Android Studio IDE Quick Reference

A Pocket Guide to Android Studio Development

Ted Hagos

Apress®

Android Studio IDE Quick Reference

A Pocket Guide to Android Studio Development

Ted Hagos

Apress®

Android Studio IDE Quick Reference: A Pocket Guide to Android Studio Development

Ted Hagos
Manila, National Capital Region, Philippines

ISBN-13 (pbk): 978-1-4842-4952-9 ISBN-13 (electronic): 978-1-4842-4953-6
https://doi.org/10.1007/978-1-4842-4953-6

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Steve Anglin
Development Editor: Matthew Moodie
Coordinating Editor: Mark Powers

Cover designed by eStudioCalamar

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a Delaware corporation.

For information on translations, please e-mail editorial@apress.com; for reprint, paperback, or audio rights, please email bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484249529. For more detailed information, please visit www.apress.com/source-code.

Printed on acid-free paper

For Adrianne and Stephanie.

Contents

About the Author

Ted Hagos is the CTO and Data Protection Officer of RenditionDigital International, a software development company based out of Dublin. Before he joined RDI, he had various software development roles and also spent time as trainer at IBM Advanced Career Education, Ateneo ITI, and Asia Pacific College. He spent many years in software development dating back to the days of Turbo C, Clipper, dBase IV, and Visual Basic. Eventually, he found Java and spent many years working with it. Nowadays, he's busy with full-stack JavaScript and Android.

About the Technical Reviewers

Marcos Placona is a developer evangelist at Twilio and a GDE. He serves communities in London and all over Europe. He is passionate about technology and security and he spends a great deal of his time building mobile and web apps and occasionally connecting them to physical devices. Marcos is a great believer in open source projects. When he's not writing open source code, he's probably blogging about code at https://androidsecurity.info, https://androidthings.rocks, or https://realkotlin.com. He's also a great API enthusiast and believes they bring peace to the software engineering world.

Massimo Nardone has more than 24 years of experience in security, web/mobile development, cloud, and IT architecture. His true IT passions are security and Android. He has been programming and teaching how to program with Android, Perl, PHP, Java, VB, Python, C/C++, and MySQL for more than 20 years. He holds a Master of Science degree in Computing Science from the University of Salerno, Italy. He has worked as a Project Manager, Software Engineer, Research Engineer, Chief Security Architect, Information Security Manager, PCI/SCADA Auditor, and Senior Lead IT Security/Cloud/SCADA Architect. His technical skills include security, Android, cloud, Java, MySQL, Drupal, Cobol, Perl, web and mobile development, MongoDB, D3, Joomla, Couchbase, C/C++, WebGL, Python, Pro Rails, Django CMS, Jekyll, Scratch, etc. He was a visiting lecturer and supervisor for exercises at the Networking Laboratory of the Helsinki University of Technology (Aalto University). He holds four international patents (PKI, SIP, SAML, and Proxy areas). He currently works as the Chief

Information Security Officer (CISO) for Cargotec Oyj and he is member of the ISACA Finland Chapter Board. Massimo has reviewed more than 45 IT books for different publishers; he is the coauthor of *Pro JPA in Java EE 8* (Apress, 2018), *Beginning EJB in Java EE 8* (Apress, 2018), and *Pro Android Games* (Apress, 2015).

Acknowledgments

To Stephanie and Adrianne, my thanks and my love.

Introduction

Welcome to *Android Studio IDE Quick Reference*. I wrote this book to serve as a handy reference to the notable capabilities of Android Studio.

This book is comprised of 13 short chapters. Each chapter breezes through some features of Android Studio. While the book is focused on the IDE, many chapters actually deal with Android programming as well, so you're going to see some code samples.

Who This Book Is For

This book is for the experienced programmer who needs a quick reference on how to do some common tasks in Android Studio IDE. The code examples are in Java, so it's for those devs who build Android apps in Java. This book might also work for someone new to Android Studio but not new in Android development—such as an Eclipse user who wants to try out Android Studio.

Source Code

Source code for this book can be downloaded by clicking the Download Source Code button located at www.apress.com/us/book/9781484249529.

Chapter 1

Setup

What this chapter covers:

- Installing Android Studio
- Setting up the IDE
- Basic configuration

Developing applications for Android was not always as convenient as it is today. When Android 1.0 was released in 2008, what developers got by way of a development kit was little more than a handful of command line tools and Ant build scripts. Building apps with Vim, Ant, and other command line tools wasn't so bad if you're used to that kind of thing, but many developers were not used to it. The lack of IDE capabilities like code hinting, project setups, and integrated debugging was somewhat of a barrier to entry.

Thankfully, Android Development Tools (ADT) for the Eclipse IDE were released, also in 2008. Eclipse was, and still is, a favorite and the IDE of choice for many Java developers. It was natural that it would also be the go-to IDE for Android developers.

From 2009 to 2012, Eclipse remained the preferred IDE for Android development. The Android SDK has undergone both major and incremental changes in structure and in scope. In 2009, the SDK manager was released; it was used to download tools, individual SDK versions, and Android images for use with the emulator. In 2010, additional images were released for the ARM processor and x86 CPUs.

2012 was a big year because Eclipse and the ADT were finally bundled. Until that time, developers had to install Eclipse and ADT separately, and the installation process wasn't always smooth. So the bundling

T. Hagos, *Android Studio IDE Quick Reference*,
https://doi.org/10.1007/978-1-4842-4953-6_1

of the two made it a whole lot easier to get started with Android development. 2012 also marked the last year of Eclipse being the dominant IDE for Android.

In 2013, Android Studio (AS) was released. To be sure, it was still beta, but the writing on the wall was clear: it would be the official IDE for Android development. Android Studio is based on JetBrains's IntelliJ. IntelliJ is a commercial Java IDE that also has a community (non-paid) version. This version serves as the base for Android Studio.

Setting Up Android Studio

Android Studio's version at the time of writing is 3.2.1; hopefully the version won't be very different by the time you read this book. You can download it from https://developer.android.com/studio. It's available for Windows (both 32- and 64-bit), macOS, and Linux. I ran the installation instructions on macOS (Mojave), Windows 10 64-bit, and Ubuntu 18. I work primarily in a macOS environment, which explains why most of the screen grabs for this book looks like macOS. Android Studio looks, runs, and feels (mostly) the same on all three platforms, with very minor differences like key bindings and the main menu bar in macOS.

Before we go further, let's look at the system requirements for Android Studio. At a minimum, you'll need the following:

- Microsoft Windows 7/8/10 (32 or 64-bit) or
- macOS 10.10 (Yosemite or higher) or
- Linux (Gnome or KDE Desktop), Ubuntu 14.04 or higher (64-bit but capable of running 32-bit applications)
- GNU C Library (glibc 2.19 or later) if you're on Linux

For hardware, your workstation needs to have at least

- 3GB RAM (8GB or more recommended)
- 2GB of available HDD space
- 1280 x 800 minimum screen resolution

This list came from the official Android website (developer.android.com/studio) and, of course, more is better. If you can snag 16GB RAM, 512GB SSD (or bigger), and a full HD (or UHD) monitor, that wouldn't be bad—not at all.

And now we get to the JDK (Java Development Kit) requirement. Starting with Android Studio 2.2, the installer comes with OpenJDK embedded. This way, a beginner programmer won't have to bother with the installation

of a separate JDK, but you can still install a separate JDK if that's your preference. In this book, I'll assume that you will use the embedded OpenJDK that comes with Android Studio.

Download the installer from https://developer.android.com/studio/ and get the proper binary file for your platform.

If you're in macOS, do the following:

1. Unpack the installer zipped file.
2. Drag the application file into the `Applications` folder.
3. Launch Android Studio.
4. Android Studio will prompt you to import some settings if you have a previous installation. You can import that—it's the default option.

> **Note** If you have an existing installation of Android Studio, you can keep using that version and still install Android Studio 3. It can coexist with your existing version of Android Studio because its settings will be kept in a different directory.

If you're in Windows, do the following:

1. Unzip the installer file.
2. Move the unzipped directory to a location of your choice, such as `C:\Users\myname\AndroidStudio`.
3. Drill down to the `AndroidStudio` folder. Inside it is the `studio64.exe` file. This is the file you need to launch. It's a good idea to create a shortcut for this file. If you right-click `studio64.exe` and choose Pin to Start Menu, you can make Android Studio available from the Windows Start menu. Alternatively, you can also pin it to the taskbar.

The Linux installation requires a bit more work than simply double-clicking and following the installer prompts. In future releases of Ubuntu (and its derivatives), this might change and become as simple and frictionless as its Windows and macOS counterparts, but for now, you need to do some tweaking. The extra activities on Linux are mostly because AS needs some 32-bit libraries and hardware acceleration.

> **Note** The installation instructions in this section are meant for Ubuntu 64-bit and other Ubuntu derivatives (e.g. Linux Mint, Lubuntu, Xubuntu, Ubuntu MATE, etc.). I chose this distribution because it is a very common Linux flavor, so readers of this book will be using this distribution. If you are running a 64-bit version of Ubuntu, you will need to pull some 32-bit libraries in order for AS to function well.

To start pulling the 32-bit libraries for Linux, run the following commands in a terminal window:

```
sudo apt-get update && sudo apt-get upgrade -y
sudo dpkg --add-architecture i386
sudo apt-get install libncurses5:i386 libstdc++6:i386 zlib1g:i386
```

When all the prep work is done, you need to do the following:

1. Unpack the downloaded installer file. You can unpack the file using command line tools or the GUI tools. You can, for example, right-click the file and select the "Unpack here" option, if your file manager has it.
2. After unzipping the file, rename the folder to `AndroidStudio`.
3. Move the folder to a location where you have read, write, and execute privileges. Alternatively, you can also move it to `/usr/local/AndroidStudio`.
4. Open a terminal window, go to the `AndroidStudio/bin` folder, and run the `./studio.sh` command.
5. At first launch, Android Studio will ask if you want to import some settings. If you have installed a previous version of Android Studio, you may want to import those settings.

Configuring Android Studio

If this is the first time you've installed Android Studio, you may want to configure a couple of things before diving into coding work. In this section, I'll walk you through the following:

- Acquiring some software that you'll need in order to create programs that target specific versions of Android
- Making sure you have all the SDK tools you need
- And, optionally, changing the way you get updates

Launch the IDE if you haven't done so yet, and click the Configure option, as shown in Figure 1-1. Choose Preferences from the drop-down list.

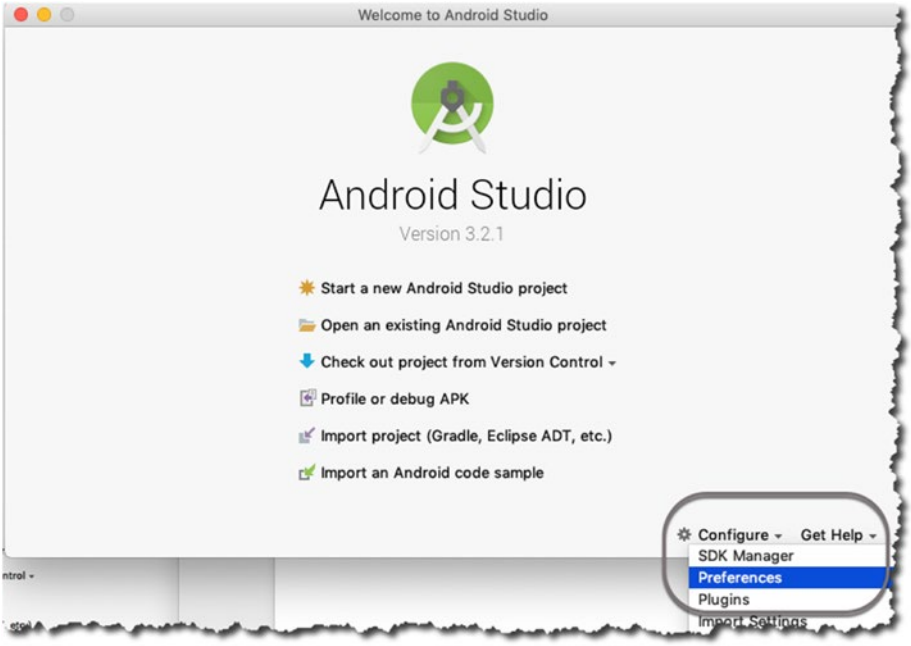

Figure 1-1. *Go to Preferences from Android Studio's opening dialog*

Clicking the Preferences option opens the Preferences dialog, as shown in Figure 1-2. On the left-hand side of the dialog, choose the Android SDK option.

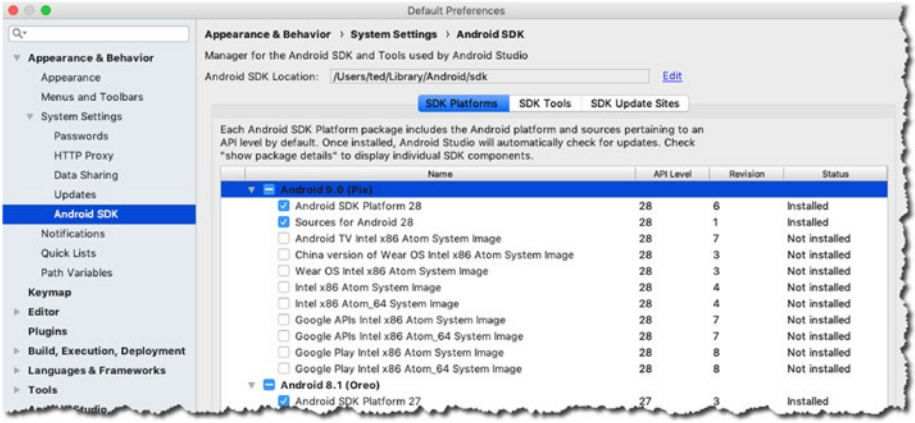

Figure 1-2. *SDK platforms*

When you get to the SDK window, enable the Show Package Details option so you can see a more detailed view of each API level. You don't need to download everything in the SDK window. You will get only the items you need.

SDK levels or platform numbers are specific versions of Android. Android 9 (Pie) is API level 28, Android 8 (Oreo) is API levels 26 and 27, and Nougat is API levels 24 and 25. You don't need to memorize the platform numbers, at least not anymore, because the IDE shows the platform number with the corresponding Android nickname.

Download the API levels you want to target for your applications, but for the purpose of this book, please download API level 27 (Oreo). It's what you will use for the sample projects. Make sure that together with the platforms, you also download the Google APIs Intel x86 Atom_64 System Image. You will need it when you get to the part where you test-run your applications.

Choosing an API level may not be a big deal right now because at this point you're working with practice apps. When you plan to release your application to the public, you may not be able to take this choice lightly, though. Choosing a minimum SDK or API level for your app will determine how many people will be able to use your application. At the time of writing, 25% of all Android devices are using Marshmallow, 22% for Nougat, and 4% for Oreo. These stats are from dashboard page of `developer.android.com`. It's a good idea to check these stats from time to time at `http://bit.ly/droiddashboard`.

Figure 1-3 shows the SDK Tools section.

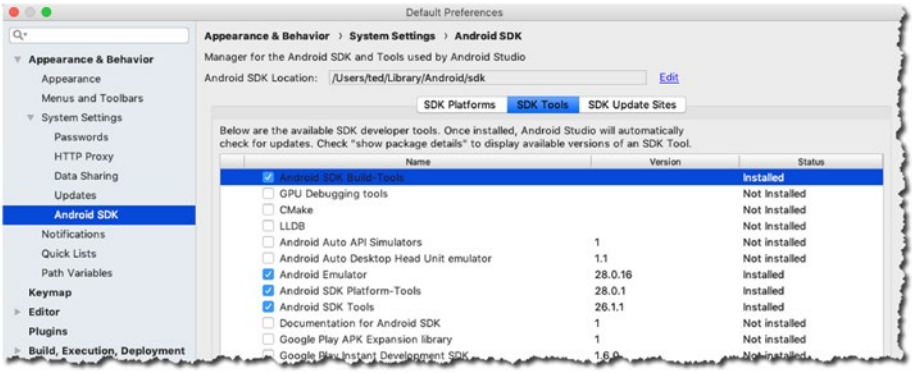

Figure 1-3. SDK Tools section

You don't generally have to change anything in this window, but it wouldn't hurt to check if you have the tools, shown in the list below, marked as Installed:

- Android SDK Build Tools
- Android SDK Platform Tools
- Android SDK Tools
- Android Emulator
- Support Repository
- HAXM Installer

Checking for these tools ensures that you get tools like adb, sqlite, aapt, and zipalign. These tools help with debugging, creating builds, working with databases, running emulations, and so on.

> **Note** If you are on the Linux platform, you cannot use HAXM even if you have an Intel processor. KVM will be used in Linux instead of HAXM.

Once you're happy with your selection, click the OK button to start downloading the packages.

The last configuration check you need to do is to set the update channel. It's in the same Preferences window. Click the Updates item on the right-hand side to show the Updates settings, as shown in Figure 1-4.

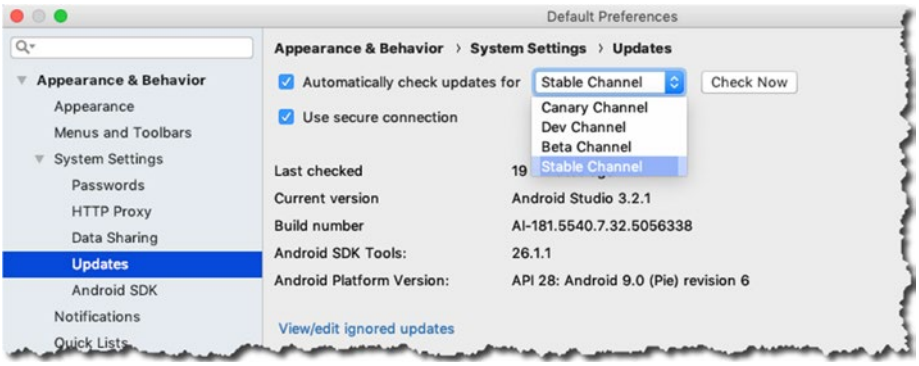

Figure 1-4. *Updates*

Android Studio is configured by default to get updates from the channel where you originally downloaded the installer. Since you downloaded the installer from the stable channel, it will get its update from that channel by default. You can change the channel to one of these four:

- **Canary channel**: This channel is for bleeding edge releases. Because it can be updated as often as every week, you don't want to use it for production code.
- **Dev channel**: This channel is just like the Canary channel but a bit more stable. You still don't want to use it for production.
- **Beta channel**: This channel contains release candidates. The devs are basically waiting for feedback before it gets fed to the stable channel.
- **Stable channel**: This is the official stable release and it is suitable for production work.

Hardware Acceleration

As you write your app, it's useful to test and run it from time to time in order to get immediate feedback and find out if it is running as expected, or if it is running at all. To do this, you will use either a physical or a virtual device. Each option has its pros and cons and you don't have to choose one over the other; in fact, you will have to use both options eventually.

An Android Virtual Device (AVD) is an emulator where you can run your apps. Running on an emulator can sometimes be slow; this is why Google and Intel came up with HAXM, an emulator acceleration tool that makes testing your app a bit more bearable. This is definitely a boon to developers. That is, if you are using a machine that has an Intel processor that supports virtualization and if you are not on Linux. But don't worry if you're not lucky enough to fall into that part of the pie; there are ways to achieve emulator acceleration in Linux, as you'll see later.

macOS users probably have it the easiest because HAXM is automatically installed with AS3. You don't have to do anything to get it because the AS3 installer took care of it for you.

Windows users can get HAXM either by

- Downloading it from `https://software.intel.com/en-us/android`. Install it like you would any other Windows software: double-click and follow the prompts.
- Alternatively, you can get HAXM via AS3's SDK manager. This is the recommended method.

For Linux users, the recommended software is KVM instead. KVM (Kernel-based Virtual Machine) is a virtualization solution for Linux. It contains virtualization extensions (Intel VT or AMD-V).

To get KVM, you need to pull some software from the repos. But even before you can do that, you need to do the following first:

1. Make sure that virtualization is enabled on your BIOS or UEFI settings. Consult your hardware manual on how to get to these settings. It usually involves shutting down the PC, restarting it, and pressing an interrupt key like F2 or DEL as soon as you hear the chime of your system speaker, but like I said, consult your hardware manual.
2. Once you've made your changes and rebooted to Linux, find out if your system can run virtualization. This can be accomplished by running the following command from a terminal: `egrep -c '(vmx|svm)' /proc/cpuinfo`. If the result is a number higher than zero, you can go ahead with the installation.

To install KVM, type the commands shown in example 1-1 in a terminal window.

Example 1-1. Commands to Install KVM

```
sudo apt-get install qemu-kvm libvirt-bin ubuntu-vm-builder bridge-utils
sudo adduser your_user_name kvm
sudo adduser your_user_name libvirtd
```

You may have to reboot the system to complete the installation.

Hopefully everything went well and you now have a proper development environment. In the next chapter, you will familiarize yourself with the various parts of Android Studio IDE.

Chapter Summary

- You can get Android and Android Studio for macOS, Windows, and Linux. Each platform has an available precompiled binary available on the Android website.
- HAXM gives you a way to accelerate emulation on Android Virtual Devices. You will automatically get HAX when you're on macOS or Windows (with an Intel processor). If you're on Linux, you can use KVM instead of HAXM.

Chapter 2

Quick How-Tos

What this chapter covers:

- Creating a project
- Creating an activity
- Creating a class and an interface
- Generating code for method overrides
- Running a project

In this chapter, you'll take a look at some of the basic activities in Android project development, such as creating a project, activities, classes, interfaces, methods, and how to run an app. A lot of these elements can be managed manually (by hand) without much difficulty, but Android Studio has some nifty capabilities that can help you with these tasks. Android Studio can make these tasks quicker and more accurate. For example, typing override signatures by hand can be prone to error, especially if the method signature is long.

Creating a Project

To create an Android app, you need to create a project. A project is simply a folder that holds all the things you need to build an app: the Java program files, images, XML resources, and so on. You can create a project from the opening screen of Android Studio (when there aren't any projects open yet), as shown in Figure 2-1, or from the main menu bar (when another project is currently open), as shown in Figure 2-2.

If it's your first time using Android Studio and the very first time you are creating a project, you'll probably do it from the opening screen, as shown in Figure 2-1.

T. Hagos, *Android Studio IDE Quick Reference*,
https://doi.org/10.1007/978-1-4842-4953-6_2

Figure 2-1. Creating a project from the opening screen

If you have an existing project open in the IDE, and you'd like to start another one, you can do so from the main menu bar via File ➤ New ➤ Project, as shown in Figure 2-2.

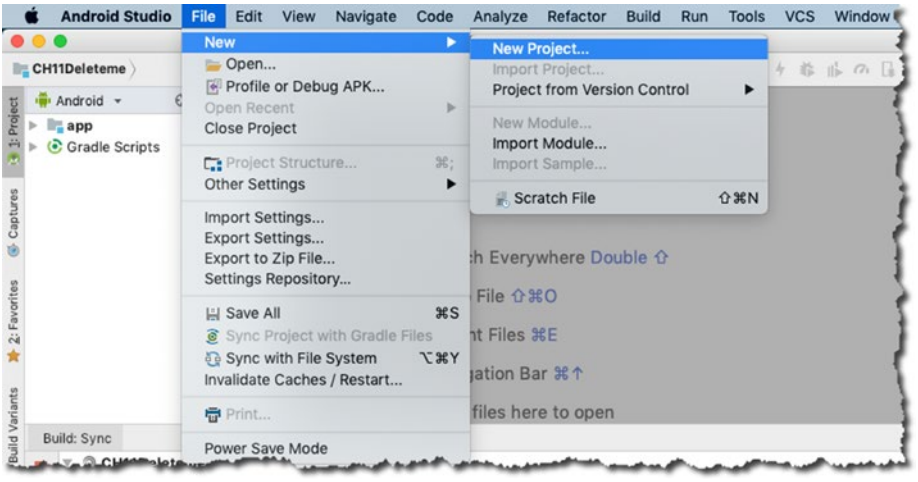

Figure 2-2. Creating a project from the main menu bar

No matter how you start the project, you'll see the next screen, shown in Figure 2-3, where you can fill in some details about the project.

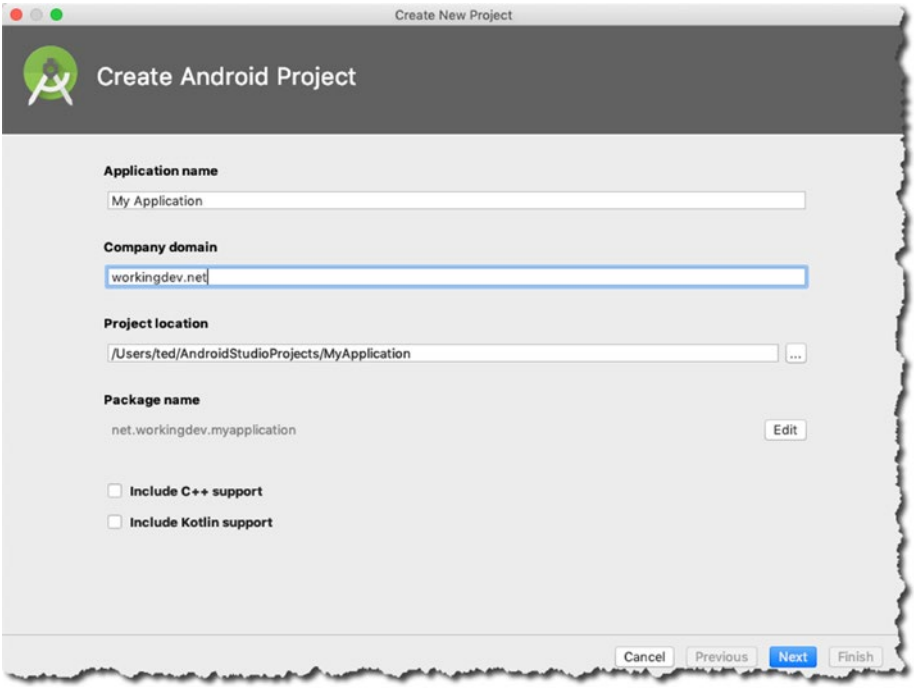

Figure 2-3. *Details of the new project*

Details like application name, company domain, and the actual location of the project will be on this screen. I left the C++ and Kotlin support boxes unchecked because you won't need them in this project. If you intend to mix Java code with C or C++ (NDK, or Native Development Kit), you need to check the C++ box; otherwise, leave it alone, like I did. If you would like to use Kotlin instead of Java for development, you need to check the Kotlin box; otherwise, leave it unchecked, as I did.

The next screen, shown in Figure 2-4, lets you configure the project some more.

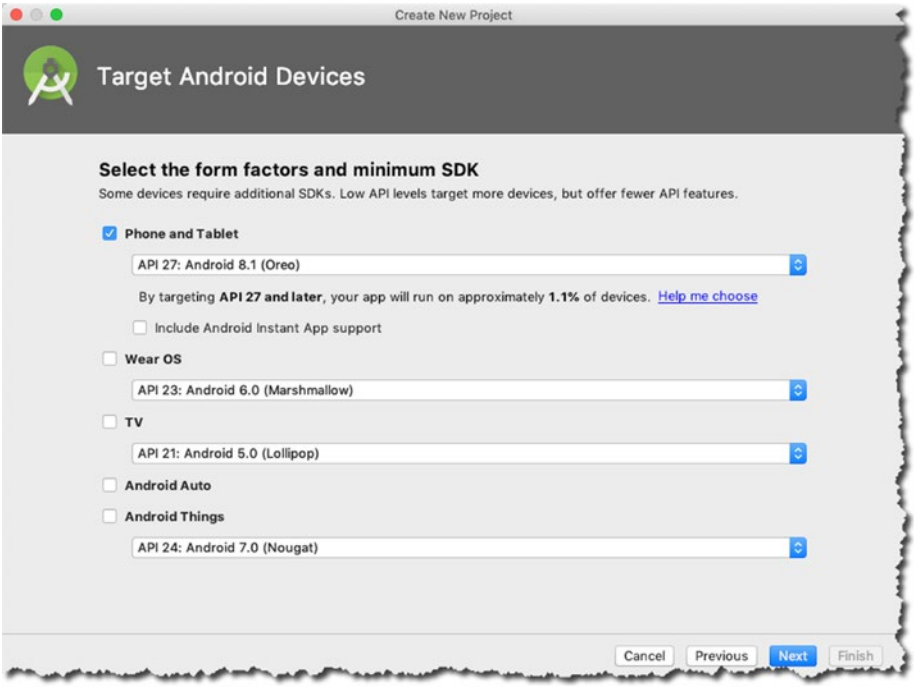

Figure 2-4. *The SDK and form factor*

On this screen, you get to configure the project for Android watch (wear OS), TV, or IoT (Internet of Things). I'm assuming that most beginner Android developers, like the probable readers of this book, will want to start with an app that runs on a phone or tablet, so only the "`Phone and Tablet`" box is checked.

In the next screen, shown in Figure 2-5, you decide if you want the wizard to generate an activity. There are a couple of choices for the activity, like basic, bottom navigation, and so on. Most Android applications will include at least one activity. You can choose the empty activity.

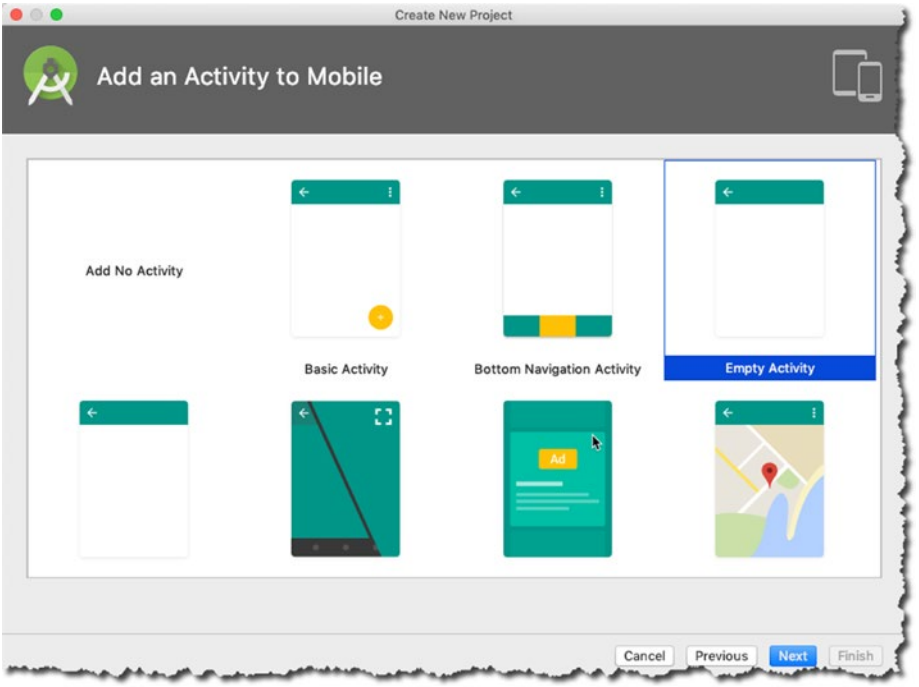

Figure 2-5. *The activity options*

Now you come to the final screen of the wizard, shown in Figure 2-6, where you get to decide on the name of the activity and the name of the XML layout for the activity. Check the AppCompat checkbox so you can use modern Android features even if the app will run on older Android versions.

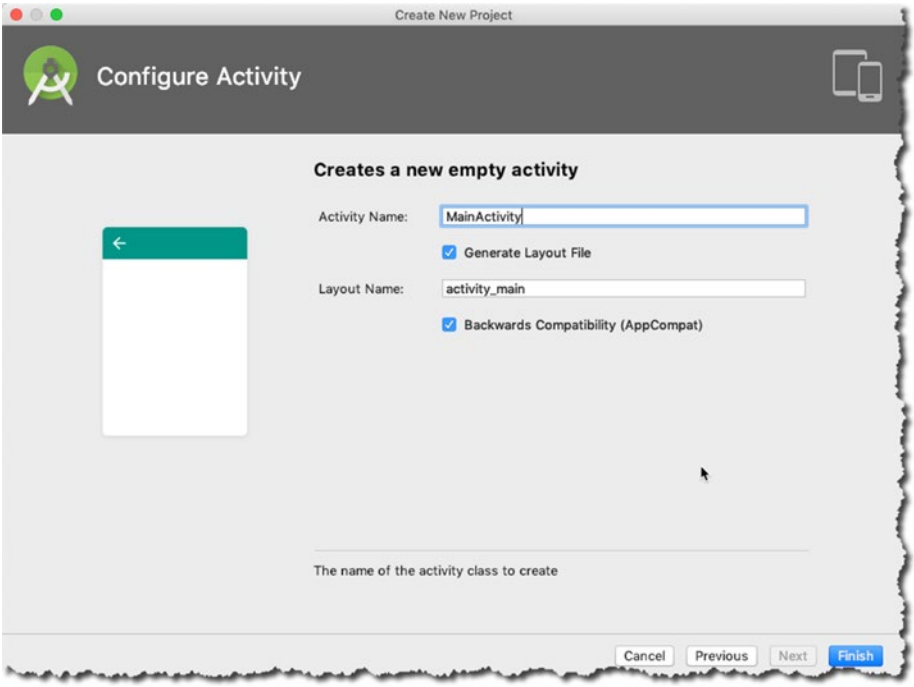

Figure 2-6. *The activity name*

Click the Finish button and the IDE will start to generate and scaffold an Android project with one empty activity. Figure 2-7 shows the `MyApplication` project open in the IDE.

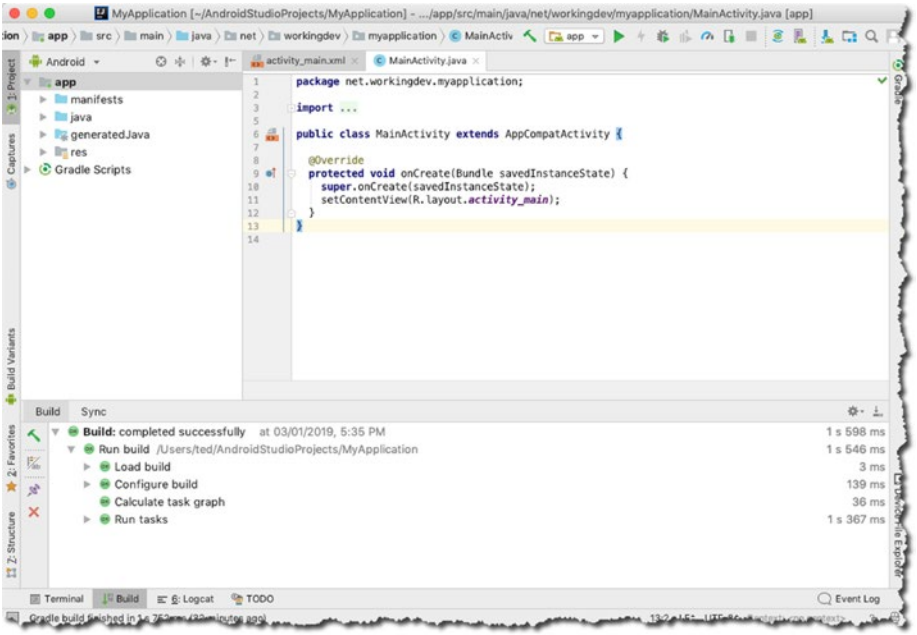

Figure 2-7. The MyApplication open in the IDE

A simple app that shows a screen to the user requires at least three things:

- An **Activity** class that acts as the main program file
- A **layout file** that contains all the UI definitions (in XML)
- A **manifest file** that ties all the project's contents together (also in XML)

The project creation wizard took care of all of these things.

Creating an Activity

An activity is largely responsible for what a user sees on the screen. The `Activity` class, together with an XML layout file, is what makes up the UI. Some apps only have one activity and some apps have more. If you need to add another activity to the project, you can do so on the main menu bar via File ➤ New ➤ Activity ➤ Empty Activity. Alternatively, you can also create an activity using the context menu shown in Figure 2-8.

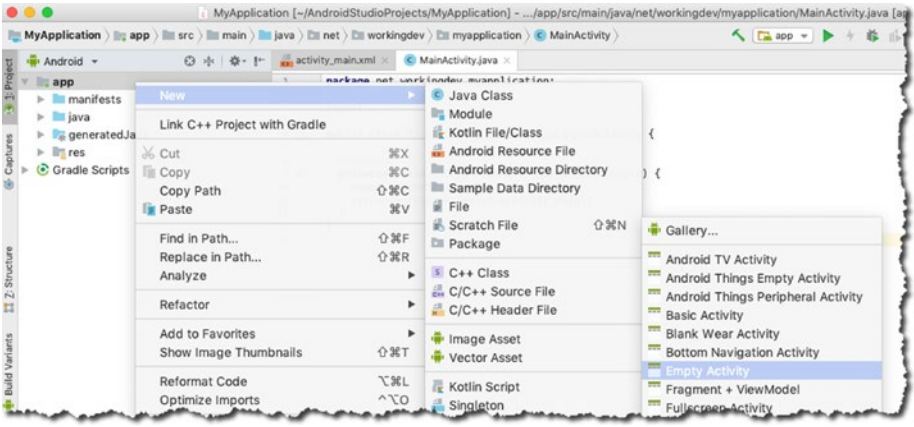

Figure 2-8. *Creating a new activity*

To use the context menu, right-click the app folder (in the project tool window), as shown in Figure 2-8, and then go to New ➤ Activity ➤ Empty Activity. Whichever method you use to add another activity, you'll get to a dialog screen (shown in Figure 2-9) where you can fill in some details about the new activity.

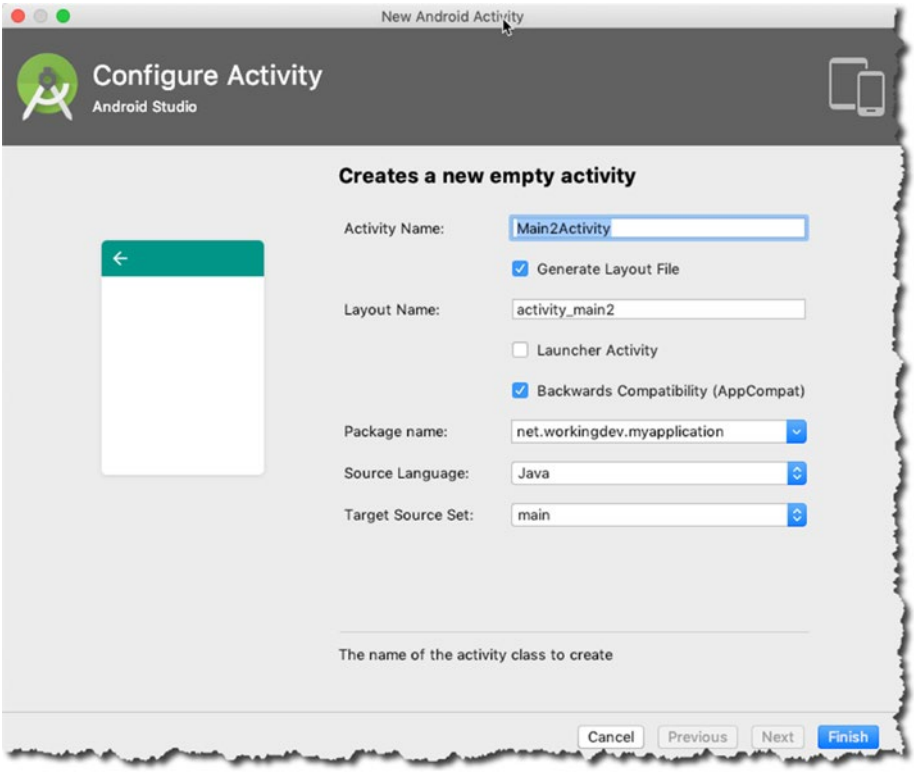

Figure 2-9. *New Android activity*

Most of these fields should be familiar to you because you filled them up during the project creation process. The only checkbox that may be a bit mysterious is the Launcher Activity checkbox. As you can see from Figure 2-9, it's unchecked, and you should leave it that way. If you check the Launcher Activity box, you're telling Android Studio to replace the launcher activity (the first activity you created during the project creation) with this new activity; that's not what you'd like to do, so leave the box unchecked.

> **Note** A launcher activity is the activity that is first shown to the user after an app has been launched. The configuration of the launcher activity is found in `ActivityMain.XML`.

