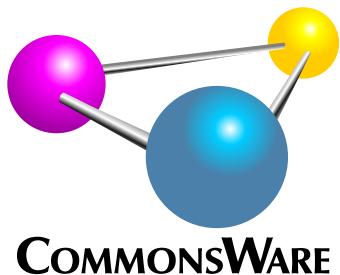


Version 0.5

Elements of Android Jetpack



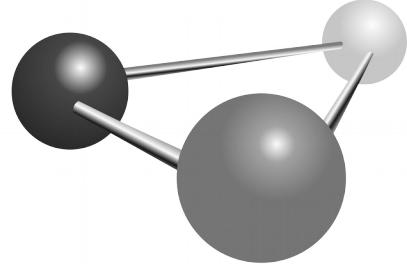
Mark L. Murphy



COMMONSWARE

Elements of Android Jetpack

by Mark L. Murphy



COMMONSWARE

Licensed solely for use by Patrocinio Rodriguez

Elements of Android Jetpack
by Mark L. Murphy

Copyright © 2019 CommonsWare, LLC. All Rights Reserved.
Printed in the United States of America.

Printing History:
September 2019: Version 0.5

The CommonsWare name and logo, “Busy Coder’s Guide”, and related trade dress are trademarks of CommonsWare, LLC.

All other trademarks referenced in this book are trademarks of their respective firms.

The publisher and author(s) assume no responsibility for errors or omissions or for damages resulting from the use of the information contained herein.

Table of Contents

Headings formatted in ***bold-italic*** have changed since the last version.

• <u>Preface</u>	◦ Prerequisites	ix
	◦ About the Updates	ix
	◦ <i>What's New in Version 0.5?</i>	x
	◦ Warescription	x
	◦ Book Bug Bounty	xi
	◦ Source Code and Its License	xii
	◦ <i>Creative Commons and the Four-to-Free (42F) Guarantee</i>	xii
	◦ Acknowledgments	xiii
• <u>Introducing Android</u>	◦ Focus: Apps, Not Operating Systems	1
	◦ <i>What You Need</i>	1
	◦ How This Book Is Organized	5
• <u>Setting Up the Tools</u>	◦ But First, Some Notes About Android's Emulator	7
	◦ <i>Step #1: Install Android Studio</i>	8
	◦ Step #2: Running Android Studio for the First Time	10
• <u>Getting Your First Project</u>	◦ <i>Step #1: Importing the Project</i>	17
	◦ Step #2: Get Ready for the x86 Emulator	19
	◦ <i>Step #3: Set Up the AVD</i>	20
	◦ <i>Step #4: Set Up the Device</i>	24
	◦ <i>Step #5: Running the Project</i>	30
• <u>Taking a Tour of Android Studio</u>	◦ The Project Tree	34
	◦ The Editing Pane	37
	◦ The Docked Views	38
	◦ <i>Popular Menu and Toolbar Options</i>	39
	◦ Android Studio and Release Channels	46
• <u>Examining Your Code</u>	◦ The Top Level	49
	◦ <i>The Project Contents</i>	50
	◦ The App Module Contents	51
	◦ The Generated Source Sets	52
	◦ <i>Language Differences</i>	56

◦ <i>Introducing the Activity</i>	56
◦ Other Things in the Project Tree	60
• <u>Exploring Your Resources</u>	
◦ What You See in res/	64
◦ OS Versions and API Levels	64
◦ Decoding Resource Directory Names	67
◦ Our Initial Resource Types	67
◦ About That R Thingy	73
◦ The Resource Manager	74
• <u>Inspecting Your Manifest</u>	
◦ The Root Element	77
◦ The Application Element	78
◦ The Activity Element (And Its Children)	79
• <u>Reviewing Your Gradle Scripts</u>	
◦ Gradle: The Big Questions	81
◦ Obtaining Gradle	82
◦ Examining the Gradle Files	84
◦ Requesting Plugins	89
◦ Android Plugin for Gradle Configuration	89
◦ Other Stuff in the android Closure	92
◦ Libraries and Dependencies	92
• <u>Inspecting the Compiled App</u>	
◦ What We Build	95
◦ Where They Go	96
◦ Building the APK	97
◦ Analyzing the APK	98
• <u>Touring the Tests</u>	
◦ Instrumented Tests	101
◦ Unit Tests	112
• <u>Introducing Jetpack</u>	
◦ What, Exactly, is Jetpack?	117
◦ Um, OK, So, What's the Point?	117
◦ Key Elements of Jetpack	118
◦ What Came Before: the Android Support Library	123
• <u>Introducing the Sampler Projects</u>	
◦ The Projects	125
◦ Getting a Sampler Project	126
◦ The Modules	128
◦ Running the Samples	128
• <u>Starting Simple: TextView and Button</u>	
◦ First, Some Terminology	129

◦ <i>Introducing the Graphical Layout Editor</i>	135
◦ <i>TextView: Assigning Labels</i>	143
◦ <i>Button: Reacting to Input</i>	149
◦ The Curious Case of the Missing R	157
• <u>Debugging Your App</u>	
◦ Get Thee To a Stack Trace	162
◦ Running Your App in the Debugger	165
◦ <i>So, Where Did We Go Wrong?</i>	171
• <u>Introducing ConstraintLayout</u>	
◦ The Role of Containers	174
◦ Layouts and Adapter-Based Containers	175
◦ ConstraintLayout: One Layout To Rule Them All	176
◦ Getting ConstraintLayout	177
◦ Using Widgets and Containers from Libraries	177
◦ A Quick RTL Refresher	179
◦ <i>Simple Rows with ConstraintLayout</i>	180
◦ Starting from Scratch	184
◦ ConstraintLayout and the Attributes Pane	186
◦ EditText: Making Users Type Stuff	186
◦ More Complex Forms	188
◦ Turning Back to RTL	194
◦ More Fun with ConstraintLayout	196
◦ Notes on the Classic Containers	197
• <u>Integrating Common Form Widgets</u>	
◦ ImageView and ImageButton	201
◦ Compound Buttons	206
◦ SeekBar	219
◦ ScrollView: Making It All Fit	222
◦ <i>Other Notes About the Sample</i>	225
• <u>Contemplating Contexts</u>	
◦ It's Not an OMG Object, But It's Close	233
◦ <i>The Major Types of Context</i>	234
◦ Key Context Features	236
◦ Know Your Context	238
◦ Context Anti-Patterns	238
• <u>Icons</u>	
◦ App Icons... And Everything Else	241
◦ Creating an App Icon with the Asset Studio	242
◦ Creating Other Icons with the Asset Studio	248
• <u>Adding Libraries</u>	
◦ <i>Depending on a Local JAR</i>	249

◦ <i>Artifacts and Repositories</i>	250
◦ <i>Requesting Dependencies</i>	251
• <u>Employing RecyclerView</u>	
◦ <i>Recap: Layouts vs. Adapter-Based Containers</i>	255
◦ The Challenge: Memory	256
◦ Enter RecyclerView	257
◦ <i>A Trivial List</i>	258
◦ <i>Hey, What About ListView?</i>	275
• <u>Coping with Configurations</u>	
◦ What’s a Configuration? And How Do They Change?	277
◦ Configurations and Resource Sets	278
◦ <i>Implementing Resource Sets</i>	279
◦ Resource Set Rules	282
◦ Activity Lifecycles	286
◦ When Activities Die	290
◦ Context Anti-Pattern: Outliving It	291
• <u>Integrating ViewModel</u>	
◦ Configuration Changes	293
◦ What We Want... and What We Do Not Want	294
◦ Enter the ViewModel	295
◦ <i>Applying ViewModel</i>	295
◦ <i>ViewModel and the Lifecycle</i>	299
◦ <i>Changing Data in the ViewModel</i>	303
◦ ViewModel and AndroidViewModel	316
• <u>Understanding Processes</u>	
◦ When Processes Are Created	319
◦ <i>What Is In Your Process</i>	320
◦ <i>BACK, HOME, and Your Process</i>	320
◦ Termination	321
◦ <i>Foreground Means “I Love You”</i>	322
◦ Tasks and Your App	323
◦ Instance State	325
◦ <i>Pondering Parcelable</i>	327
◦ <i>A State-Aware ViewModel</i>	333
• <u>Binding Your Data</u>	
◦ <i>The Basic Steps</i>	341
◦ Why Bother?	353
• <u>Defining and Using Styles</u>	
◦ Styles: DIY DRY	355
◦ Elements of Style	356
◦ <i>Themes: Would a Style By Any Other Name...</i>	360

◦ Context Anti-Pattern: Using Application Everywhere	365
• <u>Configuring the App Bar</u>	
◦ So. Many. Bars.	368
◦ Vector Drawables	370
◦ <i>Menu Resources</i>	375
◦ <i>Using Toolbar Directly</i>	380
◦ <i>Using Toolbar as the Action Bar</i>	393
◦ Having Fun at Bars	396
• <u>Implementing Multiple Activities</u>	
◦ Multiple Activities, and Your App	397
◦ Creating Your Second (and Third and...) Activity	398
◦ Starting Your Own Activity	400
◦ Extra! Extra!	401
◦ <i>Seeing This In Action</i>	401
◦ Using Implicit Intents	406
◦ <i>Asynchronicity and Results</i>	408
◦ The Inverse: <intent-filter>	418
• <u>Adopting Fragments</u>	
◦ The Six Questions	421
◦ Where You Get Your Fragments From	425
◦ Static vs. Dynamic Fragments	426
◦ Fragments, and What You Have Seen Already	427
◦ <i>ToDo, or Not ToDo? That Is the Question</i>	427
◦ The Fragment Lifecycle Methods	456
◦ Context Anti-Pattern: Assuming Certain Types	459
• <u>Navigating Your App</u>	
◦ What We Get from the Navigation Component	461
◦ <i>Elements of Navigation</i>	465
◦ <i>A Navigation-ized To-Do List</i>	470
◦ So... Was It Worth It?	481
• <u>Thinking About Threads and LiveData</u>	
◦ The Main Application Thread	483
◦ The UI Thread is for UI	485
◦ <i>Introducing LiveData</i>	486
◦ <i>Colors... Live!</i>	488
◦ <i>Sources of Owners</i>	496
◦ <i>Where Do Threads Come From? Um, Besides From Me?</i>	497
◦ <i>Coroutines and ViewModel</i>	499
• <u>Adding Some Architecture</u>	
◦ GUI Architectures	501
◦ Why People Worry About GUI Architectures	502

◦ <i>Why Others Ignore GUI Architectures</i>	504
◦ Repositories	506
◦ Unidirectional Data Flow	507
◦ <i>A UDF Implementation</i>	508
• <u>Working with Content</u>	
◦ <i>The Storage Access Framework</i>	529
• <u>Using Preferences</u>	
◦ The Preferred Preferences	543
◦ <i>Collecting Preferences with PreferenceFragmentCompat</i>	544
◦ Types of Preferences	550
◦ Working with SharedPreferences	553
• <u>Requesting Permissions</u>	
◦ Frequently-Asked Questions About Permissions	559
◦ <i>Dangerous Permissions: Request at Runtime</i>	562
• <u>Handling Files</u>	
◦ <i>The Three Types of File Storage</i>	573
◦ What the User Sees	576
◦ <i>Storage, Permissions, and Access</i>	576
◦ <i>Reading, Writing, and Debugging Storage</i>	577
◦ <i>Serving Files with FileProvider</i>	596
◦ What You Should Use	606
• <u>Accessing the Internet</u>	
◦ An API Roundup	607
◦ <i>Android's Restrictions</i>	611
◦ <i>Forecasting the Weather</i>	613
• <u>Storing Data in a Room</u>	
◦ <i>Room Requirements</i>	634
◦ <i>Room Furnishings</i>	634
◦ <i>Other Fun Stuff in the App</i>	641
◦ What Else Does Room Offer?	647
• <u>Inverting Your Dependencies</u>	
◦ <i>The Problem: Test Control</i>	649
◦ <i>The Solution: Dependency Inversion</i>	650
◦ <i>Dependency Inversion in Android</i>	651
◦ <i>Applying Koin</i>	651
• <u>Testing Your Changes</u>	
• <u>Working with WorkManager</u>	
◦ <i>The Role of WorkManager</i>	665
◦ <i>WorkManager Dependencies</i>	666
◦ <i>Workers: They Do Work</i>	667
◦ <i>Performing Simple Work</i>	670

◦ <i>Work Inputs</i>	671
◦ <i>Constrained Work</i>	672
◦ <i>Tagged Work</i>	673
◦ <i>Monitoring Work</i>	674
◦ <i>Cancelling Work</i>	682
◦ <i>Delayed Work</i>	683
◦ <i>Parallel Work</i>	683
◦ <i>Chained Work</i>	684
◦ <i>Periodic Work</i>	694
◦ <i>Unique Work</i>	694
◦ <i>Testing Work</i>	695
◦ <i>WorkManager and Side Effects</i>	698
• <u>Creating a New Project</u>	
• <u>Shrinking Your App</u>	
• <u>Using the SDK Manager</u>	
• <u>Using the AVD Manager and the Emulator</u>	
• <u>Configuring Your Project</u>	
• <u>Configuring Android Studio</u>	

Licensed solely for use by Patrocinio Rodriguez

Preface

Thanks!

Thanks for your interest in developing applications for Android! Android has grown from nothing to the world's most popular smartphone OS in a few short years. Whether you are developing applications for the public, for your business or organization, or are just experimenting on your own, I think you will find Android to be an exciting and challenging area for exploration.

And, most of all, thanks for your interest in this book! I sincerely hope you find it useful and at least occasionally entertaining.

Prerequisites

This book is written for developers with prior experience in Java or Kotlin. If you are not familiar with either of these languages, it will be difficult for you to follow the code samples in the book. The author of this book has published [*Elements of Kotlin*](#). If you obtained this book via [the Warescription](#), you are eligible to download a copy of *Elements of Kotlin*, along with the rest of the CommonsWare line of books.

This book is written for people who have used an Android device before. If you are not familiar with basic Android concepts — such as the home screen and launcher, the HOME button, the BACK button, and so on — you will want to spend time with an Android device.

About the Updates

This book will be updated a few times per year, to reflect newer versions of Android,

PREFACE

Android Studio, and the Jetpack family of libraries.

If you obtained this book through [the Warescription](#), you will be able to download updates as they become available, for the duration of your subscription period.

If you obtained this book through other channels... um, well, it's still a really nice book!

Each release has notations to show what is new or changed compared with the immediately preceding release:

- The Table of Contents in the ebook formats (PDF, EPUB, MOBI/Kindle) shows sections with changes in ***bold-italic*** font
- Those sections have changebars on the right to denote specific paragraphs that are new or modified

And, there is the “What’s New” section, just below this paragraph.

What’s New in Version 0.5?

This update adds two new chapters:

- [“Inverting Your Dependencies”](#)
- ["Working with WorkManager"](#)

It also updates everything for Android Studio 3.5, updates many of the dependencies used in the book, and fixes lots of bugs.

Warescription

If you purchased the Warescription, read on! If you obtained this book from other channels, feel free to [jump ahead](#).

The Warescription entitles you, for the duration of your subscription, to digital editions of this book and its updates, in PDF, EPUB, and Kindle (MOBI/KF8) formats, plus the ability to read the book online at [the Warescription Web site](#). You also have access to other books that CommonsWare publishes during that subscription period.

Each subscriber gets personalized editions of all editions of each title. That way,

PREFACE

your books are never out of date for long, and you can take advantage of new material as it is made available.

However, you can only download the books while you have an active Warescription. There is a grace period after your Warescription ends: you can still download the book until the next book update comes out after your Warescription ends. After that, you can no longer download the book. Hence, **please download your updates as they come out**. You can find out when new releases of this book are available via:

1. The [CommonsBlog](#)
2. The [CommonsWare](#) Twitter feed
3. Opting into emails announcing each book release — log into the [Warescription](#) site and choose Configure from the nav bar
4. Just check back on the [Warescription](#) site every month or two

Subscribers also have access to other benefits, including:

- “Office hours” — online chats to help you get answers to your Android application development questions. You will find a calendar for these on your Warescription page.
- A Stack Overflow “bump” service, to get additional attention for a question that you have posted there that does not have an adequate answer.
- A discussion board for asking arbitrary questions about Android app development.

Book Bug Bounty

Find a problem in the book? Let CommonsWare know!

Be the first to report a unique concrete problem in the current edition, and CommonsWare will extend your Warescription by six months as a bounty for helping CommonsWare deliver a better product.

By “concrete” problem, we mean things like:

1. Typographical errors
2. Sample applications that do not work as advertised, in the environment described in the book
3. Factual errors that cannot be open to interpretation

PREFACE

By “unique”, we mean ones not yet reported. Be sure to check [the book’s errata page](#), though, to see if your issue has already been reported. One coupon is given per email containing valid bug reports.

We appreciate hearing about “softer” issues as well, such as:

1. Places where you think we are in error, but where we feel our interpretation is reasonable
2. Places where you think we could add sample applications, or expand upon the existing material
3. Samples that do not work due to “shifting sands” of the underlying environment (e.g., changed APIs with new releases of an SDK)

However, those “softer” issues do not qualify for the formal bounty program.

Questions about the bug bounty, or problems you wish to report for bounty consideration, should be sent to bounty@commonsware.com.

Source Code and Its License

The source code in this book is licensed under the [Apache 2.0 License](#), in case you have the desire to reuse any of it.

Copying source code directly from the book, in the PDF editions, works best with Adobe Reader, though it may also work with other PDF viewers. Some PDF viewers, for reasons that remain unclear, foul up copying the source code to the clipboard when it is selected.

Creative Commons and the Four-to-Free (42F) Guarantee

Each CommonsWare book edition will be available for use under the [Creative Commons Attribution-Noncommercial-ShareAlike 3.0](#) license as of the fourth anniversary of its publication date, or when 4,000 copies of the edition have been sold, whichever comes first. That means that, once four years have elapsed (perhaps sooner!), you can use this prose for non-commercial purposes. That is our Four-to-Free Guarantee to our readers and the broader community. For the purposes of this guarantee, new Warescriptions and renewals will be counted as sales of this edition, starting from the time the edition is published.

PREFACE

This edition of this book will be available under the aforementioned Creative Commons license on *1 September 2023*. Of course, watch the CommonsWare Web site, as this edition might be relicensed sooner based on sales.

For more details on the Creative Commons Attribution-Noncommercial-ShareAlike 3.0 license, visit [the Creative Commons Web site](#)

Note that future editions of this book will become free on later dates, each four years from the publication of that edition or based on sales of that specific edition. Releasing one edition under the Creative Commons license does not automatically release *all* editions under that license.

Acknowledgments

The author would like to thank the Google team responsible for Android and the Android Jetpack.

Licensed solely for use by Patrocinio Rodriguez

Introducing Android

No doubt, you are in a hurry to get started with Android application development. After all, you are reading this book, aimed at busy coders.

However, before we dive into getting tools set up and starting in on actual programming, it is important that we “get on the same page” with respect to several high-level Android concepts. This will simplify further discussions later in the book.

Focus: Apps, Not Operating Systems

This book is focused on writing Android applications (“apps”). An app is something that a user might install from the Play Store or otherwise download to their device. That app usually has some user interface, and it might have other code designed to work in the background.

This book is not focused on modifications to the Android firmware, such as writing device drivers. For that, you will need to seek [other resources](#).

What You Need

In order to get started as an Android developer — and to get the most out of this book — you are going to need several things, outlined in the following sections.

A Development Machine

For the purposes of this book, you will need a Windows, macOS, or Linux computer on which to write your Android apps. This is how the vast majority of Android app developers do their work, though there are tools (e.g., AIDE) that allow you to write

INTRODUCING ANDROID

Android apps directly on an Android device.

Your development machine should be as powerful as you can manage:

- A fast CPU (e.g., quad-core Intel Core i5/i7 with at least 2.0 GHz clock speed per core)
- As much RAM as you can manage (8GB minimum)
- As fast of a hard drive as you can find (an SSD is an excellent choice in general)
- A screen with enough resolution to use the development tools (1280x800 minimum resolution)

Language Experience

In general, to write Android apps, you need to know how to work with computer programming languages. In particular, Android app development is focused heavily on Java and Kotlin, with Groovy also playing a role.

Java

The original programming language used for Android app development was Java. Right now, most Android code in the world is written in Java, and most educational material is written around Java.

As a result, to be an Android app developer today, it helps to know Java.

This book does not teach you Java. Java has been around for around two decades, and so there are *lots* of existing books, courses, videos, and the like to help you learn Java. However, there are many things in Java that are not really relevant for Android app development, such as Swing desktop GUIs and Java servlets for Web applications. You do not need to know *everything* about Java, as Java is vast. Rather, focus on:

- [Language fundamentals](#) (flow control, etc.)
- [Classes and objects](#)
- [Methods and data members](#)
- [Public, private, and protected](#)
- [Static and instance scope](#)
- [Exceptions](#)
- [Threads](#)
- [Collections](#)

INTRODUCING ANDROID

- [Generics](#)
- [File I/O](#)
- [Reflection](#)
- [Interfaces](#)

The links are to Wikibooks material on those topics, though there are countless other Java resources for you to consider.

Kotlin

The primary current language for Android app development is Kotlin. Kotlin is a fairly new language, having only reached 1.0 status in 2016. That causes some problems, as there is less material about how to write Kotlin than there is on how to write Java. On the other hand, Kotlin adopts newer approaches and discards legacy “cruft”. The resulting language can be much more concise, getting more work done with fewer lines of code.

This book does not teach you Kotlin. The author of this book is also the author of [*Elements of Kotlin*](#), which was written with an eye towards it being a companion to the book that you are reading now. From time to time, you will find this book pointing out relevant chapters and sections in *Elements of Kotlin*, to help newcomers to both Android *and* Kotlin learn both subjects.

Note that at the 2019 Google I/O conference, Google indicated that the Android SDK will be “Kotlin first” going forward. While Java development is still possible, Google will be focusing on Kotlin in terms of documentation, samples, education, and some new technologies. So, while this book will present material in both Java and Kotlin, you should strongly consider learning Kotlin in the not-too-distant future.

Groovy and Gradle

The code that causes your app to do stuff will be written mostly in Java and Kotlin.

The code that causes your app to be *built* out of that Java and Kotlin will be written in Groovy... though you may not notice this much.

Most Android apps are built using a build tool called Gradle. Gradle is a program for building other programs. We will be working with “Gradle build scripts” to configure how Gradle turns our source code into a (hopefully) working app. The Gradle build scripts that we use today usually are written using the Groovy

programming language.

However, for most basic uses of Gradle — including pretty much everything in this book — you will not need to think much about Groovy syntax. Just follow the recipes described in the book, and you can put off learning Groovy until such time as you really want to start creating elaborate build scripts.

An Android Test Environment

Writing Android apps is fun!

(no, really!)

However, that fun only appears when you can actually run the app that you created. Otherwise, you just have a hunk of source code that sits around doing nothing. To run the app, you will need an Android device or emulator.

Devices

Every Android developer should have at least one Android device. Every Android device that legitimately has the Play Store on it is able to be used for app development. You can enable the super-secret “developer options” in the device, to allow you to install apps that you have written yourself on the device and test them out — we will see how to do that in this book.

Typically, that Android device will be a phone, though you could test on something else, such as a tablet, if you wish. Android app development puts few requirements on the device itself; for example, you do not necessarily need to have a usable SIM installed in the phone.

In [an upcoming chapter](#), you will see how to configure your Android device for use with app development.

Emulators

All Android developers should have at least one device. Some Android developers, such as the author of this book, have lots of Android devices. However, inevitably, you run into cases where hardware is a problem:

- You want to test your app on different versions of Android, but you do not

INTRODUCING ANDROID

- have a device for a particular Android version
- You want to test your app for various screen sizes and resolutions, but you do not have devices for all of the scenarios that you wish to test
- You want to test your app in unusual situations, such as running on a Chromebook, and you do not have a device that matches

For those cases, the Android tools come with an emulator. The emulator gives you an app for your development machine that pretends to be an Android device. You decide what sort of device it is: Android version, screen size and resolution, and so on. You can run your app on the emulator and get a sense for what it would be like for the app to be running on a real device with those same characteristics.

In [an upcoming chapter](#), you will see how to set up the Android emulator.

Patience and Serenity

Android app development often can be a frustrating experience:

- Advice that you get from older sources may not work, due to changes in Android
- Dealing with multiple programming languages makes it more difficult to make use of advice that you get, if you have to keep converting code snippets between languages
- The GUI that you wrote that works fine on one device does not work quite as well on the next device
- And so on

You will be able to address all of these challenges in time. Early on, though, you should expect that these sorts of problems will arise, and you will need to cope with them when they do.

How This Book Is Organized

This book is divided into two major parts: a “Hello, World” walk-through and “deeper dives” into major Android app development topics.

“Hello, World!”, Front to Back

For decades, the classic first program for a person in a given programming environment is dubbed [“hello, world”](#).

INTRODUCING ANDROID

In a typical programming environment, a “hello, world!” app is usually trivial, offering little to learn from.

Android is complicated, which makes even “hello, world!” a place to learn all sorts of things, from how user interfaces get constructed, to how our tools work, to how the tools know how to build that app and show us that user interface.

So, in the first several chapters, we will examine various facets of a “hello, world!” app generated from Android Studio’s new-project wizard.

Deeper Dives

Of course, a “hello, world!” app is very shallow. You will not get very far in Android app development if all you know is what “hello, world!” shows you.

So, after the tour of the “hello, world!” app, we will expand upon the concepts seen there, exploring different aspects of Android app development. We will not cover *everything* in Android — that would take thousands upon thousands of pages. This book will give you a basic foundation for Android app development and will help point you to places to learn other facets of what Android apps can do.

Setting Up the Tools

Now, let us get you set up with the pieces and parts necessary to build an Android app.

NOTE: The instructions presented here are accurate as of the time of this writing. However, the tools change rapidly, and so these instructions may be out of date by the time you read this. Please refer to the [Android Developers Web site](#) for current instructions, using this as a base guideline of what to expect.

But First, Some Notes About Android's Emulator

As noted in the previous chapter, the Android tools include an emulator, a piece of software that pretends to be an Android device. This is very useful for development — not only does it mean you can get started on Android without a device, but the emulator can help test device configurations that you do not own.

Emulators not only emulate Android itself, but also the CPU of the Android device. Most Android devices have ARM CPUs... but it is likely that your development machine has an x86 CPU. The emulator can emulate an ARM CPU when running on your x86 CPU, but it is slow. Fortunately, the emulator can also emulate an Android device that has an x86 CPU, and this runs much more quickly. You *really* want to be able to use the x86 emulator.

However, to use the x86 emulator, you will need some additional software:

- Linux users need KVM
- macOS and Windows users need the “Intel Hardware Accelerated Execution Manager” (a.k.a., HAXM), which the Android Studio installer will attempt to install for you

SETTING UP THE TOOLS

And, the x86 emulator will only work for CPUs meeting certain requirements:

Development OS	CPU Manufacturer	CPU Requirements
mac OS	Intel	any modern Mac should work
Linux/ Windows	Intel	support for Intel VT-x, Intel EM64T (Intel 64), and Execute Disable (XD) Bit functionality
Linux	AMD	support for AMD Virtualization (AMD-V) and Supplemental Streaming SIMD Extensions 3 (SSSE3)
Windows 10 April 2018 or newer	AMD	support for Windows Hypervisor Platform (WHPX) functionality

If your CPU does not meet those requirements, you will want to have one or more Android devices available to you, so that you can test on hardware rather than the emulator.

If you are running Windows or Linux, you need to ensure that your computer's BIOS is set up to support "virtualization extensions". Unfortunately, many PC manufacturers disable this by default. The details of how to get into your BIOS settings will vary by PC, but usually it involves rebooting your computer and pressing some function key on the initial boot screen. In the BIOS settings, you are looking for references to "virtualization". Enable them if they are not already enabled. macOS machines come with virtualization extensions pre-enabled.

Part of the Android Studio installation process will try to set you up to be able to use the x86 emulator. Make note of any messages that you see in the installation wizard regarding "HAXM" (or, if you are running Linux, KVM), as those will be important later.

Step #1: Install Android Studio

At the time of this writing, the current production version of Android Studio is 3.5, and this book covers that version. If you are reading this in the future, you may be on a newer version of Android Studio, and there may be some differences between

SETTING UP THE TOOLS

what you have and what is presented here.

You have two major download options. You can get the latest shipping version of Android Studio from [the Android Studio download page](#).

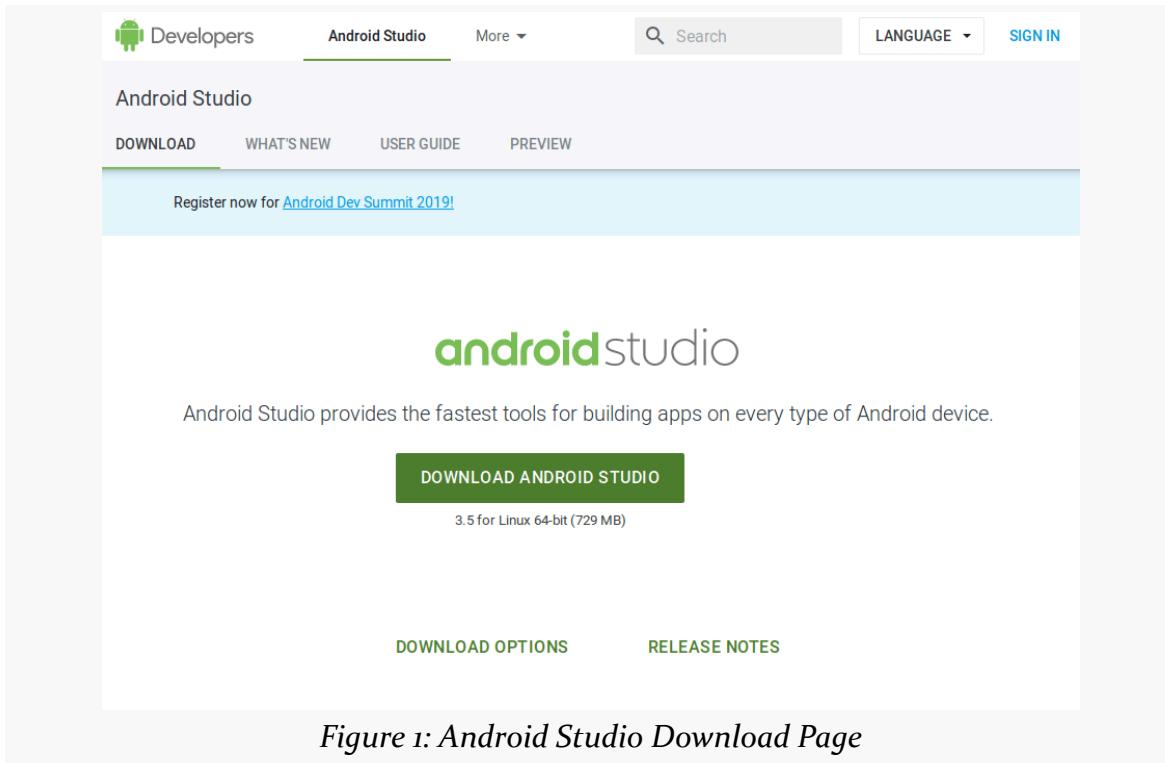


Figure 1: Android Studio Download Page

Or, you can download Android Studio 3.5 directly, for:

- [Windows](#)
- [macOS](#)
- [Linux](#)

Windows users can download a self-installing EXE, which will add suitable launch options for you to be able to start the IDE.

macOS users can download a DMG disk image and install it akin to other macOS software, dragging the Android Studio icon into the Applications folder.

Linux users (and power Windows users) can download a ZIP file, then unZIP it to some likely spot on your hard drive. Android Studio can then be run from the `studio` batch file or shell script from your Android Studio installation's `bin/`

SETTING UP THE TOOLS

directory.

Step #2: Running Android Studio for the First Time

When you first run Android Studio, you may be asked if you want to import settings from some other prior installation of Android Studio:

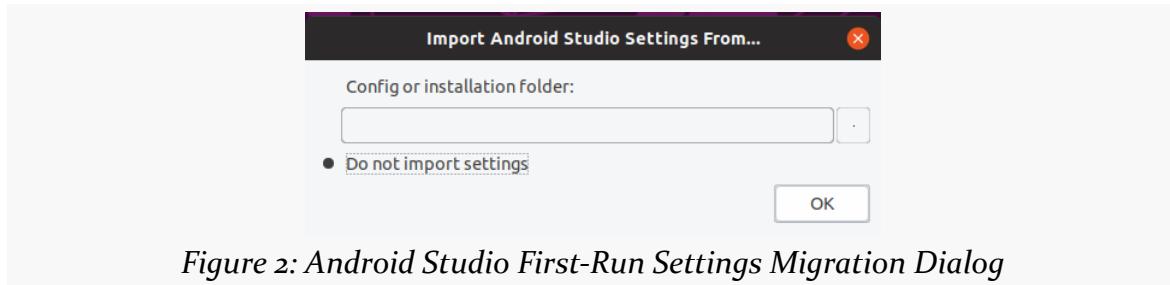


Figure 2: Android Studio First-Run Settings Migration Dialog

If you are using Android Studio for the first time, the “Do not import settings” option is the correct choice to make.

Then, after a short splash screen, you may be presented with a “Data Sharing” dialog:

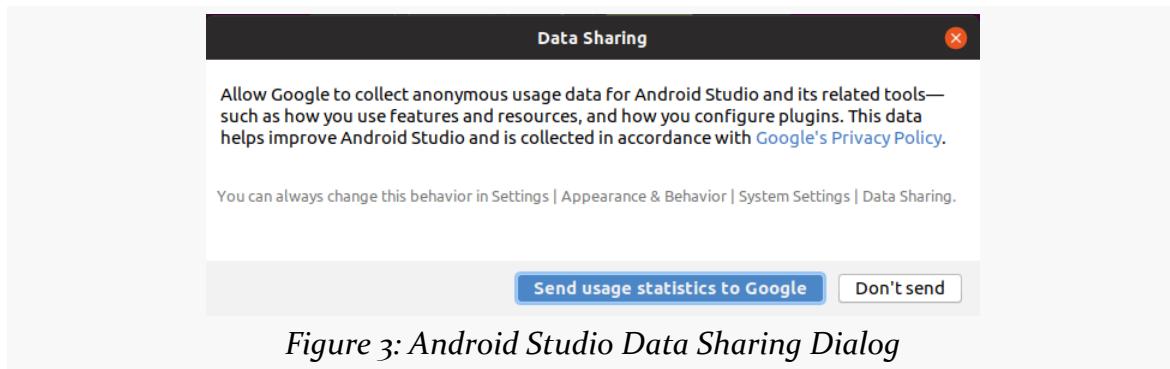


Figure 3: Android Studio Data Sharing Dialog

Click whichever button you wish.

SETTING UP THE TOOLS

Then, after a potentially long “Finding Available SDK Components” progress dialog, you will be taken to the Android Studio Setup Wizard:

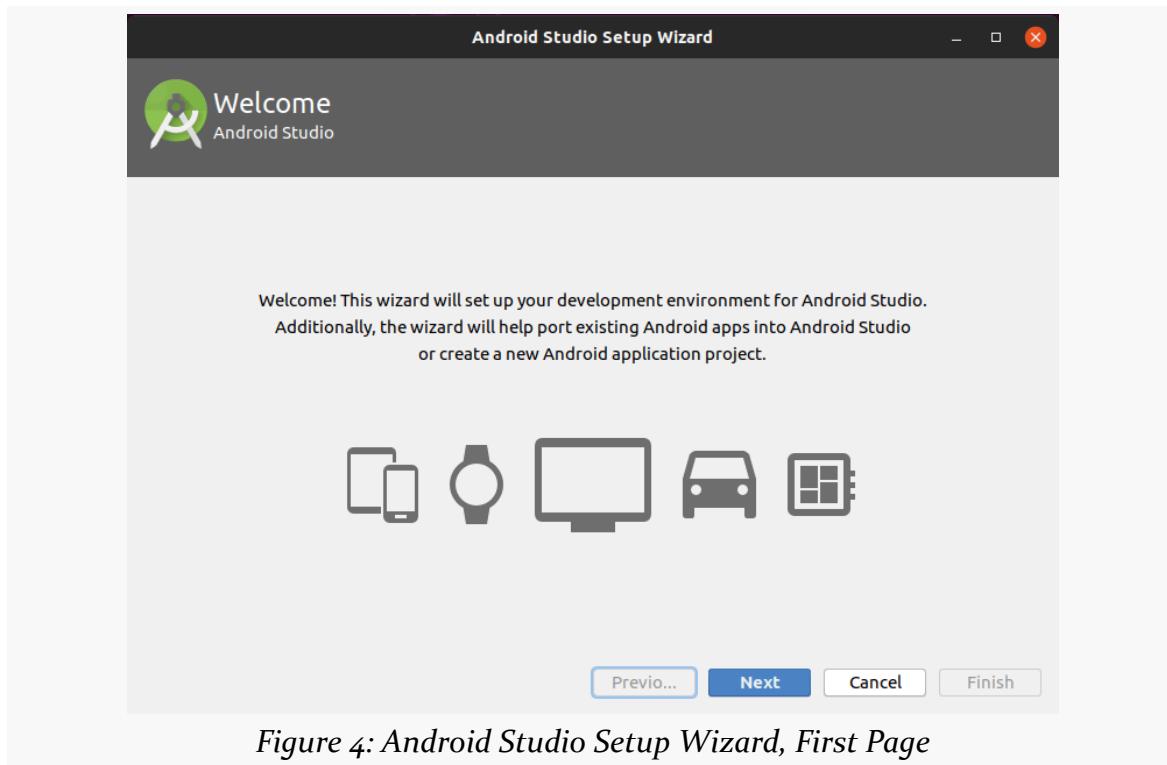


Figure 4: Android Studio Setup Wizard, First Page

SETTING UP THE TOOLS

Just click “Next” to advance to the second page of the wizard:

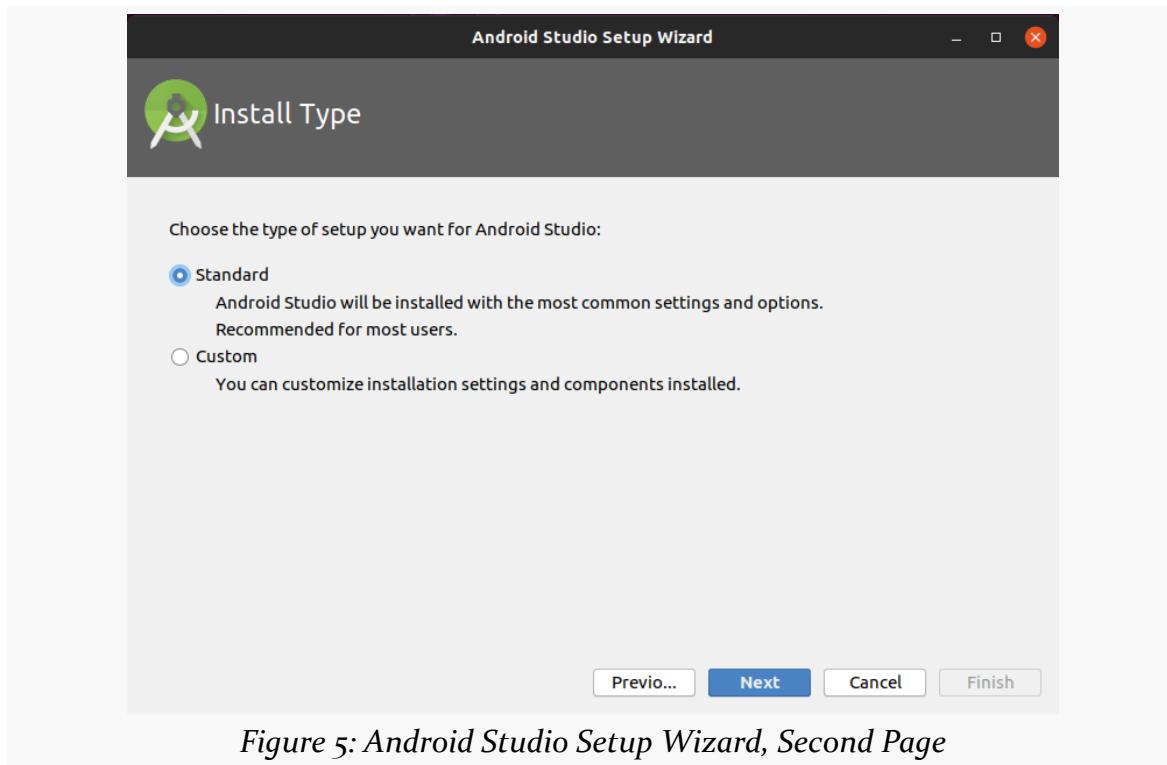


Figure 5: Android Studio Setup Wizard, Second Page

Here, you have a choice between “Standard” and “Custom” setup modes. Most likely, right now, the “Standard” route will be fine for your environment.

SETTING UP THE TOOLS

If you go the “Standard” route and click “Next”, you should be taken to a wizard page where you can choose your UI theme:

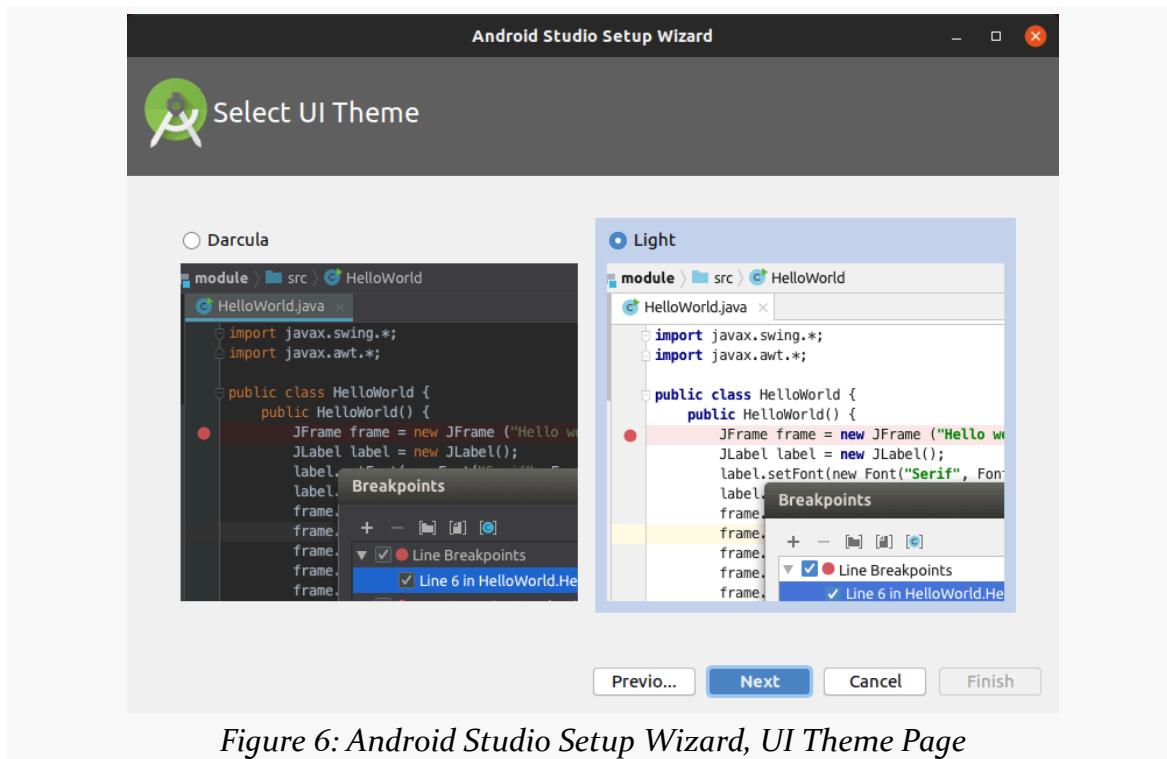


Figure 6: Android Studio Setup Wizard, UI Theme Page

SETTING UP THE TOOLS

Choose whichever you like, then click “Next”, to go to a wizard page to verify what will be downloaded and installed:

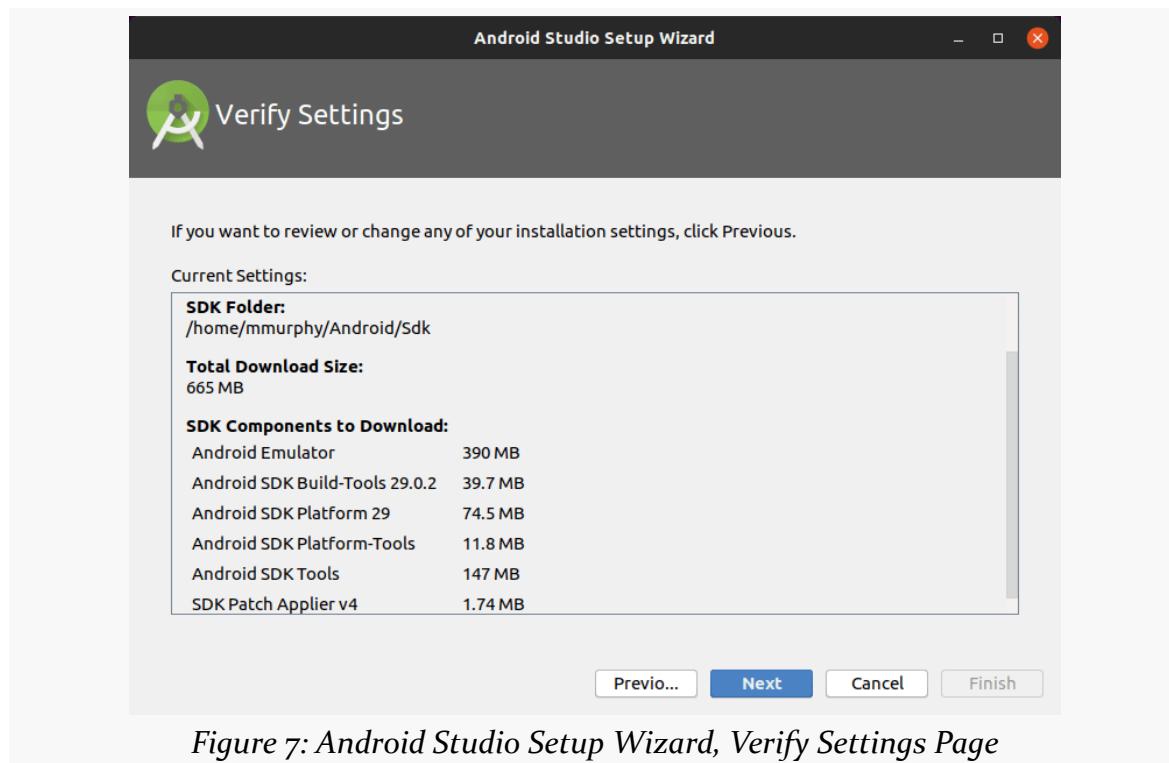


Figure 7: Android Studio Setup Wizard, Verify Settings Page

SETTING UP THE TOOLS

Clicking “Next” may take you to a wizard page explaining some information about the Android emulator:

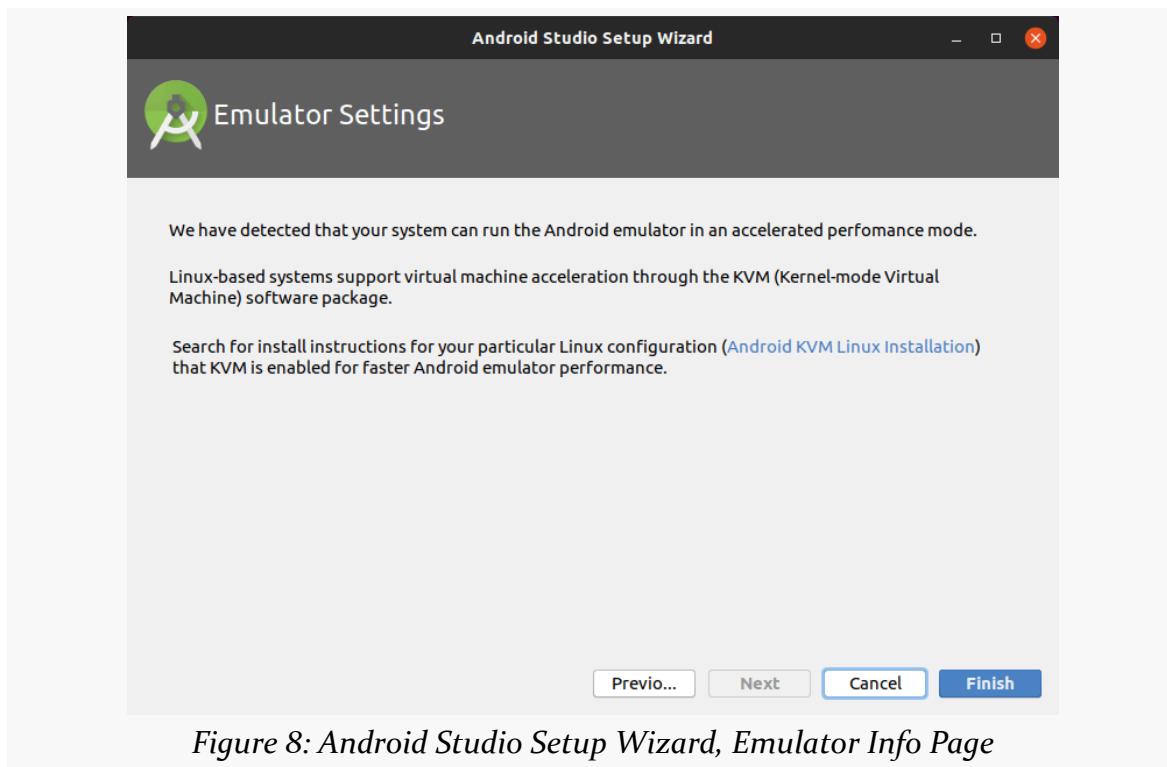


Figure 8: Android Studio Setup Wizard, Emulator Info Page

What is explained on this page may not make much sense to you. That is perfectly normal, and we will get into what this page is trying to say later in the book. Just click “Finish” to begin the setup process. This will include downloading a copy of the Android SDK and installing it into a directory adjacent to where Android Studio itself is installed.

When that is done, Android Studio will busily start downloading stuff to your development machine.

SETTING UP THE TOOLS

Clicking “Finish” when that is done will then take you to the Android Studio Welcome dialog:

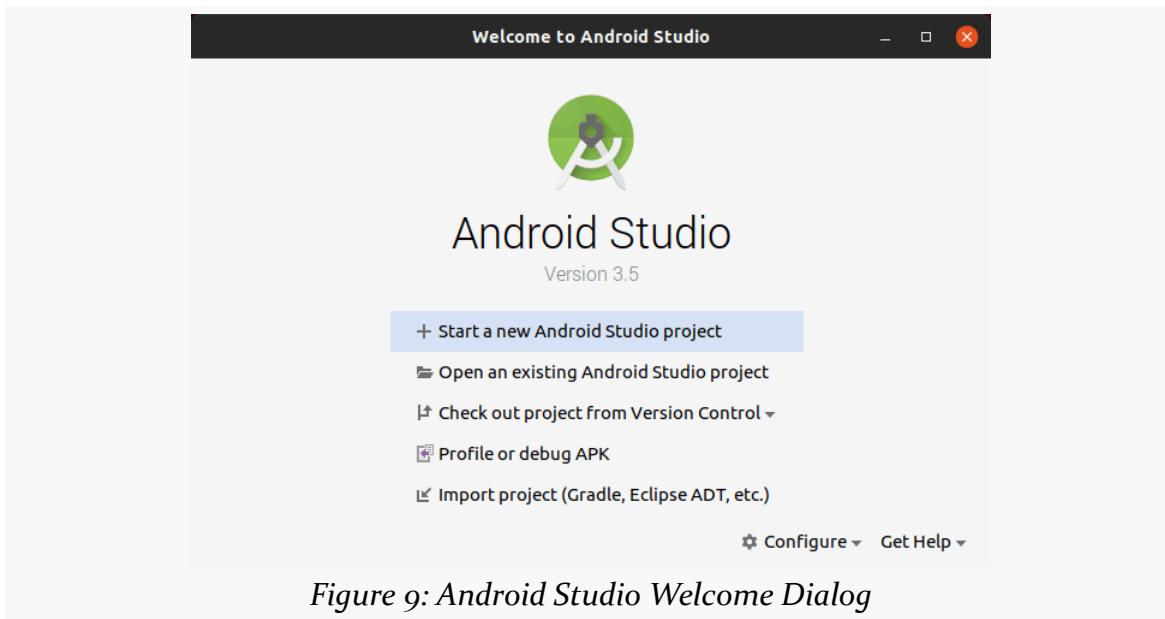


Figure 9: Android Studio Welcome Dialog

Getting Your First Project

Creating an Android application first involves an Android “project”. As with many other development environments, the project is where your source code and other assets (such as icons) reside. And, the project contains the instructions for your tools for how to convert that source code and other assets into an Android app.

So, let’s work on setting up a “hello, world!” application to examine.

As with the rest of this book, these instructions are for Android Studio 3.5. If you are using a newer or older version of this IDE, while the instructions are likely to be close to what you will see, there will be differences.

Step #1: Importing the Project

Roughly speaking, there are two ways to start with a project with Android Studio: creating a new project or importing an existing project.

It might sound like creating a new project would be the more common scenario. In truth, many developers import an existing project, because they are working on a development team, and somebody else on the team created the project. Often, that project was created quite some time ago, with developers coming and going from the team.

So, while we will see how to create a project later in the book, for now, let’s import an existing project, one set up for use by this book. It will closely resemble the sort of project that you get when creating a brand-new project in Android Studio.

You can download this project from [the CommonsWare site](#). Then, unZIP that project to some place on your development machine. It will unZIP into an

GETTING YOUR FIRST PROJECT

HelloWorld/ directory.

Then, import the project. From the Android Studio welcome dialog — where we ended up at the end of the previous chapter — you can import a project via the “Import project (Eclipse ADT, Gradle, etc.)” option. If you already have a project open in Android Studio, you can import a project via File > New > Import Project... from the main menu.

Importing a project brings up a typical directory-picker dialog. Pick the HelloWorld/ directory and click OK to begin the import process. This may take a while, depending on the speed of your development machine. A “Tip of the Day” dialog may appear, which you can dismiss.

At this point, you should have a mostly-empty Android Studio IDE window:

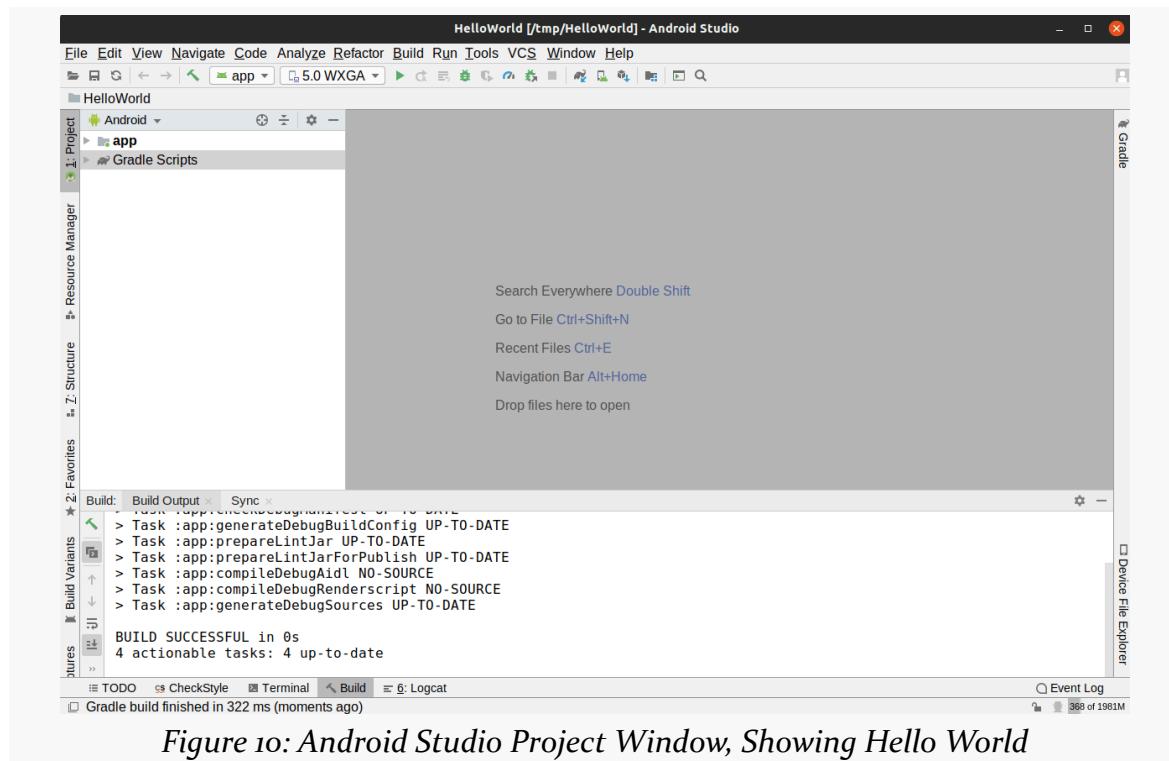


Figure 10: Android Studio Project Window, Showing Hello World

We will examine what is all in this window coming up in future chapters. But, first, let's get things set up for you to be able to run this sample app and see its results.

Step #2: Get Ready for the x86 Emulator

Your next decision to make is whether or not you want to bother setting up an emulator image right now. If you have an Android device, you may prefer to start testing your app on it, and come back to set up the emulator at a later point. In that case, skip to [Step #4](#).

Otherwise, here is what you may need to do, based on the operating system on your development machine.

Windows

If your CPU met the requirements, and you successfully enabled the right things in your system's BIOS, the Android Studio installation should have installed HAXM, and you should be ready to continue with [the next step](#).

If, on the other hand, you got some error messages in the installation wizard regarding HAXM, you would need to address those first. Unfortunately, there is so much variety in PC hardware and possible problems that this book cannot help you diagnose and fix your HAXM problems.

Mac

The wizards of Cupertino set up their Mac hardware to be able to run the Android x86 emulator without additional configuration. This is really nice of them, considering that Android competes with iOS. The Android Studio installation wizard should have installed HAXM successfully, and you should be able to continue with [the next step](#).

Linux

The Android x86 emulator on Linux does not use HAXM. Instead, it uses KVM, a common Linux virtualization engine.

If, during the Android Studio installation process, the wizard showed you a page that said that you needed to configure KVM, you will need to do that before you can set up and use the x86 emulator. The details of how to set up KVM will vary by Linux distro (e.g., [Ubuntu](#)).

Step #3: Set Up the AVD

The Android emulator can emulate one or several Android devices. Each configuration you want is stored in an “Android virtual device”, or AVD. The AVD Manager is where you create these AVDs.

To open the AVD Manager in Android Studio, choose Tools > AVD Manager from the main menu.

