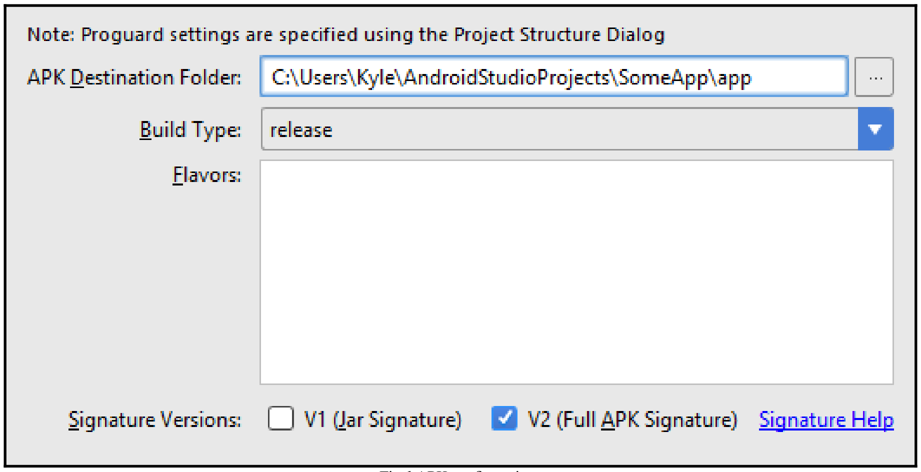
3. If you are creating a new keystore, you will be presented with the **New Key Store** dialog, which must be completed along these lines:



Key store path:	C:\Users\Kyle\Keystores\some_keystore.jks
Password:	••••••••••
Confirm:	••••••••••
Key	
Alias:	some_key
Password:	••••••••••
Confirm:	••••••••••
Validity (years):	25
Certificate	
First and Last Name:	Kyle Mew
Organizational Unit:	Mastering Android Studio 3.0
Organization:	Packt Publishing
City or Locality:	
State or Province:	
Country Code (XX):	

The New Key Store dialog

4. The final dialog allows you to select **Build Type** and **APK Destination Folder**, as well as any flavors you may have created. Ensure that you select the **V2 (Full APK Signature)** box.



Note: Proguard settings are specified using the Project Structure Dialog

APK Destination Folder: C:\Users\Kyle\AndroidStudioProjects\SomeApp\app

Build Type: release

Flavors:

Signature Versions: [ ] V1 (Jar Signature) [✓] V2 (Full APK Signature) Signature Help

Final APK configurations

5. The final APK(s) will be stored in, `...\app\release\app-release.apk`.

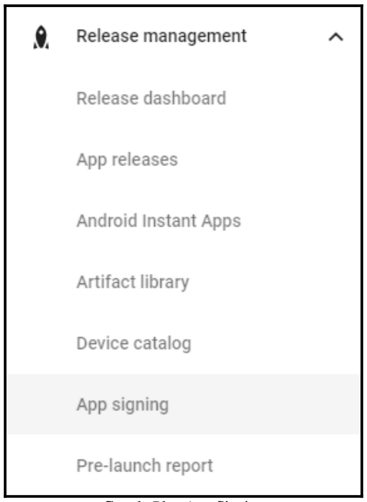
The selection of the V2 signature version is an important inclusion. Introduced in API level 24 (Android 7.0), the Signature Theme v2 provides faster installation and protects against hackers reverse-engineering our APKs. Unfortunately, it doesn't work for all builds but is well worth applying when it does.

Managing our own key stores is how it has generally always been done, and provided we keep our keys secure, it is a perfectly acceptable way to manage our certificates. Nevertheless, App signing using Google Play offers some distinct advantages and is well worth considering.

Google Play app signing

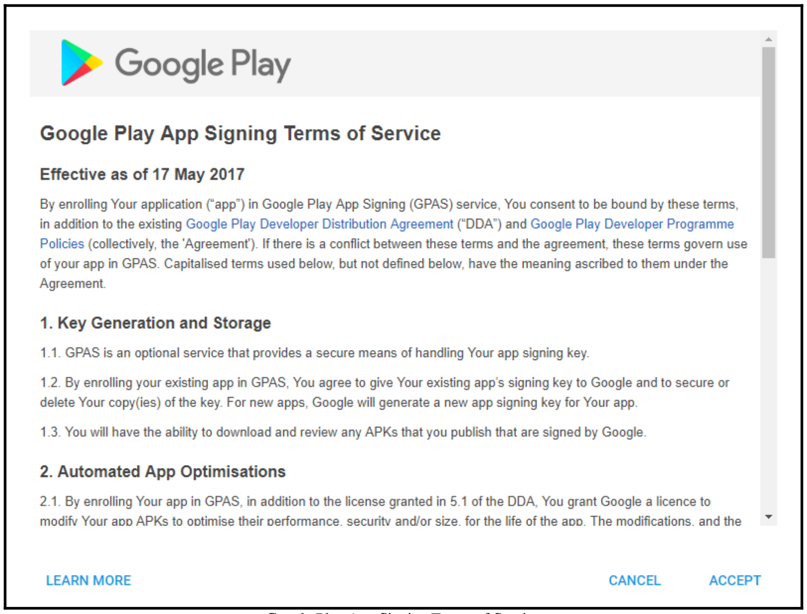
The main advantage of using Google Play to sign our apps is that Google maintain key information and if, by some misfortune, we lose ours, this can be retrieved. One very important thing to note about this system is that, once adopted, there is **no opt-out option**. This is because this would itself represent a possible security breach.