Creating a Class

To create a new class, it's best to select the project's package from the Project Tool window, as shown in Figure 2-10. From there, you can either use the context menu (right-click) ➤ New ➤ Java class. Alternatively, you can also use the main menu bar via File ➤ New ➤ Java class.

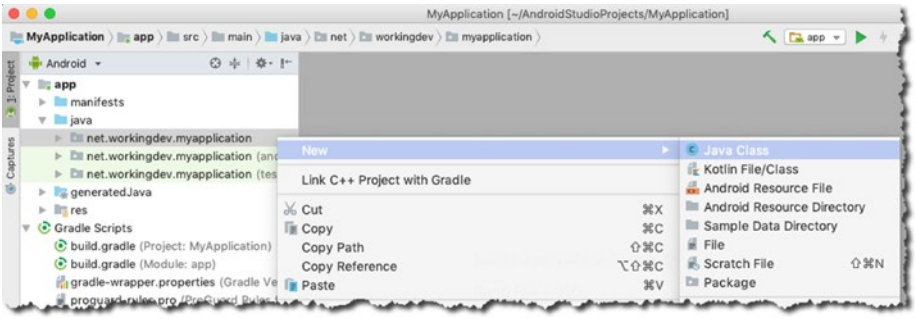

Figure 2-10. Using the context menu to create a new class

In the next screen, shown in Figure 2-11, you get to fill in some details for the new class.

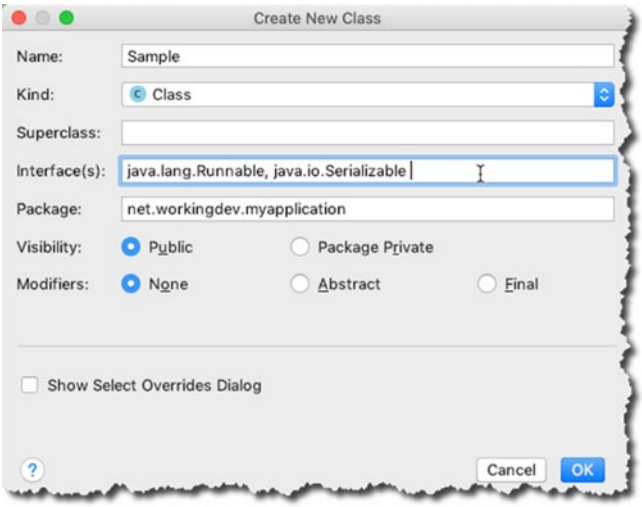

Figure 2-11. The Create New Class dialog

The Package section of the dialog window is already prepopulated with the correct package name because you selected the project's package (in the Project Tool window) before creating the class. If you hadn't selected the

project's package, the Package field would be empty. It's not a big deal to type the package name yourself, but it's a potential source of error if you mistype it.

Creating an Interface

To create an interface, follow the same steps you took when you created a class.

1. Using the context menu, right-click a project's package to select it.
2. Choose New ➤ Java class.

Figure 2-12 shows the Create New Class dialog. Click the Kind drop-down, as shown, and choose the Interface option.

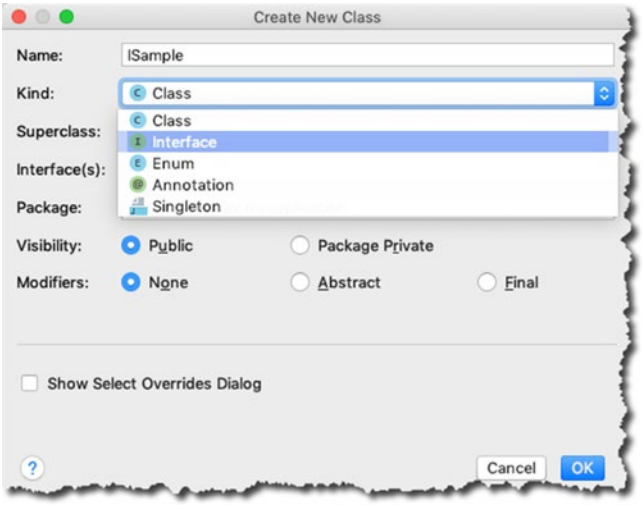

Figure 2-12. *Creating a new interface*

Override Methods

Typing all the override method signatures by hand shouldn't be a big deal, but Android Studio actually has some nifty features to help you out with those override signatures.

Figure 2-13 shows the `Sample.java` class you created earlier. The red squiggly lines mean the class has errors. Hovering your mouse on the red squiggly lines shows a balloon tip which contains some information about the error.

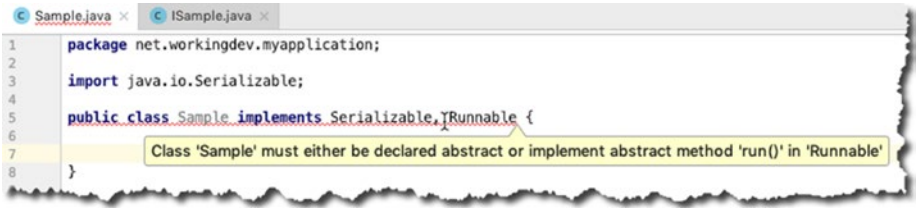

Figure 2-13. Sample.java with errors

To solve the error, you have to override the `run()` method of the `Runnable` interface. You can use the Quick Fix feature of Android Studio to solve this issue. Press `ALT-Enter` (Linux or Windows) or `Option + Enter` (macOS) while the keyboard cursor is somewhere along the red squiggly lines, as shown in Figure 2-14.

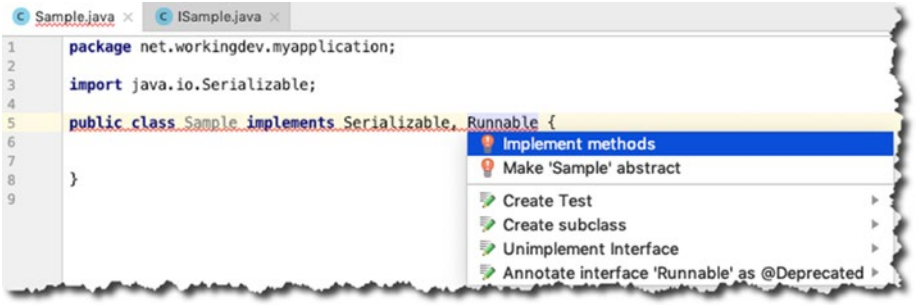

Figure 2-14. A quick fix

Choose the Implement methods option. The Select Methods to Implement dialog, shown in Figure 2-15, appears next.

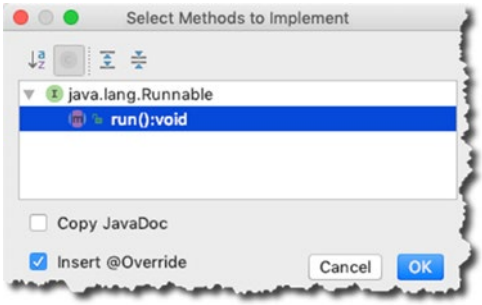

Figure 2-15. Selecting methods to implement

Select `run():void` and then click `OK`. Android Studio will generate the method signature for you.

Android Studio's Quick Fix feature isn't only for overriding methods. You can use it whenever you see a red squiggly line or any other indication in the IDE that shows an error. The IDE is robust enough to figure out how to help you in most situations.

Another way to generate code for overridden methods is to use Android Studio's Generate feature. Figure 2-16 shows how to go about this.

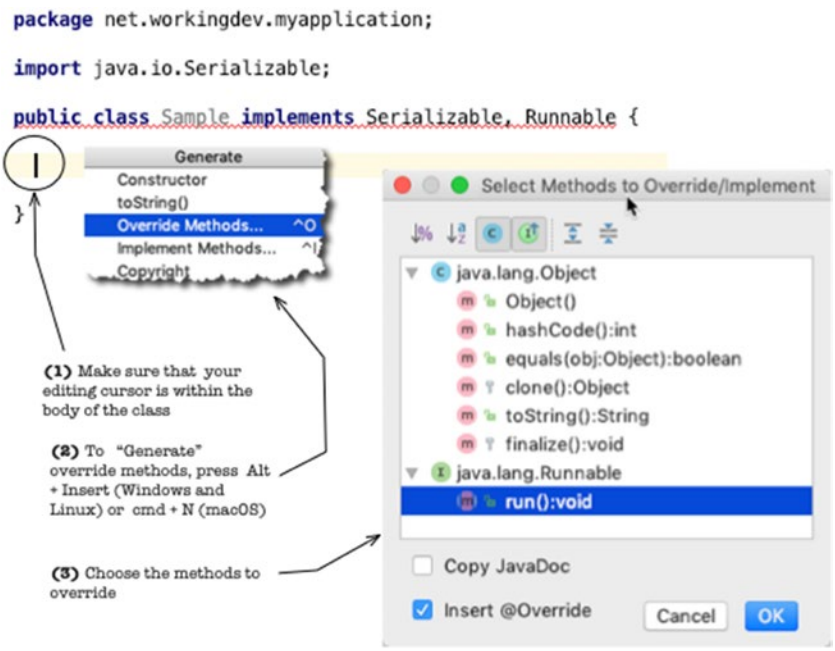

Figure 2-16. *Overriding methods using the Generate feature*

The Generate feature is versatile. You can use it to generate quite a few things like getters/setters, constructors, new files, and new classes but for your purpose now, you'll use it to generate methods to override. Figure 2-16 shows the general flow of the code generation process, but let's recap the steps anyway.

1. Make sure that your editing cursor is inside the class. The Generate feature is context sensitive, so if the editing cursor is outside a class body, you won't get the options shown in Figure 2-16.
2. Press `Alt + Insert` (if you're on Windows or Linux) or `Cmd + N` (if you're on macOS).

3. Choose the Override Methods option.
4. Choose the method to override.

Running a Project

Android Studio has a couple of options for building and running a project; see Table 2-1.

Table 2-1. Keyboard Shortcuts for Building and Running a Project

	Windows and Linux	macOS	Description
Build	Ctrl + F9	Cmd + F9	Runs the build process of Android Studio and produces an APK (Android Package)
Build and Run	Shift + F10	Ctrl + R	Same as "Build" but it also pushes the APK into a connected device or a running emulator (AVD)
Apply changes (with Instant Run)	Ctrl + F10	Ctrl + Cmd + R	Allows you to push code changes to a connected device or emulator without building a new APK. It's faster than "Build and Run," so use it whenever you possible.

Chapter Summary

- Android Studio has more than one way to accomplish a task. You can use both the main menu bar and the context menu to create quite a few things like a new activity, class, or interface.
- Keyboard shortcuts are much faster than using the main menu bar or the context menu, so it's worth the time to memorize some of these shortcuts.
- Android Studio's Generate feature can do more than just generate override methods. You'll explore its other capabilities in later chapters.
- The Quick Fix in Android Studio is a versatile tool. Just click a red squiggly line, and press `Alt + Enter` (Windows or Linux) or `Option + Enter` (macOS).

Chapter 3

The IDE

What this chapter covers:

- Parts of the IDE
- Main editor
- Project tool window
- SDK Manager
- Code styles

In the previous chapter, you learned how to create a project and open it in the IDE. In this chapter, you'll take a closer look at its parts.

If you haven't launched Android Studio yet, now is a good time to do so. After opening Android Studio, you'll see the Welcome Screen, as shown in Figure 3-1.

T. Hagos, *Android Studio IDE Quick Reference*,
https://doi.org/10.1007/978-1-4842-4953-6_3

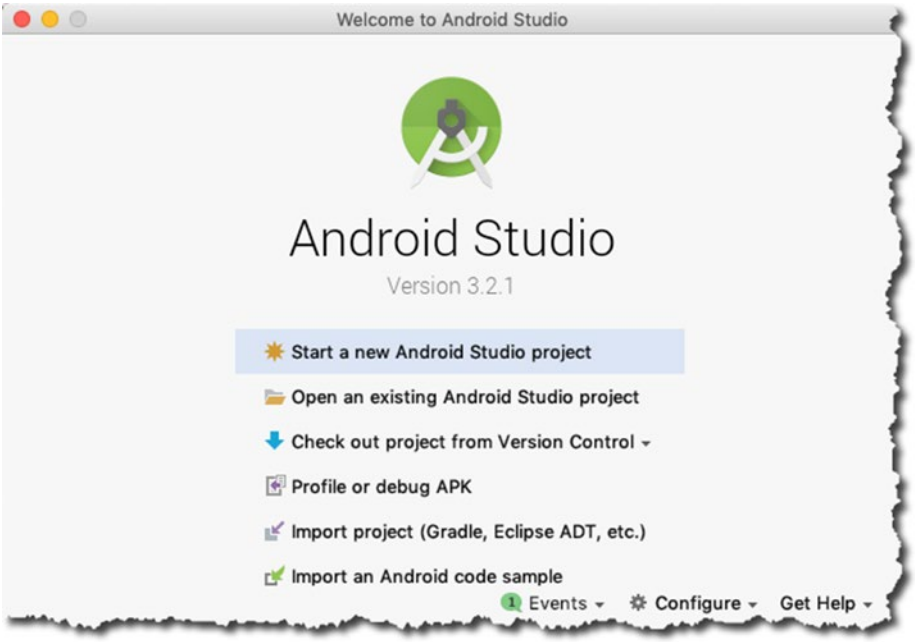

Figure 3-1. Welcome to Android Studio

The IDE

From the opening screen, assuming you either created a new project or opened an existing one, Figure 3-2 shows the various parts of Android Studio while a project is open in it.

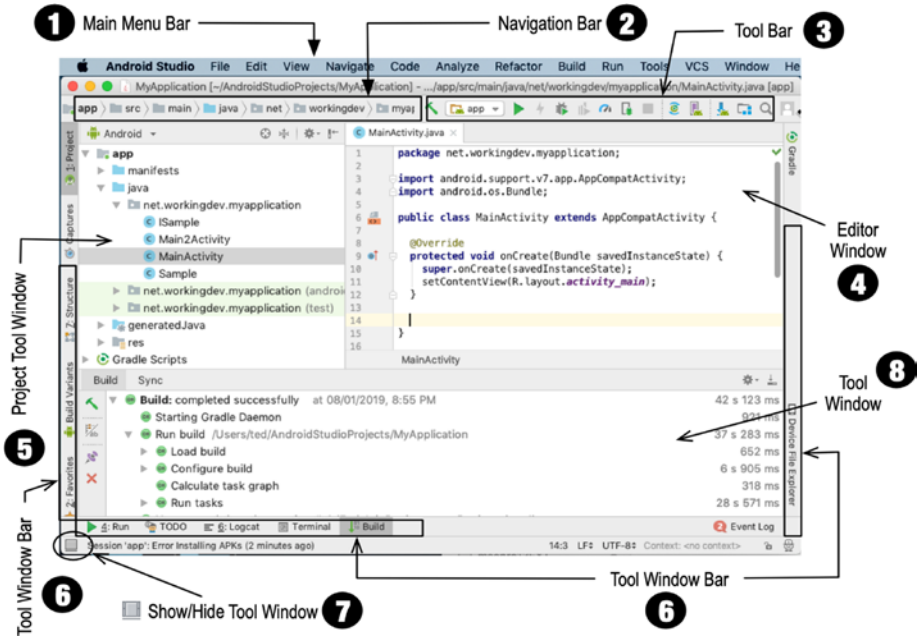

Figure 3-2. Main parts of Android Studio

❶ **Main menu bar**: You can navigate Android Studio in various ways. Often, there's more than one way to do a task, but the primary navigation is done in the main menu bar. If you're in Linux or Windows, the main menu bar sits directly at the top of the IDE; if you're in macOS, the main menu bar is disconnected from the IDE (which is how all macOS software works).

❷ **Navigation bar**: This bar lets you navigate the project files. It's a horizontally arranged collection of chevrons that resembles the breadcrumb navigation you find on some websites. You can open your project files either through the navigation bar or the Project tool window.

❸ **Tool bar**: It lets you do a wide range of actions (e.g., save files, run the app, open the AVD Manager, open the SDK Manager, and undo and redo actions).

❹ **Main editor window**: This is the most prominent window and has the most screen real estate. The editor window is where you can create and modify project files. It changes its appearance depending on what you are editing. If you're working on a program source file, this window will show just the source files. When you are editing layout files, you may see either the raw XML file or a visual rendering of the layout.

❺ **Project tool window**: This window shows the contents of the project folders. You'll be able to see and launch all your project assets (source code, XML files, graphics, etc.) from here.

❻ **Tool window bar**: The tool window bar runs along the perimeter of the IDE window. It contains the individual buttons you need to activate specific tool windows (e.g., TODO, Logcat, Project window, Connected Devices, etc.).

❼ **Show/hide tool window**: It shows (or hides) the tool window bar. It's a toggle.

❽ **Tool window**: You will find tool windows on the sides and bottom of the Android Studio workspace. They're secondary windows that let you look at the project from different perspectives. They also let you access the typical tools you need for development tasks (e.g., debugging, integration with version control, looking at the build logs, inspecting Logcat dumps, looking at TODO items, etc.). Here are couple of things you can do with the tool windows:

- You can expand or collapse them by clicking the tool's name in the tool window bar. You can also drag, pin, unpin, attach, and detach the tool windows.
- You can rearrange the tool windows, but if you feel you need to restore the tool window to the default layout, you can do so from the main menu bar; click Window ➤ Restore Default Layout. Also, if you want to customize the default layout, you can rearrange the windows to your liking from the main menu bar by clicking Window ➤ Store Current Layout as Default.

Main Editor

Like in most IDEs, the main editor window lets you modify and work with source files. What makes it stand out is how well it understands Android development assets. Android Studio lets you work with a variety of file types, but you'll probably spend most of your time editing these types of files:

- Java source files
- XML files
- UI layout files

When you're working with Java source files, you get all the code hinting and completions that you've come to expect from a modern editor. What's more, it gives you plenty of early warning when something is wrong with your code. Figure 3-3 shows a Java class file open in the main editor. The class file is an activity and it's missing a semicolon in one of its statements. Android Studio peppers the IDE with (red) squiggly lines, which indicate that the class won't compile.

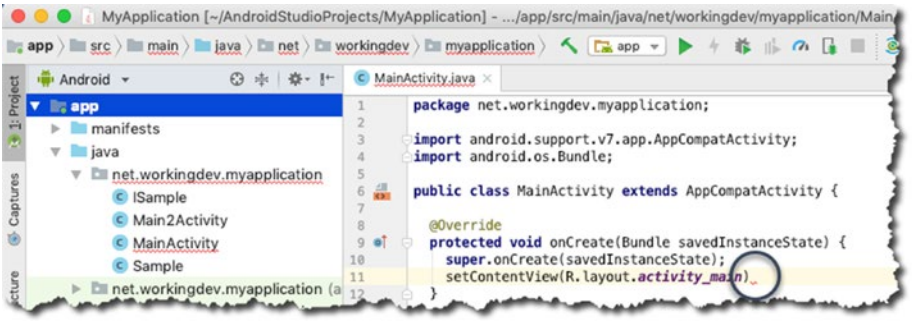

Figure 3-3. Main editor showing error indicators

Android Studio places the squiggly lines very near the offending code. As you can see in Figure 3-3, the squiggly lines are placed right at the point where the semicolon is expected.

Editing Layout Files

The screens that your user sees are made up of activity source files and layout files. The layout files are written in XML. Android Studio undoubtedly can edit XML files, but what sets it apart is how intuitively it can render the XML files in a WYSIWYG mode (what you see is what you get). Figure 3-4 shows the two ways you can work with layout files.

Figure 3-4. Design mode and Text mode editing of layout files

Figure 3-5 shows the various parts of Android Studio that are relevant when working on a layout file during Design mode.

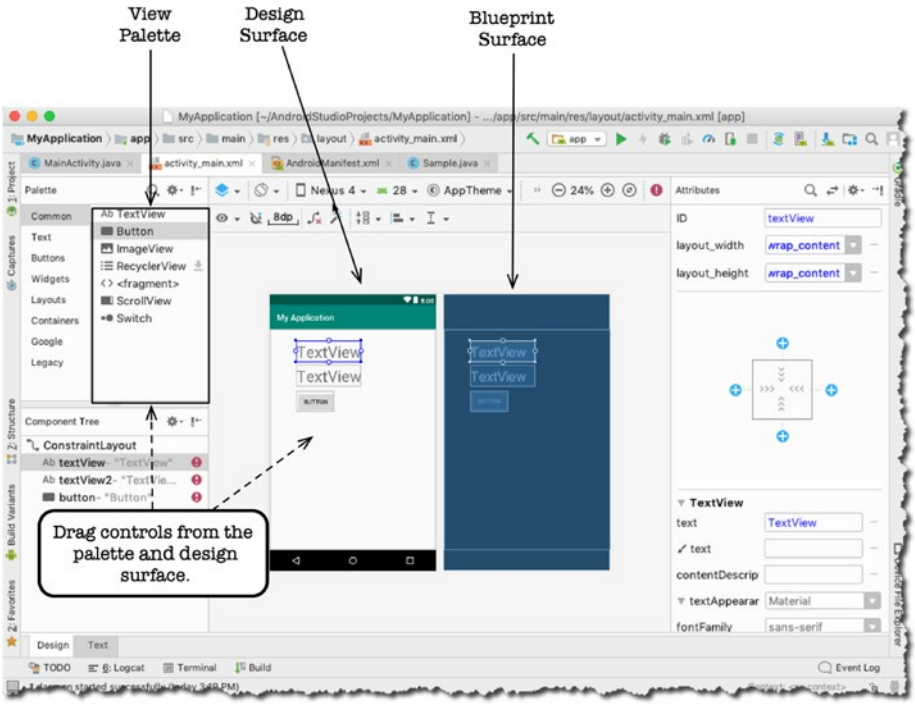

Figure 3-5. Layout design tools of Android Studio

- **View palette**: The View palette contains the views (widgets) that you can drag and drop on either the Design surface or Blueprint surface.
- **Design surface**: It acts like a real-world preview of your screen.
- **Blueprint surface**: Similar to the Design surface, but it only contains the outlines of the UI elements.
- **Attributes window**: You can change the properties of the UI element (view) in here. When you make a change on the properties of a view using the Attributes window, that change will be automatically reflected in the layout's XML file. Similarly, when you make a change in the XML file, it will automatically be reflected in the Attributes window.

Inserting TODO Items

You don't have to create a separate file to keep track of your TODO list for your app. When you create a comment with TODO text like

```
// TODO This is a sample todo
```

Android Studio will keep track of all the TODO comments in all of your source files. See Figure 3-6.

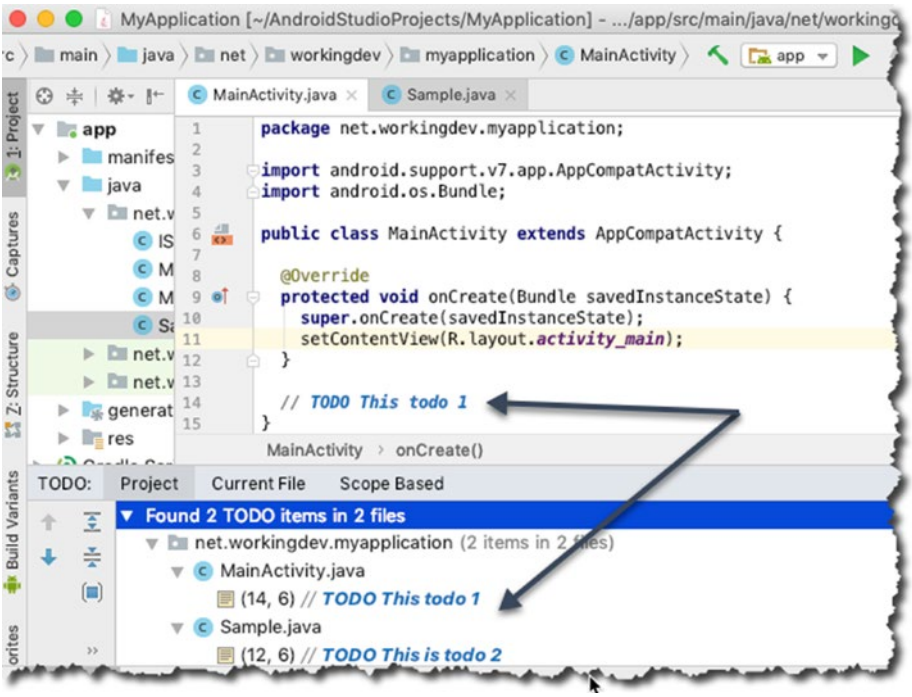

Figure 3-6. TODO items

To view all your TODO items, click the TODO tab in the tool window bar.

How to Get More Screen Space for Code

You can get more screen real estate by closing all tool windows. Figure 3-7 shows a Java source file open in the main editor window while all the tool windows are closed. You can collapse any tool window by simply clicking its name, so to collapse the Project tool window, click Project.

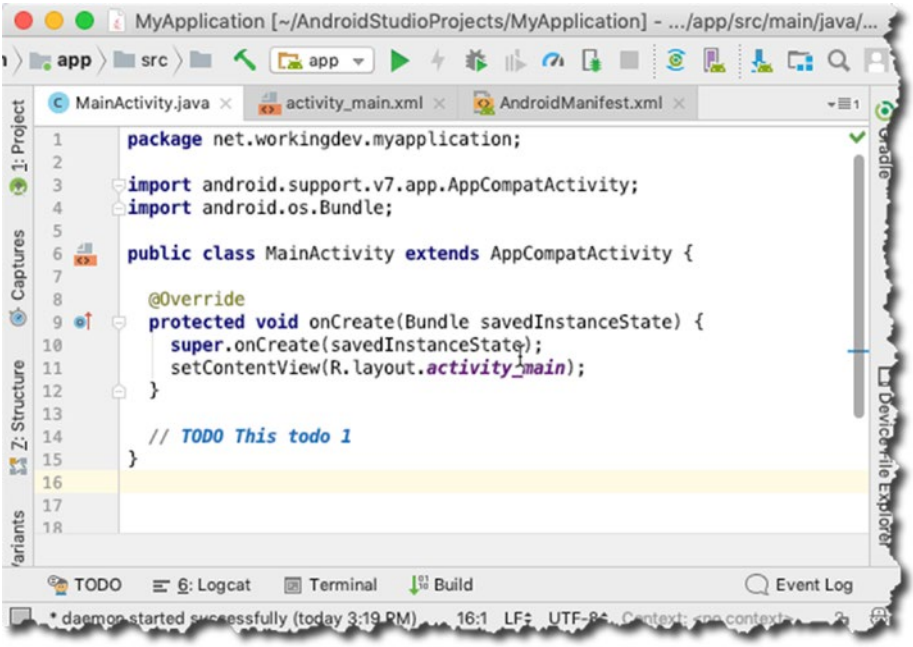

Figure 3-7. Main editor, with all tool windows closed

You can get even more screen real estate by hiding all the tool window bars, as shown in Figure 3-8.

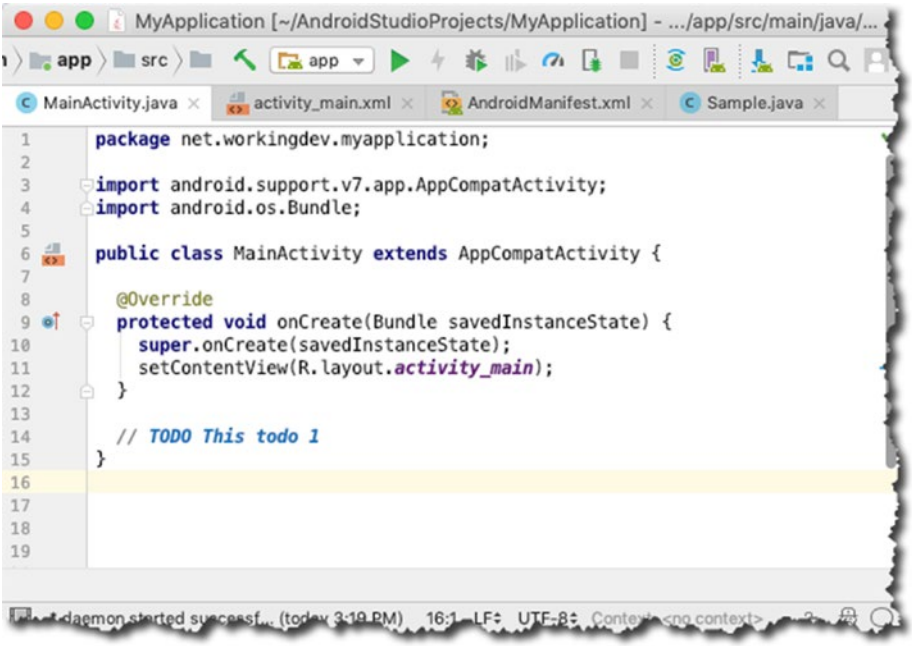

Figure 3-8. Main editor, with all tool windows closed and toolbars hidden

You can get even more screen space by entering distraction-free mode, as shown in Figure 3-9. You can enter distraction-free mode from the main menu bar via View ➤ Enter Distraction Free Mode. To exit the mode, click View ➤ Exit Distraction Free Mode.

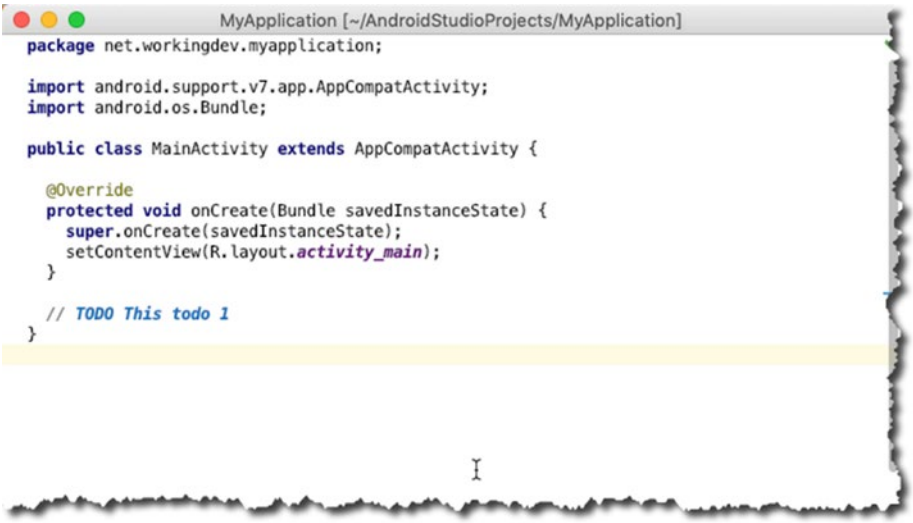

Figure 3-9. Distraction-free mode

You may also try two other modes that can increase screen real estate. They're also found in the View menu from the main menu bar.

- Presentation mode
- Full screen

Project Tool Window

You can get to your project's files and assets via the Project tool window, shown in Figure 3-10. It has a tree-like structure and the sections are collapsible. You can launch any file from this window. If you want to open a file, you simply double-click the file from this window.

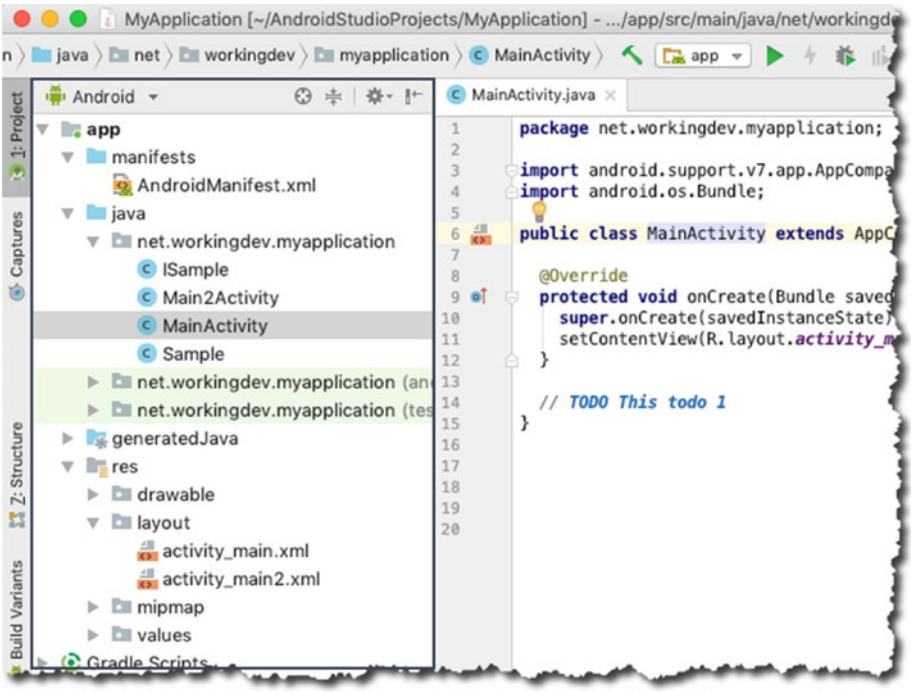

Figure 3-10. *Project tool window*

By default, Android Studio displays the project files in Android view, as shown in Figure 3-10. The Android view is organized by modules to provide quick access to the project's most relevant files. You change how you view the project files by clicking the down arrow on top of the Project window, as shown in Figure 3-11.

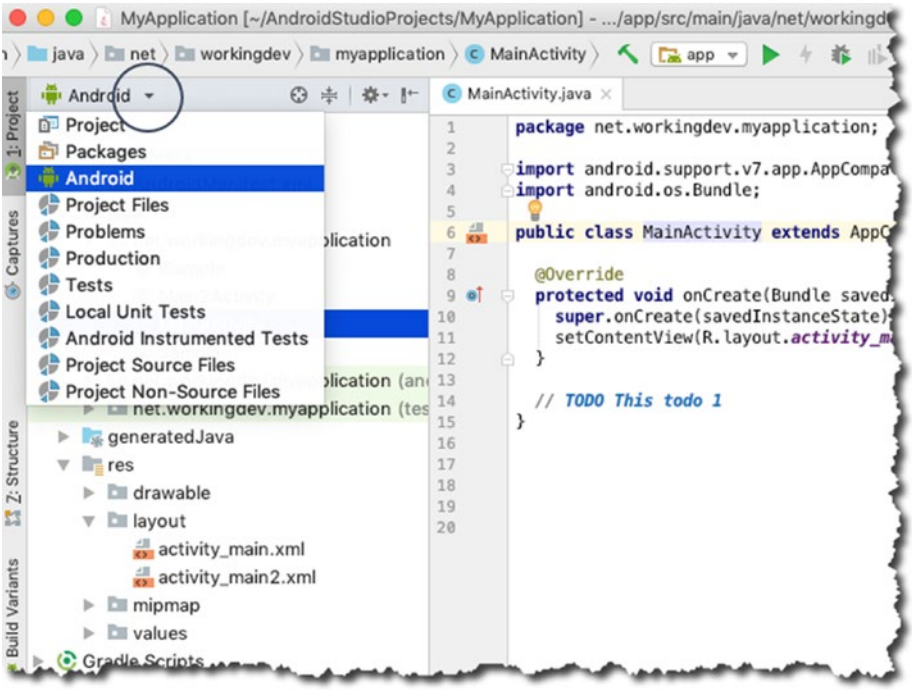

Figure 3-11. How to change views in the Project tool window

Preferences/Settings

If you want to customize the behavior or look of Android Studio, you can do so in its Settings or Preferences window; it's called Settings if you're in Windows or Linux and Preferences if you're in macOS. See Figure 3-12.

For Windows and Linux users, you can get to the Settings window in one of two ways:

- From the main menu bar, click File ➤ Settings.
- Use the keyboard shortcut Ctrl + Alt + S.

For macOS users, you can do it this way:

- From the main menu bar, click Android Studio ➤ Preferences.
- Use the keyboard shortcut Command + ,.

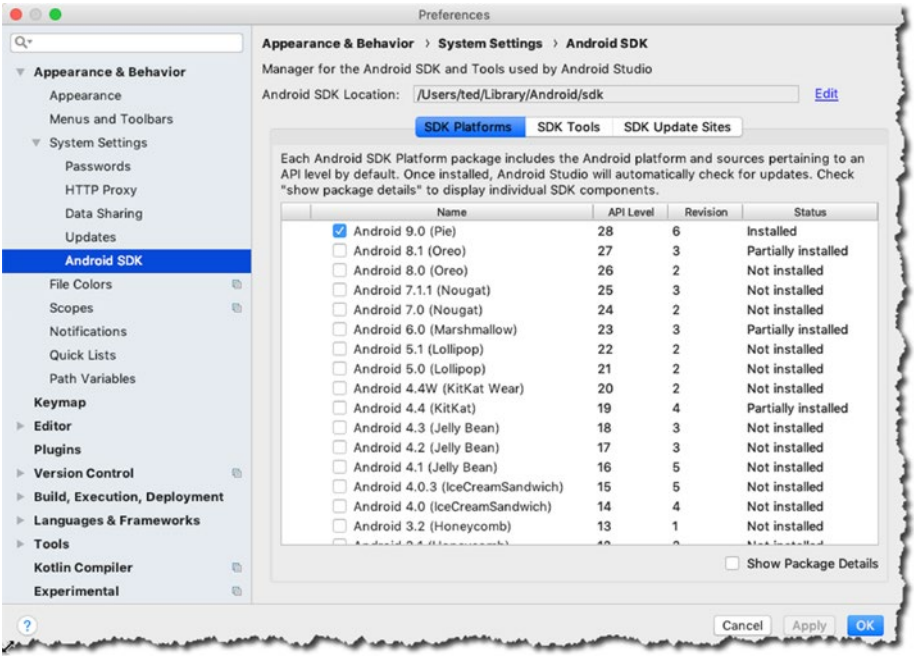

Figure 3-12. Settings/Preferences window

You can access a variety of settings in this window, such as how Android Studio looks, whether to use spaces or tabs in the editor, how many spaces to use for tabs, which version control to use, what API to download, and what system images to use for AVD.

The SDK Manager

The SDK Manager is used to manage what API levels (Android versions) and other Android tools you want downloaded into your local system. You can get to the SDK Manager via the Settings or Preferences window, and then you click the Android SDK item (found on the left side), as shown in Figure 3-13.

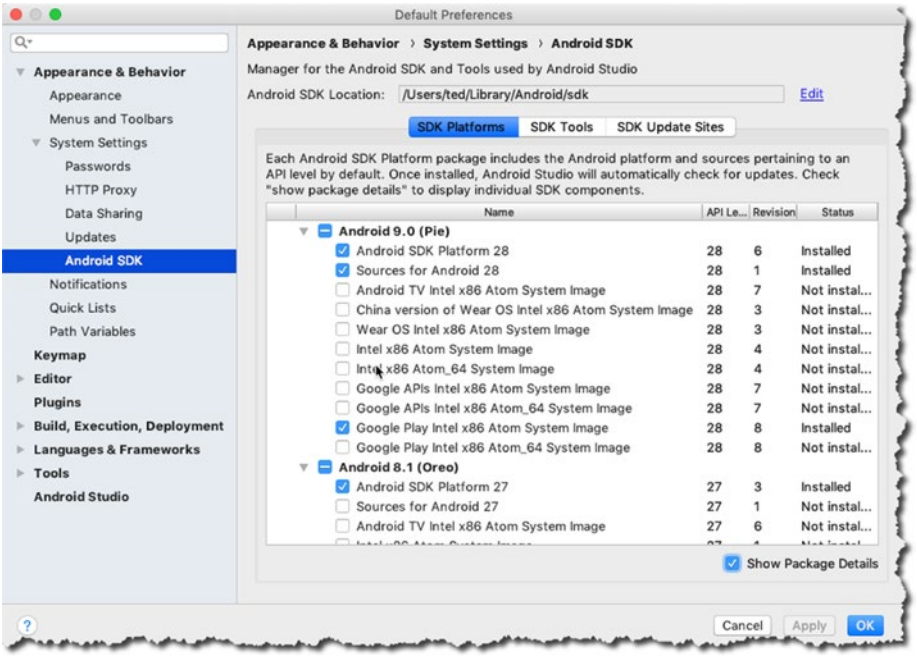

Figure 3-13. SDK Manager

The SDK platforms (where you can select the Android versions to download), SDK tools, and SDK update sites can be accessed by clicking their tabs, which are found in the upper middle portion of the window.

You can also launch the SDK Manager from the tool bars, as shown in Figure 3-14.