You should be taken to “welcome”-type screen:

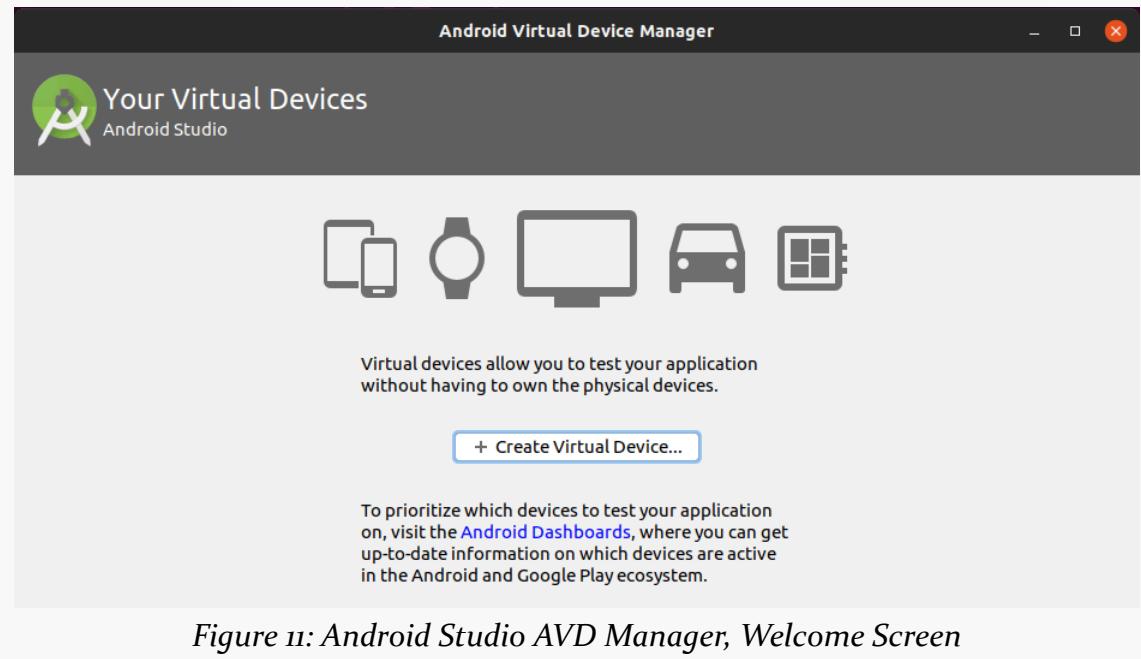


Figure 11: Android Studio AVD Manager, Welcome Screen

GETTING YOUR FIRST PROJECT

Click the “Create Virtual Device” button, which brings up a “Virtual Device Configuration” wizard:

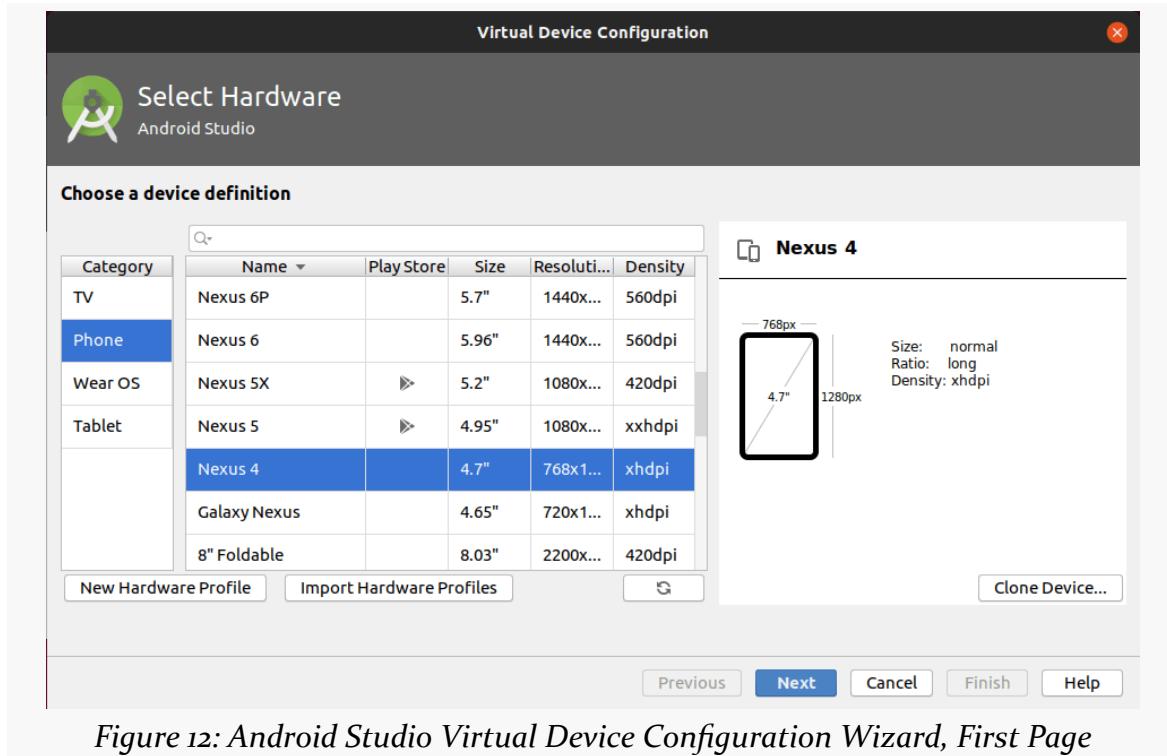


Figure 12: Android Studio Virtual Device Configuration Wizard, First Page

The first page of the wizard allows you to choose a device profile to use as a starting point for your AVD. The “New Hardware Profile” button allows you to define new profiles, if there is no existing profile that meets your needs.

Since emulator speeds are tied somewhat to the resolution of their (virtual) screens, you generally aim for a device profile that is on the low end but is not completely ridiculous. For example, a 1280x768 or 1280x720 phone would be considered by many people to be fairly low-resolution. However, there are plenty of devices out there at that resolution (or lower), and it makes for a reasonable starting emulator.

If you want to create a new device profile based on an existing one — to change a few parameters but otherwise use what the original profile had — click the “Clone Device” button once you have selected your starter profile.

However, in general, at the outset, using an existing profile is perfectly fine. The Nexus 4 image is a likely choice to start with.

GETTING YOUR FIRST PROJECT

Clicking “Next” allows you to choose an emulator image to use:

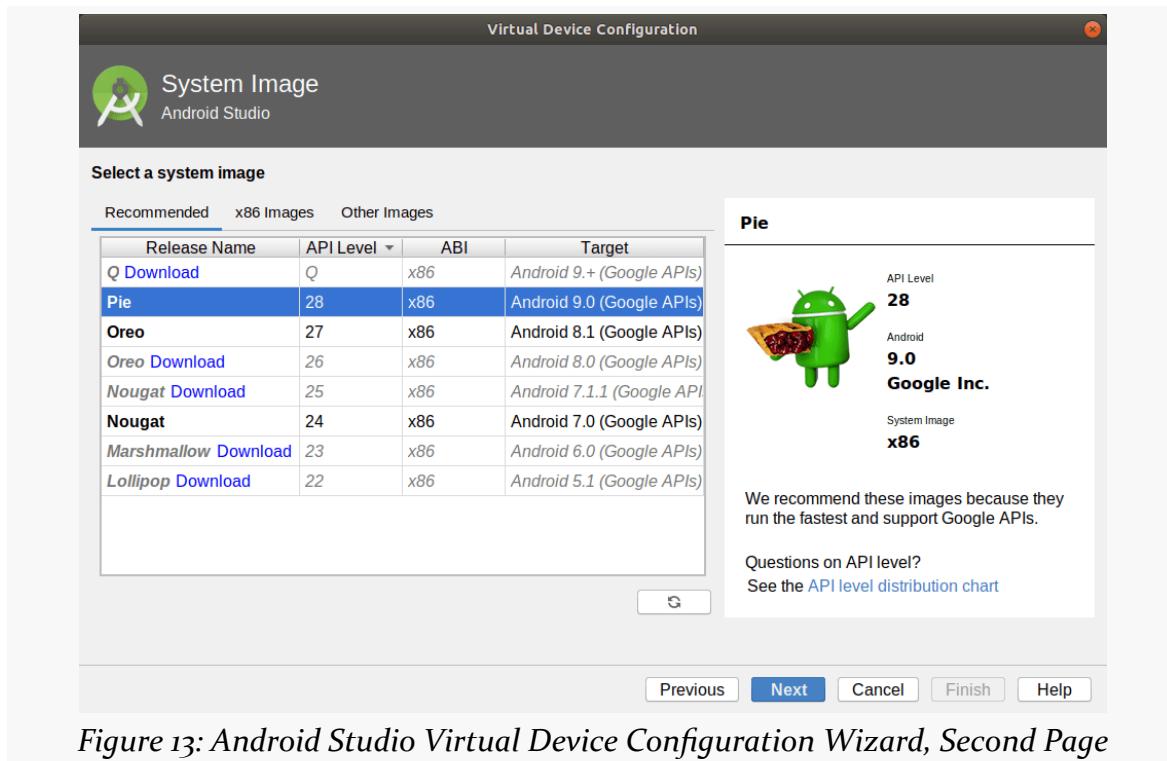


Figure 13: Android Studio Virtual Device Configuration Wizard, Second Page

The emulator images are spread across three tabs:

- “Recommended”
- “x86 Images”
- “Other Images”

Each of those tabs lists a bunch of possible emulator images, where those tables have cryptic columns like “API Level” and “Release Name”. We will get into what those are a bit later in the book. For right now, the key column is the “Target” column. This will tell you what version of Android the emulator emulates, such as “Android 8.1” or “Android 5.1”. For the time being, whether the “Target” has “(Google APIs)” or not does not matter much.

GETTING YOUR FIRST PROJECT

On some of these tabs, you should see some entries with a “Download” link, and you might see others without it. The emulator images with “Download” next to them will trigger a one-time download of the files necessary to create AVDs for that particular API level and CPU architecture combination, after another license dialog and progress dialog:

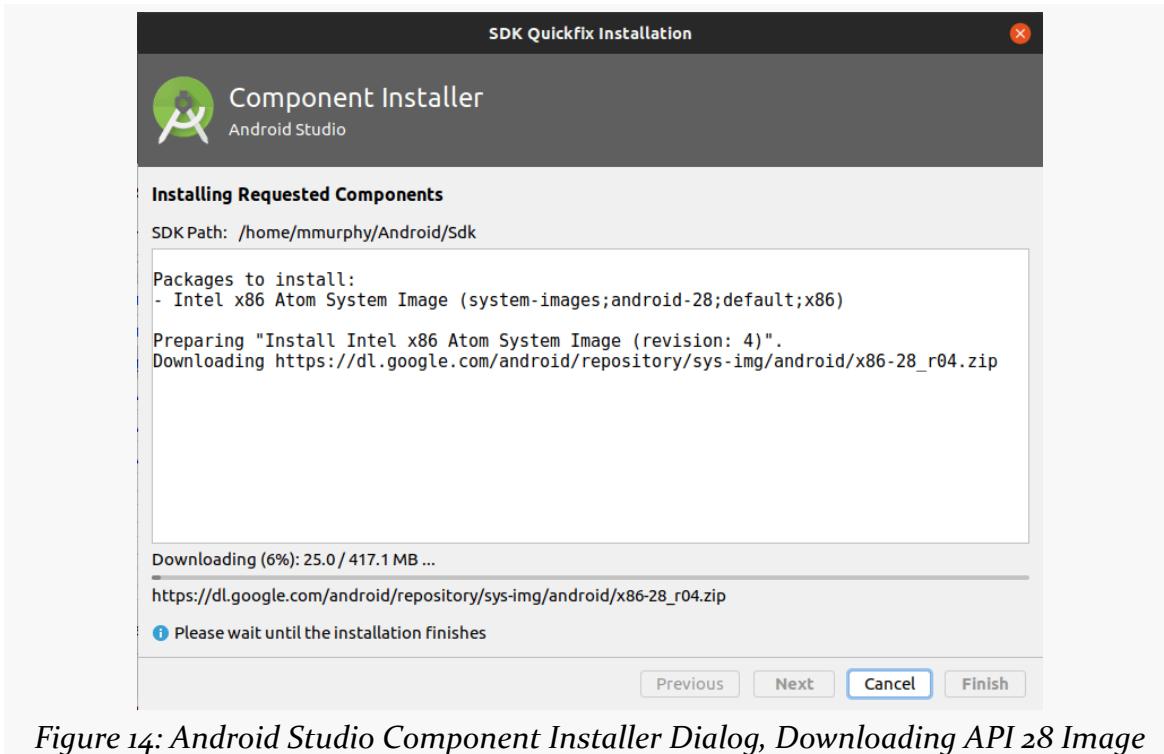


Figure 14: Android Studio Component Installer Dialog, Downloading API 28 Image

GETTING YOUR FIRST PROJECT

Once you have identified the image that you want — and have downloaded it if needed — click on one of them in the wizard. Clicking “Next” allows you to finalize the configuration of your AVD:

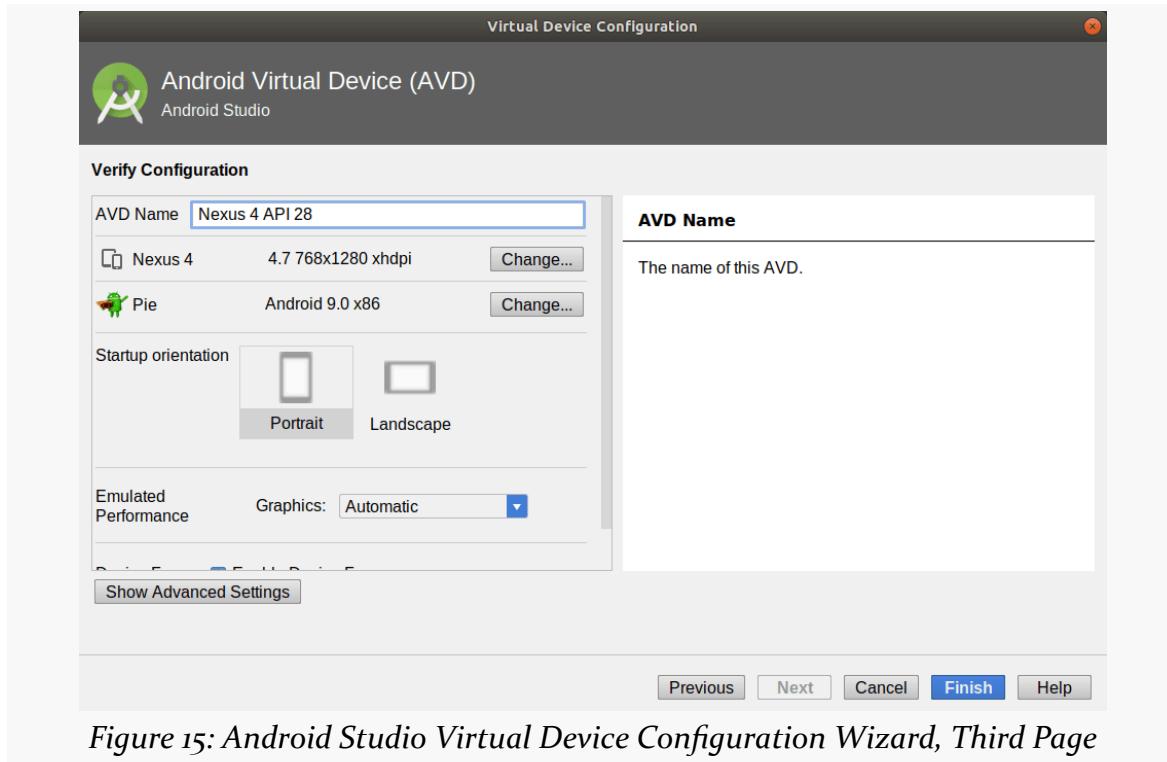


Figure 15: Android Studio Virtual Device Configuration Wizard, Third Page

A default name for the AVD is suggested, though you are welcome to replace this with your own value. However, that name must be something valid: only letters, numbers, spaces, and select punctuation (e.g., ., _, -, (,)) are supported.

The rest of the default values should be fine for now.

Clicking “Finish” will return you to the main AVD Manager, showing your new AVD. You can then close the AVD Manager window.

Step #4: Set Up the Device

You do not need an Android device to get started in Android application development. Having one is a good idea before you try to ship an application (e.g., upload it to the Play Store). And, perhaps you already have a device — maybe that is what is spurring your interest in developing for Android.

GETTING YOUR FIRST PROJECT

If you do not have an Android device that you wish to set up for development, you can skip this step and [jump ahead to Step #5](#).

First, we need to enable USB debugging on the device. To do this, go into the Settings app.

You will need to find the “Build number” entry in here. Normally, that is on an “About” screen in Settings, though some devices have it in a separate screen (e.g., “Software Info”) off of the “About” screen.

Once you find the “Build number” entry, tap it seven times.

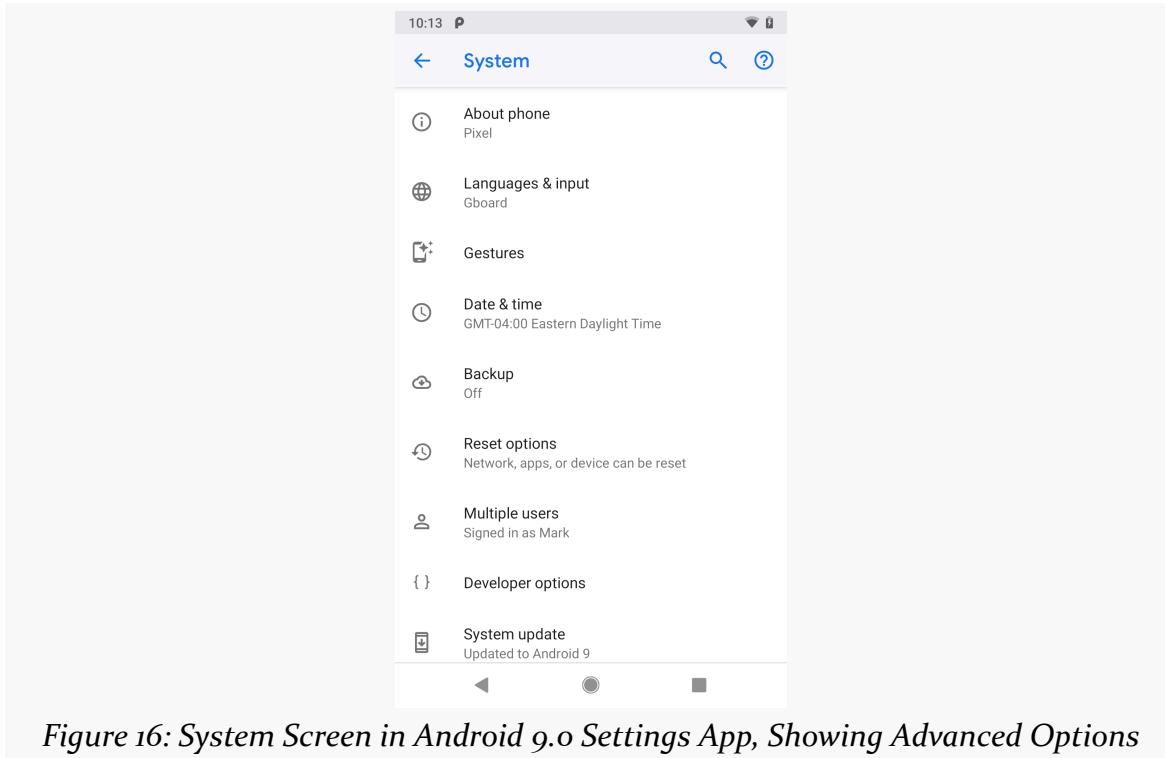
(yes, this is silly — just roll with it)

You should then see a brief popup message (a `Toast`) indicating that you are now a developer.

Then, you should have access to a “Developer options” item. Once again, the exact location of this varies by device, but usually it is either:

GETTING YOUR FIRST PROJECT

- An entry on the main Settings screen
- An entry in “System” > “Advanced options”, particularly on Android 8.0+ devices:



GETTING YOUR FIRST PROJECT

Tapping on “Developer options” will bring up the Developer Options screen:

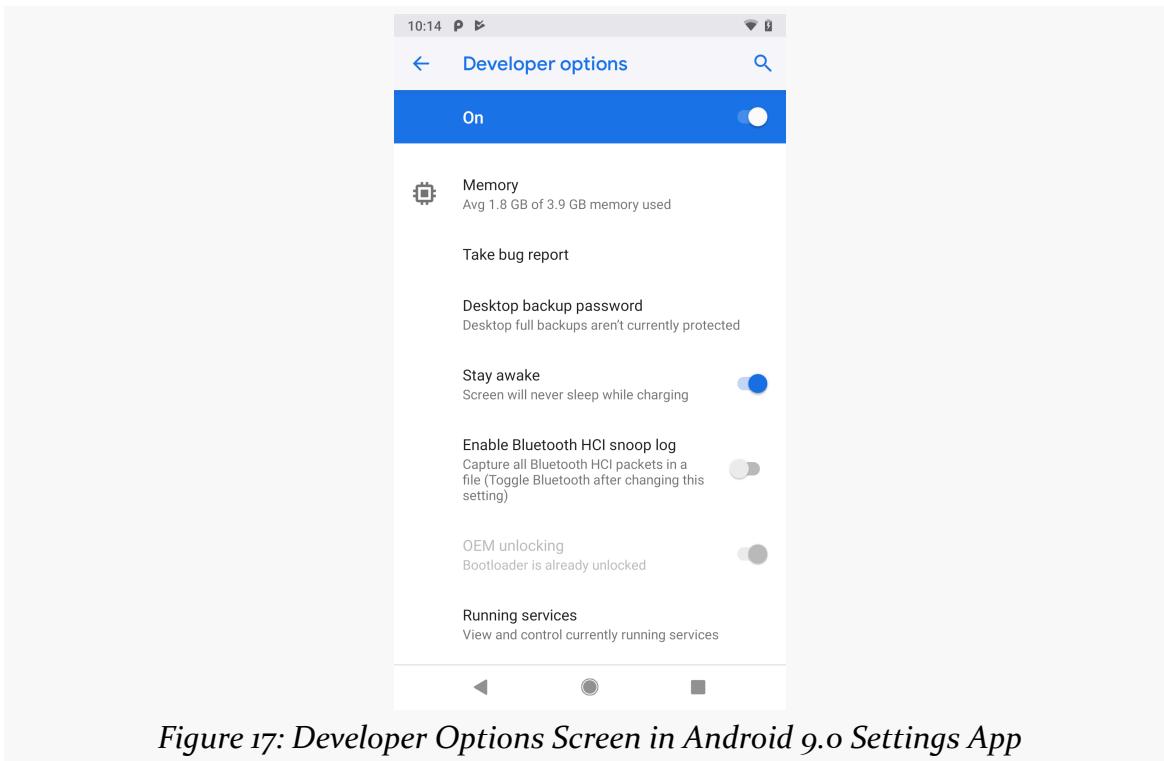


Figure 17: Developer Options Screen in Android 9.0 Settings App

You may need to slide a switch in the upper-right corner of the screen to the “ON” position to modify the values on this screen.

GETTING YOUR FIRST PROJECT

Then, scroll down and enable USB debugging, so you can use your device with the Android build tools:

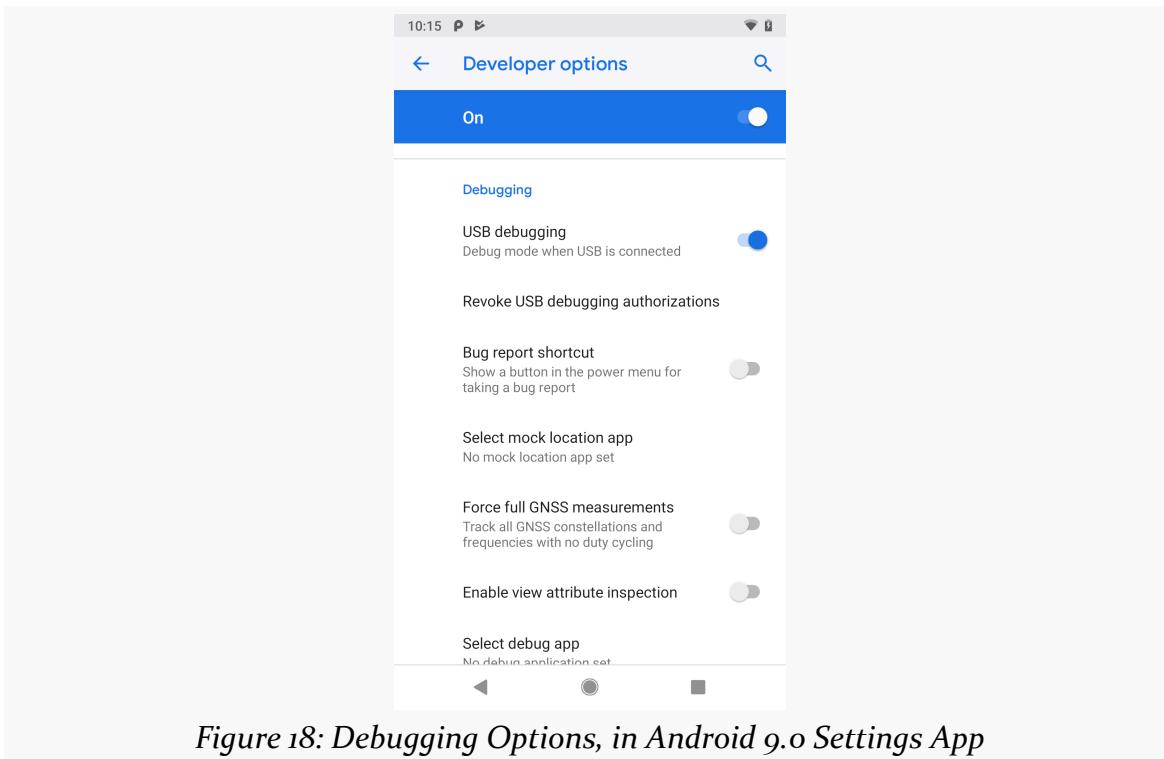


Figure 18: Debugging Options, in Android 9.0 Settings App

You can leave the other settings alone for now if you wish, though you may find the “Stay awake” option to be handy, as it saves you from having to unlock your phone all of the time while it is plugged into USB.

GETTING YOUR FIRST PROJECT

On devices running Android 4.2.2 or higher, before you can actually use the setting you just toggled, you will be prompted to allow USB debugging with your *specific* development machine via a dialog box:

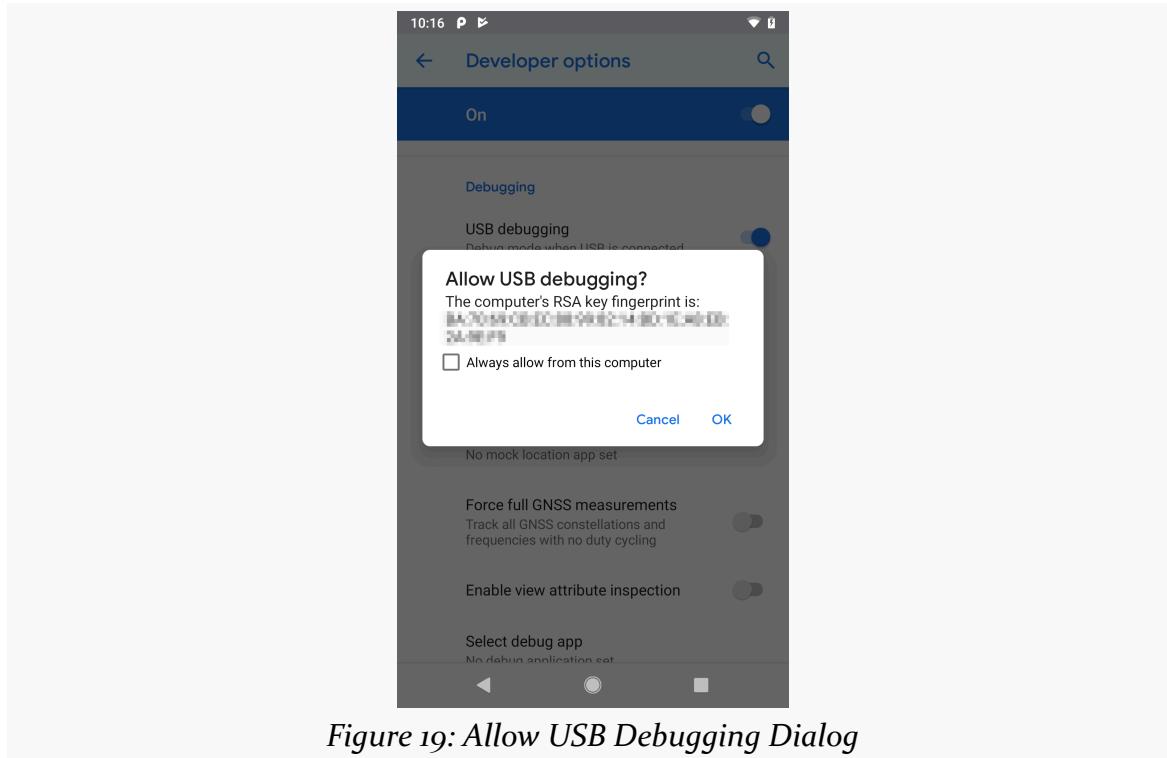


Figure 19: Allow USB Debugging Dialog

This occurs when you plug in the device via the USB cable and have the driver appropriately set up. That process varies by the operating system of your development machine, as is covered in the following sections.

Windows

When you first plug in your Android device, Windows will attempt to find a driver for it. It is possible that, by virtue of other software you have installed, that the driver is ready for use. If it finds a driver, you are probably ready to go.

If the driver is not found, you can try to get one from the manufacturer links on [the Android Developer site](#).

macOS and Linux

It is likely that simply plugging in your device will “just work”.

GETTING YOUR FIRST PROJECT

If you are running Ubuntu (or perhaps other Linux variants), and when you later try running your app it appears that Android Studio does not “see” your device, you may need to add some udev rules. [This GitHub repository](#) contains some instructions and a large file showing the rules for devices from a variety of manufacturers, and [this blog post](#) provides more details of how to work with udev rules for Android devices.

Step #5: Running the Project

Now, we can confirm that our project is set up properly by running it on a device or emulator.

Android Studio has a toolbar just below the main menu. In that toolbar, you will find two drop-down lists, followed by the Run toolbar button (usually depicted as a green rightward-pointing triangle):



The first drop-down says “this is what I want to run”. Right now, your only viable option is “app”, referring to the app that this project builds.

The second drop-down says “this is where I want to run it”. Here, you will find a list of devices and emulators that are available to you.

GETTING YOUR FIRST PROJECT

To run the app, choose your desired device or emulator in the second drop-down, then click the Run toolbar button. If you choose an emulator, and the emulator is not already running, Android Studio will start it up. Then, after a short wait, your app should appear on it:

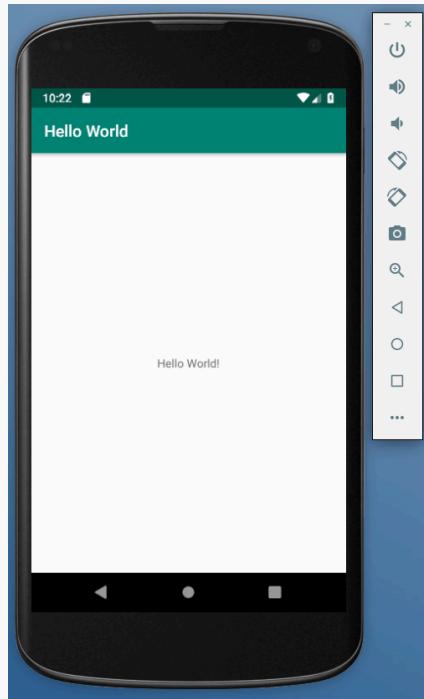


Figure 21: Android 9.0 Emulator with HelloWorld App

Note that you may have to unlock your device or emulator to actually see the app running.

The first time you launch the emulator for a particular AVD, you may see this message:

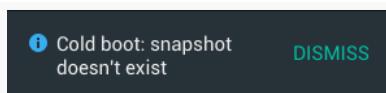


Figure 22: Android Emulator Cold-Boot Warning

The emulator behaves a bit like an Android device. Closing the emulator window is more like tapping the POWER button to turn off the screen. The next time you start that particular AVD, it will wake up to the state in which you left it, rather than booting from scratch ("cold boot"). This speeds up starting the emulator.

GETTING YOUR FIRST PROJECT

Occasionally, though, you will have the need to start the emulator as if the device were powering on. To do that, in the AVD Manager, in the drop-down menu in the Actions column, choose “Cold Boot Now”.

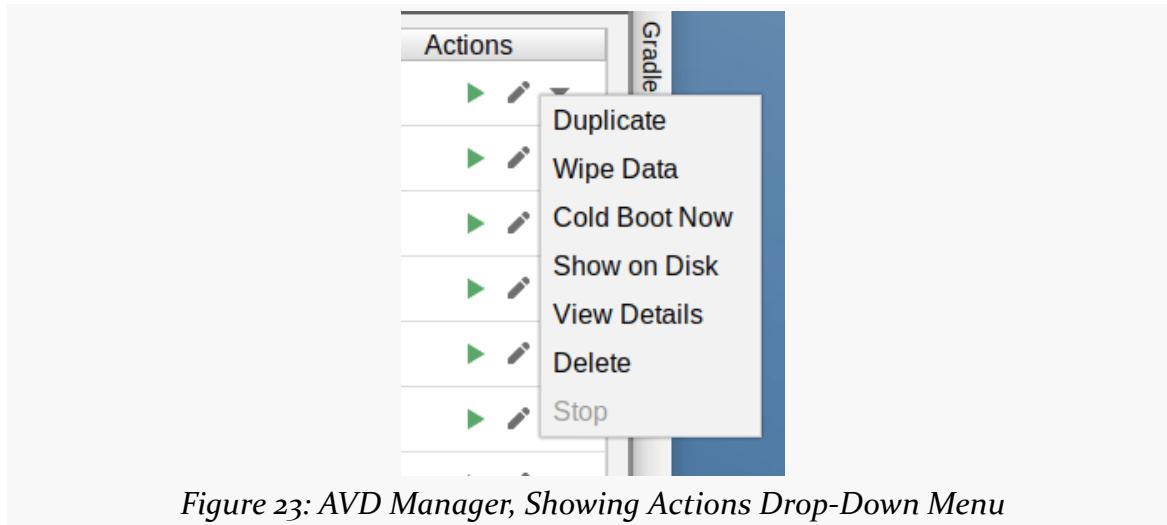


Figure 23: AVD Manager, Showing Actions Drop-Down Menu

Taking a Tour of Android Studio

At this point, you have Android Studio set up, you have imported a project, and you have run that app on a device or emulator. Congratulations!

However, it may be useful for you to understand exactly what all of this does.

So, in this chapter and the next few that follow it, we are going to walk through what you set up in the previous two chapters, to explain what the pieces are and how they work together. We will start by examining Android Studio itself and the major things that you will be using.

The Project Tree

The “Project” view — docked by default on the left side, towards the top — brings up a way for you to view what is in the project.

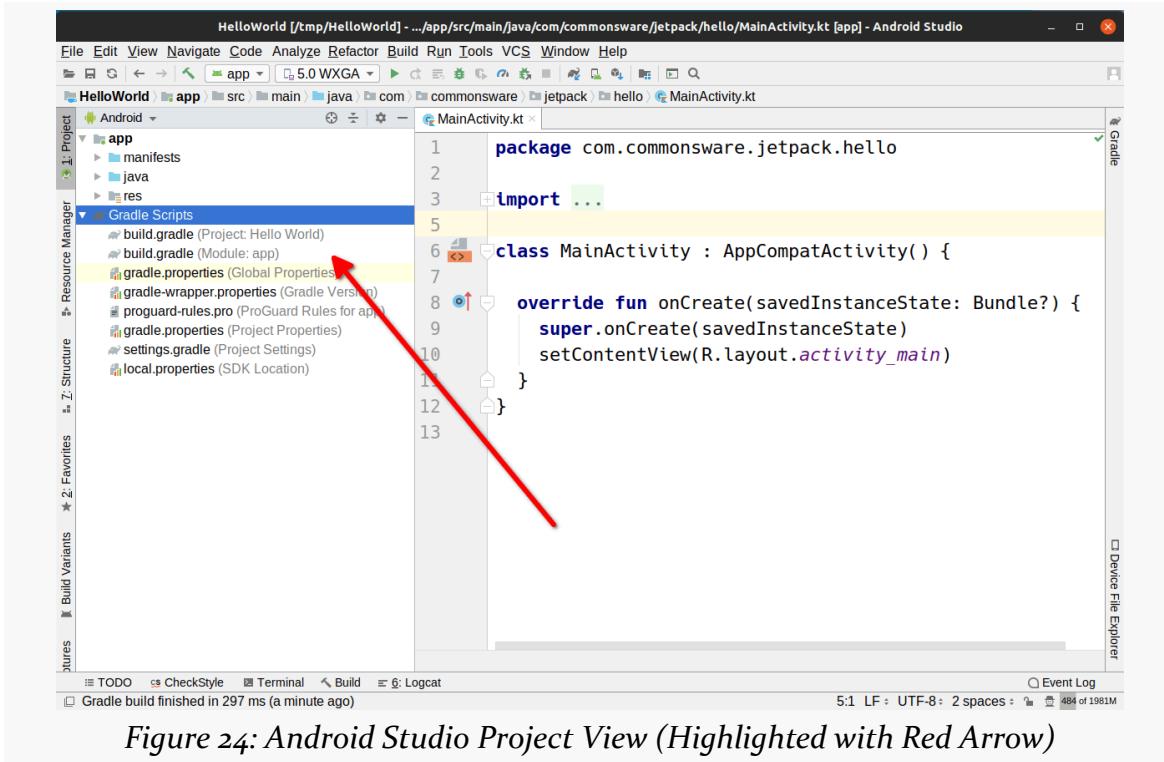


Figure 24: Android Studio Project View (Highlighted with Red Arrow)

What appears in this view is determined by the drop-down list above the tree itself:



Figure 25: Android Studio Project View Drop-Down (Highlighted with Red Arrow)

TAKING A TOUR OF ANDROID STUDIO

The default is known as the “Android view”:

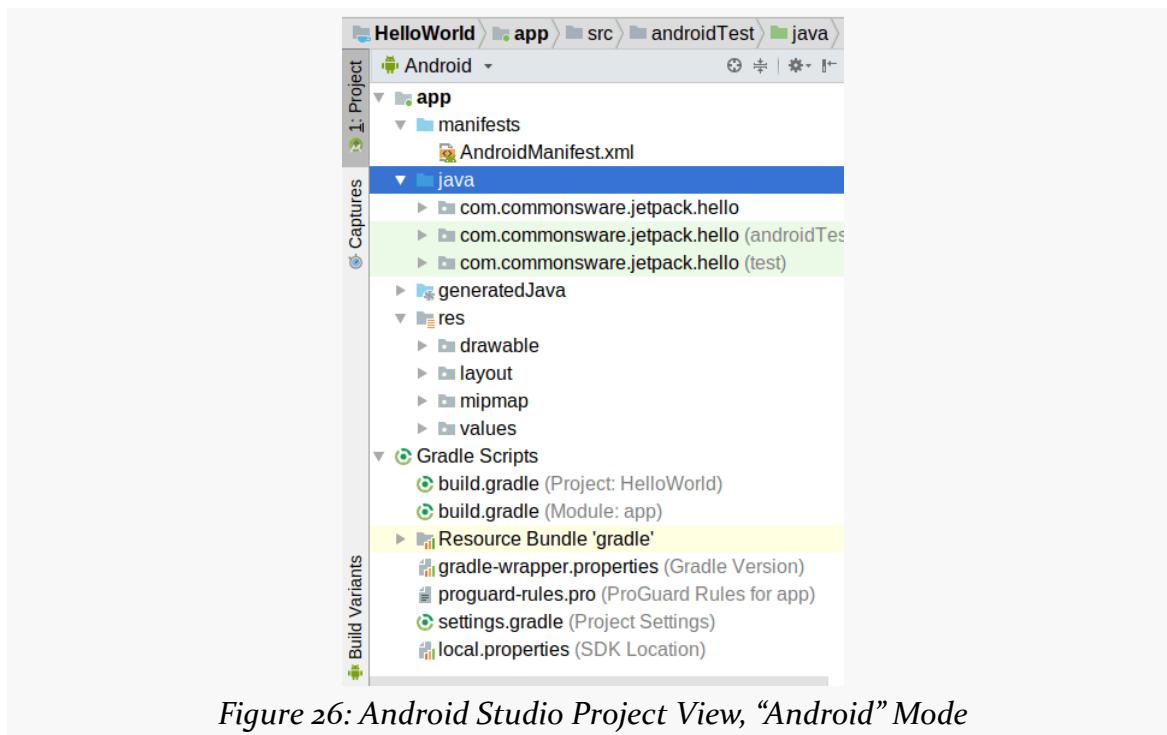


Figure 26: Android Studio Project View, “Android” Mode

TAKING A TOUR OF ANDROID STUDIO

You are welcome to use this if you wish. Many newcomers to Android are more comfortable changing that drop-down to be “Project”. This converts the tree to one showing the files that make up the project, with a typical sort of directories-and-files structure:

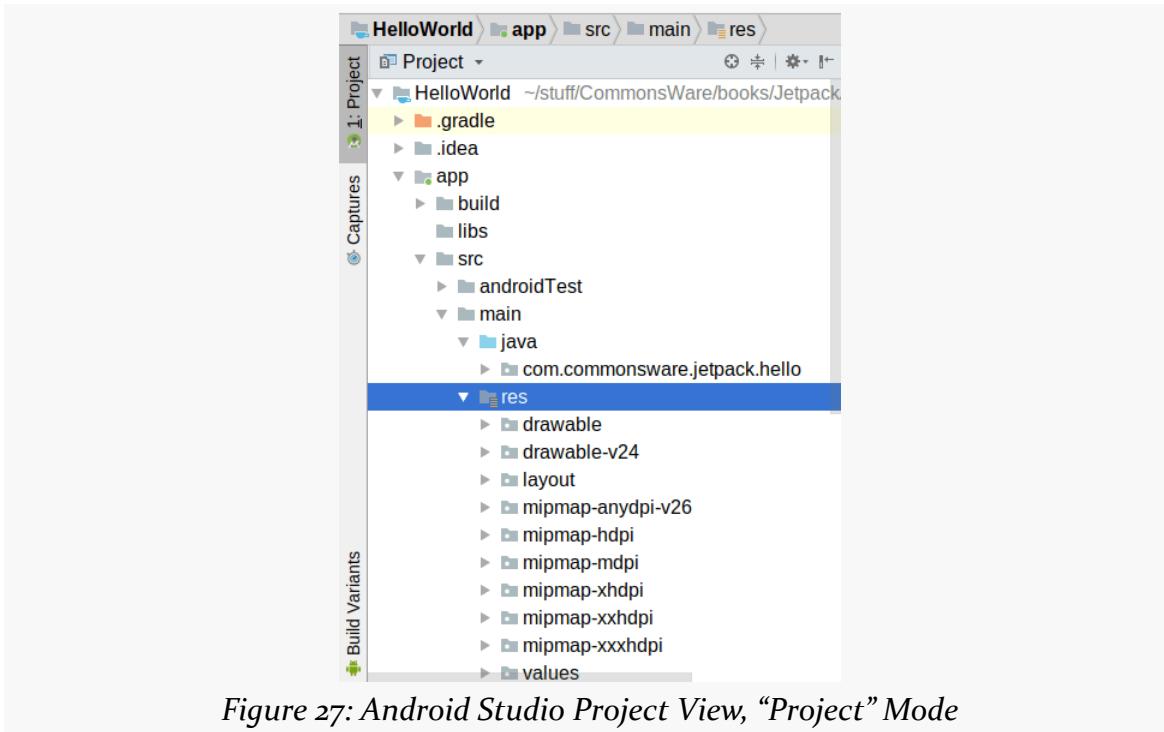


Figure 27: Android Studio Project View, “Project” Mode

This book usually will show the “Project” edition of the tree.

The Editing Pane

The biggest area of the IDE is devoted to the editing pane. Most files that you double-click on in the Project view will open in the editing pane. Each such file gets its own tab:

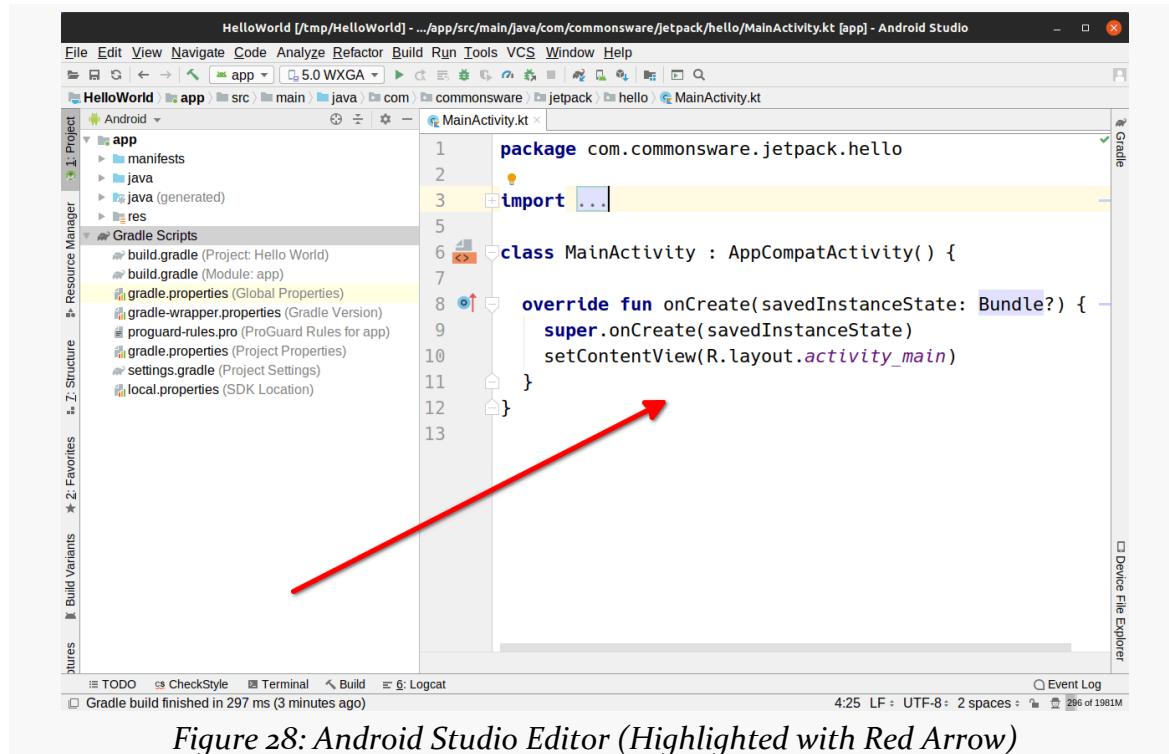


Figure 28: Android Studio Editor (Highlighted with Red Arrow)

TAKING A TOUR OF ANDROID STUDIO

In some cases, such as for Java and Kotlin code, the editing pane has one area for work (e.g., an editor for changing that Java/Kotlin code). In some cases, opening up a file will offer two or more sub-tabs, with those sub-tabs appearing at the bottom:

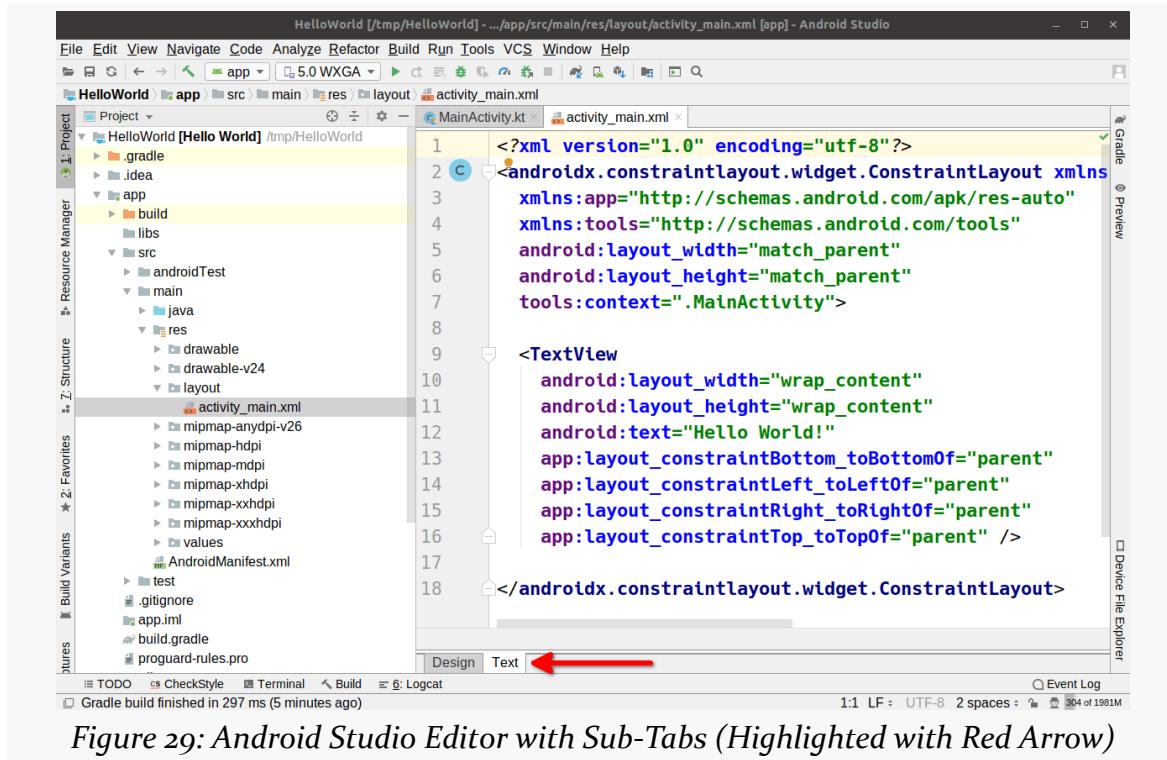


Figure 29: Android Studio Editor with Sub-Tabs (Highlighted with Red Arrow)

We will see plenty of examples of both styles of editing tab as we proceed with examining the sample app.

The Docked Views

The “Project” view is not the only such view docked along the edges of the IDE. A variety of other such views are docked there by default, and the View > Tool Windows menu will offer other such views that you can display.

TAKING A TOUR OF ANDROID STUDIO

Some of these are “general purpose”, not strictly tied to Android app development. For example, the “Terminal” tool, docked by default towards the bottom of the IDE, brings up a terminal or command prompt, so you can execute command-line programs right from within the IDE:

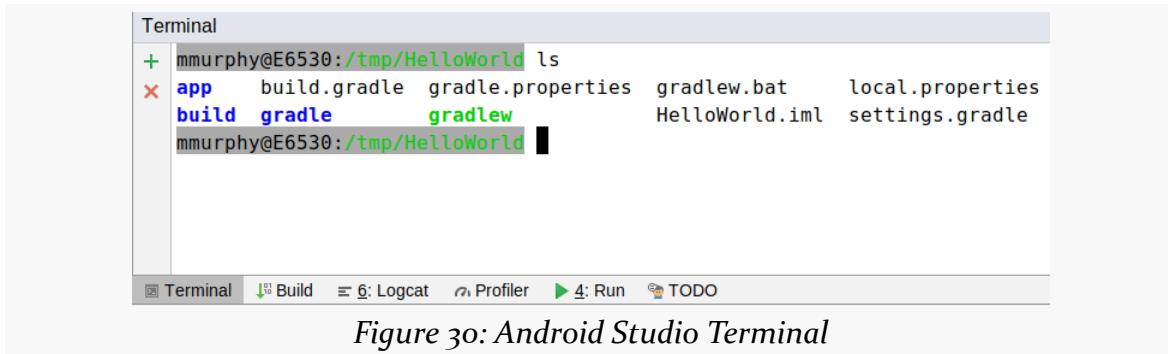


Figure 30: Android Studio Terminal

Others of these, such as “Gradle” and “LogCat”, are tied to Android app development, and we will examine those in greater detail as we explore different aspects of how to write Android apps.

Popular Menu and Toolbar Options

Across the top of the IDE are toolbars. These represent a subset of the items that are available in the IDE’s main menu. There are *lots* of toolbar options and *lots* of menu items. We will use some of these in the course of this book, but we will not be examining all of them.

By contrast, there are several menu items — many with corresponding toolbar buttons — that are fairly popular and are worth mentioning now.

We already used one of these toolbar buttons: the Run option, represented by a green triangle:



Figure 31: Android Studio Toolbar Segment

Additional Run Options

After you have run your app on a device or emulator, a few new toolbar buttons will show up, replacing the original Run button:



Figure 32: Post-Run Android Studio Toolbar Segment

The black square with a green curved arrow will re-run your app on the device or emulator. This will do the same thing as the Run button did back when it was just the simple green triangle.

The “A” with a green curved arrow, and the four lines with a tiny green curved arrow, will try to take changes that you have made in the IDE and simply “apply” them to the running copy of your app. For larger projects, this might be substantially faster than just running the app. However, it is somewhat risky — the resulting patched app might not be exactly the same as what you would get by just running it normally.

Debug

Next to those toolbar buttons is a large green bug:



Figure 33: Android Studio Debug Toolbar Button (Highlighted with Red Arrow)

This is the “Debug” button. By default, clicking this has much the same effect as clicking the Run button, other than Debug happening a bit more slowly. However, this runs your app under the control of the Android Studio debugger, where you can set breakpoints, inspect objects, and otherwise see what is going on when the app is running.

We will examine how to use the Android Studio debugger more [later in the book](#).

Open Project/Open Recent

In Android Studio, you can have several projects open at one time. Each project gets its own separate window, with the same menus, toolbar, and so on as do the other windows.

TAKING A TOUR OF ANDROID STUDIO

To open another project, you can:

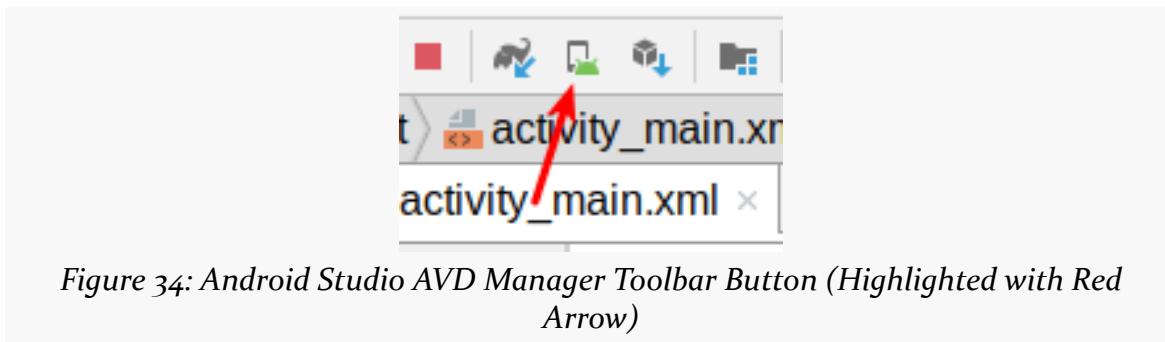
- Choose File > Open... from the main menu, to open an existing Android Studio project
- Choose File > Open Recent from the main menu, which will bring up a fly-out menu containing a list of projects that you have recently worked on, to be able to re-open those projects rapidly
- Choose File > New > Import Project from the main menu to import another project and set it up for use with Android Studio
- Choose File > New > New Project from the main menu to create a new project using a new-project wizard, which we will examine [later in the book](#)

To stop working on a project, just close its window. To stop Android Studio completely, close all of its windows. When you re-launch Android Studio, it will re-open the last project you had worked in, and you can get to other recent projects quickly via File > Open Recent.

AVD Manager

We saw how to set up an emulator with the AVD Manager in [the previous chapter](#).

To return to the AVD Manager, you can use the Tools > AVD Manager main menu option that you did before. Also, there is a toolbar button for more rapid access to the AVD Manager:



We will explore the AVD Manager, and working with the emulator, in greater detail in [an upcoming chapter](#).

SDK Manager

When you installed Android Studio and ran it for the first time, a lot of the tools,

TAKING A TOUR OF ANDROID STUDIO

libraries, and related materials that form “the Android SDK” were also downloaded and installed for you.

When updates to Android Studio or installed pieces of the Android SDK are available, you will be prompted with a dialog or other form of pop-up, where you can elect to allow Android Studio to install the update. Note that installing updates may take a few minutes, depending on your Internet connection speed. Also note that installing updates may require you to restart Android Studio afterwards to apply those updates.

On occasion, you may find instructions telling you to go to “the SDK Manager” to install something. You can get to the SDK Manager via Tools > SDK Manager in the main menu or via its corresponding toolbar button:

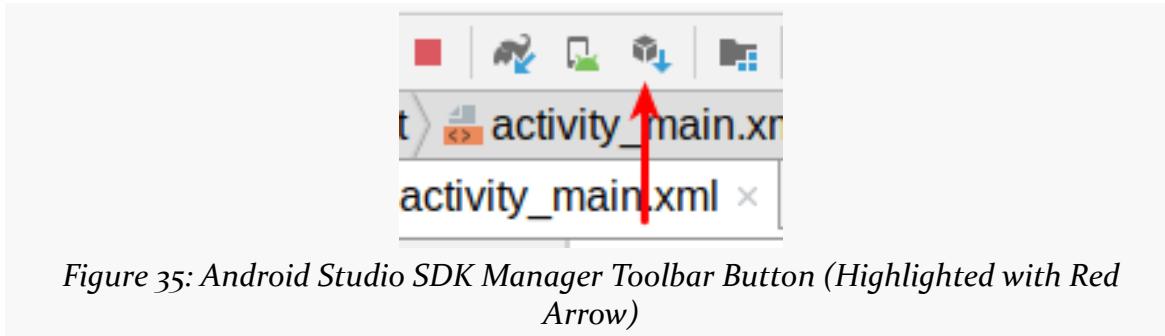


Figure 35: Android Studio SDK Manager Toolbar Button (Highlighted with Red Arrow)

We will explore the SDK Manager in greater detail [later in the book](#).

Settings

To control the overall behavior of Android Studio, there is a Settings dialog that you can display via File > Settings (on Windows and Linux; macOS has an equivalent option in the Apple menu):

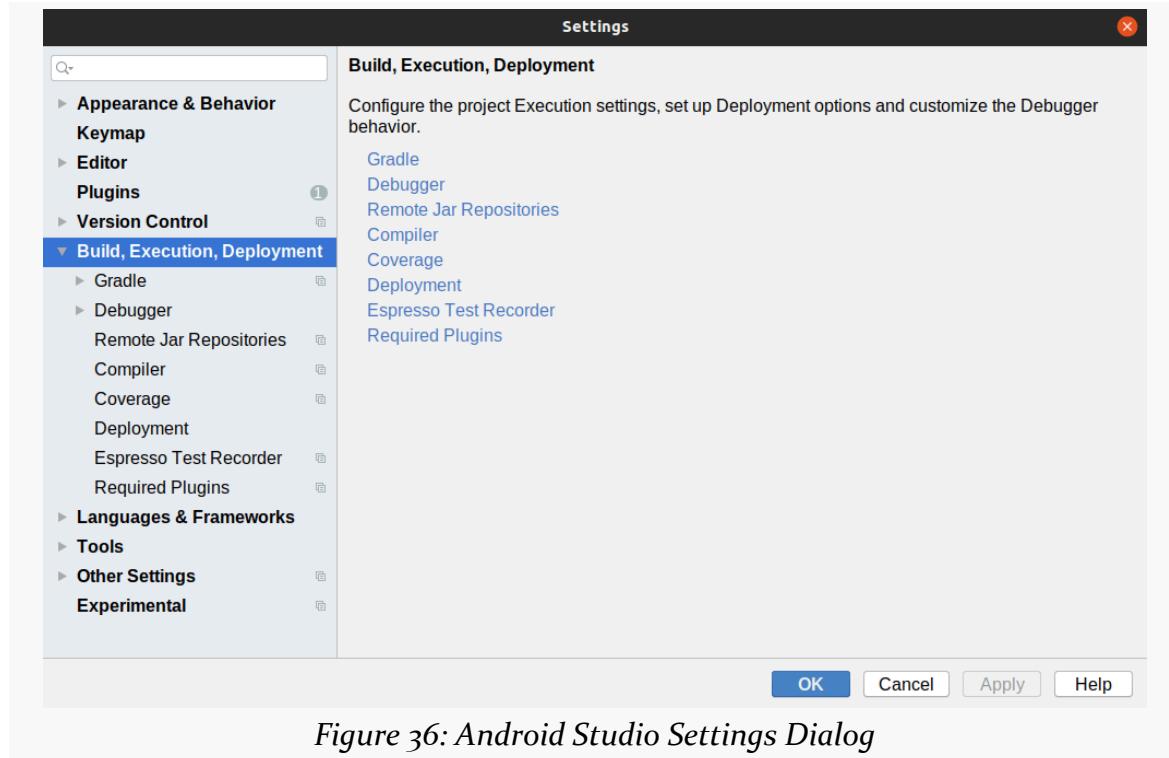


Figure 36: Android Studio Settings Dialog

There are a *lot* of settings that you can configure in the Settings dialog. You can either navigate via the tree on the left or via the search field.