To enable Google Play app signing, prepare your signed APK as described in the preceding steps, and then open the Google developer console. Google Play app signing is available from the **Release management** menu.



Google Play App Signing

This will then open the **Terms of Service** dialog.



Google Play App Signing Terms of Service

Effective as of 17 May 2017

By enrolling Your application ("app") in Google Play App Signing (GPAS) service, You consent to be bound by these terms, in addition to the existing Google Play Developer Distribution Agreement ("DDA") and Google Play Developer Programme Policies (collectively, the 'Agreement'). If there is a conflict between these terms and the agreement, these terms govern use of your app in GPAS. Capitalised terms used below, but not defined below, have the meaning ascribed to them under the Agreement.

1. Key Generation and Storage

1.1. GPAS is an optional service that provides a secure means of handling Your app signing key.

1.2. By enrolling your existing app in GPAS, You agree to give Your existing app's signing key to Google and to secure or delete Your copy(ies) of the key. For new apps, Google will generate a new app signing key for Your app.

1.3. You will have the ability to download and review any APKs that you publish that are signed by Google.

2. Automated App Optimisations

2.1. By enrolling Your app in GPAS, in addition to the license granted in 5.1 of the DDA, You grant Google a licence to modify Your app APKs to optimise their performance, security and/or size, for the life of the app. The modifications, and the

Google Play App Signing Terms of Service

To enroll with **Google Play App Signing Terms of Service**, you will need to follow the steps outlined here:

1. First, encrypt your signing key, using the **Play Encrypt Private Key** (PEPK) tool, which can be downloaded from the console in the left-hand navigation bar.
2. Create a second upload key and register it with Google.
3. Use this key to sign your app for publication and upload it to Google Play.
4. Google then uses this to authenticate you and then sign the app with the encrypted key.

More information on the process can be found by clicking on **LEARN MORE** on the **Terms of Service** dialog.

Providing that we are happy to commit ourselves, enrolling with the app signing service provides a more secure system than the traditional approach. However, when we decide to sign our apps, it is always good to have more control over the process, for example by configuring Gradle.

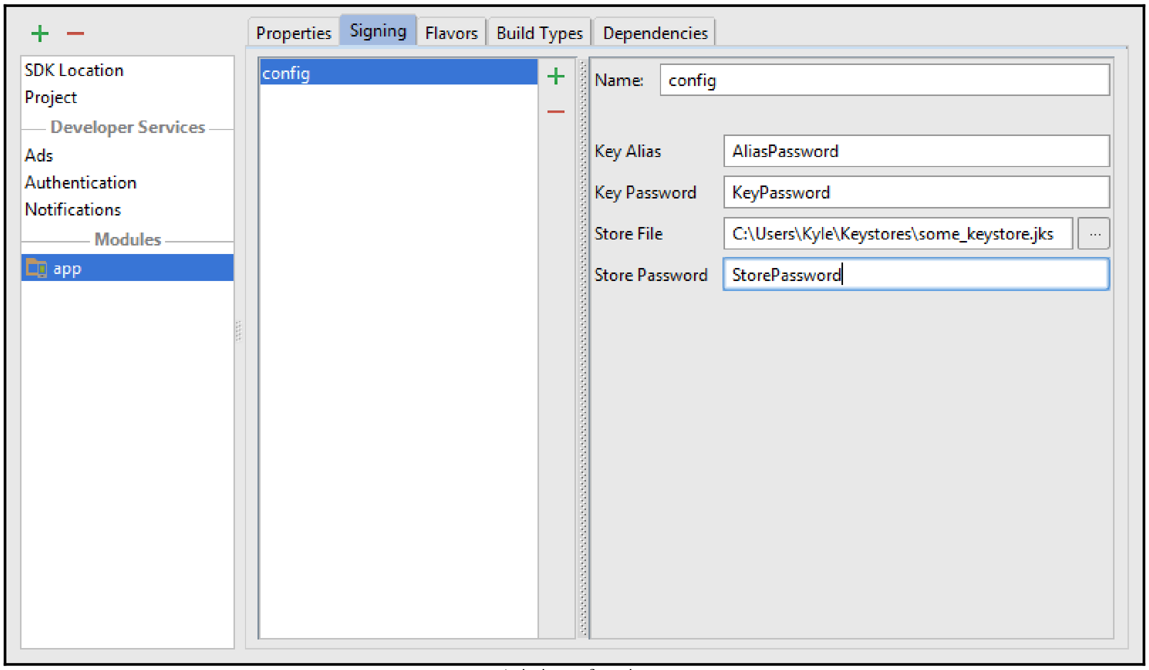
Automatic signing

Signing configurations are created automatically for us each time we sign an app or flavor. Fortunately, these configurations can be custom built to suit our specific purposes. For example, one might want to automatically sign an app on each build. Android Studio makes this possible through **Project Structure...,** which can be found in the main **File** menu.