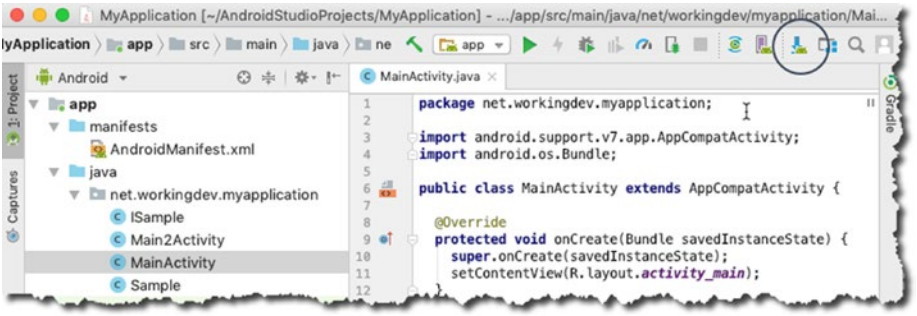

Figure 3-14. Launch the SDK Manager from the tool bar

Code Styles

You can change the code style scheme in Settings/Preferences ➤ Editor ➤ Code Style, as shown in Figure 3-15.

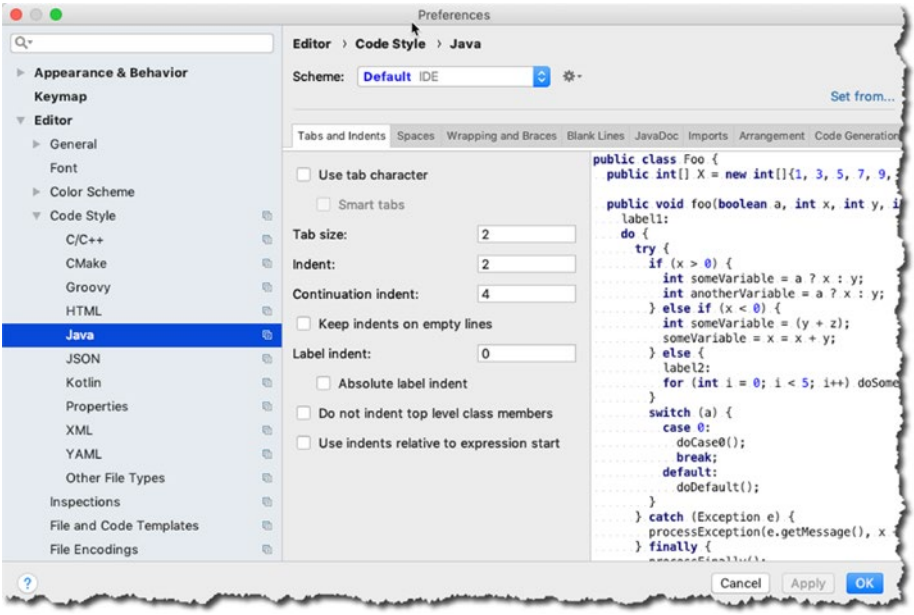

Figure 3-15. Code styles

In this window, you can customize tab sizes, indents, how much white space is the default for your code, how getters and setters are generated, code generation, how imports are done, and more.

Chapter Summary

- You can see more of your code by increasing the screen real estate for the main editor. You can do this by
  - Collapsing all the tool windows
  - Hiding the tool window bars
  - Entering distraction-free mode
  - Going to full-screen mode

- You can change the coding scheme for Java (or any language you like) in Settings/Preferences ➤ Editor ➤ Code Style.
- You can change how you view the project files from switching the view in the Project tool window.
- Adding a TODO item is easy in Android Studio. Just add a single line comment followed by some TODO text, like this:

```
// TODO This is my todo list
```

Chapter 4

Debugging

What this chapter covers:

- Common errors you'll encounter
- Logging debug statements
- Using the debugger

Dealing with errors is a big part of your life as a developer. This chapter will discuss the kinds of errors you've faced and will still face in the foreseeable future. You'll see how you can use Android Studio to ease the difficulty of dealing with these errors.

Types of Errors

The three most common errors you'll face in programming are

- Syntax errors
- Runtime errors
- Logic errors

Syntax Errors

Syntax errors are exactly what you think they are: errors in the syntax. They happen because you wrote something in the code that's not allowed by the set of rules of the Java compiler. In other words, the compiler doesn't understand it. The error can be as simple as forgetting to close a parenthesis or a missing

T. Hagos, *Android Studio IDE Quick Reference*,
https://doi.org/10.1007/978-1-4842-4953-6_4

pair of curly braces. It can be as complex as passing the wrong type of argument to a function or using a parameterized class when using generics.

You can catch syntax errors in Android Studio with ease. The clue is the red squiggly lines in the main editor, as in Figure 4-1.

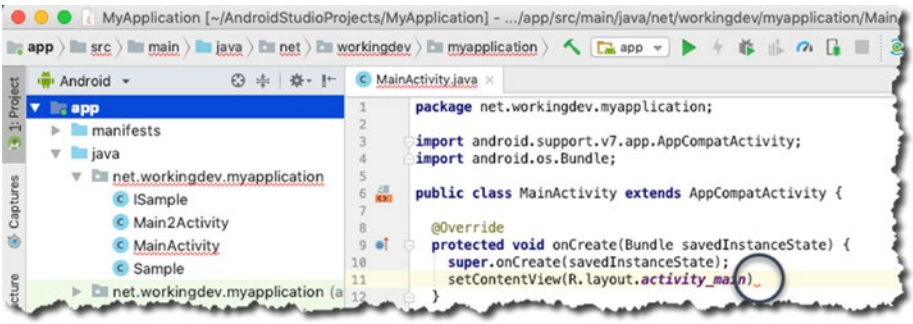

Figure 4-1. *Main editor showing error indicators*

They mean something is syntactically wrong with your code. The IDE places the red squiggly lines very near the offending code. If you hover your mouse on a red squiggly line, most of the time Android Studio can tell you, with a high degree of accuracy, what's wrong with the code. What's more, you can quickly fix these kinds of errors using a technique that's aptly named "quick fix".

To do a quick fix, bring the cursor anywhere within the red squiggly lines and then press Alt + Enter (for Windows or Linux) or Option + Enter (for macOS). The IDE takes care of the rest. If there's more than one way to fix the error, the IDE will show you some options.

Runtime Errors

Runtime errors happen when your code hits a situation it doesn't expect. As the name implies, this error happens only when your program is running. It's not something you'll see during compilation.

Java has two types of exceptions, *checked* and *unchecked*. Android Studio gives you lots of assistance with checked exceptions. Figure 4-2 shows what happens in the main editor when you try to call to a method that throws a checked exception; with unchecked exceptions, you're still on your own.

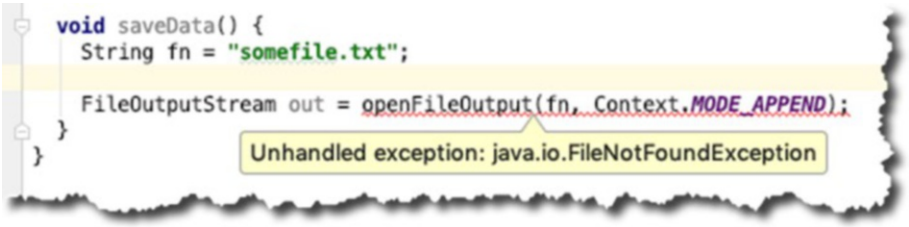

Figure 4-2. The IDE reminder that you need to handle the exception

There are two ways to resolve the error shown in Figure 4-2: you can enclose the `openFileOutput()` method call inside a *try-catch* structure or you add an exception to the method signature, as shown in Figure 4-3.

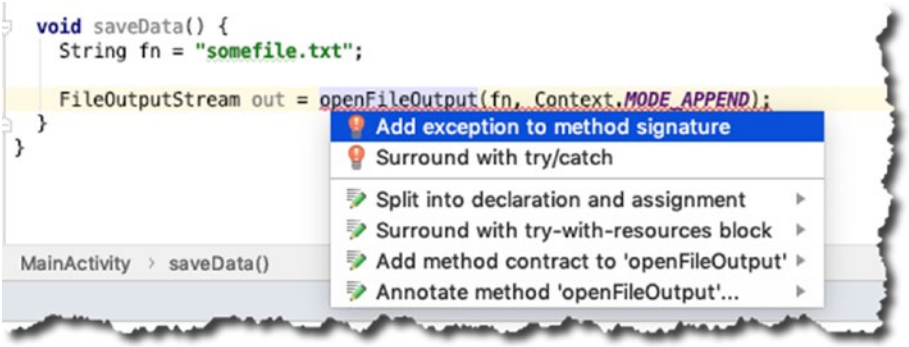

Figure 4-3. Quick fixes

Listing 4-1 shows how to handle the `FileNotFoundException` by adding a throws clause to the method signature.

Listing 4-1. FileNotFoundException Thrown in saveData()

```
import java.io.FileNotFoundException;
...
void saveData() throws FileNotFoundException {
  String fn = "somefile.txt";

  FileOutputStream out = openFileOutput(fn, Context.MODE_APPEND);
}
```

Listing 4-2 shows the code for handling the same exception using a *try-catch* block.

Listing 4-2. Handling an Exception Using try-catch

```
void saveData() {
  String fn = "somefile.txt";

  try {
    FileOutputStream out = openFileOutput(fn, Context.MODE_APPEND);
  } catch (FileNotFoundException e) {
    e.printStackTrace();
  }
  finally {
    ...
  }
}
```

I use a *try-catch* when I want to handle the exception locally, meaning in the same block of code where the exception might be thrown. Most of the time, the only things to do are to (1) log the error and (2) try to recover from the error, if at all possible, and let the user try again.

Using a *throws* clause, on the other hand, means you don't want to handle the error on the local block; you'd like the calling method to take care of the error instead. If the calling method also uses a throws clause in its signature, then the error handling is passed along up the call stack.

Logic Errors

Logic errors are the hardest to find. As the name suggests, it's an error in your logic. When your code is not doing what you thought it should be doing, that's logic error. There are many ways to cope with a logic error, but in this section, you'll take a look at two: printing debugging statements in certain places of your code and walking through your code using the debugger.

As you inspect your code, you will recognize certain areas where you're pretty sure about what's going on, and then there are areas where you are not so sure. You can place debugging statements in the latter areas. It's like leaving breadcrumbs to follow. There are a couple of ways to print debugging statements. You can either use `println`, `Log`, or the `Logger` class.

When you set Logcat's mode to verbose, info, or debug, you will see all the messages that Android's runtime generates. If you want to be able to filter out messages such as warn or error, you need to use either the `Log` or `Logger` class.

The Log class has five static methods:

```
Log.v(tag, message) // verbose
Log.d(tag, message) // debug
Log.i(tag, message) // info
Log.w(tag, message) // warning
Log.e(tag, message) // error
```

In each case, *tag* is a String literal or variable, typically the name of the class where Log is called. The *message* is also String literal or variable which contains what you actually want to see in the log. See Listing 4-3.

Listing 4-3. Typical Use of the Log Class

```
package net.workingdev.myapplication;

import android.content.Context;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.util.Log;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;

public class MainActivity extends AppCompatActivity {

  String TAG = this.getClass().getName();

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    ...
  }

  void saveData() {
    String fn = "somefile.txt";

    try {
      FileOutputStream out = openFileOutput(fn, Context.MODE_APPEND);
    } catch (FileNotFoundException e) {
      e.printStackTrace();
      Log.e(TAG, "error");
    }
    finally {
      Log.v(TAG, "My Message");
    }
  }
}
```

When the app is running, you can see the Log messages in the **Logcat** tool window, as shown in Figure 4-4. You can launch the Logcat window either by clicking its tab in the menu strip at the bottom of the IDE or from the main menu bar via View ➤ Tool Windows ➤ Logcat.

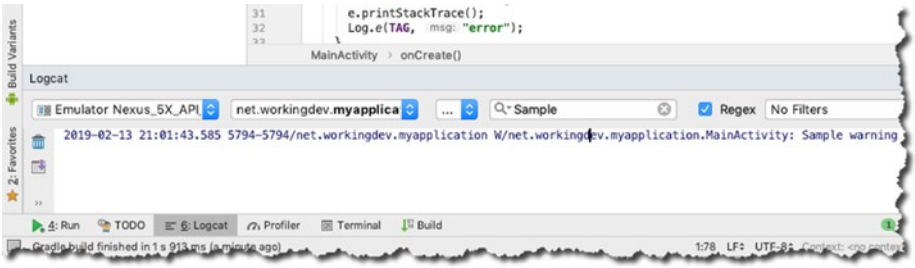

Figure 4-4. Logcat tool window

Debugger

Android Studio includes an interactive debugger which allows you to walk and step through the code while it's running. With the interactive debugger, you can inspect snapshots of the application—values of variables, running threads, and so on—at specific locations in the code and at specific points in time. These specific locations in the code are called *breakpoints*; you get to choose these breakpoints.

To set a breakpoint, choose a line that has an executable statement and then click its line number in the gutter. When you set a breakpoint, there will be a pink circle icon in the gutter and the whole line will be lit in pink, as shown in Figure 4-5.

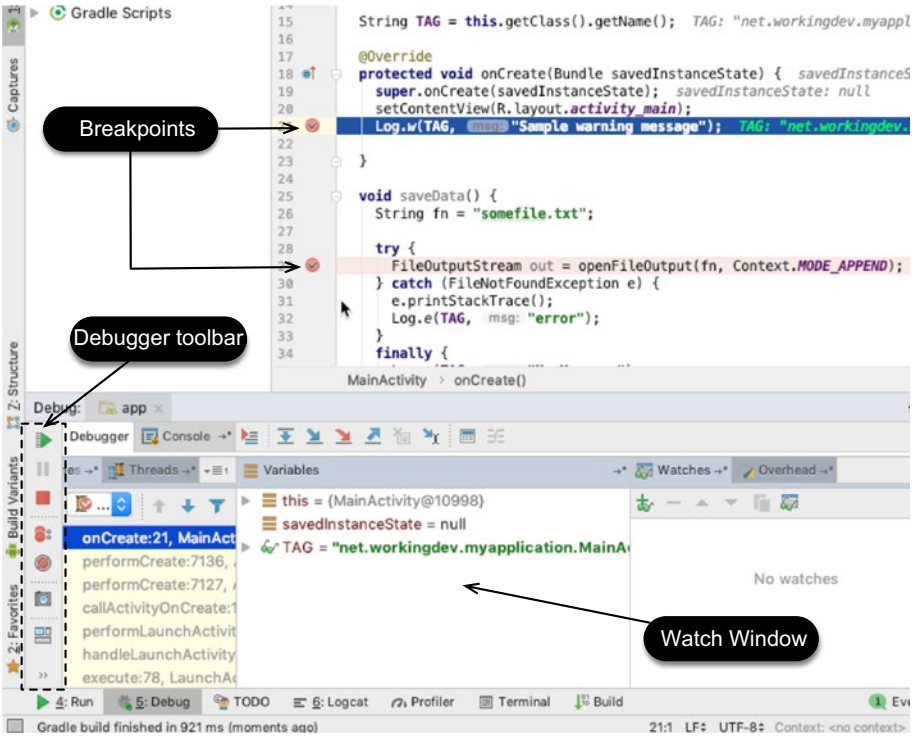

Figure 4-5. *Debugger window*

After the breakpoints are set, you have to run the app in debug mode. Stop the app if it is currently running and from the main menu bar, click Run ➤ Debug App.

> **Note** Running the app in debug mode isn't the only way to debug the app. You can also attach the debugger process in a currently running application. There are situations where this second technique is useful. For example, when the bug you are trying to solve occurs in very specific conditions, you may want to run the app for a while, and when you think you are close to the point of error, you can then attach the debugger.

Use the application as usual. When the execution comes to a line where you set a breakpoint, the line turns from pink to blue. This is how you know the code execution is at your breakpoint. At this point, the debugger window opens, the execution stops, and Android Studio gets into interactive debugging mode. While you are here, the state of the application is

displayed in the Debug tool window. During this time, you can inspect values of variables and even see the threads running in the app.

You can also add variables or expression in the Watch window by clicking the plus sign with the spectacles icon. You'll get a text field where you can enter any valid expression. When you press Enter, Android Studio will evaluate the expression and show you the result. To remove a watch expression, select the expression and click the minus sign icon in the Watch window.

Single Stepping

Like most debuggers, Android Studio lets you step line by line through your program. When the debugger stops at a breakpoint, you have a couple of tools at your disposal. You'd typically want to know how to do the following.

- **Resume**: Resumes execution until you get to the next breakpoint. If there aren't any more breakpoints, then the program runs like it would in normal execution.
- **Step into**: If the next line has a method call, this will jump to the method and pause it at the first line.
- **Step over**: Executes whatever happens on the next line and then jumps to the next line.
- **Step out**: Executes the remainder of the current method and then pauses at the next statement after the method. It essentially gets out of the method.

You can get to these actions from the main menu bar under the Run menu. You can also get to them from the Debugger tool bar (shown in Figure 4-6).

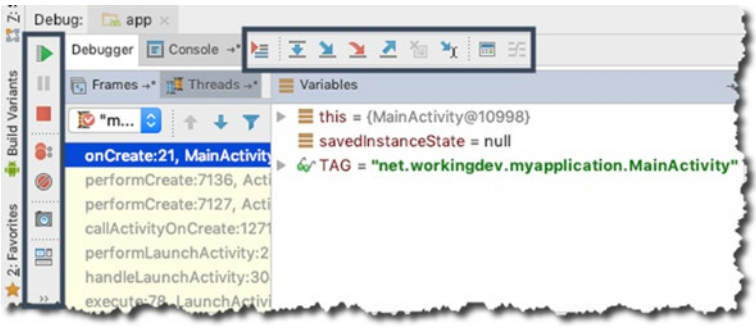

Figure 4-6. Debugger tool bar

Lastly, you can get the single-step actions via the keyboard shortcuts explained in Table 4-1.

Table 4-1. Debugger Keyboard Shortcuts

	Windows/Linux	macOS
Debug	`Shift + F9`	`Ctrl + D`
Resume	`F9`	`Command + Option + R`
Step into	`F7`	`F7`
Step over	`F8`	`F8`
Step out	`Shift + F8`	`Shift + F8`

Chapter Summary

- The three kinds of errors you may encounter are compile type or syntax errors, runtime errors, and logic errors.
- Syntax errors are the easiest to fix. Android Studio bends over backwards for you so you can quickly spot syntax errors. There are various ways to fix syntax errors with AS3, but most of the time, the quick fix should do it.
- You can walk through your code line by line by setting breakpoints and using the single-step actions.

Chapter 5

Unit Testing

What this chapter covers:

- Testing basics
- How to run efficient unit tests
- Adding and implementing unit tests

The practice of software testing has been around since the dawn of software development. When you write programs, you need a way to verify that they work in the intended manner. In the early days of software development, testing was done in a somewhat ad hoc way: you'd write some code, write a few manual tests to see if it looks right, go back to the code for some adjustment, and rinse-repeat. That's not the case anymore. Modern software has become so sophisticated that it cannot simply rely on a few manual tests to ensure it works properly.

In this chapter, you'll breeze through how to do unit testing in Android. Unit testing is functional testing that's done by a developer; not a QA. A unit test is usually simple; it's a particular thing that a method might do or produce. An application typically has many unit tests because each test is a very narrowly defined set of behaviors. So, you'll need lots of tests to cover the whole functionality. You will use JUnit to write your tests.

JUnit is a regression testing framework written by Kent Beck and Erich Gamma. Among their other achievements, you might remember one of them as the guy who created extreme programming and the other one from Gang of Four (GoF, Design Patterns), respectively.

Java developers have long used JUnit for unit testing. Android Studio comes with JUnit and is very well integrated in it. You don't have to do much by way of setup. You only need to write your tests.

T. Hagos, *Android Studio IDE Quick Reference*,
https://doi.org/10.1007/978-1-4842-4953-6_5

JVM Test vs. Instrumented Test

If you examine any Android application, you'll see that it has two parts: a Java-based behavior and an Android-based behavior.

The Java part is where you code business logic, calculations, and data transformations. The Android part is where you actually interact with the Android platform. This is where you receive input from users or show results to them. It makes perfect sense if you can test the Java-based behavior separate from the Android part because it's much quicker to execute. Fortunately, this is already the way it's done in Android Studio. When you create a project, Android Studio creates two separate folders: one for the JVM tests and another for the instrumented tests. Figure 5-1 shows the two test folders in the Android view and Figure 5-2 shows the same two folders in the Project view.

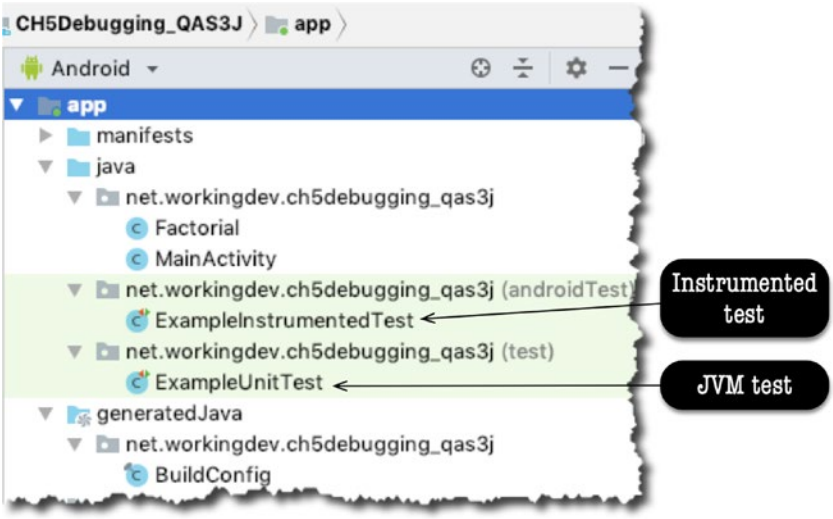

Figure 5-1. The JVM test and the instrumented test in the Android view

The JVM test folder is simply referred to as `test` while the one for the instrumented test is called `androidTest`.

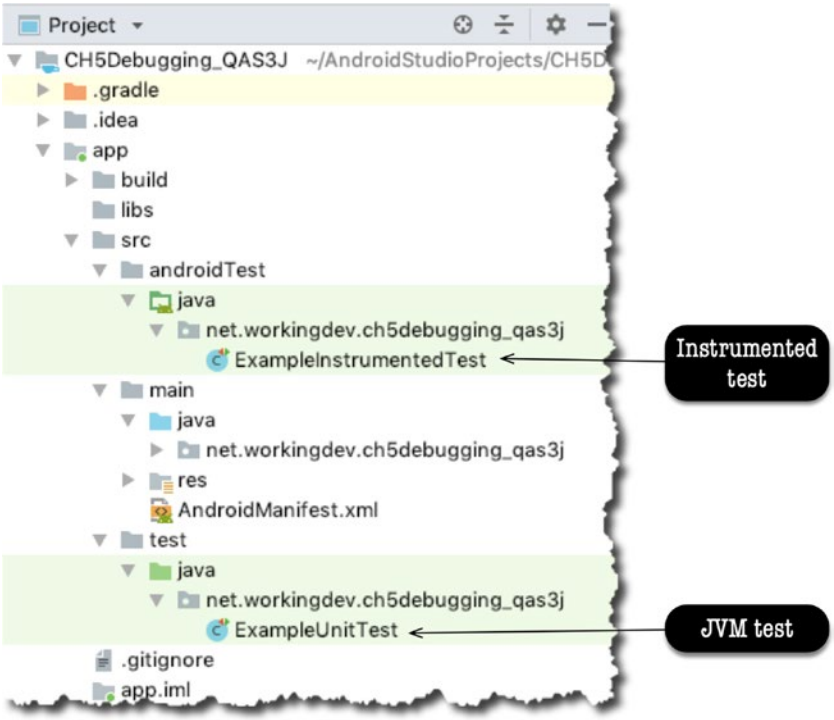

Figure 5-2. The JVM test and the instrumented test in the Project view

As you can see from either Figure 5-1 or 5-2, Android Studio went the extra mile to generate sample test files for both the JVM and instrumented tests. The example files are there to serve as quick references; they show what unit tests might look like.

A Simple Demo

To get started, create a project with an empty activity in Android Studio. Create a class and name it `Factorial.java`. Fill it up with the code shown in Listing 5-1.

Listing 5-1. Factorial.java

```
public class Factorial {
  public static double factorial(int arg) {
    if (arg == 0) {
      return 1.0;
    }
```

```
    else {
      return arg + factorial(arg - 1);
    }
  }
}
```

Make sure that `Factorial.java` is open in the main editor, as shown in Figure 5-3. Then, from the main menu bar, go to Navigate ➤ Test. Similarly, you can also create a test using the keyboard shortcut (`Shift + Command + T` in macOS and `Ctrl + Shift + T` for Linux and Windows).

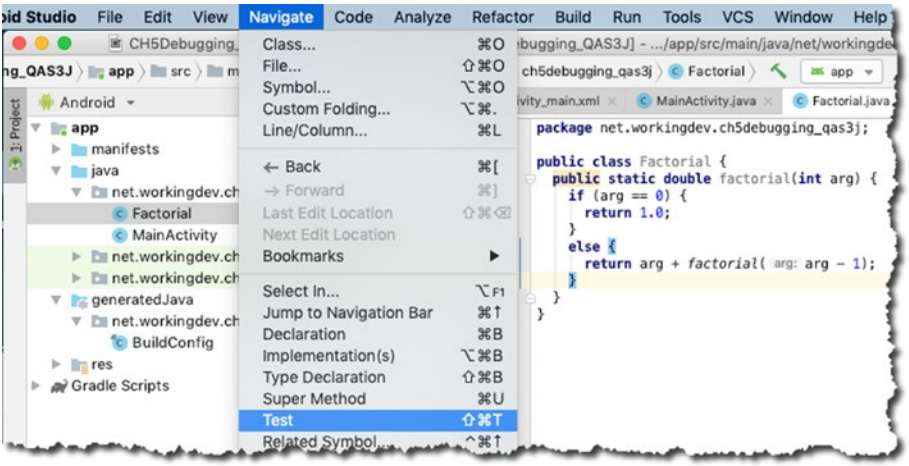

Figure 5-3. *Creating a test for Factorial.java*

Right after you click the Test option, a pop-up dialog (Figure 5-4) will prompt you to click another link. Click the Create New Test option, as shown in Figure 5-4.

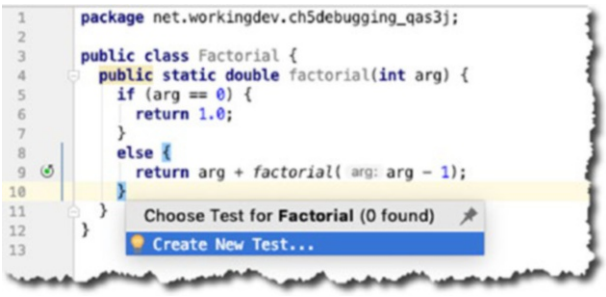

Figure 5-4. *The Create New Test pop-up*

Right after creating a new test, you'll see another pop-up dialog, shown in Figure 5-5, which I've annotated. Please follow the annotations and instructions in Figure 5-5.

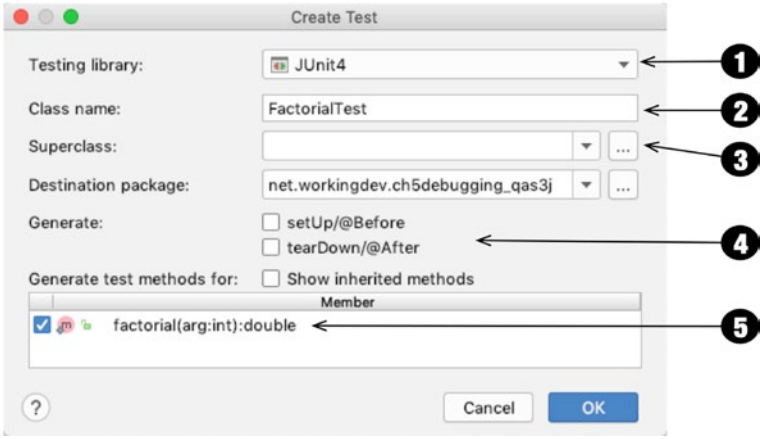

Figure 5-5. *Creating FactorialTest*

❶ You can choose which testing library you want to use. You can choose JUnit 3, 4, or 5. You can even choose Groovy JUnit, Spock, or TestNG. I used JUnit4 because it comes installed with Android Studio.

❷ The convention for naming a test class is "name of the class to test" + "Test". Android Studio populates this field using this convention.

❸ Leave this field blank because you don't need to inherit from anything.

❹ You don't need the `setUp()` and `tearDown()` routines for now, so leave them unchecked.

❺ Check the `factorial()` method because you want to generate a test for this.

When you click the OK button, Android Studio will ask where you want to save the test file. This is a JVM test, so you want to save it in the `test` folder (not in `androidTest`), as shown in Figure 5-6. Click OK.

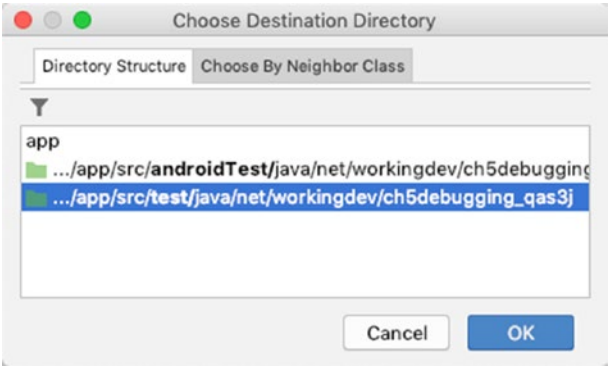

Figure 5-6. Choosing the destination directory

Android Studio will now create the test file for you. If you open `FactorialTest.java`, you'll see the generated skeleton code shown in Figure 5-7.

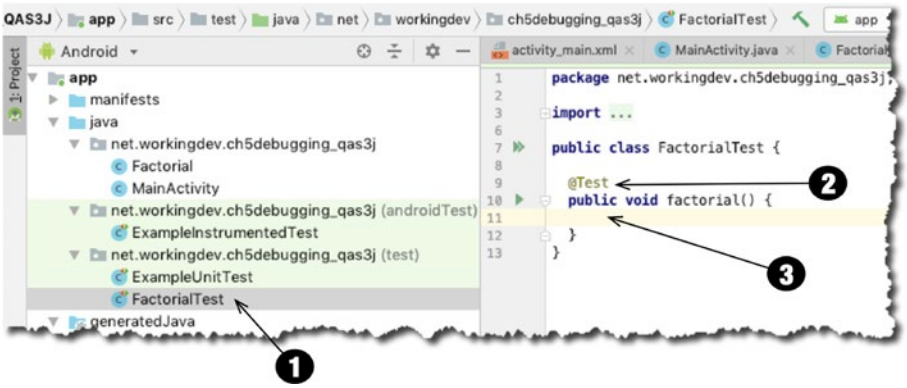

Figure 5-7. FactorialTest.java in Project view and the main editor

❶ The file `Factorial.java` was created under the `test` folder.

❷ A `factorial()` method was created and it's annotated as `@Test`. This is how JUnit will know that this method is a unit test. You can prepend your method names with "test," as in `testFactorial()`, but that is not necessary. The `@Test` annotation is enough.

❸ This is where you put your assertions.

See how simple that was? Creating a test case in Android Studio doesn't really involve you that much in terms of setup and configuration. All you need to do now is write your test.

Implementing a Test

JUnit supplies several static methods that you can use in your test to make assertions about your code's behavior. You use assertions to show an expected result, which is your control data. It's usually calculated independently and is known to be true or correct—that's why you use it as control data. When the expected data is returned from the assertion, the test passes; otherwise, the test fails. Table 5-1 shows the common `assert` methods you might need for your code.

***Table 5-1.** Common Assert Methods*

Method	Description
`assertEquals()`	Returns true if two objects or primitives have the same value
`assertNotEquals()`	The reverse of `assertEquals()`
`assertSame()`	Returns true if two references point to the same object
`assertNotSame()`	Reverse of `assertSame()`
`assertTrue()`	Tests a Boolean expression
`assertFalse()`	The reverse of `assertTrue()`
`assertNull()`	Tests for a null object
`assertNotNull()`	The reverse of `assertNull()`

Now that you know a couple of `assert` methods, you're ready to write some tests. Listing 5-2 shows the code for `FactorialTest.java`.

***Listing 5-2.** FactorialTest.java*

```
import org.junit.Test;
import static org.junit.Assert.*;

public class FactorialTest {

  @Test
  public void factorial() {
    assertEquals(1.0, Factorial.factorial(1),0.0);
    assertEquals(120.0, Factorial.factorial(5), 0.0);
  }
}
```

Your `FactorialTest` class has only one method because it's for illustration purposes only. Real-world code would have many more methods than this, to be sure.

Notice that each test (method) is annotated by `@Test`. This is how JUnit knows that `factorial()` is a test case. Notice also that `assertEquals()` is a method of the `Assert` class, but you're not writing the fully qualified name here because you've got a static import on `Assert`. It certainly makes life easier.

The `assertEquals()` method takes three parameters. They're illustrated in Figure 5-8.

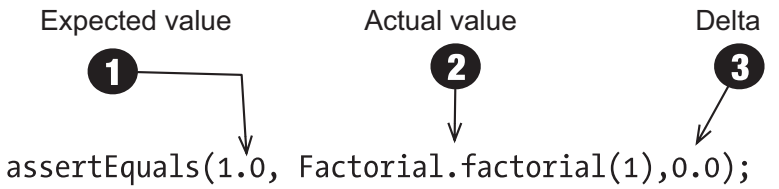

***Figure 5-8.** The assertEquals Method*

❶ The **Expected** value is your control data; this is usually hard-coded in the test.

❷ The **Actual** value is what your method returns. If the expected value is the same as actual value, the `assertEquals()` passes; your code is behaving as expected.

❸ **Delta** is intended to reflect how close the *actual* and *expected* values can be and still be considered equal. Some developers call this parameter the "fuzz factor." When the difference between the expected and actual value is greater than the fuzz factor, then `assertEquals()` will fail. I used 0.0 here because I don't want to tolerate any kind of deviation. You can use other values like 0.001, 0.002, and so on; it depends on your use-case and how much fuzz your app is willing to tolerate.

Now your code is complete. You can insert a couple more asserts in the code so you can get into the groove of things, if you prefer.

There are couple of things I did not include in this sample code. I did not override the `setUp()` and `tearDown()` methods because I didn't need to. You would normally use the `setUp()` method if you need to set up database connections, network connections, and so on. Use the `tearDown()` method to close whatever you opened in the `setUp()`.

Now let's run the test.

Running a Unit Test

You can run just one test or all the tests in the class. The little green arrows in the gutter of the main editor are clickable. Clicking the little arrow beside the name of the class will run all the tests in the class. When you click the one beside the name of test method, it will run only that test case. See Figure 5-9.

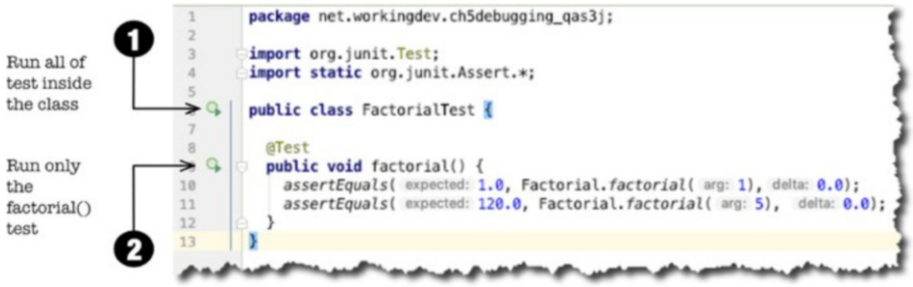

Figure 5-9. *FactorialTest.java in the main editor*

Similarly, you can also run the test from the main menu bar via Run ➤ Run.

Figure 5-10 shows the result of the text execution.

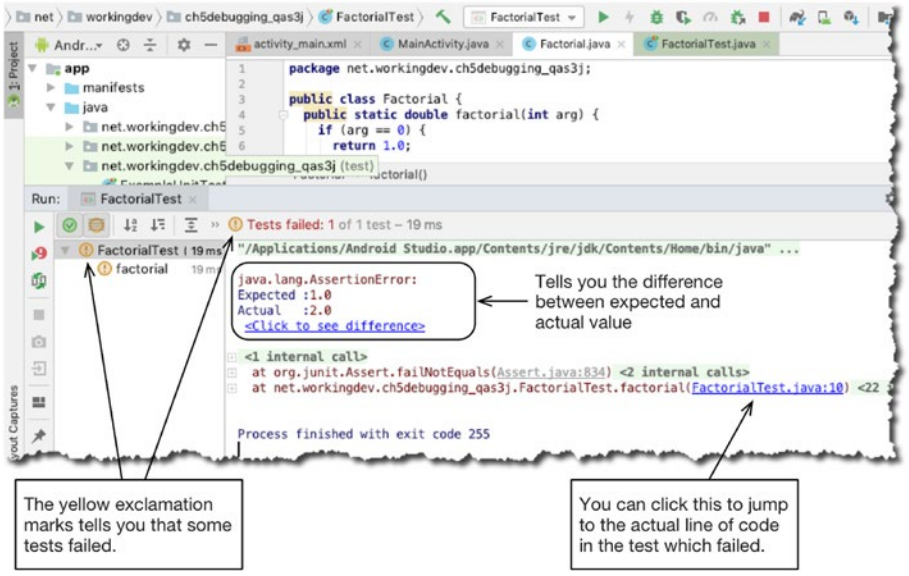

Figure 5-10. *Result of running FactorialTest.java*

Android Studio gives you plenty of cues so you can tell if your tests are passing or failing. The first run tells you that there's something wrong with `Factorial.java`; the `assertEquals()` has failed.

> **Tip** When a test fails, it's best to use the debugger to investigate the code. `FactorialTest.java` is no different than any other class in the project; it's just another Java file, so you can definitely debug it. Put some breakpoints in strategic places of your test code, and then instead of "running" it, run the debugger so you can walk through it.

The test failed because the factorial of 1 isn't 2, it's 1. If you look closer at `Factorial.java`, you'll notice that the factorial value isn't calculated properly.

Edit the `Factorial.java` file to change

```
return arg + factorial(arg - 1);
```

to

```
return arg * factorial(arg - 1);
```

If you run the test again, you will see successful results, as shown in Figure 5-11.

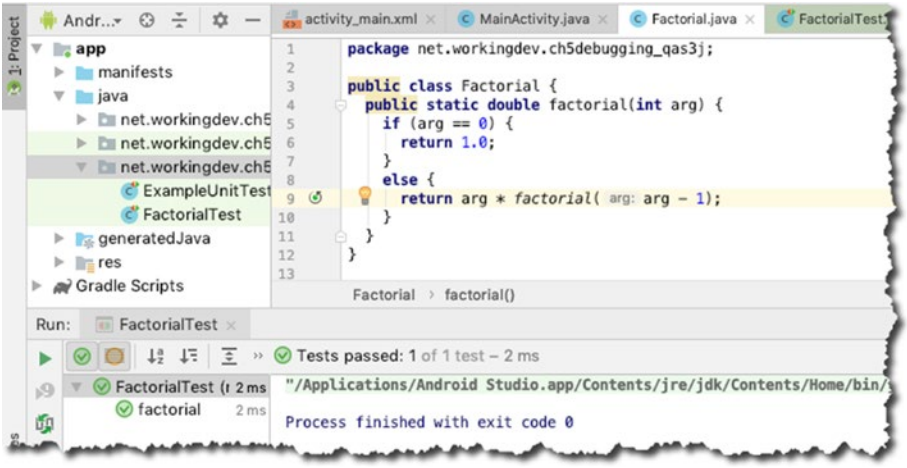

Figure 5-11. Successful test

Instead of yellow exclamation marks, you now see green check marks. Instead of seeing "Test failed," you now see "Test passed." Now you know that your code works as expected.

Test First

Having a way to test your code in an automated way and at a high frequency is important. It's so important that many developers advocate writing your test suite first, even before you write your actual code. This is called test-driven development, or TDD for short. The basic idea is that the tests represent the requirements of your application which the code must satisfy.

To see how to do TDD in Android Studio, delete the `Factorial.java` file in your project and run the tests again. Of course, it will fail; that's the idea. You always start with failing tests. Now, create the `Factorial` class and start putting in the methods and the code that will make the tests pass.

Once you have a couple of test cases, a bit more code, and no errors, try running the tests again. The idea is to run the tests every time you make significant (enough) changes to the code. That way, when all your tests pass, you know that you're not introducing any breaking changes to your code.