Some popular things to tailor include:

TAKING A TOUR OF ANDROID STUDIO

- The color scheme to use for the IDE:

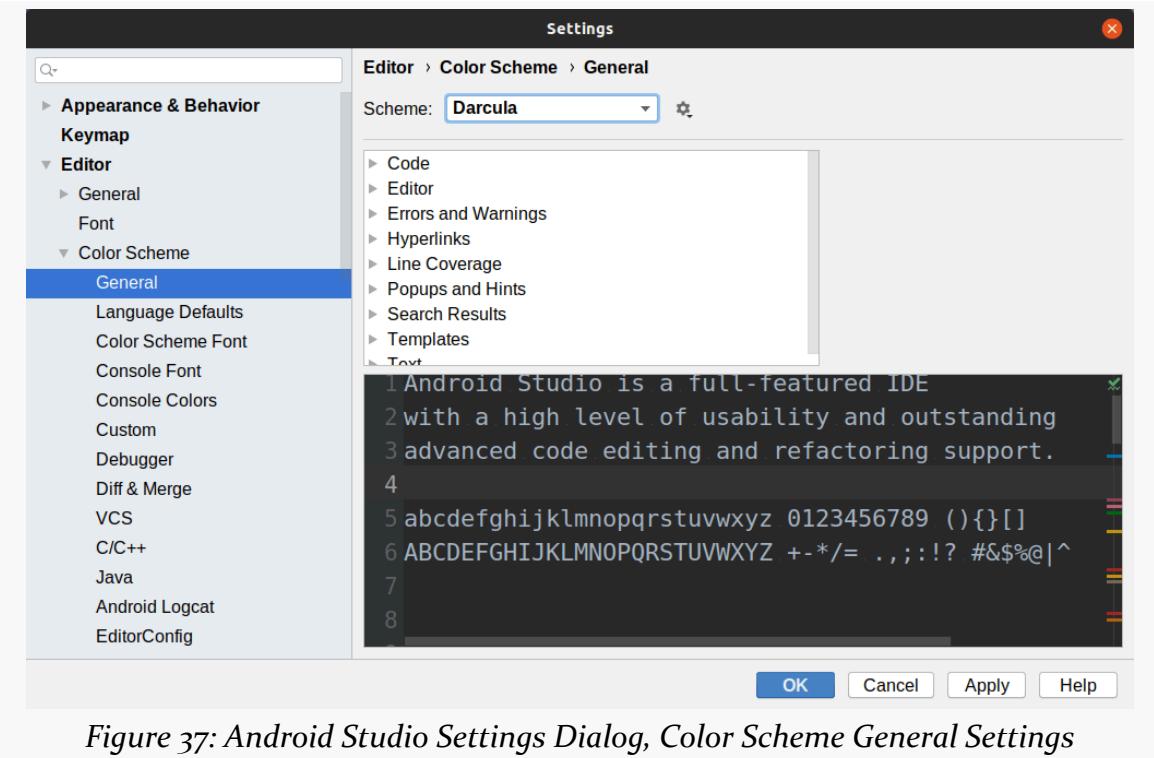


Figure 37: Android Studio Settings Dialog, Color Scheme General Settings

TAKING A TOUR OF ANDROID STUDIO

- The font and font size to use in the editing pane:

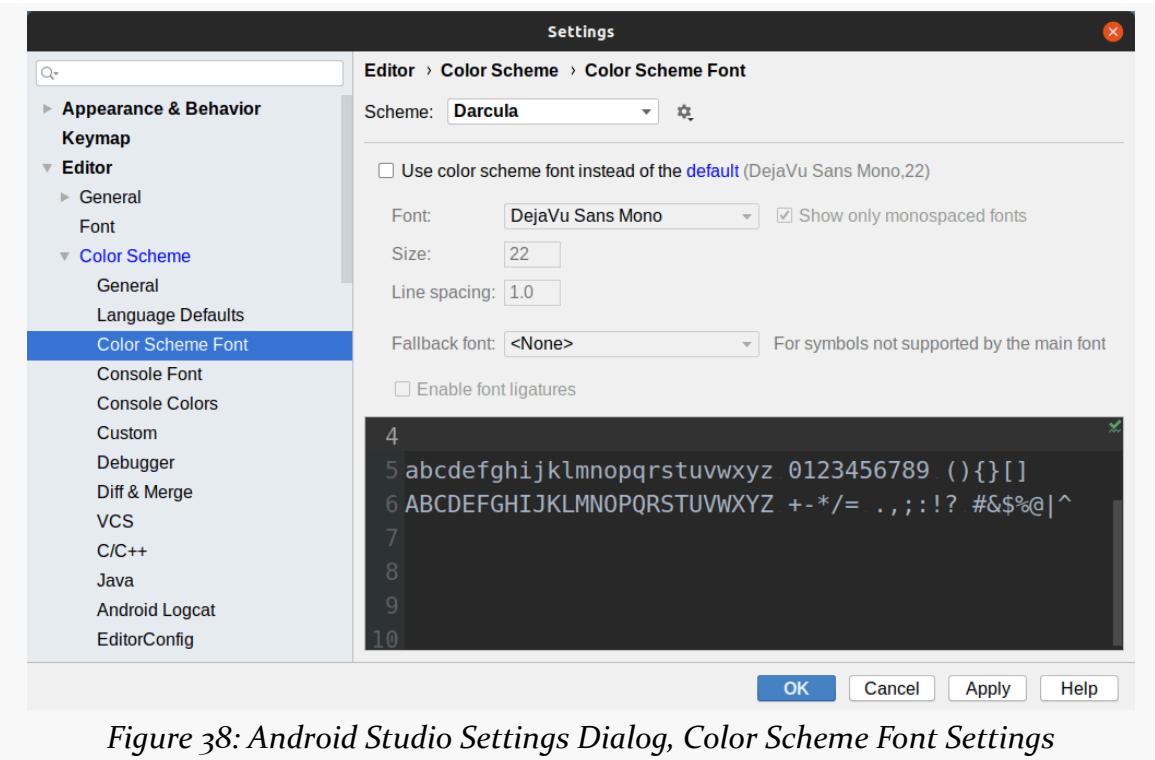


Figure 38: Android Studio Settings Dialog, Color Scheme Font Settings

TAKING A TOUR OF ANDROID STUDIO

- The coding style rules to apply to your source code, for languages like Java and Kotlin:

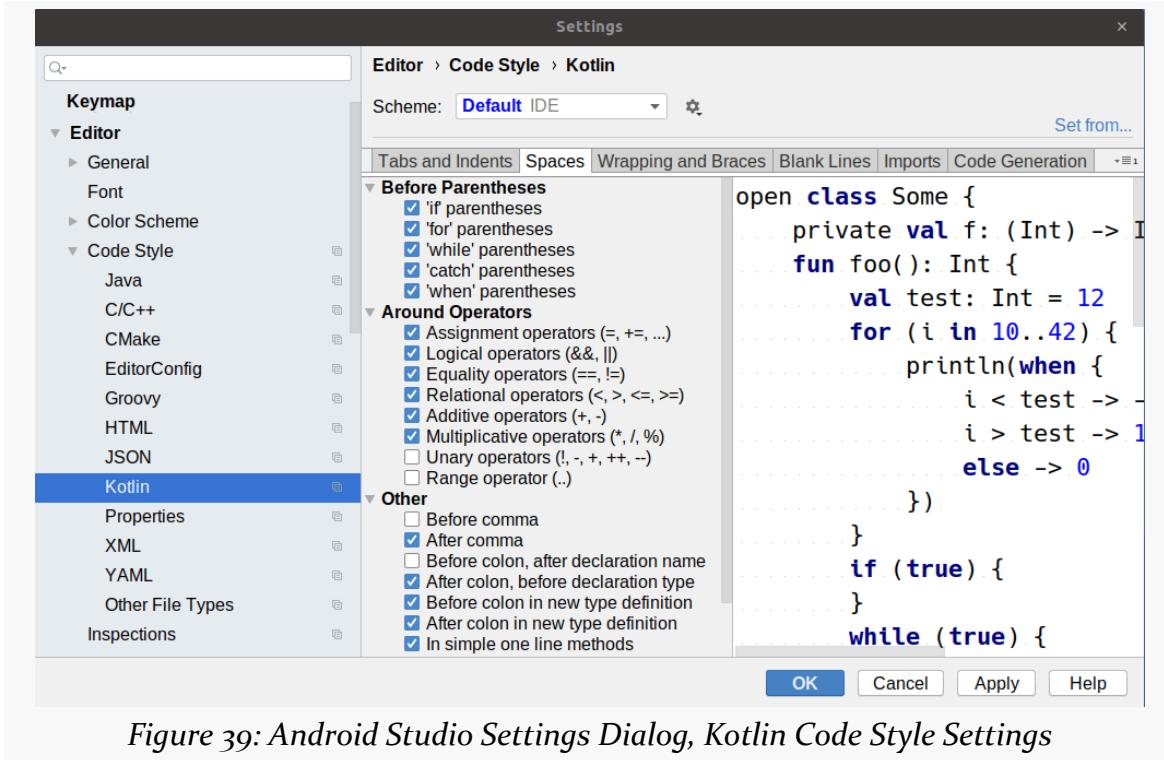


Figure 39: Android Studio Settings Dialog, Kotlin Code Style Settings

Android Studio and Release Channels

When you install Android Studio for the first time, your installation will be set up to get updates on the “stable” release channel. Here, a “release channel” is a specific set of possible upgrades. The “stable” release channel means that you are getting full production-ready updates. Android Studio will check for updates when launched, and you can manually check for updates via the main menu (e.g., Help > Check for Update... on Windows and Linux).

If an update is available, you will be presented with a dialog box showing you details of the update, allowing you to view release notes, and encouraging you to apply the update. If you choose the latter, the dialog downloads the update and restarts the IDE, applying the update along the way.

To control which channel’s worth of updates you are getting, go to the Settings dialog (from File > Settings in the main menu on Windows and Linux, with an

TAKING A TOUR OF ANDROID STUDIO

equivalent option in the Apple menu on macOS). In there, go to “Appearance & Behavior > System Settings > Updates”:

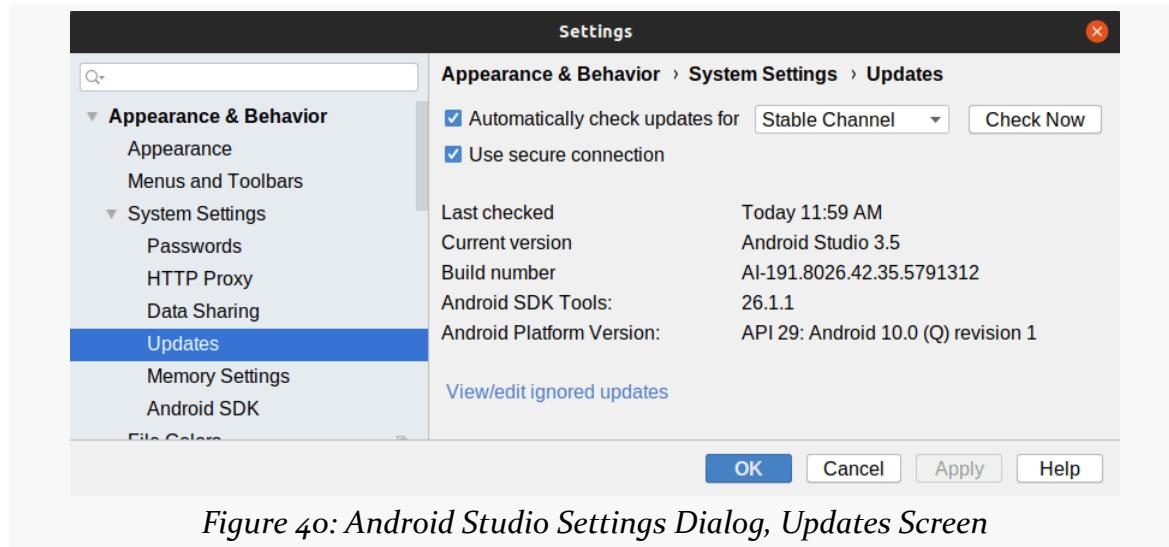


Figure 40: Android Studio Settings Dialog, Updates Screen

You have four channels to choose from:

- Stable, which is appropriate for most developers
- Beta, which will get updates that are slightly ahead of stable
- Dev, which is even more ahead than is the beta channel
- Canary, which is updated *very* early (and the name, suggestive of a “canary in a coal mine”, indicates that you are here to help debug the IDE)

For most developers, Stable is the best choice. Power users might consider one of the other channels.

TAKING A TOUR OF ANDROID STUDIO

When an update is available, Android Studio will tell you via a dialog:

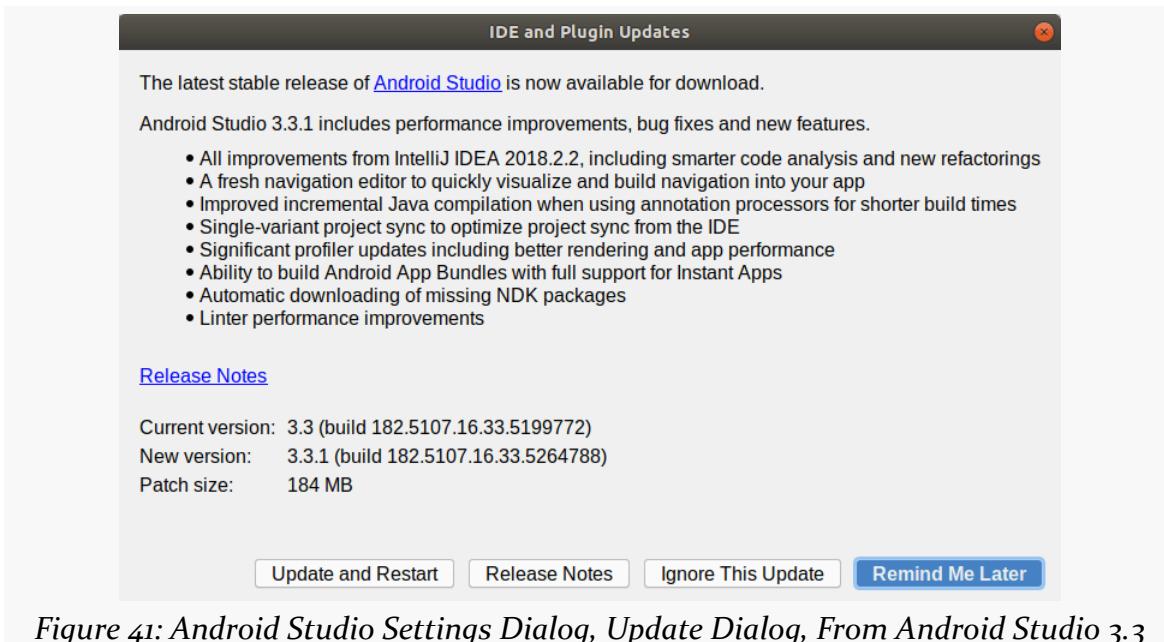


Figure 41: *Android Studio Settings Dialog, Update Dialog, From Android Studio 3.3*

“Remind Me Later” will pop up the dialog in the future, while “Update and Restart” will apply the upgrade now and restart the IDE after upgrading it. “Ignore This Update” will stop the dialog from appearing automatically, but it will not apply the update... and usually you want the updates.

Examining Your Code

When you decided to learn how to write Android apps, most likely you were thinking about traditional computer programming, using programming languages like Java and Kotlin. There is a fair amount of such programming involved in Android apps, though perhaps less than you might think.

In this chapter, we will explore what our starter project contains in terms of the code and how that code is organized.

The Top Level

Let's look at our starter project's tree, as shown in the "Project" view (in the "Project" mode in the drop-down), focusing on the top level of entries:

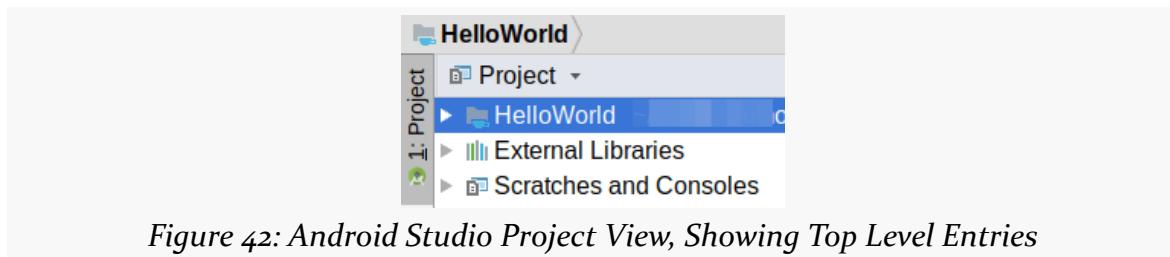


Figure 42: Android Studio Project View, Showing Top Level Entries

You will spend the *vast* majority of your time in the `HelloWorld/` portion of the project tree, which represents the files that make up your Android app. We will examine the other two items — “External Libraries” and “Scratches and Consoles” — a bit [later in this chapter](#).

The Project Contents

That `HelloWorld/` entry contains a fair number of files and subdirectories:

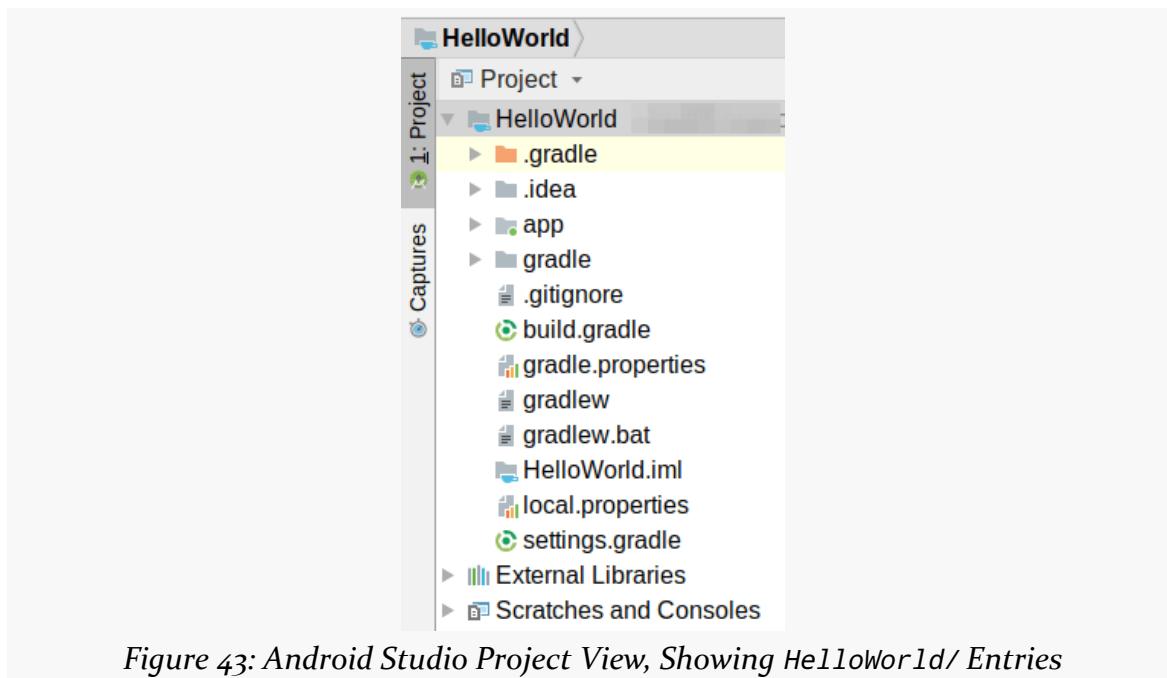


Figure 43: Android Studio Project View, Showing `HelloWorld/` Entries

Most of the time, you will be working in the `app/` directory. This is called a “module”, and it represents something that you are trying to build:

- An app for an Android device
- An app for some other specialty scenario, such as an app to be deployed to a Wear OS smartwatch
- A library to be used by multiple other modules
- And so on

Your project can have one or several modules; by default, it will just have one, named `app/`, for building your Android app. In Android Studio 3.5, a module directory is denoted by the small dot in the corner of the folder icon.

Some of the files and directories in `HelloWorld/` are tied to the Gradle build system, which we will discuss [later in this book](#):

- `.gradle/`
- `gradle/`

EXAMINING YOUR CODE

- build.gradle
- gradle.properties
- gradlew
- gradlew.bat
- local.properties
- settings.gradle

The .idea/ directory and .iml files, along with the build/ directory, are generated from the rest of the files in your project. You will not need to do anything with these manually — Android Studio will handle all of that for you.

Android Studio can work with a variety of version control systems, but it has the tightest integration with [Git](#). When you create a project in Android Studio, it will create a .gitignore file for you, set up to indicate which files do not need to go into version control. If you are using Git, this file should be a great starting point, though you can modify it as needed (e.g., to ignore other files or directories). If you are not using Git, you can ignore or delete the .gitignore file.

The App Module Contents

The app/ directory contains the files necessary to build your app:

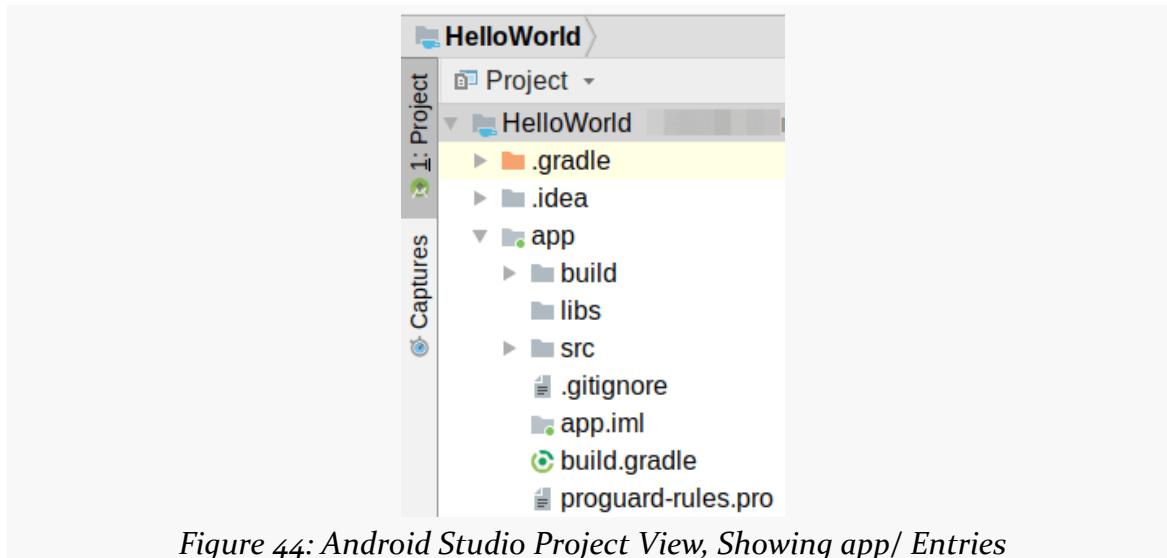


Figure 44: Android Studio Project View, Showing app/ Entries

Most of the time, you will be working in the src/ directory, which contains the source code and other files that you will be creating and editing to define your app.

EXAMINING YOUR CODE

The `build/` directory here is similar to the `build/` directory that is under the `HelloWorld/` root directory. It contains the output of building your app. In this case, we will use this `build/` directory a bit more often, as it contains the actual app itself that we can distribute through the Google Play Store or other app distribution channels. We will examine this `build/` directory a bit more [later in the book](#).

In a typical starter project, the `libs/` directory is empty. It is there to support some old ways of attaching libraries to your module, for code written by others that you wish to use. We will explore what libraries are and what ones this project uses [in an upcoming chapter](#).

The `build.gradle` file, like its counterpart in the `HelloWorld/` root directory, contains Gradle instructions for how to build your app. We will examine this file in detail [later in the book](#).

The `app.iml` file, like the `HelloWorld.iml` file in the `HelloWorld/` root directory, is generated by Android Studio from the rest of your files. These `.iml` files contain specific instructions for Android Studio, but you will not need to maintain them directly yourself.

The `.gitignore` file, like its counterpart in the `HelloWorld/` root directory, identifies files and directories that can be skipped when putting this project into a Git repository. While you may need to maintain this file if you are using Git, using Git and `.gitignore` is outside the scope of this book.

Last, the `proguard-rules.pro` file is there in support of [ProGuard](#) and related tools. These are used in Android project for two reasons:

1. They help to reduce the size of the apps, by eliminating stuff from libraries that we are not really using
2. They “obfuscate” the code in the apps, to make it slightly more difficult for people to “reverse engineer” the apps and figure out how they work

Until you are ready to think about distributing your app to other users, though, ProGuard and similar tools are not necessary. Hence, we will postpone looking at that stuff until [much later in the book](#).

The Generated Source Sets

Inside the `src/` directory are “source sets”. These identify different directories of

EXAMINING YOUR CODE

source code (and related files) that will be used in different circumstances:

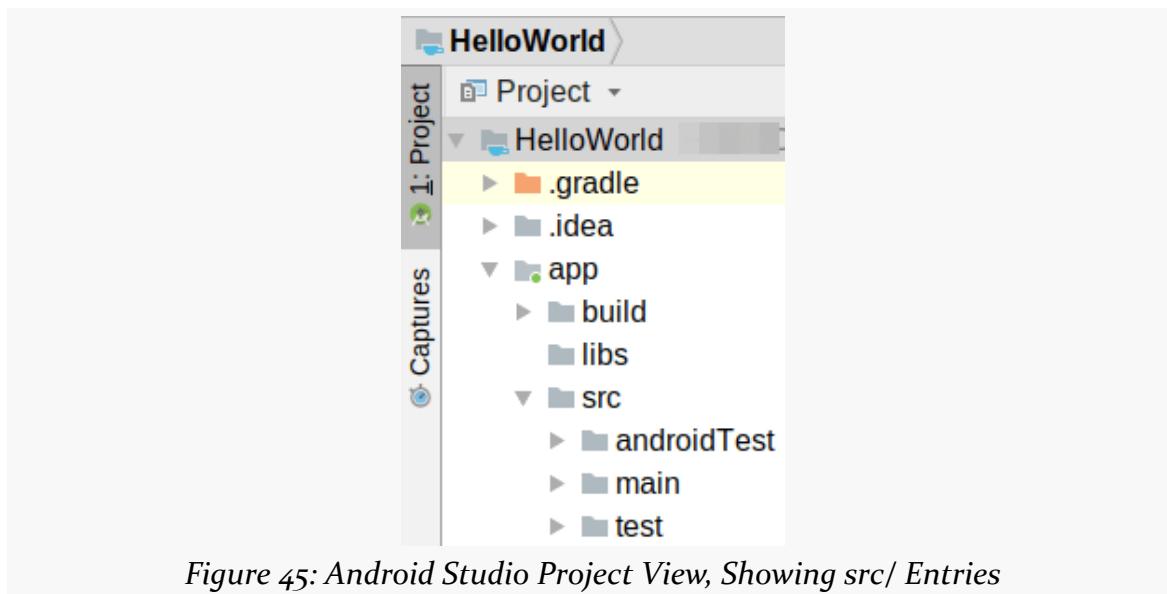


Figure 45: Android Studio Project View, Showing src/ Entries

There are three source sets that will be created by default for an Android project: `main/`, `androidTest/` and `test/`.

EXAMINING YOUR CODE

`main/`

The source set that you will spend most of your time in is `main/`. This represents the “main” source code for your app. For most apps, this source set contains all of your code (and related files) that make up the app itself:

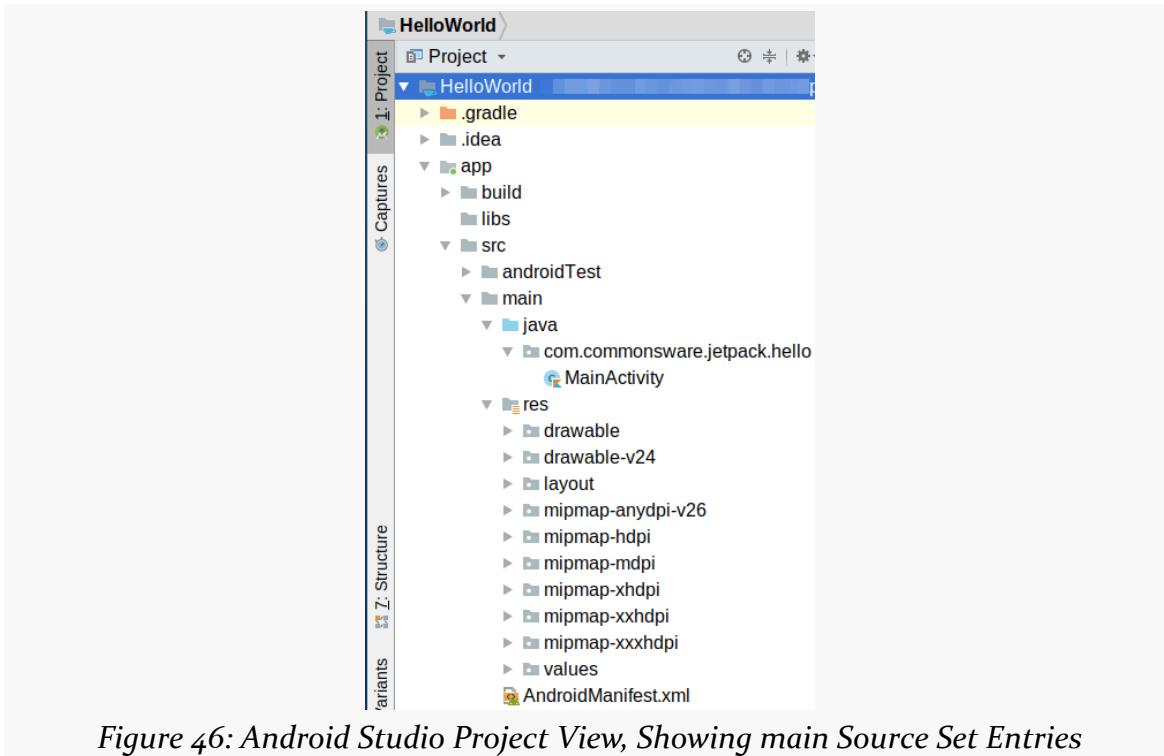


Figure 46: Android Studio Project View, Showing main Source Set Entries

Your Java and Kotlin code will go in a `java/` directory inside of the source set. Here, we see one Kotlin file, named `MainActivity.kt` (where the `.kt` part is left off in the tree).

You will also find:

- Resources, in the `res/` directory, which are files that are not source code but contribute to your app, such as your icons and other images, as we will see [in an upcoming chapter](#)
- `AndroidManifest.xml`, which is the “table of contents” of what is in your app, as we will see [in another upcoming chapter](#)
- Optionally other things (`assets/`, `jni/`, `aidl/`, etc.), though these will not be seen in every Android project

EXAMINING YOUR CODE

androidTest/

The androidTest/ source set can have its own Java/Kotlin source code, resources, and manifest file. Typically, it only has Java and Kotlin source code:

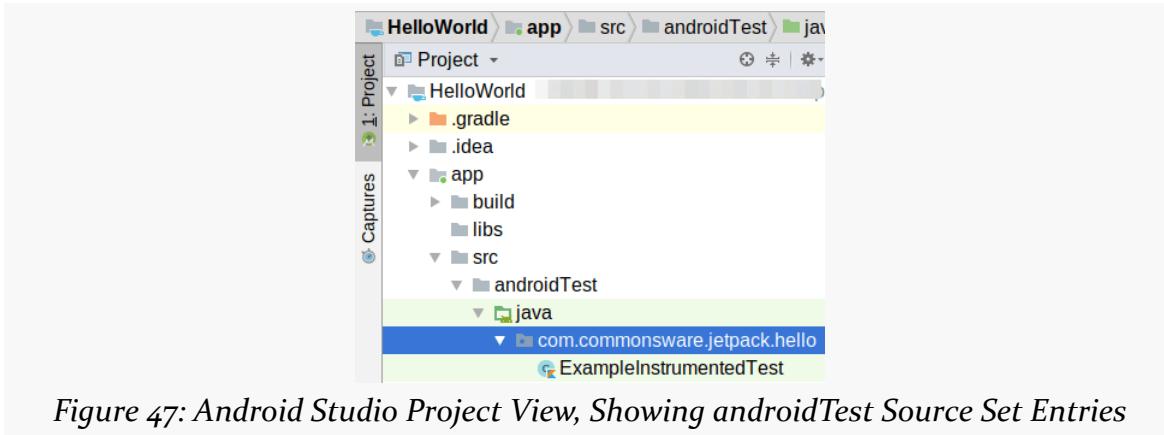


Figure 47: Android Studio Project View, Showing androidTest Source Set Entries

This source set's files will *not* go into your app. Instead, they are for testing your app, to make sure that your app does what it is supposed to do.

test/

There is a very similar source set, named test/, in a typical Android project:

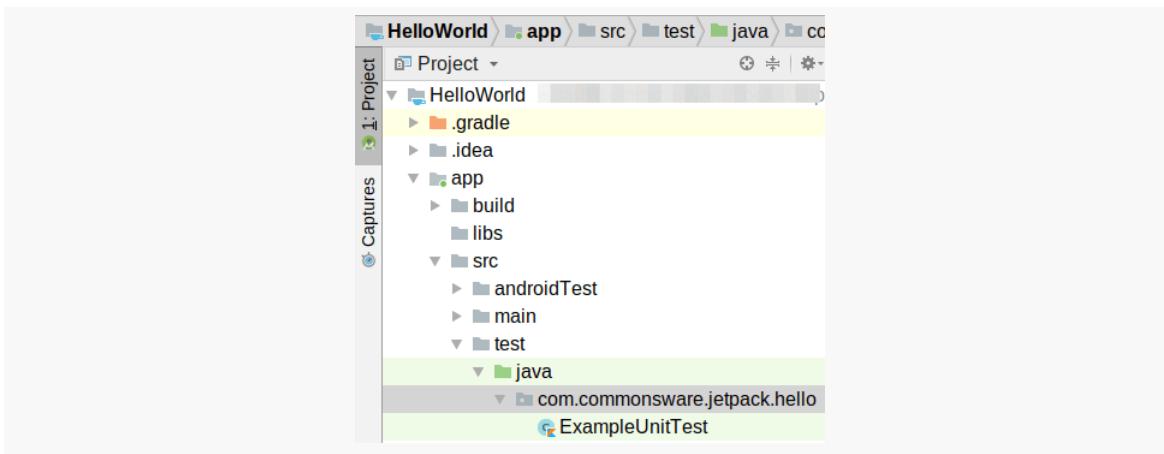


Figure 48: Android Studio Project View, Showing test Source Set Entries

As with androidTest/, test/ can contain its own Java and Kotlin code. And, as with the androidTest/ source set, the test/ code is *not* part of your app, but instead is

EXAMINING YOUR CODE

for testing your app.

The difference between `androidTest/` and `test/` is in where your tests run:

- `androidTest/` tests run inside of an Android device or emulator
- `test/` tests run directly on your development machine, in a Java virtual machine

We will explore that distinction in greater detail, along with the test code in our starter project, [later in the book](#).

Language Differences

When you import an Android project into Android Studio, you get whatever source code was in that project. This could be Java, Kotlin, or some combination.

When you create a project from scratch — as we will examine in [an upcoming chapter](#) — you will be able to tell the new-project wizard whether you want it to generate Java or Kotlin files. When you start working on adding new stuff to your project, you can add in new Java or Kotlin files.

The `HelloWorld` sample project that you imported was created using the Android Studio 3.5 new-project wizard, where the author asked for Kotlin files. Therefore, our two test source sets and the main source set contain Kotlin.

If you would prefer, you can download a Java edition of the same project from [the CommonsWare site](#). UnZIPping and importing that project gives you the same thing as the `HelloWorld` Kotlin project, except that the source code (`main/`, `androidTest/`, and `test/`) is Java, not Kotlin.

Introducing the Activity

Ignoring the test code for a while, our one-and-only source file in our project implements a `MainActivity` class, either in Java or Kotlin. This class represents an “activity”, one of the core components in an Android app.

The Role of the Activity

The building block of the user interface is the activity. You can think of an activity as being the Android analogue for the window or dialog in a desktop application, or

EXAMINING YOUR CODE

the page in a classic Web app. It represents a chunk of your user interface and, in many cases, a discrete entry point into your app (i.e., a way for other apps to link to your app).

Normally, an activity will take up most of the screen, leaving space for things like a status bar (the strip across the top with the clock, battery icon, etc.) and a navigation bar (the strip across the bottom with buttons for going back, going to the home screen, etc.)

However, bear in mind that on some devices, the user will be able to work with more than one activity at a time, such as split-screen mode on a phone or multi-window mode on a Chrome OS device. So, while it is easy to think of activities as being equivalent to the screen, just remember that this is a simplification, and that reality is more complicated (as reality often is).

In a simple app with one activity, such as this sample app, that activity will serve as the entry point for the app itself. The user's home screen will often have an "app drawer" or similar thing with a bunch of icons. While the user thinks of those as "running an app", in reality those icons pass control to an activity inside of the app, one designated as being something that should appear in a "launcher" or home screen. What makes an activity appear in this app drawer is based upon stuff found in the `AndroidManifest.xml` file, and we will see how that works [later in the book](#).

Examining the Generated Code

When you create a new project via the new-project wizard — as this sample app was — usually you will have that wizard create your first activity for you. The activity will have the same functionality regardless of whether you asked for Java or Kotlin code. Since Java and Kotlin do not have the same syntax, those files will not be identical in code, but they will be identical in functionality.

So, let's see what our sample app's activity looks like, in both languages.

Java

The `HelloWorldJava.zip` version of the sample app was created using Java as the requested programming language. So, our main source set has `MainActivity.java` in a `com.commonsware.jetpack.hello` Java package:

```
package com.commonsware.jetpack.hello;
```

EXAMINING YOUR CODE

```
import androidx.appcompat.app.AppCompatActivity;
import android.os.Bundle;

public class MainActivity extends AppCompatActivity {

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

Kotlin

The HelloWorld.zip version of the sample app was created using Kotlin as the requested programming language. So, our main source set has `MainActivity.kt` in that same Java-style package (`com.commonsware.jetpack.hello`):

```
package com.commonsware.jetpack.hello

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle

class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
    }
}
```

Code Commonalities

While Java and Kotlin differ in syntax, both activities are doing the same thing and are using the same things from the Android SDK.

AppCompatActivity

All activities in Android inherit from an `android.app.Activity` base class. In our case, `MainActivity` does not inherit directly from that class. Instead, it extends `androidx.appcompat.app.AppCompatActivity`. That, in turn, inherits from `android.app.Activity`, so `MainActivity` has `Activity` in its inheritance hierarchy.

Technically, you do not need `AppCompatActivity` — you could inherit from

EXAMINING YOUR CODE

something else, even from `Activity` itself. However, Google is making it difficult for you to extend from anything else *other* than `AppCompatActivity`. When you create a new project, it is very likely that you will be given an activity that extends from `AppCompatActivity`.

The theory is that `AppCompatActivity` makes it easier for you to develop apps that will behave consistently across many versions of Android, compared to inheriting from `Activity` or some other subclass of `Activity`.

We will see more about where `AppCompatActivity` comes from [a bit later in this chapter](#).

onCreate()

`MainActivity` has one Java method or Kotlin function: `onCreate()`. This overrides an `onCreate()` method that we inherit. Our job in `onCreate()` is to set up the basic UI that is to be shown by this activity.

In reality, `onCreate()` is just one of a series of “lifecycle methods”, methods or functions that get called as our activity is coming onto the screen, leaving the screen, and so on. We will see more about lifecycles [later in the book](#).

The very first thing that we do in `onCreate()` is chain to the inherited implementation of `onCreate()`, via a call to `super.onCreate()`. This is a very typical pattern for `onCreate()` of an activity, as the activity is not fully initialized until after `super.onCreate()` has been called. So, we try to get that out of the way early, so we are safe to do the rest of our work afterwards.

setContentView()

The other thing that we do in `onCreate()` is call a `setContentView()` method. This says “Hey, Android! The UI that we want to show starts with this!”. We supply something to serve as the foundation for our UI, which we can further tailor if needed. Calling `setContentView()` is not required, but it is a fairly typical approach.

In this case, we pass in a funny-looking value to `setContentView()`: `R.layout.activity_main`. This serves as a reference to a layout resource, named `activity_main`. We will explore resources in [the next chapter](#), including an explanation of [what this R thing is](#).

Other Things in the Project Tree

The project tree, starting at the `HelloWorld/` root node, contains all of the stuff that makes up the project. However, the project tree itself also contains two other root nodes: “External Libraries” and “Scratches and Consoles”.

External Libraries

The more important of the two, by far, is “External Libraries”, though you will only need to examine this area of the project tree on occasion.

`MainActivity` is fairly small. It can be that small because [“it stands upon the shoulders of giants”](#). In this case, those “giants” are libraries.

Our tiny app pulls in a very long list of libraries:

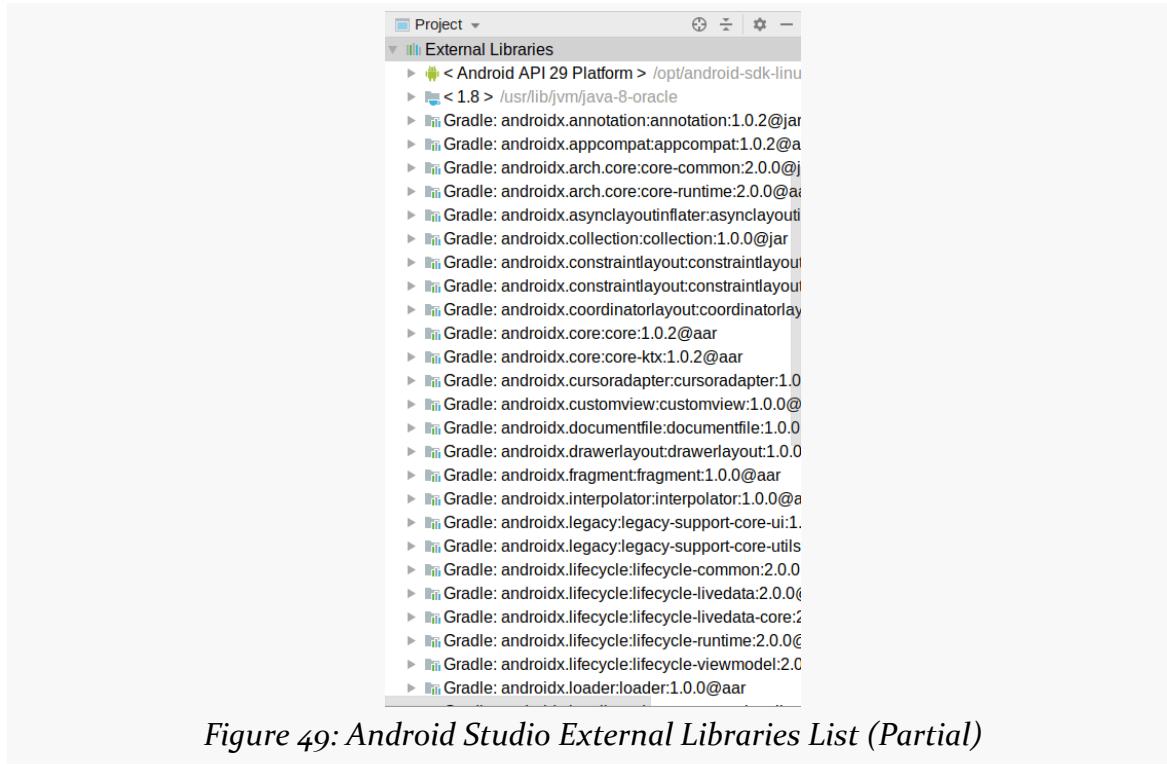


Figure 49: Android Studio External Libraries List (Partial)

This is not a complete list — the list is so long, it cannot fit in a single screenshot.

Most of these libraries come from Google and are part of the Android SDK. Some are

EXAMINING YOUR CODE

from other developers, such as libraries from JetBrains in support of Kotlin.

We will see [in an upcoming chapter](#) where these libraries come from and why they are all being used to build this little app.

Scratches and Consoles

The last item, “Scratches and Consoles”, is almost completely undocumented and seems to be infrequently used. Among other things, via the right-mouse context menu, you can create new “scratch files” here, useful for notes or testing programming language syntax outside the scope of actual project code.

Licensed solely for use by Patrocinio Rodriguez

Exploring Your Resources

Resources are static bits of information held outside the Java/Kotlin source code. As we discussed previously, resources are stored as files under the `res/` directory in your source set (e.g., `app/src/main/res/`). Here is where you will find all your icons and other images, your externalized strings for internationalization, and more.

These are separate from the Java/Kotlin source code not only because they are different in format. They are separate because you can have *multiple* definitions of a resource, to use in different circumstances. For example, with internationalization, you will have strings for different languages. Your Java/Kotlin code will be able to remain largely oblivious to this, as Android will choose the right resource to use, from all candidates, in a given circumstance (e.g., choose the Spanish string if the device's locale is set to Spanish).

In this chapter, we will examine the resources in our starter project and what their roles are in Android app development. Later chapters will cover more about these resources and describe other types of resources that your project can have.

What You See in res/

If you look at the app/src/main/res/ directory of your project, you will see a fairly long list of subdirectories:

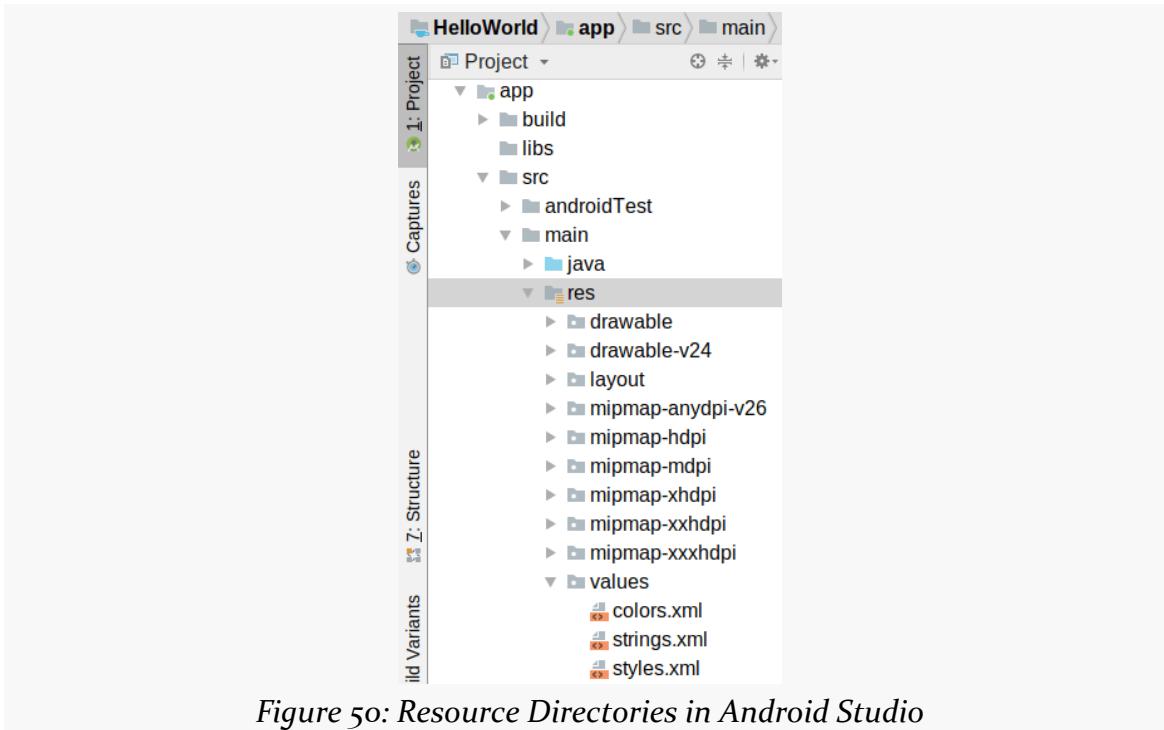


Figure 50: Resource Directories in Android Studio

Resources are placed into directory based in part on the resource type. That forms the base name of the directory, such as `drawable` and `layout`. Some directories contain a suffix after this, such as the `-v24` part of `drawable-v24`. That suffix indicates a particular resource set, which we will examine more [shortly](#).

But first, we need to talk about API levels.

OS Versions and API Levels

Android has come a long way since the early beta releases from late 2007. Each new Android OS version adds more capabilities to the platform and more things that developers can do to exploit those capabilities.

Moreover, the core Android development team tries very hard to ensure forwards

EXPLORING YOUR RESOURCES

and backwards compatibility. An app you write today should work unchanged on future versions of Android (forwards compatibility), albeit perhaps missing some features or working in some sort of “compatibility mode”. And there are techniques for creating apps that will work both on the latest and on previous versions of Android (backwards compatibility).

To help us keep track of all the different OS versions that matter to us as developers, Android has *API levels*. A new API level is defined when an Android version ships that contains changes that affect developers. When you create an emulator AVD to test your app, you will indicate what API level that emulator should emulate. When you distribute your app, you will indicate the oldest API level your app supports, so the app is not installed on older devices.

We started with API Level 1 and Android 1.0... but that was a long time ago. Nowadays, the focus tends to be on newer versions of Android and corresponding higher API levels. Here, though, Android gets a bit complicated, as there are a *lot* of different versions of Android being used today.

EXPLORING YOUR RESOURCES

The Android developer site has its [dashboards page](#). The first dashboard shows the breakdown of Android OS versions, in terms of how popular they are among devices that are active and accessing the Play Store. You will want to check this page from time to time, though the details of the sporadic dashboard updates also get reported by many Android enthusiast sites and blogs.

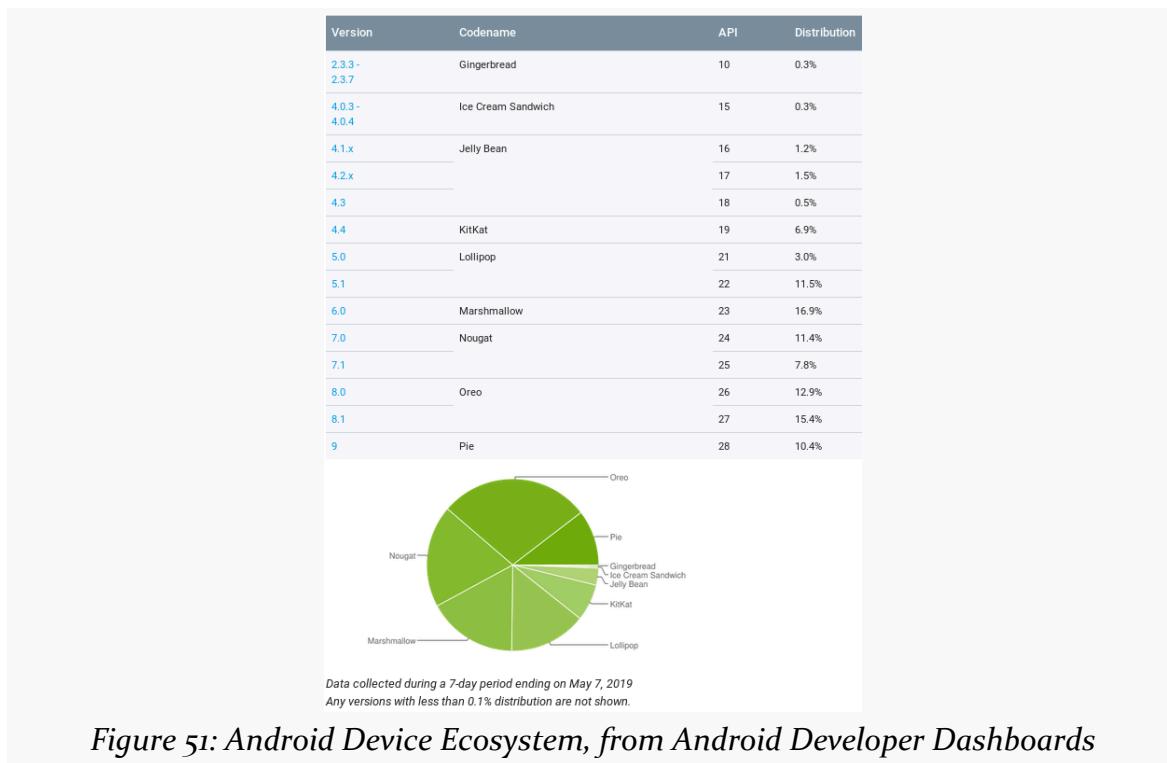


Figure 51: Android Device Ecosystem, from Android Developer Dashboards

This book focuses on Android 5.0 (API Level 21) and higher. There are ways to support older devices than that, but supporting older than Android 4.4 (API Level 19) gets complicated, so this book skips that to help keep the explanations simple.

At the time that this chapter was last updated, the latest production version of Android was 10 (API Level 29).

Beyond the latest production version, from time to time we are given “developer previews” of an upcoming version of Android. These are not good choices for new Android developers to worry about, but experienced developers may be interested in testing on pre-release Android versions and trying to use upcoming features.

Decoding Resource Directory Names

Our app has a bunch of resource directories. Some have simple names, like `drawable/` and `layout/`. Others have suffixes, like `drawable-v24/` and `mipmap-hdpi/`.

The initial segment of the directory name — or the whole name for those that lack suffixes — usually indicates the type of the resource. There are many resource types in Android, and we will explore a few of them in this chapter. The exception is the `values/` directory, which can contain a variety of smaller resource types, not just one “values” type.

The suffixes represent “resource sets”, and they indicate that this directory contains resources of a particular type that should only be used in certain scenarios. We call these scenarios “configurations”; the suffixes indicate what configurations those resources are used for.

For example:

- `drawable/` has resources that are good for any configuration, but `drawable-v24/` has resources that are only going to be used on API Level 24 and higher devices (i.e., Android 7.0 and higher)
- `mipmap-anydpi-v26/` has resources that are good for any screen density, but they will only be used on API Level 26 and higher devices
- `mipmap-mdpi/` has resources that are designed around “medium density” screens, where the density is around 160dpi (dpi = dots per inch)

We will explore these rules more [later in the book](#), as they get fairly complex fairly quickly.

Our Initial Resource Types

Our starter app contains six types of resources, though two of them (drawables and mipmaps) are pretty much the same thing.

Layouts

The resource type that will consume most of your time is the layout resource. This describes a chunk of our app’s user interface. That chunk could be:

- A screen

EXPLORING YOUR RESOURCES

- A row in a list
- A cell in a grid
- A reusable piece that you want to apply to several different screens
- And so on

Layout resources are XML files that either you create by hand or create through the use of drag-and-drop GUI builders built into Android Studio. We will be spending quite a bit of time covering layout resources throughout this book, starting with [a chapter on widgets](#), our smallest pieces of a layout resource.

The starter project has a fairly simple layout... though it could be even simpler:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

As we will see, XML elements generally will map to widgets (things that users touch) and containers (things that organize widgets and other containers). So, here, we have a ConstraintLayout container that wraps around a single TextView widget.

In [the chapter on widgets](#), we will explore this XML structure in detail, plus show you how you can set up this XML through the drag-and-drop GUI builder.

Drawables and Mipmaps

All apps have some amount of artwork, mostly in the form of icons. For example, most apps have an icon that will appear in the home screen or launcher app, that allows the user to bring up the app's UI. Apps might have other icons in that UI, to

EXPLORING YOUR RESOURCES

appear on buttons or other tappable things. Some apps may use a “splash screen” as an introductory bit of UI, and so they have some large graphic that they want to use for that screen. And there are many other uses of artwork within an Android app, of relevance to some apps but perhaps not to others.

Sometimes, these graphics are downloaded from the Internet as part of running the app. Most of the rest are packaged with the app itself: the graphic designer creates the artwork and the developer arranges to use it in the app in the appropriate place.

Most of these pre-packaged bits of artwork are in the form of `drawable` and `mipmap` resources.

Many of these are bitmap images: PNG, JPEG, etc. They can also be:

- Vector art, imported from SVG files that your graphic designer might prepare in tools like Adobe Illustrator
- Specialized XML files, usually with rules for how to combine two or more other resources together

There are really boring technical distinctions between drawables and mipmaps, and tedious historical explanations for why we have two different resource types for the same stuff. For the purposes of this book — and so you do not fall asleep while reading it — you do not need to worry about all of that. The rules for the vast majority of Android developers are fairly simple:

- Your home screen launcher icon is a `mipmap`
- Everything else is a `drawable`

We will start exploring these resources more in an upcoming chapter where [we change your launcher icon](#).

Strings

Keeping your labels and other bits of text outside the main source code of your application is generally considered to be a very good idea. In particular, it helps with internationalization (I18N) and localization (L10N). Even if you are not going to translate your strings to other languages, it is easier to make corrections if all the strings are in one spot instead of scattered throughout your source code.

Strings are one of the “values” resource types. So, in the `values/` directory, we can have one or several files that contain string resources. Typically, you have just one

EXPLORING YOUR RESOURCES

such file, named `strings.xml`.

The starter app's `strings.xml` file contains... not very much:

```
<resources>
    <string name="app_name">HelloWorld</string>
</resources>
```

All files in the `values/` directory will be XML files with a root `<resources>` element. What appears inside that root element define the actual resources contained in that file.

Inside `strings.xml`, the `<resources>` element contains just one child element: a `<string>`, defining a single string resource. Each string resource has a name, which is how we will refer to that string from elsewhere in the app. And, each string resource has a value, consisting of the text between the `<string>` and `</string>` tags. Here, we define `app_name` to be “Hello World”.

The starter app does not have translations of this resource, but it could. For example, it could contain a `res/values-es/` directory, containing strings to be used for devices whose locale is set to Spanish. In there, `app_name` might be defined as “Hola Mundo”. On the fly, Android will choose the right translation to use, based on the translations that you provide and the locale of the device.

We will be working with a bunch of string resources in this book, and we will explore the issues of translations a bit more [in a later chapter](#).

Colors

Another type of “values” resource is the color resource. As you might expect, it provides a symbolic name for colors. This allows us to give names that have semantic meaning (e.g., “the standard accent color”) and use those names in our code. It also then gives us one place to define what the actual color is for that name, so if we need to change the color, we can change it in one place.

Color resources are defined by `<color>` elements in a “values” resource file. Convention says that your colors go into a `colors.xml` resource file, and that is what the starter app has:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <color name="colorPrimary">#008577</color>
```

EXPLORING YOUR RESOURCES

```
<color name="colorPrimaryDark">#00574B</color>
<color name="colorAccent">#D81B60</color>
</resources>
```

Here, we define three colors:

- colorPrimary
- colorPrimaryDark
- colorAccent

As with our `app_name` string resource, just having these colors does not cause anything to *use* those colors. That requires additional code or additional resources, ones that happen to reference these resources.

Styles

A place where color resources are often used is in style resources. Style resources are reminiscent of CSS stylesheets in Web development. Styles allow you to give a name to a collection of UI properties, then apply those properties to various scenarios.