The following exercise demonstrates how to automatically sign a release version of an application:

1. Open the **Project Structure** dialog, as explained previously or by selecting **Open Module Settings** from your module's drop-down menu in the Project Explorer, or by selecting the module and pressing *F4*.

2. Select the **Signing** tab, click on the **+** icon, and fill in the appropriate fields.



Name: config

Key Alias: AliasPassword

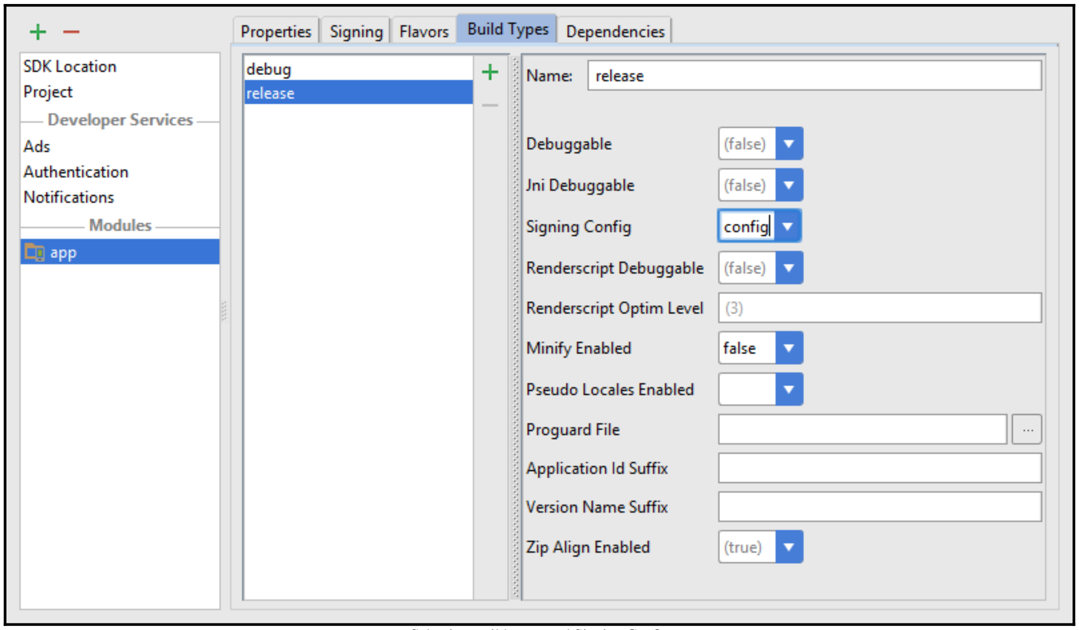
Key Password: KeyPassword

Store File: C:\Users\Kyle\Keystores\some_keystore.jks

Store Password: StorePassword

A signing configuration

3. Next, open the **Build Types** tab and select the **debug** or **release** type, enter the **Signing Config** field, as shown in the following screenshot, and any other settings, such as enabling **Minify**.



Name:	release
Debuggable	(false)
Jni Debuggable	(false)
Signing Config	config
Renderscript Debuggable	(false)
Renderscript Optim Level	(3)
Minify Enabled	false
Pseudo Locales Enabled	
Proguard File	
Application Id Suffix	
Version Name Suffix	
Zip Align Enabled	(true)

Selecting Build Type and Signing Config

There are a few other preparations to make before publication. A few more tests must be performed on the release APK, and various promotional resources and materials need to be gathered, but from an Android Studio point of view, the signed, released APK is pretty much the finished product.

Summary

The production of a signed APK is the final step in what will have been a long journey. Beginning with nothing more than an idea, each application will have grown through countless cycles of design, development, and testing before finally being placed on a shelf in a store such as Android Play Store.

Android Studio has been designed to assist developers on every step of this journey, and one of the reasons that Google has put so much into the product; is because, by investing in the developers of the future and making it easier for them to put ideas into practice, the Android platform can only become better.

In this book we have explored the only IDE created specifically for Android development, and we have seen how this specialized approach provides many benefits to the developer. The visual and intuitive nature of the Layout Editor and the way that Constraint Layouts can be designed with little more than a click or two of the mouse, will leave most of us feeling quite sorry for those developers still using other IDEs.

Coding too becomes far less of a chore with Android Studio's simple code completion and refactoring features. Add this to the incorporation of Kotlin as an official development language, and choosing Android Studio will seem to many mobile developers to be the only choice. Even compiling and testing apps can be quicker and easier with Android Studio and of course developing for new and exciting form factors, such as wearables and IoT, is made easier with the tools provided by the IDE.

Throughout this book we have explored the advantages of choosing Android Studio 3. The IDE is of course a work in progress and no doubt, as a project that Google appears to be seriously invested in, it will continue to grow and improve for many years to come. In many ways Android Studio 3 is only the beginning and it is hoped that this book will help the reader master one small step of that journey.

Index