Compare this practice to what a lot of devs do, which is write a bunch of code first and then write a little test program to verify the code, maybe something with a static `main()` method and a couple of `println`s. Then they throw away the test code—as if their code won't ever break again! Hopefully with this chapter, you now know better than to throw away test code. You should cultivate it and run it often to make sure that your code is to spec.

Chapter Summary

- Unit testing is a core development task. It should be a core development task because modern and sophisticated software shouldn't rely on puny, ad hoc, throw-away tests.
- A JVM test is different from an instrumented test. A JVM test is used for the Java part of your code—the part that doesn't need to interact with the Android platform.
- Android Studio lets you separate the JVM test from instrumented tests.
- JVM tests are like regular class files: you can debug and step through them.
- Each JUnit test is annotated by `@Test`. This is how JUnit knows which methods are supposed to be test cases.

Chapter 6

Instrumented Testing

What this chapter covers:

- Details about instrumented testing
- How to create UI test interactions
- Basic test interactions
- Implementing test verifications
- Test recording

You learned how to perform JVM testing in the previous chapter. In this chapter, you'll do some testing that deals with the Android part of an application. Unit testing that interacts with the Android platform is known as instrumented testing, and you will use the Espresso framework to do this.

About Espresso

Google released Espresso in 2013, and with its 2.0 release, Espresso became a part of the Android Support Repository. The general steps when working with Espresso tests are the following:

1. **Match**: Use a matcher to target a specific component like a button or TextView. A ViewMatcher lets you find a View object in the hierarchy.
2. **Act**: Use a ViewAction object to perform an action like a click on a targeted View object.
3. **Assert**: Use an assertion on the state of a View.

T. Hagos, *Android Studio IDE Quick Reference*,
https://doi.org/10.1007/978-1-4842-4953-6_6

Imagine you have a simple screen that has a button and a TextView. When you click the button, you see the text “Hello World” on the TextView. You can write the test like this:

```
onView(withId(R.id.button))   ❶
      .perform(Click())       ❷
onView(withId(R.id.textview)) ❸
      .check(matches(withText("Hello World"))); ❹
```

❶ Use a ViewMatcher to find a View object. You’re looking for a View with an id of `button`. Remember that when you’re using `onView()`, Espresso waits until all synchronization conditions are met before it performs the corresponding UI action.

❷ When you find it, use a ViewAction to do something with it; in this case, you want to click it.

❸ Once again, you use a ViewMatcher to find a View object. This time, you’re trying to find a TextView with an id of `textview`.

❹ When you find it, you want to check if its text property is a match to “Hello World.”

Setting Up a Simple Test

Let’s set up a simple project, something that has an empty activity. Listing 6-1 shows the XML layout code and Listing 6-2 shows the MainActivity code.

Listing 6-1. activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout 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"
  tools:context=".MainActivity">

  <TextView
    android:id="@+id/textView"
    android:layout_width="241dp"
    android:layout_height="wrap_content"
    android:layout_marginTop="147dp"
    android:text="TextView"
    android:textSize="36sp"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
```

```
  <Button
    android:id="@+id/btnhello"
    android:layout_width="146dp"
    android:layout_height="wrap_content"
    android:layout_marginTop="20dp"
    android:onClick="onClick"
    android:text="hello"
    android:textSize="36sp"
    app:layout_constraintEnd_toEndOf="@+id/textView"
    app:layout_constraintHorizontal_bias="0.494"
    app:layout_constraintStart_toStartOf="@+id/textView"
    app:layout_constraintTop_toBottomOf="@+id/textView" />

  <Button
    android:id="@+id/btnworld"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="8dp"
    android:onClick="onClick"
    android:text="world"
    android:textSize="36sp"
    app:layout_constraintEnd_toEndOf="@+id/btnhello"
    app:layout_constraintHorizontal_bias="0.0"
    app:layout_constraintStart_toStartOf="@+id/btnhello"
    app:layout_constraintTop_toBottomOf="@+id/btnhello" />
</android.support.constraint.ConstraintLayout>
```

The layout code is fairly simple, as you can see: it has one TextView and two buttons. Both buttons call the `onClick()` method in MainActivity when the user clicks them.

Listing 6-2. MainActivity

```
public class MainActivity extends AppCompatActivity {

  TextView txtview;

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    txtview = (TextView) findViewById(R.id.textView);
  }

  public void onClick(View view) {
    switch(view.getId()) {
      case R.id.btnhello:
        txtview.setText("hello");
        break;
```

```
      case R.id.btnworld:
        txtview.setText("world");
        break;
    }
  }
}
```

The `onClick()` method in MainActivity basically tries to get the id of the button that was clicked and routes program logic according to it. If `btnhello` is clicked, you set text content of the TextView to "hello" and if `btnworld` is clicked, you set the content to "world"—it's simple enough. To verify this behavior, you can set up an instrumented test.

In the previous chapter, you wrote the test class in `src/test` because it was a JVM test. Now, you will write the test class inside `src/androidTest` because this will be an instrumented test. Listing 6-3 shows the code for the instrumented test class.

Listing 6-3. MainActivityTest

```
import android.support.test.rule.ActivityTestRule;
import android.support.test.runner.AndroidJUnit4;
import org.junit.Rule;
import org.junit.Test;

import static android.support.test.espresso.Espresso.onView;  ❶
import static android.support.test.espresso.action.ViewActions.click;
import static android.support.test.espresso.assertion.ViewAssertions.matches;
import static android.support.test.espresso.matcher.ViewMatchers.withId;
import static android.support.test.espresso.matcher.ViewMatchers.withText;

public class MainActivityTest {

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

  @Test ❸
  public void buttonHelloTest() {
    onView(withId(R.id.btnhello)) ❹
        .perform(click());        ❺

    onView(withId(R.id.textView)) ❻
        .check(matches(withText("hello"))); ❼
  }
```

```
  @Test
  public void buttonWorldTest() {
    onView(withId(R.id.btnworld))
        .perform(click());

    onView(withId(R.id.textView))
        .check(matches(withText("world")));
  }
}
```

❶ You want to statically import the Espresso matchers, actions, and asserts so you won't have to fully qualify them later in the code.

❷ This just intercepts your test method calls and makes sure that the activity is launched before you perform any test.

❸ You need to annotate each test method with `@Test`.

❹ Find the `btnhello` object using the `withId()` method.

❺ Then you simulate a click using a `ViewAction.click()`.

❻ Then you find the TextView, using a `withId()` method again.

❼ Finally, you assert if the TextView contains the text "hello."

You can run the instrumented test the same way you run the JVM test. You can either

- Click the arrows in the IDE gutter,
- Right-click the test and use the context-sensitive menu, and then choose the Run MainActivityTest option, or
- Go to the main menu bar, choose Run ➤ Run, and then choose MainActivityTest.

Recording Espresso Tests

Android Studio includes a feature where you can run your app, record the interaction, and create an Espresso test using the recording. To get started, go to the main menu bar and then choose Run ➤ Record Espresso Test, as shown in Figure 6-1.

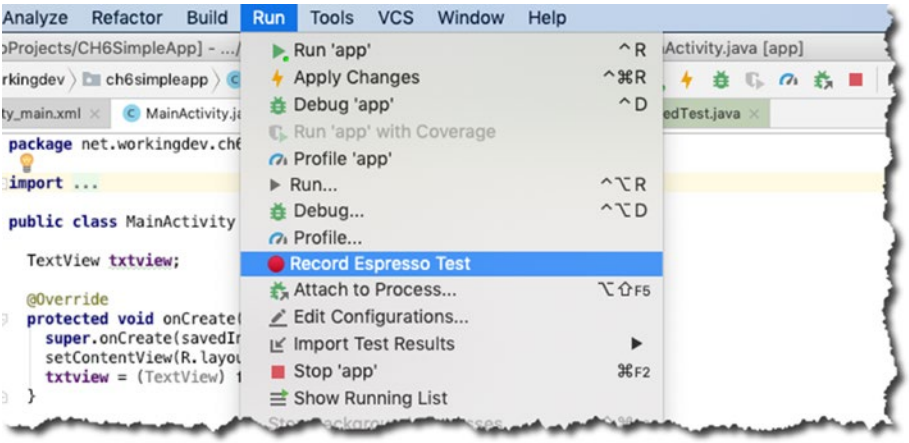

Figure 6-1. Recording an Espresso test

After choosing the Record Espresso Test option, you can now interact with the app like usual, but this time, the interaction is recorded. If you click one of the buttons, say the HELLO button, the test recorder screen will pop up, as shown in Figure 6-2.

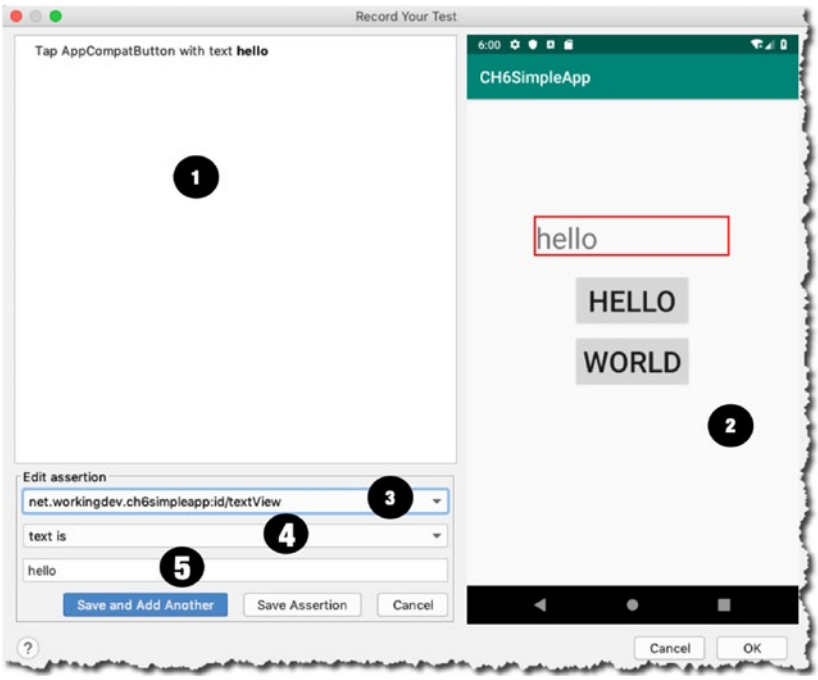

Figure 6-2. Espresso recorder

❶ This section shows each interaction with the app. At this point, I clicked the app only once; I clicked the HELLO button.

❷ This section is the ViewMatcher, but done visually. If you click the TextView, like I did here, it goes over as an item to the Edit Assertion section.

❸ The TextView is selected here because I clicked it in the ViewMatcher section (item 2).

❹ This is where you choose the assertion. In this case, it's the hamcrest "text is."

❺ The actual value of the TextView you'd like to assert.

You can click the Save and Add Another option if you'd like to add another test, or the Save the Assertion option and finish the recording. Figure 6-3 shows the next screen.

Figure 6-3. Espresso recorder with assertion saved

When you click OK, the recorder will prompt for the name of the class where it will save the recording as a test class, as shown in Figure 6-4.

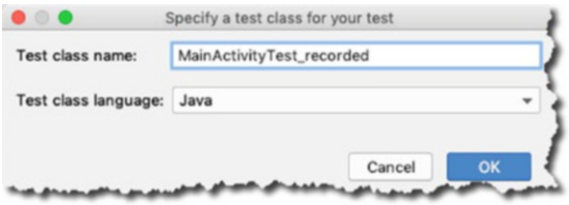

Figure 6-4. Espresso test with test saved

When you go the `src/androidTest` folder, you'll find the newly generated test class from your recording. You can now run the generated test the same way you ran MainActivityTest earlier.

> **Note** Two factoids about Espresso: 1) The Espresso recorder is one of the most used tools when using Espresso, according to Android Studio analytics, and 2) The Espresso recorder was originally named "cassette."

More on Espresso Matchers

Espresso has a variety of matchers but the one that's commonly used is the ViewMatcher; it's what you used in the earlier examples. Here are the other matchers in Espresso:

- **CursorMatcher**: Used for Android Adapter Views that are backed by Cursors to match specific data rows
- **LayoutMatcher**: To match and detect typical layout issues, such as TextViews that have ellipses or multi-line text
- **RootMatcher**: To match Root objects that are dialogs or Roots that can receive touch events
- **PreferenceMatcher**: To match Android Preferences and let you find View components based on their key, summary text, etc.
- **BoundedMatcher**: To create your own custom matcher for a given type

In the previous examples, you used the ViewMatcher to find Views via their ids. You can find Views using other things, such as

- **Its value**: You can use the `withText()` method to find a View that matches a certain String expression.
- **The number of its child**: Using the `hasChildCount()` method, you can match a View that has a very specific child count.
- **Its class name**: Using the `withClassName()` method.

ViewMatchers can also tell you whether a View object is

- Enabled, by using the `isEnabled()` method
- Focusable, by using the `isFocusable()` method
- Displayed via `isDisplayed()`
- Checked via `isChecked()`
- Selected via `isSelected()`

There are plenty more methods you can use in the ViewMatchers class so make sure to check them out at https://bit.ly/viewmatchers.

Espresso Actions

Espresso Actions let you interact with View objects programmatically during a test. You used the click earlier, but there's a lot more that ViewActions will let you do. The method names are very descriptive so they don't need further explanations; you can clearly see what they do. Here are a few of them:

- `clearText()`
- `closeSoftKeyboard()`
- `doubleClick()`
- `longClick()`
- `openLink()`
- `pressBack()` presses the back button
- `replaceText(String arg)`
- `swipeDown()`

- `swipeRight()`
- `swipeUp()`
- `typeText(String arg)`

There are more actions available so make sure you visit the API documentation for the ViewAction object.

Also, make sure you visit the official documentation for Espresso on the Android Develop website at https://bit.ly/androidstudioespresso. You've only scratched the surface of Espresso here!

Chapter Summary

- Put JVM tests in `src/test` and put instrumented tests in `src/androidTest`.
- You can use Espresso to create instrumented tests; the two things you need in Espresso are the ViewMatchers and ViewActions.
- The general steps for writing Espresso tests are 1) find the View object using ViewMatchers, 2) perform an action on the View with ViewActions, and 3) do your assertions.
- An easy way to create Espresso tests is to use the Espresso recorder.

Chapter 7

Android Studio Profiler

What this chapter covers:

- Android Studio Profiler
- CPU view
- Memory view

Android Studio 3.0 replaces the old Android Monitor with the new Android Profiler. It's an integrated view for profiling an app's memory consumption, network usage, CPU usage, and power usage. In this chapter, you'll take a quick look at the Profiler.

The Profiler

The Profiler is new in Android Studio 3. It replaces the Android Monitor and offers a new, unified, shared timeline view for CPU, memory, network, and energy graphs.

You can get to the Profiler by going to the main menu bar and then View ➤ Tool Windows ➤ Profiler. Figure 7-1 shows the Profiler.

T. Hagos, *Android Studio IDE Quick Reference*,
https://doi.org/10.1007/978-1-4842-4953-6_7

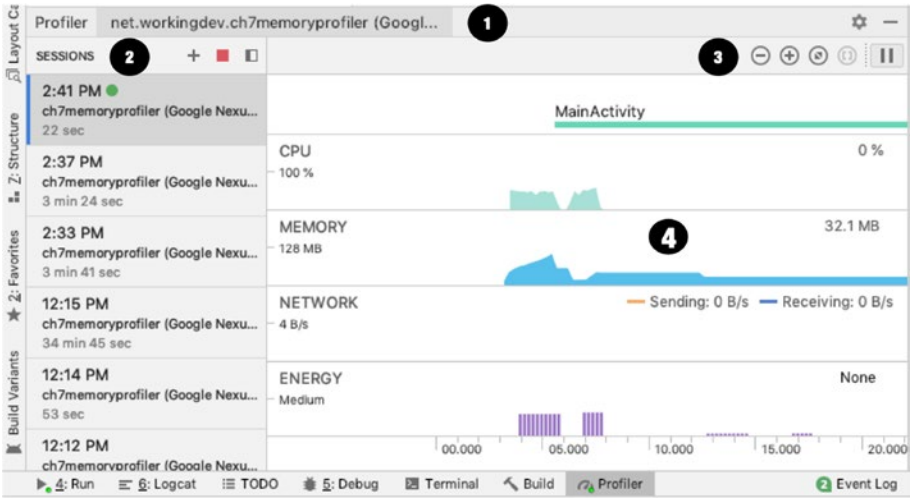

Figure 7-1. Profiler

❶ Shows the process and device being profiled.

❷ Shows you which sessions to view. You can also add new sessions from here by clicking the + button.

❸ Use the zoom buttons to control how much of the timeline to view.

❹ The new shared timeline view lets you see all the graphs for CPU, memory, network, and energy usage. At the top, you will also see important app events, like user inputs or activity state transitions.

As soon as you launch an application, either on an attached device or an emulator, you'll see its graph on the Profiler.

> **Note** If you try to profile an APK with a version lower than API level 26, you will see some warnings because Android Studio needs to fully instrument your code. You will need to enable advance profiling, but if your APK is Oreo or higher, you won't see any warnings.

If you click any of the charts, the Profiler window will take you to one of the detailed views. If you click the CPU, for example, you'll see the detailed view for the CPU utilization.

CPU

Figure 7-2 shows the detailed view for the CPU utilization on the sample app I was running.

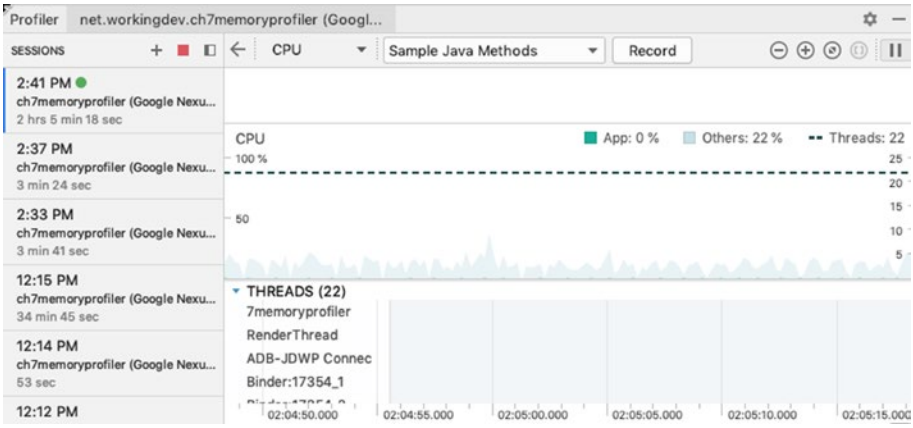

Figure 7-2. CPU view

Aside the from live utilization graph, the CPU detailed view also shows a list of all the threads in the app and their states. You can see if the threads are waiting for I/O or when they are active.

You may have noticed the Record button in Figure 7-2. If you click that button, you can get a report on all the methods that were executed in a given period. Notice also the selected trace type in the dropdown menu (Sample Java Methods); this *trace type* has a smaller overhead but it's not as detailed or as accurate as the *instrumented type* (Trace Java Methods), meaning the sampled type may miss the execution of a very short-lived method. You might think, "Just always use the instrumented type then." You have to remember, though, that while an instrumented type can record every method call, on Android Devices before version 8, there is a limit on how much data can be captured, so if you use the instrumented trace, that limit will be reached quickly. You can change that limit by editing the configuration for the instrumented capture. On the trace type dropdown, choose the Edit Configurations options, as shown in Figure 7-3.

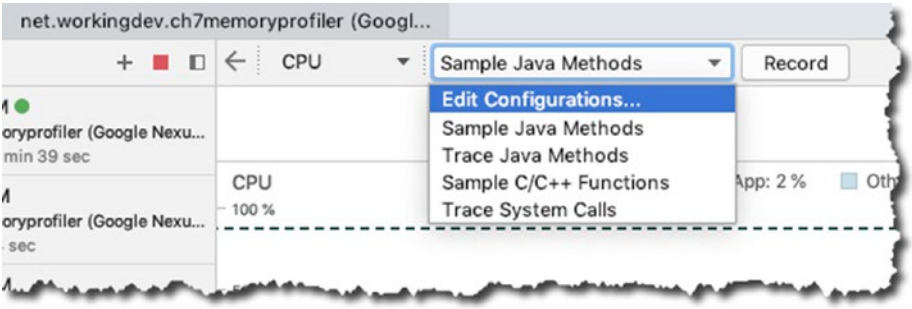

Figure 7-3. Editing the configuration

Figure 7-4 shows the sampling interval and file size limit settings, which you can use to adjust how frequent the sampling will be and how big of a file size you'd like to allocate for the recording. Just to reiterate, the file size limitation is only present on Android devices that are running Android 8.0 or lower (< API level 26). If your device has a higher Android version, you're not constrained by these limitations.

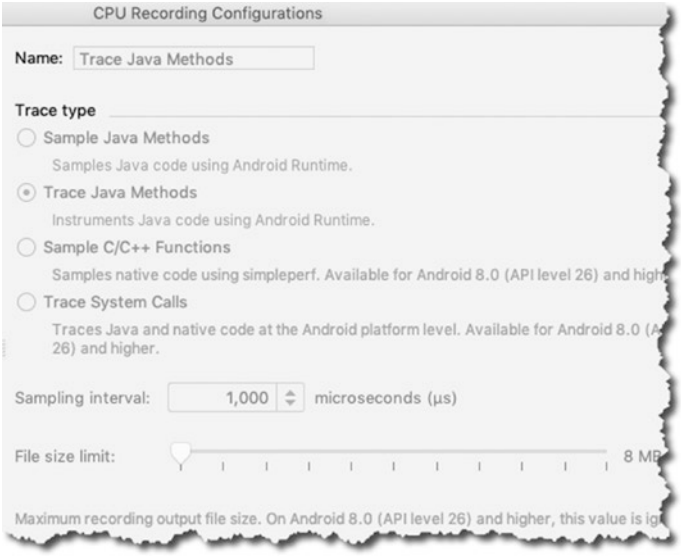

Figure 7-4. CPU recording configuration

If you click Record, Android Studio will begin capturing data. Click the Stop button when you'd like to stop recording, as shown in Figure 7-5.

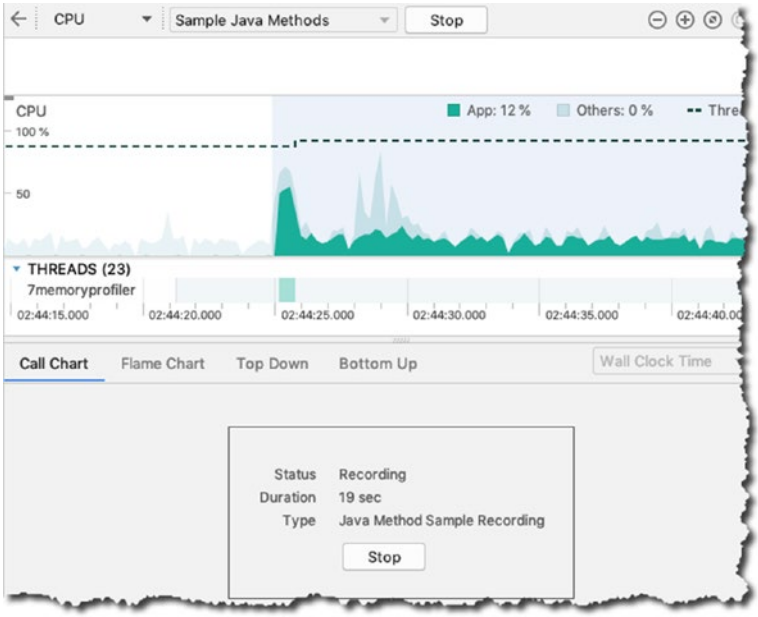

Figure 7-5. Recording a session

When you press Stop, you can take look at the individual threads, as shown in Figure 7-6.

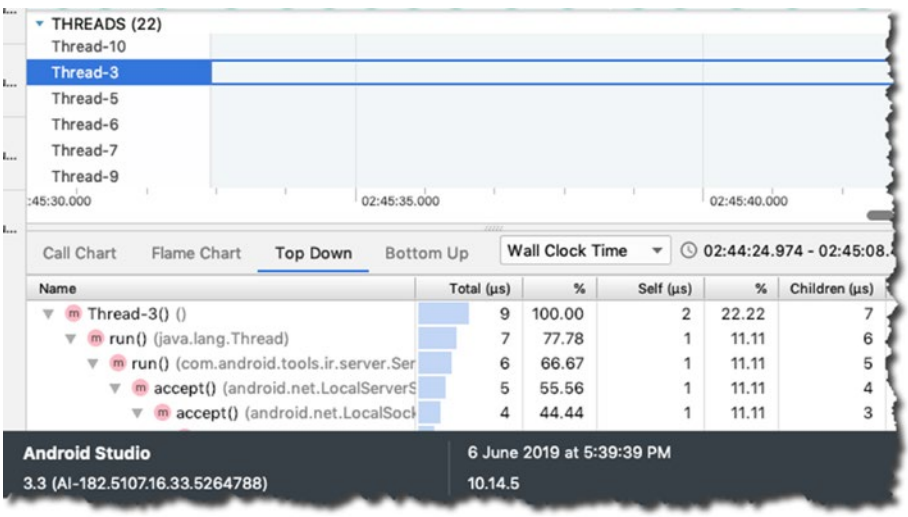

Figure 7-6. Inspecting the threads

Memory

The Memory profiler shows, in real time, how much memory your app is consuming. Figure 7-7 shows a snapshot of the memory view as I captured the memory footprint of a test app. As you can see, not only does the graph show how much memory your app is gulping, it also shows the breakdown, such as how much memory is used by the code, stack, graphics, Java, and so on.

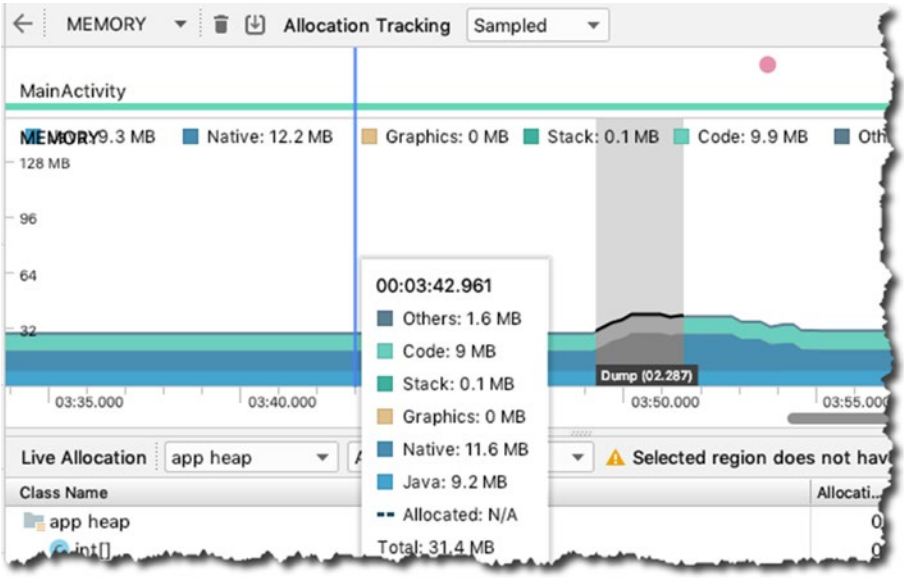

Figure 7-7. Memory view

You can force garbage collection in the Memory view. See that garbage can icon at the top? Yup, if you click that, it'll force a GC. The button to its right is also useful: the icon with a down-pointing arrow inside a box is a *memory dump*. If you click it, the Java heap will be dumped, and then you can inspect it, as shown in Figure 7-8.

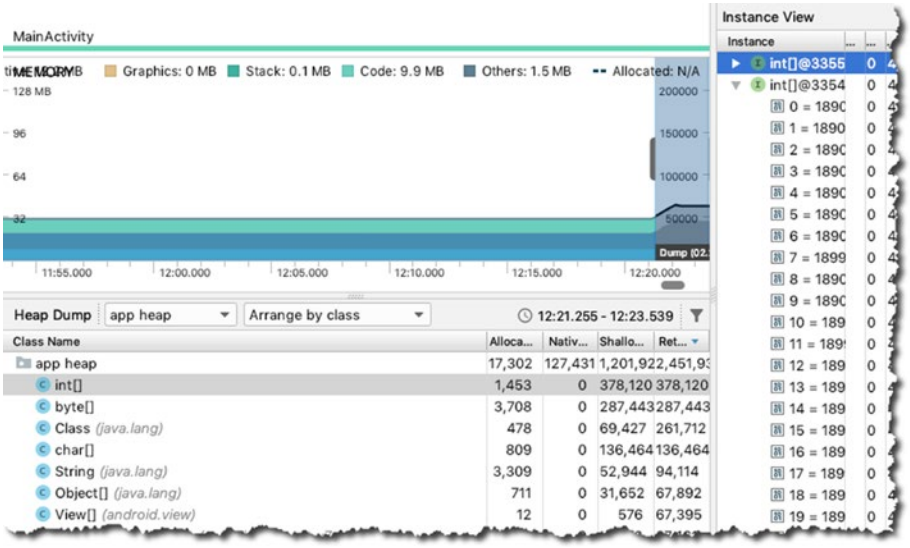

Figure 7-8. Java heap

The heap is a preserved amount of storage memory that the Android runtime allocates for the app. When you dump the heap, it gives you a chance to examine instance properties of objects, as shown in Figure 7-9.

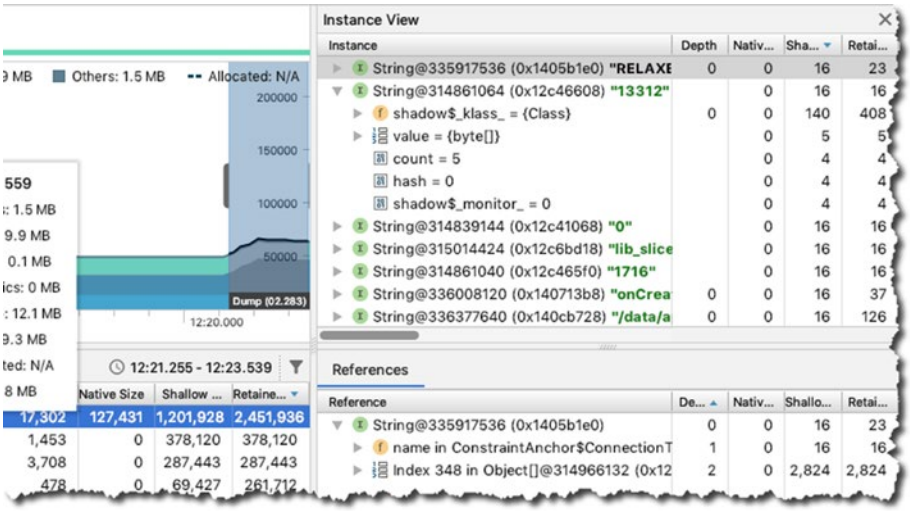

Figure 7-9. Instance view, Reference tab

The Reference tab can be very useful in finding memory leaks because it shows all the references pointing to object you're examining.

Another useful tool in the memory view is the Allocation tracker, shown in Figure 7-10.

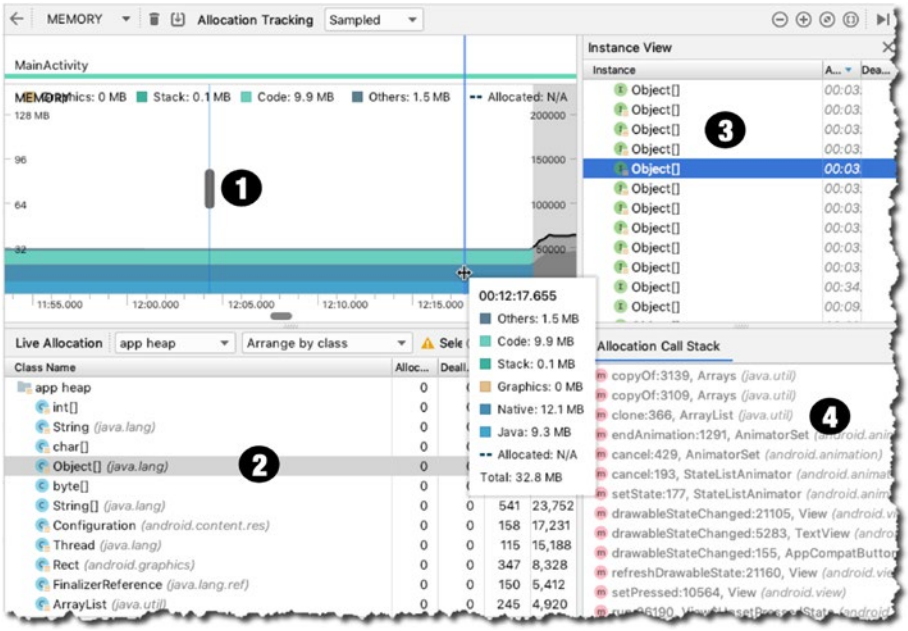

Figure 7-10. Allocation tracker

❶ Click anywhere in the timeline of the memory graph to view the allocation tracker. This will show you a list of all objects that were allocated and de-allocated at that point in time.

❷ This shows a list of all classes being used by the app at a point in time.

❸ This shows the list of all those objects allocated and de-allocated at a specific point in time.

❹ The tracker even includes the call-stack of the allocation.

Network

Like the other views in the Profiler, the Network view also shows real-time data. It lets you see and inspect data that is sent and received by your app. It also shows the total number of connections. Figure 7-11 shows a snapshot of the Network profiler.

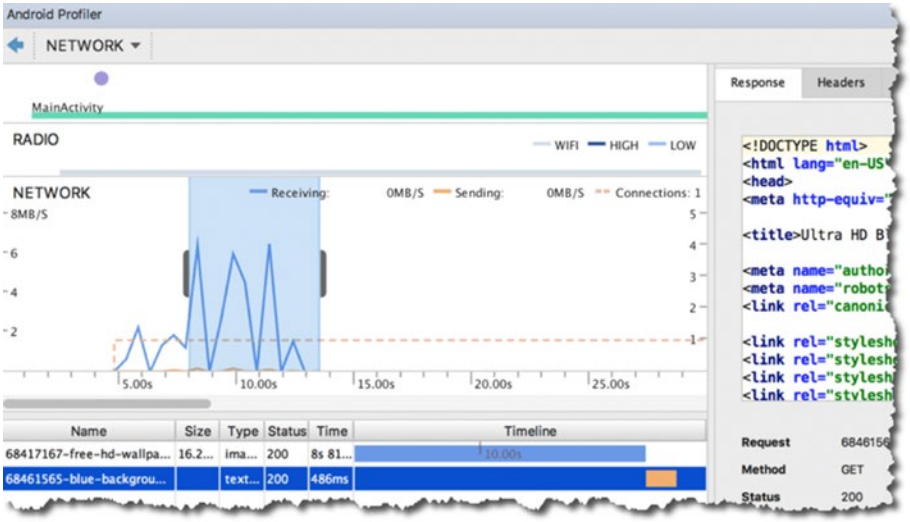

Figure 7-11. Network profiler

Every time your app makes a request to the network, it uses the WiFi radio to send and receive data. The radio isn't the most energy efficient; it's power-hungry, and if you don't pay attention to how your app makes network requests, that's a sure way to drain the device battery faster than usual.

When you use the Network profiler, a good way to start is to look for short spikes of network activity. When you see sharp spikes that rise and fall abruptly and are scattered all over the timeline, that smells like you could use some optimization by batching your network requests so as to reduce the number of times the WiFi radio needs to wake up and send or receive data.

Energy

By now you're probably seeing a pattern of how the Profiler works. It shows you real-time data. In the case of the Energy profiler, it shows data on how much energy your app is guzzling. Though it doesn't really show the direct measure of energy consumption, the Energy profiler shows an estimation of the energy consumption of the CPU, the radio, and the GPS sensor. Figure 7-12 shows a snapshot of the Energy profiler.

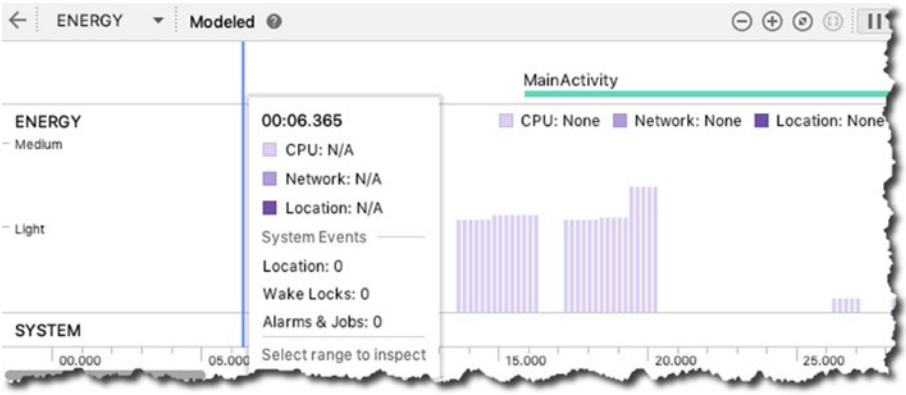

Figure 7-12. *Energy profiler*

You can also use the Energy profiler to find system events that affect energy consumption, such as wake locks, jobs, and alarms.

- A *wake lock* is a mechanism for keeping the screen on when the device would otherwise go to sleep. For example, when an app plays a video, it may use a wake lock to keep the screen on even when there's no user interaction. Using a wake lock isn't a problem, but forgetting to release one is; it keeps the CPU on longer than necessary, which will surely drain the battery faster.
- *Alarms* can be used to run background tasks that are outside your application's context at specific intervals. When an alarm goes off, the app can run some tasks. If it runs an energy-intensive piece of code, you'll definitely see it in the Energy profiler.
- A *job* can perform actions when certain conditions are met, such as when the network becomes available. You would usually create a job with JobBuilder and use JobScheduler to schedule the execution. When a job kicks in, you will be able to see it in the Energy profiler.

That was a quick tour of the Android Studio Profiler. Make sure you check out the official documentation at `https://bit.ly/androidstudioprofiler`.

Chapter Summary

- The old Android Monitor is gone. We use the Profiler now.
- The Profiler shows a unified view of how your app consumes memory, CPU, network, and battery resources.

Chapter 8

Gradle

What this chapter covers:

- The Android build process
- Gradle files
- Dependencies
- The Android Support Library

The Build Process

Building an APK or a bundle requires many involved steps. Figure 8-1 roughly illustrates the process.

> **Note** A bundle is a newer format for delivering Android executables. I'll talk about bundles a bit more in Chapter 12.

An app is a combination of loosely related Java source files, XML configuration files, UI definitions in XML, and more. Then it goes through compilation; there are resource compilers and the Java compiler. There are also libraries and other dependencies that need to be considered.

The compilation process produces some intermediate files like DEX executables and other compiled resources. A packager combines DEX executables, compiled resources, and certificates, and eventually produces either an APK or a bundle.

T. Hagos, *Android Studio IDE Quick Reference*,
https://doi.org/10.1007/978-1-4842-4953-6_8

> **Note** A DEX file is similar to a Java class file; they are executables but they're intended to run in Android's runtime—they're not the same as the Java runtime you have on your desktop. It's a byte-code runtime that's specifically designed for Android.

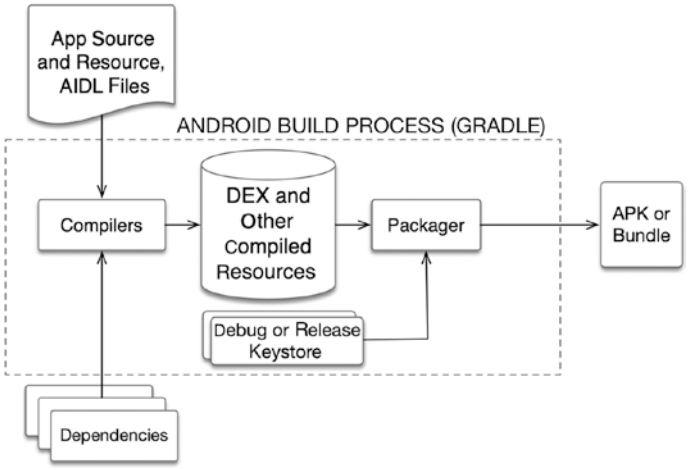

Figure 8-1. Android build process

Gradle and Android Studio take care of all the heavy lifting. If you've used ANT or Maven in your Java development before, Gradle is a lot like that, except instead of using XML for the syntax, it uses its own DSL (domain-specific language), which is based on the Groovy language.

The Build Files

Gradle relies on a couple of files. Figure 8-2 points them out.

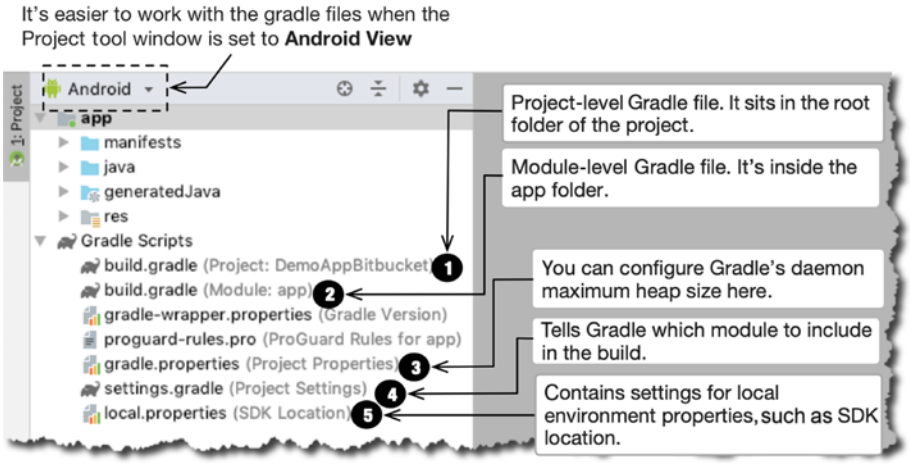

Figure 8-2. Gradle files

❶ `build.gradle` **(Project)**: This is it the root folder of the project. Some projects may have more than one module and it may be useful to define some properties that can be shared by all modules. This is the file where you do that.

❷ `build.gradle` **(module: app)**: The module-level `build.gradle` file is located inside, well, a module. In this example, the module-level Gradle file is inside the `app` folder; the `app` module is a default module that Android Studio generates when you create a project. This is the file that you're most likely to spend more time on, instead of the project-level Gradle file.

❸ `gradle.properties`: You can configure project-wide Gradle settings in this file, such as the amount of memory to allocate for the Gradle daemon. I usually just leave this alone.

❹ `settings.gradle`: This tells Gradle which module you to include in the build. If you're working on just the one module (which is "app"), then you also can simply leave this file alone.

❺ `local.properties`: This contains settings for your local environment, such as the location of the SDK, user credentials for Firebase, etc.

Module-Level Gradle File

Since you're most likely to interact with the module-level Gradle file, let's get to know it better. Listing 8-1 shows an annotated example of a module-level `build.gradle` file.

Listing 8-1. build.gradle (Module)