One such scenario is where a style is used as a “theme”. This provides the defaults for UI properties for an entire activity, or perhaps even the entire app. The sample project defines one such theme, `AppTheme`, in its `styles.xml` file:

```
<resources>

    <!-- Base application theme. -->
    <style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar">
        <!-- Customize your theme here. -->
        <item name="colorPrimary">@color/colorPrimary</item>
        <item name="colorPrimaryDark">@color/colorPrimaryDark</item>
        <item name="colorAccent">@color/colorAccent</item>
    </style>

</resources>
```

The `parent` attribute on the `<style>` indicates that we are inheriting existing UI property definitions from something called `Theme.AppCompat.Light.DarkActionBar`. That name has lots of pieces:

- Theme indicates that we are starting from the base system theme
- AppCompat indicates that this theme comes from the same AppCompat library that [gives us AppCompatActivity](#)

EXPLORING YOUR RESOURCES

- Light indicates that the general look is dark text on a light background
- DarkActionBar says that the “action bar” — the toolbar structure towards the top of the screen of most Android activities — should have white icons on a dark background

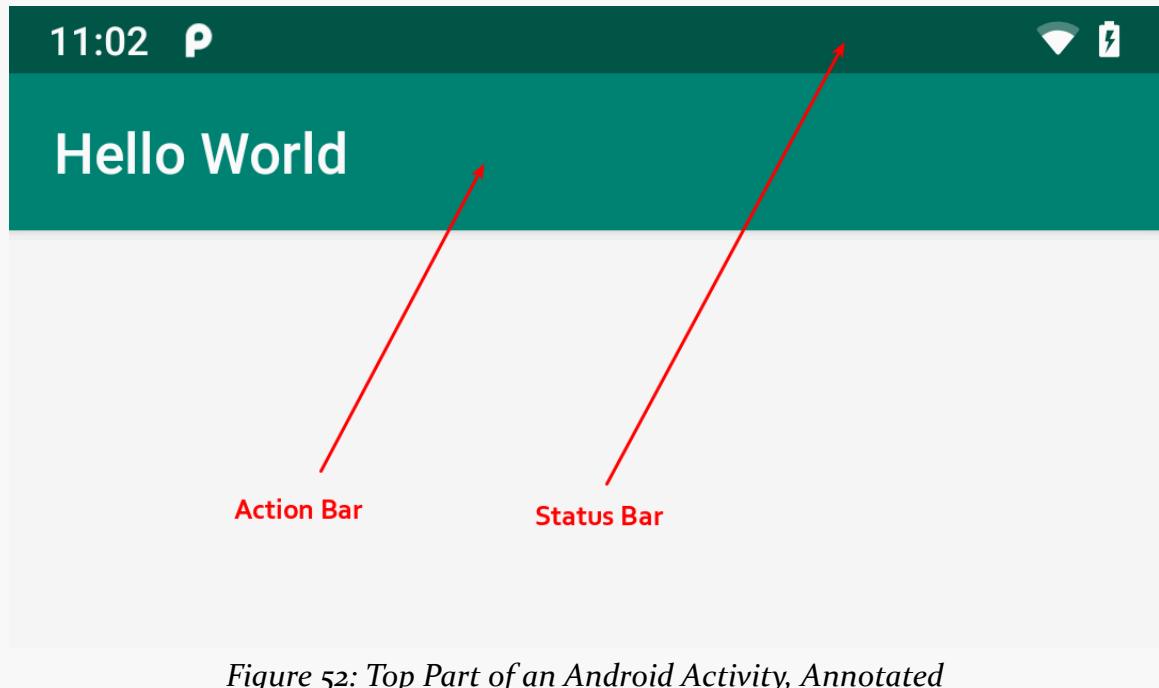


Figure 52: Top Part of an Android Activity, Annotated

AppTheme inherits from Theme.AppCompat.Light.DarkActionBar, so we get lots of stuff “for free” as a result. We then override additional UI properties as we see fit. Here, we override three of them:

- colorPrimary, which AppCompat will use for the action bar background, the foreground text in the main UI area, and a few other roles
- colorPrimaryDark, which AppCompat will use for the status bar background
- colorAccent, which AppCompat will use as a contrasting color in a variety of places, such as the text cursor for fields

The AppTheme style refers to the color resources that were defined in the `colors.xml` file. In resources, when you need to refer to another resource, you do so using the syntax `@type/name`, where type is the type of the resource (color, string, drawable, mipmap, etc.), and name is the name of the resource. For “values” resources, like our colors, the name comes from the name attribute of the element that defines the

EXPLORING YOUR RESOURCES

resource. For all other types of resources, the name comes from the filename of the resource file, without the file extension. So, here, @color/colorPrimary refers to the colorPrimary color resource.

Note that the resource references do *not* include any of those suffixes on the directory names that we use for resource sets. If you look in the various directories for mipmap resources, you will see that we have *six* different variations of an ic_launcher mipmap:

- mipmap-anydpi-v26/ic_launcher.xml
- mipmap-hdpi/ic_launcher.png
- mipmap-mdpi/ic_launcher.png
- mipmap-xhdpi/ic_launcher.png
- mipmap-xxhdpi/ic_launcher.png
- mipmap-xxxhdpi/ic_launcher.png

However, when things like [our manifest](#) refer to these, it is always as @mipmap/ic_launcher. Android will decide, on the fly, which of these six definitions to use, based on the rules encoded in those directory names and the configuration of the device at the time we are trying to use the resource. We will get much more into all of that complexity [later in the book](#).

About That R Thingy

When we were looking at the source code to `MainActivity`, we saw this line:

```
setContentView(R.layout.activity_main)
```

`setContentView()` tells the activity “this is the UI to display”. The `R.layout.activity_main` value is a reference to our `activity_main.xml` layout resource.

Just as we refer to our app’s resources from other resources using `@type/name` syntax, we refer to our app’s resources from Java and Kotlin using `R.type.name` syntax. The same rules apply:

- The type is the type of the resource (e.g., layout), not counting any suffixes that might be on the directory name
- The name is the name attribute of a “values” resource or the filename of other types of resources, excluding the file extension

EXPLORING YOUR RESOURCES

Occasionally, you will try to refer to an R value and the IDE will say that it cannot find that value. We will explore this problem more in [a bit later in the book](#).

The Resource Manager

Android Studio also has a “Resource Manager” tool. This is designed to help you navigate some key types of resources more easily:

- Drawables (though not mipmaps for some reason)
- Colors
- Layouts

This tool is accessible via a “Resource Manager” button, docked by default on the left edge of the Android Studio window.

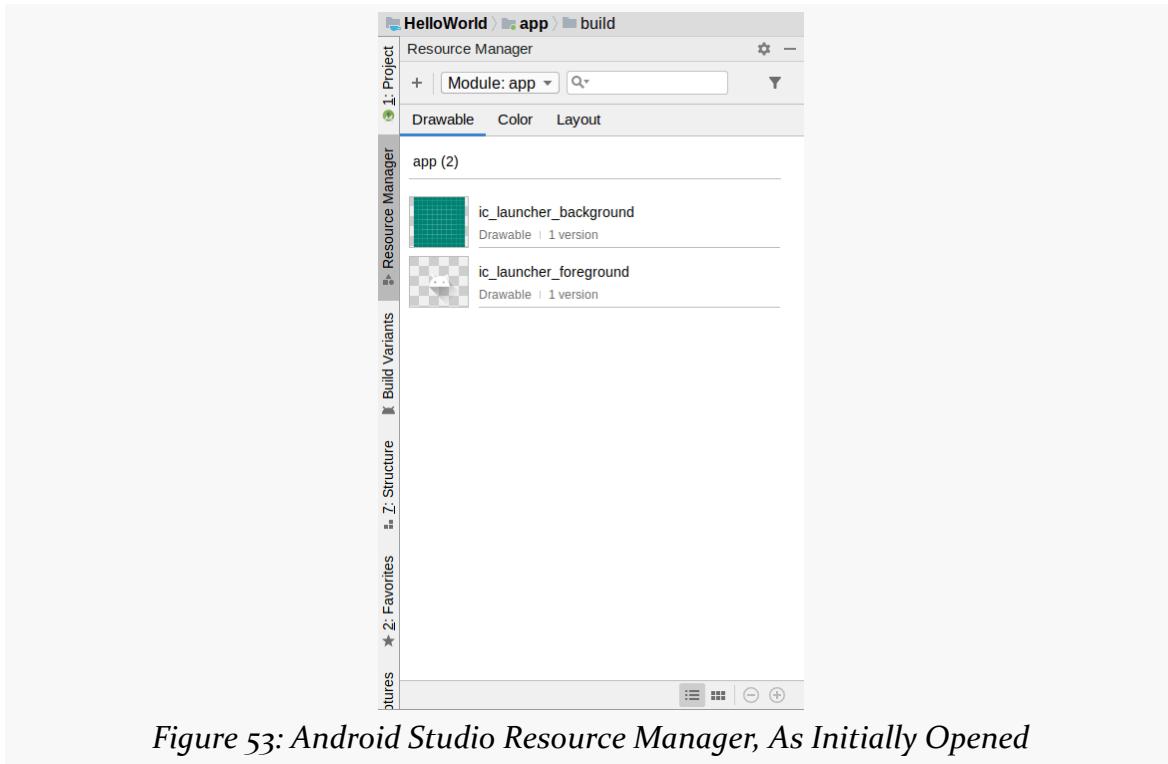


Figure 53: Android Studio Resource Manager, As Initially Opened

EXPLORING YOUR RESOURCES

There are a set of tabs towards the top that allow you to toggle between the three sets of resources that this tool supports. By default, each tab's contents is a list, but there are buttons towards the bottom of the tool to toggle between list and grid modes:

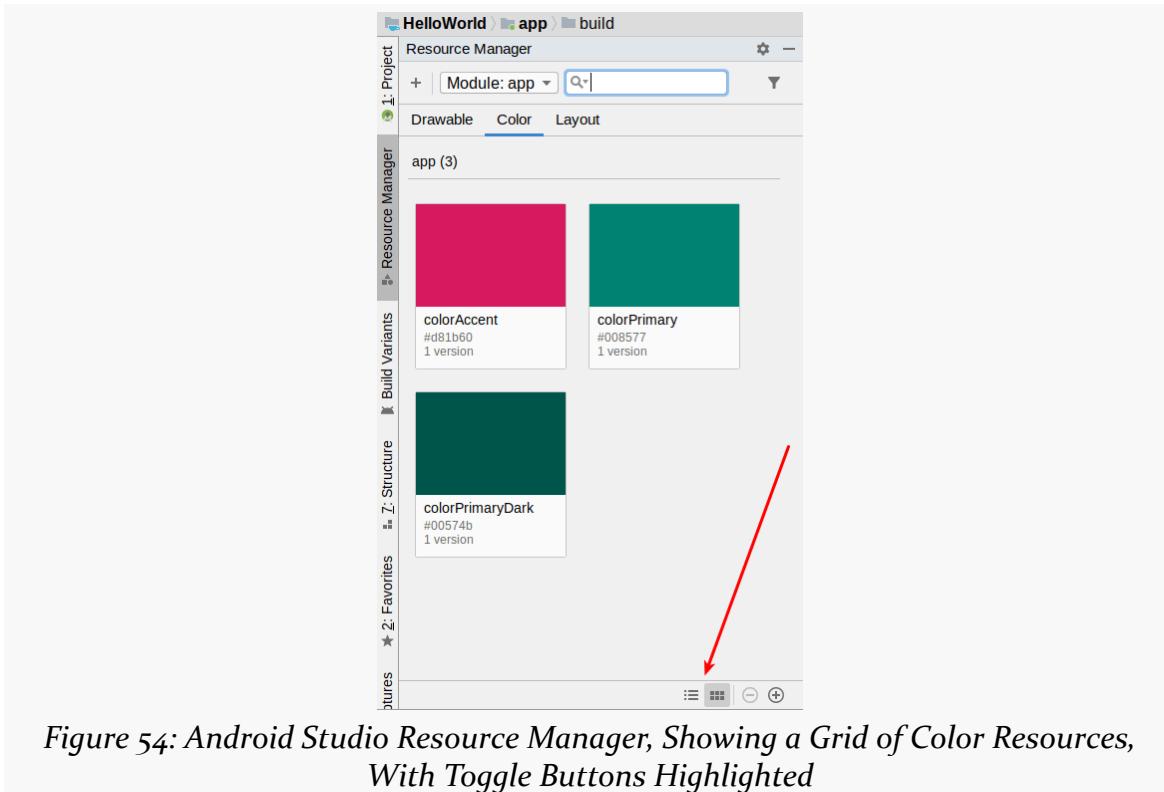


Figure 54: Android Studio Resource Manager, Showing a Grid of Color Resources, With Toggle Buttons Highlighted

Double-clicking on a resource opens up an editor window for it, while right-clicking over a resource provides options for copying it, renaming it, etc.

The + icon in the toolbar towards the top lets you add new drawable resources, using the Image Asset Wizard or Vector Asset Wizard, both of which we will see in upcoming chapters.

For larger projects, this can be useful to help you find resources. In particular, the Drawable tab can be very handy for identifying if you already have a piece of artwork in your app or whether you need to add it.

Licensed solely for use by Patrocinio Rodriguez

Inspecting Your Manifest

A key part of the foundation for any Android application is the manifest file: `AndroidManifest.xml`. This will be in your app module's `src/main/` directory (the main source set) for typical Android Studio projects.

Here is where you declare what is inside your application, such as your activities. You also indicate how these pieces attach themselves to the overall Android system; for example, you indicate which activity (or activities) should appear on the device's launcher.

When you create your application, you will get a starter manifest generated for you. For a simple application, offering a single activity and nothing else, the auto-generated manifest will require a few minor modifications, but otherwise it will be fine. Some apps will have a manifest that has 1,000+ lines. Your production Android applications probably will fall somewhere in the middle.

The Root Element

Here is the `AndroidManifest.xml` file from the starter project:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.commonsware.jetpack.hello" >

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme" >
```

INSPECTING YOUR MANIFEST

```
<activity android:name=".MainActivity" >
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />

        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
</application>

</manifest>
```

The root `<manifest>` element usually does not contain too much. It will have one or more XML namespace declarations. Here, we have just one, defining the `android` namespace, which is used for most of the attributes that you will find in the manifest file. We will see other manifests later on that have other namespace declarations (e.g., `tools`), but usually there are not too many of them.

The key attribute in the `<manifest>` element is `package`. This indicates where the build tools will generate some Java code for use by your app. We will explore that generated code [later in the book](#).

The Application Element

There can be many child elements of the root `<manifest>` element. Over the course of this book, we will see ones like `<uses-permission>` that appear in these manifests.

However, the most important child element by far is `<application>`. This describes the app that is using this manifest.

In a significant Android app, most of what goes in the manifest consists of child elements of `<application>`, such as [the `<activity>` element](#). Beyond that, the `<application>` element:

- Provides defaults for behavior of those activities, such as what theme is used to specify colors and such (`android:theme`)
- Provides details about the app that get used by other apps (e.g., `Settings`), such as the app's display name (`android:label`) and icon (`android:icon` and, sometimes, `android:roundIcon`)
- Configures overall app behavior, such as whether it handles right-to-left languages (a.k.a., `RTL`), such as Arabic and Hebrew (`android:supportsRtl`)
- Configures certain aspects of how the app integrates with the rest of the

INSPECTING YOUR MANIFEST

operating system, such as whether it wishes to participate in device-wide backups (`android:allowBackup`)

We will explore many of these attributes as we proceed in this book.

The Activity Element (And Its Children)

The children of `<application>` mostly represent the “table of contents” for the app.

Android has four major types of “components”:

- Activities, representing the UI
- Services, representing background processing that is decoupled from the UI
- Content providers, which expose databases or data streams to other apps or the operating system
- Broadcast receivers, which supply the “subscriber” side of a publish/subscribe messaging system used by apps and the operating system to communicate

Most of these will be registered in the manifest via corresponding child elements of `<application>`:

Component	Element
Activity	<code><activity></code>
Service	<code><service></code>
Content Provider	<code><provider></code>
Broadcast Receiver	<code><receiver></code>

Your app may have several of one type, such as having several activities. Your app may have none of a particular type, such as having no broadcast receivers registered in the manifest.

Our starter app has a single `<activity>` element, and nothing more:

```
<activity android:name=".MainActivity" >
    <intent-filter>
```

INSPECTING YOUR MANIFEST

```
<action android:name="android.intent.action.MAIN" />

<category android:name="android.intent.category.LAUNCHER" />
</intent-filter>
</activity>
```

`<activity>` elements have an `android:name` attribute. This will identify the Java or Kotlin class that contains the implementation of the activity. The `android:name` attribute, in this case, has a bare Java class name prefixed with a single dot (`.MainActivity`). Sometimes, you will see `android:name` with a fully-qualified class name (e.g., `com.commonsware.helloworld.MainActivity`). Sometimes, you will see just a bare Java class name (e.g., `MainActivity`). Both `MainActivity` and `.MainActivity` refer to a Java class that will be in your project's package — the one you declared in the `package` attribute of the `<manifest>` element.

Sometimes, an `<activity>` element will have an `<intent-filter>` child element describing under what conditions this activity will be displayed. Most apps will have at least one `<activity>` element that sets up your activity to appear in the launcher, so users can choose to run it. That is what this `<intent-filter>` element does, though the details of how that works are beyond the scope of this particular book. Suffice it to say that whenever you see an `<activity>` element with this particular `<intent-filter>` (an `<action>` of `android.intent.action.MAIN` and a `<category>` of `android.intent.category.LAUNCHER`), you know that this activity should appear in the launcher for the user to be able to start.

The other component elements — `<service>`, `<provider>`, `<receiver>` — will have similar characteristics:

- They all will have an `android:name` attribute, identifying the code that serves as the implementation for that component
- They might have an `<intent-filter>`
- They might have other attributes as well (e.g., `android:permission`)

Reviewing Your Gradle Scripts

In the discussion of Android Studio, this book has mentioned something called “Gradle”, without a lot of explanation.

In this chapter, the mysteries of Gradle will be revealed to you.

(well, OK, *some* of the mysteries...)

Gradle: The Big Questions

First, let us “set the stage” by examining what this is all about, through a series of fictionally-asked questions (FAQs).

What is Gradle?

[Gradle](#) is software for building software, otherwise known as “build automation software” or “build systems”. You may have used other build systems before in other environments, such as `make` (C/C++), `rake` (Ruby), Ant (Java), Maven (Java), etc.

These tools know — via intrinsic capabilities and rules that you teach them — how to determine what needs to be created (e.g., based on file changes) and how to create them. A build system does not compile, link, package, etc. applications directly, but instead directs separate compilers, linkers, and packagers to do that work.

Gradle, as used in Android Studio 3.5, uses a domain-specific language (DSL) built on top of Groovy to accomplish these tasks.

What is Groovy?

There are many programming languages that are designed to run on top of the Java VM. Kotlin is one of particular importance for Android developers. [Groovy](#) is another.

As with Java, Groovy supports:

- Defining classes with the `class` keyword
- Creating subclasses using `extends`
- Importing classes from external JARs using `import`
- Defining method bodies using braces (`{` and `}`)
- Objects are created via the `new` operator

Groovy also resembles Kotlin in some ways:

- You can have free-standing statements outside of a class
- You can use string interpolation (e.g., `"Hello, $name"` to dynamically insert a `name` value into the string)
- You can skip types on variable declarations (e.g., `def foo = 1;`, akin to `var foo = 1` in Kotlin)

What Does Android Have To Do with Gradle?

Google has published the Android Gradle Plugin, which gives Gradle the ability to build Android projects. Google is also using Gradle and the Android Gradle Plugin as the build system behind Android Studio.

Obtaining Gradle

If you will only be using Gradle in the context of Android Studio, the IDE will take care of getting Gradle for you. If, however, you are planning on using Gradle outside of Android Studio (e.g., command-line builds), you will want to consider where your Gradle is coming from. This is particularly important for situations where you want to build the app outside of an IDE, such as using a continuous integration (CI) server, like Jenkins or Circle CI.

Also, the way that Android Studio works with Gradle — called the Gradle Wrapper — opens up security issues for your development machine, if you like to download open source projects from places like GitHub and try using them.

Direct Installation

What some developers looking to use Gradle outside of Android Studio will wind up doing is installing Gradle directly.

The [Gradle download page](#) contains links to ZIP archives for Gradle itself: binaries, source code, or both.

You can unZIP this archive to your desired location on your development machine.

OS Packages

You may be able to obtain Gradle via [a package manager](#) for your particular operating system

The gradlew Wrapper

A brand new Android Studio project — and many of those that you will find in places like GitHub — will have a gradlew and gradlew.bat file in the project root, along with a gradle/ directory. This represents [the “Gradle Wrapper”](#).

The Gradle Wrapper consists of three pieces:

- the batch file (gradlew.bat) or shell script (gradlew)
- the JAR file used by the batch file and shell script (in the gradle/wrapper/ directory)
- the gradle-wrapper.properties file (also in the gradle/wrapper/ directory)

Android Studio uses the gradle-wrapper.properties file to determine where to download Gradle from, for use in your project, from the distributionUrl property in that file:

```
#Fri Aug 23 09:51:19 EDT 2019
distributionBase=GRADLE_USER_HOME
distributionPath=wrapper/dists
zipStoreBase=GRADLE_USER_HOME
zipStorePath=wrapper/dists
distributionUrl=https\://services.gradle.org/distributions/gradle-5.4.1-all.zip
```

When you create or import a project, or if you change the version of Gradle referenced in the properties file, Android Studio will download the Gradle pointed to by the distributionUrl property and install it to a .gradle/ directory (note the

REVIEWING YOUR GRADLE SCRIPTS

leading .) in your project. That version of Gradle will be what Android Studio uses.

RULE #1: Only use a distributionUrl that you trust.

If you are importing an Android project from a third party — such as something that you download from GitHub — and they contain the gradle/wrapper/gradle-wrapper.properties file, examine it to see where the distributionUrl is pointing to. If it is loading from services.gradle.org, or from an internal enterprise server, it is probably trustworthy. If it is pointing to a URL located somewhere else, consider whether you really want to use that version of Gradle, as it may have been modified by some malware author.

The batch file, shell script, and JAR file are there to support command-line builds. If you run the `gradlew` command, it will use a local copy of Gradle installed in `.gradle/` in the project. If there is no such copy of Gradle, `gradlew` will download Gradle from the distributionUrl, as does Android Studio. Note that Android Studio does not use `gradlew` for this role — that logic is built into Android Studio itself.

RULE #2: Only use a gradlew that you REALLY trust.

It is relatively easy to examine a `.properties` file to check a URL to see if it seems valid. Making sense of a batch file or shell script can be cumbersome. Decompiling a JAR file and making sense of it can be rather difficult. Yet, if you use `gradlew` that you obtained from somebody, that script and JAR are running on your development machine, as is the copy of Gradle that they install. If that code was tampered with, the malware has complete access to your development machine and anything that it can reach, such as servers within your organization.

Examining the Gradle Files

An Android Studio project usually has two `build.gradle` files, one at the project level and one at the “module” level (e.g., in the `app/` directory).

The Project-Level File

The `build.gradle` file in the project directory controls the Gradle configuration for all modules in your project. The starter project has the single `app` module, and many projects only need one module.

REVIEWING YOUR GRADLE SCRIPTS

If you downloaded the Kotlin edition of the starter project, your top-level `build.gradle` looks like this:

```
// Top-level build file where you can add configuration options common to all sub-projects/modules.

buildscript {
    ext.kotlin_version = '1.3.50'
    repositories {
        google()
        jcenter()

    }
    dependencies {
        classpath 'com.android.tools.build:gradle:3.5.0'
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
        // NOTE: Do not place your application dependencies here; they belong
        // in the individual module build.gradle files
    }
}

allprojects {
    repositories {
        google()
        jcenter()

    }
}

task clean(type: Delete) {
    delete rootProject.buildDir
}
```

If instead you have the Java-based starter project, you will have a very similar `build.gradle` file, one without the Kotlin references:

```
// Top-level build file where you can add configuration options common to all sub-projects/modules.

buildscript {
    repositories {
        google()
        jcenter()

    }
    dependencies {
        classpath 'com.android.tools.build:gradle:3.5.0'

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

allprojects {
    repositories {
        google()
        jcenter()

    }
}
```

REVIEWING YOUR GRADLE SCRIPTS

```
}
```

```
task clean(type: Delete) {
    delete rootProject.buildDir
}
```

In either case, the file contains three things (besides an opening comment):

- a `buildscript` closure
- an `allprojects` closure
- a `clean` task

In Groovy terms, a “closure” is a block of code wrapped in braces (`{ }`).

There are three main closures in the project-level `build.gradle` file.

`buildscript`

The `buildscript` closure in Gradle is where list sources of plugins that you want to use in the project. Hence, here you are not configuring your *project* so much as you are configuring *the build itself*.

The `repositories` closure inside the `buildscript` closure indicates where plugins can come from. Here, `jcenter()` is a built-in method that teaches Gradle about [JCenter](#), a popular location for obtaining open source libraries. Similarly, `google()` is a built-in method that teaches Gradle about a site where it can download plugins from Google.

The `dependencies` closure indicates libraries that contain Gradle plugins. In the Kotlin edition of the `build.gradle` file, there are two such dependencies:

- `com.android.tools.build:gradle`, which is where the Android Plugin for Gradle comes from, which teaches Gradle how to build Android apps
- `org.jetbrains.kotlin:kotlin-gradle-plugin`, which teaches Gradle how to compile Kotlin source code

(the Java edition of the project will lack the Kotlin plugin)

The identifiers of the libraries (e.g., `com.android.tools.build:gradle`) are followed by a version number, indicating what particular version of those libraries should be used. From time to time, Android Studio will ask you to update those versions, just as it will ask on occasion for you to upgrade the version of Gradle specified in

REVIEWING YOUR GRADLE SCRIPTS

`gradle-wrapper.properties`. Google maintains a page [listing the Gradle versions supported by each Android Gradle Plugin version](#)

`allprojects`

The `allprojects` closure says “apply these settings to all modules in this project”. Here, we are setting up `jcenter()` and `google()` as places to find libraries used in any of the modules in our project. We will use lots of libraries in our projects — having these “repositories” set up in `allprojects` makes it simpler for us to request them. We will talk a bit more about libraries [later in this chapter](#).

`clean`

Like many build systems, Gradle is based around tasks. Plugins and your own Gradle files teach Gradle about various tasks that it should be able to perform when requested. The `clean()` task in the top-level `build.gradle` file is one such task. As written, this task is almost useless, and it is unclear why Google includes it, other than perhaps to point out that you are able to define custom tasks.

The Module-Level Gradle File

In your app/ module, you will also find a `build.gradle` file. This has settings unique for this module, independent of any other module that your project may have in the future.

The Kotlin project’s edition of `app/build.gradle` includes a number of Kotlin references:

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

apply plugin: 'kotlin-android'

apply plugin: 'kotlin-android-extensions'

android {
    compileSdkVersion 29
    buildToolsVersion "29.0.2"
    defaultConfig {
        applicationId "com.commonsware.jetpack.hello"
        minSdkVersion 21
        targetSdkVersion 29
        versionCode 1
        versionName "1.0"
        testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
    }
    buildTypes {
```

REVIEWING YOUR GRADLE SCRIPTS

```
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'), 'proguard-rules.pro'
        }
    }

dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
    implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlin_version"
    implementation 'androidx.appcompat:appcompat:1.0.2'
    implementation 'androidx.core:core-ktx:1.0.2'
    implementation 'androidx.constraintlayout:constraintlayout:1.1.3'
    testImplementation 'junit:junit:4.12'
    androidTestImplementation 'androidx.test:runner:1.2.0'
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.2.0'
}
```

...while the Java edition does not:

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

android {
    compileSdkVersion 29
    buildToolsVersion "29.0.2"
    defaultConfig {
        applicationId "com.commonsware.jetpack.hello"
        minSdkVersion 21
        targetSdkVersion 29
        versionCode 1
        versionName "1.0"
        testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
    }
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'), 'proguard-rules.pro'
        }
    }
}

dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
    implementation 'androidx.appcompat:appcompat:1.0.2'
    implementation 'androidx.constraintlayout:constraintlayout:1.1.3'
    testImplementation 'junit:junit:4.12'
    androidTestImplementation 'androidx.test:runner:1.2.0'
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.2.0'
}
```

The android closure contains all of the Android-specific configuration information. The Android plugin will use this closure, where the plugin itself comes from the apply plugin: 'com.android.application' line at the top, coupled with the classpath line from the project-level build.gradle file. We will explore some of the specific values defined in this closure [in the next section](#).

REVIEWING YOUR GRADLE SCRIPTS

This build.gradle file also has a dependencies closure. Whereas the dependencies closure in the buildscript closure in the top-level build.gradle file is for libraries used by the build process, the dependencies closure in the module's build.gradle file is for libraries used by your code in that module. We will talk more about libraries [later in the chapter](#).

Requesting Plugins

The first lines in app/build.gradle usually request various plugins. The lines that you added to the top-level build.gradle file specify *sources* of plugins, but those libraries can have many different plugins, and you only need some of them. Plus, if your project grows and you have more modules than just app, you might need a different mix of plugins per module.

As a result, each module's build.gradle file starts off with a series of `apply plugin` statements to indicate what plugins are needed:

```
apply plugin: 'com.android.application'  
apply plugin: 'kotlin-android'  
apply plugin: 'kotlin-android-extensions'
```

Both the Java and the Kotlin editions of the starter project will request the `com.android.application` plugin. This teaches Gradle how to build Android apps. There are other options here, such as `com.android.library` to teach Gradle how to build an Android library, but nearly every project will have at least one module using `com.android.application`.

Kotlin-based projects will also request the `kotlin-android` and `kotlin-android-extensions` plugins. These teach Gradle how to compile Kotlin code, particularly in the context of building an Android application.

Android Plugin for Gradle Configuration

One of the most important areas for configuration in app/build.gradle is inside the `android` closure. That configures the Android Plugin for Gradle, teaching it the details of how you want your app to be assembled from its source code, resources, etc.

REVIEWING YOUR GRADLE SCRIPTS

In theory, this closure can get *very* complex. In practice, most apps will configure a few things, as the starter app does:

```
android {  
    compileSdkVersion 29  
    buildToolsVersion "29.0.2"  
    defaultConfig {  
        applicationId "com.commonsware.jetpack.hello"  
        minSdkVersion 21  
        targetSdkVersion 29  
        versionCode 1  
        versionName "1.0"  
        testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"  
    }  
    buildTypes {  
        release {  
            minifyEnabled false  
            proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'), 'proguard-rules.pro'  
        }  
    }  
}
```

Package Name and Application ID

In [the previous chapter](#), we saw that there is a package attribute on the root <manifest> element of a manifest, and that indicates where some code-generated classes will go.

The package value also serves as the default value for your application ID. However, you can override it, as the starter app does, via an applicationId statement in the defaultConfig closure inside that android closure.

The application ID is a unique identifier for our app, such that:

- no two apps can be installed on the same device at the same time with the same application ID
- no two apps can be uploaded to the Play Store with the same application ID (and other distribution channels may have the same limitation)

Convention says that the application ID starts with a reversed edition of some domain name that you control, to reduce the likelihood of an accidental collision with the application ID of some other developer.

compileSdkVersion, minSdkVersion, and targetSdkVersion

Back in the chapter introducing resources, we discussed [the concept of API levels](#). API levels are integers, with higher numbers indicating newer versions of Android. A

REVIEWING YOUR GRADLE SCRIPTS

new API level is created for:

- Every major version of Android (e.g., Android 8.0 is API Level 26)
- Every minor version of Android (e.g., Android 8.1 is API Level 27)
- Some patch versions of Android (e.g., Android 4.0 was API Level 14, but Android 4.0.3 was API Level 15)

We use those API levels in three key places in the module's `build.gradle` file: `compileSdkVersion`, `minSdkVersion`, and `targetSdkVersion`.

`compileSdkVersion` indicates what version of Android do we want to compile against. Classes, methods, and other symbols that existed in Android at that time (or from before) will be available to us at compile time, but newer things will not. Usually, therefore, we set the `compileSdkVersion` to be a fairly modern API level, such as the latest production version of Android.

`minSdkVersion` indicates what is the oldest version of Android you are willing to support. The value of the attribute is an integer representing the Android [API level](#). So, if you are only supporting your app on Android 5.0 and newer versions of Android, you would set your `minSdkVersion` to be 21. Older devices will be incapable of installing your app, and your app will not appear in the Play Store app for devices running an older version of Android.

`targetSdkVersion` indicates what version of Android you are thinking of as you are writing your code. If your application is run on a newer version of Android, Android may do some things to try to improve compatibility of your code with respect to changes made in the newer Android. However, from a practical standpoint, nowadays the `targetSdkVersion` usually is the same value as the `compileSdkVersion` — we update both of them at the same time to the same value.

Version Code and Version Name

The `defaultConfig` closure has `versionCode` and `versionName` properties. These two values represent the versions of your application.

The `versionName` value is what the user will see for a version indicator in places like the Settings app and the Play Store. This can be whatever string you want, using whatever naming or numbering system that you want. However, for customer support purposes, you should have *some* system that varies by release, rather than using the same string all of the time.

REVIEWING YOUR GRADLE SCRIPTS

The `versionCode`, on the other hand, must be an integer, and newer versions must have higher version codes than do older versions. Android and the Play Store will compare the version code of a new APK to the version code of an installed application to determine if the new APK is indeed an update. The typical approach is to start the version code at 1 and increment it with each production release of your application, though you can choose another convention if you wish. During development, you can leave these alone, but when you move to production, these attributes will matter greatly.

Other Stuff in the android Closure

The `android` closure has a `testInstrumentationRunner` statement — we will explore that more [in an upcoming chapter, to see how testing works](#).

The `android` closure also has a `buildTypes` closure. This provides specific configuration for different “build types”, such as `debug` for development builds and `release` for production builds. The defaults provided in the starter project are fine for many basic apps.

Libraries and Dependencies

Roughly speaking, the code and assets that make up an app come from three sources:

1. The source that you tend to focus on personally is the code that you and people that you know are writing for this app.
2. The source that comes from your `compileSdkVersion`, representing the Android SDK that you are linking to.
3. Everything else, generally referred to as dependencies. These are libraries, written by Google, independent Android developers, major firms, and so on. Every modern Android app uses libraries, and bigger apps use more libraries.

From a pure technical standpoint, most dependencies are listed in `build.gradle` files in `dependencies` closures. We have seen two of these in this chapter.

One `dependencies` closure appears in the project-level `build.gradle` file, inside of a `buildscript` closure:

```
dependencies {  
    classpath 'com.android.tools.build:gradle:3.5.0'
```

REVIEWING YOUR GRADLE SCRIPTS

```
classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
// NOTE: Do not place your application dependencies here; they belong
// in the individual module build.gradle files
}
```

Those list places where Gradle plugins come from. You are always depending upon the Android Gradle Plugin, and some other developers publish Gradle plugins that you may elect to use in the future. Kotlin users will also use a Kotlin plugin.

However, the dependencies closure that we tend to think about the most is the one in our module's build.gradle file, such as app/build.gradle, such as this one from the starter project:

```
dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
    implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlin_version"
    implementation 'androidx.appcompat:appcompat:1.0.2'
    implementation 'androidx.core:core-ktx:1.0.2'
    implementation 'androidx.constraintlayout:constraintlayout:1.1.3'
    testImplementation 'junit:junit:4.12'
    androidTestImplementation 'androidx.test:runner:1.2.0'
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.2.0'
}
```

Here, there are three types of statements:

- `implementation` says “here is a dependency that I want to use for my actual app”
- `androidTestImplementation` says “here is a dependency that I want to use for testing”
- `testImplementation` says “here is a dependency that I want to use for... a slightly different type of testing”

(we will explore the differences between those in [an upcoming chapter](#))

Our starter project has eight statements in app/build.gradle attempting to pull in dependencies:

- Three are tied to testing. We will be discussing those [soon](#).
- Two are tied to Kotlin (`org.jetbrains.kotlin:kotlin-stdlib-jre7` and `androidx.core:core-ktx`) and will not be seen in pure-Java projects.
- The first line — the `fileTree(dir: 'libs', include: ['*.jar'])` one — allows you to add plain JARs to your project, by putting them in a `libs/`

REVIEWING YOUR GRADLE SCRIPTS

directory.

- The constraintlayout and appcompat dependencies are specifically part of Jetpack and are foundations for modern Android UI development. We will be spending quite a bit of the book going over what these offer and how you use them.

Most likely, you will be adding other similar statements to this set, plus perhaps deleting ones that you are not actually using (e.g., the `fileTree()` one). We will see how to add other libraries as the book progresses, as many of the things that are common in Android app development require additional libraries.

Inspecting the Compiled App

Our starter project has a bit of code, a bunch of resources, a manifest, and Gradle build instructions. And, we can see the app running on a device or emulator.

However, Android is not running directly that bit of code and other stuff. Android Studio compiles the things in our project into something that Android then runs.

In this chapter, we will take a brief look at what Android Studio is building for us and how we can examine that output, beyond running it on Android.

What We Build

From the user's standpoint, we are building an app. Usually, that is true. The story can get complicated with specialized scenarios like Android Auto and Android Wear, but we can ignore those for the time being.

From a programming standpoint, what you are creating is an APK, perhaps by way of an app bundle.

APKs

An APK is the Android executable format. It is what gets installed on an Android device, and it is what the device runs.

An APK is a ZIP archive. You can unpack one using your favorite unZIP tool, though we will see a better option for many cases [later in this chapter](#).

The APK will contain:

INSPECTING THE COMPILED APP

- Your compiled code
- Your resources, usually in their own “compiled” format
- Your manifest, also in a “compiled” format
- Other miscellaneous files

This includes not only the things that you create yourself but also the stuff that you wind up using from libraries that are part of your app module.

App Bundles

The APK not only is what an Android device runs, but usually it is what Android app developers publish. You use Android Studio to create the APK from your project, and you upload that APK to the Play Store or other app distribution channels. The Play Store or other site then distributes the APK to users.

You will hear Google talk about “app bundles” as an alternative. App bundles are designed for large, complex apps, where only part of what goes in an APK will be relevant for any given device. For example, a major brand APK may have strings translated to many languages, but the user typically uses only one or two languages. Having translations in other languages just takes up disk space and adds no value to that individual user. App bundles allow Google to craft tailored APKs on your behalf that match the needs of individual devices and users.

However, now you are no longer in control over what is in your app. Google is. While Google likes to comment on what they can remove from your app (e.g., unnecessary languages), they do not discuss whether they might *add* anything to your app or otherwise modify its behavior. That could be for their benefit or for the benefit of some third party.

You will need to decide for yourself whether the benefits of app bundles outweigh the risks.

Where They Go

When you run your app in Android Studio, it creates an APK to install on the device, at least initially. Instant Run, if you choose to leave it enabled, makes the story a bit more complex, but there is always an APK at the beginning.

INSPECTING THE COMPILED APP

That APK will wind up inside of the module's build/ directory. build/ contains all sorts of outputs of the build process, the APK being chief among them:

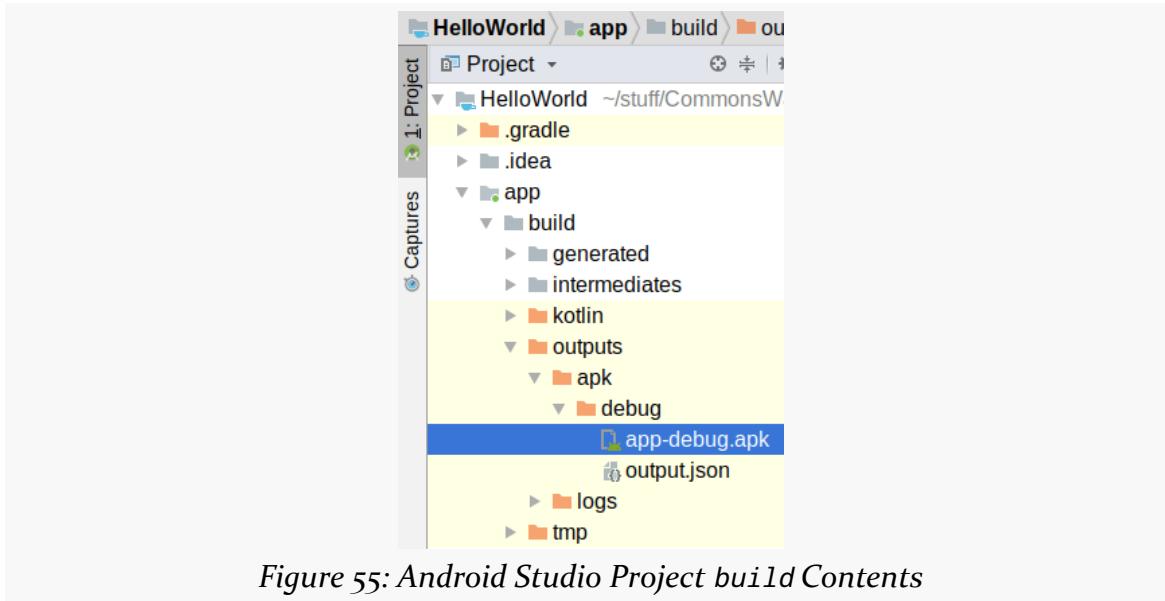


Figure 55: Android Studio Project build Contents

Specifically, when you start off, you will find the APK in build/outputs/debug/app-debug.apk. By default, the filename will be based on your Gradle module name, so an app/ module winds up with an APK with app in the name. And, by default, you are creating a debug build — the sort of build that you as a developer will use — and so the APK winds up in a debug/ directory and has debug in the name.

Building the APK

You can manually build the APK at any time through the Build menu in Android Studio. For example, you might want to do that before trying to [analyze the APK](#).

For creating an ordinary debug APK, choose Build > Build Bundles(s) / APK(s) > Build APK(s) from the Android Studio main menu. After a few moments, a “toast”-style popup window should appear announcing the results:



Figure 56: Android Studio, Toast Showing Build Results

Analyzing the APK

One of the links in that toast is to “analyze” the APK.

Android Studio offers an APK Analyzer tool. Mostly, it exists to help developers see exactly what is in the APK, blended from all of the possible sources (your code, library code, etc.). For developers looking to reduce the size of the APK, the APK Analyzer is great for seeing where the space goes.

To bring up the APK Analyzer, either click that “analyze” link from a fresh build, or choose Build > Analyze APK at any time. The latter will bring up a typical file chooser dialog for you to navigate through the build/ directory to select the APK that you want to analyze:

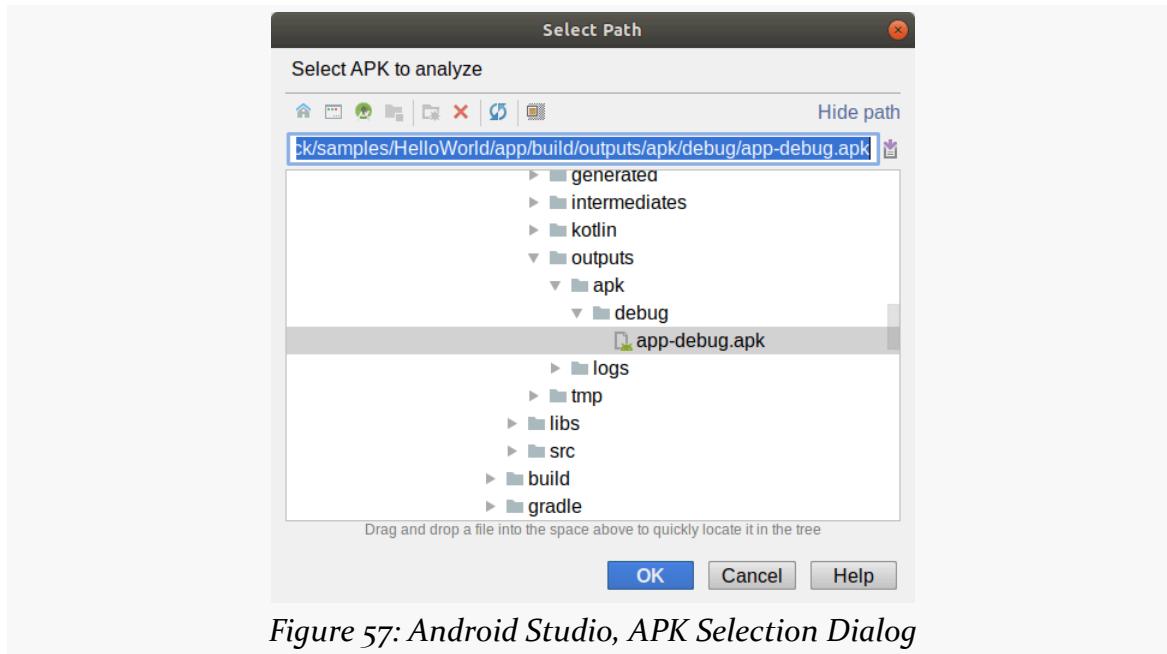


Figure 57: Android Studio, APK Selection Dialog

INSPECTING THE COMPILED APP

The APK Analyzer then opens in a fresh tab:

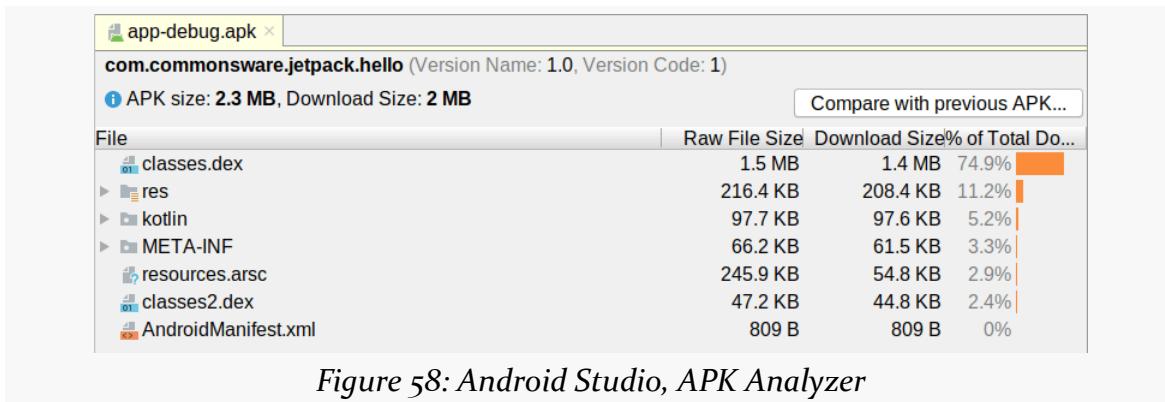


Figure 58: Android Studio, APK Analyzer

The starter project's APK is a 2.3MB file. This may seem rather large for an app that does not really do much. That is because:

- Google's starter projects include a lot of [libraries](#), and
- A normal debug build does not eliminate unused stuff from those libraries

We will look into how to address that latter problem [much later in the book](#).

The table shows the breakdown of where the space goes. Of note:

- The .dex files — and, where relevant, the kotlin/ directory — represent your compiled code, both that you wrote, what was code-generated for you by Android Studio, and what you get from libraries
- The res/ directory and the resources.arsc file represent your resources
- The AndroidManifest.xml file is your manifest

We will explore these things much more [later in the book](#) and help you see how you might make your app smaller.

Licensed solely for use by Patrocinio Rodriguez

Touring the Tests

When we write Android apps, a chunk of our time is spent testing those apps. Some of that testing is manual: poking at the UI and seeing if everything works as expected. But some of that testing is automated, with test classes that test our “real” classes and confirm that everything is OK.

With that in mind, let’s take a look at the types of tests that we have in the starter project and how to run them.

Instrumented Tests

There are two major types of test in an Android app:

- Instrumented tests, which run in Android on a device or emulator
- Unit tests, which run on your development machine or similar places

Unit tests run much faster, but they cannot test as much, because they do not have access to everything inside of Android. For example, while we could test our ability to talk to a Web service from unit tests, we cannot test our ability to get GPS locations using Android APIs from unit tests. For those, we need instrumented tests. Similarly, most automated UI testing needs instrumented tests, as the Android UI system is only really available in Android.

Since they are more flexible, and since test speed only becomes a major issue with larger projects, let’s focus first on instrumented tests.

Where They Run

As noted above, instrumented tests will run on an Android device or emulator. For

TOURING THE TESTS

your own personal test runs, you can use the same devices or emulators that you use for manually running the app.

Projects that employ continuous integration (CI) servers will need to configure them to support running tests on server-hosted emulators. Some hosted CI services — such as [CircleCI](#) — have that capability readily available to you. For self-hosted CI servers, there should be recipes available to teach you how to configure them for Android app testing.

What You Can Test

Because you are running the tests in an actual Android environment, you can test anything that you want. You have the full Android SDK at your disposal.

However, from a practical standpoint, there will be limits as to what you can test:

- Emulators do not emulate everything about hardware. For example, you will not be able to test readings that you get from nearby cell towers, as an emulator is not in communication with any actual cell towers.
- You want your tests to be repeatable. Hence, even on hardware, you may need to limit testing what you really get from the hardware, as you do not control that hardware and what it might return. For example, while in theory you could test getting actual location data via GPS from a device, you cannot guarantee the precise values that will get returned, as GPS is inexact by its very nature.
- Any given device has one set of hardware characteristics. Any given emulator will mimic one set of hardware characteristics. Testing things that vary based on hardware characteristics will require multiple test runs across a fleet of devices or emulators that will reflect the varying characteristics.

What the Starter Project Has

The starter project not only has a “hello, world” sort of UI for you, but it has a similar instrumented test set up, ready for you to run.

Or, at least, in theory it would be ready for you to run. Unfortunately, you will not be able to run it in its current form, as we will see shortly.

TOURING THE TESTS

The androidTest Source Set

Instrumented test code resides in an androidTest/ directory:

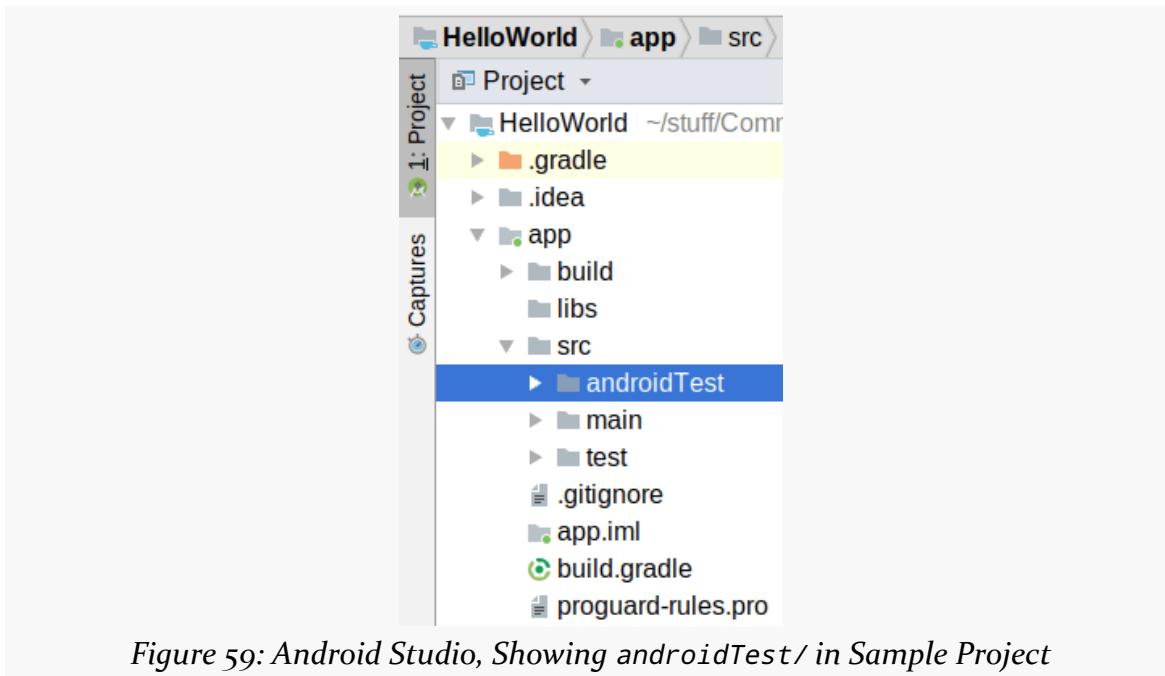


Figure 59: Android Studio, Showing androidTest/ in Sample Project

This is a peer to the main/ directory that contains your “real” application code. androidTest/ is a “source set” that will be used only when running instrumented tests. The stuff in the androidTest/ source set will *not* be included in your app when you ship it.

TOURING THE TESTS

The Test Class

Inside of there you will find a `java/` directory, with a Java package matching the application ID of your app, and an `ExampleInstrumentedTest` Java or Kotlin file:

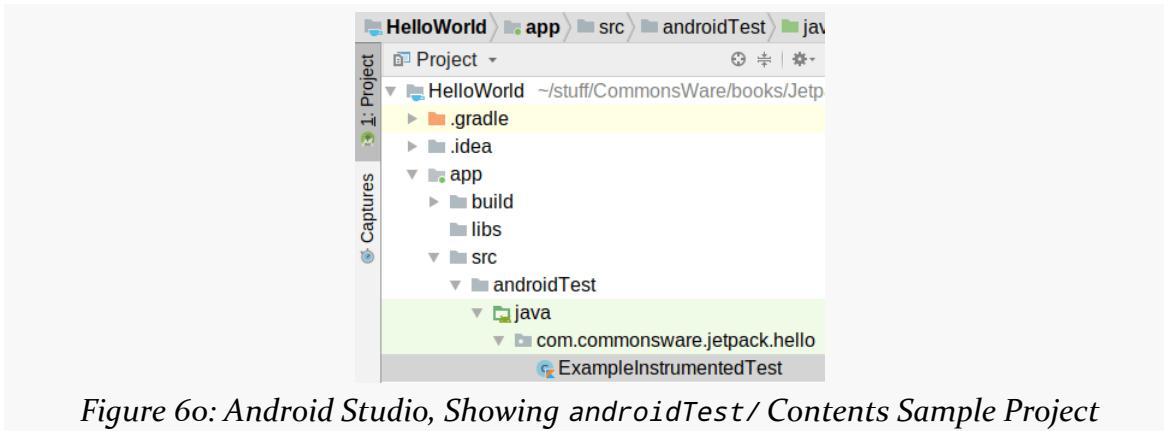


Figure 6o: Android Studio, Showing `androidTest/` Contents Sample Project

When you create a new project, and you choose whether or not to have Kotlin support, that choice will determine not only whether your `MainActivity` is in Java or Kotlin, but also whether your test code is in Java or Kotlin.

The Kotlin class is fairly short:

```
package com.commonware.jetpack.hello

import androidx.test.platform.app.InstrumentationRegistry
import androidx.test.ext.junit.runners.AndroidJUnit4

import org.junit.Test
import org.junit.runner.RunWith

import org.junit.Assert.*

/**
 * Instrumented test, which will execute on an Android device.
 *
 * See [testing documentation](http://d.android.com/tools/testing).
 */
@RunWith(AndroidJUnit4::class)
class ExampleInstrumentedTest {
    @Test
    fun useApplicationContext() {
        // Context of the app under test.
        val applicationContext = InstrumentationRegistry.getInstrumentation().targetContext
        assertEquals("com.commonware.jetpack.hello", applicationContext.packageName)
    }
}
```

TOURING THE TESTS

The Java equivalent is not much longer:

```
package com.commonsware.jetpack.hello;

import android.content.Context;
import androidx.test.platform.app.InstrumentationRegistry;
import androidx.test.ext.junit.runners.AndroidJUnit4;

import org.junit.Test;
import org.junit.runner.RunWith;

import static org.junit.Assert.*;

/**
 * Instrumented test, which will execute on an Android device.
 *
 * @see <a href="http://d.android.com/tools/testing">Testing documentation</a>
 */
@RunWith(AndroidJUnit4.class)
public class ExampleInstrumentedTest {
    @Test
    public void useApplicationContext() {
        // Context of the app under test.
        Context applicationContext =
            InstrumentationRegistry.getInstrumentation().getTargetContext();

        assertEquals("com.commonsware.jetpack.hello", applicationContext.getPackageName());
    }
}
```

The Annotations

The Java and Kotlin editions of the test class are equivalent, other than language syntax.

Both have a single class, named `ExampleInstrumentedTest`, annotated with a `@RunWith(AndroidJUnit4::class)` annotation. Android presently uses [JUnit 4](#) for instrumented tests. This annotation tells JUnit — and, more importantly, some Android Studio stuff for running tests — that this class contains test code that should be run as part of an instrumented test.

Both editions of `ExampleInstrumentedTest` have one method (or function, in Kotlin). It is called `useApplicationContext()`, and it is marked with the `@Test` annotation. A test class can contain one or more of these `@Test` methods/functions. When it comes time to run the tests, Android Studio will:

TOURING THE TESTS

- Create an instance of your test class
- Call one of the @Test methods/functions on that instance
- Create another instance of your test class
- Call another of the @Test methods/functions on that new instance
- And so on, until all of the @Test methods/functions have been executed

Um, Why Am I Seeing Errors?

In an ideal world, when you create a new project using the IDE, the new project contains correct code.

This is not an ideal world.

In Android Studio 3.5, when you create a new project, [the IDE creates an invalid instrumented test](#). The code refers to a class (androidx.test.ext.junit.runners.AndroidJUnit4) that does not exist. When there is a compile error like this, you will see lots of red in your source editor:

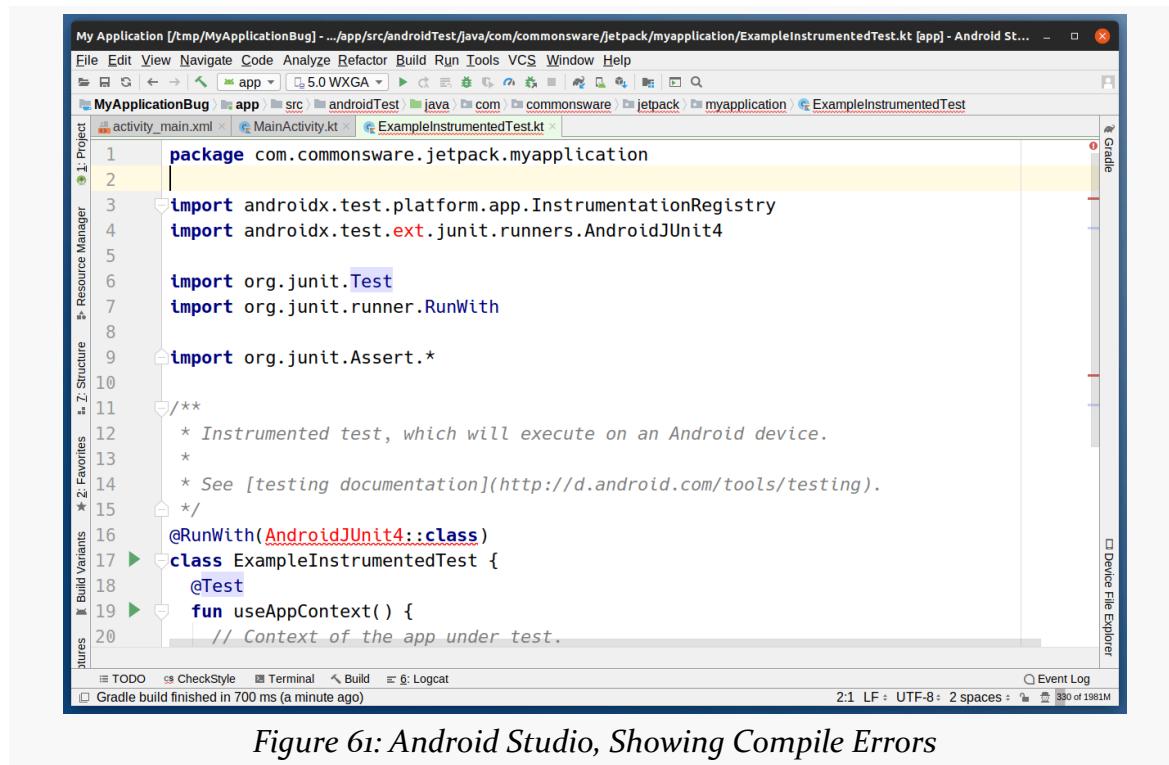


Figure 61: Android Studio, Showing Compile Errors

The simple fix is to replace the broken import statement with one for androidx.test.runner.AndroidJUnit4, which is available in the project. That, in

TOURING THE TESTS

turn, gives us a strikethrough on our use of AndroidJUnit4:

```
16  @RunWith(AndroidJUnit4::class)
17 > class ExampleInstrumentedTest {
18   @Test
19 >   fun useApplicationContext() {
20     // Context of the app under test.
21     val applicationContext = InstrumentationRegistry.getTargetContext()
22     assertEquals("com.commonsware.jetpack.hello", applicationContext.packageName)
23   }
24 }
25 }
```

Figure 62: Android Studio, Showing Strikethrough Formatting

Strikethrough marks things that are deprecated. In Android, “deprecated” means “we have a better solution that you should consider using”. In this case, Google is migrating some of their test classes to new packages and class names. The new-project wizard from time to time generates deprecated code, or uses pre-release dependencies in the build.gradle files that it generates. For the purposes of showing you Android Studio, having these deprecated-code uses is fine. For a real project, you should read the documentation for the deprecated classes, find out what the replacements are, and aim to migrate to those replacements when appropriate.

A better fix is to modify the dependencies to pull in the version of AndroidJUnit4 that the class referred to originally. androidx.test.ext:junit contains androidx.test.ext.junit.runners.AndroidJUnit4, so adding a dependency on that would allow the original import statement to work. We will see this approach in [the main chapter on testing](#).

The Test Code

So, if we ignore the error or strikethrough for now... what is useApplicationContext() testing?

In truth, it is not testing very much.

We will explore what a Context is [a bit later in the book](#). For the moment, take it on faith that this code is:

- Finding out what our application ID is, by calling getPackageName() on a

Context

- Confirming whether it matches the expected value of
`com.commonsware.jetpack.hello`

`assertEquals()` is supplied by JUnit 4 and will fail the test if the two values are not equal.

We will explore much more about JUnit 4 and how to write more elaborate tests [much later in the book](#).

How You Run Them

For a single test method or function, you will notice a green triangle “run” icon in the “gutter” area of the code editor:



```
16  | @RunWith(AndroidJUnit4::class)
17  | > class ExampleInstrumentedTest {
18  |   | @Test
19  |   | > fun useApplicationContext() {
20  |     | // Context of the app under test.
21  |     | val applicationContext = InstrumentationRegistry.getTargetContext()
22  |     | assertEquals("com.commonsware.jetpack.hello", applicationContext.packageName)
23  |   }
24  | }
25 }
```

Figure 63: Android Studio, Showing Instrumented Test in Kotlin

Clicking that will allow you to run that individual test method/function. Or, optionally, you will be able to [debug that test method](#).

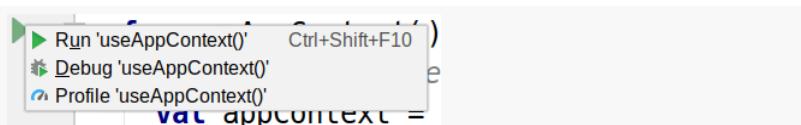


Figure 64: Android Studio, Showing Pop-Up Menu for Running a Test Function

TOURING THE TESTS

Similarly, there is a “run” icon in the gutter next to the class name, to run all of the test functions in the Kotlin test class. For a Java class, the class-level test icon is a “double-run” pair of overlapping green triangles:



```
@RunWith(AndroidJUnit4.class)
public class ExampleInstrumentedTest {
    @Test
    public void useApplicationContext() {
        // Context of the app under test.
        Context applicationContext=InstrumentationRegistry.getTargetContext();

        assertEquals("com.commonsware.jetpack.hello", applicationContext.getPackageName());
    }
}
```

Figure 65: Android Studio, Showing Instrumented Test in Java

(Why is there a difference? [Ask Google.](#))

For smaller projects, those may suffice. For larger projects, you can set up a custom “run configuration” that can run all of the instrumented tests in your project, for example. We will see this [much later in the book](#).

What the Test Results Look Like

When you run a test method or a test class, rather than focusing on the output in the emulator or device, you will focus instead on the “Run” view in Android Studio. This will show you which tests succeeded and which tests failed.

TOURING THE TESTS

The nice people who created Android Studio elected to write tests that succeed, and Android Studio's output will reflect that:



A screenshot of the Android Studio interface, specifically the 'Run' tab. The title bar says 'Run: ExampleInstrumentedT...'. Below it, there are several icons: a green circle with a checkmark, a brown circle with a play button, a downward arrow, a left arrow, a right arrow, a magnifying glass, a gear, and a refresh symbol. To the right of these icons, it says 'Tests passed: 1 of 1 test - 0 ms'. Below this, a message 'All Tests Passed' is displayed with a green checkmark icon. To the right of the message, the text 'Testing started at 1:30 PM ...' is shown. The main content area contains a log of command-line output:

```
10/27 13:30:35: Launching
ExampleInstrumentedT...
$ adb push /home/mmurphy/stuff/CommonsWare
/books/Jetpack/samples/HelloWorld/app/build
/outputs/apk/debug/app-debug.apk
/data/local/tmp/com.commonsware.jetpack
.hello
$ adb shell pm install -t -r
"/data/local/tmp/com.commonsware.jetpack
.hello"
Success
APK installed in 2 s 359 ms
$ adb push /home/mmurphy/stuff/CommonsWare
/books/Jetpack/samples/HelloWorld/app/build
/outputs/apk/androidTest/debug/app-debug
-androidTest.apk /data/local/tmp/com
.commonsware.jetpack.hello.test
$ adb shell pm install -t -r
"/data/local/tmp/com.commonsware.jetpack
.hello.test"
Success
APK installed in 991 ms
Running tests

$ adb shell am instrument -w -r -e debug
false -e class 'com.commonsware.jetpack
.hello.ExampleInstrumentedTest' com
.commonsware.jetpack.hello.test/android
.support.test.runner.AndroidJUnitRunner
Client not ready yet..
Started running tests
Tests ran to completion.
```

Figure 66: Android Studio, Showing Successful Test Results

Note the “All Tests Passed” message with the green checkmark-in-circle icon.

TOURING THE TESTS

If a test fails — such as a modified version of the sample’s test class that compares the application ID to this.is.wrong — you will see the failed test in a tree on the left and details of what went wrong on the right:

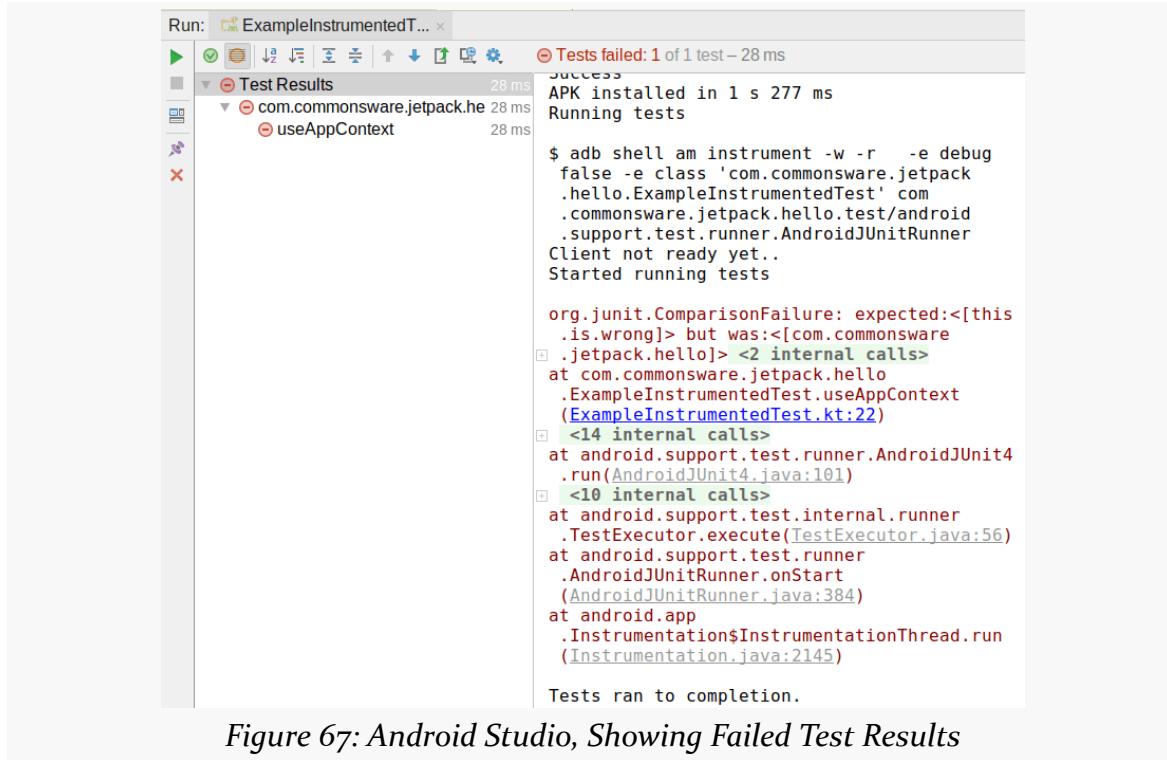


Figure 67: Android Studio, Showing Failed Test Results

JUnit assertions, such as `assertEquals()`, will provide some details as to what went wrong, shown in a stack trace in the test output:

```
org.junit.ComparisonFailure: expected:<[this.is.wrong]> but  
was:<[com.commonware.jetpack.hello]>
```

Here, we see that we expected `this.is.wrong`, but we instead got `com.commonware.jetpack.hello`, and so the test failed.

About That `testInstrumentationRunner`

Back in [the chapter on Gradle](#), we saw this line in the `defaultConfig` closure:

```
testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
```

The `testInstrumentationRunner` indicates what code should be used to execute the

TOURING THE TESTS

JUnit tests themselves. The runner shown here is the standard runner for instrumented tests. Various third-party testing tools might have you replace this value with some class from their library, so this is not *always* the test runner that we use.

The androidTestImplementation Dependencies

Such a library would also show up as an `androidTestImplementation` dependency in the list of dependencies for the module. We have a couple of those already:

```
androidTestImplementation 'androidx.test:runner:1.2.0'  
androidTestImplementation 'androidx.test.espresso:espresso-core:3.2.0'
```

The one that is really required is `androidx.test:runner`. This is the library that supplies the `AndroidJUnitRunner` class and other core classes for writing and running our instrumented tests.

The other one is `androidx.test.espresso:espresso-core`. This is the core of Espresso, a powerful library for writing GUI tests. However, our existing instrumented test does not actually *use* Espresso, so this particular dependency is unnecessary at the moment. We will cover Espresso [much later in the book](#).

Unit Tests

The other type of tests are unit tests. They too use JUnit4, which causes some confusion when developers try to determine the difference between the instrumented tests and unit tests.

Where They Run

Unit tests run in Windows, macOS, or Linux, not Android. Unit tests are the same sorts of tests that you would run in an ordinary Java or Kotlin project that had no ties at all to building Android apps.

You can run unit tests on your own development machine. You can also arrange to run unit tests on a CI server or some similar environment.

Because unit tests avoid a lot of the work in getting code over into an Android environment, unit tests can run more quickly. Plus, a development machine or CI server is likely to be faster than Android hardware, and even the Android emulator

adds some amount of runtime overhead.

What You Can Test

You can test your non-Android business logic fairly easily, just using standard JUnit tests. However, any code that uses Android-related classes is going to have a problem, though.

You can get past this somewhat by the use of mocking engines, such as Mockito. They allow you to create “fake” Android objects based on real Android classes, where you teach the mocks how to respond to particular method calls.

What the Starter Project Has

Our starter project has a similar set of stuff for unit tests as it did for instrumented tests, with some modifications.

The test Source Set

Whereas instrumented tests go in an androidTest/ source set, unit tests go in a test/ source set:

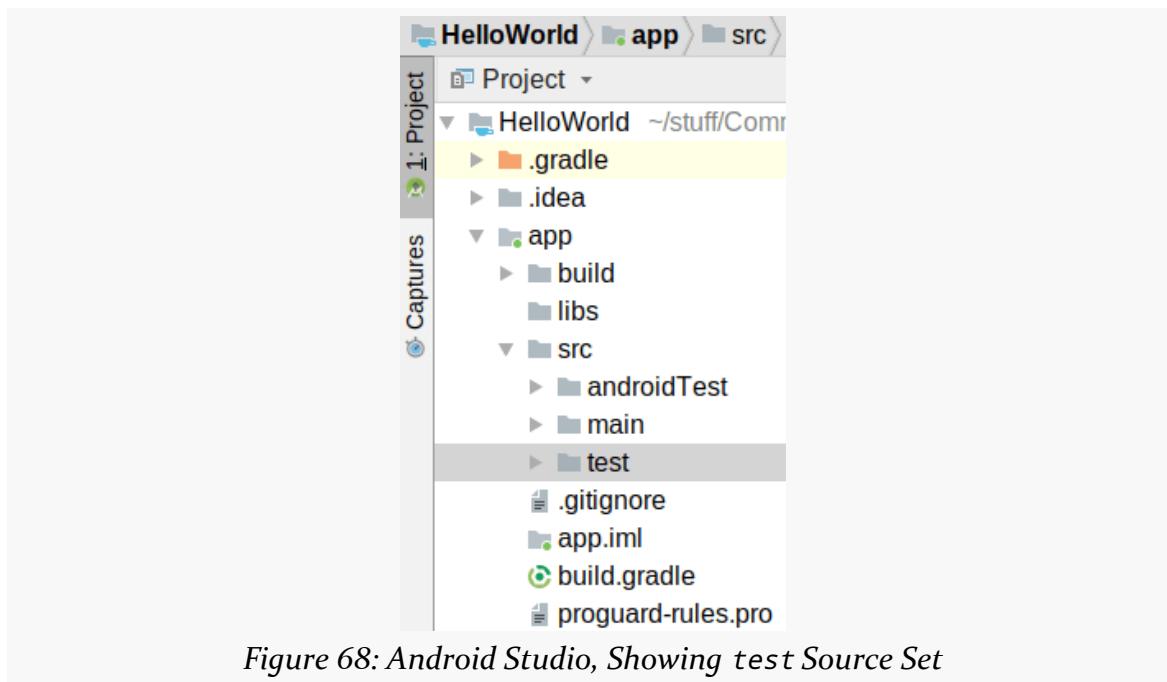


Figure 68: Android Studio, Showing test Source Set

TOURING THE TESTS

The Test Class

In there, as with the androidTest/ source set, there is a java/ directory and a com.commonware.jetpack.hello package. And, once again, there is a single class, though this time it is named ExampleUnitTest:

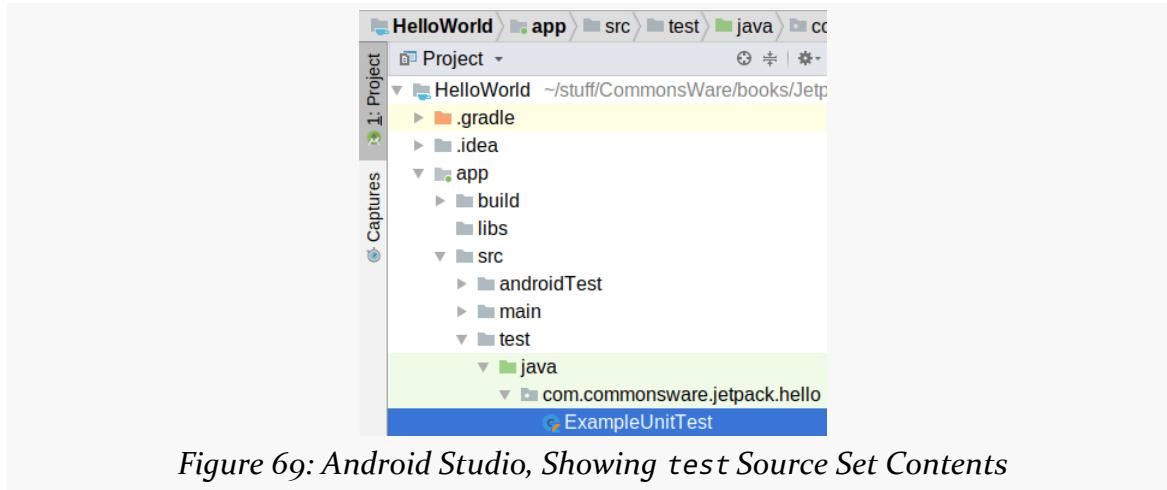


Figure 69: Android Studio, Showing test Source Set Contents

The Annotations and the Test Code

Once again, the class will either be in Java or Kotlin:

```
package com.commonware.jetpack.hello

import org.junit.Test
import org.junit.Assert.*

/**
 * Example local unit test, which will execute on the development machine (host).
 *
 * See [testing documentation](http://d.android.com/tools/testing).
 */
class ExampleUnitTest {
    @Test
    fun addition_isCorrect() {
        assertEquals(4, 2 + 2)
    }
}

package com.commonware.jetpack.hello;
import org.junit.Test;
import static org.junit.Assert.*;
```

TOURING THE TESTS

```
/**  
 * Example local unit test, which will execute on the development machine (host).  
 *  
 * @see <a href="http://d.android.com/tools/testing">Testing documentation</a>  
 */  
public class ExampleUnitTest {  
    @Test  
    public void addition_isCorrect() {  
        assertEquals(4, 2 + 2);  
    }  
}
```

Unit test classes do not normally get a @RunWith annotation. However, we still put @Test annotations on the methods or functions that contain the tests to be run.

In both Java and Kotlin, the tests use JUnit's assertEquals() to compare two values for equality. This time, the test is to confirm that $2 + 2$ equals 4. This test usually succeeds.

How You Run Them

You can run unit tests in much the same way as you run instrumented tests:

- Click the green “run” icon next to a test method/function to run it alone:

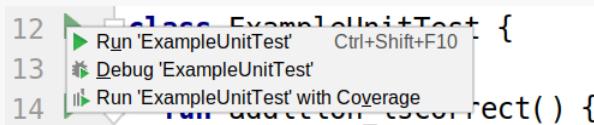


Figure 70: Android Studio, Showing Run Icons and Menu for Kotlin Unit Test Class

- Click the “run” icon next to a test class to run all of its test code
- Set up a “run configuration” to run all of your unit tests, which we will see [much later in the book](#)

What the Test Results Look Like

When you run unit tests, you will not be prompted for a device or emulator to run them on. Unit tests run directly on your development machine, not on an Android device or emulator.

TOURING THE TESTS

They will run much faster, and they will provide similar output in the “Run” view of Android Studio:

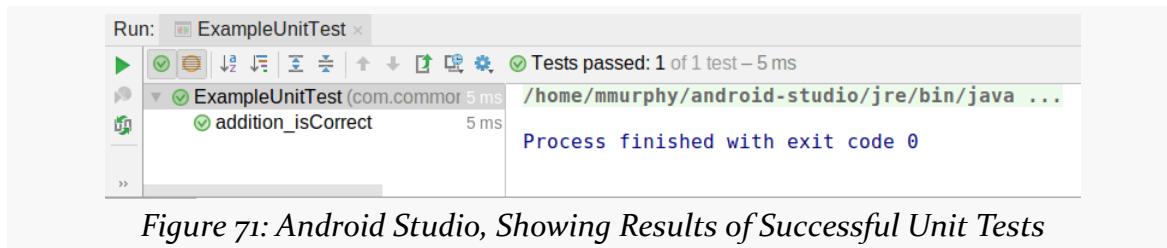


Figure 71: Android Studio, Showing Results of Successful Unit Tests

The testImplementation Dependencies

Any `testImplementation` statement in your dependencies in your module’s `build.gradle` file will be compiled into your tests and available for use. The one default `testImplementation` dependency is `junit:junit`, which pulls in the JUnit classes.

Introducing Jetpack

This book is titled “Elements of Android Jetpack”. Yet, we have not yet discussed Jetpack at all. So, it’s about time that we saw what Jetpack is and what it has to do with Android app development, your future projects, and the rest of this book.

What, Exactly, is Jetpack?

Jetpack... is a brand name.

In 2018, Google elected to use the term “Jetpack” to refer to a seemingly-random collection of Android technologies and tools. In many respects, the only things that those technologies and tools have in common is:

- They are involved in Android app development
- Google thinks that they are part of Jetpack

There is no singular “Jetpack” tool or library. In fact, outside of the cover and the contents of this chapter, “Jetpack” will not be used much as a term in this book, as it focuses on what is *in* Jetpack more so than Jetpack itself.

Um, OK, So, What’s the Point?

The first Android devices shipped to the public in October 2008. Public Android app development had been going on for about a year prior to this, with the first preview releases and Android SDK bits available in 2007. Google had been working on Android apps for a while prior even to that preview release.

In short, Android app development has been around for a while.

INTRODUCING JETPACK

Along the way, many techniques and approaches were tried and discarded. Many programming patterns were applied, rejected, and replaced. Many UI designs were rolled out to great fanfare, only to be shunted aside in favor of yet new UI designs. And so on. Android app development has a lot of history, and while some of that history remains relevant today, a lot of it is just baggage.

Jetpack is a way of thinking about Android app development that dumps some of that baggage.

The focus of Jetpack — and this book by extension — is on the current recommended practices in Android. We try to ignore most of the history, or at least put it in its appropriate place. Instead, we try to help you build using up-to-date approaches, ignoring the approaches of yesteryear.

Key Elements of Jetpack

[The definition of Jetpack](#) most likely will change over time. Not everything in the Jetpack is of equal importance. This book focuses on those elements that are commonplace for most ordinary Android apps or are tied closely to other commonplace Android app development steps.

Layout

By “Layout”, Google is referring to the overall widget-based UI setup. We saw in the previous chapters how Android gives us layout resources that we can work with, and that we can use Android Studio for editing them. Layout resources — and the widgets that they use — are essential for most Android apps.

We will begin seeing how to assemble user interfaces [shortly](#).

Fragment

A layout does not exist on its own. Something has to arrange to use a layout resource and show it on the screen. We saw in the previous chapters that an `Activity` can be used for that, and we will start there for seeing how to build Android user interfaces.

However, in many cases, you will wind up using fragments for loading and displaying your layout resources. Fragments are wrappers around layout resources and their widgets. Many apps might consist of a single activity, with each different “screen” being represented by a fragment that uses associated layout resources and

code. The user, as part of using the app, will switch from fragment to fragment as needed — for example, clicking a button might cause another fragment to be shown on the screen.

We will look at fragments [a bit later in the book](#).

AppCompat

As was mentioned earlier in this chapter, Android is over 10 years old. Android changes with every release.

However, Android devices do not get very many OS updates from their manufacturer. Even the best might only get updates for 2-3 years, and there are plenty that *never* get an update. As a result, Android users use a wide range of Android OS versions. While the changes from release to release may be small or may be large, the combined changes from older versions of Android to newer ones can be vast.

AppCompat tries to help. It gives us an API for our activities and fragments that resembles the latest-and-greatest version of Android. When your app runs on older devices, AppCompat tries to fill in the gaps of UI functionality where it can. While using AppCompat makes your APK a lot larger and makes app development more confusing, many developers are grateful for the backwards-compatibility that it offers.

We will dive more deeply into what AppCompat is, and its relationship to your app, [in an upcoming chapter](#).

Data Binding

Our Android user interfaces are going to have all sorts of widgets: fields, buttons, labels, checkboxes, list, and more. Some of them, such as labels, usually are static: we set their properties in a layout resource and do not need to change them. Others, though, will need to be filled in at runtime, such as populating a form with data that we obtain from a Web service or database. Similarly, we will need to respond to user input supplied to those widgets, such as reading in that form and updating the database when the user clicks “Save”.

We can do all of that work from our Java/Kotlin code. However, in some cases, it is more convenient to do “data binding”. This allows us to add smarts to our layout resources, using snippets of code that look a bit like Java/Kotlin. Our layout

INTRODUCING JETPACK

resources can contain simple rules for pulling data from our app, rather than us writing code to push data to the layout from our activities and fragments. Data binding is less popular than many of the other elements of Jetpack, but Google is putting increased emphasis on it going forward.

We will explore data binding [later in the book](#).

Navigation

While some apps might have just one screen of information, most have more than one. Users click on widgets and are taken to other screens. Users then press the BACK button and return to the screen they had been on originally. And so forth.

We have had ways of accomplishing this sort of navigation since Android's introduction. However, it involves Java/Kotlin code, and sometimes quite a bit of it, for complex interactions. Google keeps trying to move some of that sort of work into resources, where it can.

One of the newest technologies to be part of the Jetpack is the Navigation library. This provides a way to declare how to move from screen to screen via resources, with a new editor in Android Studio to assist in defining those resources and depicting your navigation flow.

We will explore this Navigation library as part of our discussion of [fragments](#).

Lifecycles, ViewModel, and LiveData

Most Android devices are phones or tablets. These can be held in portrait or landscape. Most apps appear to seamlessly transition between the two as the user rotates the screen.

In reality, screen rotations and other types of “configuration changes” are among the most annoying aspects of Android app development. It turns out that our activities and fragments have “lifecycles”, and they come and go not only based on user navigation but also these configuration changes. Keeping our state as the user rotates the screen is important, as users get irritated if we lose stuff.

To that end, Google nowadays offers two Jetpack pieces to help with this:

- Special code for dealing with lifecycles
- A `ViewModel` class that helps maintain our state across configuration changes

INTRODUCING JETPACK

For some developers, configuration changes are the most annoying thing in Android app development. For others, it is threads. We need to do work on background threads to keep our UI “responsive”, accepting and processing user input while that background work is ongoing. This is a pain to manage in its own right, and configuration changes make it that much more difficult. LiveData objects — held by our ViewModel — make it somewhat easier to do work in background threads while not losing track of that work if the user rotates the screen, for example.

In this book, we will explore [ViewModel and lifecycle handlers](#) as well as [LiveData and threads](#).

Permissions

Some things that we may want to do in our app require user permission. For example, we need permission to get at personal data of their contacts, or to send SMS messages on their behalf. Sometimes, the concern is privacy and security. Sometimes, the concern is cost, whether in terms of money or in terms of system resources, such as battery consumption.

Google has declared certain APIs to be protected by permissions. Before we can use those APIs, we need to ask permission from the user. This involves putting some entries in our manifest to say what we want in general. It also involves having some code that specifically requests the permission the first time that we have a concrete need for it.

We will be seeing permissions in action [later in the book](#).

Room

Many apps need a database on the device. Sometimes, that is the “system of record”, as the data is only on the device. Sometimes, the database serves as more of a local cache, for data that we obtained from a server or for data that we need to get to a server.

Android has had a relational database, called SQLite, in its SDK since the beginning. However, the standard Android APIs for working with this database are a bit crude by modern standards. To that end, Google has created Room, a lightweight object wrapper around SQLite databases, to simplify your database I/O. Room also works nicely with LiveData, to help you do your database I/O on background threads.

INTRODUCING JETPACK

We will explore the basics of Room [much later in the book](#).

WorkManager

Many times, the work that we need to do on background threads has to be performed in near-real time. But sometimes the work that we need to do can happen totally independently from what the user may (or may not) be doing in our UI. So while a “pull to refresh” UI operation requires us to do a refresh right now, a periodic refresh from the server might happen every few hours, even if the user is not in the app right now.

While there have been many solutions over the years for this problem, Google is steering us towards WorkManager right now. We can teach WorkManager about background work to be performed, then schedule that work to occur. That schedule might be based in part on time (“do it soon”, “do it every few hours”, etc.). That schedule might be based in part on the state of the device (“do it when the device is on a charger”, “do it when the device is on WiFi”, etc.). Then, WorkManager will arrange to do the work, even if the user leaves our UI.

We will examine WorkManager [towards the end of the book](#).

Android KTX

The Android SDK is based on Java, and it has a fairly typical Java API, albeit one with a variety of patterns, based in part on the age of the code.

Kotlin is Google’s recommended language for Android developers going forward, with Java being relegated to the SDK itself. This is similar to how Apple steers developers towards Swift, with Objective-C being used only sporadically.

One of Kotlin’s features is the “extension function”. This is a way for a library to add functions (Java methods) to somebody else’s Java/Kotlin class. For developers using the class, the extension functions are seamlessly integrated with the functions that were part of the class from the outset.

Android KTX is a collection of such extension functions, designed to make common aspects of the Android SDK a bit easier to use.

We will see Android KTX code sprinkled throughout this book.



You can learn more about extension functions in the "Extension Functions" chapter of [Elements of Kotlin!](#)

Testing

No software project is complete without testing. We have already seen how Android projects can have instrumented tests and unit tests. We are given stub tests when we create our project, just to get us going, but we will need to write a lot more tests before we are done.

We will be exploring testing more [later in the book](#).

What Came Before: the Android Support Library

Many of the capabilities described here as being part of Jetpack have existed for a while, in some cases for many years.

The biggest change with Jetpack — besides the branding — is the set of libraries that we use. The Jetpack libraries are all “AndroidX” libraries, where we will be referring to libraries and Java packages that have androidx in their names.

Prior to Jetpack, we had a set of libraries that evolved over the years, with a variety of naming schemes for libraries and Java packages. Much of what is in the Jetpack originated in what we called the Android Support Library. Some came from the Architecture Components, which was another family of libraries.

Since Google debuted Jetpack in May 2018, anything written prior to that would be referring to these older library collections. Most of what was in those libraries has been replaced by AndroidX equivalents, though not everything. This will cause some amount of confusion, particularly since a single Android project cannot easily combine the older libraries (Support Library and Architecture Components) and the AndroidX libraries.

As a result, particularly as you learn Android app development, it will help if you know the dates for your educational materials and online resources. Whenever you see a date prior to May 2018 — and even for some stuff after that date — just bear in mind that you may need to convert some older library and Java package names into their AndroidX equivalents.

Licensed solely for use by Patrocinio Rodriguez

Introducing the Sampler Projects

It is time to move past what Android Studio gives you by default for a new project and start seeing how we can add our own stuff to these projects.

To that end, this book has a pair of Android Studio projects that contain the code samples that will be shown in the remainder of the book. You are welcome to download one or both of these projects to be able to run the samples on your own devices or emulators.

The Projects

The two projects are very similar, in terms of the Android SDK features that they demonstrate. The primary difference: one is in Kotlin, and the other is in Java. The resources, manifests, and much of the Gradle build files will be the same, but the source code will be in the programming language for the project:

- Sampler is in Kotlin
- SamplerJ is in Java

This way, you can choose which project to work with, based on which programming language you want to use for your initial Android work. In general, the rest of the book will show you both the Java and the Kotlin editions of the code, so you can compare and contrast the two languages and Android's support for each. In particular, if you are new to Kotlin, these corresponding samples can help you see how Java approaches get translated into Kotlin. Note that there will be a few samples in the latter half of the book that are Kotlin-only.

These two projects are hosted on [GitLab](#):

INTRODUCING THE SAMPLER PROJECTS

- The Java project is at: <https://gitlab.com/commonsguy/cw-jetpack-java>
- The Kotlin project is at: <https://gitlab.com/commonsguy/cw-jetpack-kotlin>

Getting a Sampler Project

Relevant portions of the Sampler projects are shown directly in this book, including links to the GitLab Web site where you can view the full source files. So, if all you want to do is to read the code, you do not need anything more than this book and a Web browser.

If, on the other hand, you want to run the sample code, or perhaps modify it as part of your own experiments, you will need to get one or both of the Sampler projects onto your development machine and into Android Studio. You can do that completely from inside Android Studio if you wish, or you can get the source code separately and then import it into Android Studio.

Direct From Android Studio

Android Studio has built-in support for Git. You can download and import a Git project into Android Studio via File > New > Project from Version Control > Git from an existing Android Studio window.

If you are at the Android Studio “welcome” dialog, click “Check out project from Version Control” and choose Git from the menu. If you already have a project open in Android Studio, choose File > New > Project from Version Control > Git from the main menu.

This will bring up a “Clone Repository” dialog box:



Figure 72: Android Studio Clone Repository Dialog

Here, “clone” is a term used in Git to refer to the act of both downloading the source code and setting up a connection for you to be able to get updates later on.

INTRODUCING THE SAMPLER PROJECTS

In the “URL” field, fill in the Git URL for the project that you want to clone:

- Java: <https://gitlab.com/commonsguy/cw-jetpack-java.git>
- Kotlin: <https://gitlab.com/commonsguy/cw-jetpack-kotlin.git>

You can click the “Test” button to confirm that you are able to connect to that URL.

In the “Directory” field, you can provide the path to the directory on your development machine where you want to put the source code. This needs to be a new empty directory, but otherwise it can be wherever you want. The “...” button next to that field will bring up a directory chooser window to help you pick or create the directory that you want to use.

Then, click the “Clone” button to download the code and begin importing it into Android Studio. After a short while, your project will then be opened into an Android Studio window and be ready for use.

Manually

If you prefer to use other tools to work with Git repositories, you are welcome to do that too.

Clone or Download

Using your favorite Git client, clone the repository from its associated URL:

- Java: <https://gitlab.com/commonsguy/cw-jetpack-java.git>
- Kotlin: <https://gitlab.com/commonsguy/cw-jetpack-kotlin.git>

Import

Then, in an existing Android Studio window, you can choose File > New > Import Project to bring up a file/directory chooser. Alternatively, if you are at the welcome dialog, choose “Import an existing project” to bring up that same chooser.

Select the directory into which you cloned or unZIPped the Git repo, then click OK. It should open up right into an Android Studio window.

The Modules

Each project has the same set of modules:

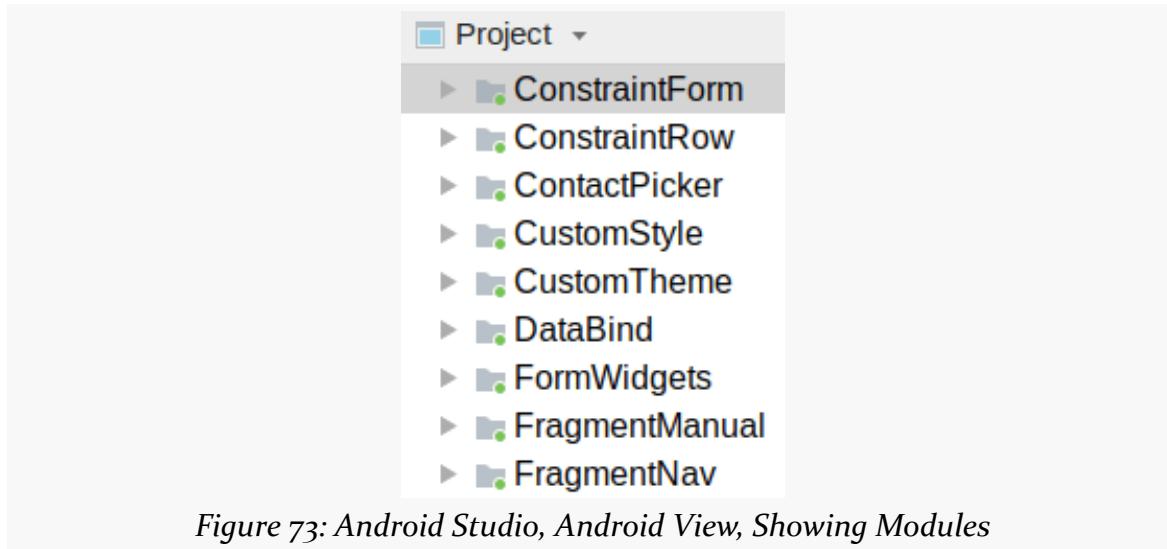


Figure 73: Android Studio, Android View, Showing Modules

Each module represents one sample app. Some chapters will use a few sample apps. Some sample apps will be used across multiple chapters. Each chapter will point out the modules that it is showing you, so you can synchronize your IDE with what the book is covering.

Running the Samples

Each module is ready to run!

To the left of the green triangle “Run” toolbar button is a drop-down list:



Figure 74: Android Studio, Run Configurations Drop-Down

This lists the “run configurations”, which basically means “the things that Android Studio knows how to run”. By default, you will get one run configuration for each app module, with the same name as the module. Selecting that run configuration and clicking the “Run” toolbar button will run that app on your chosen device or emulator.

Starting Simple: TextView and Button

Android offers a wide assortment of UI elements. Some are part of the Android SDK. Others come from third-party libraries. If the UI element is something commonplace, seen in multiple platforms or GUI environments, then it is likely that Android has an analogue of it somewhere.

But, as the saying goes, we must learn to walk before we can run.

So, in this chapter, we will explore the basics of Android's widget-based UI system by looking at two extremely common widgets: `TextView` and `Button`.

First, Some Terminology

Let's start off by defining some terms as they will be used in Android app development — and, by extension, in this book.

Widgets

Wikipedia has [a nice definition of a widget](#):

A control element (sometimes called a control or widget) in a graphical user interface is an element of interaction, such as a button or a scroll bar. Controls are software components that a computer user interacts with through direct manipulation to read or edit information about an application.

(quote from the 29 November 2018 version of the page)

STARTING SIMPLE: TEXTVIEW AND BUTTON

Take, for example, this Android screen:

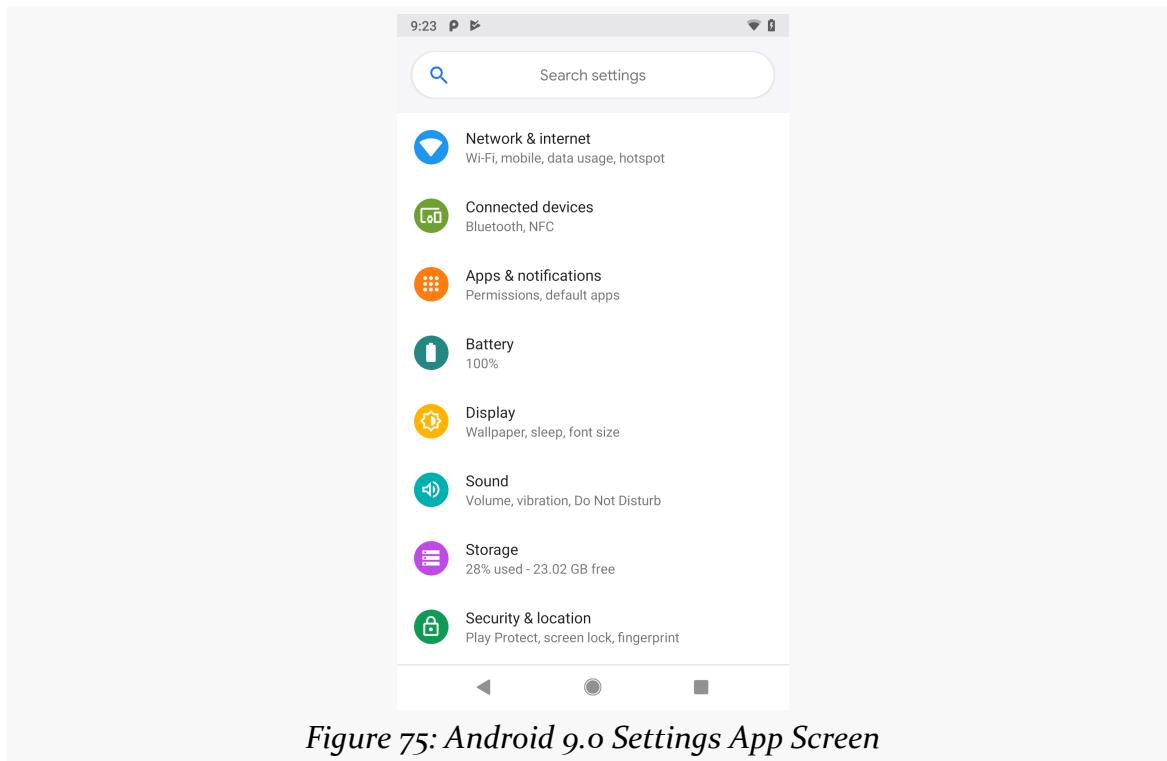


Figure 75: Android 9.0 Settings App Screen

Here, we see:

- a vertically-scrolling list, with rows containing two pieces of text and an icon
- a search field, with a magnifying glass icon and a space to type in a search term, contained in a rounded rectangle

Everything listed above is made of widgets. The user interface for most Android screens is made up of one or more widgets.

This does not mean that you cannot do your own drawing. In fact, all the existing widgets are implemented via low-level drawing routines, which you can use for everything from your own custom widgets to games.

However, for most non-game applications, your Android user interface will be made up of several widgets.

From a coding standpoint, widgets extend from a base class named `android.view.View`.

Containers

Containers are ways of organizing multiple widgets into some sort of structure. Widgets do not naturally line themselves up in some specific pattern — we have to define that pattern ourselves.

In most GUI toolkits, a container is deemed to have a set of children. Those children are widgets, or sometimes are other containers. Each container has its basic rule for how it lays out its children on the screen, possibly customized by requests from the children themselves.

Android supplies a handful of containers, designed to handle most common scenarios. However, we will use one most of the time: `ConstraintLayout`. This particular container gives us a flexible set of rules to decide where to place widgets within the screen. We will explore `ConstraintLayout` in [an upcoming chapter](#).

From a coding standpoint, containers extend from a base class called `android.view.ViewGroup`. `ViewGroup` itself inherits from `View`, and so anything that we can do to any widget we can also do to any container.

Attributes

Widgets have attributes that describe how they should look and behave. In a layout resource, these are literally XML attributes on the widget's element in the file. Usually, there are corresponding getter and setter methods for manipulating this attribute at runtime from your Java/Kotlin code.

For example, widgets and containers have a “visibility” attribute. This is set by:

- `android:visibility` attribute in a layout resource
- `setVisibility()` in Java
- `visibility =` in Kotlin

If you visit the JavaDocs for a widget, such as [the JavaDocs for `TextView`](#), you will see an “XML Attributes” table near the top. This lists all of the attributes defined uniquely on this class, and the “Inherited XML Attributes” table that follows lists all those that the widget inherits from superclasses. Of course, the JavaDocs also list the fields, constants, constructors, and public/protected methods that you can use on the widget itself.

This book does not attempt to explain each and every attribute on each and every

STARTING SIMPLE: TEXTVIEW AND BUTTON

widget. We will, however, cover a variety of popular widgets and the most commonly-used attributes on those widgets.

Widget IDs

Most widgets and containers in a layout resource will have an ID associated with them, by means of an `android:id` attribute. This serves two roles:

1. It allows you to reference that widget from within the layout file, for relative positioning rules (e.g., put this Button below that other Button)
2. It allows you to reference that widget from Java/Kotlin code

For example, here is a `TextView` widget with an ID of `@+id/hello`:

```
<?xml version="1.0" encoding="utf-8"?>
<TextView android:id="@+id/hello"
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:padding="@dimen/hello_padding"
    android:text="@string/hello"
    android:textSize="@dimen/hello_size"
    android:textStyle="bold" />
```

(from [SimpleText/src/main/res/layout/activity_main.xml](#))

Usually we use `@+id/...` as the `id` value, where the `...` represents your locally-unique name for the widget. For example, you might have a widget whose ID is `@+id/postalCode` or a container whose ID is `@+id/addressInfo`. You may find `android:id` values that lack the `+` sign (e.g., `@id/postalCode`) — this just means that somewhere else in the same layout resource, there will be one for the same ID value, but with the `+` sign.

Size, Margins, and Padding

Widgets have some sort of size, since a zero-pixel-high, zero-pixel-wide widget is not especially user-friendly. Sometimes, that size will be dictated by what is inside the widget itself, such as a label (`TextView`) having a size dictated by the text in the label. Sometimes, that size will be dictated by the size of whatever holds the widget (a “container”, described in the next section), where the widget wants to take up all remaining width and/or height. Sometimes, that size will be a specific set of dimensions. We will see attributes like `android:layout_height` and

STARTING SIMPLE: TEXTVIEW AND BUTTON

android:layout_width that can be used by a widget to suggest sizing rules to its parent.

Let's look again at that TextView:

```
<?xml version="1.0" encoding="utf-8"?>
<TextView android:id="@+id/hello"
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:padding="@dimen/hello_padding"
    android:text="@string/hello"
    android:textSize="@dimen/hello_size"
    android:textStyle="bold" />
```

(from [SimpleText/src/main/res/layout/activity_main.xml](#))

Here, we have both android:layout_height and android:layout_width set to wrap_content. wrap_content tells Android “please size this based on whatever is inside of it”. For example, with a label (TextView), “whatever is inside of it” is the text that should be displayed. For a container (e.g., ConstraintLayout), “whatever is inside of it” is the set of children that it manages.

Widgets can have margins. As with CSS, margins provide separation between a widget and anything adjacent to it (e.g., other widgets, edges of the screen). Margins are designed to help prevent widgets from running right up next to each other, so they are visually distinct. We will see attributes like android:layout_margin for specifying margins.

Widgets can have padding. As with CSS, padding provides separation between the contents of a widget and the widget’s edges. This is mostly used with widgets that have some sort of background, like a button, so that the contents of the widget (e.g., button caption) does not run right into the edges of the button, once again for visual distinction. We will see attributes like android:padding for specifying padding.

Hey, What Are Those @dimen Things?

You will notice that the android:padding attribute has a value of @dimen/hello_padding. That is a reference to a dimension resource. Like string and color resources, dimension resources reside in files in res/values/, typically named dimens.xml. There, you give the dimension a name and a corresponding value:

```
<?xml version="1.0" encoding="utf-8"?>
```

STARTING SIMPLE: TEXTVIEW AND BUTTON

```
<resources>
    <dimen name="hello_size">24sp</dimen>
    <dimen name="hello_padding">8dp</dimen>
</resources>
```

(from [SimpleText/src/main/res/values/dimens.xml](#))

Here, we have two dimension resources, `hello_size` and `hello_padding`.

The value of a dimension resource is made up of two pieces:

- An integer or floating-point number, and
- A unit of measure

`hello_padding` is defined as `8dp`. `dp` is short for “density-independent pixels”. There are 160 `dp` in an inch. `dp` is the typical unit of measure for most things in Android.

However, `hello_size` is `24sp`. `sp` is short for “scaled pixels”. One `sp` is the same size as one `dp` for the default font scale. However, users can go into the Settings app and change their font scale, to make text bigger or smaller. `sp` takes that into account. So, if you use `sp` dimensions for your text size — say, in a `TextView` widget — your text will also adjust based on the user’s chosen font scale.

There are other units of measure that you could use but are almost never used:

- `in` for inches
- `mm` for millimeters
- `px` for hardware pixels

In particular, trying to use `px` for a dimension will result in compiler warnings. Using `px` does not take screen density differences into account, so a dimension measured in `px` will be different sizes on different devices. This is almost never what you want.

Introducing the Graphical Layout Editor

If you open a layout resource in Android Studio, by default you will see the graphical layout editor:

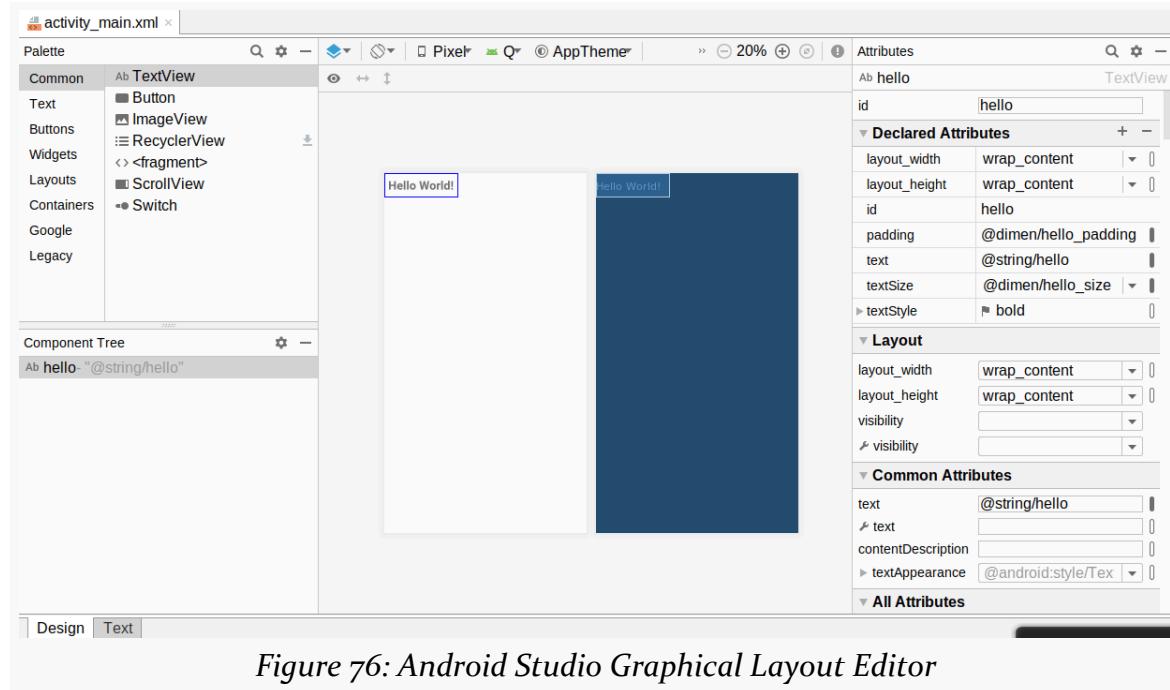


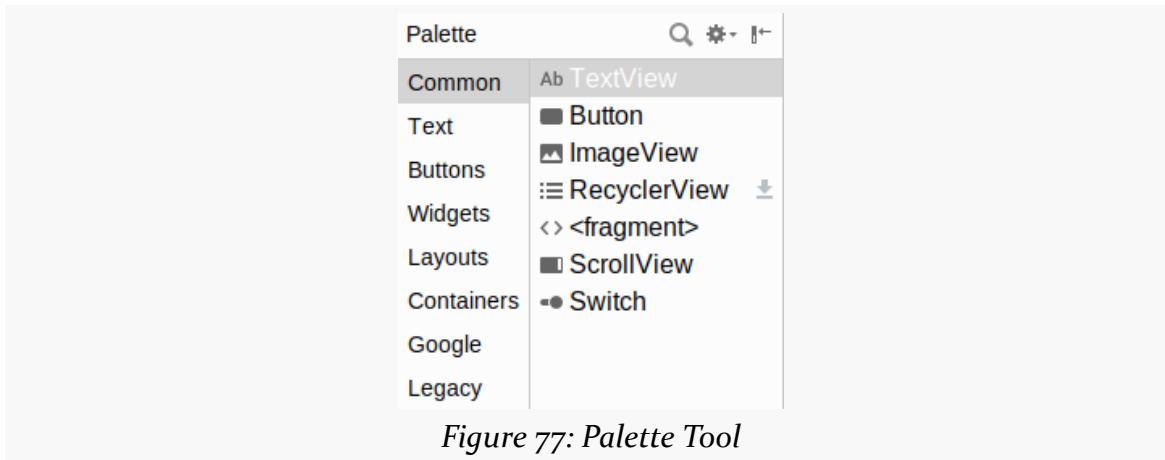
Figure 76: Android Studio Graphical Layout Editor

This offers a drag-and-drop means of defining the contents of that layout resource.

STARTING SIMPLE: TEXTVIEW AND BUTTON

Palette

The upper-left side of the graphical layout editor is the Palette tool:



This lists all sorts of widgets and containers that you can drag and drop. They are divided into categories (“Widgets”, “Text”, “Layouts”, etc.) with many options in each. A few are not strictly widgets or containers but rather other sorts of XML elements that you can have in a layout resource (e.g., `<fragment>`, `<requestFocus>`). Some — such as the RecyclerView shown in the above screenshot — are from libraries and may have a “download” icon adjacent to them to help illustrate that.

As we cover how to use the graphical layout editor, we will see how to create and configure several of these widgets, containers, and other items.

STARTING SIMPLE: TEXTVIEW AND BUTTON

Preview

The main central area of the graphical layout editor consists of two perspectives on your layout resource contents. The one on the left is a preview of what your UI should resemble, if this layout were used for the UI of an activity:



Figure 78: UI Preview

This pours your layout resource contents into a preview frame that has aspects of a regular Android device, such as the navigation bar at the bottom and the status bar at the top.

If you drag items out of the Palette and drop them into the preview area, they will be added to your layout resource.

Blueprint

To the right of the preview area is the blueprint view. This also visually depicts your layout resource. However, rather than showing you a preview of what your UI might look like, it visually represents what widget and container classes you are using. And, for some types of containers, it will show some of the sizing and positioning rules that you are using for children of that container:

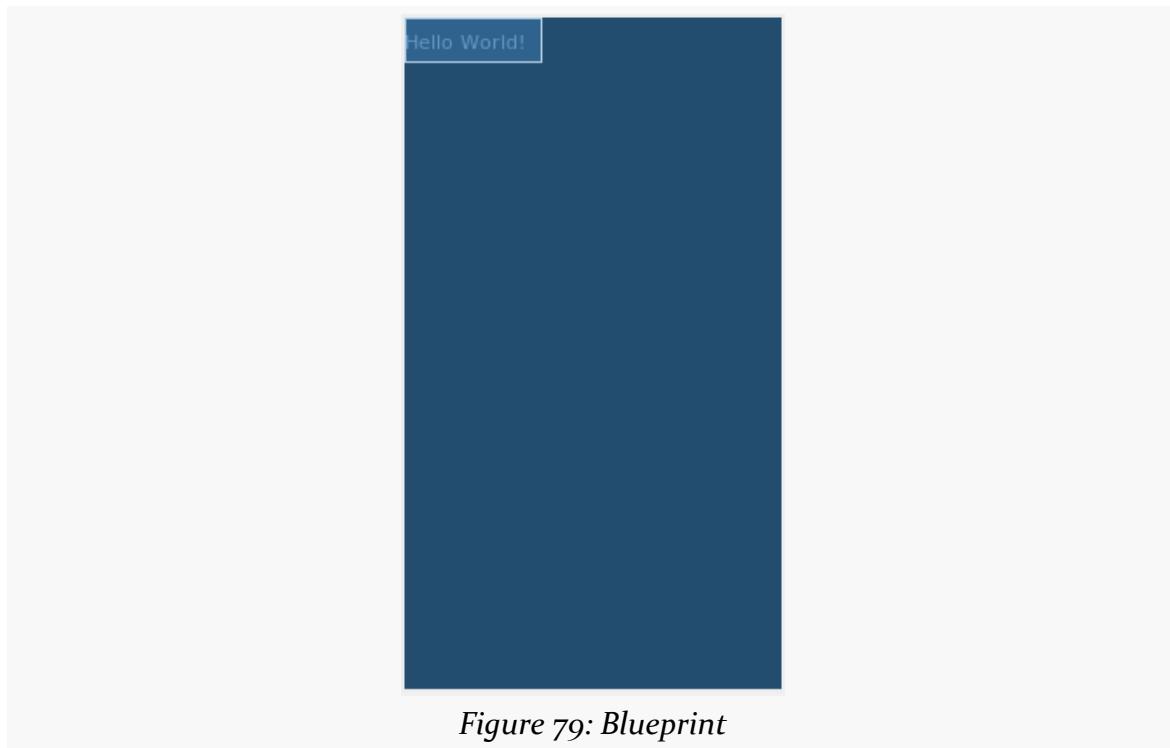


Figure 79: Blueprint

For a trivial layout resource, the blueprint view does not show you much. It will become more useful with more complex layout resources.

Preview Toolbar

Above the preview and blueprint is a bi-level toolbar that allows you to configure various aspects of the preview and blueprint appearance and behavior.

Upper Left Toolbar



Figure 8o: Preview Toolbar, Top Level

From left to right, the upper left portion of the toolbar contains:

- A drop-down to toggle whether you see the preview, the blueprint, or both
- A toggle to control whether you are seeing the layout as applied to portrait or landscape perspectives
- A drop-down to choose what device size and resolution should be used for the preview, culled from your emulator images and the available device definitions
- A drop-down to choose what API level should be used for the simulated UI of the preview
- A button to choose what [theme](#) to use for presenting the UI of the preview
- A button to choose what language to use for determining which of your string resources gets used in the preview

A couple of those — particularly the theme selector — pertain to topics that we will explore later in the book.

Upper Right Toolbar



Figure 8i: Preview Toolbar, Bottom Level, Right Side

On the upper right side of the toolbar are:

- Zoom controls
- A button to reset the zoom to fill the area available for the preview and/or blueprint
- An “info” icon indicating if there are any warnings or errors associated with your layout resource

Component Tree

Towards the bottom-left corner is the component tree:

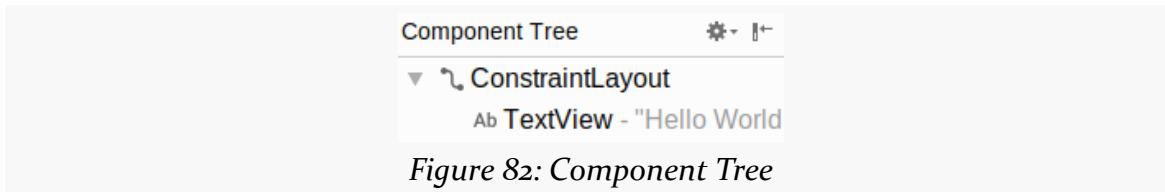


Figure 82: Component Tree

This gives you a full tree of all of the widgets and containers inside of this layout resource. It corresponds to the tree of XML elements in the layout resource itself.

Clicking on any item in the component tree highlights it in both the preview and blueprint views, plus it switches to that widget or container for the attributes pane.

Attributes

When a widget or container is selected — whether via the component tree, clicking on it in the preview, or clicking on it in the blueprint — the Attributes pane on the right will allow you to manipulate how that widget or container looks and behaves.

STARTING SIMPLE: TEXTVIEW AND BUTTON

By default, it will bring up a condensed roster of the most important attributes:

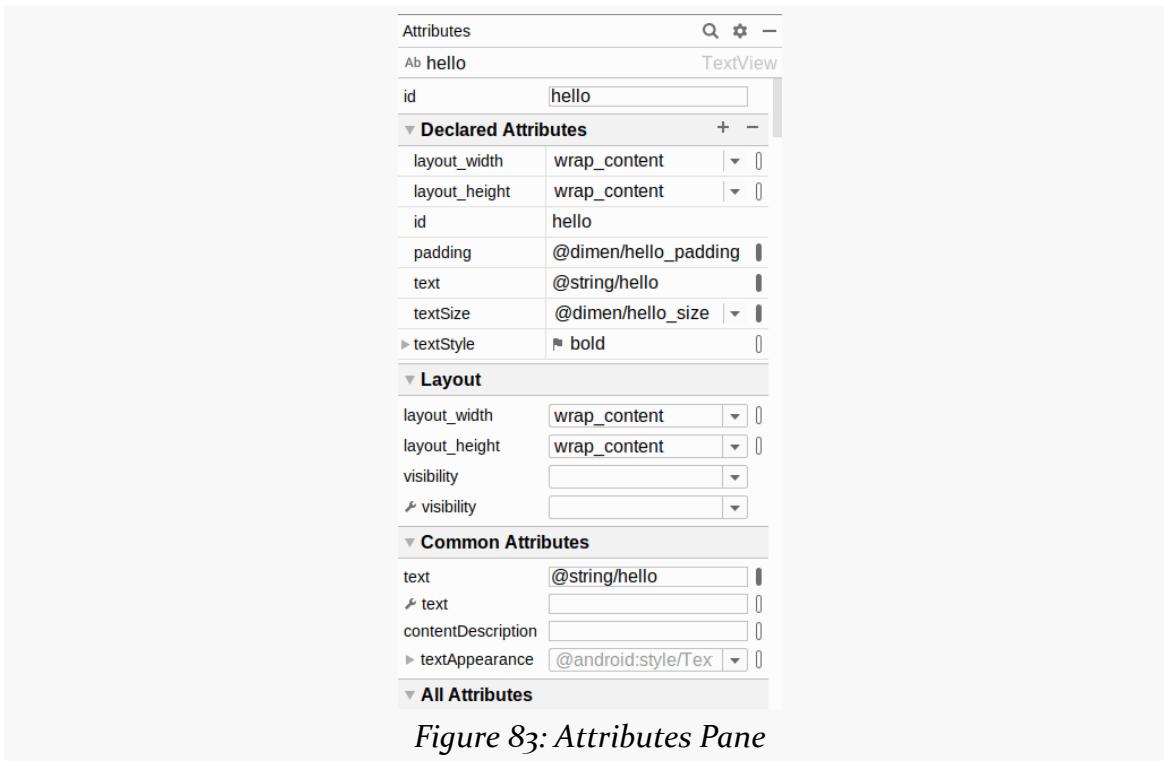


Figure 83: Attributes Pane

This is divided into sections. The “All Attributes” section, as the name suggests, lists all available attributes for this widget. The other sections highlight common subsets of the attributes. Each section can be expanded or collapsed via the triangle icon in the section header.

You can also click the magnifying glass icon in the toolbar of this pane to search for available attributes by name:



Figure 84: Attributes Pane, Showing Search Results

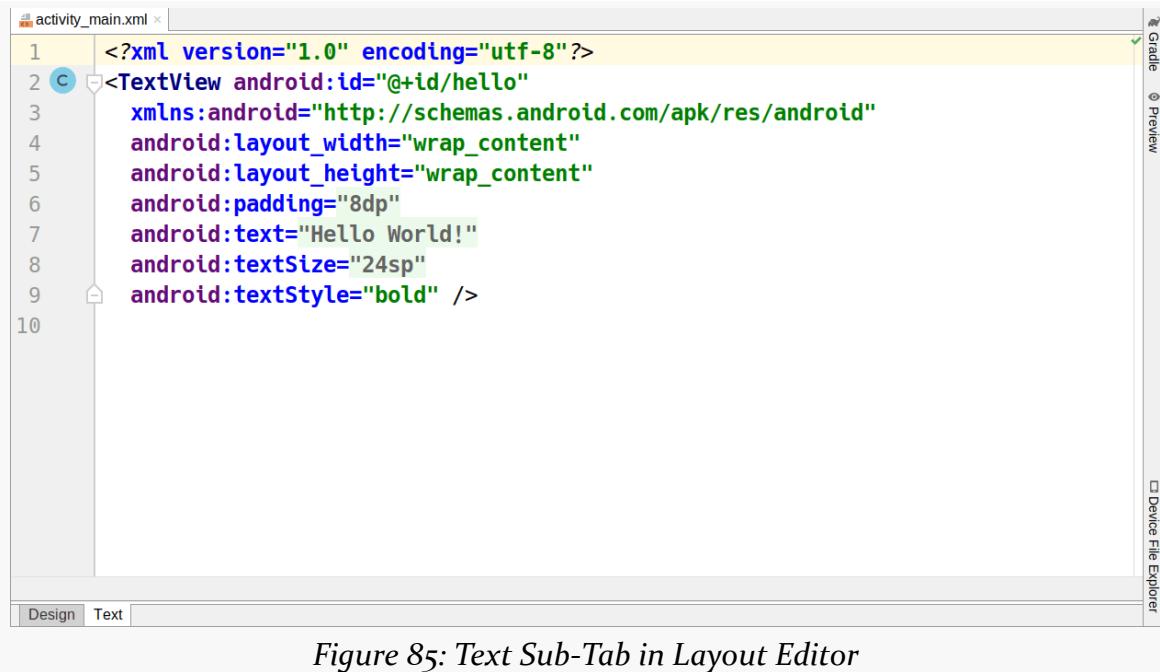
We will see what some of these attributes are and how to work with them over the course of the next few chapters.

STARTING SIMPLE: TEXTVIEW AND BUTTON

For the attributes in the full roster, you can click the star icon on the left to mark them as “favorites”, as seen with the “visibility” attribute in the above screenshot. Those favorite attributes show up in the condensed roster, in the section labeled “Favorite Attributes”.

Text Tab

Towards the bottom of the graphical layout tool, you will see that it contains two sub-tabs. One, “Design”, encompasses everything described above. The other, “Text”, allows you to edit the raw XML that is the actual content of the layout resource:



```
activity_main.xml <?xml version="1.0" encoding="utf-8"?>
<TextView android:id="@+id/hello"
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:padding="8dp"
    android:text="Hello World!"
    android:textSize="24sp"
    android:textStyle="bold" />
```

The screenshot shows the Android Layout Editor interface. At the top, there's a tab bar with 'activity_main.xml' and other tabs like 'Preview' and 'Device File Explorer'. Below the tabs is a toolbar with icons for 'Design', 'Text', 'Preview', and 'Device File Explorer'. The main area is a code editor showing the XML for a TextView. The XML code is as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<TextView android:id="@+id/hello"
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:padding="8dp"
    android:text="Hello World!"
    android:textSize="24sp"
    android:textStyle="bold" />
```

The 'Text' tab is currently selected, indicated by a blue border around its tab. The code is color-coded: blue for tags, green for attributes, and black for values. A small circular icon with a 'C' is visible next to the first line of code.

Figure 85: Text Sub-Tab in Layout Editor

STARTING SIMPLE: TEXTVIEW AND BUTTON

By default, the entire area is devoted to the text editor. However, when the Text sub-tab is active, a “Preview” tool will appear docked on the right side of the Android Studio window. Clicking that will display the preview from the Design sub-tab:

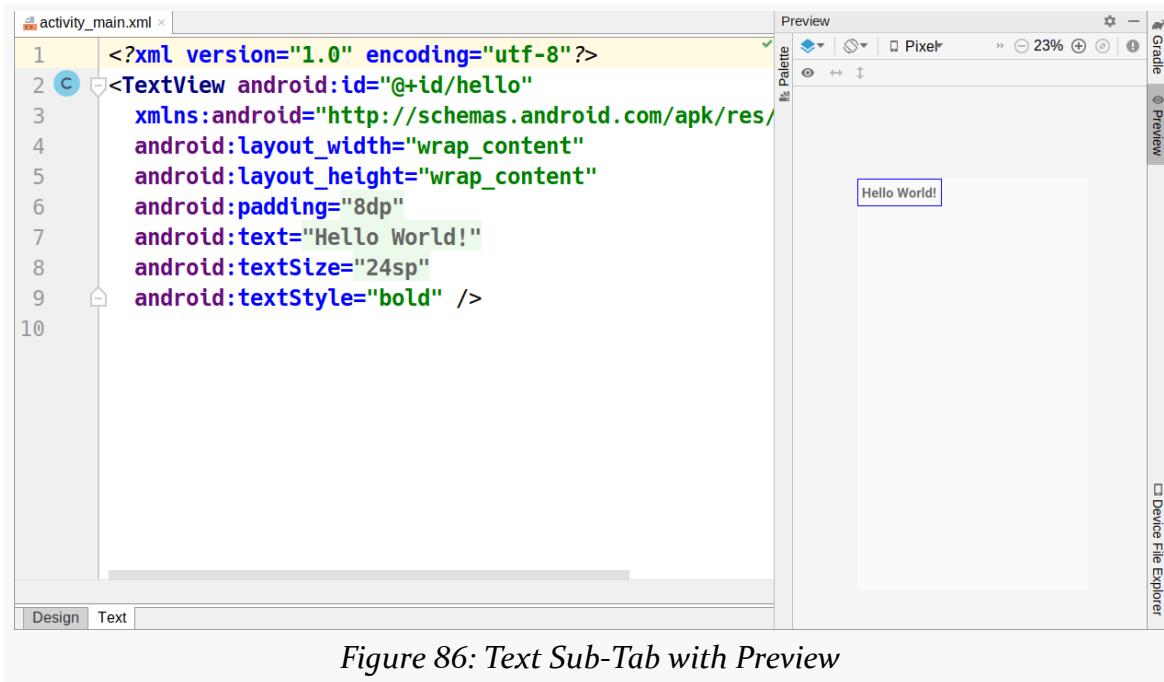


Figure 86: Text Sub-Tab with Preview

Clicking on items in the preview will highlight the corresponding XML element in the text editor.

TextView: Assigning Labels

The simplest widget is the label, referred to in Android as a `TextView`. Like in most GUI toolkits, labels are bits of text not editable directly by users. Typically, they are used to identify adjacent widgets (e.g., a “Name:” label before a field where one fills in a name) or display other text of relevance to users (e.g., messages in pop-up dialog).

As with any widget, you can create instances of `TextView` in your Java or Kotlin code by invoking a constructor, then use `setText()` methods to set the text to be displayed by the `TextView`. However, for ordinary UIs, typically you will use XML layout resources — in there, you can add a `TextView` element to the layout, with an `android:text` attribute to set the text of the label itself.

A Sample TextView

Our first sample app — SimpleText — is even simpler than the “Hello, World” ones from earlier in the book. The `activity_main` layout just has a `TextView` in it:

```
<?xml version="1.0" encoding="utf-8"?>
<TextView android:id="@+id/hello"
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:padding="@dimen/hello_padding"
    android:text="@string/hello"
    android:textSize="@dimen/hello_size"
    android:textStyle="bold" />
```

(from [SimpleText/src/main/res/layout/activity_main.xml](#))

A `TextView` displays some text, set by the `android:text` attribute. Because of the `wrap_content` values for width and height, the size of this `TextView` will be determined by:

- The text we are putting in it (`android:text`)
- The font size of that text (`android:textSize`)
- The font style of that text (`android:textStyle`)
- Any padding that we put on it (`android:padding`)

The [SamplerJ/SimpleText](#) edition of `MainActivity` just displays this layout, via `setContentView()`:

```
package com.commonsware.jetpack.samplej.simpletext;

import androidx.appcompat.app.AppCompatActivity;
import android.os.Bundle;

public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
}
```

(from [SimpleText/src/main/java/com/commonsware/jetpack/samplej/simpletext/MainActivity.java](#))

...while the [Sampler/SimpleText](#) edition does the same thing, but in Kotlin instead

STARTING SIMPLE: TEXTVIEW AND BUTTON

of Java:

```
package com.commonsware.jetpack.sampler.simpletext

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle

class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
    }
}
```

(from [SimpleText/src/main/java/com/commonsware/jetpack/sampler/simpletext/MainActivity.kt](#))

Android Studio Graphical Layout Editor

The TextView widget is available in the “Common” category of the Palette in the Android Studio graphical layout editor:

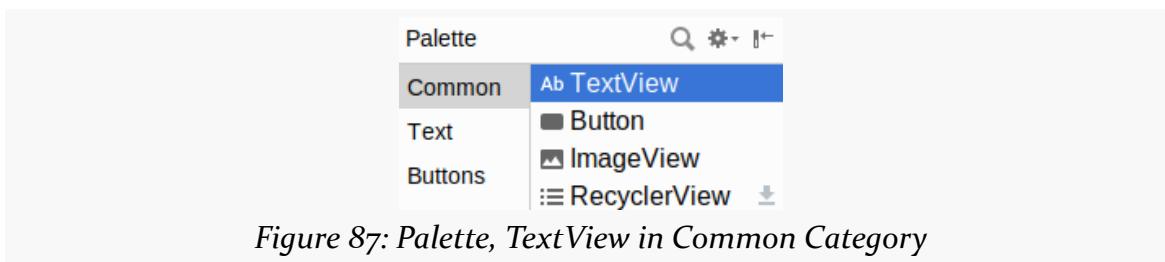


Figure 87: Palette, TextView in Common Category

(it also appears in the “Text” category)

If you want to add a TextView to a layout, just drag the TextView from the Palette into a layout file in the main editing area to add the widget to the layout. Or, drag it over the top of some container you see in the Component Tree pane of the editor to add it as a child of that specific container.

Clicking on the new TextView will set up the Attributes pane with the various attributes of the widget, ready for you to change as needed.

STARTING SIMPLE: TEXTVIEW AND BUTTON

Editing the Text

The “Text” attribute will allow you to choose or define a string resource to serve as the text to be displayed:

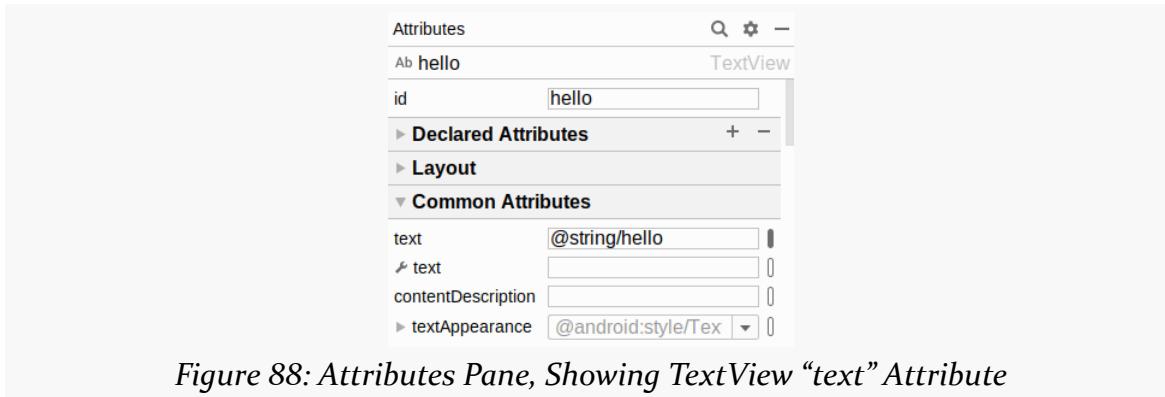


Figure 88: Attributes Pane, Showing TextView “text” Attribute

The “text” with a wrench icon allows you to provide a separate piece of text that will show up in the preview, but not be used by your app at runtime.

STARTING SIMPLE: TEXTVIEW AND BUTTON

You can either type a literal string right in the Attributes pane row, or you can click the “...” button to the right of the field to pick a string resource:

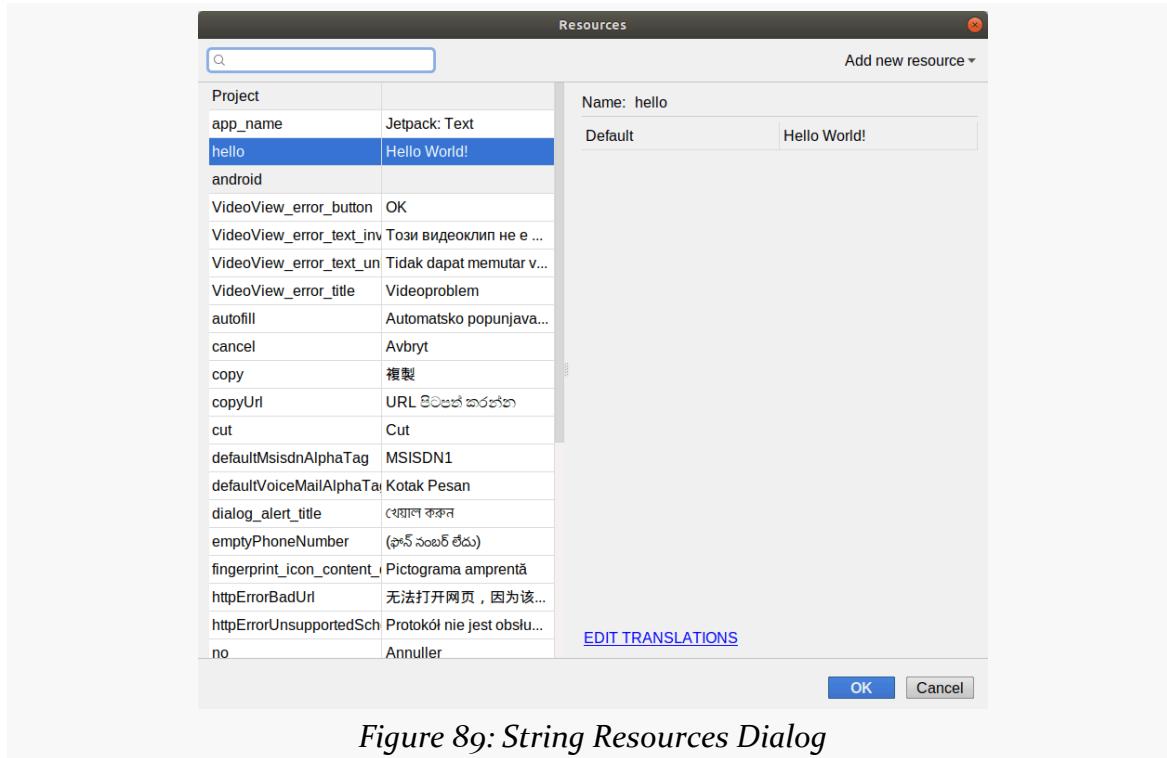


Figure 89: String Resources Dialog

STARTING SIMPLE: TEXTVIEW AND BUTTON

You can highlight one of those resources and click “OK” to use it. Or, towards the upper-right of that dialog, there is an “Add new resource” drop-down. When viewing string resources, that drop-down will contain a single command: “New string Value...”. Choosing it will allow you to define a new string resource via another dialog:

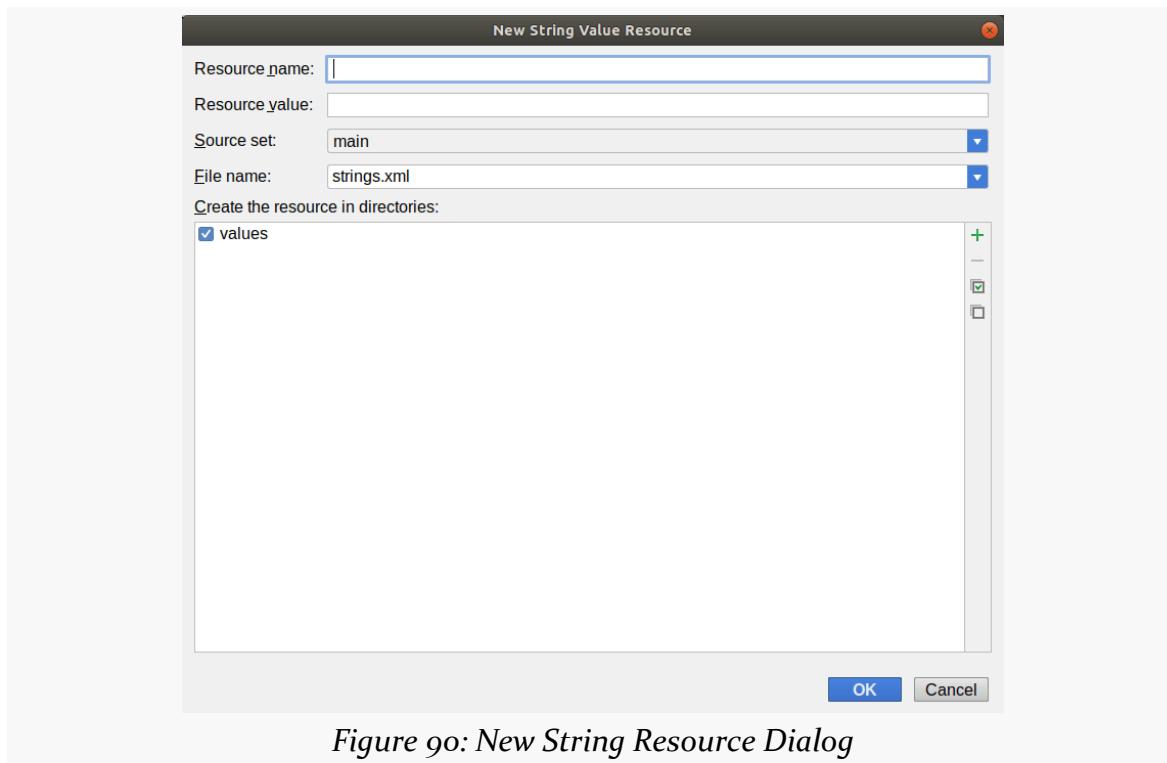


Figure 90: New String Resource Dialog

You can give your new string resource a name, the actual text of the string itself, the filename in which the string resource should reside (strings.xml by default), and which values/ directory the string should go into (values by default). You will also choose the “source set” — for now, that will just be main. Once you accept the dialog, your new string resource will be applied to your TextView.

In the SimpleText projects, the TextView has an android:text attribute set to the @string/hello string resource.

Editing the ID

The “id” attribute will allow you to change the android:id value of the widget:



Figure 91: Attributes Pane, Showing ID Field

The value you fill in here is what goes after the @+id/ portion (e.g., textView2). This works for all widgets, not just TextView.

In the SimpleText projects, the TextView has hello for its ID field contents, which results in android:id="@+id/hello" in the XML.

Notable TextView Attributes

TextView has numerous other attributes of relevance for labels, such as:

1. android:typeface to set the typeface to use for the label (e.g., monospace)
2. android:textStyle to indicate that the typeface should be made bold (bold), italic (italic), or bold and italic (bold_italic)
3. android:textColor to set the color of the label’s text, in RGB hex format (e.g., #FF0000 for red), ARGB hex format (e.g., #88FF0000 for a translucent red), or as a reference to a color resource (e.g., @color/colorAccent)

These attributes, like most others, can be modified through the Attributes pane, though many of these are in the “All Attributes” section. The SimpleText app sets the textSize, textStyle, and padding attributes.

Button: Reacting to Input

Android has a Button widget, which is your classic push-button “click me and something cool will happen” widget. As it turns out, Button is a subclass of TextView, so everything discussed in the preceding section in terms of formatting the face of the button still holds.

A Sample Button

Our next sample app — SimpleButton — has a similar activity_main layout as the

STARTING SIMPLE: TEXTVIEW AND BUTTON

SimpleText app had:

```
<?xml version="1.0" encoding="utf-8"?>
<Button android:id="@+id/showElapsed"
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textSize="@dimen/hello_size"
    tools:text="123 seconds since started!" />
```

(from [SimpleButton/src/main/res/layout/activity_main.xml](#))

The primary difference is that this is a Button widget instead of a TextView widget.
Also:

- The android:id is now showElapsed
- We skip the padding and textStyle, to adopt the standard look for buttons
- We skip the android:text that normally would be here, as we are going to provide the text at runtime via our Java or Kotlin code
- We have a tools:text attribute that provides the text that will appear in the graphical layout editor, since we are not providing android:text in the layout itself

Hey, What Is That tools: Thing?

You will notice that our Button element has a tools:text attribute.

Attributes in the tools: namespace are suggestions to the development tools and have no impact on the behavior of your app when it runs. They are here to help make Android Studio work a bit better, particularly with respect to the graphical layout editor.

Normally, we have an android:text attribute on TextView and subclasses, and that provides the text. Here, though, we skip that attribute, as we are going to provide the text at runtime. However, that makes the graphical layout editor less useful:

- The caption of the Button shows up blank
- Since the width and height of the Button are each wrap_content, the graphical layout editor does not know how big to make the Button

tools:text says, “hey, Android Studio! use this for the Button caption for the graphical layout editor!”. This value will not show up when you run the app; it will

STARTING SIMPLE: TEXTVIEW AND BUTTON

only appear in the IDE. Typically, you set tools:text to either:

- A reasonable value, to see what that looks like
- Some extreme value, such as a really long string, to see what that looks like, in case you find that you need to adjust the GUI design to deal with that

Android Studio Graphical Layout Editor

As with the TextView widget, the Button widget is available in the “Buttons” portion of the Palette in the Android Studio graphical layout editor:

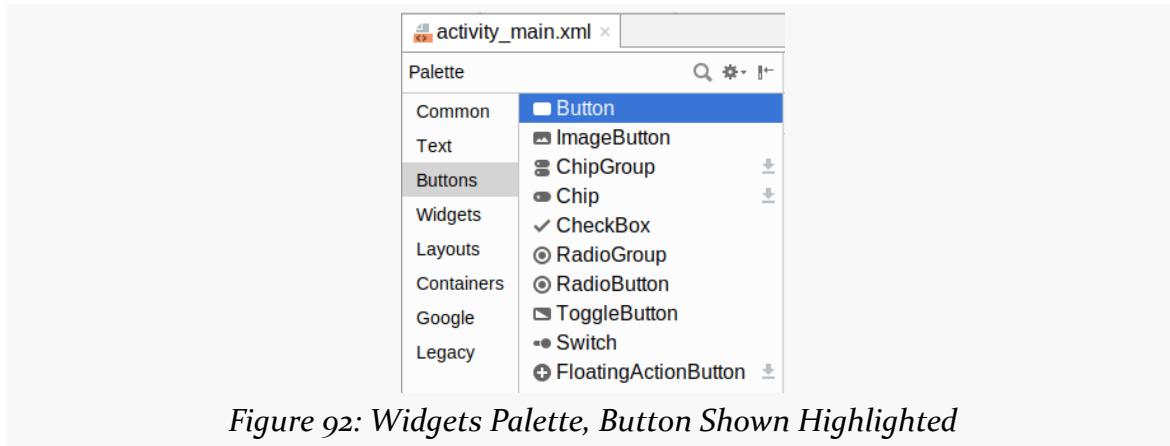


Figure 92: Widgets Palette, Button Shown Highlighted

You can drag that Button from the palette into a layout file in the main editing area to add the widget to the layout. The Attributes pane will then let you adjust the various attributes of this Button. Since Button inherits from TextView, most of the options are the same (e.g., “Text”).

Tracking Button Clicks

In the SimpleButton sample, we want to show the time since the activity was displayed in the button caption. Tapping the button should update the caption to show the now-current elapsed time.

This implies that:

- We can find out when the button is clicked, and
- We can set the android:text attribute at runtime with a generated value

Updating the caption of the Button is a matter of calling `setText()` on the Button

STARTING SIMPLE: TEXTVIEW AND BUTTON

with the desired caption. However, there are a few approaches for doing that, as well as for finding out about the button clicks.

The Classic Way

The primary approach for these things, for the first decade of Android app development, was:

- To call `findViewById()` on the Activity to retrieve widgets loaded from your layout
- To call `setOnClickListener()` on the Button to register an `OnClickListener`, whose `onClick()` method would get control when the user clicked the button

[The SamplerJ/SimpleButton edition](#) of `MainActivity` is written that way:

```
package com.commonsware.jetpack.samplej.simplebutton;

import android.os.Bundle;
import android.os.SystemClock;
import android.widget.Button;
import androidx.appcompat.app.AppCompatActivity;

public class MainActivity extends AppCompatActivity {
    private Button showElapsed;
    private long startTimeMs = SystemClock.elapsedRealtime();

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        setContentView(R.layout.activity_main);
        showElapsed = findViewById(R.id.showElapsed);
        showElapsed.setOnClickListener(v -> updateButton());
        updateButton();
    }

    void updateButton() {
        long nowMs = SystemClock.elapsedRealtime();
        String caption = getString(R.string.elapsed, (nowMs - startTimeMs) / 1000);

        showElapsed.setText(caption);
    }
}
```

STARTING SIMPLE: TEXTVIEW AND BUTTON

(from [SimpleButton/src/main/java/com/commonsware/jetpack/samplerj/simplebutton/MainActivity.java](#))

In `onCreate()`, we first call `setContentView()`, to tell the activity to display the UI that we have defined in the `activity_main` layout resource. After that, we call `findViewById()`, which will look in the widgets defined from that `activity_main` resource for one with an `android:id` value of `showElapsed`. We hang onto that in a `Button` field named `showElapsed`, so we do not have to keep looking it up via `findViewById()`.

Then, we call `setOnClickListener()` on the `Button`. Here, we are using a Java lambda expression that Java will convert into an `OnClickListener` for us, with the body of the lambda expression forming the body of the `onClick()` method and being called when the user clicks the button. There, we call an `updateButton()` method, which we also call from `onCreate()`.

`updateButton()` uses `SystemClock.elapsedRealtime()` to get the number of milliseconds that have elapsed since the device booted. `SystemClock` is a class supplied by the Android SDK that returns various time values that are tied to device activity. We use that same `elapsedRealtime()` method to populate a `startTimeMs` field in the activity, which will record when the activity was displayed. Hence, the number of milliseconds between the activity being displayed and now is a matter of subtracting `startTimeMs` from `nowMs`.

To display that in the `Button`, `updateButton()` calls `getString()` on the activity. `getString()` returns the value of a string resource, given the ID of that string resource. Here, we are looking to pull in an `elapsed` string resource. The `SimpleButton` project defines that using a combination of a message and a placeholder:

```
<resources>
    <string name="app_name">Jetpack: Button</string>
    <string name="elapsed">%d seconds since started!</string>
</resources>
```

(from [SimpleButton/src/main/res/values/strings.xml](#))

String resources support the same placeholder patterns as does Java's `String.format()` method, which is largely the same as what you might use in `sprintf()` in C/C++ development. Here, `%d` says "we will supply an integer to fill in here", which we do via the second parameter to `getString()`. The result is that `caption` will hold the string resource with the `%d` replaced by the number of seconds since the activity was started.

STARTING SIMPLE: TEXTVIEW AND BUTTON

That caption value is then passed to `setText()` on our Button, which causes it to show the caption.

Initially, that caption will show that 0 seconds have elapsed:

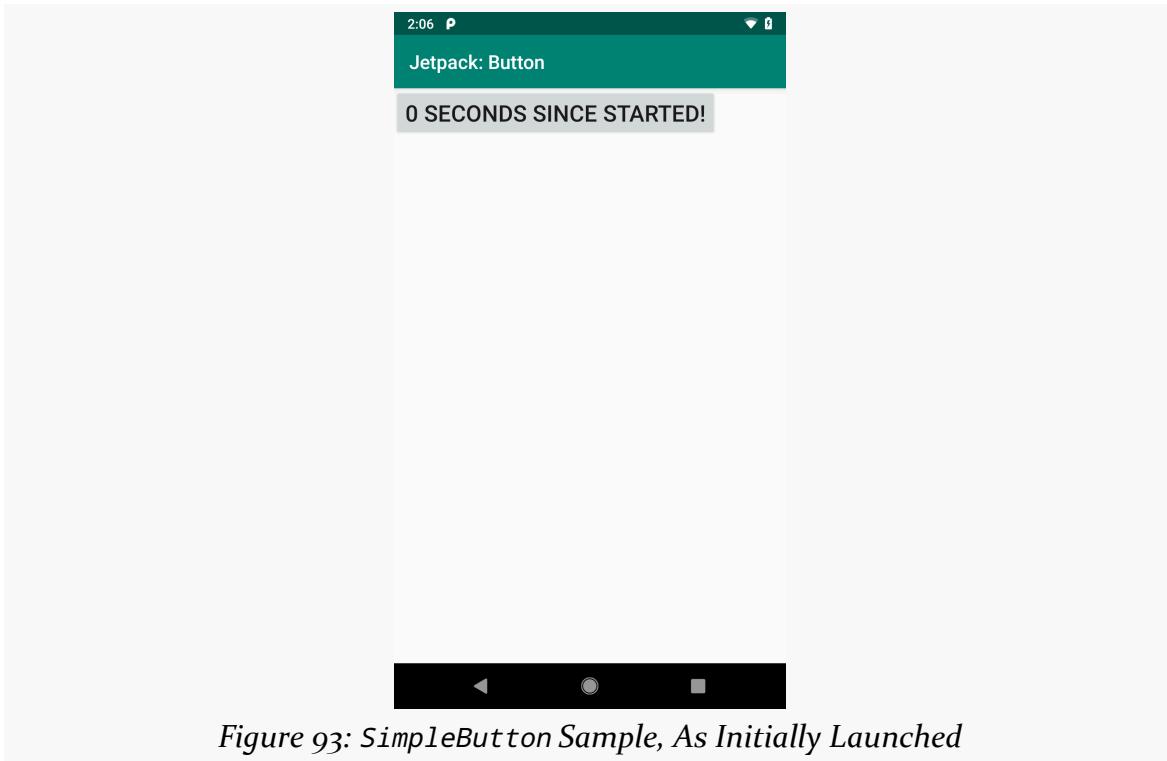


Figure 93: SimpleButton Sample, As Initially Launched

STARTING SIMPLE: TEXTVIEW AND BUTTON

If you tap it later, it will show the number of seconds since the activity was started:

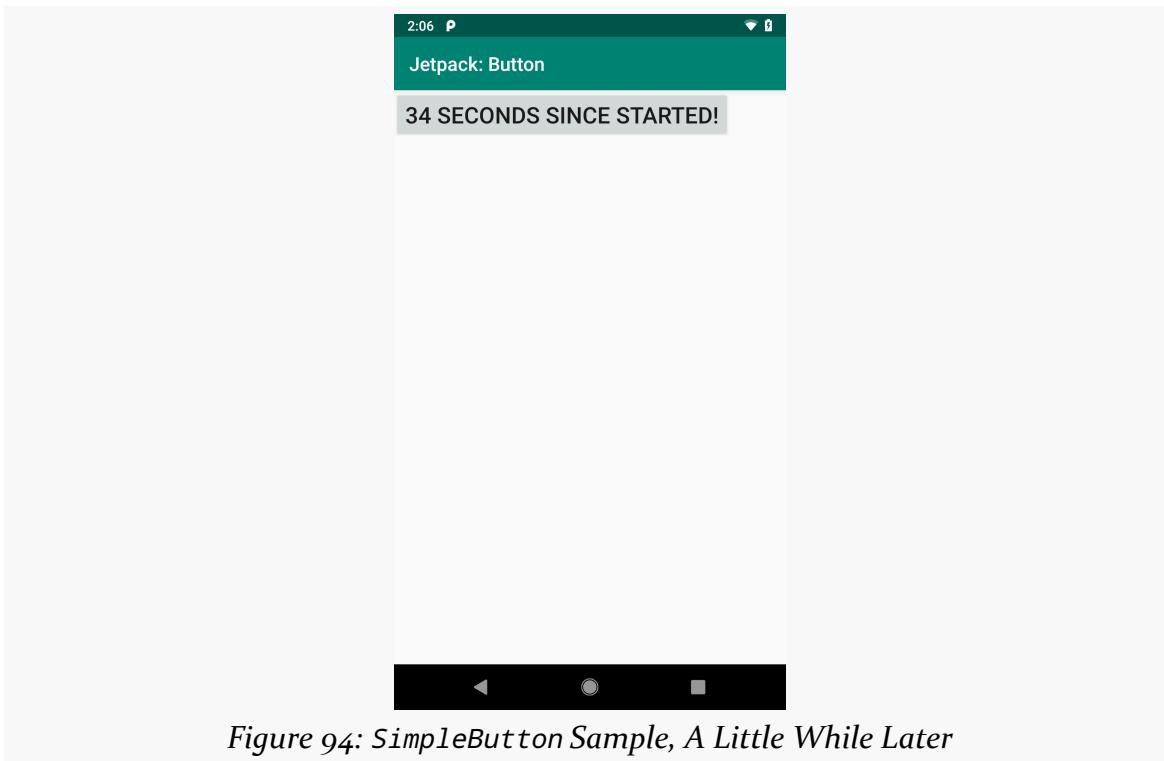


Figure 94: SimpleButton Sample, A Little While Later

Note that the Button changes size as the caption changes length. That is because the Button uses `wrap_content` for its width, so as the content changes, so will the Button width.

The Kotlin Improvements

Kotlin slightly simplifies this, as seen in [the Sampler/SimpleButton module](#):

```
package com.commonsware.jetpack.sampler.simplebutton

import android.os.Bundle
import android.os.SystemClock
import androidx.appcompat.app.AppCompatActivity
import kotlinx.android.synthetic.main.activity_main.*

class MainActivity : AppCompatActivity() {
    private val startTimeMs = SystemClock.elapsedRealtime()

    override fun onCreate(savedInstanceState: Bundle?) {
```

STARTING SIMPLE: TEXTVIEW AND BUTTON

```
super.onCreate(savedInstanceState)

setContentView(R.layout.activity_main)
showElapsed?.setOnClickListener { updateButton() }
updateButton()
}

private fun updateButton() {
    val nowMs = SystemClock.elapsedRealtime()
    val caption = getString(R.string.elapsed, (nowMs - startTimeMs) / 1000)

    showElapsed?.text = caption
}
}
```

(from [SimpleButton/src/main/java/com/commonsware/jetpack/sampler/simplebutton/MainActivity.kt](#))

One of the plugins that a Kotlin Android project gets by default is `kotlin-android-extensions`. One thing that we get from this plugin is “Kotlin synthetic accessors”. The effect is that we can add an `import` statement that references our layout resource (`import kotlinx.android.synthetic.main.activity_main.*`, referencing the `activity_main` resource). With that, we get properties for each of the named widgets in the layout, such as a `showElapsed` property. Under the covers, generated code will perform the `findViewById()` call for us.

Those properties return nullable Kotlin types, as it is possible that we screwed up and are asking for widgets from the wrong layout. So, when we call functions on it — such as `setOnClickListener()` — we need to use a safe call (e.g., `showElapsed?.setOnClickListener()`, `showElapsed?.text`).

Once again, we use a lambda expression to provide the `View.OnClickListener` implementation.



You can learn more about lambda expressions in the “Collections and Lambdas” chapter of [*Elements of Kotlin*](#)!

From the user’s standpoint, this looks and works the same as its Java equivalent.

Kotlin synthetic accessors work, but [they are a bit contentious](#). This book will use Kotlin synthetic accessors in most of its Kotlin samples, as it is the simplest approach for accessing views in Kotlin, but “simplest” != “best”, and you may wish to

use other approaches in other projects.

The Legacy Way

Some really old samples will use a slightly different approach:

- Have some Java method on the activity that returns void, takes a View as a parameter, and is public
- Have an android:onClick attribute on the Button, where the value of the attribute is the name of that Java method

Android would then call that method when the user clicks the Button.

The Data Binding Way

Jetpack's approach to this sort of work is to use what is known as "data binding". This allows us to add additional smarts to our layout resources beyond what ordinary layout resources can do. We will explore data binding and how it works [in an upcoming chapter](#).

The Curious Case of the Missing R

The Java and Kotlin code both refer to these R values. The R class gets code generated by the IDE and build tools based on our current roster of resources. Each resource gets a corresponding R value based on the type of resource (R.layout, R.id, R.string, etc.) and the name of the resource (R.layout.activity_main, R.id.showElapsed, R.string.elapsed, etc.).

Sometimes, when you are working on your Java or Kotlin code, you will run into cases where you cannot seem to use the R class to reference your resources. There are two main scenarios:

- The IDE says that it does not know what R is
- The IDE says that it knows what R is but does not recognize a particular resource that you are trying to reference (e.g., you have R.layout.foo, and either layout or foo are reported as not being recognized)

These problems come about due to where that R class lives and when it is created (or updated).

Where R Lives

All of our Java and Kotlin classes reside in some “package”. In the starter project, that package is `com.commonware.helloworld`. Our one-and-only class, `MainActivity`, is in that package. The samples shown in this chapter are in different packages, such as `com.commonware.jetpack.sampler.simplebutton` for the Kotlin edition of the `SimpleButton` sample.

The `R` class will be code-generated by the build tools in the Java package identified by the `package` attribute in the `<manifest>` element. So, for a project whose package is `com.commonware.jetpack.sampler.simplebutton`, the fully-qualified class name for `R` is `com.commonware.jetpack.sampler.simplebutton.R`.

Java and Kotlin share a common rule: you do not need to `import` classes that are in the same package as the class that you are in. So, in our projects, `MainActivity` can refer to `R` without an `import` statement, as both `MainActivity` and `R` are in the project’s package (e.g., `com.commonware.jetpack.sampler.simplebutton`).

If you try referencing `R`, and Android Studio complains that it does not know what `R` is, consider whether your class is in a different package than the one defined in the `<manifest>` element. If it is, you will need to add an `import` statement to pull in your `R` class.

When R Is Created

`R` is created as part of building your project. To allow code-completion and other assistance features of Android Studio to work, `R` gets re-created whenever you add, rename, or remove a resource, as `R` is an “index” of all of the resources in your app. However, occasionally, you will use some feature of Android Studio that happens to add, rename, or remove a resource, but Android Studio fails to regenerate that `R` class.

If Android Studio knows what `R` is but does not recognize some particular resource, try manually building the project, via `Build > Rebuild Project` from the Android Studio main menu. This forces `R` to get regenerated, and it may clear up your problem.

When R Is Not Created

Sometimes, though, Android Studio does not know what `R` is because `R` is missing

entirely.

If there is some bug in one of your resources — or, as it turns out, if there is some bug in your manifest — R may not be regenerated. If Android Studio says that it does not know what R is, and either you already have the `import` statement or you should not need one (since R and your code are in the same package), then it is likely that there is some problem in a resource or your manifest.

If you manually build the project — again, via Build > Rebuild Project from the Android Studio main menu — you should get a build error telling you exactly what resource is flawed, so you can try to fix it.

Package Names

Since the package value is used for Java code generation, it has to be a valid Java package name. Java convention says that the package name should be based on a reverse domain name (e.g., `com.commonsware.myapplication`), where you own the domain in question. That way, it is unlikely that anyone else will accidentally collide with the same name.

Licensed solely for use by Patrocinio Rodriguez

Debugging Your App

Now that we are starting to manipulate widgets, the odds increase that we are going to somehow do it wrong, and our app will crash. Usually, when your app crashes, the OS will show a dialog box:

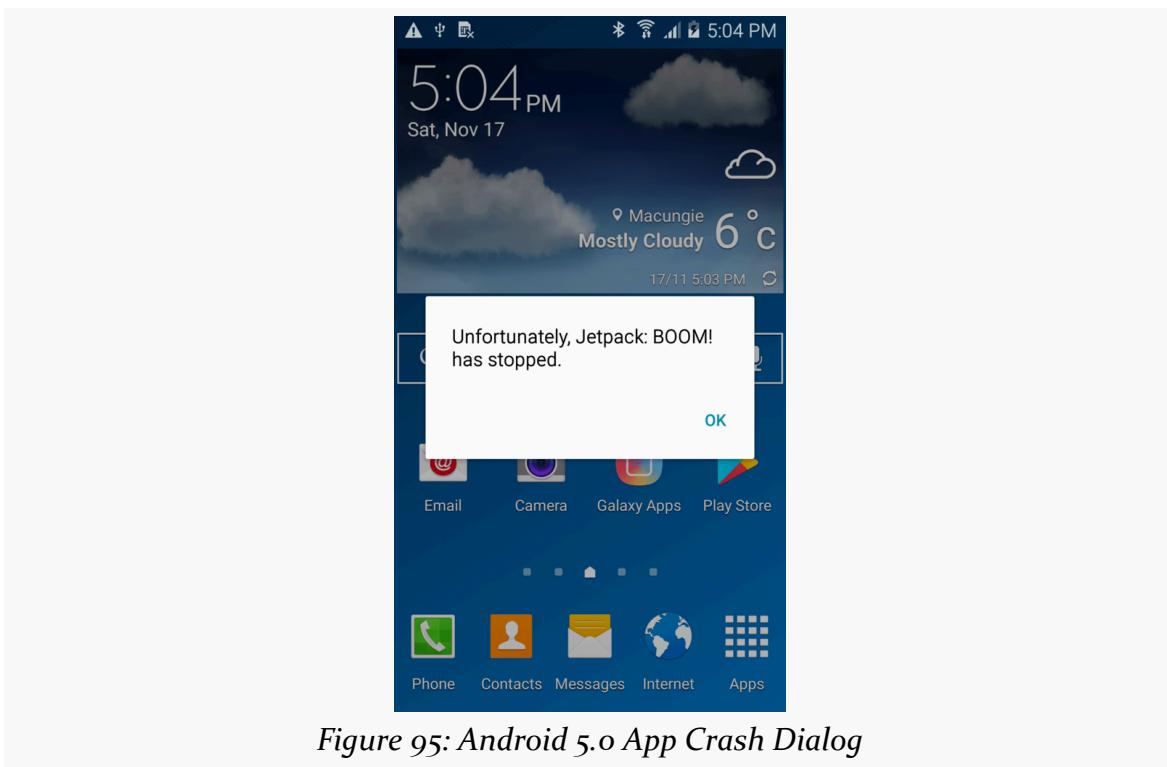


Figure 95: Android 5.0 App Crash Dialog

DEBUGGING YOUR APP

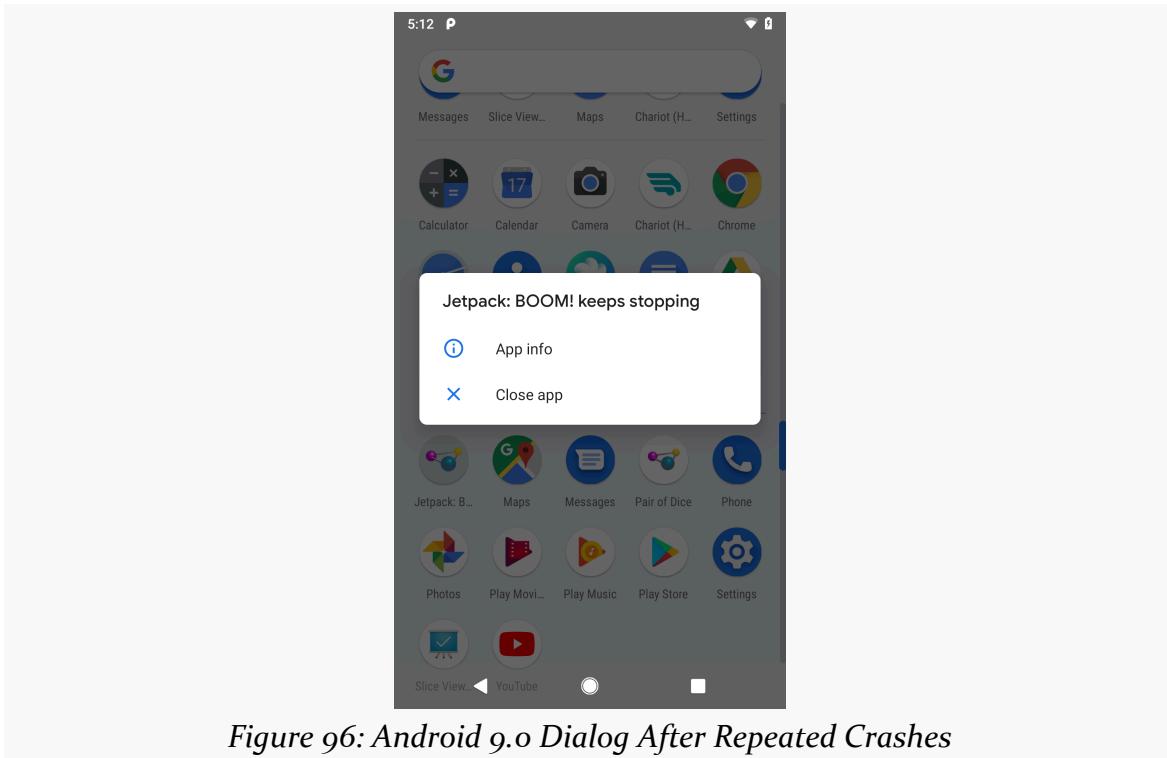


Figure 96: Android 9.0 Dialog After Repeated Crashes

In this chapter, we will cover a few tips on how to debug these sorts of issues.

Get Thee To a Stack Trace

If it seems like your app has crashed, the first thing you will want to do is examine the stack trace that is associated with this crash. These are logged to a facility known as Logcat, on your device or emulator. You can view those logs using the Logcat tool in Android Studio.

DEBUGGING YOUR APP

The Logcat tool is available at any time, from pretty much anywhere in Android Studio, by means of clicking on the Android tool window entry, usually docked at the bottom of your IDE window:



Figure 97: Minimized Tool Windows in Android Studio, Showing Logcat Tool

Clicking on that will bring up some Android-specific logs in an Logcat tool window:

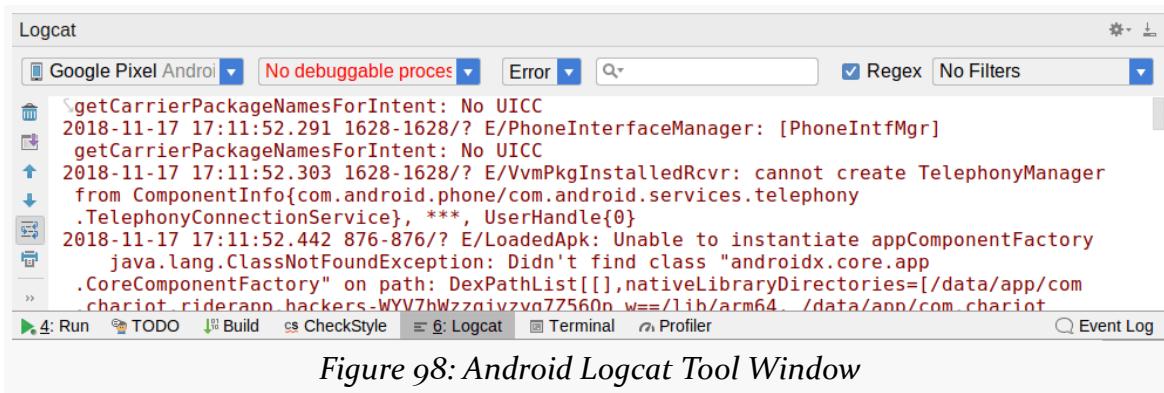


Figure 98: Android Logcat Tool Window

Logcat will show your stack traces, diagnostic information from the operating system, and anything you wish to include via calls to static methods on the `android.util.Log` class. For example, `Log.e()` will log a message at error severity, causing it to be displayed in red.

The toolbar across the top of the Logcat window has a drop-down list of available devices and running emulators. Whichever one is selected there is the source of the log messages seen in the main area of the Logcat tool window.

If your app crashes, most of the time, there will be an associated Java stack trace (even if you are writing Kotlin code). For example the stack trace that triggered the crash dialog shown at the start of this chapter is:

```
E/AndroidRuntime: FATAL EXCEPTION: main
    Process: com.commonsware.jetpack.sampler.simpleboom, PID: 14064
    java.lang.RuntimeException: Unable to start activity...
```

DEBUGGING YOUR APP

```
at android.app.ActivityThread.performLaunchActivity(ActivityThread.java:2693)
at android.app.ActivityThread.handleLaunchActivity(ActivityThread.java:2758)
at android.app.ActivityThread.access$900(ActivityThread.java:177)
at android.app.ActivityThread$H.handleMessage(ActivityThread.java:1448)
at android.os.Handler.dispatchMessage(Handler.java:102)
at android.os.Looper.loop(Looper.java:145)
at android.app.ActivityThread.main(ActivityThread.java:5942)
at java.lang.reflect.Method.invoke(Native Method)
at java.lang.reflect.Method.invoke(Method.java:372)
at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run...
at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:1195)
Caused by: android.content.res.Resources$NotFoundException: String resource...
at android.content.res.Resources.getText(Resources.java:1334)
at android.widget.TextView.setText(TextView.java:4917)
at com.commonsware.jetpack.sampler.simpleboom.MainActivity...
at com.commonsware.jetpack.sampler.simpleboom.MainActivity...
at android.app.Activity.performCreate(Activity.java:6283)
at android.app.Instrumentation.callActivityOnCreate(Instrumentation.java:1119)
at android.app.ActivityThread.performLaunchActivity(ActivityThread.java:2646)
at android.app.ActivityThread.handleLaunchActivity(ActivityThread.java:2758)
at android.app.ActivityThread.access$900(ActivityThread.java:177)
at android.app.ActivityThread$H.handleMessage(ActivityThread.java:1448)
at android.os.Handler.dispatchMessage(Handler.java:102)
at android.os.Looper.loop(Looper.java:145)
at android.app.ActivityThread.main(ActivityThread.java:5942)
at java.lang.reflect.Method.invoke(Native Method)
at java.lang.reflect.Method.invoke(Method.java:372)
at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run...
at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:1195)
```

(note: some lines were truncated with ... to try to make the output fit the page width better)

Most of the time — though not always — you should be able to find references to your code in the stack trace, to know where you were when you crashed.

If you want to send something from Logcat to somebody else, such as via an issue tracker, just highlight the text and copy it to the clipboard, as you would with any text editor.

The “trash can” icon atop the tool strip on the left is the “clear log” tool. Clicking it will appear to clear Logcat. It definitely clears your Logcat view, so you will only see messages logged after you cleared it. Note, though, that this does not actually clear the logs from the device or emulator.

In addition, you can:

- Use the “Log level” drop-down to filter lines based on severity, where messages for your chosen severity or higher will be displayed (e.g., only show “Error” severity)
- Use the search field to the right of the “Log level” drop-down to filter items based on a search string
- Set up more permanent filters via the drop-down to the right of the search field (e.g., “No Filters”)

Running Your App in the Debugger

The `SimpleBoom` module in the [Sampler](#) and [SamplerJ](#) projects is very similar to the `SimpleButton` example we looked at in [the chapter on widgets](#). There are three main differences:

- The button now has a text caption defined in the layout via the `android:text` attribute:

```
<?xml version="1.0" encoding="utf-8"?>
<Button android:id="@+id/showElapsed"
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/elapsed_caption"
    android:textSize="@dimen/hello_size" />
```

(from [SimpleBoom/src/main/res/layout/activity_main.xml](#))

DEBUGGING YOUR APP

- We no longer immediately update the button when starting the activity, so that caption shows up:

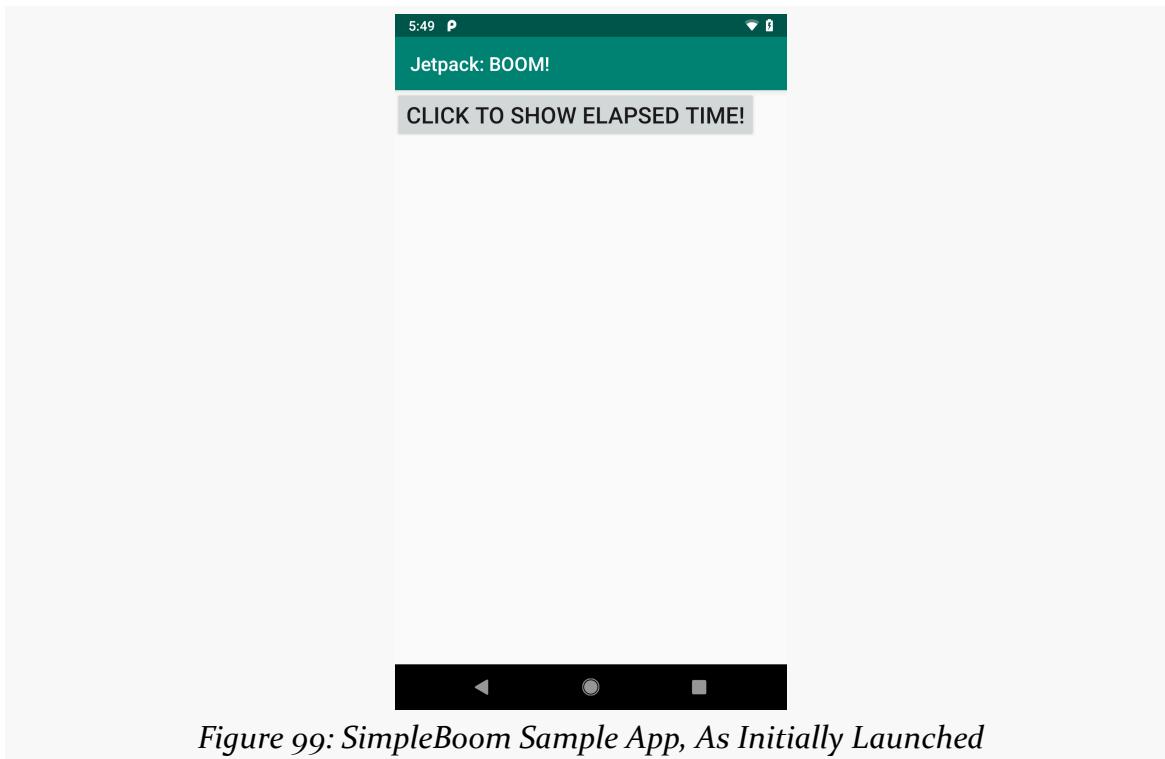


Figure 99: SimpleBoom Sample App, As Initially Launched

- We crash when the user taps the button

Sometimes, the stack trace alone will give you enough information to know how to fix the bug. If not, you can run your app in the debugger to see exactly how your code is executing and perhaps get a better sense for your mistake.

Setting Breakpoints

In our Java and Kotlin code editors in Android Studio, the vertical bar on the left is called “the gutter”:



Figure 100: Editor, with Gutter Area Highlighted

The author of this book likes having the line numbers visible. To control whether those show up, go to Files > Settings > Editor > General > Appearance — the “Show line numbers” checkbox controls whether the gutter shows line numbers.

Clicking in the gutter on a line of source code will set a breakpoint, which appears as a red dot in the gutter and a pink highlight on the line:

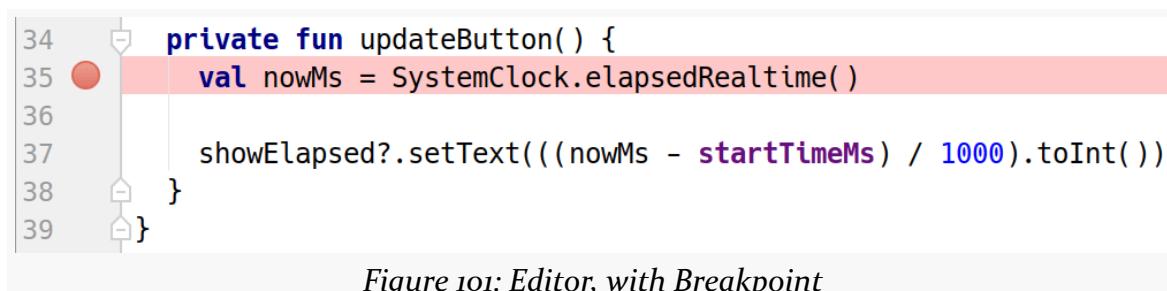
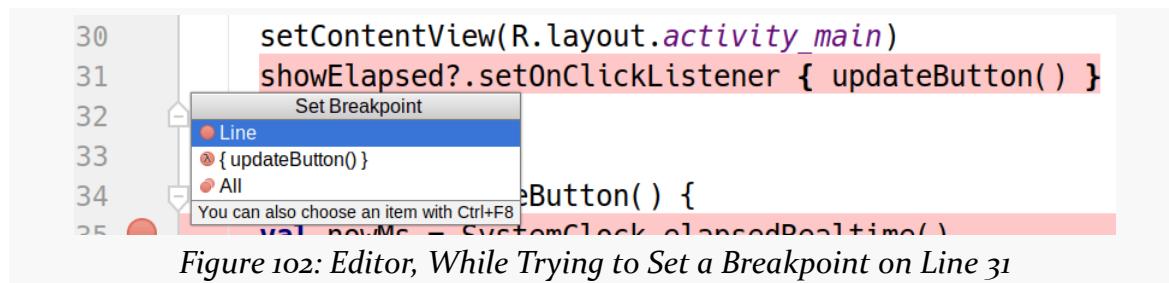


Figure 101: Editor, with Breakpoint

DEBUGGING YOUR APP

When you run your app using the debugger, when code execution reaches lines with a breakpoint, execution stops and you can use the IDE to learn more about what is going on.

For lines containing lambda expressions, when you click in the gutter to set a breakpoint, you will get a drop-down menu of options for what the breakpoint should affect:



Your options are:

- Set the breakpoint for the initial code on the line (“Line”)
- Set the breakpoint to trigger when the lambda expression is executed (“{ updateButton() }”)
- Both (“All”)

Launching the Debugger

To run the app and have the breakpoints take effect, you need to run it in the debugger. You can do this from the Run > Debug main menu option or the “green run triangle over a dark gray bug” toolbar icon:



DEBUGGING YOUR APP

This behaves a lot like when you normally run the app. However, the app will run more slowly, and when the breakpoint is reached, your IDE window will open a Debug tool that shows you what is going on:

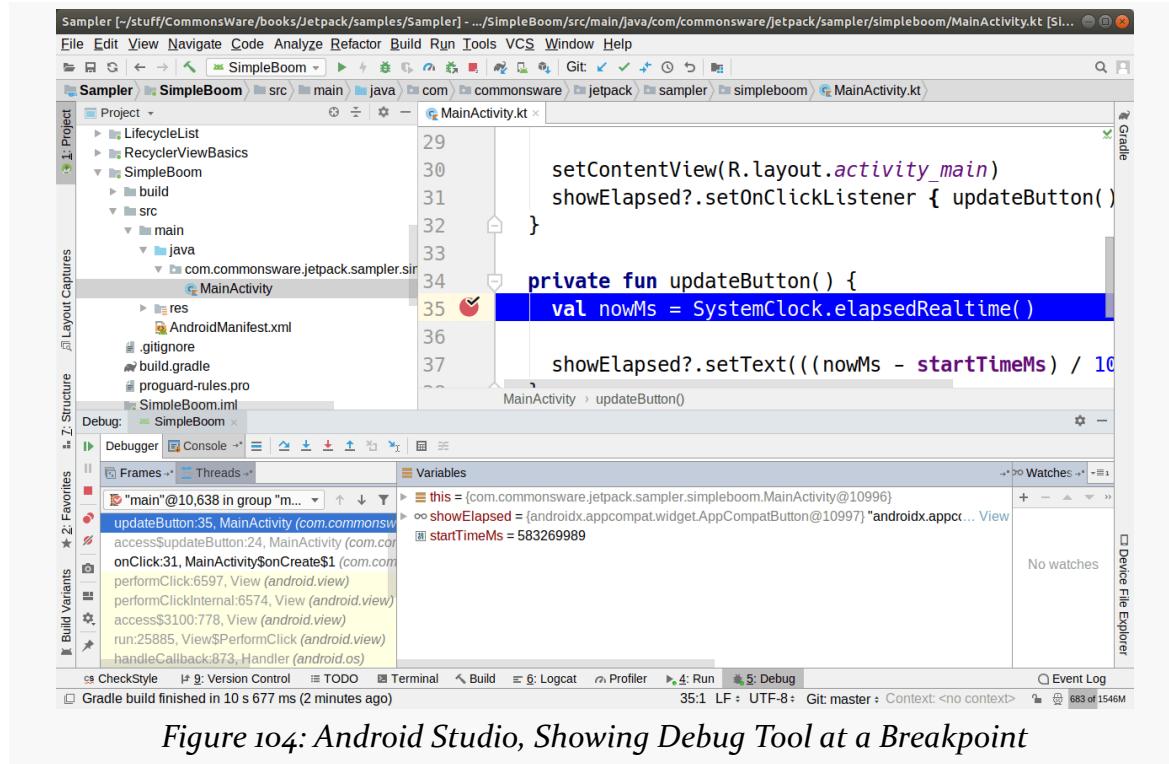


Figure 104: Android Studio, Showing Debug Tool at a Breakpoint

Examining Objects

The middle of the Debug tool shows a “Variables” list. This will contain local variables, fields/properties of the current object, and other values that your code might be referencing:



Figure 105: Variables Pane of Debug Tool

DEBUGGING YOUR APP

Simple types, like the Long value nowMs, just show their value. More complex types will show you their type and can be explored in turn using the gray triangle to expand the object tree:



Figure 106: Variables Pane of Debug Tool, Showing Object Contents

This can help you understand the data at this point in the code.