```
apply plugin: 'com.android.application'

android {
    compileSdkVersion 28 ❶
    defaultConfig {  ❷
        applicationId "net.workingdev.demoappbitbucket" ❸
        minSdkVersion 27 ❹
        targetSdkVersion 28 ❺
        versionCode 1  ❻
        versionName "1.0" ❼
        testInstrumentationRunner "android.support.test.runner.
        AndroidJUnitRunner"
    }
...
```

❶ `compileSdkVersion`: Indicates what API level or what version of Android you are using to build your app.

❷ `defaultConfig`: This block contains the default configuration for the module. Can you have more than one config? Yes, you can have more than one configuration for the app. This will happen when you have different behaviors targeting different platforms and so on. You could have a couple of build variants. But for now, let's just work with the default config.

❸ `applicationId`: This is the name your app will use in Google Play; it's how the app will be known.

❹ `minSdkVersion`: This will tell Google Play the minimum version of Android that the application supports.

❺ `targetSdkVersion`: If Android ever changes the API, this setting tells the runtime that the app expects the API to behave the way it did at API level 24. This way you have a consistent environment for your app.

❻ `versionCode`: Identifies the app's version. This is an incrementing integer.

❼ `versionName`: This is the string representation of `versionCode`. This is the one that's displayed in Google Play and the device's settings.

When you change anything in the Gradle file, Android Studio will prompt you to sync the file. A bar will be displayed on the top portion of the IDE telling you to "Sync now." You may also sync Gradle files by clicking the Sync button on the toolbar, as shown in Figure 8-3. Another way to do a sync is from the main menu bar: go to File ➤ Sync Project with Gradle Files.

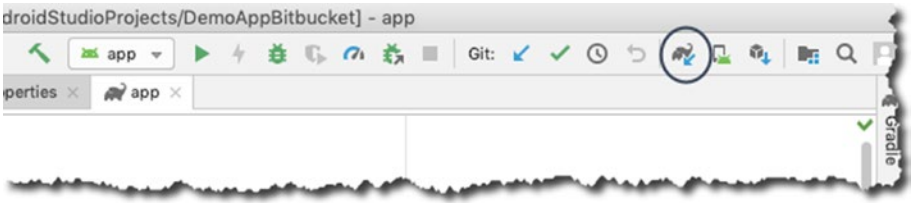

Figure 8-3. Sync the project with the Gradle files

If you don't want to edit the Gradle file by hand, there are ways to effect the changes via the GUI. Go to the main menu bar and choose File ➤ Project Structure, then click the app section, as shown in Figure 8-4.

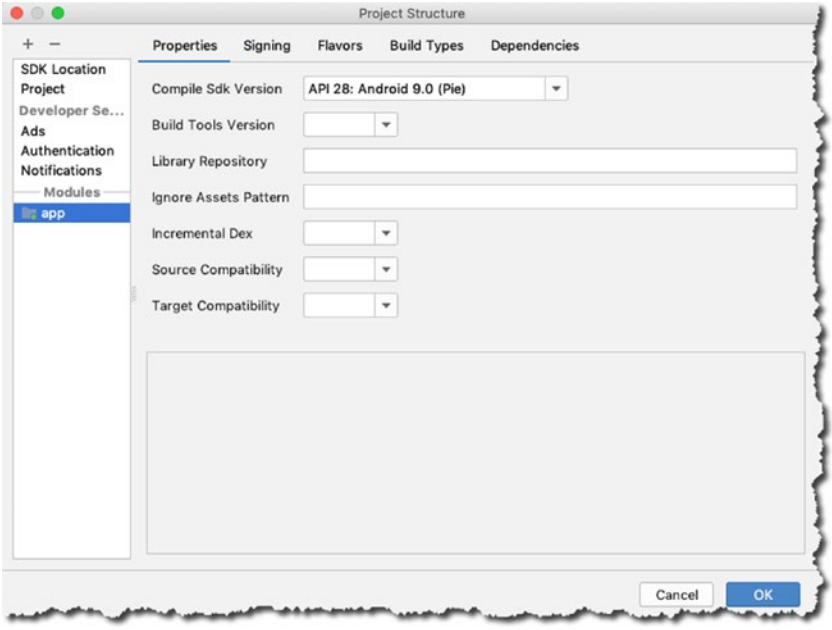

Figure 8-4. Project structure

If you explore the tabs of the Project Structure window, you'll see that you can also edit the entries of Gradle file from there.

Dependencies

Most applications rarely stand alone; they rely on other code. Your apps may either rely on external binaries or another library that you're building within the same project. Whatever the case, these dependencies have to be declared in the dependencies block of the Gradle file.

In the dependencies block, you must list all the direct dependencies of your app; if those dependencies in turn have their own dependencies, Gradle will also fetch them for you. There are three kinds of dependencies of note:

- **Module dependency**: If you created a module in your app—other than the default app module—and you want it to serve as a library for your project, you can declare it as a module dependency.
- **Jar dependency**: When you want to use external libraries in your app, like `jdom.jar`, you simply drop the jar file into the `libs` folder of the project and declare it as a jar dependency. Remember that you can't just use any jar file here; you can only use jars that are intended to be used as libraries. It's best to always read the docs of a library before using it.
- **Library dependency**: This kind of dependency will pull things from a repository like a Maven repo or `jcenter`. A repository is simply a collection of binaries that are available for your use. You already have a repository on your local machine when you installed Android Studio and then pulled some additional software during installation. The Android Support repository is already locally available to you—so this repo doesn't need to be declared in the Gradle file (the project-level Gradle file), but any repo other than this needs to be declared. You'll notice that when you create a new project in Android Studio, the generated Gradle file already includes a reference to `jcenter` and `google`, which are massive Java and Android repositories. Listing 8-2 shows an excerpt from the project-level Gradle file; notice the repositories section both in the buildscript and allprojects blocks—they already contain references to `jcenter` and `google`. Of course, if you need to reference any repo outside `jcenter` and `google`, you can simply add it to the Gradle file.

Listing 8-2. build.gradle (Project Level)

```
buildscript {
  repositories {
    google()
    jcenter()
  }
  dependencies {
    classpath 'com.android.tools.build:gradle:3.3.1'

    // NOTE: Do not place your application dependencies here; they belong
    // in the individual module build.gradle files
  }
}

allprojects {
  repositories {
    google()
    jcenter()
  }
}
...
```

Notice that there's a note in the project-level Gradle file to **NOT** place your application dependencies in that file, and to do it in the app module Gradle file instead. Anything you define in the project-level Gradle file affects all module-level Gradle files.

In addition to understanding the dependency types, you also need to understand how to associate dependencies. There are at least three directives to consider:

- `implementation`: It means that this dependency will be applied to your build variants or all the different kinds of builds that you do.
- `testImplementation`: This is a testing-specific variant. In here, you declare the things you need for the JVM testing. It's the kind of unit testing that only needs the JVM and not the full Android runtime.
- `androidTestImplementation`: In here you declare what libraries you need for the instrumented testing—the kind of testing that needs the full Android runtime.

Listing 8-3 shows an annotated excerpt of the module-level `build.gradle` file.

> **Note** A build variant is the combination of a build type and build config. Build type examples are *release* and *debug*. A build config was discussed in an earlier section of this chapter. You can see the current build variants by going to the main menu bar and choosing View ➤ Tool Windows ➤ Build Variant.

Listing 8-3. /app/build.gradle

```
apply plugin: 'com.android.application'

android {
  ...
}

dependencies {
  implementation fileTree(dir: 'libs', include: ['*.jar'])  ❶
  implementation 'com.android.support:appcompat-v7:28.0.0'  ❷
  implementation 'com.android.support.constraint:constraint-layout:1.1.3' ❸
  testImplementation 'junit:junit:4.12'  ❹
  androidTestImplementation 'com.android.support.test:runner:1.0.2' ❺
  androidTestImplementation 'com.android.support.test.espresso:espresso-
  core:3.0.2'
}
```

❶ This `implementation` directive with the `fileTree` command, a `dir` parameter, and the ANT style glob pattern `*.jar` means you want all jar files inside the `lib` folder of the project to become dependencies. So, including a jar dependency for the project is as easy as dropping a jar file into your project's `lib` folder.

❷ This directive means you're pulling a dependency from a repository—a local repository, to be exact. This pulls in the compatibility library from the Android Support Library. Use AppCompat so that the newer capabilities of Android can run on older versions. Don't worry too much about the versions; Android Studio does a decent job of pulling the right and most recent versions of the compatibility libs most of the time.

❸ Same as number 2, this is also a library dependency coming from the Android Support Library. This time, you're referencing the constraint layout library that you need to use when creating activities that use the Constraint layout.

❹ This `testImplementation` directive pulls the JUnit 4 library that you use for JVM unit testing.

❺ This `androidTestImplementation` pulls the libraries you'll use for instrumented testing.

Android Support Library

The topic of Android Support Library is a big one. I won't deal with all of it here, but you need a passing knowledge of it, at least in the context of our current discussion.

The Android Support Library is a very important part of Android. It basically supplements the Android SDK. One of the most important things it does is to provide backwards compatibility; at least it started out that way.

Backwards compatibility was important to support because newer versions of Android are being released at a rapid pace while quite a few devices are still running older versions of Android. Newer Android versions mean newer capabilities and features; does that mean devices running older versions of Android cannot use the newer features? No. Backwards compatibility means that the newer platform features can be made available to older platform versions.

As time went on, the Android Support Library got bigger. It started providing capabilities that were not part of the platform like in the areas of user interfaces, like Recycler Views or Card Views. The Support Library also included capabilities in the areas of debugging and testing, and so it needed to be reorganized.

In the early days of the Android Support Library, it was just a single library; but as it grew, its organization evolved. So, instead of having just a single library, the support library became a bunch of smaller and more manageable libraries. The groupings of these libraries were organized mainly according to platform support, which meant that the name of a support library indicated what platform or API level it supported.

You might still see libraries named v4, v7, and v13, which historically meant v4 supported API level 4 and up, v7 supported level 7 and up, and v13 supported API level 13 and up. I said "*historically*" because that's no longer true. There's a lot that goes on every time a newer Android version comes along; deprecations happen left and right. So you must make it a point to read the Support Library documentation from time to time.

At the time writing, v4 no longer supports level 4 and up. It now supports only API level 14 and higher; v7 as well no longer supports level 7 and up, but instead supports only level 14 and up. This effectively means that the minimum SDK version for all support library packages is now level 14 (Ice Cream Sandwich).

More importantly, with the release of Android 9 (API level 28), there is a new version of the Android Support Library called AndroidX, which is part of Jetpack. You can continue to use the support libraries (`android.support.*` versions 27 and earlier). They will remain on Google Maven, but note that all development will happen in AndroidX.

> **Note** Android Jetpack is a collection of components that makes it easier to develop apps. The libraries are in the `androidx.*` packages and they are unbundled from the platform APIs. This is where the backward compatibility libraries now reside. Jetpack is big. It's not just about compatibility libraries; it has lots of other software components that deal with architecture, foundation, UI, and behavior. You can read more about it at https://developer.android.com/jetpack.

In Listing 8-3, you saw the line

```
implementation 'com.android.support:appcompat-v7:28.0.0'
```

What this means is, even if the device running your app is at API level 14, you can still use

- An ActionBar
- Material design
- `AppCompatActivity` classes
- `AppCompatDialog`
- `ShareActionProvider`

All these capabilities are pretty modern, but you can still make them available on older platforms because of the AppCompat libraries.

> **Note** When you target the lower API levels, your app can rely on fewer modern Android features, but a larger percentage of Android devices are able to run your app. The opposite is true when you target higher level APIs. You can always look at the data or the cumulative distribution of the Android versions by clicking the "Help me choose" link when you create a new Android project.

Chapter Summary

- The build process can be very involved. Thankfully, Gradle takes care of all the heavy lifting.
- There are two `build.gradle` files: one in the project root and one for each module. The project-level Gradle file is a good place to define directives that you want to share to all module-level Gradle files.
- The module-level Gradle file is where you define most of your configurations. Module dependencies and target SDKs are defined in here.

Chapter 9

Git

What this chapter covers:

- How to get Git
- Setting up GitHub in Android Studio
- Sharing and cloning with GitHub
- Sharing and cloning from other Git repos

Android Studio provides integration with quite a few version control systems, and one of the more popular version control systems is Git. In this chapter, you'll look at how to use Git in Android Studio.

Getting Git

Git is available for macOS, Linux, and Windows. If you're on macOS and you've installed the XCode CLI, chances are you already have Git. On Windows and Linux, you'll have to get it somehow before you can proceed with the rest of these instructions.

The simplest way to get Git on your system is to head over to `https://git-scm.com/download` and get the version for your platform. You can get precompiled binaries for Windows and macOS; for Linux, you get instructions on how to use your package manager to get Git.

If you already use a package manager like chocolatey (Windows), MacPorts (macOS), or HomeBrew (also macOS), you can get Git using these package managers.

T. Hagos, *Android Studio IDE Quick Reference*,
https://doi.org/10.1007/978-1-4842-4953-6_9

On Windows with chocolatey, you can run the following on a command line or PowerShell:

```
choco install git
```

On macOS with HomeBrew, you can try

```
brew install git
```

On macOS with MacPorts, you can use

```
sudo port install git
```

Next, let's move on to Android Studio. Open the IDE, go to Preferences (macOS) or the Settings window (Windows and Linux), and then go to Version Control ➤ Git, as shown in Figure 9-1.

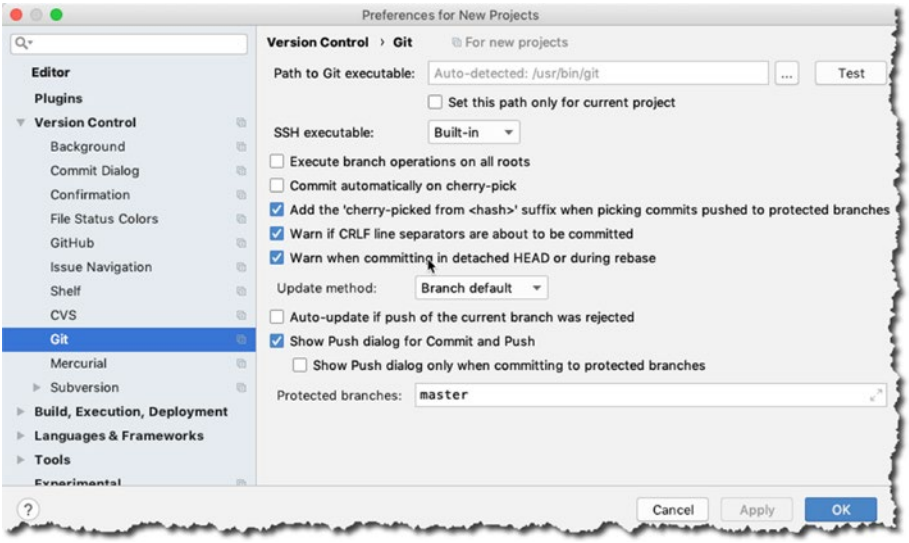

Figure 9-1. *Preferences ➤ Version Control ➤ Git*

Notice the Path to Git executable field. As you can see from Figure 9-1, mine was filled up and auto-detected; this is usually right, and you don't have to do anything else. If Android Studio did not detect the location of the Git executable automatically or you don't like to use the auto-detected version, you can always change it. To change the Git executable, click the ellipsis (it's the button with three dots on the left side of the Test button, shown also in Figure 9-1).

Locate the Git executable that you'd like to use for Android Studio. After that, click the Test button. See the results in Figure 9-2.

> **Tip** for Windows users. If you used the installer from git-scm and installed it using the default options, the Git executable will be in the folder `C:\Program Files\(x86)\Git\bin\`.

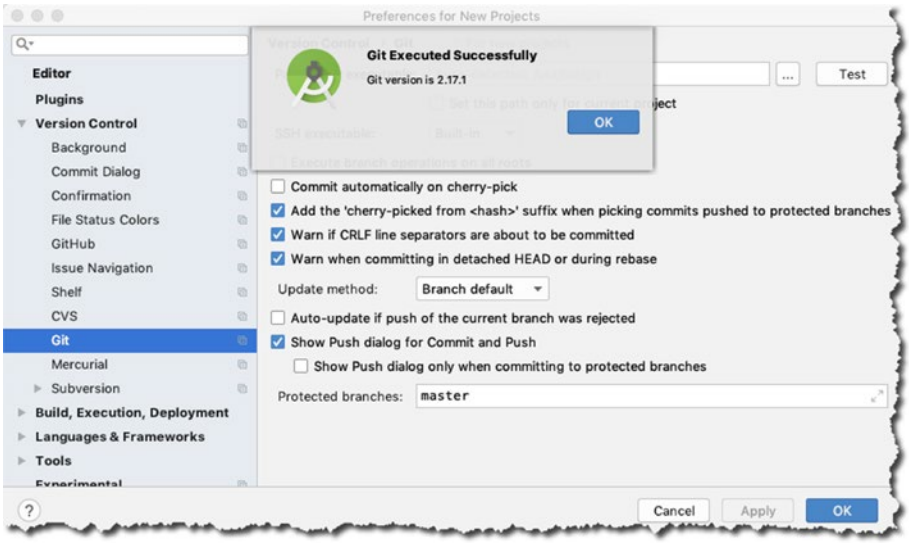

Figure 9-2. *Git executed successfully*

Now you're ready to use Android Studio with Git version control. You can now integrate with Git repos.

Using Android Studio with GitHub

Go to the Preferences/Settings window again and then go to GitHub, as shown in Figure 9-3.

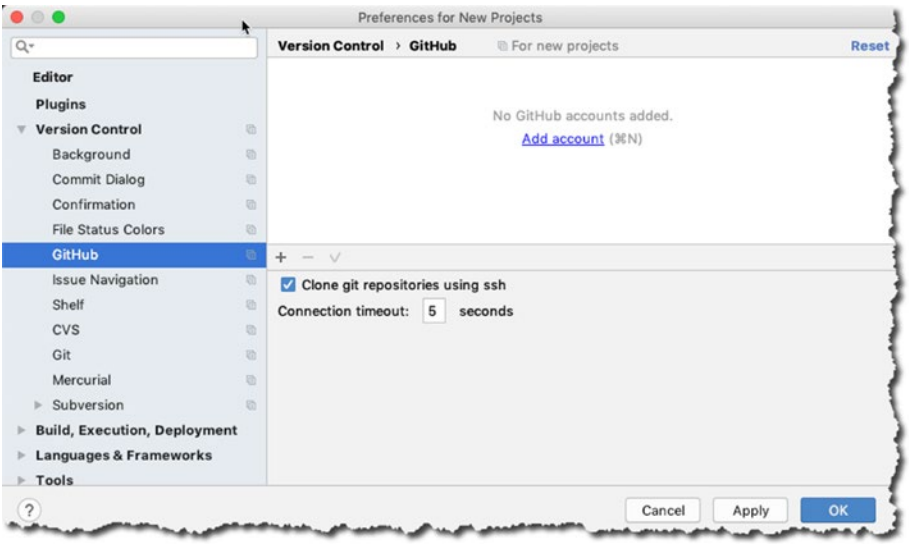

Figure 9-3. Preferences, GitHub

Click the Add account link. You'll see the Login window shown in Figure 9-4.

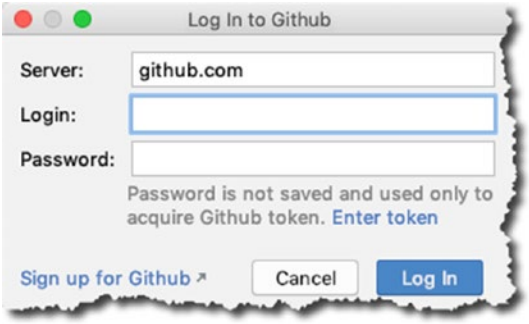

Figure 9-4. Log in to GitHub

If you already have a GitHub account, you can provide your GitHub login credentials in this window; if you don't have an account yet, you can click the Sign up for GitHub link (shown in Figure 9-4) and you'll be taken to the GitHub web page where you can sign up for a new account.

You can create either private or public repos with your free GitHub account. A public repo is shared with everyone; you don't have control over its visibility—everybody sees it. A private repo, on the other hand, isn't shared with everyone; you can control the admission to the repo.

When you have your GitHub credentials ready, put them in the login and password fields shown in Figure 9-4 and click OK. If everything goes well, you should see your GitHub avatar in the Preferences/Settings window, as shown in Figure 9-5.

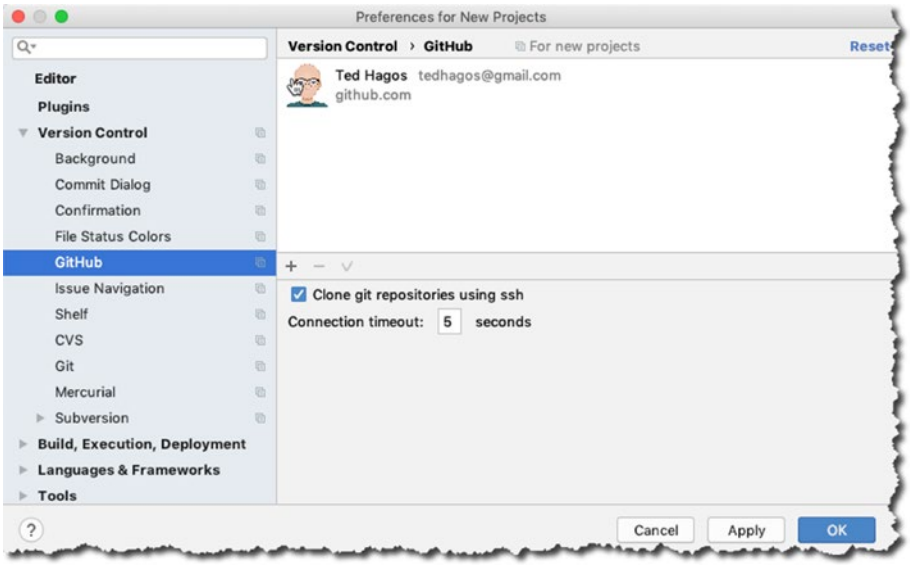

Figure 9-5. GitHub settings

Click the OK button, and you should be ready pull and share projects in GitHub.

Sharing a Project on GitHub

If this is the first time you've installed Git, you might have to configure the global variables `user.email` and `user.name` for Git. To do this, type the code in Listing 9-1 from a command line window.

Listing 9-1. Set Git Global Variables

```
Git config --global user.name "your name"
Git config --global user.email "your email"
```

Don't forget to substitute your actual name and actual email for "your name" and "your email," respectively.

> **Tip** for Windows users. The Git installer won't update the PATH variable, so you might encounter the "bad command or filename" error. In order to do the commands shown in Listing 9-1, you must navigate first to the location of the Git executable. So, change directory to `C:\Program Files\(x86)\Git\bin\` and then execute the commands shown in Listing 9-1.

Open any project that you'd like to share via GitHub; then, from the main menu bar, go to VCS ➤ Import into Version Control ➤ Share Project on GitHub, as shown in Figure 9-6.

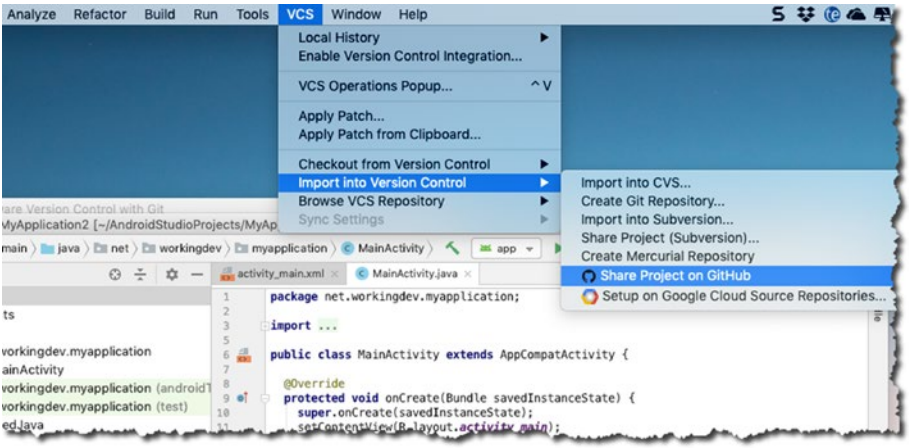

Figure 9-6. *Importing into version control*

Provide the necessary information, as shown in Figure 9-7. The following is a brief explanation of what the fields mean:

- **Repository name**: The name of the public repository you'd like to create. When you share a project on GitHub, it is called a repository (or repo for short), which is simply a place where data is stored and managed. In this case, you're creating a repo for just the one project. So, it's best that you store only project-related artifacts in it.
- **Private**: If you leave this box unchecked, as I'm doing here, it means the repo will be public. You need to decide on the visibility of the repo. Remember, if it's public, everybody sees it; if it's private, you control who sees it.

- **Remote**: Leave this as is. It reads as "remote" because the files will be stored on a remote server (on GitHub's servers), as opposed to "local," which means the files will be stored on your computer.
- **Description**: It's best to provide some words that describe the project here.

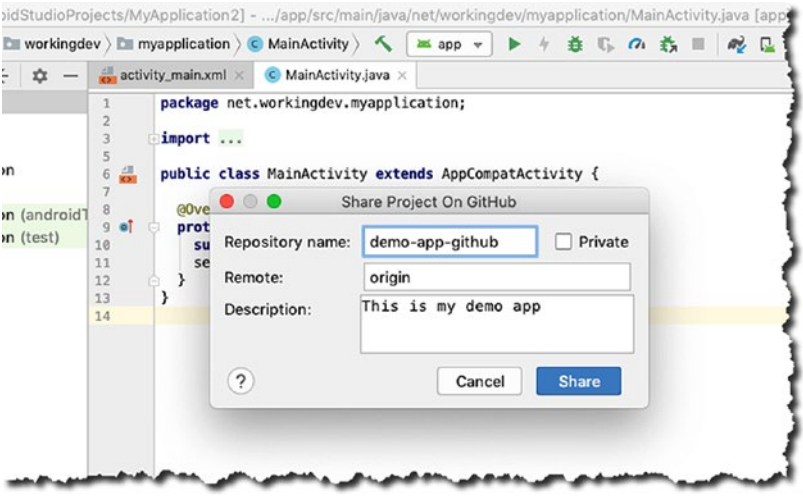

Figure 9-7. Share a project on GitHub

When you click the Share button, Android Studio will prompt you for which files you want to share. In version control parlance, instead of saying "share" we say "commit." See Figure 9-8.

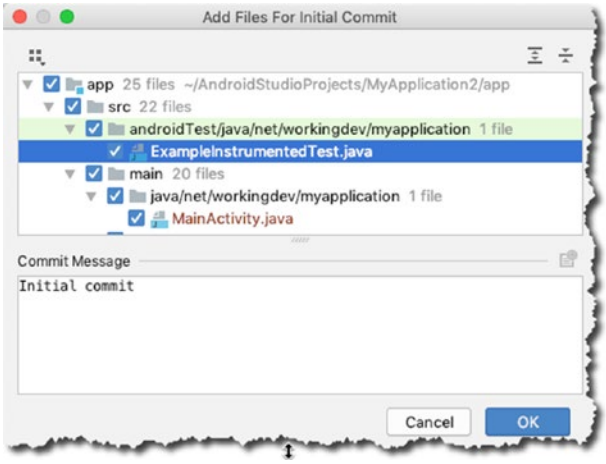

Figure 9-8. Adding files for the initial commit

By default, everything shown in the dialog box is checked, which means everything in the project folder will be committed, but not quite. Try to scroll down so you can see all the files that have checkmarks. You might notice a file named `.gitignore`, as shown in Figure 9-9.

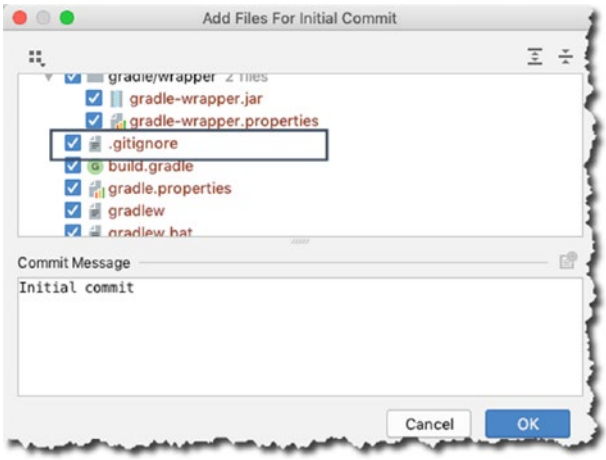

Figure 9-9. Adding .gitignore

`.gitignore` is a special file that contains a list of files that won't be included in the repo—these files will be, well, ignored.

If the view in the Project tool window is set to Android, you won't be able to see the `.gitignore` file, but if you change the view of the Project tool window to Project, as shown in Figure 9-10, you should see the `.gitignore` file.

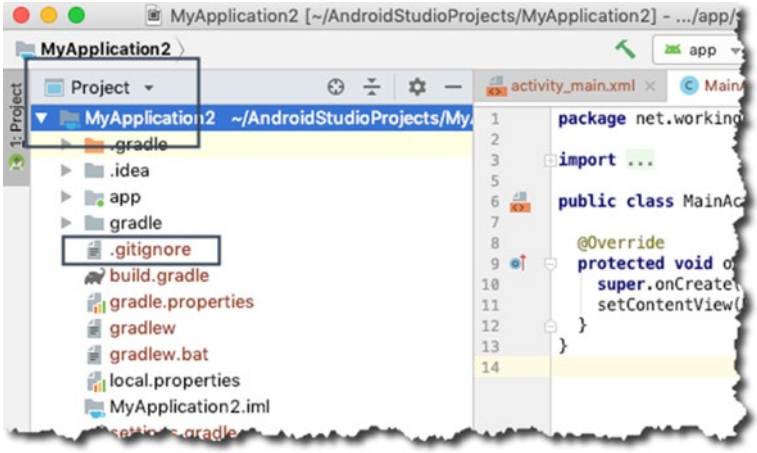

Figure 9-10. Project tool window set to Project view

Listing 9-2 shows the contents of `.gitignore`.

Listing 9-2. .gitignore