Stepping Through the Code

The toolbar towards the top of the Debug tool has a few buttons that let you step through the code to follow along as it gets execute:

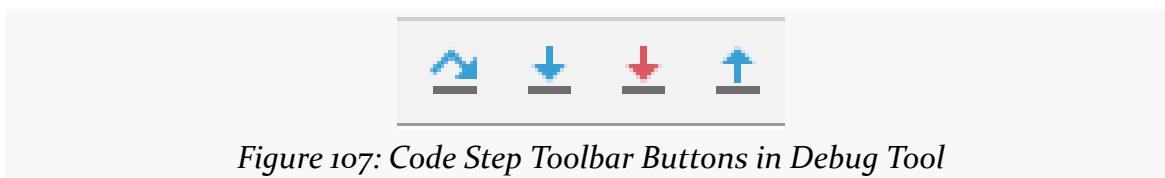


Figure 107: Code Step Toolbar Buttons in Debug Tool

From left to right, these are:

- Execute this statement and move into the next one
- Step into the function being called in this statement, if it is not an Android SDK function
- Step into the function being called in this statement, even if it is an Android SDK function
- Execute far enough to exit the function that we are in and then stop

DEBUGGING YOUR APP

Also, there is a tool strip on the left with a few useful buttons:

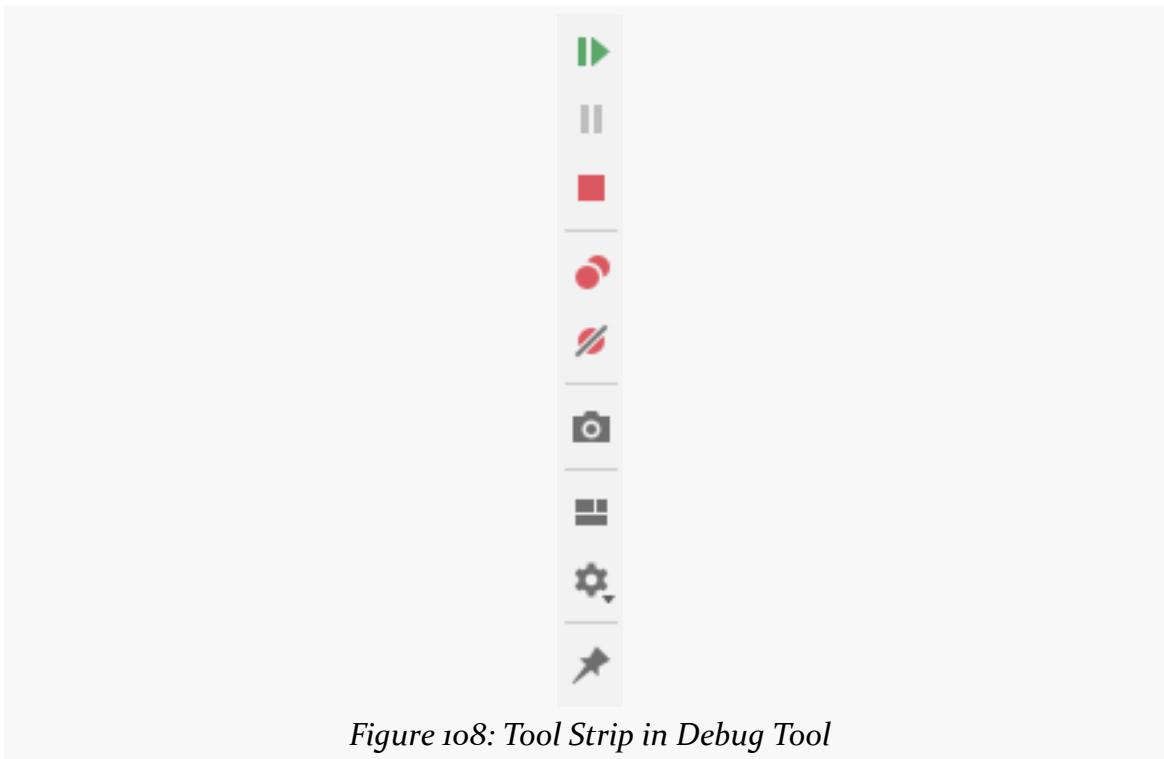


Figure 108: Tool Strip in Debug Tool

Of particular note:

- The top-most button (green triangle) will start executing code until the next breakpoint or crash
- The red square terminates your process
- The fourth button (two overlapping red dots) brings up a dialog box showing you all of your breakpoints

So, Where Did We Go Wrong?

In both the Java and Kotlin editions of `MainActivity` and its `updateButton()` method, we have a simpler bit of code that tries to set the caption to just be the number of seconds that have elapsed:

```
private fun updateButton() {  
    val nowMs = SystemClock.elapsedRealtime()  
  
    showElapsed?.setText(((nowMs - startTimeMs) / 1000).toInt())
```

DEBUGGING YOUR APP

```
}
```

(from [SimpleBoom/src/main/java/com/commonsware/jetpack/sampler/simpleboom/MainActivity.kt](#))

```
void updateButton() {
    long nowMs = SystemClock.elapsedRealtime();
    int seconds = (int)((nowMs - startTimeMs) / 1000);

    showElapsed.setText(seconds);
}
```

(from [SimpleBoom/src/main/java/com/commonsware/jetpack/samplerj/simpleboom/MainActivity.java](#))

This code compiles, because `setText()` on a Button has a variant that accepts an Int as a parameter. We are passing an Int that represents the number of seconds.

Let's look at the relevant portion of the stack trace again:

```
Caused by: android.content.res.Resources$NotFoundException: String resource ID #0x11
    at android.content.res.Resources.getText(Resources.java:1334)
    at android.widget.TextView.setText(TextView.java:4917)
    at com.commonsware.jetpack.sampler.simpleboom.MainActivity.updateButton...
```

As it turns out, `setText(Int)` does not show that Int value as text in the Button caption. Instead, Android is expecting that to be a reference to a string resource, such as `R.string.elapsed`. Those R values are all integers, and so many functions in the Android SDK that accept an Int are expecting a resource ID, not some arbitrary value as we are providing. In particular, if you get crash with an `android.content.res.Resources$NotFoundException`, there is a good chance that you are passing in a value to a function that is not a valid resource ID.

The correct way to implement this would be to convert the numeric value to a `String`, then pass that to `setText()`:

```
private fun updateButton() {
    val nowMs = SystemClock.elapsedRealtime()

    showElapsed?.setText(((nowMs - startTimeMs) / 1000).toString())
}

void updateButton() {
    long nowMs = SystemClock.elapsedRealtime();
    int seconds = (int)((nowMs - startTimeMs) / 1000);

    showElapsed.setText(Integer.toString(seconds));
}
```

Introducing ConstraintLayout

The starter app that we examined in the opening chapters had a layout like this:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

This wraps a single `TextView` in a `ConstraintLayout`. For the purposes of what this “hello, world!” app does, the `ConstraintLayout` is pointless — we saw activities using just a `TextView` in [the chapter on widgets](#).

However, more often than not, you will have more than one widget in your layout resources. When you do, you will always have some sort of container, and frequently that container will be a `ConstraintLayout`.

In this chapter, we will review what containers are and why we use them, plus

INTRODUCING CONSTRAINTLAYOUT

examine some basic scenarios for using ConstraintLayout.

The Role of Containers

Some apps have screens that contain just one widget.

Not many, though.

Most of the time, our user interface is more complicated than that... even if it does not seem that complicated when you look at it:

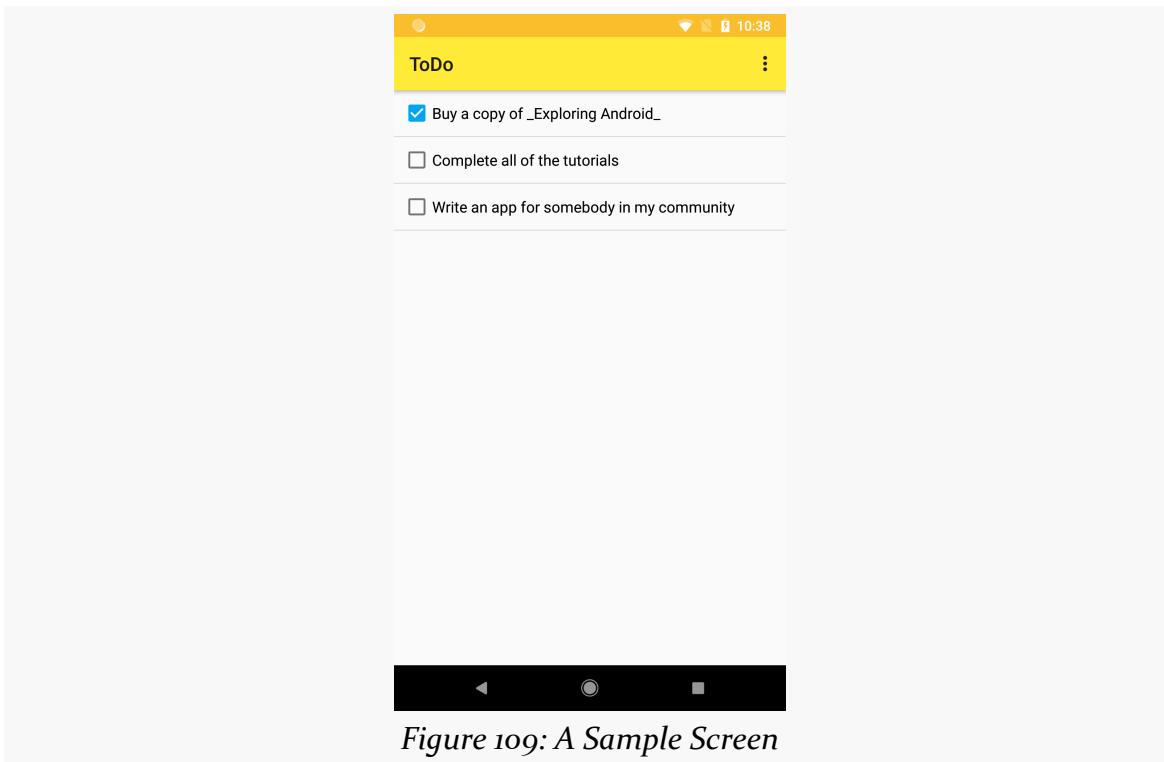


Figure 109: A Sample Screen

Here we have:

- Three checkboxes
- Three labels adjacent to those checkboxes
- Some sort of title (“ToDo”)
- A strange little “...” icon in the upper right
- A yellow background behind the title and icon

INTRODUCING CONSTRAINTLAYOUT

This takes more than one XML element to represent in a layout resource. In fact, this takes more than one layout resource. This app is a “to-do list” sort of app, and it can handle an arbitrary number of to-do items. So while this screenshot shows three rows of checkboxes-and-labels, really the user could have 0 to N of them.

As we covered previously, containers extend from `ViewGroup`, and the job of a `ViewGroup` is to organize some number of children and display them on the screen. `ViewGroup` itself extends from `View`, so the children of a `ViewGroup` can be a mix of `View` and `ViewGroup` objects.

So, in this screenshot, we clearly have at least one `ViewGroup` that is organizing all of those widgets and displaying them.

Layouts and Adapter-Based Containers

Roughly speaking, there are two major types of `ViewGroup` implementations:

- There are those that organize a set of children known at the time that you are writing the app
- There are those that organize a set of children that cannot be known until the app is run

For example, if we were to write a to-do list app like the one shown above, we do not know how many to-do items there are in the list. That depends on how many the user enters that we save in a database, a Web service, or some other place. However, for each individual to-do list item, we know that the visual representation of that item is a checkbox and a label showing the task to be performed.

So, in this case, we have both types of `ViewGroup`. The list of to-do items is managed by a `ViewGroup` that is optimized for organizing data at runtime, while each row in that list is managed by a `ViewGroup` that is optimized for organizing data when we write the app.

The most popular `ViewGroup` for organizing data at runtime is `RecyclerView`, though there are other options. We will explore `RecyclerView` more [later in the book](#). The recommended `ViewGroup` for organizing widgets in layout resources is a `ConstraintLayout`, though there are other options.

ConstraintLayout: One Layout To Rule Them All

If you survey the Internet, you will find that ConstraintLayout is popular but relatively new. You will find references to other ...Layout classes, such as LinearLayout and RelativeLayout. We will explore those briefly [later in this chapter](#). Suffice it to say that LinearLayout and RelativeLayout existed from the beginning (2008's Android 1.0), while ConstraintLayout debuted in 2016. And, there is nothing that ConstraintLayout can do that cannot also be accomplished by some mix of the classic layout classes (e.g., LinearLayout).

So, why did Google bother with ConstraintLayout?

Drag-and-Drop GUI Builders

Google would like everyone to use Android Studio, and in particular for everyone to use Android Studio's drag-and-drop GUI builder.

How well a drag-and-drop GUI builder works depends a lot on how the rules for laying out a UI get defined. With drag-and-drop gestures, the developer is only providing you with X/Y coordinates of a widget, based on where the developer releases the mouse button and completes the drop. It is up to the GUI builder to determine what that really means in terms of layout rules.

However, in 2008, we had no drag-and-drop GUI builder. LinearLayout, RelativeLayout, and kin were not designed with drag-and-drop in mind. As it turns out, some of those (e.g., LinearLayout) work well in a drag-and-drop model, while others (e.g., RelativeLayout) do not.

By contrast, ConstraintLayout was designed with drag-and-drop GUI building in mind.

Performance

The classic layout classes were not written with performance in mind. Some of their features work well but are a bit slow to execute. They are not so slow as to be unusable, but they do make the app a bit more sluggish, particularly when scrolling the screen.

The objective of ConstraintLayout was to offer all of the capabilities of the classic containers — and more — while keeping performance in mind. In truth, the authors

INTRODUCING CONSTRAINTLAYOUT

of ConstraintLayout had partial success here, as ConstraintLayout helps some scenarios but has limited benefit in others.

Library vs. Framework

LinearLayout, RelativeLayout, and the other classic containers are framework classes. As a result, they could have different implementations on different versions of Android. Since that causes a lot of pain for maintaining backwards compatibility, Google does not change these classes that often.

ConstraintLayout, by contrast, comes from a library. This has a cost, as now each app needs its own copy of the ConstraintLayout code. However, it also means that Google can keep improving ConstraintLayout, and developers can adopt newer versions of the library that add new features or fix bugs.

Getting ConstraintLayout

Since ConstraintLayout comes from a library, we need to ensure that we have this library in our module. If you create the project from the Android Studio new-project wizard, it is very likely that you already have the library in your dependencies.

What you are looking for is androidx.constraintlayout:constraintlayout for some version:

```
dependencies {  
    implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlin_version"  
    implementation 'androidx.appcompat:appcompat:1.0.2'  
    implementation 'androidx.constraintlayout:constraintlayout:1.1.3'  
}
```

(from [ConstraintRow/build.gradle](#))

You may find that you have a different artifact, com.android.support.constraint:constraint-layout. This too provides ConstraintLayout, but it is the older artifact, from [the Android Support Library](#). If you have other artifacts in your dependencies that are androidx-based, you will want to use the androidx edition of the constraintlayout artifact.

Using Widgets and Containers from Libraries

In [an earlier chapter](#), we saw that layout resources use XML elements to describe the

INTRODUCING CONSTRAINTLAYOUT

UI. The elements that we explored — `TextView` and `Button` — had simple element names, plus attributes that were prefixed with `android:`. This is how you will use most of the widgets and containers that are part of the “framework classes” — the classes that are part of Android itself and ship with Android devices.

However, `ConstraintLayout` is from a library, so it is not part of the framework classes. A copy of the code for `ConstraintLayout` is packaged in your APK and ships with your app.

Using widgets and containers from libraries is very similar to using widgets and containers from the framework classes, with two key differences.

Fully-Qualified Class Name

In [the chapter on widgets](#), we focused on `TextView` and `Button`. Those widgets have XML elements named `<TextView>` and `<Button>`, respectively.

Most framework widgets work that way: the XML element can be the bare class name. No library widgets work that way. For those, the XML element needs to be the *fully-qualified* class name. So, when using `ConstraintLayout`, you will see XML elements like `<androidx.constraintlayout.widget.ConstraintLayout>`. This gets a bit wordy if you are working with the XML, but if you are mostly using the drag-and-drop GUI builder, you will barely notice the difference.

app: Attributes

You will also find that some of the XML attributes do not get the `android:` prefix, but instead get an `app:` prefix. These are XML attributes defined by the library code, whereas `android:`-prefixed attributes come from framework code.

This means that the first time you start using library widgets and containers in a layout resource, you might see `app` show up in red:

```
<TextView android:id="@+id/label"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="I am a label for:"
    app:layout_constraintBaseline_toBaselineOf="@+id/button"
    app:layout_constraintStart_toStartOf="parent" />
```

Figure 110: Android Studio Layout XML Editor, Showing Red app

INTRODUCING CONSTRAINTLAYOUT

This means that the app XML namespace has not been declared. The simplest way to add it is to:

- Put your text cursor somewhere in the red-highlighted app text
- Press Alt-Enter (or the equivalent for macOS) to bring up the “quick-fix” menu:



Figure 111: Android Studio Layout XML Editor, Showing Quick-Fix Menu

- Choose “Create namespace declaration” from the menu

This will add `xmlns:app="http://schemas.android.com/apk/res-auto"` to your root XML element of the layout resource.

A Quick RTL Refresher

Most of the world’s languages are written left-to-right. So, in this paragraph, you read the letters and words starting from the left edge of a line across to the right edge.

Arabic and Hebrew, [among others](#), are written right-to-left. The abbreviation “RTL” refers to these languages. LTR, in turn, refers to left-to-right languages.

Android supports both LTR and RTL. We noted in [the chapter introducing the manifest](#) how your app can advertise that it supports RTL. This, in turn, will cause you to refer to “start” and “end” instead of “left” and “right” when positioning widgets on the screen:

Language Direction	“Start” Means...	“End” Means...
LTR	Left	Right
RTL	Right	Left

INTRODUCING CONSTRAINTLAYOUT

In general, we want the GUI to flow with the language direction. Things that you might have on the left with an LTR language usually go on the right with an RTL language, and so forth.

Simple Rows with ConstraintLayout

The ConstraintRow sample module in the [Sampler](#) and [SamplerJ](#) projects is very similar to the SimpleText sample. The layout now has a ConstraintLayout wrapped around a TextView and a Button.

The XML

Our root element is now

```
<androidx.constraintlayout.widget.ConstraintLayout>:
```

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="@dimen/container_padding">

    <TextView android:id="@+id/label"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/label_caption"
        app:layout_constraintBaseline_toBaselineOf="@id/button"
        app:layout_constraintStart_toStartOf="parent" />

    <Button
        android:id="@+id/button"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:text="@string/button_caption"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintStart_toEndOf="@+id/label"
        app:layout_constraintEnd_toEndOf="parent"/>

</androidx.constraintlayout.widget.ConstraintLayout>
```

(from [ConstraintRow/src/main/res/layout/activity_main.xml](#))

We give that ConstraintLayout a height and width of `match_parent`, which

INTRODUCING CONSTRAINTLAYOUT

indicates that this widget or container should fill up the available space of its parent container. In this case, the “parent” is a container that occupies most of the space on the screen, so the ConstraintLayout will fill that space. We also give the ConstraintLayout padding on all four sides, so the contents of the ConstraintLayout will be inset from the edges by however much @dimen/container_padding calls for (8dp in this case).

There are two XML elements that are contained in the ConstraintLayout: our TextView and our Button. Both have IDs, sizes, and captions (via android:text). However, both also have anchoring rules, indicating where the widgets should be positioned within the ConstraintLayout.

The TextView has app:layout_constraintStart_toStartOf="parent". All attributes starting with app:layout_constraint are rules for children of ConstraintLayout. app:layout_constraintStart_toStartOf="parent" says:

- We are trying to constrain the “start” edge of the widget (constraintStart), where the “start” edge is on the left for left-to-right languages and on the right for right-to-left languages
- We are trying to anchor that edge to the start edge of something else (toStartOf)
- The “something else” is the ConstraintLayout itself (“parent”)

So, app:layout_constraintStart_toStartOf="parent" will anchor the start edge of the TextView to the start side of the ConstraintLayout.

Similarly, the Button has:

- app:layout_constraintTop_toTopOf="parent" to anchor the top of the Button to the top of the ConstraintLayout
- app:layout_constraintEnd_toEndOf to anchor the “end” of the Button to the “end” of the ConstraintLayout, where “end” is on the right for left-to-right languages and on the left for right-to-left languages

The TextView also has app:layout_constraintBaseline_toBaselineOf="@id/button". Here, “baseline” refers to the invisible line that text appears to “sit” upon. A TextView — and any subclasses, like Button — has a baseline. Here, we are anchoring the baseline of the TextView to the baseline of the Button, so wherever the Button winds up, the TextView will have a matching vertical position. The "@id/button" ties into the @+id/button declared in the android:id attribute of the Button, so the ConstraintLayout knows what widget is the target of this anchoring

INTRODUCING CONSTRAINTLAYOUT

rule.

The Button also has `app:layout_constraintStart_toEndOf="@+id/label"`. This says that we want to anchor the start edge of the Button to the end edge of the TextView (whose `android:id` value is `@+id/label`).

The Button has an unusual `android:layout_width` value: `0dp`. Normally, this would mean a width of zero dp, which would be a bit short. For a child of ConstraintLayout, though, `0dp` means “the width is determined by the horizontal constraints”. In the case of the Button:

- The start edge is anchored to the end edge of the TextView
- The end edge is anchored to the end edge of the ConstraintLayout

As a result, with a width of `0dp`, the Button will be *stretched* to fill the space between those two anchor positions. If instead the width were `wrap_content`, the Button width would be determined by its caption, and it would be *centered* in between those two anchor positions.

The Android Studio Graphical Layout Editor

If you click on a widget in the ConstraintLayout in the blueprint view, that view will show squares on the corners and circles centered on the edges:

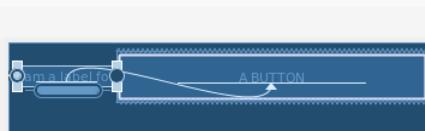


Figure 112: Android Studio Graphical Layout Editor, Blueprint View

The squares are resize handles. Most likely, you have seen this pattern before, whether in IDEs, drawing tools, or other programs. You would use this resizing approach if you wanted a fixed size for the widget. Later switching to using dimension resources, rather than hard-coded values, for the size values would be a good idea. You can also change the width and height through the Attributes pane.

INTRODUCING CONSTRAINTLAYOUT

The circles are more important, as they allow you to define the constraints, by dragging a circle to some anchor point:

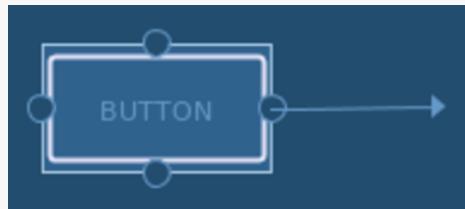


Figure 113: Blueprint View, Showing Constraint Being Created

You can drag the circle to an equivalent circle on another widget or to the edges of the ConstraintLayout, to establish an anchoring rule between those two points. That rule is represented by an arrow connecting the two widgets, with the start and end of the arrow showing the sides that were constrained.

The Result

If you run either edition of the ConstraintRow sample, you will see the result matches the graphical layout editor and what we asked for in the XML attributes:

INTRODUCING CONSTRAINTLAYOUT

- The TextView is aligned on the start edge of the screen and is vertically aligned with the baseline of the Button
- The TextView takes up its “natural” space based upon its caption
- The Button fills the space between the TextView and the end edge of the screen
- The Button is aligned with the top of the screen
- Everything is inset 8dp from the edges

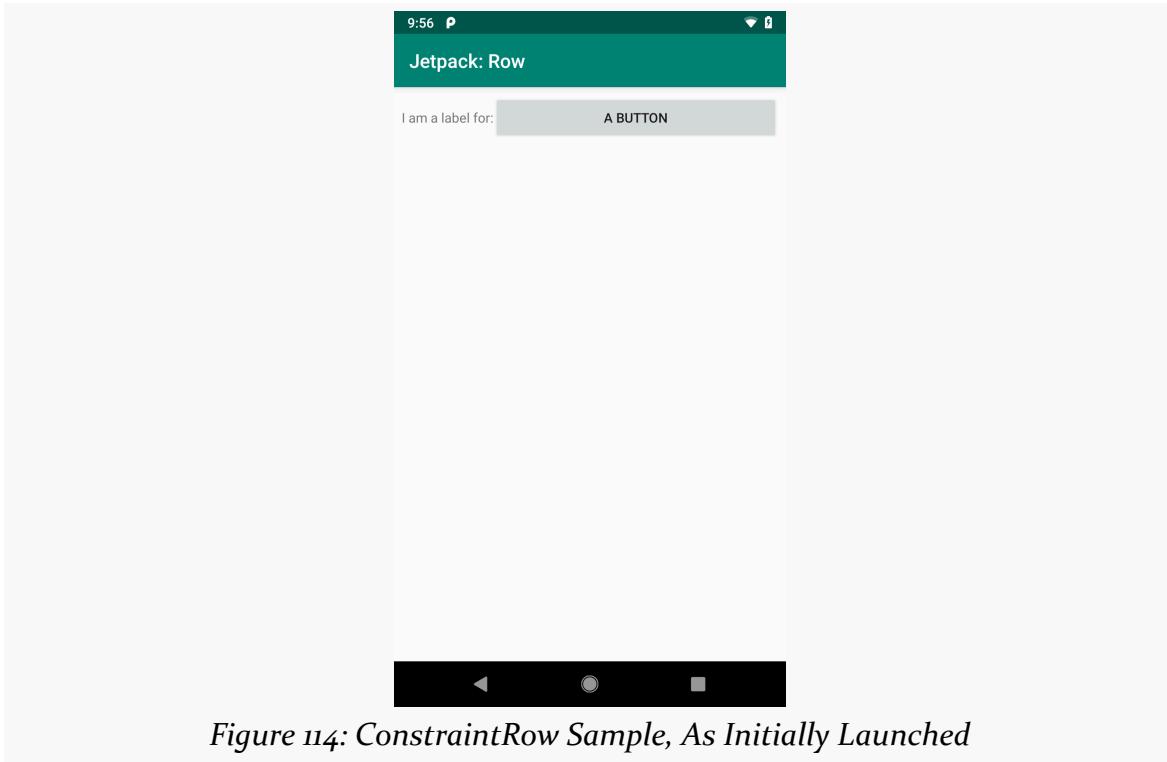


Figure 114: ConstraintRow Sample, As Initially Launched

Starting from Scratch

As with other containers, you can create a new layout resource with a ConstraintLayout as the root, by right-clicking over a layout resource directory, choosing New > “Layout resource file” from the context menu, and typing in ConstraintLayout for the root element. Fortunately, auto-complete on the “Root element” field allows you to just start typing ConstraintLayout, then choose the fully-qualified class name from the drop-down list.

If you drag a widget into the ConstraintLayout and drop it in an arbitrary spot, what you get at design time will be different than what you get when you run the

INTRODUCING CONSTRAINTLAYOUT

app. In the graphical layout editor, the Button shows up where you drop it:

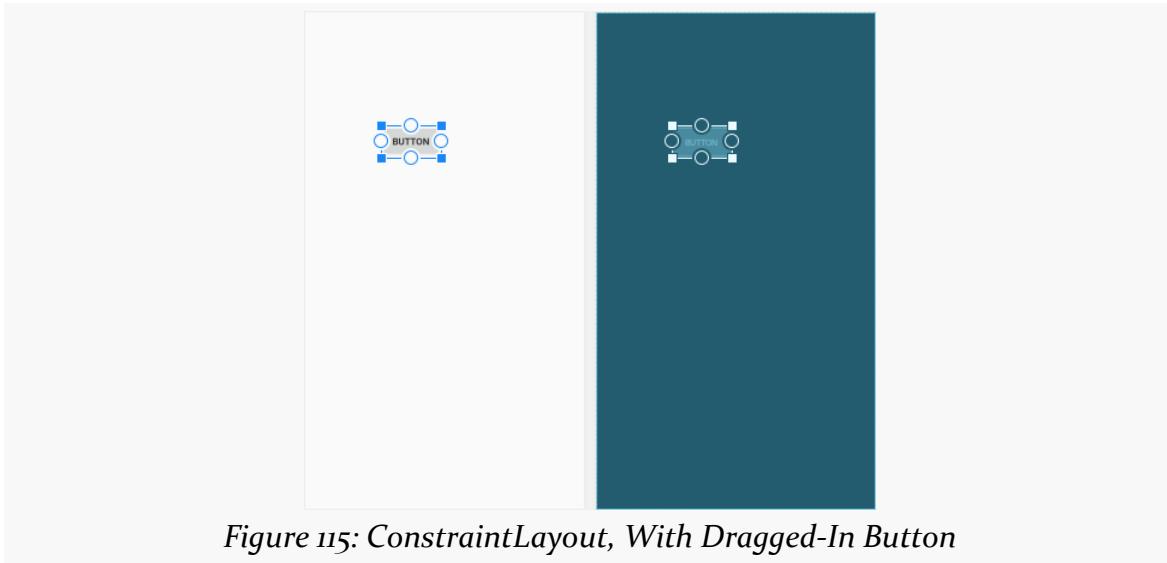


Figure 115: ConstraintLayout, With Dragged-In Button

However, if you look at the XML that was generated, you will see that the Button has no constraints. It *does* have a pair of attributes with the tools: prefix:

tools:layout_editor_absoluteX and tools:layout_editor_absoluteY:

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

    <Button
        android:id="@+id/button9"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Button"
        tools:layout_editor_absoluteX="77dp"
        tools:layout_editor_absoluteY="36dp" />
</android.support.constraint.ConstraintLayout>
```

As we discussed earlier in the book, attributes in the tools: namespace are suggestions to the development tools and have no impact on the behavior of your app when it runs. In this case, Android Studio remembers the upper-left corner of where you dropped the Button. But, as a warning on the Button in the layout editor will tell you, a Button without constraints will wind up at coordinate (0, 0) at

INTRODUCING CONSTRAINTLAYOUT

runtime (basically, upper-left for LTR languages and upper-right for RTL languages).

Dragging in a widget is insufficient. You also need to use the graphical layout editor to define the constraints, dragging the circles on the widget's edges and connecting them to the edges of other widgets or to the ConstraintLayout itself.

ConstraintLayout and the Attributes Pane

When you click on a widget inside a ConstraintLayout, the Attributes pane has a strange-looking control in the “Layout” category of attributes:

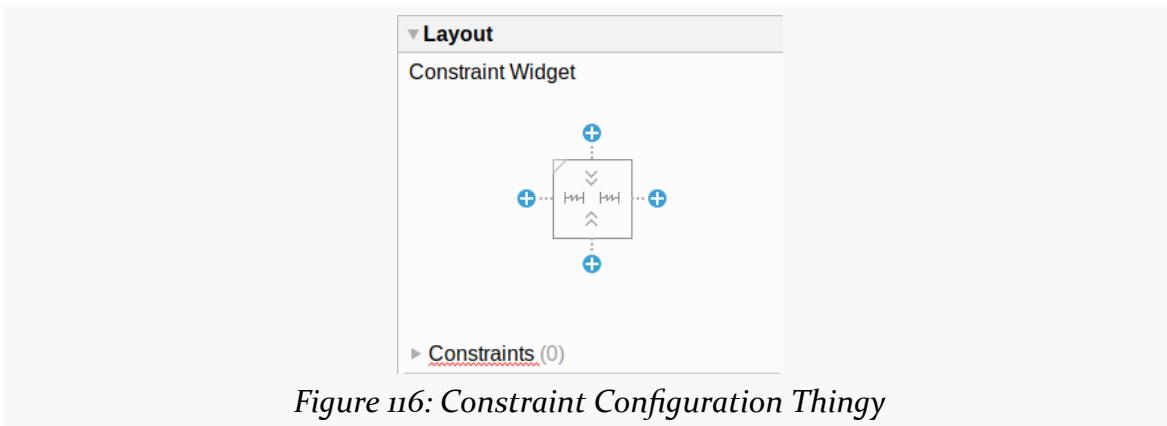


Figure 116: Constraint Configuration Thingy

The sawtooth lines inside the square indicate that the sizing rule is 0dp for the horizontal axis. Clicking on one of the sawtooth lines will toggle between states for that axis:

- `wrap_content`, indicated by chevrons (what the screenshot shows for the vertical axis)
- a fixed width, indicated by a sizing bar
- 0dp, for stretching the size to the available space, indicated by a sawtooth line

EditText: Making Users Type Stuff

Along with buttons and labels, fields are the third “anchor” of most GUI toolkits. In Android, they are implemented via the `EditText` widget, which is a subclass of the `TextView` used for labels.

Along with the standard `TextView` attributes (e.g., `android:textStyle`), `EditText`

INTRODUCING CONSTRAINTLAYOUT

has others that will be useful for you in constructing fields, notably `android:inputType`, to describe what sort of input your `EditText` expects (numbers? email addresses? phone numbers?).

Graphical Layout Editor

The Palette in the graphical layout editor has a whole section dedicated primarily to `EditText` widgets, named “Text”:

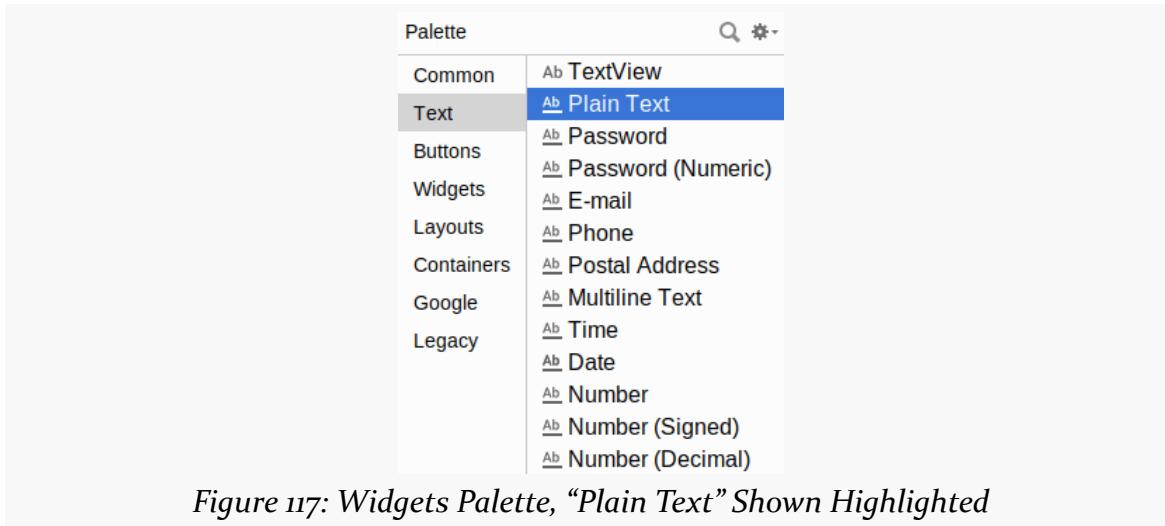


Figure 117: Widgets Palette, “Plain Text” Shown Highlighted

The first entry is a `TextView`. The second entry (“Plain Text”) is a general-purpose `EditText`. The rest come pre-configured for various scenarios, such as a password.

You can drag any of these into your layout, then use the Attributes pane to configure relevant attributes, or edit the `EditText` XML as you see fit. The “Id” and “Text” attributes are the same as found on `TextView`, as are many other attributes, inherited from `TextView`.

Notable Attributes

The “Hint” item in the Attributes pane allows you to set a “hint” for this `EditText`. The “hint” text will be shown in light gray in the `EditText` widget when the user has not entered anything yet. Once the user starts typing into the `EditText`, the “hint” vanishes. This might allow you to save on screen space, replacing a separate label `TextView`.

The “Input Type” item in the Attributes pane allows you to describe what sort of

INTRODUCING CONSTRAINTLAYOUT

input you are expecting to receive in this `EditText`, lining up with many of the types of fields you can drag from the Palette into the layout:

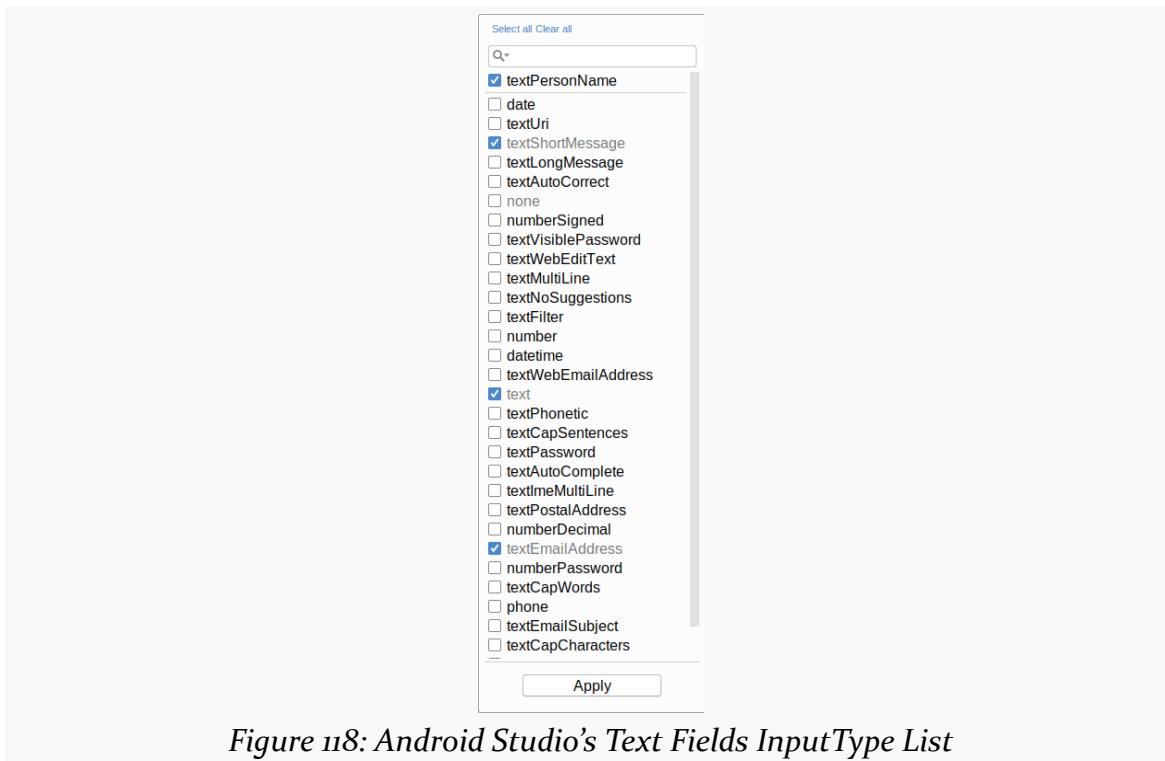


Figure 118: Android Studio's Text Fields InputType List

More Complex Forms

Let's look at a bit more elaborate example, found in the `ConstraintForm` modules of the [Java](#) and [Kotlin](#) projects.

Here, we have a form that collects the first and last name of a person, with a `Button` to submit the form. We are not reacting to that button click — the purpose of this sample is to see how the form is constructed.

What We Want

This seems straightforward enough. We can have a `ConstraintLayout` that collects the first name and last name in two rows, with the `Button` below the last name row.

INTRODUCING CONSTRAINTLAYOUT

However, if we limit ourselves to the sort of ConstraintLayout anchoring rules that we have seen above, we would wind up with a layout that looks like this:

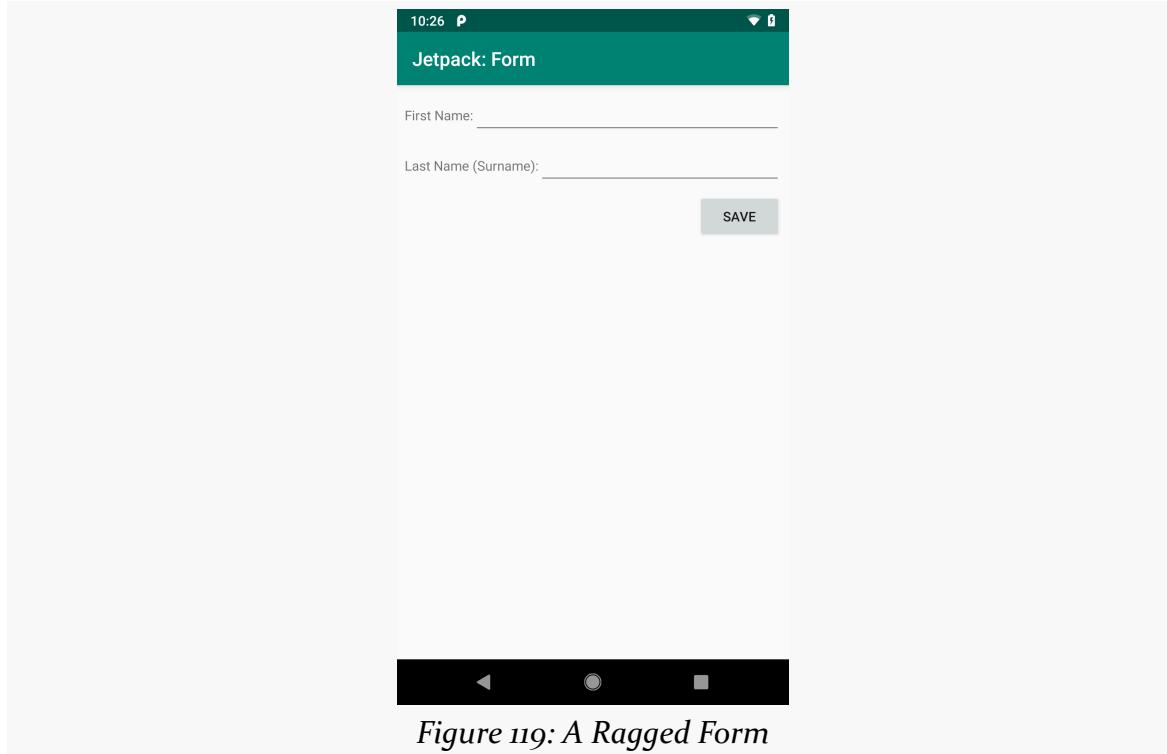


Figure 119: A Ragged Form

INTRODUCING CONSTRAINTLAYOUT

This works, but it would be nicer if we had columns for the labels and the fields:

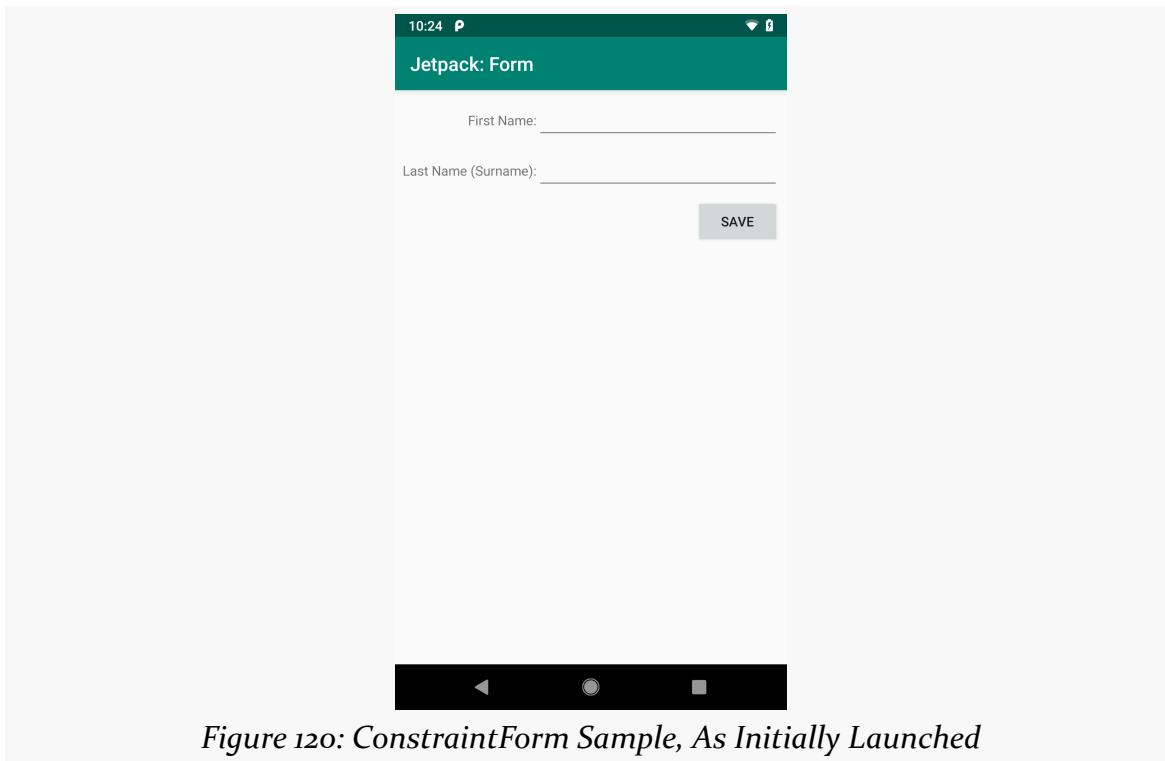


Figure 120: ConstraintForm Sample, As Initially Launched

Fortunately, ConstraintLayout offers some features for helping us get this sort of look.

How We Get There

Our layout is more complex than the earlier example, owing in part to having five children of the ConstraintLayout: two TextView widgets, two EditText widgets, and a Button:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="@dimen/container_padding">

    <androidx.constraintlayout.widget.Barrier
        android:id="@+id/barrier"
```

INTRODUCING CONSTRAINTLAYOUT

```
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginStart="@dimen/margin_barrier"
    android:layout_marginEnd="@dimen/margin_barrier"
    app:barrierDirection="end"
    app:constraint_referenced_ids="labelFirst,labelLast" />

<TextView
    android:id="@+id/labelFirst"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:labelFor="@+id/firstName"
    android:text="@string/label_first"
    app:layout_constraintEnd_toStartOf="@+id/barrier"
    app:layout_constraintBaseline_toBaselineOf="@+id/firstName"
    app:layout_constraintHorizontal_bias="1.0"
    app:layout_constraintStart_toStartOf="parent" />

<EditText
    android:id="@+id/firstName"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:inputType="textPersonName"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toEndOf="@+id/barrier"
    app:layout_constraintTop_toTopOf="parent" />

<TextView
    android:id="@+id/labelLast"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:labelFor="@+id/lastName"
    android:text="@string/label_last"
    app:layout_constraintEnd_toStartOf="@+id/barrier"
    app:layout_constraintBaseline_toBaselineOf="@+id/lastName"
    app:layout_constraintHorizontal_bias="1.0"
    app:layout_constraintStart_toStartOf="parent" />

<EditText
    android:id="@+id/lastName"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:inputType="textPersonName"
    android:layout_marginTop="@dimen/margin_row"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toEndOf="@+id/barrier"
    app:layout_constraintTop_toBottomOf="@+id/firstName" />
```

INTRODUCING CONSTRAINTLAYOUT

```
<Button  
    android:id="@+id/button"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:text="@string/save_caption"  
    android:layout_marginTop="@dimen/margin_row"  
    app:layout_constraintEnd_toEndOf="parent"  
    app:layout_constraintTop_toBottomOf="@+id/lastName" />  
  
</androidx.constraintlayout.widget.ConstraintLayout>
```

(from [ConstraintForm/src/main/res/layout/activity_main.xml](#))

If you look at that XML, there are *six* child elements of the ConstraintLayout — we will explore that sixth one shortly.

Naming the Widgets

Each of the widgets gets a fairly simple android:id:

- `firstName` and `lastName` for the two `EditText` widgets
- `firstLabel` and `lastLabel` for the two `TextView` widgets
- `button` for the `Button`

Since we are not doing anything with these widgets from our Java/Kotlin code, the widget IDs are purely for internal use within this layout resource.

Barrier: You Shall Not Pass

The sixth child of the ConstraintLayout is a Barrier. Barrier is one of a couple of options with ConstraintLayout for creating a virtual anchor point: a place for you to reference in layout anchoring rules that is not itself a widget in the layout.

Specifically, this Barrier has:

- `app:barrierDirection="end"`
- `app:constraint_referenced_ids="labelFirst,labelLast"`

Here, the “barrier direction” indicates where the barrier is established relative to the widgets referenced in the `app:constraint_referenced_ids` attribute. In our case, the end for `app:barrierDirection` means “put this Barrier at the end position of the children that is farthest from the start position”. Both of the `TextView` widgets are set up to start from the start edge of the ConstraintLayout and take up as

INTRODUCING CONSTRAINTLAYOUT

much room as their content requires (`android:layout_width="wrap_content"`). As a result, the Barrier is placed at the end of whichever of those two `TextView` widgets is longest. In the sample app, that is the second label, which uses `Last Name (Surname)` as the caption (courtesy of `@string/label_last`). So, the Barrier is placed where the `labelLast` `TextView` ends.

We then set up the `firstName` and `lastName` widgets to have their `start` position be where the Barrier is, via `app:layout_constraintStart_toEndOf="@+id/barrier"`. This is why both field have the same starting position, and that position is determined by the longest of the labels.



Figure 121: Android Studio Graphical Layout Editor, Showing Form with Barrier

As a result, our four main widgets — the two labels and the two fields — are organized into two columns, where the width of the first column is determined by the width of the labels inside of it.

Your Position Shows Some Bias

Our two labels do not have the same width. The `labelLast` text was set up to be excessively long to illustrate this.

By default, if we just anchored the labels to the `start` edge of the `ConstraintLayout`, their text would wind up on that edge. That is a bit awkward, though, in that our `labelFirst` text would wind up relatively far away from its associated field.

To address this, our layout has each label do two things:

INTRODUCING CONSTRAINTLAYOUT

1. Constrain both the start and end edges, where the end is anchored to the Barrier via `app:layout_constraintEnd_toStartOf="@id/barrier"`
2. Shove the text to the end side via
`app:layout_constraintHorizontal_bias="1.0"`

Bias, in terms of ConstraintLayout, means “if there is extra room, slide the widget in this direction along the axis”. The default bias is 0.5, meaning that the widget is centered in the available space. For the horizontal axis, 0.0 means “slide the widget all the way towards the start side” and 1.0 means “slide the widget all the way towards the end side”.

By having the label end at the Barrier and having its associated field start at the Barrier, we ensure that the label and the field are close together.

Declaring the Rows

In terms of the rows, the fields drive the vertical positioning:

- The top of `firstName` is anchored to the top of the ConstraintLayout via `app:layout_constraintTop_toTopOf="parent"`
- The top of `lastName` is anchored to the bottom of `firstName` via `app:layout_constraintTop_toBottomOf="@id/firstName"`
- The top of `button` is anchored to the bottom of `lastName` via `app:layout_constraintTop_toBottomOf="@id/lastName"`
- The labels have their baselines aligned with their associated fields

In addition:

- Each label has `android:labelFor` set, pointing to its associated field, for the benefit of screen readers and other assistive technologies
- The Button has `app:layout_constraintEnd_toEndOf="parent"` and no start-side anchor, so it will be flush on the end side of the ConstraintLayout

Turning Back to RTL

In order for the “start”/“end” attributes to reverse their positions based on language direction, you need to have `android:supportsRtl="true"` in your `<application>` element in your manifest. Most newly-created projects will have this attribute already set for you by the new-project wizard.

INTRODUCING CONSTRAINTLAYOUT

To see how your app behaves with RTL — without having to learn Arabic or Hebrew, if you are not literate in those languages — you can force Android to use RTL layout rules with any language on Android 4.2+ devices. To do this, go into the Settings app of the device or emulator and choose “Developer options”. In there, scroll down to the “Force RTL layout direction” item. By default, this is turned off, and so layout direction is determined by the user’s chosen language:

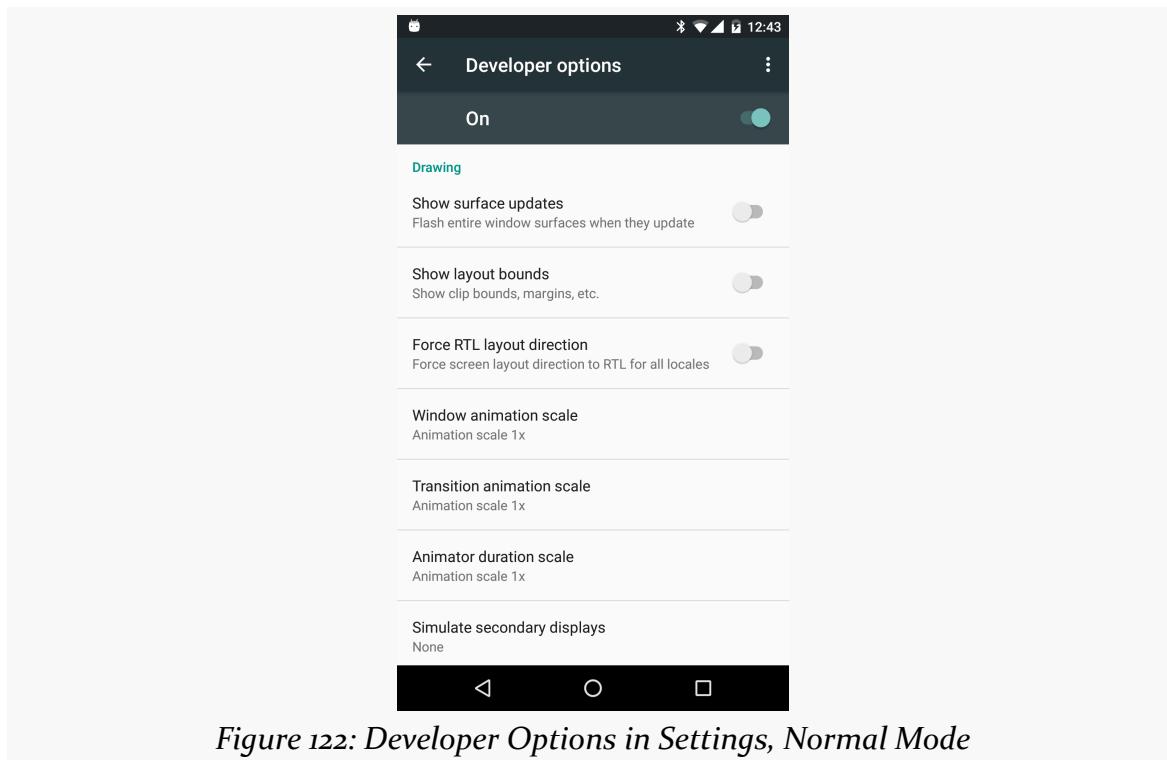


Figure 122: Developer Options in Settings, Normal Mode

INTRODUCING CONSTRAINTLAYOUT

Tapping that switch uses RTL layout rules — with “start” referring to the right and “end” referring to the left — for all languages:

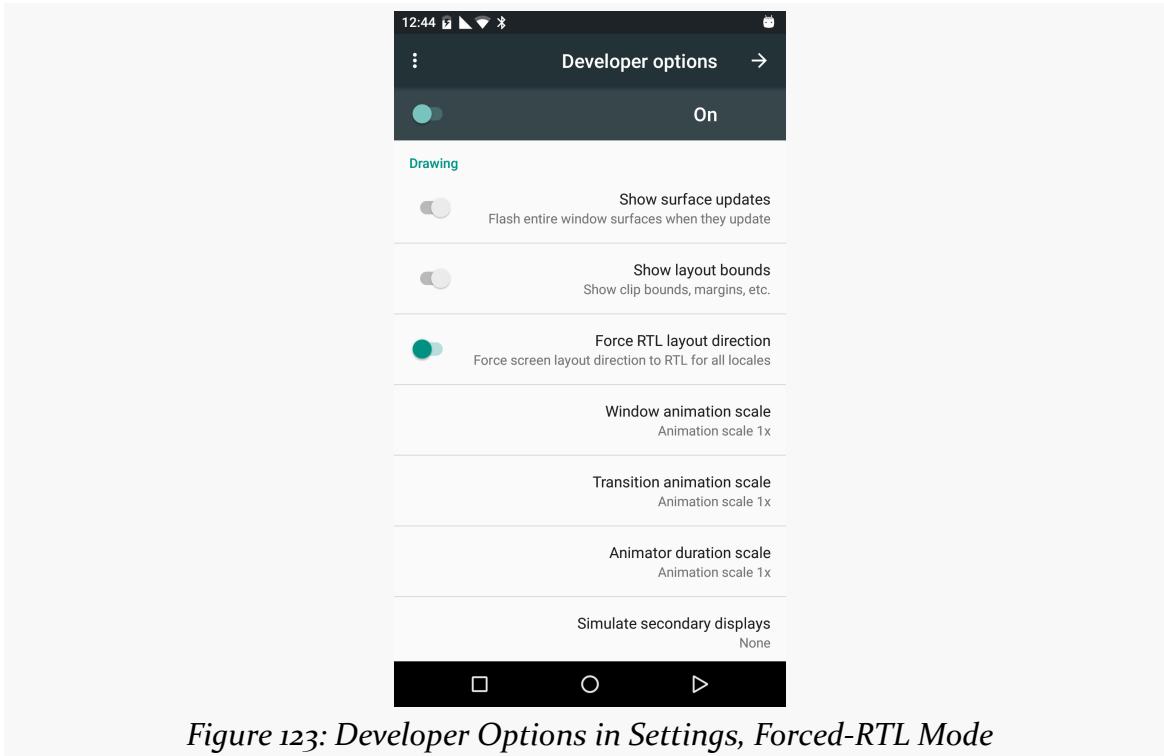


Figure 123: Developer Options in Settings, Forced-RTL Mode

As a reminder, if “Developer options” is not in the list of Settings categories, go into the “About device” category, find the build number item, and tap on it seven times. This will enable “Developer options” back on the main list of Settings categories.

More Fun with ConstraintLayout

There are many more capabilities of ConstraintLayout than what we have covered here. You can:

- Use Guideline to set up anchoring points, a bit like Barrier, but set at arbitrary locations
- Use `app:layout_constraintWidth_percent` and `app:layout_constraintHeight_percent` to size children as a percentage of the ConstraintLayout size along a particular axis
- Use `app:layout_constraintDimensionRatio` to force a child to adhere to a particular aspect ratio

INTRODUCING CONSTRAINTLAYOUT

- Use ConstraintSet to alter the anchoring and sizing rules for children of ConstraintLayout at runtime
- And more!

We will see ConstraintLayout used in many of the samples through the rest of the book.

Notes on the Classic Containers

While ConstraintLayout is the recommended solution for Jetpack-based Android app development, it is not the only option. And since ConstraintLayout is relatively new, there is a lot of existing code that does not use it.

Four container classes were the dominant options starting with Android 1.0 and can still be used today: LinearLayout, RelativeLayout, TableLayout, and FrameLayout.

LinearLayout

LinearLayout represents Android's approach to a box model — widgets or child containers are lined up in a column or row, one after the next. Since many other GUI toolkits use this sort of an approach, a lot of developers used LinearLayout extensively. Plus, it is *very* easy to use: other than an android:orientation attribute to indicate if it should be a row (horizontal) or column (vertical), nothing else is required.

And, for fairly simple scenarios, there is nothing wrong with using LinearLayout.

However, LinearLayout was designed around simple usage and simple implementation, not performance. Having lots of nested LinearLayout containers can slow things down. And for a complex form, you would need lots of LinearLayout containers. Even something as simple as ConstraintForm would take three LinearLayout instances:

- Two horizontal ones for the rows containing the labels and fields
- A vertical one for holding those rows plus the Button

RelativeLayout

On the surface, RelativeLayout looks a lot like ConstraintLayout. RelativeLayout, as the name suggests, lays out widgets based upon their relationship to other

INTRODUCING CONSTRAINTLAYOUT

widgets in the container and the parent container. You can place Widget X below and to the left of Widget Y, or have Widget Z's bottom edge align with the bottom of the container, and so on. And it does so via special attributes on the children, saying what they are anchored to.

All of that sounds like ConstraintLayout.

However, RelativeLayout was not created with performance or drag-and-drop GUI builders in mind. ConstraintLayout has a superset of RelativeLayout features, and RelativeLayout has its own set of attributes similar to, but distinct from, those in ConstraintLayout. There is little reason to use RelativeLayout today.

TableLayout

If you like HTML tables, you will like Android's TableLayout. It allows you to position your widgets in a grid to your specifications. You control the number of rows and columns, which columns might shrink or stretch to accommodate their contents, and so on.

TableLayout works in conjunction with TableRow. TableLayout controls the overall behavior of the container, with the widgets themselves poured into one or more TableRow containers, one per row in the grid. Using a TableLayout with TableRow works a lot like using an HTML <table> with <tr> elements.

When Android started out, HTML tables were very popular, not only for tabular data but for general page layout. CSS had only caught on in recent years, so lots of Web developers were used to using HTML tables to try to position things on the screen in the desired locations. Having a TableLayout that mimicked HTML tables was a logical move, to help ease the transition for those early Web designers.

With Barrier, you can set up a ConstraintLayout that can handle table structures, with rows and columns, where the columns can vary in width based on contents. However, it is likely that there will be some scenarios that would be very difficult to implement with ConstraintLayout that TableLayout could handle easily. Overall, though, ConstraintLayout is much more powerful than is TableLayout. Plus, TableLayout suffers from the same performance problems of LinearLayout, with lots of nested containers.

So, in general, if you see a grid sort of structure, try using a ConstraintLayout, and consider falling back to TableLayout only if needed.

FrameLayout

Android has a `FrameLayout` class. Like `ConstraintLayout`, `LinearLayout`, `RelativeLayout`, and `TableLayout`, `FrameLayout` exists to size and position its children. However, `FrameLayout` has a very simple pair of layout rules:

1. All children go in the upper-start corner (e.g., upper-left for LTR languages), unless `android:gravity` indicates to position the children elsewhere
2. Children can overlap on the Z axis (which `ConstraintLayout` and `RelativeLayout` also support)

The result is that all the widgets are stacked one on top of another.

This may seem useless.

Primarily, `FrameLayout` is used in places where we want to reserve space for something, but we do not know what the “something” is at compile time. The decision of what the “something” is will be made at runtime, where we will use Java/Kotlin code to put something in the `FrameLayout`. We will see this pattern used with fragments, [later in the book](#).

Occasionally, `FrameLayout` is literally used for “framing”, where we want some sort of a border around a child. In this case, the background of the `FrameLayout` (e.g., `android:background`) defines what the frame should look like.

Licensed solely for use by Patrocinio Rodriguez

Integrating Common Form Widgets

`TextView`, `Button`, and `EditText` form the foundation of many user interfaces in Android, and their analogues form the foundation of other GUI toolkits.

Overall, the Android SDK has a reasonable range of widgets to choose from, though not everything is covered. In this chapter, we will review a number of other commonly-used widgets in the Android SDK and explore some other capabilities of Android's UI system.

All of the code in this chapter comes from the `FormWidgets` module of the [Java](#) and [Kotlin](#) sample projects.

ImageView and ImageButton

Android has two widgets to help you embed images in your activities: `ImageView` and `ImageButton`. As the names suggest, they are image-based analogues to `TextView` and `Button`, respectively.

Each widget takes an `android:src` attribute (in an layout resource) to specify what picture to use. These usually reference a drawable resource (e.g., `@drawable/icon`) or sometimes a mipmap resource (e.g., `@mipmap/ic_launcher`).

`ImageButton`, a subclass of `ImageView`, mixes in the standard `Button` behaviors. While both `ImageView` and `ImageButton` can call your code when they are clicked, `ImageButton` has built-in logic to visually respond to the click, the way that `Button` does.

Our sample app has an `ImageView` named `icon` and an `ImageButton` named `button`:

INTEGRATING COMMON FORM WIDGETS