```
*.iml
.gradle
/local.properties
/.idea/caches
/.idea/libraries
/.idea/modules.xml
/.idea/workspace.xml
/.idea/navEditor.xml
/.idea/assetWizardSettings.xml
.DS_Store
/build
/captures
.externalNativeBuild
```

You want to keep all the entries in `.gitignore` most of the time. Of course you can add to the list, but you don't want to remove anything from it, as shown in Listing 9-2. By default, the list of files in `.gitignore` refers to things that are unique to your computer; for example, the `/local.properties` lists the location of the JDK on your computer. It may also contain data that's specific to you as a user, like Firebase keys; clearly, you don't want to commit that info into source control. Another example is the `.DS_Store` file, which you will only have if you're using a Mac; macOS creates this file every time you create a directory. This file isn't important to your project.

Going back to the commit dialog window, when you click OK, all the (checked) files in the commit window get *committed* and then *pushed* to the remote repo.

Opening a Project from GitHub

You can restore a project by fetching it from a remote repo like GitHub; in Git parlance, this is called "cloning" a repo. The basic scenario when you do this is when one of your teammates has shared a project repo on GitHub and you'd like to work on it as well. The first step is to fetch the project from the remote repo. You can do this in one of two ways.

When you already have an open project in Android Studio, you can go to the main menu bar and choose File ➤ New ➤ Project from Version Control ➤ Git.

You may also do this from the Android Studio welcome screen, as shown in Figure 9-11.

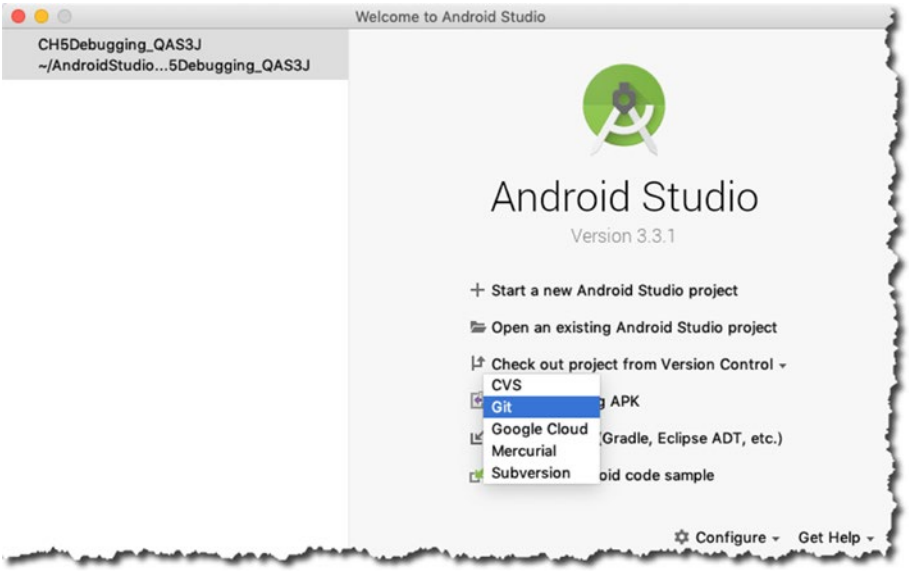

Figure 9-11. Welcome Screen, Git

Either way you do it is fine. The next screen is the Clone Repository window, as shown in Figure 9-12.

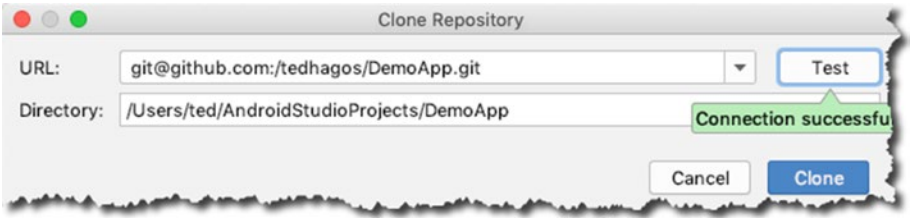

Figure 9-12. Clone repository

The fields are described below:

- **URL**: This is the remote URL of the repo. I'm using the URL git@github.com:/etc because I'm fetching the project via SSH. I can do this because I've already set up my SSH keys in GitHub. If you haven't set up your SSH keys yet, you may have to use the HTTPS URL. You can find the HTTPS URL for the repo on its GitHub page, shown in Figure 9-13. Click the Use HTTPS option to get the HTTPS URL.

- **Directory**: This is the location on your hard drive where you would like to store the project.

> **Note** GitHub allows for both SSH and HTTPS to be used when you interact with the repo. GitHub published some guidelines as to when to use SSH or HTTPS at `https://bit.ly/sshvshttps`. If you want to know more about the Git protocols, you can get more information from `https://bit.ly/gitprotocols`.

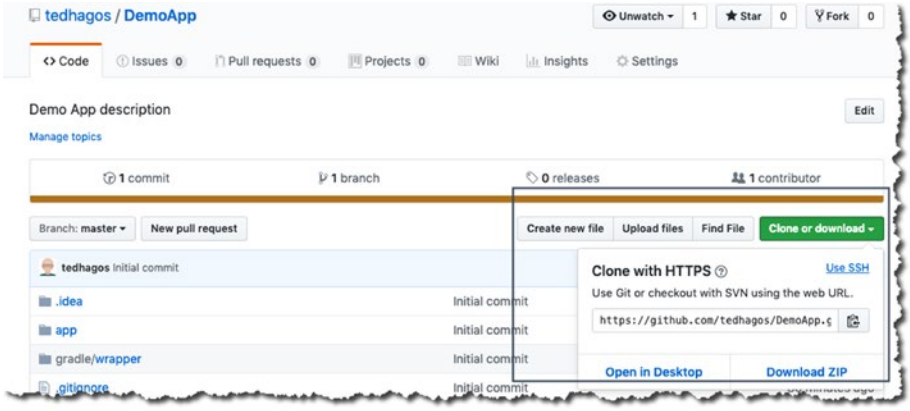

Figure 9-13. GitHub page for a repo

To complete the process of cloning, click the Clone button, as shown in Figure 9-12. You can now open the project as you would any other project in Android Studio.

Updating Git Projects

When a project is under source control management, Android Studio will give you additional prompts every time you create a new file, whether it's a Java class, XML file, picture, or other file. If you add, say, a new Java class into a project (under Git source control), you'll see a dialog window like the one shown in Figure 9-14.

Figure 9-14. Adding a file to Git

If you want to add the file to Git, of course press Yes. If you accidentally press No, you can still add the new file in two ways:

1. Select the file you'd like to add to Git in the Project tool window and then from the main menu bar choose VCS ➤ Git ➤ Add.
2. You can also use the context menu by right-clicking the file and choosing Git ➤ Add.

Adding the file doesn't automatically update the remote repo. You must do two things to achieve this. You need to *commit* the changes you made and then do a *push*. To do so, select the whole project in the Project tool window, as shown in Figure 9-15, and from the main menu bar, choose VCS ➤ Git ➤ Commit Directory.

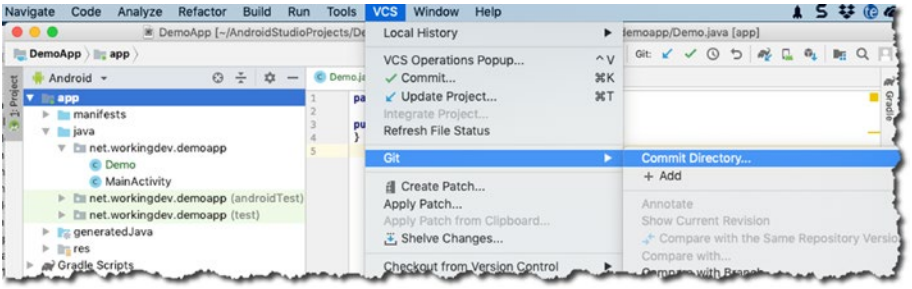

Figure 9-15. Git ➤ Commit directory

Next, you'll see the Commit Changes window, shown in Figure 9-16.

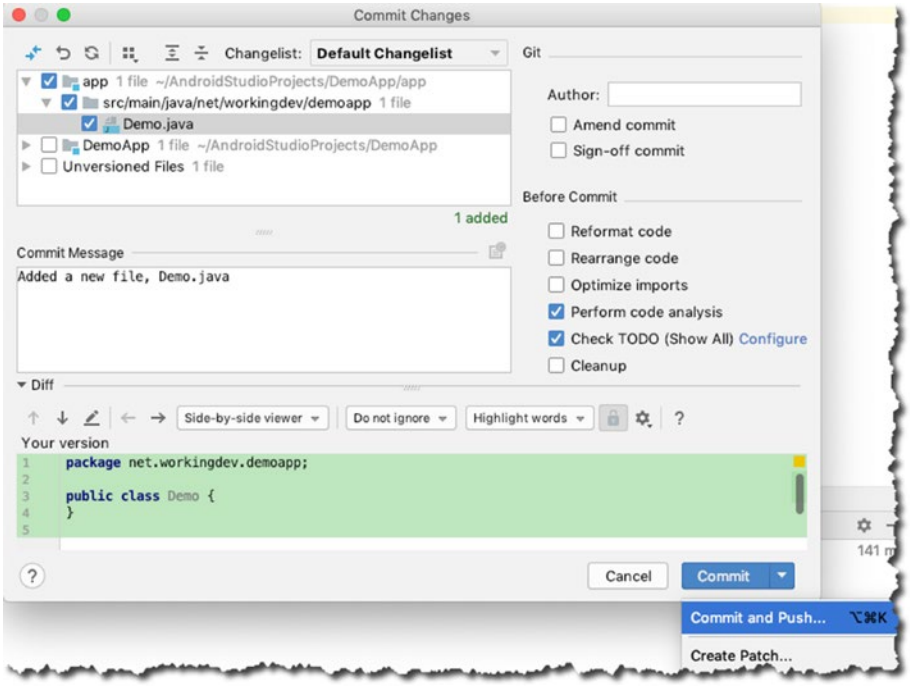

Figure 9-16. Committing and pushing changes

Click the down arrow on the Commit button and choose the Commit and Push option. Now your local project is completely in sync with the remote repo.

Using Other Git Repos

GitHub is a very popular repo platform among developers, so no wonder that it's well integrated with Android Studio, but what if your teammates or company don't use GitHub. What if they use BitBucket or some other Git repo? Well, you can handle that too. In this section, you'll use a BitBucket repo.

With BitBucket, just like in GitHub, you need an account. Head over to `https://bitbucket.org` to sign up, if you don't have an account yet, and then log into that account. After that, create a new repo, as shown in Figure 9-17.

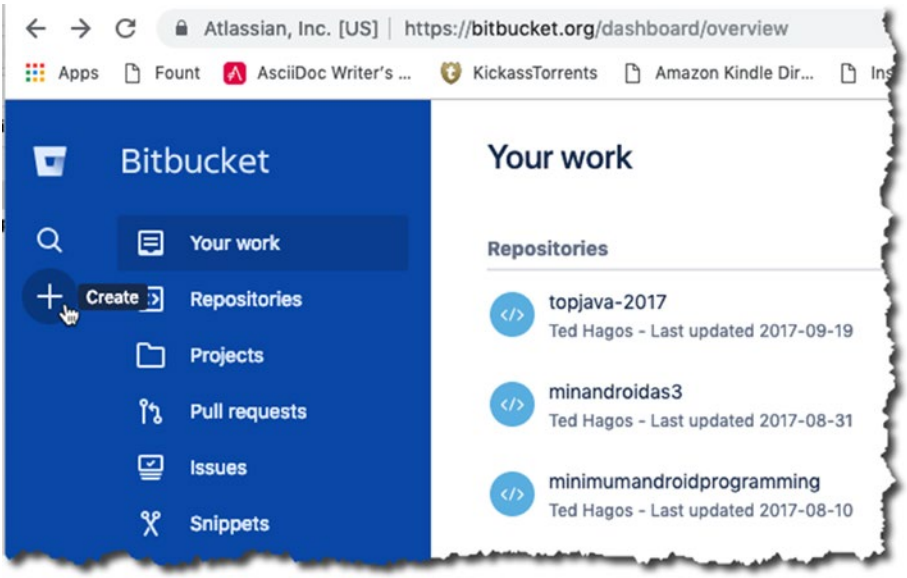

Figure 9-17. *Creating a repo*

When you click the big plus sign and choose to create a new repo, you'll be asked for some details, as shown in Figure 9-18. Don't forget to click the Advanced Settings option so you can see all the fields.

Take note of the following;

- **Repository name**: The name of your repo on BitBucket. It doesn't have to be the same as the project stored locally in your computer, but it's a good idea if it is.
- **Access level**: Unlike GitHub, you'll be able to create private repos in BitBucket, even if you're not a paying member.
- **Include a README**: It's good practice to have a README file in a project, but this is optional.
- **Version control**: Choose Git.
- **Description**: Some words to best describe what you're doing.
- **Language**: You're working on an Android project, so put Android.

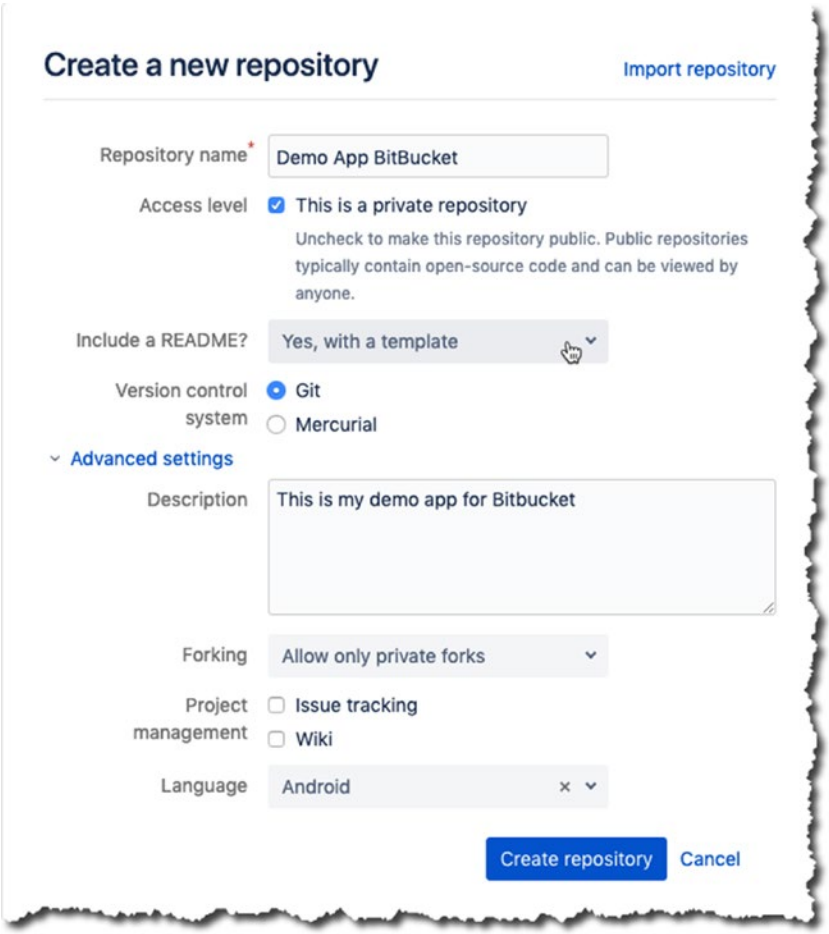

Figure 9-18. *Repo details*

After you click the Create repository button, you'll see the repo page detail, as shown in Figure 9-19. Click the Clone button.

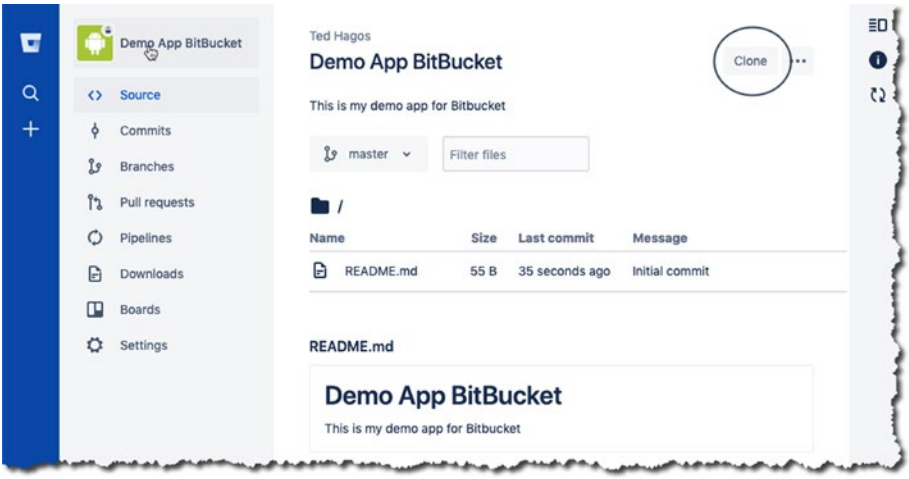

Figure 9-19. Newly created BitBucket repo

You might see SSH instead of HTTPS on the Clone this repository page. Click the dropdown button to switch over to HTTPS, as shown in Figure 9-20.

> **Note** You can use the SSH protocol as a means of cloning repos in BitBucket after you've provided your SSH keys to BitBucket. If you haven't done that yet, just use HTTPS to clone the repo.

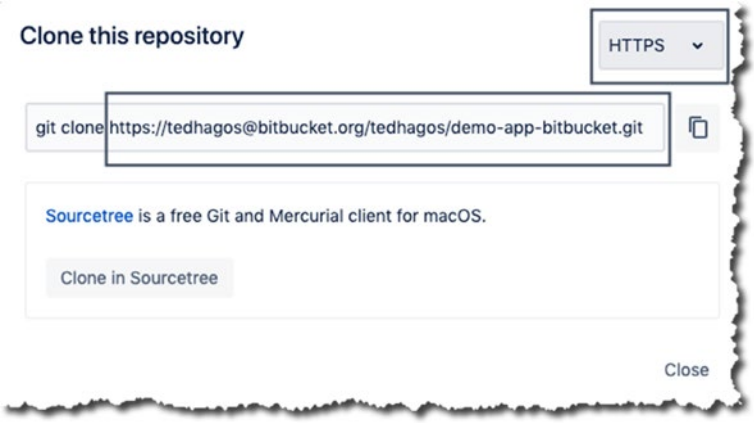

Figure 9-20. Remote address of repo

The remote URL for your repo is the one that starts with `https://` and ends with `.git`. As pointed out in Figure 9-20; you don't need the `git clone` part because that's a command that is intended to be typed from a command line. You won't be using the command line for Git, instead, you will use Android Studio's facilities.

Now that the remote repo is ready, you can either open an existing project that you'd like to share on BitBucket or create a new one; it's up to you.

Once you've chosen a project to share in BitBucket, do the following;

1. Open the project if it isn't open yet.
2. Enable version control on it, if it's not yet under any source control. To do that, go to the main menu bar and choose VCS ➤ Enable Version Control Integration.
3. Select Git for version control, as shown in Figure 9-21.

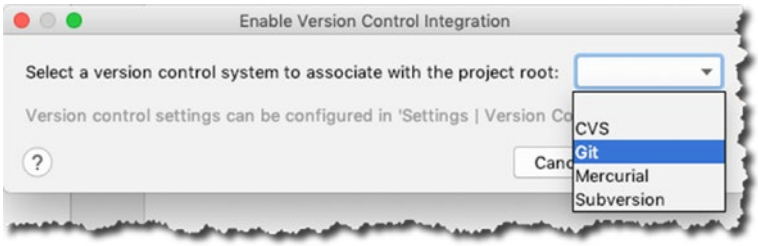

Figure 9-21. *Enabling version control integration*

The next thing to do is to associate this project with the remote repo; go to the main menu bar and choose VCS ➤ Git ➤ Remotes. You should see the Git Remotes screen, as shown in Figure 9-22.

Figure 9-22. *Git remotes*

Click the plus sign shown in Figure 9-22. After that, the Define Remote screen appears, as shown in Figure 9-23.

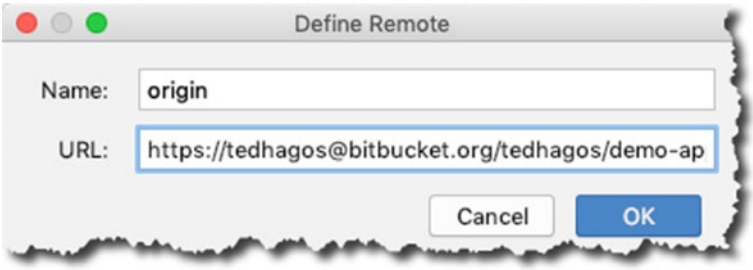

Figure 9-23. *Defining the remote*

Copy the repo's remote URL (refer to Figure 9-20; I highlighted the remote URL repo in that figure). Paste it in the URL field of the Define Remote screen, as shown in Figure 9-23. You need to give your BitBucket username and password in the next screen, as shown in Figure 9-24.

Figure 9-24. *Git login*

The next thing to do is to *pull* from the remote repo. This is a prudent thing to do especially if you generated README files while creating the repo in BitBucket because this prevents merge conflicts. You can do a pull from the main menu bar; go to VCS ➤ Git ➤ Pull. You'll see the Pull Changes screen, as shown in Figure 9-25. Click the refresh button (to the right of the Remote text field).

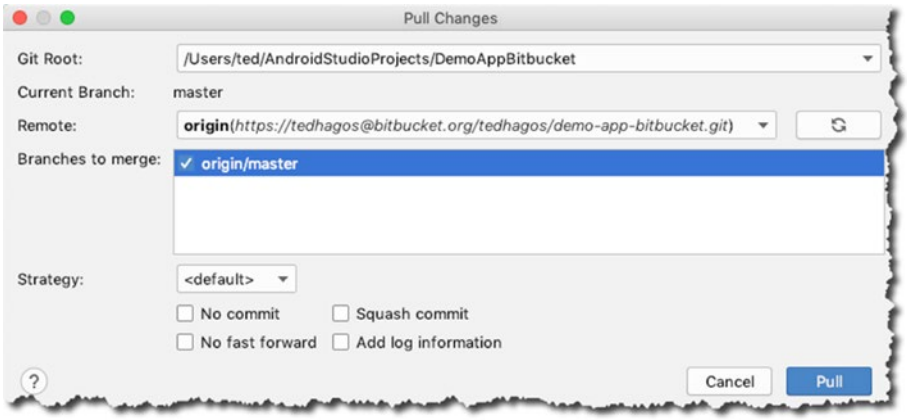

Figure 9-25. Pull changes

In the Branches to merge field, choose the origin/master option and then click OK.

Try to add some files to your new project. You'll notice that when you add a new file to the project, say `DemoApp.java`, Android Studio prompts you if you want to add the new file to Git; see Figure 9-26.

Figure 9-26. Adding a file to Git

You can save some time by checking the Remember, don't ask again box. The next time you add a new file, Android Studio will automatically add it to Git.

If you answered No to the prompt, you can still add files to Git by using the context menu on the new Java file. Select the new Java file from the Project Tool window, right-click, and go to Git ➤ Add. To be sure that all project files are added to Git, instead of selecting just the new file in the Project tool window, select the whole project instead.

The next step is to commit your changes to Git—remember all these changes are still happening locally. For the changes to reflect in the remote BitBucket repo, you must push the changes.

To commit the changes, go to the main menu bar and choose VCS ➤ Git ➤ Commit Directory. You'll see the Commit Changes dialog, as shown in Figure 9-27. You saw this in the previous section; it's the same as the one when you were using GitHub.

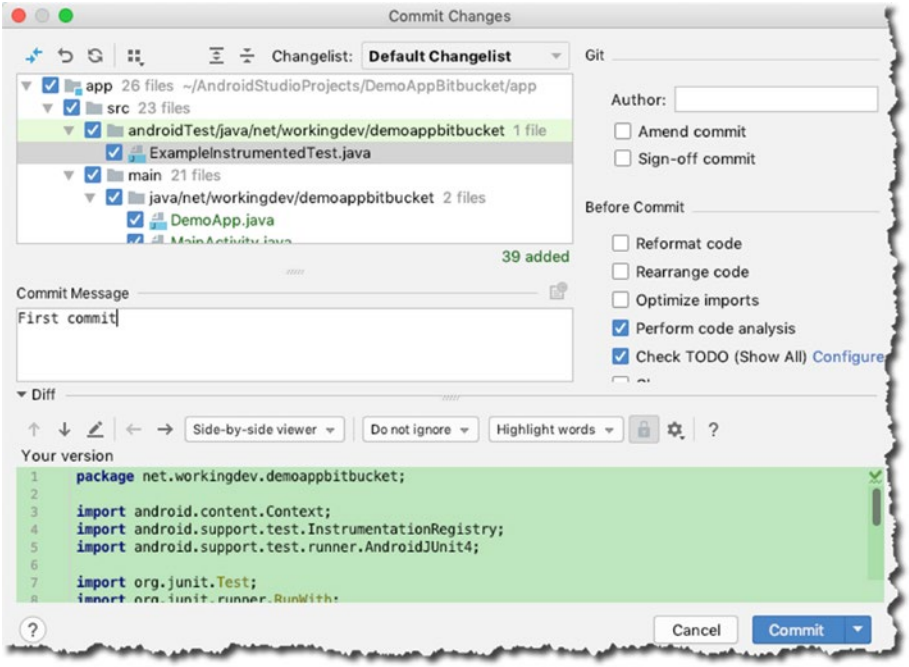

Figure 9-27. Committing changes

Remember that only checked items will be committed to Git. So, pay attention to the change list. Write something descriptive in the Commit Message box and then click the Commit button.

When you've made significant changes to the local repo and you want to reflect the changes back to the remote repo, you can do a *push*. To do that, go to the main menu bar and choose VCS ➤ Git ➤ Push. You'll see the Push Commits screen, as shown in Figure 9-28.

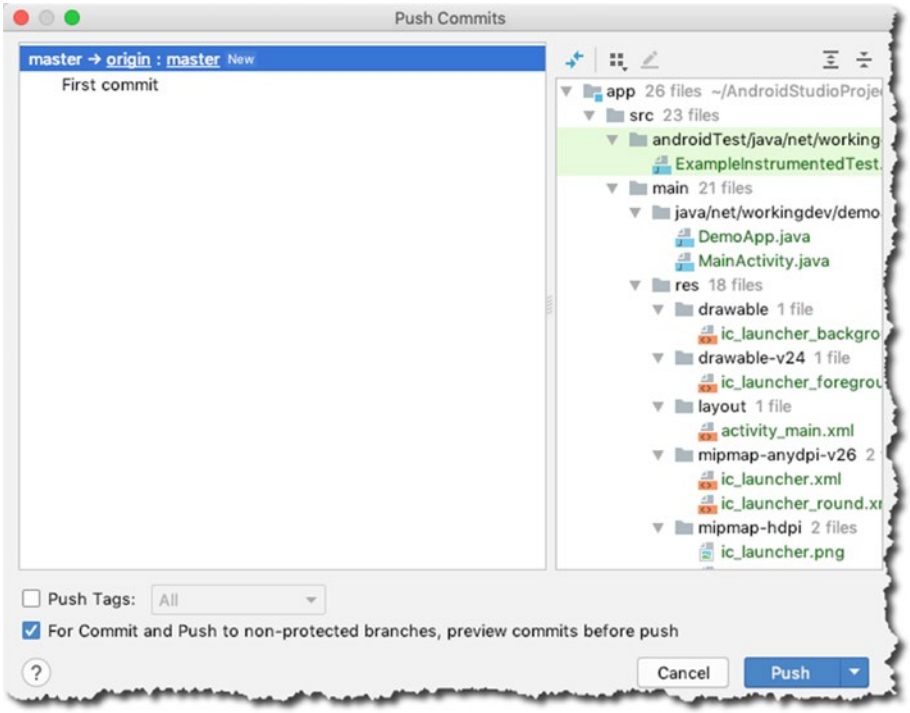

Figure 9-28. Pushing commits

Click Push to complete the operation.

That should do it. Now you can clone a repo and restore it on your local directory. You can also add, commit, and push changes to a remote repo; you also now know how to get a local repo in sync with the remote one.

Chapter Summary

- The Git software isn't included in Android Studio, so you must get it separately. You can get precompiled binaries and platform instructions from `https://git-scm.com/download`.
- GitHub is very integrated into Android Studio. There's only a few steps that you need to do in order to work with a remote GitHub repo.
- You can work with other Git platforms like BitBucket by enabling the version control integration in a project and then defining the remote repo URL.

Chapter 10

Navigation

What this chapter covers:

- A review of navigation in Android
- Navigation components

Navigation Before Architecture Components

In the early days of Android, if you had a non-trivial app, you almost certainly needed to partition your app across multiple activities. That meant you needed the skill to navigate from one activity to another, and back. So, in those days, you might have written something that looked like the code in Listing 10-1.

Listing 10-1. How to Launch an Activity

```
class FirstActivity extends AppCompatActivity
  implements View.OnClickListener {

  public void onClick(View v) {
    Intent intent = new Intent(this, SecondActivity.class);
    startActivity(intent);
  }
}

// SecondActivity.java
class SecondActivity extends AppCompatActivity { }
```

And if you needed to pass data from one activity to another, you might have coded it like the code snippet shown in Listing 10-2.

T. Hagos, *Android Studio IDE Quick Reference*,
https://doi.org/10.1007/978-1-4842-4953-6_10

Listing 10-2. How to Pass Data to Another Activity

```
Intent intent = new Intent(this, SecondActivity.class);
Intent.putExtra("key", value);
startActivity(intent);
```

This kind of screen management has the following advantages;

- It's simple to do. Just call the `startActivity()` method from any activity.
- The currently running activity can be closed programmatically by calling the `finish()` method. The user can also close the activity by pressing the back button.
- The back stack is completely managed by the Android runtime; see Figure 10-1.

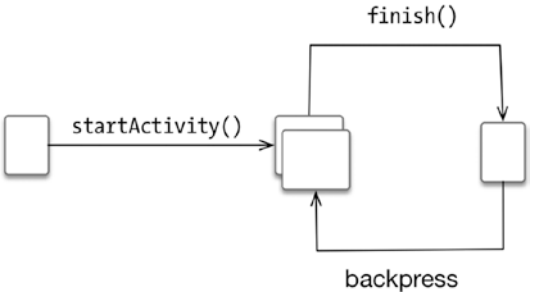

***Figure 10-1.** Simple activity workflow*

But it's not all good; activity navigation comes with some baggage. The disadvantages are

- You don't have a clear idea which activities are on the back stack because you don't manage it. Here is an example of a trait being both an advantage and disadvantage at the same time.
- Each screen requires a new activity, which can be quite heavy on resources.
- Each activity needs to be declared on the Android manifest file, which Android Studio does for you automatically each time you create an activity using the wizards, so it's not much of an issue.
- It will be difficult for you to use the more modern navigation patterns like the bottom navigation bar.

Because of these limitations, another way of screen navigation emerged. In 2011, when Google released Android 4.0, we got to use *fragments*. If you've forgotten about fragments, think of them this way: an activity is basically a composition unit for the UI, and the fragment is an even smaller composition unit.

A fragment, like an activity, is comprised of two parts: a Java or Kotlin program and a layout file. The idea is basically the same: define the UI in an XML file and then inflate the XML file during runtime so that all the UI in the XML file becomes actual Java objects.

The idea is to create multiple fragments and contain them in a single activity. You generally hide or show a fragment depending on either a user action, the orientation of the device, or the form-factor of the device; and this is usually done with the `FragmentManager` and `FragmentTransaction` objects. If you've worked with fragments before, the code snippet shown in Listing 10-3 might be familiar.

Listing 10-3. Fragment Snippet

```
FragmentManager fm = getFragmentManager();
FragmentTransaction ft = fm.beginTransaction();
Fragment fragment = new FirstFragment();
ft.add(R.id.fragment_container, fragment);
ft.commit();
```

When using fragments, you know exactly what's in the navigation stack, unlike when you use activity navigation; however, as you can see in Listing 10-1, it can get cumbersome because you must manually manage the navigation stack.

Thus far, you've only had two options for navigation: either you use activity-based navigation, which is easy and simple to use but you take a performance penalty and you don't have control over the navigation stack, or you use fragments, which offer you full control of the navigation stack but the API is cumbersome and prone to error.

Fast forward to 2017 when Google introduced the navigation components. Now you can use fragments but without the baggage of the complicated API. With navigation components, all the code in Listing 10-3 can now be replaced with a single line of code, shown in Listing 10-4.

Listing 10-4. Navigation Component Snippet

```
findNavController().navigate(destination);
// FragmentManager fm = getFragmentManager();
// FragmentTransaction ft = fm.beginTransaction();
// Fragment fragment = new FirstFragment();
// ft.add(R.id.fragment_container, fragment);
// ft.commit();
```

Navigation Components

Alright, that single line of code reference from the previous section probably got you excited—and relieved. But it's not the savings of keystrokes that's the big picture here; it's the fact that now you can get the best of both activity-based and fragment-based navigation. Now, fragment navigation also has an easy API.

But first you need to understand a bit about navigation components. They're a small part of the architecture components, which are in turn a part of a bigger thing called Android Jetpack. (I'm not getting into Jetpack or architecture components in detail here; they are large topics; but a brief background can't hurt.)

At Google I/O 2017, Google introduced the Android architecture components. These libraries are part of a larger collection called Android Jetpack. Together with architecture components, there were others like foundation, behavior, and UI.

Jetpack is a collection of Android software components to make our lives easier. It helps us follow best practices and lets us avoid writing too much boiler-plate code. You'll find the Jetpack code in the `androidx.*` package libraries.

Here's a brief description of the Jetpack components.

Foundation

- **AppCompat**: Lets you write code that degrades gracefully on older versions of Android
- **Android KTX**: So you write more concise, idiomatic Kotlin code if you're using Kotlin
- **Multidex**: Provides support for apps with multiple DEX files
- **Test**: A testing framework for unit and runtime UI tests

Behavior

- **Download manager**: Lets you write programs that schedule and manage large downloads
- **Media and Playback**: Backwards-compatible APIs for media playback and routing
- **Notifications**: Provides a backwards-compatible notification API with support for Wear and Auto
- **Permissions**: Compatibility APIs for checking and requesting app permissions
- **Preferences**: Creates an interactive settings screen
- **Sharing**: Provides a share action suitable for an app's Action bar
- **Slices**: Creates flexible UI elements that can display app data outside the app

UI

- **Animations and Transitions**: Moves widgets and transitions between screens
- **Auto**: If you're working on apps that will run in the infotainment consoles in vehicles, you'll need this. These are the components that help you build apps for Android Auto.
- **Emoji**: Enables an up-to-date emoji font on older platforms
- **Fragment**: All the fragment code is already moved here.
- **Layout**: Lays out widgets using different algorithms
- **Palette**: Pulls useful information out of color palettes
- **TV**: Components to help develop apps for Android TV
- **Wear OS by Google**: If you want to work with Android wearables like the watch, this is what you need.

Architecture

- **Data binding**: Declaratively bind observable data to UI elements
- **Lifecycles**: Manage activity and fragment lifecycles

- **LiveData**: Notifies views when underlying database changes
- **Paging**: Gradually loads information on demand from your data source. Think of when the user is scrolling through a list—this helps you handle the loading of data. It's coupled with the Recycler view.
- **ViewModel**: Manages UI-related data in a lifecycle-conscious way
- **WorkManager**: Manages background jobs
- **Navigation**: Implementation of navigation in an app. Passes data between screens. Provides deep links from outside the app.
- **Room**: Think ORM for your SQLite database

There's a lot to explore in Jetpack, so make sure you check it out.

Going back to our topic, the Navigation components simplify the implementation of, well, navigation between destinations in an app. A *destination* is any place in your app. It could be an activity, a fragment inside an activity, or a custom view; and destinations are managed using a navigation graph.

A navigation graph groups all the destinations and defines the different connections between the destinations; these connections are called *actions.* The graph is simply an XML resource file that represents all your app's navigation paths. You can have more than one navigation graph in your app.

Working with Jetpack Navigation

To get a better appreciation of the navigation components, it's best if you can work on a small project. So, create a new, empty project in Android Studio, as shown in Figure 10-2. Don't forget to check the Use AndroidX artifacts option. Remember that all support libraries are now in the AndroidX packages; you need them because you'll be working with fragments.

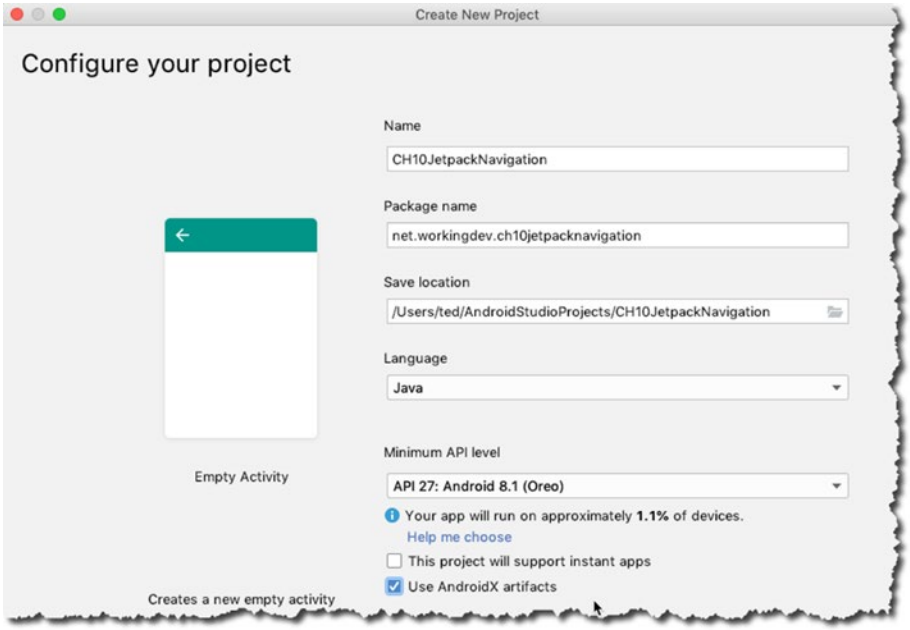

Figure 10-2. New, empty project with AndroidX artifacts

By the way, if you haven't updated or upgraded to the latest version of Android Studio, now would be a good time to do so. At the time of writing, you must use Android Studio v3.3 or higher if you want to use navigation; also, you need to add the navigation component's dependencies to your project. So, after you've created the project, locate the module-level `build.gradle` file and add the entries, as shown in Listing 10-5.

Listing 10-5. Adding Navigation to build.gradle

```
dependencies {
    def nav_version = "2.1.0-alpha02"
    ...
    implementation "androidx.navigation:navigation-fragment:$nav_version"
    implementation "androidx.navigation:navigation-ui:$nav_version"
}
```

You'll have to sync the Gradle file after adding the dependencies.

> **Note** Remember that there are two Gradle files in your project. You need to edit the module-level Gradle file, as shown in Figure 10-3.

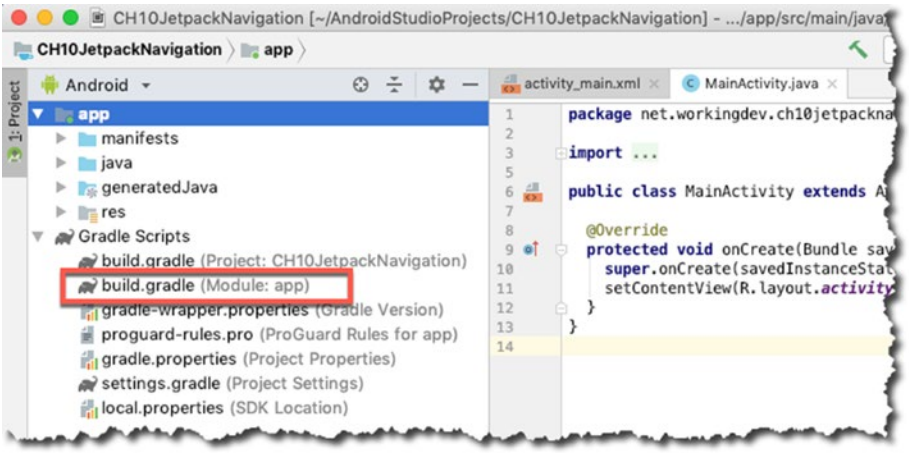

Figure 10-3. build.gradle (module)

When the sync is done, add a navigation graph to the project. You can create a navigation graph by creating a new resource file; right-click the project's `res` folder and select New ➤ Resource File, as shown in Figure 10-4.

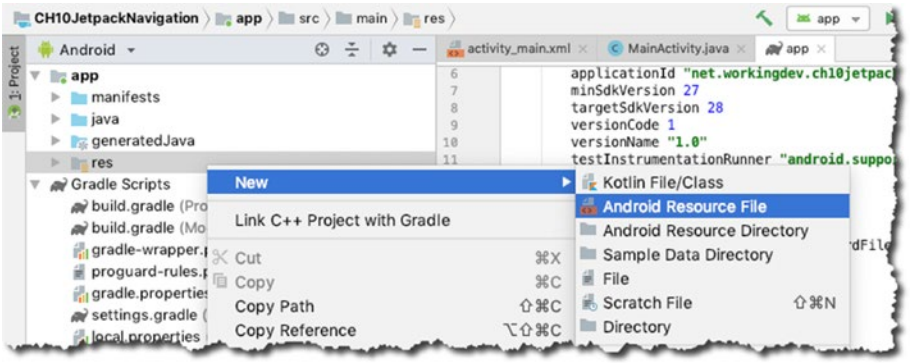

Figure 10-4. Adding a new resource file

In the New Resource File dialog, change the resource type to Navigation and supply the file name:

- **File name**: nav_graph
- **Resource type**: Navigation (you must click the down-arrow to select it)

Figure 10-5 shows the New Resource File dialog.

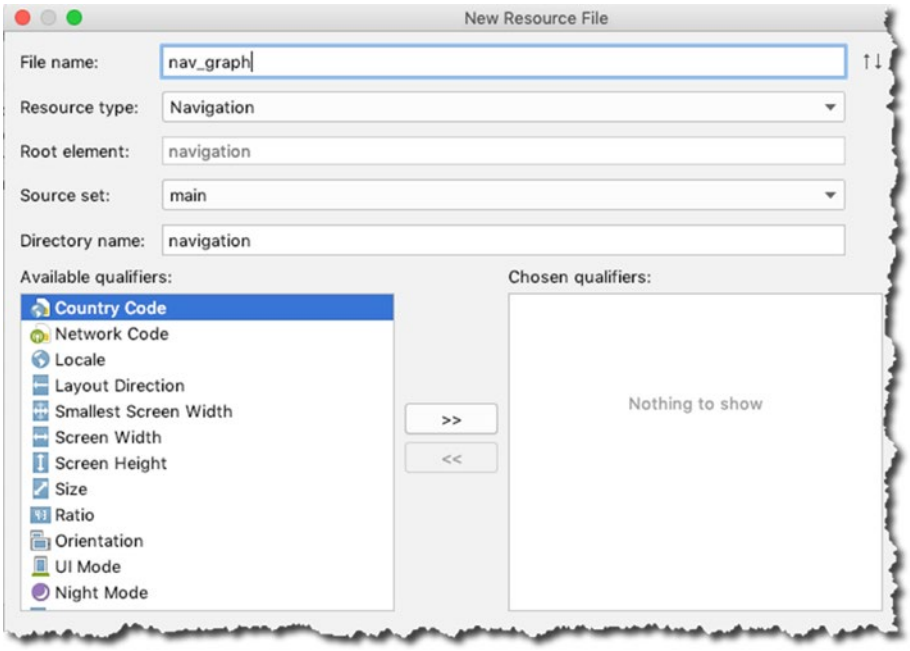

Figure 10-5. *The new resource file info*

Click OK to create the new resource file.

When the resource is created, you'll see a new folder (`navigation`) and a new file (`nav_graph.xml`) under the `res` folder of the project, as shown in Figure 10-6. Android Studio will open the newly created navigation graph in the editor. Figure 10-6 shows the newly created navigation graph; it's empty, of course.

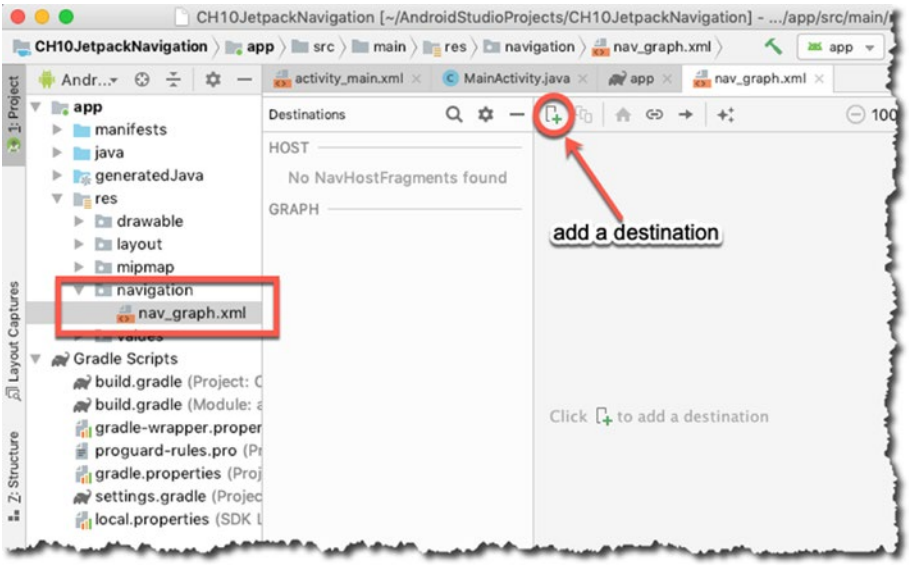

Figure 10-6. Navigation graph

When you're using navigation components, navigation happens as an interaction between destinations. Destinations are what your users can navigate to, and destinations are connected via actions. At the moment, you don't have any destination yet, so let's add one. Click the plus sign on the top panel of the navigation editor, as shown in Figure 10-7, and then choose the Create a New Destination option.

This will show the dialog for creating a new fragment. I changed the name of the new fragment and left the rest of the fields alone, as shown in Figure 10-7. The details of this new fragment are as follows:

- **Fragment Name**: One
- **Fragment Layout Name**: fragment_one
- Keep the source language as **Java**

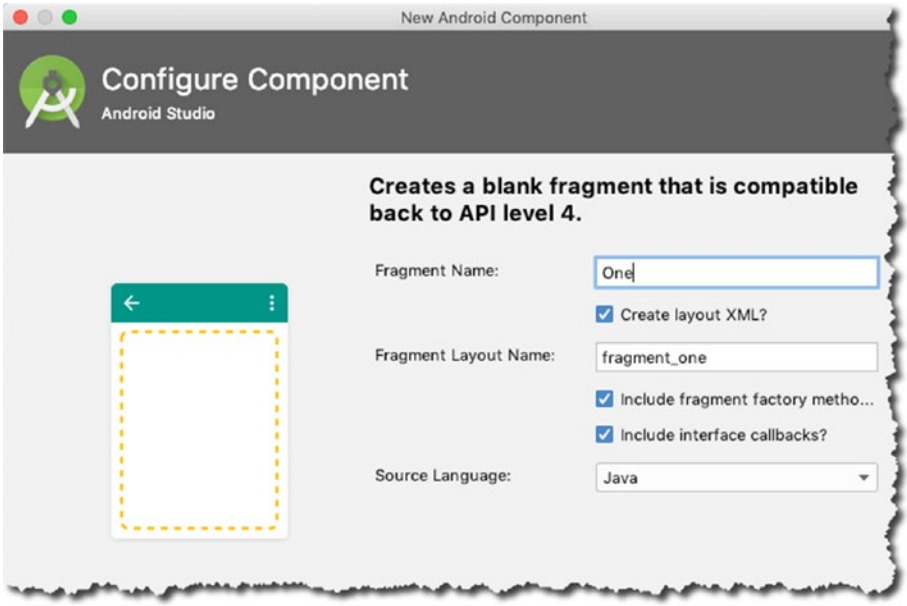

Figure 10-7. New Android component

Click Finish to start the creation of the new fragment; this fragment will become one of the destinations in your app. Create another destination and make the fragment's name "Two." The navigation editor should look like Figure 10-8.

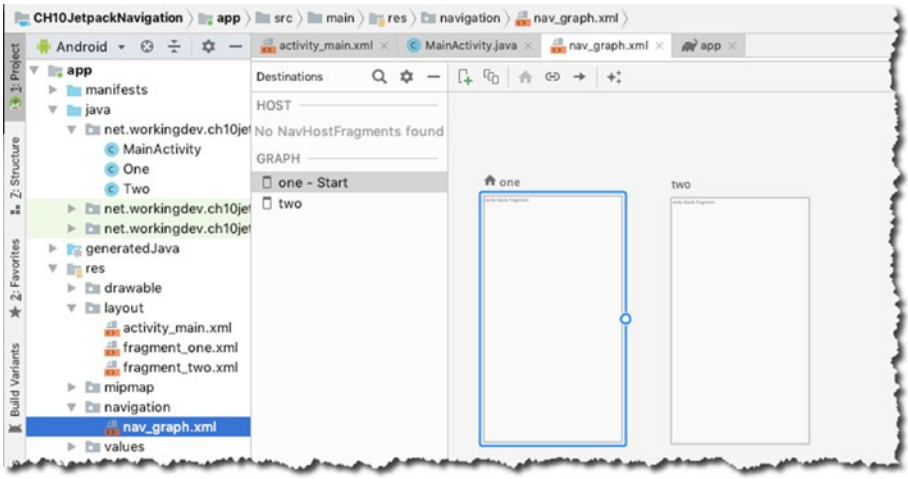

Figure 10-8. Navigation editor

Notice also there are two new Java classes (`One.java` and `Two.java`) and two new layout files (`fragment_one.xml` and `fragment_two.xml`). These files were generated when you created destinations One and Two. Notice in Figure 10-8 that fragment One has the home icon beside it. This is only because you created it first. The home destination or start destination is the first screen that your users will see. You can change the start destination any time by right-clicking any destination and then clicking the Set as start destination option, but for now, keep One as the start destination.

Now, your navigation graph doesn't have a NavHost yet; it needs one. A NavHost acts like a viewport for all your destinations. It's an empty container where destinations are swapped in and out as the user navigates through the app. The NavHost needs to be in an activity. You're going to put the NavHost in your MainActivity.

Open the layout file for your MainActivity (it's in `res/layout/activity_main.xml`) and then edit it in Text mode. The default `activity_main` contains a single TextView object; remove it and replace it with the code snippet in Listing 10-6.

Listing 10-6. Defining a NavHost in activity_main.xml

```
<fragment
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:id="@+id/nav_container"  ❶
  android:name="androidx.navigation.fragment.NavHostFragment" ❷
  app:navGraph="@navigation/nav_graph" ❸
  app:defaultNavHost="true"    ❹
  >
</fragment>
```

❶ It needs an id, just like any other element in the resource file. I used `nav_container` for this one; you can name it whatever you like.

❷ This is the fully qualified name of the `NavHostFragment` class. It belongs to the navigation component and it will be responsible for making your MainActivity the view port for all your defined destinations.

❸ The `app:navGraph` attribute tells the runtime which navigation graph you want to host in the MainActivity. Remember that you can have more than one navigation graph in the app; `nav_graph` is the name you gave to the navigation graph XML resource earlier.

❹ When you set the `defaultNavHost` to *true*, this makes sure that the `NavHostFragment` intercepts the system back button; that way, when the user clicks the back button, Android will show you the previous screen in your app, and not an external app's screen that happened to be on the back stack.

Now it's time to connect your two destinations. Open the navigation graph again; it's in `res/navigation/nav_graph`.

You want the user to navigate from destination One to destination Two. So, hover your mouse over destination One until a small circle appears on its right side. Click and drag this point over to destination Two so that the two destinations can be connected, as shown in Figure 10-9.

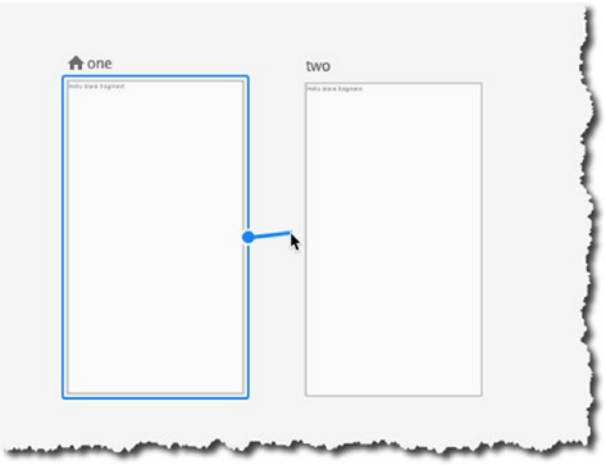

Figure 10-9. *Connecting One to Two*

Now, destination One is connected to destination Two. If you select the connection between One and Two, you'll see that it has attributes you can set, as shown in Figure 10-10. You won't deal with the attributes; you just want to connect the two destinations.

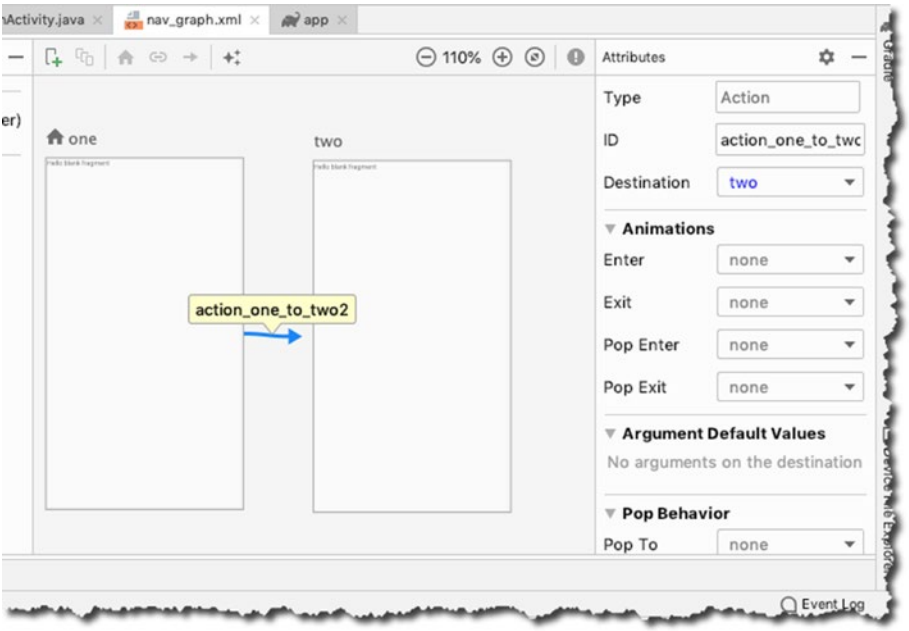

Figure 10-10. Navigation graph

In order to test this small app, you need an object in the start destination that will trigger an action, like a button. Modify the layout of the two fragments as follows:

fragment_one

- Change the layout to ConstraintLayout or whatever layout is appropriate for you.
- Remove the TextView and replace it with a button and center it.

fragment_two

- Like fragment_one, change the layout to ConstraintLayout.
- Change the text of the TextView and center it.

Figure 10-11 shows the navigation graph with a preview of the changes.

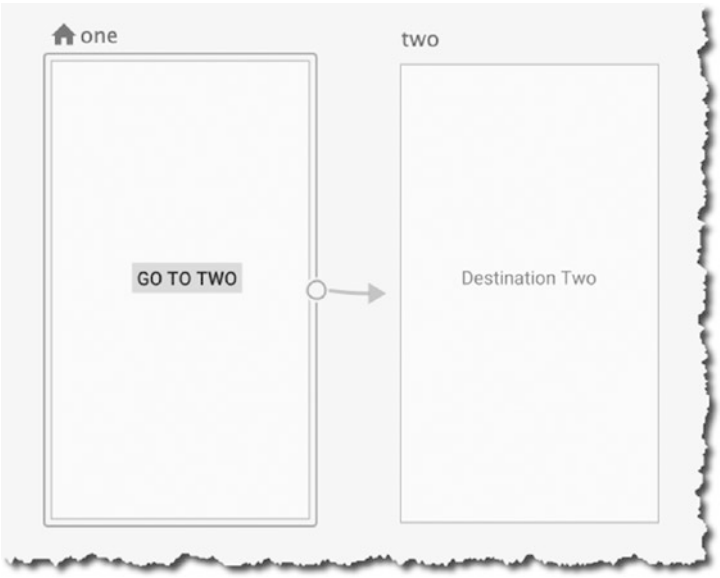

Figure 10-11. Modified fragments

Next, add a click handler to your button. You'll add the code that will make `fragment_one` navigate to `fragment_two` when the button is clicked.

Navigating to a destination is done using a NavController; it's an object that manages app navigation within a NavHost. Each NavHost has its own corresponding NavController.

A NavController lets you navigate to destinations in two ways: 1) navigate to a destination using an ID, which is what you will use here, and 2) navigate using a URI, which I will leave up to you to explore.

To add a click handler to your button, open `One.java`, which contains the Java source file for your One destination, and make sure it looks something like the one in Listing 10-7.

Listing 10-7. Class One

```
public class One extends Fragment {

  public One() {
    // Required empty public constructor
  }

  @Override
  public View onCreateView(LayoutInflater inflater, ViewGroup container,
                           Bundle savedInstanceState) {
    // Inflate the layout for this fragment

    final View view = inflater.inflate(R.layout.fragment_one, container, false);
    view.findViewById(R.id.button2).setOnClickListener(new View.OnClickListener() {
      @Override
      public void onClick(View v) {
       Navigation.findNavController(view).navigate(R.id.action_one_to_two2);
      }
    });
    return view;
  }
}
```

The most important line of code in Listing 10-7 is the `navigate()` method of the NavController object. You simply pass the ID of the action you created in the navigation graph as a parameter to `navigate()` and that does the trick. You can now launch the emulator and test the app.

This chapter merely scratched the surface of navigation components. There's a lot to discover in this area, so keep exploring.

Chapter Summary

- You can still use activity-based or fragment-based navigation in your app, just remember their pros and cons.
- Navigation components combine the best features of activity-based and fragment-based navigation; the API is easy to work with and you have more control on the back stack.
- Navigation components introduce the concept of *destinations*. Destinations can be fragments, activities, or custom views; they are what your users will navigate to.

- Destinations are grouped using a navigation graph; it's an XML resource file that contains all the *actions* between destinations.
- Destinations are connected to each other by actions.
- The basic idea of navigation is to

    1. Create a navigation graph.
    2. Create destinations.
    3. Connect the destinations; each connection becomes an action.
    4. Navigate programmatically from one destination to another using the NavController object. You can navigate using an ID or a URI.

Chapter 11

Lifecycle, ViewModel, LiveData, and Room

What this chapter covers:

- Lifecycle-aware components
- ViewModel
- LiveData
- Room

You saw a bit of the architecture components in the previous chapter. In this chapter, you'll look at some other libraries in the architecture components, namely Room. It's a persistence library that sits on top of SQLite. If you've used an ORM (object-relational mapper) before, you can think of Room as something like that.

Also in this chapter, you'll explore some more libraries in the architecture components that go hand in hand with the Room library. You'll look at lifecycle-aware components, LiveData, and ViewModel; these, together with Room, are some of the libraries you'll need to build a fluid and fluent database application.

Lifecycle-Aware Components

Lifecycle-aware components perform actions in response to a change in the lifecycle status of another component. If you're familiar with the *Observable-Observer* design pattern, lifecycle-aware components operate like that.

T. Hagos, *Android Studio IDE Quick Reference*,
https://doi.org/10.1007/978-1-4842-4953-6_11

You need to learn some new vocabulary:

- **Lifecycle Owner**: A component that has a, well, lifecycle, like an activity or a fragment. It can enter various states in its lifecycle, like *CREATED, RESUMED, PAUSED, DESTROYED,* etc. A Lifecycle Observer can tap into a Lifecycle Owner and be notified when the lifecycle status changes, like when the activity enters the CREATED state. After it enters `onCreate()`, for example, I sometimes refer to the Lifecycle Owner as an observable.
- **Lifecycle Observer**: An object that listens to the changes in the lifecycle status of a Lifecycle Owner. It's a class that implements the LifecycleObserver interface.

With the lifecycle-aware components, you can observe a component like an activity and perform actions as it enters any of its lifecycle statuses. Listings 11-1 and 11-2 show an example of how to set up an observer-observable relationship between a Lifecycle Owner (`MainActivity`) and a Lifecycle Observer (`MainActivityObserver`).

> **Note** If you're creating a project to follow the code examples in this chapter, don't forget to check the Use AndroidX artifacts box, as shown in Figure 11-1. You need the new capabilities of the AppCompatActivity to work with the lifecycle extensions.

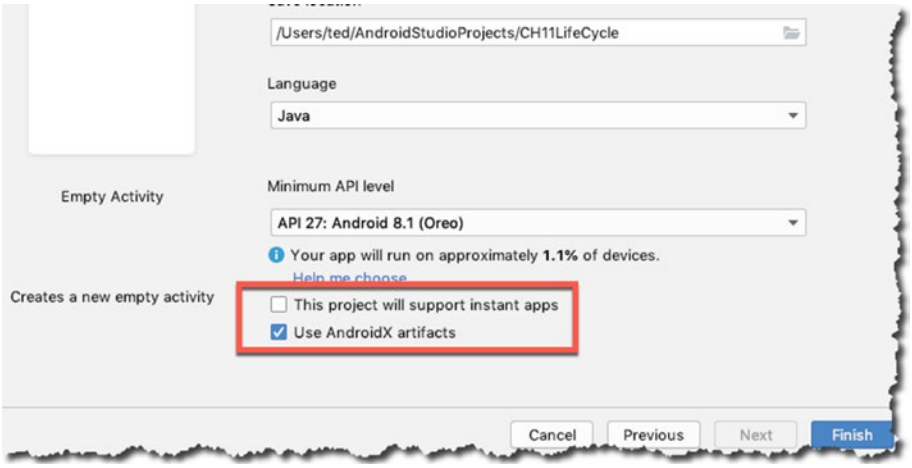

Figure 11-1. AndroidX artifacts

To demonstrate the lifecycle concepts, you will examine two classes:

- `MainActivity`: This is a simple activity, pretty much like any other activity that the IDE generates when you create a project with an empty activity. The code sample is shown in Listing 11-2.
- `MainActivityObserver`: A Java class that will implement the LifeCycleObserver interface. This will be your listener object; the code is listed and annotated in Listing 11-1.

> **Note** You can type the code examples if you prefer, or you can refer to the included book source code. If you prefer to work on the code by yourself, you need to follow this sequence:
>
> 1. Create a project with an empty activity.
> 2. Include the dependencies for Room, shown in Listing 11-3.

You'll take a look at the observer object first. Listing 11-1 shows the annotated code.

Listing 11-1. MainActivityObserver Class

```
import androidx.lifecycle.Lifecycle;
import androidx.lifecycle.LifecycleObserver;
import androidx.lifecycle.OnLifecycleEvent;

public class MainActivityObserver implements LifecycleObserver {  ❶

  @OnLifecycleEvent(Lifecycle.Event.ON_CREATE) ❷
  public void onCreateEvent() {  ❸
    System.out.println("EVENT: onCreate Event fired");  ❹
  }

  @OnLifecycleEvent(Lifecycle.Event.ON_PAUSE)
  public void onPauseEvent() {
    System.out.println("EVENT: onPause Event fired");
  }

  @OnLifecycleEvent(Lifecycle.Event.ON_RESUME)
  public void onResumeEvent() {
    System.out.println("EVENT: onResume Event fired");
  }
}
```

❶ If you want to observe another component's lifecycle changes, you need to implement the `LifecycleObserver` interface. This line makes this class an observer.

❷ Use the `OnLifecycleEvent` annotation to tell the Android runtime that the decorated method is supposed to be called when the lifecycle event happens. In this case, you're listening for the `ON_CREATE` event of the observed object. The parameter to the decorator indicates which lifecycle event you're listening for.

❸ This is the decorated method. It gets called when the object you're observing enters the `ON_CREATE` lifecycle status. You can name this method anything you want; I just named it `onCreateEvent()` because it's descriptive. Otherwise, you're free to name it to your liking; the name of the method doesn't matter because you already decorated it, so the annotation is sufficient.

❹ This is where you do something interesting in response to a lifecycle status change.

Listing 11-2. MainActivity Class

```
public class MainActivity extends AppCompatActivity {

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

    getLifecycle().addObserver(new MainActivityObserver()); ❶

  }
}
```

❶ From the point of view of the `MainActivity` (it's the one being observed), the only thing you need to do here is to add an observer object using `addObserver()` method of the `LifeCycleOwner` interface. Yes, the `AppCompatActivity` implements `LifeCycleOwner`; that's the reason you can call the `getLifecycle()` method within your activity. You simply need to pass an instance of an observer class (in your case, it's the `MainActivityObserver`) to set up lifecycle awareness between an activity and a regular class.

Again, if you're trying out this code, don't forget to add the Room dependencies in your project's `build.gradle` file (module level), as shown in the snippet in Listing 11-3.

Listing 11-3. build.gradle File, Module Level

```
dependencies {
    def lifecycle_version = "2.0.0"
    implementation "androidx.lifecycle:lifecycle-extensions:$lifecycle_
    version"
    annotationProcessor "androidx.lifecycle:lifecycle-compiler:$lifecycle_
    version"
    ...
}
```

> **Note** The lifecycle_version at the time of writing is 2.0.0, but this will be different for you since you'll be reading this at a later time. You can visit https://bit.ly/lifecyclerelnotes to find out the current version of the lifecycle libraries.

ViewModel

The Android framework manages the lifecycle of UI controllers like activities and fragments; it may decide to destroy or recreate an activity (or fragment) in response to some user action like clicking the back button or a device event like rotating the screen. These configuration changes are out of your control.

If the runtime decides to destroy the UI controller, any transient UI-related data that you're currently storing in it will be lost. Let's take the example of a simple app. Listings 11-4, 11-5, and 11-6 show a simple app that displays a random number every time the activity is created.

Listing 11-4 shows the code for the random number generator. It only has the two methods, `getNumber()` and `createRandomNumber()`; each method leaves a log statement, so you'll be able to see in the logs when and how many times they are called. The logic for the `getNumber()` method is simple: if the `minitialized` variable is false, that means you're creating an instance of the `RandomNumber` class for the first time, so you'll create the random number and then simply return it. Otherwise, you'll return the current value of `minitialized`.

Listing 11-4. RandomNumber Class

```
import android.util.Log;
import java.util.Random;

public class RandomNumber  {

  private String TAG = getClass().getSimpleName();

  int mrandomnumber;
  boolean minitialized = false;

  String getNumber() {
    if(!minitialized) {
      createRandomNumber();
    }
    Log.i(TAG, "RETURN Random number");
    return mrandomnumber + "";
  }

   void createRandomNumber() {
    Log.i(TAG, "CREATE NEW Random number");
    Random random = new Random();
    mrandomnumber = random.nextInt(100);
    minitialized = true;
  }
}
```

Listing 11-5 shows the code for the `MainActivity`. Everything happens inside the `onCreate()` method. When `MainActivity` enters the *CREATED* state, you create an instance of the `RandomNumber` class, you call the `getNumber()` method, and you set the value of the TextView to the result of the `getNumber()` method.

Listing 11-5. MainActivity Class

```
public class MainActivity extends AppCompatActivity {

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

    RandomNumber data = new RandomNumber();

    ((TextView) findViewById(R.id.txtrandom)).setText(data.getNumber());
  }
}
```

Listing 11-6 shows the layout code, in case you want to follow the code example.

Listing 11-6. activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout 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"
  tools:context=".MainActivity">

  <TextView
    android:id="@+id/txtrandom"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Hello World!"
    android:textSize="36sp"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toTopOf="parent" />

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

When you run this code for the first time, you'll see a random number displayed in the TextView—no surprises there. You'll also see the log entries for `createNumber()` and `getNumber()` in the Logcat window—no surprises there either. Now, while the app is running on the emulator, try to change the orientation of the device. You'll notice that every time the screen orientation changes, the displayed number in the TextView changes as well. Also, you'll notice that additional logs for the `createNumber()` and `getNumber()` methods show up in Logcat. The reason for this is because the runtime destroys and recreates the `MainActivity` every time the screen orientation changes. Your `RandomNumber` object also gets destroyed and recreated along with the `MainActivity`; your UI data cannot survive orientation changes.

This is a good case for using the ViewModel library so that the UI data can survive the destruction and recreation of the `Activity` class. You only need to do three things to implement ViewModel:

1. Add the lifecycle extensions to your project's dependencies, as you did earlier. Go back to Listing 11-3 for instructions.
2. To make the `RandomGenerator` class a ViewModel, you extend the `ViewModel` class from the AndroidX lifecycle libraries.

3. From the `MainActivity`, you get an instance of the `RandomNumber` class using the factory method of the `ViewModelProviders` class, instead of simply creating an instance of the `RandomNumber` class.

Listing 11-7 shows the changes in the `RandomNumber` class; the `RandomNumber` class is transformed automatically to a ViewModel object by simply extending the `ViewModel` class.

Listing 11-7. RandomNumber Extends ViewModel

```
import java.util.Random;
import androidx.lifecycle.ViewModel;

public class RandomNumber extends ViewModel {

  private String TAG = getClass().getSimpleName();

  int mrandomnumber;
  boolean minitialized = false;

  String getNumber() {
    if(!minitialized) {
      createRandomNumber();
    }
    Log.i(TAG, "RETURN Random number");
    return mrandomnumber + "";
  }

  void createRandomNumber() {
    Log.i(TAG, "CREATE NEW Random number");
    Random random = new Random();
    mrandomnumber = random.nextInt(100);
    minitialized = true;
  }
}
```

The class remains largely the same as its previous version, shown in Listing 11-4; the only difference is that now it extends ViewModel.

Now, let's implement the changes on the `MainActivity`. Listing 11-8 shows the modified `MainActivity`, which uses `ViewModelProviders` to get an instance of the `RandomNumber` class, which is the ViewModel object.

Listing 11-8. MainActivity and ViewModelProviders

```
import androidx.lifecycle.ViewModelProviders;
import android.os.Bundle;
import android.widget.TextView;

public class MainActivity extends AppCompatActivity {

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

    RandomNumber data;
    data = ViewModelProviders.of(this).get(RandomNumber.class); ❶

    ((TextView) findViewById(R.id.txtrandom)).setText(data.getNumber());
  }
}
```

❶ This is the only change you need to do in `MainActivity`. Instead of directly managing the instance of the ViewModel object (the `RandomNumber` class) by creating an instance of it inside the `onCreate()` method, you let the `ViewModelProviders` class manage the scope of your ViewModel object.

LiveData

Going back to the RandomNumber example, what you have is an app that shows a random number every time the app is launched. The app uses ViewModel already, so you don't have the problem of losing data every time the activity is destroyed and recreated. The basic data flow is shown in Figure 11-2.

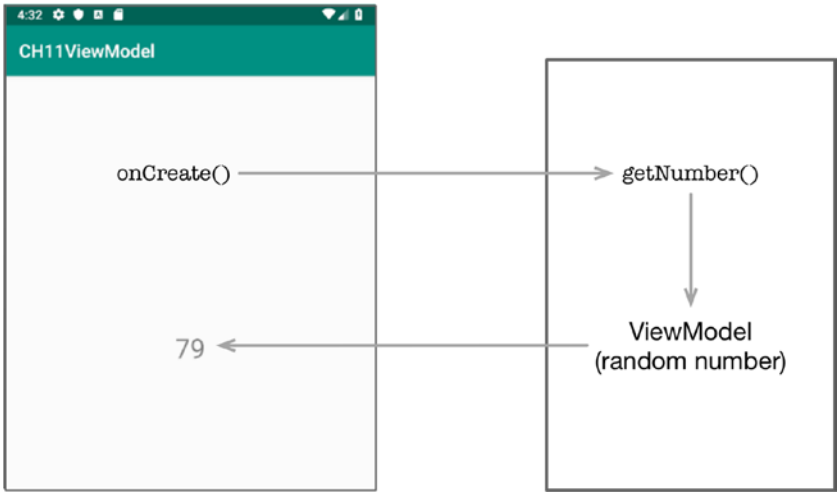

Figure 11-2. Data flow for the RandomNumber example

But what if you need to fetch another number? For that, you can add a trigger in the `MainActivity`, like a Button, and then it will call `getNumber()` on the ViewModel, but how are you going to refresh the TextView in the `MainActivity`? There are a couple of ways to do this, and you may have encountered them already. One way to facilitate data exchange between your ViewModel and an activity is the creative use of interfaces (but I won't discuss that here) or by using an EventBus like Otto (I also won't discuss that here)—but now, thankfully, because of architecture components, you can use LiveData. The new data flow is depicted in Figure 11-3.

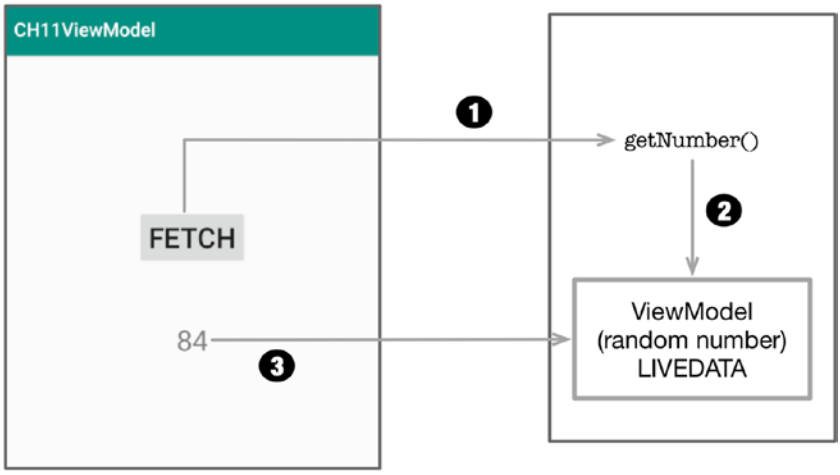

Figure 11-3. RandomNumber sample with LiveData

❶ The user clicks the **FETCH** button, which calls the `getNumber()` function; actually, it calls the `createNumber()` first and then call the `getNumber()`. This way, you're really fetching a new random number. There are ways to do this more elegantly, but this is the quickest way to do it, so bear with me.

❷ Your ViewModel object gets a new random number. This isn't a simple String anymore; it was changed to a MutableLiveData so it becomes *observable.*

❸ From the `MainActivity`, you get an instance of the LiveData coming from your ViewModel object and write some code to *observe* it. Next, you simply react to changes in the LiveData.

Let's see how that works in code. Listings 11-9 and 11-10 show the code for your ViewModel and `MainActivity`, respectively.

Listing 11-9. ViewModel with LiveData

```
import androidx.lifecycle.MutableLiveData;
import androidx.lifecycle.ViewModel;

public class RNModel extends ViewModel {
  private String TAG = getClass().getSimpleName();
  MutableLiveData<String> mrandomnumber = new MutableLiveData<>(); ❶
  boolean minitialized = false;

  MutableLiveData<String> getNumber() { ❷
    if(!minitialized) {
      createRandomNumber();
    }
    Log.i(TAG, "RETURN Random number");
    return mrandomnumber;
  }

  void createRandomNumber() {
    Log.i(TAG, "CREATE NEW Random number");
    Random random = new Random();

    mrandomnumber.setValue(random.nextInt(100) + ""); ❸
    minitialized = true;
  }
}
```

❶ The value of `mrandomnumber` is what you return to the `MainActivity`. You want this to be an *observable* object. To do this, you change its type from `int` to MutableLiveData.

❷ You must make that type change here too. Since `mrandomnumber` is now MutableLiveData, this function has to return MutableLiveData too.

❸ To set the value of the MutableLiveData, use the `setValue()` method.

Now you can move on the changes in MainActivity. Listing 11-10 shows the modified and annotated code for MainActivity.

Listing 11-10. MainActivity

```
import androidx.appcompat.app.AppCompatActivity;
import androidx.lifecycle.MutableLiveData;
import androidx.lifecycle.Observer;
import androidx.lifecycle.ViewModelProviders;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;

public class MainActivity extends AppCompatActivity {

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

    final RNModel data;
    data = ViewModelProviders.of(this).get(RNModel.class);

    final TextView txtnumber = (TextView) findViewById(R.id.txtrandom);
    MutableLiveData<String> mnumber = data.getNumber(); ❶

    Button btn = (Button) findViewById(R.id.button);  ❷
    btn.setOnClickListener(new View.OnClickListener() {
      @Override
      public void onClick(View v) {
        data.createRandomNumber();
        data.getNumber();
      }
    });

    mnumber.observe(this, new Observer<String>() { ❸
      @Override
      public void onChanged(String val) { ❹
        txtnumber.setText(val);
      }
    });
  }
}
```

❶ Fetch the random number from the ViewModel. The random number isn't of String type anymore; it's MutableLiveData.

❷ This is the boilerplate code for a button click. You need this trigger to fetch a random number from the ViewModel.

❸ To *observe* a LiveData, you call the `observe()` method. The method takes two arguments: the first argument is the lifecycle owner (`MainActivity`, so you pass this) and the second argument is an Observer object. You use an anonymous class here to create the Observer object.

❹ This `onChanged()` method is called every time the value of the random number (`mrandomnumber`) in the ViewModel changes, so when it changes, you set the value of the TextView accordingly.

Cool, right?! If you're still not sold on using LiveData, here are a couple of things to consider. When you use LiveData,

- You're sure the UI **always matches the data state**. You've already seen this from the example. LiveData follows the Observer pattern, so it notifies the observer when its value changes.
- There are **no memory leaks**. Observers are bound to lifecycle objects. If, for example, your `MainActivity` enters the paused state (for whatever reason, like maybe another activity is on the foreground), the LiveData won't be observed. If the `MainActivity` is destroyed, the LiveData again won't be observed, and it will clean up after itself—which also means you won't need to manually handle the lifecycles of `MainActivity` and the ViewModel.

Room

If you want to include database functionalities in your app, you might want to look at Room. Before Room, the two popular ways to build database apps were to either to use Realm or just use good ole SQLite. Dealing with SQLite was considered to be a bit low-level; it felt too close to the metal and as such was a bit cumbersome to use. Realm was quite popular among developers but no matter the popularity, it wasn't a first-party solution. Thankfully, we now have Room.

Room is an abstraction on top of SQLite. If you've used an ORM before, like Hibernate, it's similar to that. Room has several advantages over using plain vanilla SQLite. With Room,

- You don't have to deal with raw queries for basic database operations.
- At compile time, it verifies the SQL queries, so you don't need to worry about SQL injection—remember that?
- There is no impedance mismatch between your database objects and Java objects. Room takes care of it, so you only need to deal with Java objects.

Room has three major components: `Entity, Dao, and Database`. Their relationship with the application is shown in Figure 11-4.

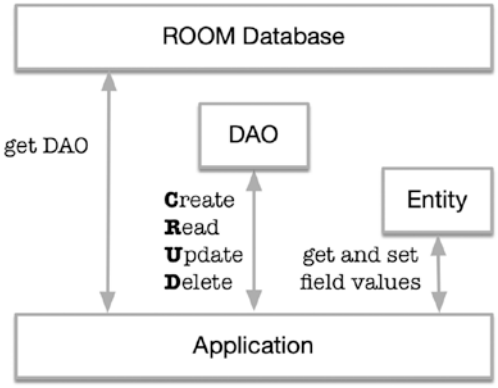

Figure 11-4. Room components

- **Entity**: An Entity is used to represent a database table. You code it as a class that's decorated by the `@Entity` annotation.
- **The Dao** (Data Access Object) is a class that contains the method to access the tables in the database. This is where you code your CRUD (create, read, update, and delete) operations. This is an *interface* that's decorated by the `@Dao` annotation.
- **Database**: This component holds a reference to the database. It's an abstract class that is annotated by the `@Database` annotation.

Before you can use Room in a project, you need to add its dependencies to the `build.gradle` file (module level), as shown in Listing 11-11.

Listing 11-11. Room Dependencies

```
dependencies {
    def room_version = "2.1.0-alpha07"
    implementation "androidx.room:room-runtime:$room_version"
    annotationProcessor "androidx.room:room-compiler:$room_version"
    . . .
}
```

Listings 11-12, 11-13, 11-14, and 11-15 show the four Java source files that demonstrate a very basic usage of Room.

Listing 11-12. Person Class, the Entity

```
import androidx.annotation.NonNull;
import androidx.room.Entity;
import androidx.room.PrimaryKey;

@Entity(tableName = "person") ❶
public class Person {
  @PrimaryKey(autoGenerate = true) ❷
  @NonNull public int uid;

  @ColumnInfo(name="last_name") ❸
  public String last_name;
  public String first_name;

  public Person(String lname, String fname) {
    last_name = lname;
    first_name = fname;
  }

  public Person() {}
}
```

❶ The `@Entity` annotation makes this an Entity. If you don't pass the `tableName` argument, the name of the table will default to the name of the decorated class. You only need to pass this argument if you want the table's name to be different from the decorated class. So, what I wrote here is unnecessary and redundant because I set the value of `tableName` to "person", which is the same as the name of the decorated class.

❷ You're making the `uid` member variable as the primary key; you're also saying it can't be null.

❸ The member variables of the class will automatically become the fields on the table. The column names on the table will take after the names of the member variables unless you use the `@ColumnInfo` annotation. If you want the name of the table field (column) to be different from the name of the member variable, use the `@ColumnInfo` decoration, as shown here, and set the `name` to your preferred column name.

Listing 11-13. PersonDAO, the Data Access Object

```
import java.util.List;
import androidx.room.Dao;
import androidx.room.Delete;
import androidx.room.Insert;
import androidx.room.Query;
import androidx.room.Update;

@Dao ❶
interface PersonDAO { ❷
  @Insert  ❸
  void insertPerson(Person person);

  @Update
  void updatePerson(Person person);

  @Delete
  void deletePerson(Person person);

  @Query("SELECT * FROM person") ❹
  public List<Person> listPeople();
}
```

❶ A DAO needs to be annotated by the `@Dao` decorator.

❷ DAOs must be written as interfaces.

❸ Use the `@Insert` decorator to indicate that the decorated method will be used for inserting records to the table. Similarly, you decorate methods for update, query, and delete with `@Update`, `@Query`, and `@Delete`, respectively.

❹ Use the `@Query` to write SQL select statements. Each `@Query` is verified at compile time; if there is a problem with the query, a compilation error occurs instead of a runtime error. This should put your mind at ease.

Listing 11-14. AppDatabase, the Database Holder

```
import android.content.Context;
import androidx.room.Database;
import androidx.room.Room;
import androidx.room.RoomDatabase;

@Database(entities = {Person.class}, version = 1) ❶
public abstract class AppDatabase extends RoomDatabase { ❷

  private static AppDatabase minstance;
  private static final String DB_NAME = "person_db";

  public abstract PersonDAO getPersonDAO(); ❸
```

```
  public static synchronized AppDatabase getInstance(Context ctx) { ❹
    if(minstance == null) {
      minstance = Room.databaseBuilder(ctx.getApplicationContext(), ❺
          AppDatabase.class,
          DB_NAME)
          .fallbackToDestructiveMigration()
          .build();
    }
    return minstance;
  }
}
```

❶ Use the `@Database` to signify that this class is the Database holder. Use the `entities` argument to specify the Entities that are in the Database. If you have more than one Entity, use commas to separate the list. The second argument is the `version`; this is an integer value that specifies the version of your db.

❷ A `Database` class is *abstract* and extends the `RoomDatabase`.

❸ You need to provide an abstract class that will return an instance of the DAO object.

❹ You need to provide a static method to get an instance of the Database. It doesn't have to be a Singleton, like what I did here, but I imagine you don't want more than one instance of the `Database` class.

❺ Use the `databaseBuilder()` method to create an instance of the RoomDatabase. There are three arguments to the builder method: 1) an application context, 2) the abstract class, which is annotated by `@Database`, and 3) the name of the database file. This is file name of the SQLite db.

Now that all of your Room components are in place, you can use them from your app. Listing 11-15 shows how to use Room components from an activity.

Listing 11-15. MainActivity

```
public class MainActivity extends AppCompatActivity {

  private AppDatabase db;

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

```
    Button btn = (Button) findViewById(R.id.button);
    btn.setOnClickListener(new View.OnClickListener() {
      @Override
      public void onClick(View v) {
        saveData();
        System.out.println("Clicked");
      }
    });
    db = AppDatabase.getInstance(this); ❶
  }

  private void saveData() {
    final String mlastname = ((TextView) findViewById(R.id.txtlastname)).
    getText().toString();
    final String mfirstname = ((TextView) findViewById(R.id.txtfirstname)).
    getText().toString(); ❷

    new Thread(new Runnable() { ❸
      @Override
      public void run() {
        Person person = new Person(mlastname, mfirstname); ❹
        PersonDAO dao = db.getPersonDAO(); ❺
        dao.insertPerson(person); ❻
        List<Person> people = dao.listPeople(); ❼
        for(Person p:people) { ❽
          System.out.printf("%s , %s\n", p.last_name, p.first_name );
        }
      }
    }).start();
  }
}
```

❶ To begin using the RoomDatabase, get an instance of it using the factory method you coded in the AppDatabase earlier.

❷ Let's collect the data from the TextViews.

❸ Room follows best practices, so it won't allow you to run any database query on the main UI thread. You need to create a background thread and run all your Room commands in there. In here, I used quick and dirty Thread and Runnable objects, but you're free to use any other means of background execution, like AsyncTask.

❹ Create a Person object using the inputs from the TextViews.

❺ Get an instance of the DAO.

❻ Do an *insert* using the `insertPerson()` method you coded in the DAO earlier.

❼ Do a `SELECT;`.

❽ List all entries in the `person` table.

In a real app, you probably wouldn't access the database from a UI controller like an activity. You might want to put in a ViewModel class; that way, the UI controller's responsibility is strictly to present data and not to act as a model.

If you use Room with ViewModel and LiveData, it can provide a more responsive UI experience. I didn't cover it here, but it's a great exercise to pursue after this chapter.

Chapter Summary

- AppCompatActivity objects are now lifecycle owners. You can write another class and listen to the lifecycle changes of a lifecycle owner and then react accordingly. Fragments too are lifecycle owners—don't forget to use AndroidX artifacts on your project when working with lifecycle-aware components.
- ViewModel makes your UI data resilient to the destruction and recreation of UI controllers (like activities and fragments).
- LiveData makes the relationship between your UI object and model data bidirectional. A change in one is automatically reflected in the other.
- Room is an ORM for SQLite. It's a first-party solution and it's part of the architecture components. There's little reason why you shouldn't use it.

Chapter 12

Release Builds

What this chapter covers:

- Preparing for release
- Signing the app
- Google Play
- The app bundle

You can distribute your app quite freely and without many restrictions. You can let your users download it from your website, Google Drive, Dropbox, and more; you may even email the app directly to users, if you wish. Many developers choose to distribute their app on a marketplace like Google or Amazon to maximize reach. Another reason to put your app on a trusted digital marketplace is, well, it's trusted. When an app is not downloaded from a trusted source, the apps won't be instantly usable because the user will get a notification that the app is not from a trusted source.

In this chapter, I'll discuss the things you need to do to get your app out in Google Play.

Preparing the App for Release

There are three things you need to keep in mind when preparing an app for release:

1. Prepare the material and assets for release.
2. Configure the app for release.
3. Build a release-ready app.

T. Hagos, *Android Studio IDE Quick Reference*,
https://doi.org/10.1007/978-1-4842-4953-6_12

Preparing the Material and Assets for Release

Your code is great and you might even think it's clever, but the user will never see it. What they will see are your View objects, the icons, and the other graphical assets. You should polish them.

If you think the app's icon isn't a big deal, think again. The icon helps the users identify your app as it sits on the home screen. This icon also appears in other areas like the launcher window, the downloads section, and more importantly, it appears in the store where the app is published. The icon creates the first impression of your app. It's a good idea to put some work into it. Start by reading Google's guidelines for app icons at `http://bit.ly/androidreleaseiconguidelines`. While you're at it, also visit `https://romannurik.github.io/AndroidAssetStudio/`; this resource will save you plenty of time when generating assets for your app.

Other things to consider if you want to publish the app in Google's marketplace are graphical assets like screen captures and the text for promotional copy. Make sure to read Google's guideline for graphical assets, which can be found at `http://bit.ly/androidreleasegraphicassets`.

Configuring the App for Release

1. **Check the package name**. The app may have started as an exercise or throw-away code, and then it grew and took on a life on its own. You may want to check the package name of the app. Make sure it isn't still `com.example.myapp`. The package name makes the app unique across the Google marketplace, and once you decide on a package name, you can't change it. So give it some thought. You already know how to change this; it's covered in the Gradle chapter, remember?

2. **Deal with the debug information**. Make sure that the `android:debuggable` attribute in the `<application>` tag of the Manifest is removed. You just need to check, really, because Android Studio would have removed this automatically when you change the mode to "release."

3. **Remove the log statements**. Different developers do this differently. Some painstakingly go through the code and remove the statements manually. Some write sed or awk programs to strip away the log statements. Some use ProGuard, and others use

third-party tools like Timber to take care of logging activities. It's up to you which you will use, but make sure that your users won't accidentally see the log information. If you haven't made up your mind yet, I really urge you to try Timber.

4. **Check the application's permissions**. During development, you may have experimented on some features of the application and you may have set permissions on the manifest like permission to use the network, write to external storage, etc. Review the `<uses-permission>` tag on the manifest and make sure that you don't grant permissions that the application does not need.

5. **Check remote servers and URLs**. If your application relies on web APIs or cloud services, make sure that the release build of the app is using production URLs and not test paths. You may have been given sandboxes and test URLs during development; you need to switch them up to the production version.

Building a Release-Ready Application

During development, Android Studio did the following things for you:

1. Created a debug certificate
2. Assembled all your project's assets, config files, and runtime binaries into an APK
3. Signed the APK using a debug certificate
4. Deployed the APK to an emulator or a connected device

All these things happened in the background; you didn't have to do anything else but write your code. Now you need to take care of that certificate. Google Play and other similar marketplaces won't distribute an app that's signed with a debug certificate. It needs to be a proper certificate. You don't need to go a certificate authority like Thawte or Verisign; a self-signed certificate will suffice.

In the next steps, I'll walk you through how to generate a signed bundle or APK. You already know what an APK is—it's the package that contains your application. It's what you upload to Google Play. A bundle, on the other hand, is a lot like an APK but it's a newer upload format. Like the APK, it

also includes all your app's compiled code and resources, but it defers the APK generation. It's Google Play's new app serving model called Dynamic Delivery. It uses your app bundle to generate and serve an optimized APK for each user's device configuration—so they download only the code and resources that they need to run your app. You don't have to build, sign, and manage multiple APKs anymore.

In Android Studio, the steps to generate an APK and a bundle are almost identical. In the following steps, you'll see how to generate both the bundle and an APK.

Launch Android Studio, if it isn't open yet. Open the project and from the main menu bar, go to Build ➤ Generate Signed Bundle/APK, as shown in Figure 12-1.

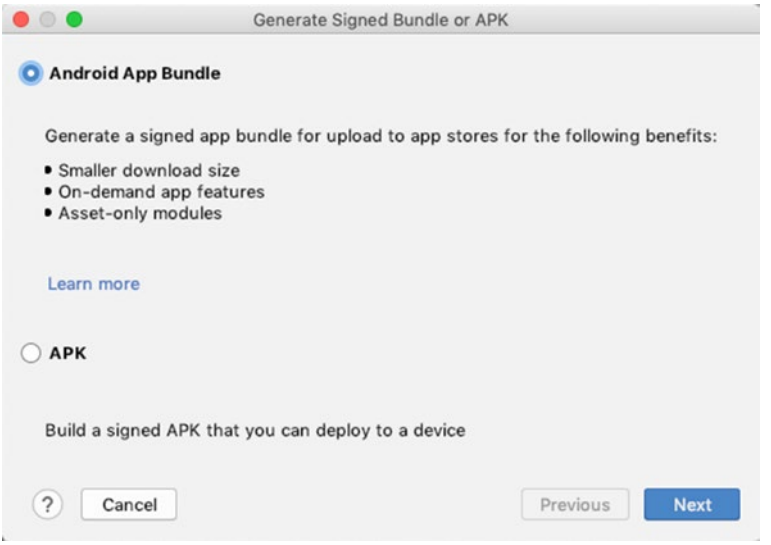

Figure 12-1. Generating a signed APK

Choose either Bundle or APK, and then click Next. In this example, I chose to create a bundle. When you click Next, you will see the Key Store dialog, as shown in Figure 12-2.

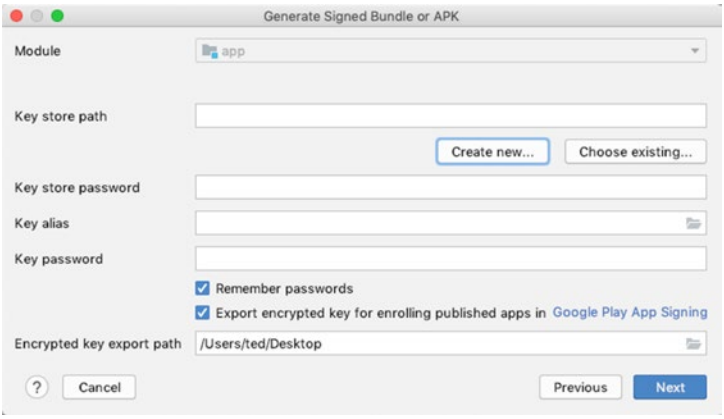

Figure 12-2. Key store dialog

The Key store path field is asking for the location of your Java Key Store (JKS) file. At this point, you don't have it yet. So, click the Create new button. You'll see the dialog window for creating a new key store, as shown in Figure 12-3.

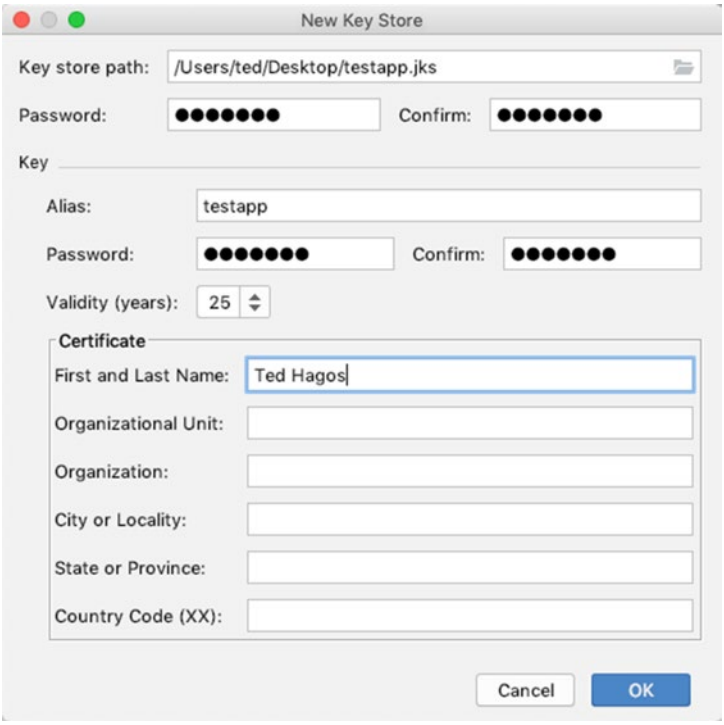

Figure 12-3. New key store

Table 12-1 shows the description for the input items of the key store.

Table 12-1. Key store Items and Description

Key store items	Description
Key store path	The location where you want to keep the key store. This is entirely up to you. Just make sure you remember this location.
Password	This is the password for the key store. Don't lose it. Make sure you remember this one. Otherwise, you'll need to create another key store file.
Alias	This alias identifies the key. It's just a friendly name for it.
(Key) Password	This is the password for the key. This is **NOT** the same password as the key store's (but you can use the same password if you like).
Validity, in years	The default is 25 years; you can just accept the default. If you publish on Google Play, the certificate must be valid until October of 2033, so 25 years should be fine
Other information	Only the first and last name fields are required.

When you're done filling up the New Key Store dialog, click OK. This will bring you back to the Generate Signed APK window, as shown in Figure 12-4, but now the JKS file is created and the Key Store dialog is populated with it.

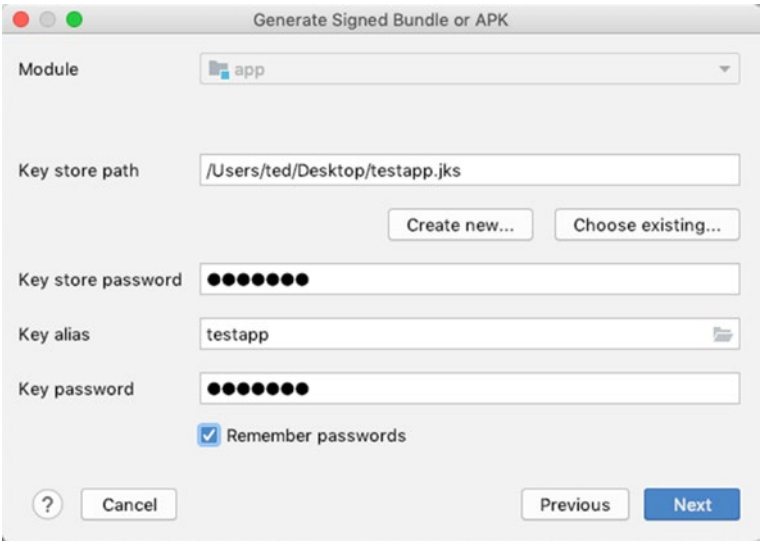

Figure 12-4. Generate Signed Bundle/APK screen, populated

Click Next. Now you'll choose the destination of the signed bundle, as shown in Figure 12-5.

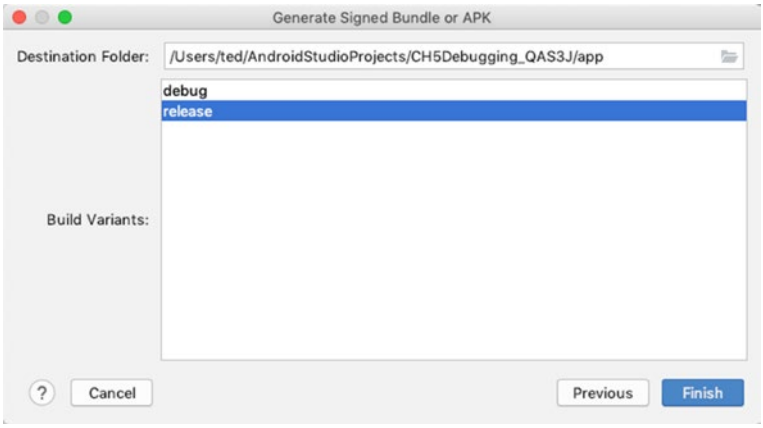

Figure 12-5. Signed APK, APK destination folder

You need to remember the location of the destination folder, as shown in Figure 12-5. This is where Android Studio will store the signed bundle. Also, make sure that the build type is set to "`release`."

When you click Finish, Android Studio will generate the signed bundle for your app. This is the file that you will submit to Google Play.

Releasing the App

Before you can submit an app to Google Play, you'll need a developer account. If you don't have one yet, you can sign up at `https://developer.android.com`. There are a lot of assumptions I'm making about the next activities. I'm assuming that

1. You already have a Google account (Gmail), and
2. You're using Google Chrome to go to `https://developer.android.com`, and
3. Your Google account is logged into Chrome.

If you're Google account isn't logged into Chrome, you might see something like Figure 12-6. Chrome will ask you go select an account (or create one).

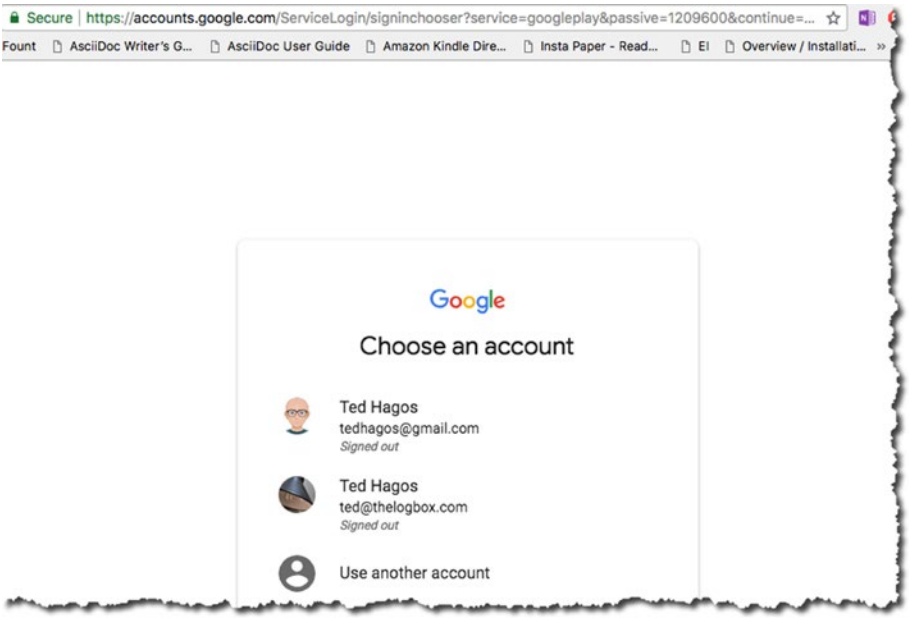

Figure 12-6. Choose an account

When you get your Google account sorted out, you'll be taken to the `developer.android.com` website, as shown in Figure 12-7.

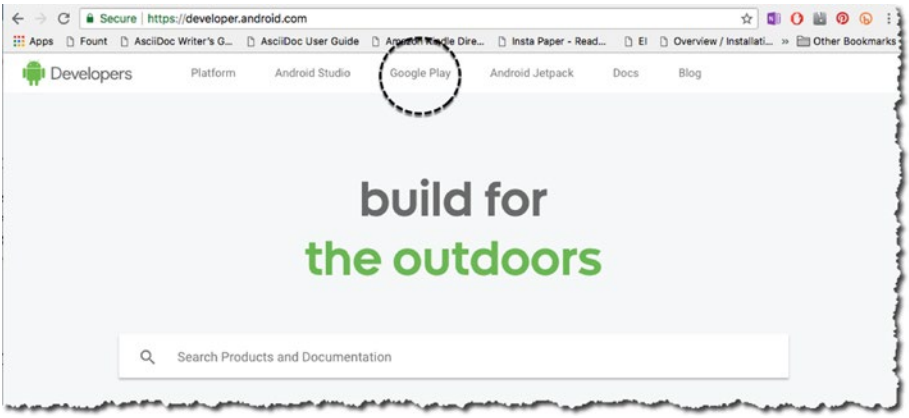

Figure 12-7. developer.android.com

Click Google Play, as shown in Figure 12-7. Click Launch Play Console, as shown in Figure 12-8.

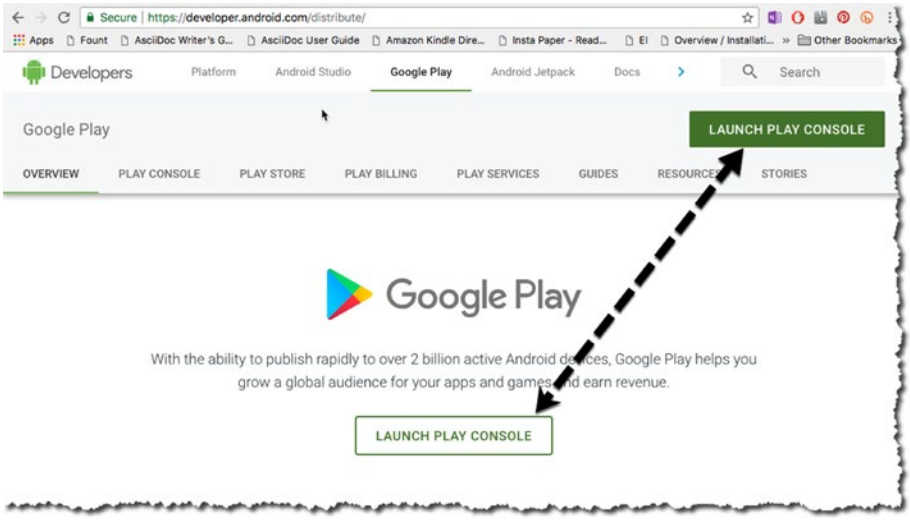

Figure 12-8. *Launching the Google Play Console*

You need to go through four steps to complete the registration (shown in Figure 12-9):

1. Sign in with your Google account.
2. Accept the developer agreement.
3. Pay the registration fee.
4. Complete your account details.

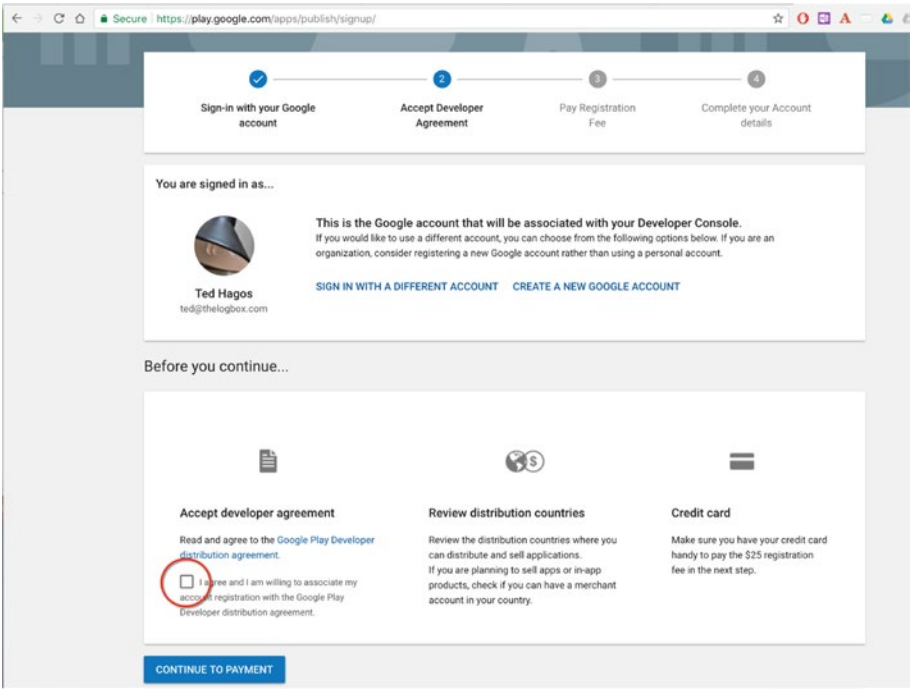

Figure 12-9. *The Google Play Console sign up*

Once you have completed the registration and one-time payment, you will now have access to the Google Play Console, as shown in Figure 12-10.

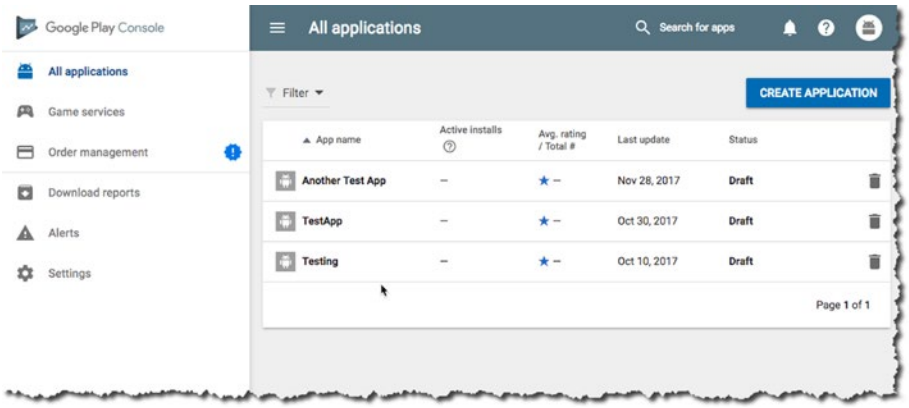

Figure 12-10. *The Google Play Console*

This is where you can start the process of submitting your app to the store. Click the Create application button to get started.

Chapter Summary

- Before the users can experience your app, they will see the icons and other graphical assets first. Make sure the graphical assets are just as polished as your code.
- Strip your code of all debug info and log statements before you build a release.
- Code review your own work. If you have buddies or other people who can review the code with you, that's much better. If your app uses servers, RESTful URLs, etc., make sure they are production ready and not sandboxed.
- Before you can upload your app to Google Play, you need to sign your app with a proper certificate.
- You'll need a Google Play account if want to sell your apps on Google Play. I paid a one-time fee of $25 USD, but that was a couple of years ago.
- Don't forget to test your app on a real device.

Chapter 13

Short Takes

What this chapter covers:

- How to import sample code
- Refactoring
- Code generators
- Live templates
- Code editor preferences
- Keyboard shortcuts

This quick reference book wouldn't be a quick reference it described every nuance of Android Studio in detail, but before I end the book, I'd like to point out some features of Android Studio that makes our coding lives easier. Like I said, I won't go into detail because that's not the goal; rather, the goal is to let you know that these features exist.

Productivity Features

What we usually mean when talking about productivity is that we want to do what we need to do in the shortest possible time, which translates to keyboard shortcuts, templates, snippets, and so on. In this section, you'll take a peek at some of what Android Studio has to offer to give your productivity a little boost. I'll show you what's available.

T. Hagos, *Android Studio IDE Quick Reference*,
https://doi.org/10.1007/978-1-4842-4953-6_13

Importing Samples

A key part of boosting your productivity is to actually learn how to create things and discover how they work in Android. So, my first productivity tip is to learn how to use the "import sample" feature. You can get to this feature from the main menu bar via File ➤ New ➤ Import Sample**.** Figure 13-1 shows the Import Sample screen.

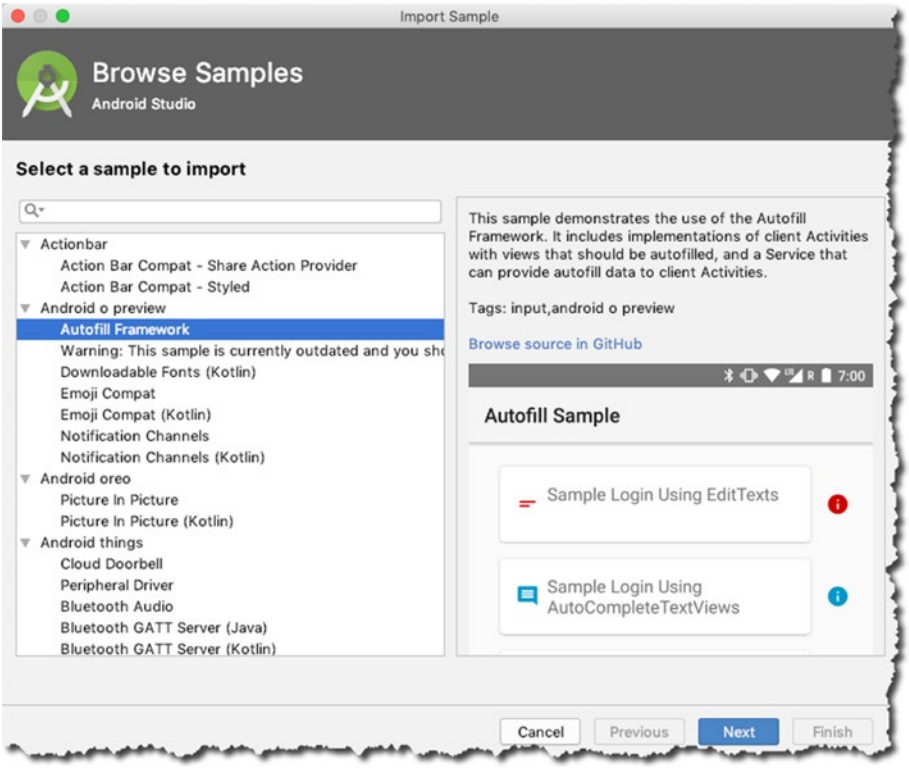

Figure 13-1. *Importing a sample*

What you see in Figure 13-1 is a list of code samples you can either browse or create as a local project.

Let's say you want to learn something about the Autofill Framework, like what you see in Figure 13-1. You can see a preview of what it looks like, and you can also click the Browse in GitHub link. When you click Next, you'll see a dialog that's somewhat similar to when creating a new project, as shown in Figure 13-2.

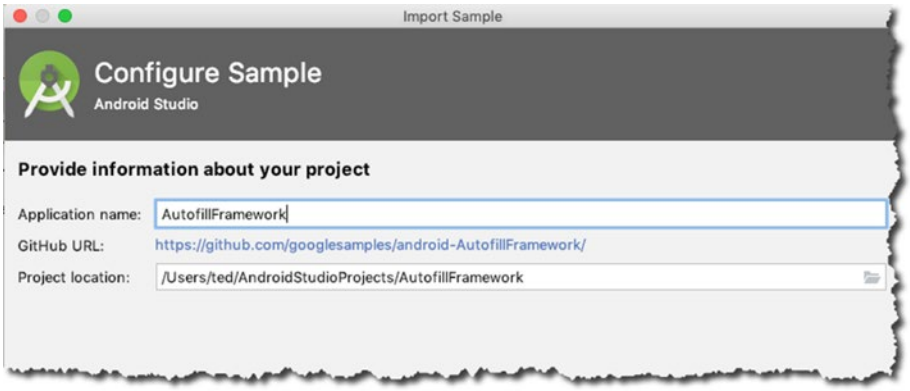

Figure 13-2. Import Sample, next window

If you click Finish in the Import Sample dialog, Android Studio will create a new project locally and download the sample file from GitHub so you can take a closer look at it and work on it right away.

Refactoring

Refactoring is basically rewriting and improving your source code without creating new functionality; this practice helps keep the code SOLID and DRY (don't repeat yourself) and thus easier to maintain.

> **Note** I spelled SOLID in all caps because it's also an acronym that stands for **S**ingle Responsibility, **O**pen-Closed Principle, **L**iskov Principle, **I**nterface Segregation, and **D**ependency Inversion Principle. These are principles of object-oriented design, which was popularized by Robert C. Martin.

Android Studio has some nifty refactoring capabilities. It's easy to get started: just select a piece of code that you'd like to refactor and then use the context-sensitive right-click, shown in Figure 13-3. Alternatively, you can use the keyboard shortcuts: `Ctrl + T` for macOS and `Ctrl + Alt + Shift + T` for Windows/Linux.

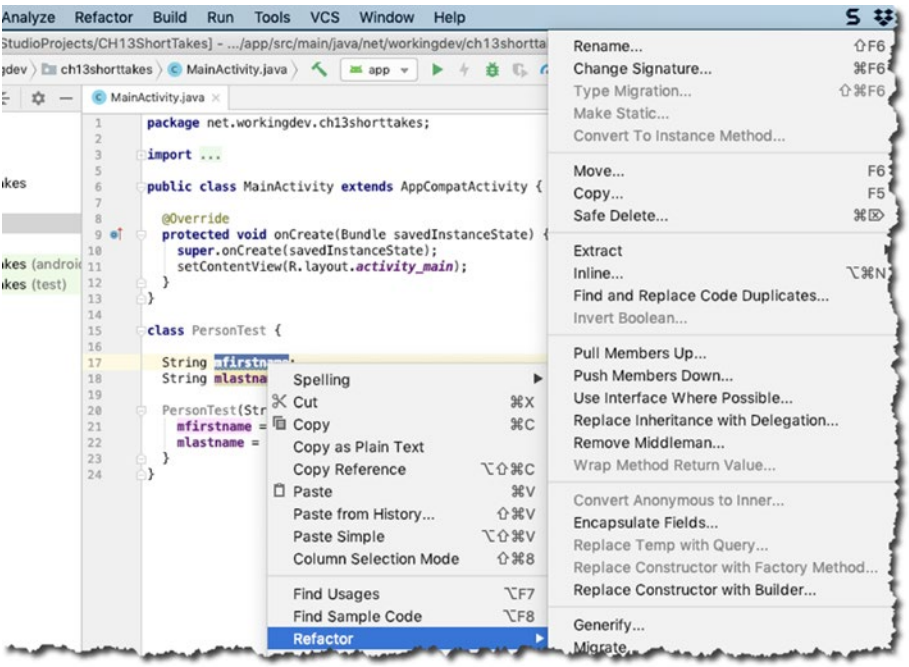

Figure 13-3. Refactoring

I'm sure you've done refactoring many times before, but let's just jog your memory here.

- **Rename**: This will let you safely rename variables and other identifiers. You should use this instead of `Find and Replace`. This works across the entire project, and not only in the current file.
- **Change Signature**: This will let you change a method, either its name or the parameters. It also works at a class level, so you can turn a class into a generic type and manipulate the type parameters.
- **Move**: Moves an element. You can move a method to another class if you want to.
- **Copy**: Lets you copy elements, like the currently selected class.
- **Safe Delete**: If you need to delete something, Android Studio will verify that what you're deleting isn't in use by anything else in the code base. If it is in use, you'll be prompted so you can address those things before you actually delete something important.

- **Extract Constant**: Avoid using hard-coded values. You shouldn't, you know it, and you know why. The Extract option for refactoring works not only for constants; you can extract fields, methods, superclasses, variables, parameters, and interfaces.

There are plenty more options in the Refactor menu; make sure to check the others out.

Generate

Another time-saving feature of Android Studio is the code generator. It's aptly named because it does exactly what you think it does—it generates code. Let's take an example. Figure 13-4 shows the keyboard cursor inside a class definition. While the cursor is within the class body, launch the Generator action; from the main menu bar, go to Code ➤ Generate.

Figure 13-4. Main menu bar, Code ➤ Generate

As you can see, you can generate quite a lot of boilerplate code. When you choose any of the Generate options, Android Studio will generate a generalized stub of code. Choose the getter and setter option. Let's say you have a `PersonTest` class, as shown in Figure 13-4. While the keyboard cursor is still within the PersonTest class, go to the main menu bar and choose Code ➤ Generate. Alternatively, you can use the keyboard shortcut `Command + N` (macOS) or `Alt + Insert` (Windows and Linux) and then choose to generate getters and setters. You'll see the dialog shown in Figure 13-5.

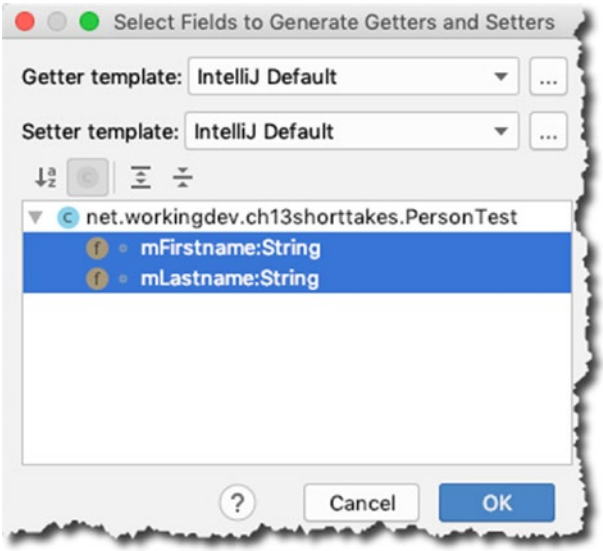

Figure 13-5. Generating getters and setters

The generator dialog shows all the autodetected fields in your class. It shows the `mFirstname` and `mLastname` member variables; it also lets you do multiple selections. Select both member variables and click OK. Listing 13-1 shows the PersonTest class after code generation.

Listing 13-1. PersonTest Class

```
class PersonTest {

  String mFirstname;
  String mLastname;

  public String getmFirstname() {
    return mFirstname;
  }

  public void setmFirstname(String mFirstname) {
    this.mFirstname = mFirstname;
  }

  public String getmLastname() {
    return mLastname;
  }

  public void setmLastname(String mLastname) {
    this.mLastname = mLastname;
  }
```

```
  public PersonTest(String mFirstname, String mLastname) {
    this.mFirstname = mFirstname;
    this.mLastname = mLastname;
  }
}
```

This is pretty neat already. Anything that lets us save on keystrokes is a good thing. I'm guessing you probably have just one thing to nitpick in this example; the method naming isn't right. You probably would prefer `setLastname()` to `setmLastname()`, wouldn't you? You'll fix that in the next section.

Coding Styles

If you go to Android Studio's Preferences or Settings and then go to Editor ➤ Code Style ➤ Java, you'll find that there are plenty of things you can change about how the editor behaves. Figure 13-6 shows the options for the code style, specifically the Java language.

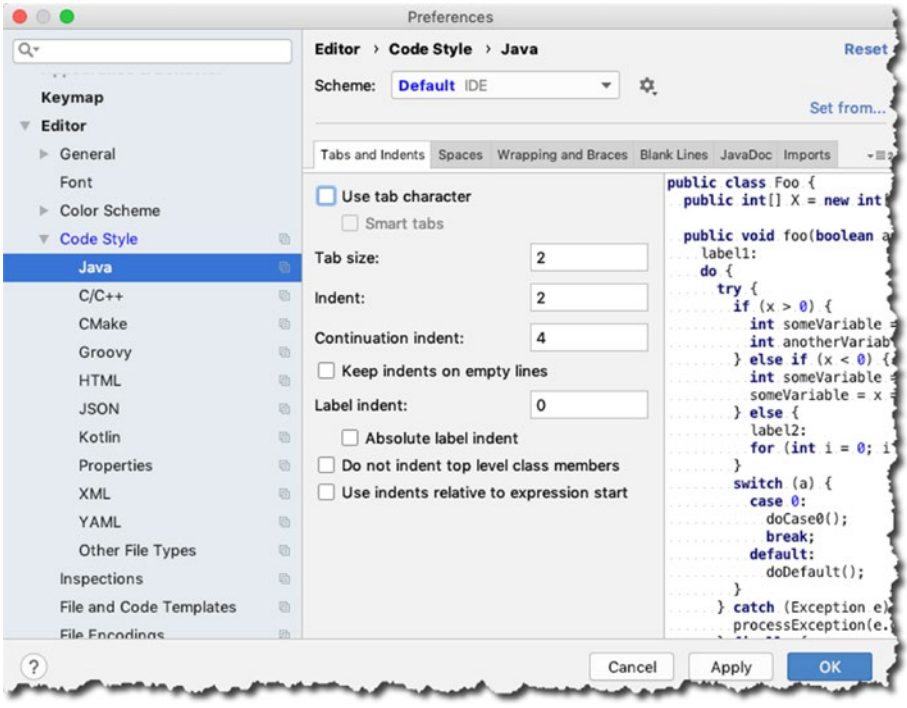

Figure 13-6. Preferences ➤ Code Style ➤ Java

If you want to change the number of spaces for tabs and indents, you can do that in the Tabs and Indents area. Be sure to check out the other options in this dialog.

Go to the Code Generation tab (shown in Figure 13-7).

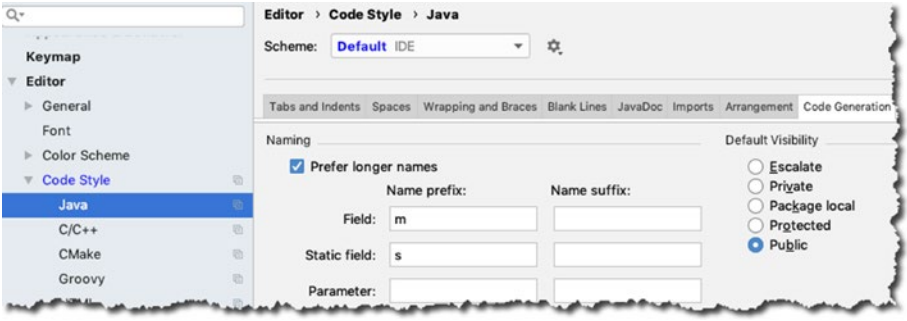

Figure 13-7. *Code generation*

This is where you can tell Android Studio how you name your variables. If you go back to Listing 13-1, you'll notice that I like to prefix my variables with *m,* like `mLastname` and `mFirstname`; initially, Android Studio didn't know this, which is why when I generated some getters and setters for the member vars, it gave me `setmLastname()` instead of just `setLastname()`.

> **Note** Prefixing a member variable with m comes from AOSP (Android Open Source Project). I used it here because quite a lot of sample code you will read online uses this convention. You can read further about it at https://bit.ly/styleguideaosp.

To tell Android Studio that I prefix my variables with *m*, I put the *m* in the Name Prefix for Field, as shown in Figure 13-7, and clicked OK.

Now, if I generate some getters and setters, I'll get the more appropriate method names. Listing 13-2 shows the regenerated code for the `PersonTest` class.

Listing 13-2. *PersonTest, Regenerated*

```
class PersonTest {

  String mFirstname;
  String mLastname;
```

```
  public String getFirstname() {
    return mFirstname;
  }

  public void setFirstname(String firstname) {
    mFirstname = firstname;
  }

  public String getLastname() {
    return mLastname;
  }

  public void setLastname(String lastname) {
    mLastname = lastname;
  }

  public PersonTest(String mFirstname, String mLastname) {
    this.mFirstname = mFirstname;
    this.mLastname = mLastname;
  }
}
```

Live Templates

You can save more time in Android Studio with the live templates, which work a lot like those text expander applications, if you have used some of them. The basic idea is when you type a series of characters, like *datetoday*, the editor will replace them with the text of the actual date today.

If you've done some Android programming in the past, you've probably made this mistake at least once:

```
Toast.makeText(MainActivity.this, "no show");
```

This is easy enough to spot, but some other errors may not be as obvious. Anyway, live templates can help you avoid these hassles. Live templates are shortcuts that are displayed as code-completion options; for example, try typing `fbc` inside an OnClick handler (or any event handler), as shown in Figure 13-8.

Figure 13-8. *Live template sample*

You'll see the code completion options. Press the Tab key and see what happens.

Table 13-1 lists some common built-in templates.

***Table 13-1.** Common Live Templates*

Abbreviation	Description	Code
fbc	Find view by ID with cast	`($cast$) findViewById(R.id.$resId$);`
const	Define an Android style constant	`private static final int $name$ = $value$;`
Toast	Create a new Toast	`Toast.makeText($classname$.this, "$text$").show();`
fori	Create a for loop	`for(int $INDEX$ = 0;$INDEX$<$LIMIT$; $INDEX++$ ) { $END$ }`

Make sure you check out the other live templates. Go to the Settings or Preferences window. If you're in Windows or Linux, go to the main menu bar and choose File ➤ Settings ➤ Editor ➤ Live Templates; if you're in macOS, go to Android Studio ➤ Preferences ➤ Editor ➤ Live Templates. You can even create your own live templates from there.

Important Keyboard Shortcuts

The Android developer website maintains a page where you can find the keyboard shortcuts for Android Studio: http://bit.ly/androidstudiokbshortcuts. You should really make it a point to read this page; but before I close this chapter, I'd like to leave you with six shortcuts that I find to be very useful. Table 13-2 lists these shortcuts.

Table 13-2. Some Useful Keyboard Shortcuts

Shortcut	What it does
Press `Shift` twice	It lets you search for a term everywhere. It searches the `assets` folder, Gradle files, images, resources, code, XML configuration files, and so on. If you don't know which folder to search, just use this.
`Ctrl + Space` | `Command + Space`	Android Studio already has code completion and code hinting; this is just a little extra. If you forgot the parameters for a method that uses lots of parameters, you can use this to preview all the variants of the method and the corresponding parameters they expect.
`Alt + Insert` | `Command + N`	You used this in the previous section where you generated some code. This is the shortcut for the code generator.
`Ctrl + O` | `Command + O`	When you want to override methods, use this shortcut.
`Ctrl + -` | `Command + -`	You can use these to expand or collapse code blocks. It's handy to be able to fold code when you're working with a large codebase. These shortcuts will make your life a bit easier when you fold/unfold blocks.
`Ctrl + Alt + L` | `Command + Option + L`	Don't manually indent or reindent your code. If you messed up the indentation of a for loop or nested conditional blocks, just highlight the code block and use this shortcut.

Chapter Summary

- You can avoid writing boilerplate code like constructors, getters, and setters by using code generators.
- Android Studio has plenty of refactoring aides. Before using the Find/Replace menu, consider using the refactoring options.
- Live templates are like text expanders; they can save you time and let you avoid common coding mistakes. You should use them.
- You can control how the Android Studio editor behaves. Go to Settings or Preferences and then Editor ➤ Code Style.

Index

A, B, C

D

E

T. Hagos, *Android Studio IDE Quick Reference*,
https://doi.org/10.1007/978-1-4842-4953-6

K

L

M

N

O

P, Q

R

S

T

U

V

W, X, Y, Z