```
<ImageView  
    android:id="@+id/icon"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:layout_marginTop="@dimen/margin_row"  
    android:contentDescription="@string/icon_caption"  
    android:src="@mipmap/ic_launcher"  
    app:layout_constraintEnd_toEndOf="parent"  
    app:layout_constraintStart_toStartOf="parent"  
    app:layout_constraintTop_toBottomOf="@+id/log" />  
  
<ImageButton  
    android:id="@+id/button"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:layout_marginTop="@dimen/margin_row"  
    android:contentDescription="@string/button_caption"  
    android:src="@mipmap/ic_launcher"  
    app:layout_constraintEnd_toEndOf="parent"  
    app:layout_constraintStart_toStartOf="parent"  
    app:layout_constraintTop_toBottomOf="@+id/icon" />
```

(from [FormWidgets/src/main/res/layout/activity_main.xml](#))

INTEGRATING COMMON FORM WIDGETS

These then show up in our overall activity's output, which stacks a bunch of widgets in a ConstraintLayout:

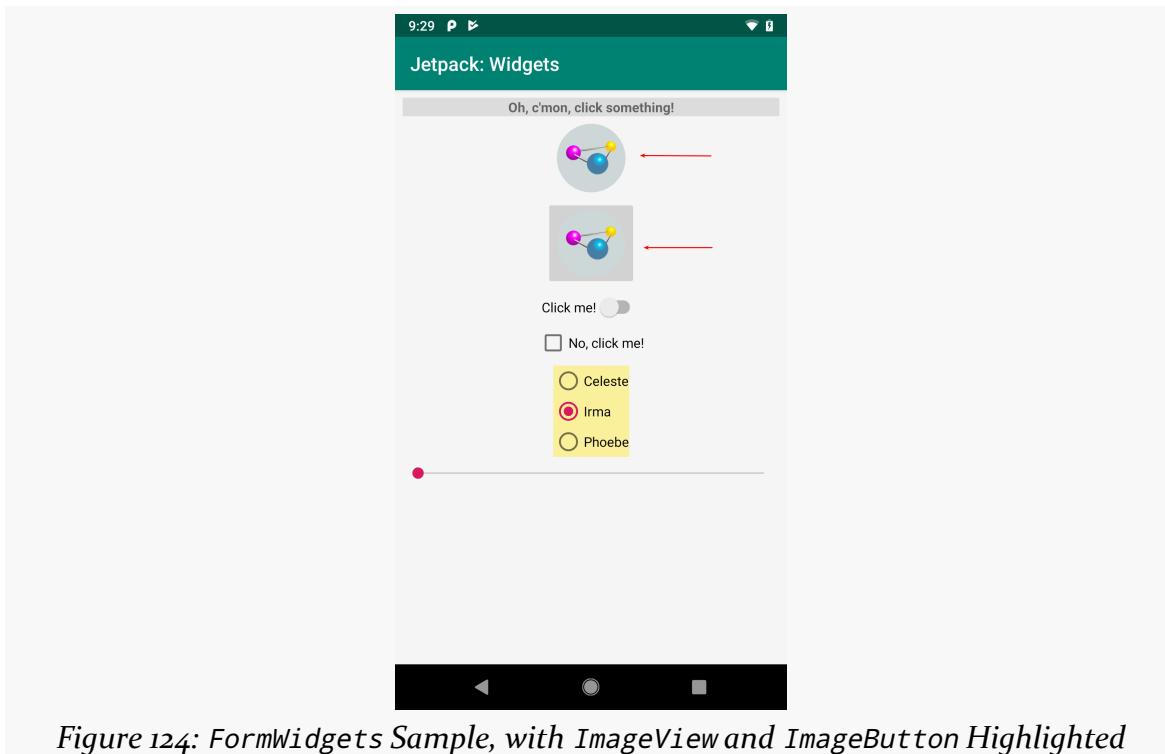


Figure 124: FormWidgets Sample, with ImageView and ImageButton Highlighted

INTEGRATING COMMON FORM WIDGETS

Android Studio Graphical Layout Editor

The ImageView widget can be found in the “Common” portion of the Palette in the Android Studio graphical layout editor:

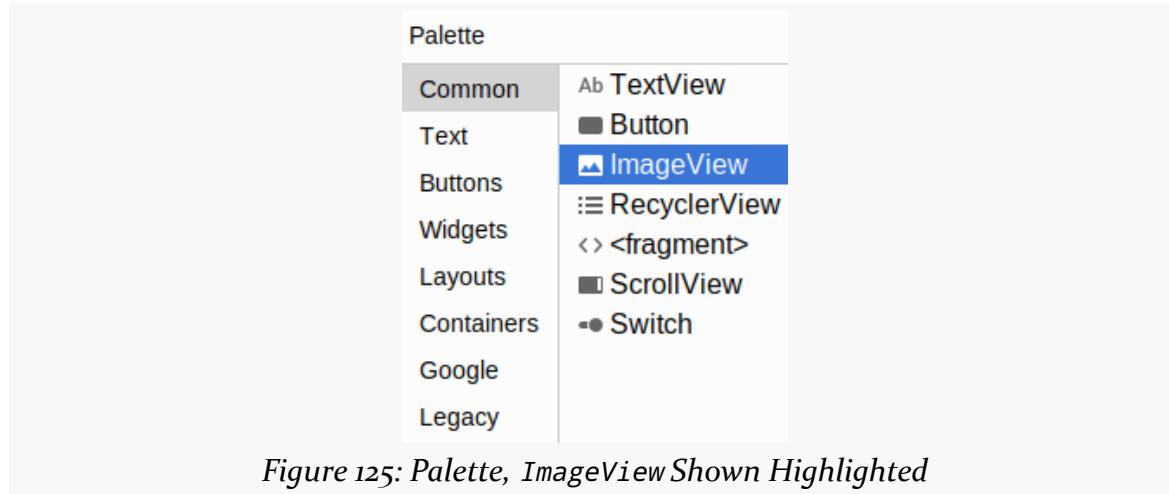


Figure 125: Palette, ImageView Shown Highlighted

ImageButton shows up in the “Buttons” portion:

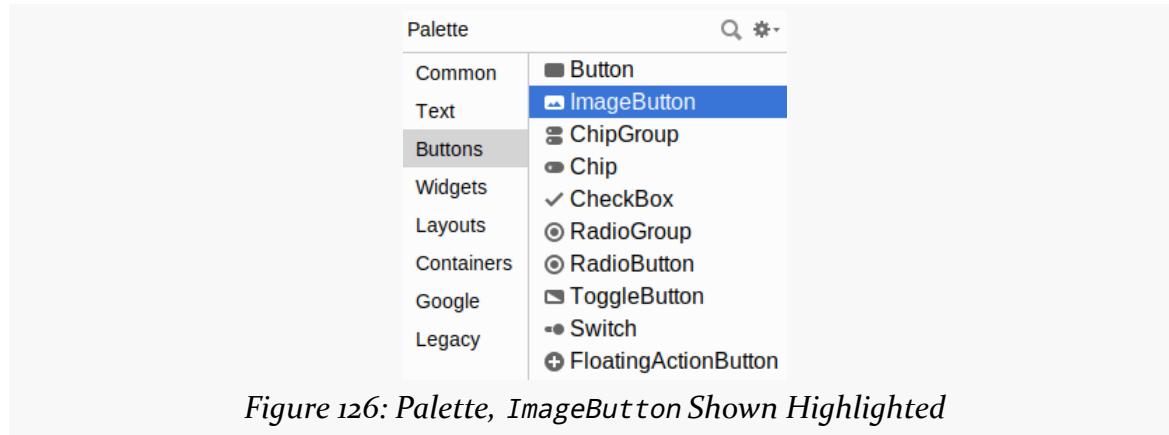


Figure 126: Palette, ImageButton Shown Highlighted

INTEGRATING COMMON FORM WIDGETS

When you drag one of these into the preview or blueprint, you are immediately greeted by a dialog to choose a drawable resource or color to use for the image:

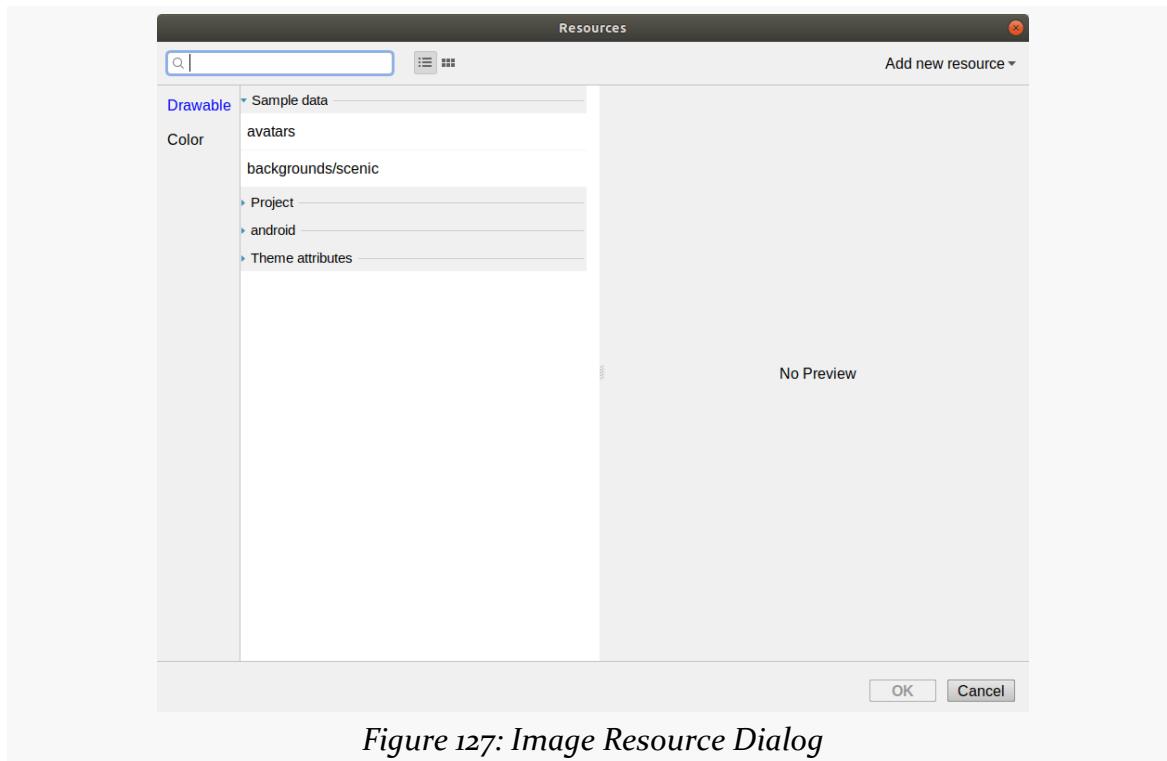


Figure 127: Image Resource Dialog

Unfortunately, you have no choice but to choose one of these, as due to [a bug](#), if you click Cancel to exit the dialog, it also abandons the entire drag-and-drop operation.

You can drag these into a layout file, then use the Attributes pane to set their attributes. Like all widgets, you will have an “id” option to set the android:id value for the widget. Two others of importance, though, are more unique to ImageView and ImageButton:

- `src` allows you to choose a drawable resource to use as the image to be displayed, which will be filled in by whatever you chose in the resource dialog
- `contentDescription` provides the text that will be used to describe the image to users that have accessibility services enabled (e.g., TalkBack), such as visually impaired users

Reacting to Events

You can call `setOnClickListener()` on an `ImageView` or `ImageButton` to find out when the user clicks the widget and do something:

```
findViewById(R.id.icon).setOnClickListener(v -> log(R.string.icon_clicked));
findViewById(R.id.button).setOnClickListener(
    v -> log(R.string.button_clicked));
```

(from [FormWidgets/src/main/java/com/commonsware/jetpack/samplerj/formwidgets/MainActivity.java](#))

```
icon.setOnClickListener { log(R.string.icon_clicked) }
button.setOnClickListener { log(R.string.button_clicked) }
```

(from [FormWidgets/src/main/java/com/commonsware/jetpack/samplerj/formwidgets/MainActivity.kt](#))

We will cover the `log()` function that both of those are using [a bit later in this chapter](#).

Compound Buttons

The Android SDK has a `CompoundButton` class that represents a widget that can be clicked to toggle a “checked” state. `CompoundButton` itself is an abstract class, but there are a few subclasses of `CompoundButton` of interest to Android app developers.

Switch

For a simple widget to show a checked or unchecked state, the modern approach is to use a `Switch`.

`Switch` extends `CompoundButton`, which in turn inherits from `TextView`. As a result, you can use `TextView` properties like `android:textColor` to format the widget, in addition to `android:text` to set the caption that should appear adjacent to the actual “switch” UI.

Within your Java/Kotlin code, you can call:

1. `isChecked()` to determine if the checkbox has been checked
2. `setChecked()` to force the checkbox into a checked or unchecked state
3. `toggle()` to toggle the checkbox as if the user clicked upon it, inverting whatever its current state is

Also, you can register a listener object (in this case, an instance of

INTEGRATING COMMON FORM WIDGETS

OnCheckedChangeListener) to be notified when the state of the checkbox changes.

Our sample app has an Switch named swytch:

```
<Switch  
    android:id="@+id/swytch"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:layout_marginTop="@dimen/margin_row"  
    android:text="@string/switch_caption"  
    app:layout_constraintEnd_toEndOf="parent"  
    app:layout_constraintStart_toStartOf="parent"  
    app:layout_constraintTop_toBottomOf="@+id/button" />
```

(from [FormWidgets/src/main/res/layout/activity_main.xml](#))

INTEGRATING COMMON FORM WIDGETS

This appears beneath the ImageButton:

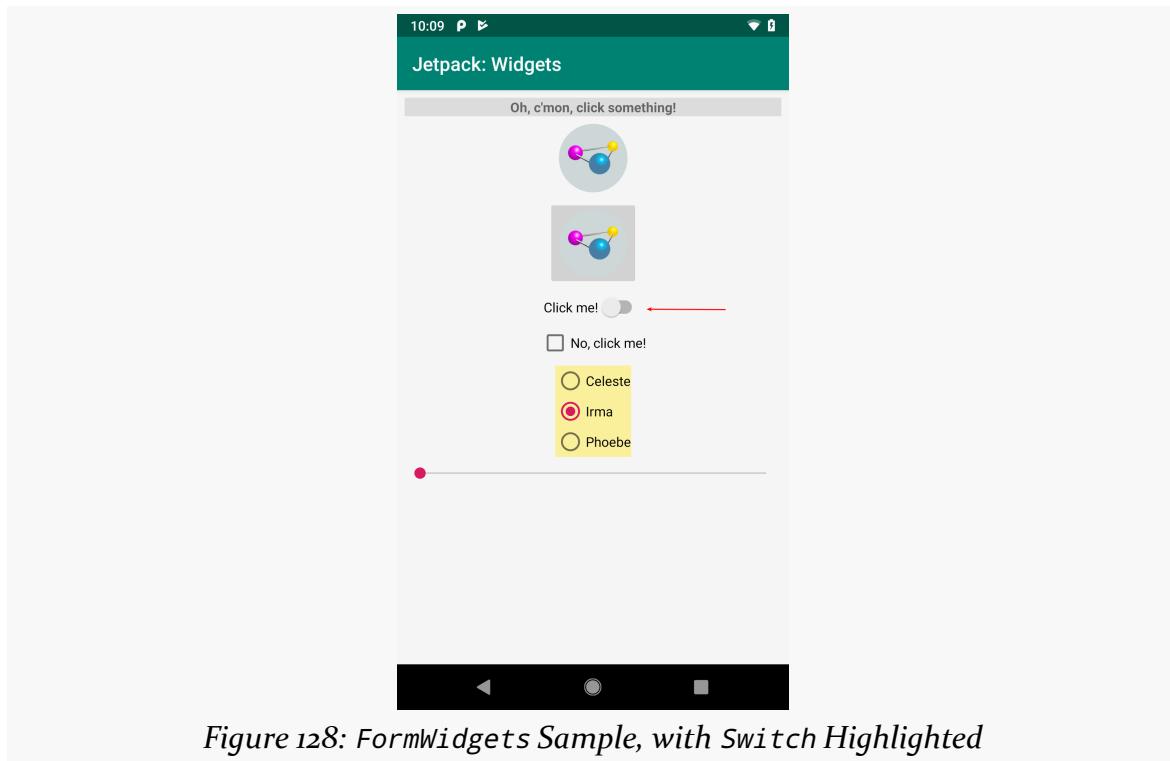


Figure 128: FormWidgets Sample, with Switch Highlighted

INTEGRATING COMMON FORM WIDGETS

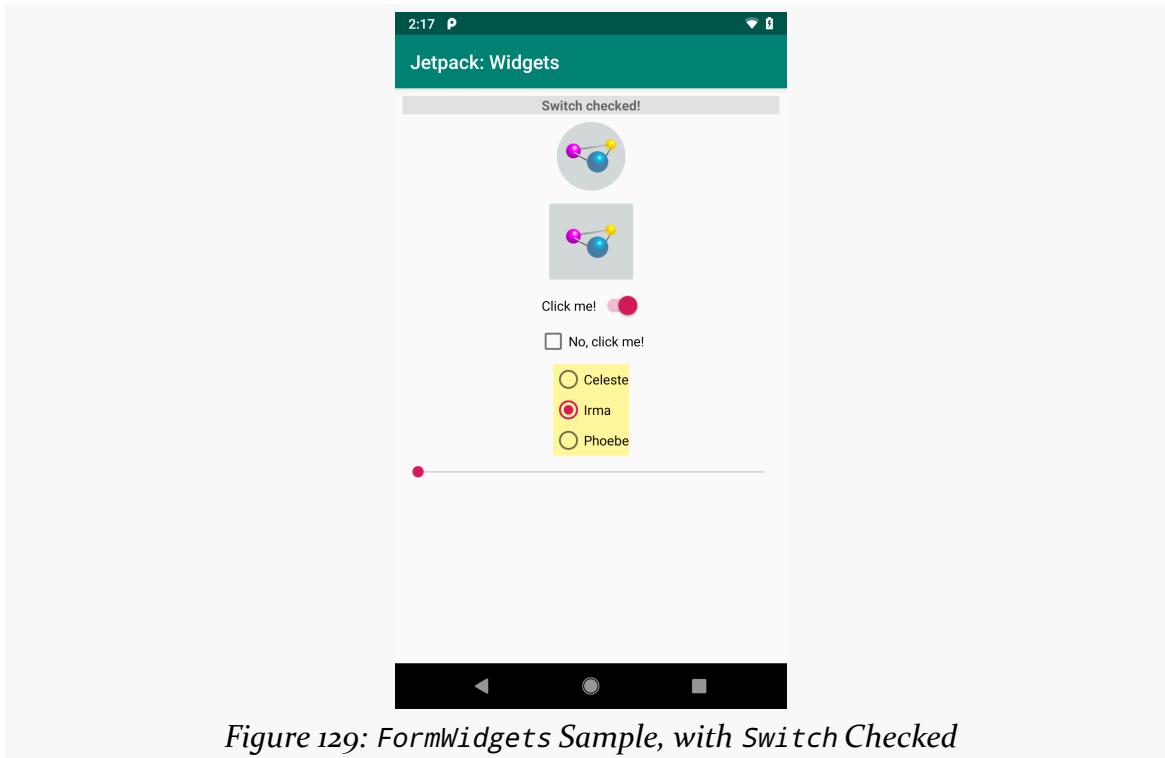


Figure 129: FormWidgets Sample, with Switch Checked

Android Studio Graphical Layout Editor

The Switch widget can be found in the “Buttons” portion of the Palette in the Android Studio Graphical Layout editor:

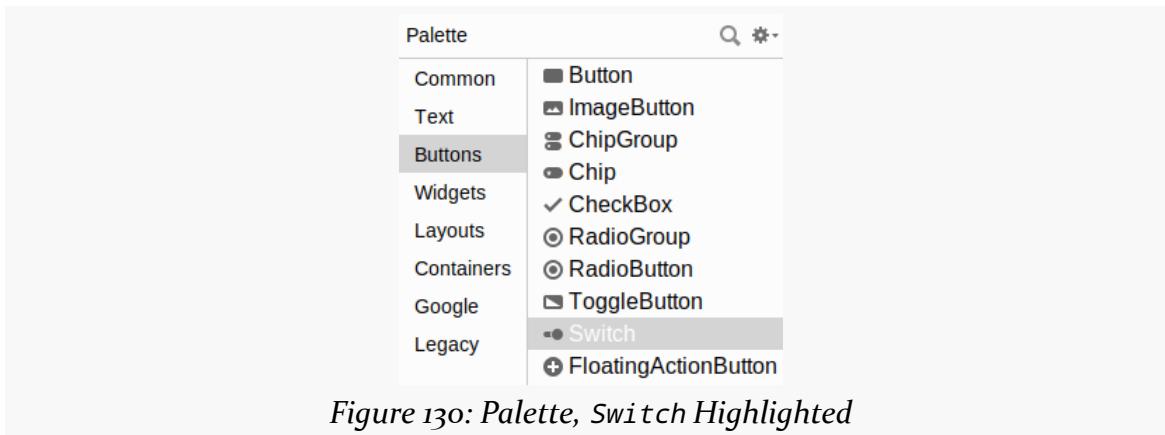


Figure 130: Palette, Switch Highlighted

You can drag one into the layout and configure it as desired using the Attributes pane. Mostly the attributes match those of TextView and Button.

Reacting to Events

The primary event listener for any form of CompoundButton is for changes in the checked state. You can register a View.OnCheckedChangeListener via `setOnCheckedChangeListener()` from Java:

```
((CompoundButton)findViewById(R.id.swtch))
    .setOnCheckedChangeListener((v, isChecked) ->
        log(isChecked ? R.string.switch_checked : R.string.switch_unchecked));
```

(from [FormWidgets/src/main/java/com/commonsware/jetpack/samplerj/formwidgets/MainActivity.java](#))

...or Kotlin:

```
swtch.setOnCheckedChangeListener { _, isChecked ->
    log(if (isChecked) R.string.switch_checked else R.string.switch_unchecked)
}
```

(from [FormWidgets/src/main/java/com/commonsware/jetpack/samplerj/formwidgets/MainActivity.kt](#))

In this case, our lambda expressions are converted into instances of `View.OnCheckedChangeListener`, with the lambda expression bodies forming the implementation of the `onCheckedChanged()` function. That function gets passed two parameters:

- The `CompoundButton` whose checked state changed, and
- A Boolean reflecting whether the new state is checked or unchecked

In our case, we just use that Boolean to choose which of two string resources to pass to `log()`, using the ternary operator in Java and an `if` expression in Kotlin.

Hey, You Have a Typo in `android:id`!

You may have noticed that the `android:id` value for the Switch is `swtch`. Considering that the ImageButton is button, you might expect that the Switch would be named `switch`.

However, this does not work. You cannot create a widget whose ID matches a Java keyword. `switch` is a Java keyword, so we cannot use `switch` as a widget ID. Similarly, we cannot have widgets with an ID of `if`, `else`, or `return`.

INTEGRATING COMMON FORM WIDGETS

CheckBox

If you would prefer something that looks more like a classic checkbox, the Android SDK has CheckBox. It too is a subclass of CompoundButton and can be used interchangeably with Switch.

Our sample app has a CheckBox named checkbox:

```
<CheckBox  
    android:id="@+id/checkbox"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:layout_marginTop="@dimen/margin_row"  
    android:text="@string/checkbox_caption"  
    app:layout_constraintEnd_toEndOf="parent"  
    app:layout_constraintStart_toStartOf="parent"  
    app:layout_constraintTop_toBottomOf="@+id/swytch" />
```

(from [FormWidgets/src/main/res/layout/activity_main.xml](#))

INTEGRATING COMMON FORM WIDGETS

This appears beneath the Switch:

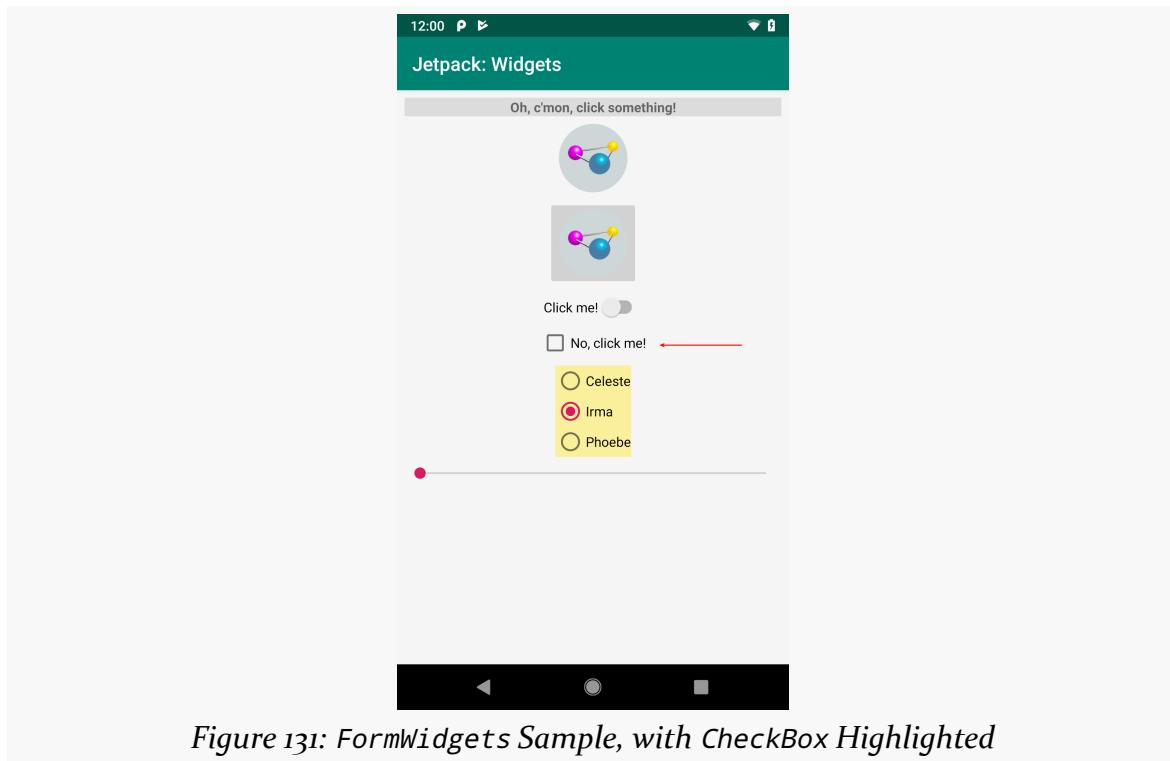


Figure 131: FormWidgets Sample, with CheckBox Highlighted

INTEGRATING COMMON FORM WIDGETS

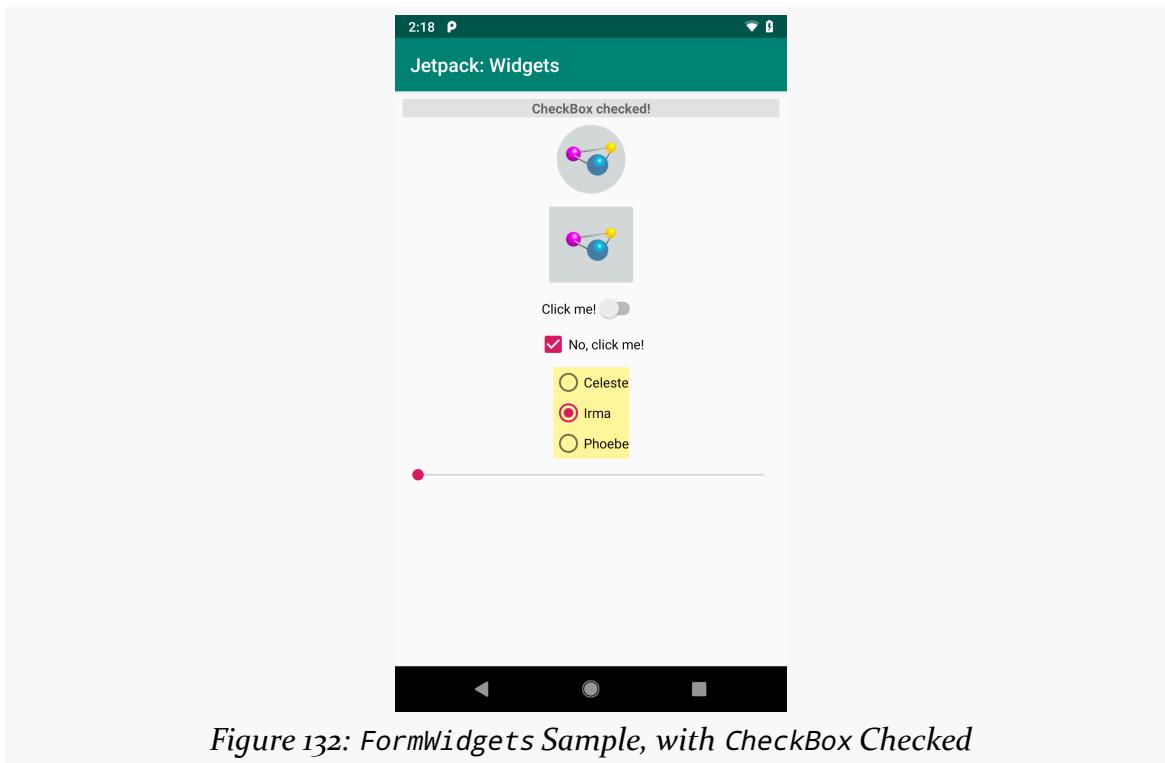


Figure 132: FormWidgets Sample, with CheckBox Checked

Android Studio Graphical Layout Editor

The CheckBox widget can be found in the “Buttons” portion of the Palette in the Android Studio Graphical Layout editor:

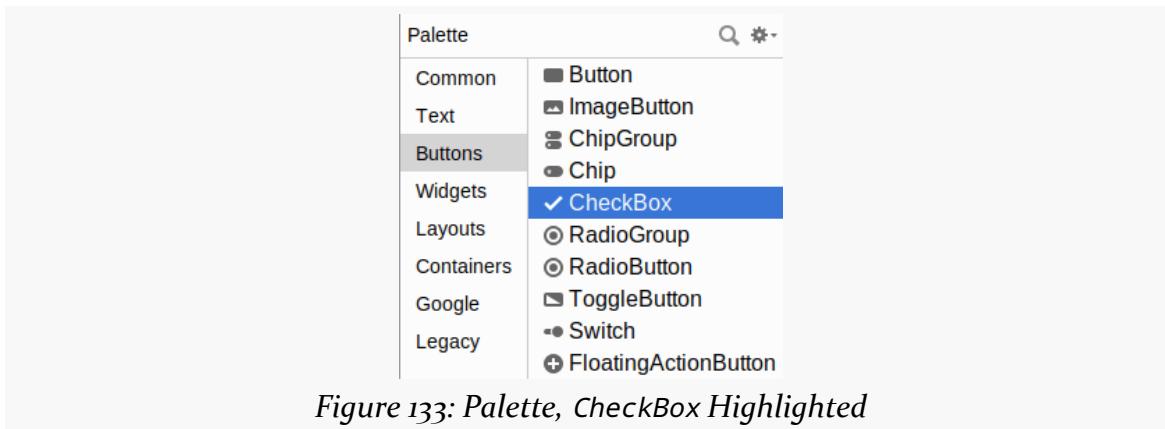


Figure 133: Palette, CheckBox Highlighted

You can drag one into the layout and configure it as desired using the Attributes pane. As CheckBox inherits from TextView, most of the settings are the same as

INTEGRATING COMMON FORM WIDGETS

those you would find on a regular `TextView`.

Reacting to Events

As with `Switch`, you can register a `View.OnCheckedChangeListener` on a `CheckBox` using `setOnCheckedChangeListener()`:

```
((CompoundButton) findViewById(R.id.checkbox))
.setOnCheckedChangeListener((v, isChecked) ->
    log(
        isChecked ? R.string.checkbox_checked : R.string.checkbox_unchecked));

```

(from [FormWidgets/src/main/java/com/commonsware/jetpack/samplerj/formwidgets/MainActivity.java](#))

```
checkbox.setOnCheckedChangeListener { _, isChecked ->
    log(if (isChecked) R.string.checkbox_checked else R.string.checkbox_unchecked)
}
```

(from [FormWidgets/src/main/java/com/commonsware/jetpack/samplerj/formwidgets/MainActivity.kt](#))

RadioButton and RadioGroup

As with other implementations of radio buttons in other toolkits, Android's radio buttons are two-state, like checkboxes, but can be grouped such that only one radio button in the group can be checked at any time. `RadioButton` is another form of `CompoundButton` and, on its own, works like `Switch` and `CheckBox`.

Most times, you will want to put your `RadioButton` widgets inside of a `RadioGroup`. The `RadioGroup` is a `LinearLayout` that indicates a set of radio buttons whose state is tied, meaning only one button out of the group can be selected at any time. If you assign an `android:id` to your `RadioGroup` in your layout resource, you can access the group from your Java/Kotlin code and invoke:

1. `check()` to check a specific radio button via its ID (e.g.,
`group.check(R.id.radio1)`)
2. `clearCheck()` to clear all radio buttons, so none in the group are checked
3. `getCheckedRadioButtonId()` to get the ID of the currently-checked radio button (or `-1` if none are checked)

Note that the radio button group is initially set to be completely unchecked at the outset. To preset one of the radio buttons to be checked, use either `setChecked()` on the `RadioButton` or `check()` on the `RadioGroup`. Alternatively, you can use the `android:checked` attribute on one of the `RadioButton` widgets in the layout file.

INTEGRATING COMMON FORM WIDGETS

Our sample app has a RadioGroup named `radioGroup`, containing three RadioButton widgets:

```
<RadioGroup
    android:id="@+id/radioGroup"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="@dimen/margin_row"
    android:background="@color/radiogroup_background"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/checkbox">

    <RadioButton
        android:id="@+id radioButton1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/radioButton1_caption" />

    <RadioButton
        android:id="@+id radioButton2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:checked="true"
        android:text="@string/radioButton2_caption" />

    <RadioButton
        android:id="@+id radioButton3"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/radioButton3_caption" />
</RadioGroup>
```

(from [FormWidgets/src/main/res/layout/activity_main.xml](#))

INTEGRATING COMMON FORM WIDGETS

They appear beneath the CheckBox:

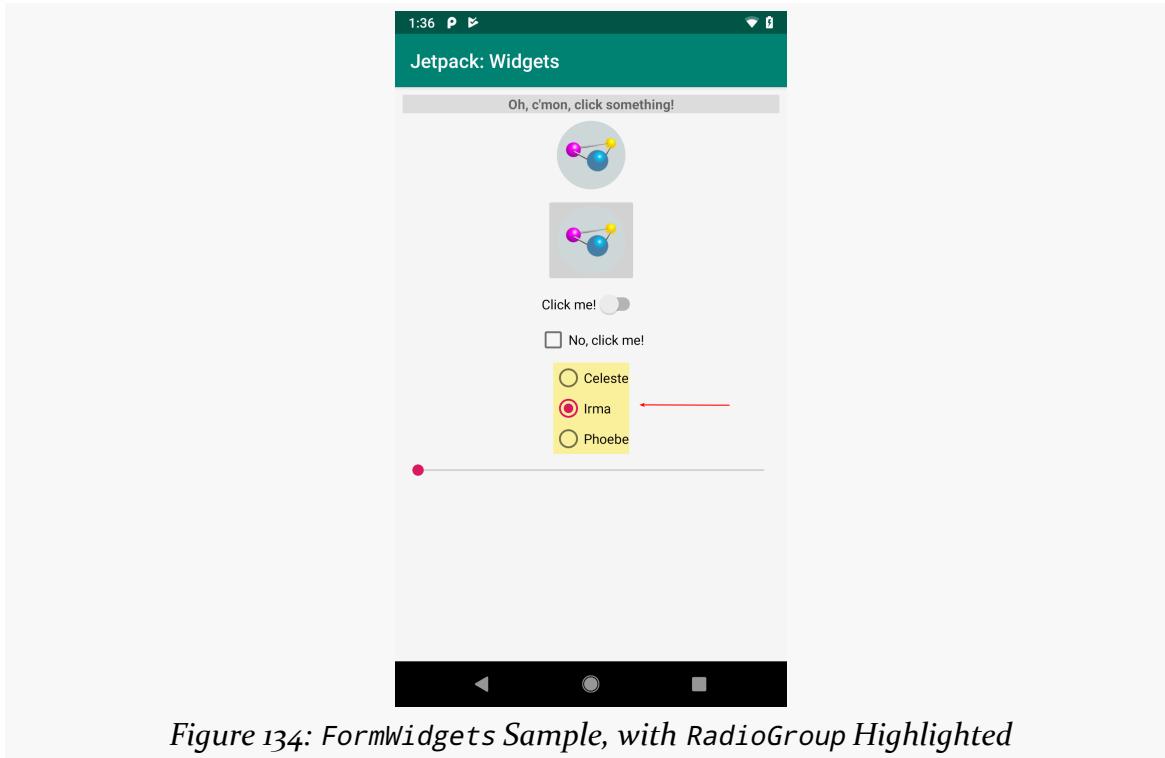


Figure 134: FormWidgets Sample, with RadioGroup Highlighted

The RadioGroup is given a pale yellow background via `android:background="@color/radiogroup_background"` so you can better see its boundaries. The three RadioButton widgets differ in ID, caption, and whether they are checked (the middle one has `android:checked="true"` to pre-select it).

RadioGroup inherits from LinearLayout, which lays its children out one after the next. By default, that is in a column, but you can use `android:orientation="horizontal"` to make them form a row.

INTEGRATING COMMON FORM WIDGETS

Android Studio Graphical Layout Editor

The RadioGroup container and RadioButton widget can be found in the “Buttons” portion of the Palette in the Android Studio Graphical Layout editor:

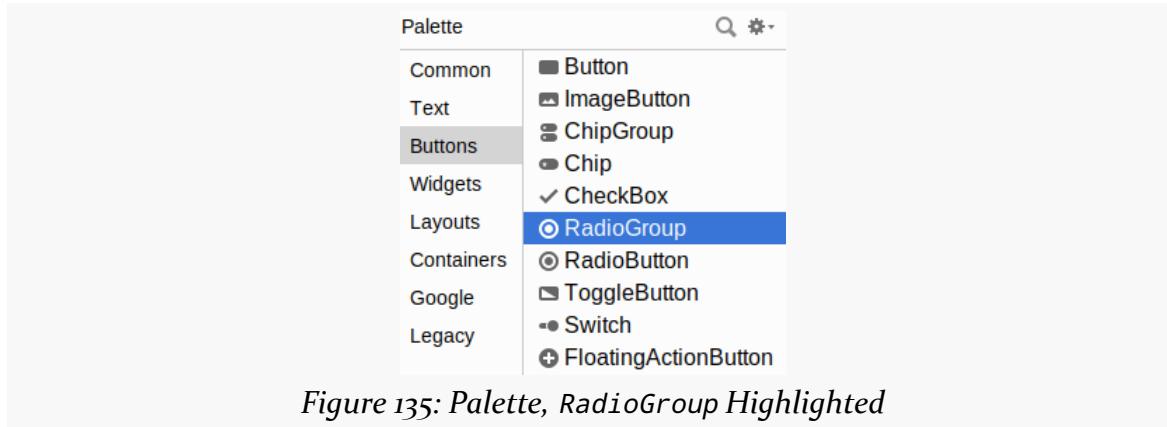


Figure 135: Palette, RadioGroup Highlighted

Dragging a RadioGroup into the preview gives you a box into which you can drag other widgets, such as the RadioButton.

Reacting to Events

It is possible to register the same sort of `View.OnCheckedChangeListener` objects on RadioButton, as it too is a CompoundButton. More often, though, it is simpler to react to events on the RadioGroup, as you only need to register one listener, rather than one per RadioButton.

RadioGroup also has a `setOnCheckedChangeListener()` function. However, it takes a `RadioGroup.OnCheckedChangeListener` implementation, which is slightly different from `CompoundButton.OnCheckedChangeListener`. Your `onCheckedChanged()` function is passed:

- The RadioGroup that the user interacted with, and
- The widget ID of the RadioButton that changed

You are *not* given a Boolean for the new state of that RadioButton, though. That is because you are always called for the newly-checked RadioButton, and so it is always checked. Your `onCheckedChanged()` function is not called for the RadioButton that may have been unchecked as a result of the user checking another RadioButton.

INTEGRATING COMMON FORM WIDGETS

In the SamplerJ edition of `MainActivity`, we use a Java method reference to have an `onRadioGroupChange()` function on `MainActivity` be called whenever the user checks a `RadioButton` in the `RadioGroup`:

```
((RadioGroup)findViewById(R.id.radioGroup))  
    .setOnCheckedChangeListener(this::onRadioGroupChange);
```

(from [FormWidgets/src/main/java/com/commonsware/jetpack/samplerj/formwidgets/MainActivity.java](#))

There, we examine the ID of the newly-checked `RadioButton` and `log()` a different string resource for each:

```
private void onRadioGroupChange(RadioGroup group, int checkedId) {  
    @StringRes int msg;  
  
    switch (checkedId) {  
        case R.id.radioButton1:  
            msg = R.string.radioButton1_checked;  
            break;  
  
        case R.id.radioButton2:  
            msg = R.string.radioButton2_checked;  
            break;  
  
        default:  
            msg = R.string.radioButton3_checked;  
            break;  
    }  
  
    log(msg);  
}
```

(from [FormWidgets/src/main/java/com/commonsware/jetpack/samplerj/formwidgets/MainActivity.java](#))

The Kotlin code in the Sampler edition of `MainActivity` uses a `when` expression to `log()` the desired string resource:

```
radioGroup.setOnCheckedChangeListener { _, checkedId ->  
    log(  
        when (checkedId) {  
            R.id.radioButton1 -> R.string.radioButton1_checked  
            R.id.radioButton2 -> R.string.radioButton2_checked  
            else -> R.string.radioButton3_checked  
        }  
    )  
}
```

INTEGRATING COMMON FORM WIDGETS

(from [FormWidgets/src/main/java/com/commonsware/jetpack/sampler/formwidgets/MainActivity.kt](#))

SeekBar

Sometimes, you want the user to pick a number along some range, such as a percentage from 0% to 100%. You could use an `EditText` and then have data validation to handle illegal entries (e.g., numbers outside of your desired range).

Or, you can use a `SeekBar`.

`SeekBar` allows the user to slide a “thumb” along a bar, where different thumb positions represent different values along a range from 0 to a maximum value that you specify (the default maximum is 100).

The bottom of our layout is a `SeekBar`:

```
<SeekBar  
    android:id="@+id/seekbar"  
    android:layout_width="0dp"  
    android:layout_height="match_parent"  
    android:layout_marginTop="@dimen/margin_row"  
    app:layout_constraintEnd_toEndOf="parent"  
    app:layout_constraintStart_toStartOf="parent"  
    app:layout_constraintTop_toBottomOf="@id/radioGroup" />
```

INTEGRATING COMMON FORM WIDGETS

(from [FormWidgets/src/main/res/layout/activity_main.xml](#))

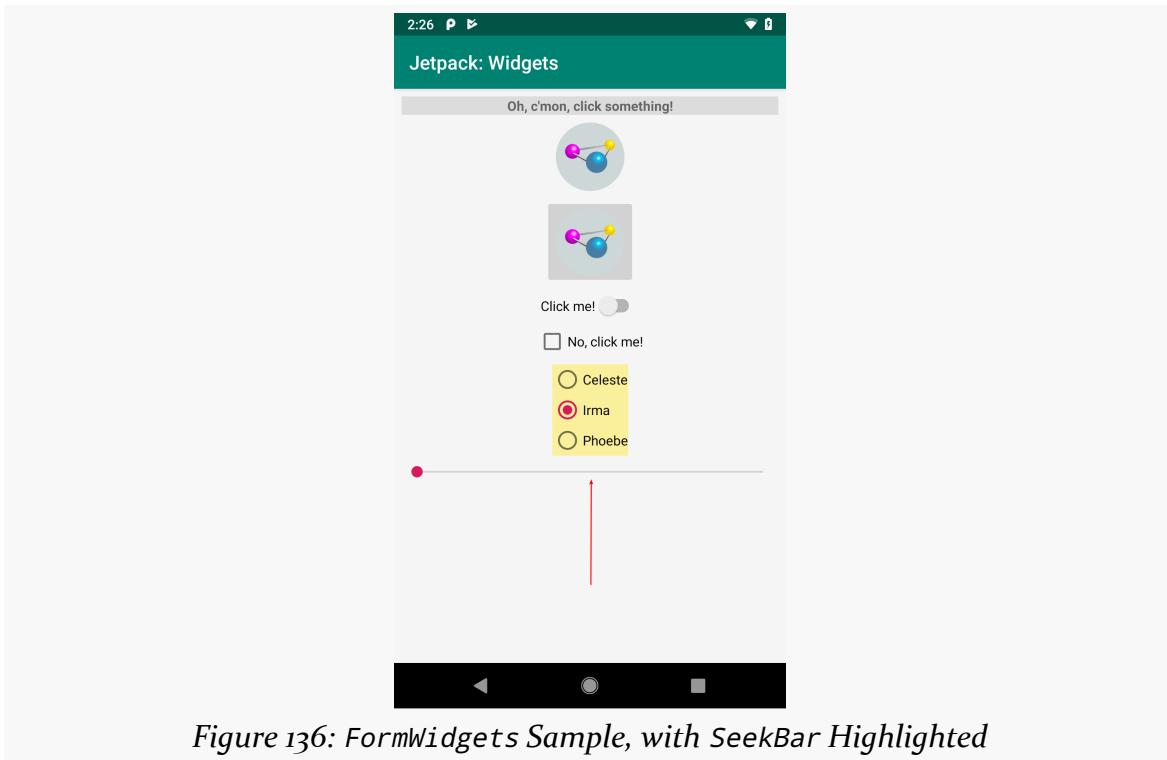


Figure 136: FormWidgets Sample, with SeekBar Highlighted

SeekBar does not inherit from TextView, so it has no caption. Most likely, you will want to use an adjacent TextView to label the SeekBar, so that the user has an idea of what this value represents.

Android Studio Graphical Layout Editor

The SeekBar widget can be found in the “Widgets” portion of the Palette in the Android Studio Graphical Layout editor:

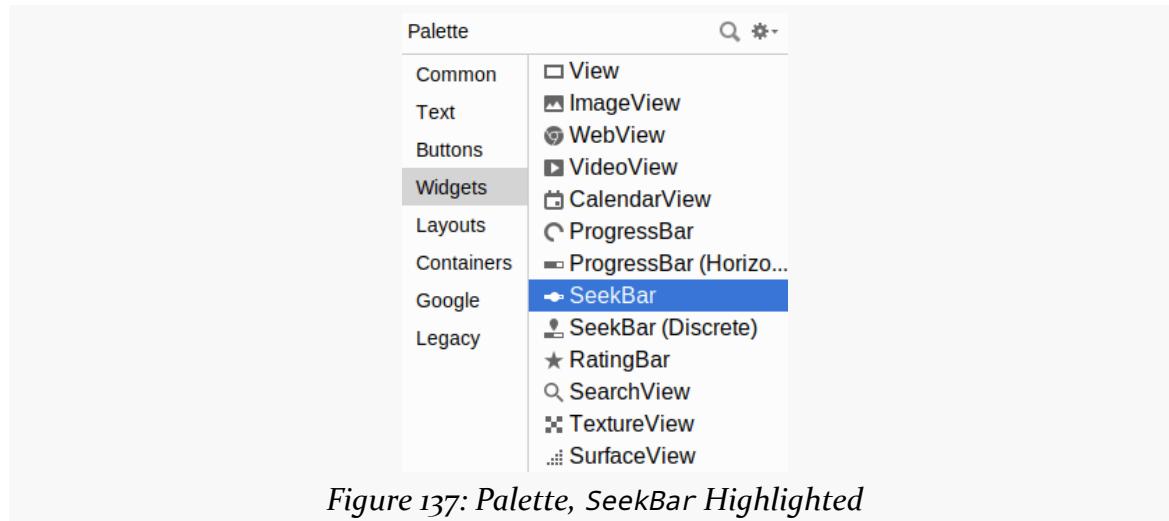


Figure 137: Palette, SeekBar Highlighted

You will notice that there is also an entry for “SeekBar (Discrete)”, where a “discrete” SeekBar shows tick marks for the values in the range. This can be useful if the range is relatively short (e.g., 0-5). To create a discrete SeekBar, you need to supply a drawable resource that provides the “tick mark”, showing the user where the thumb will snap to. If you drag a “SeekBar (Discrete)” into your layout resource, you get a SeekBar with an attribute of `style="@style/Widget.AppCompat.SeekBar.Discrete"`. This sets a particular “style” on the SeekBar, and this style sets the tick mark drawable. We will explore styles in much greater detail [later in the book](#).

Reacting to Events

SeekBar has `setOnSeekBarChangeListener()`, which tells you about changes in the SeekBar value. However, it also requires you to override a couple of other methods, and so it is a bit more complicated to use than are the other listeners that we have seen in this chapter. We will examine the `SeekBar.OnSeekBarChangeListener` more [later in this chapter](#) and see how it works.

ScrollView: Making It All Fit

Android devices come in all sizes. Some have very large screens, while others have very small screens. The problem with very small screens is that your forms do not always fit the available space, particularly in landscape mode:

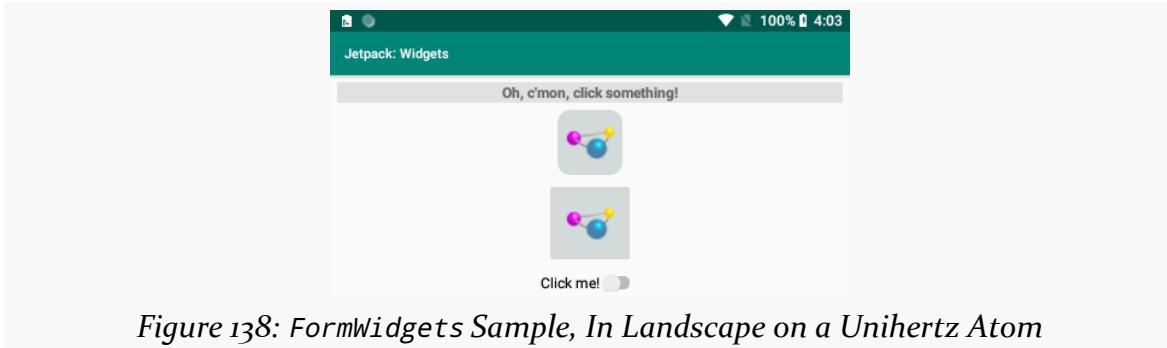


Figure 138: FormWidgets Sample, In Landscape on a Unihertz Atom

That is why the entire ConstraintLayout is wrapped in a ScrollView. ScrollView provides a vertically-scrolling area for your form. The user can swipe up and down to pan around the form and see all of it.

ScrollView itself is a very simple container class, holding exactly one child (in this case, the ConstraintLayout), and making it scrollable. So, our entire activity_main layout is:

```
<?xml version="1.0" encoding="utf-8"?>
<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="@dimen/container_padding">

    <androidx.constraintlayout.widget.ConstraintLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content">

        <TextView
            android:id="@+id/log"
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:background="@color/log_background"
            android:gravity="center_horizontal"
            android:text="@string/log_default"
            android:textStyle="bold" />
    
```

INTEGRATING COMMON FORM WIDGETS

```
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />

<ImageView
    android:id="@+id/icon"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="@dimen/margin_row"
    android:contentDescription="@string/icon_caption"
    android:src="@mipmap/ic_launcher"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/log" />

<ImageButton
    android:id="@+id/button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="@dimen/margin_row"
    android:contentDescription="@string/button_caption"
    android:src="@mipmap/ic_launcher"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/icon" />

<Switch
    android:id="@+id/swytch"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="@dimen/margin_row"
    android:text="@string/switch_caption"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/button" />

<CheckBox
    android:id="@+id/checkbox"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="@dimen/margin_row"
    android:text="@string/checkbox_caption"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/swytch" />

<RadioGroup
    android:id="@+id/radioGroup"
```

INTEGRATING COMMON FORM WIDGETS

```
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="@dimen/margin_row"
        android:background="@color/radiogroup_background"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/checkbox">

    <RadioButton
        android:id="@+id radioButton1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/radioButton1_caption" />

    <RadioButton
        android:id="@+id radioButton2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:checked="true"
        android:text="@string/radioButton2_caption" />

    <RadioButton
        android:id="@+id radioButton3"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/radioButton3_caption" />
</RadioGroup>

<SeekBar
    android:id="@+id/seekbar"
    android:layout_width="0dp"
    android:layout_height="match_parent"
    android:layout_marginTop="@dimen/margin_row"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/radioGroup" />

</androidx.constraintlayout.widget.ConstraintLayout>
</ScrollView>
```

INTEGRATING COMMON FORM WIDGETS

(from [FormWidgets/src/main/res/layout/activity_main.xml](#))

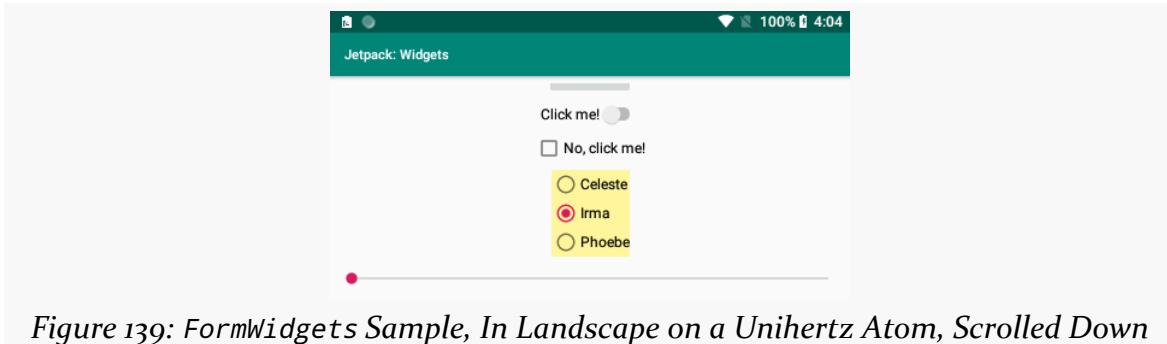


Figure 139: FormWidgets Sample, In Landscape on a Unihertz Atom, Scrolled Down

Note that there is also a `HorizontalScrollView` that will allow your form to be wider than the screen, whereas `ScrollView` allows your form to be taller than the screen.

Other Notes About the Sample

There are a few more interesting elements of the sample app that are worth noting.

`android:gravity`

There is a `TextView` at the top of the layout. Initially, it shows the message “Oh, c’mon, click something!”, but if you follow those instructions and start interacting with the other widgets, the message changes to reflect the recent event (e.g., “Button clicked!”).

The `TextView` has an `android:gravity="center_horizontal"` attribute:

```
<TextView  
    android:id="@+id/log"  
    android:layout_width="0dp"  
    android:layout_height="wrap_content"  
    android:background="@color/log_background"  
    android:gravity="center_horizontal"  
    android:text="@string/log_default"  
    android:textStyle="bold"  
    app:layout_constraintEnd_toEndOf="parent"  
    app:layout_constraintStart_toStartOf="parent"  
    app:layout_constraintTop_toTopOf="parent" />
```

(from [FormWidgets/src/main/res/layout/activity_main.xml](#))

“Gravity” means “hey, if there is room, position my content here”.

INTEGRATING COMMON FORM WIDGETS

center_horizontal gravity says “hey, if there is room horizontally, center my content”. For a TextView, the “content” is the text. The width is set to 0dp, which for a child of ConstraintLayout means that the TextView will be stretched between its start and end anchor points. Those are at the edges of the ConstraintLayout, so the TextView stretches to fill the width of the screen, less 8dp of padding that we have in the ScrollView. The TextView has a gray background, so in the screenshots, you see the full extent of the TextView plus the centered message.

Another way of centering things is through ConstraintLayout itself. If we switch the width of the TextView to wrap_content, ConstraintLayout will interpret that as meaning that the TextView should be centered between its start and end anchor points (assuming there is no bias setting to slide it one way or another):

```
<TextView  
    android:id="@+id/log"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:background="@color/log_background"  
    android:text="@string/log_default"  
    android:textStyle="bold"  
    app:layout_constraintEnd_toEndOf="parent"  
    app:layout_constraintStart_toStartOf="parent"  
    app:layout_constraintTop_toTopOf="parent" />
```

INTEGRATING COMMON FORM WIDGETS

However, in that case, the `TextView` itself is only as big as its content, so the gray background does not stretch to fill the screen:

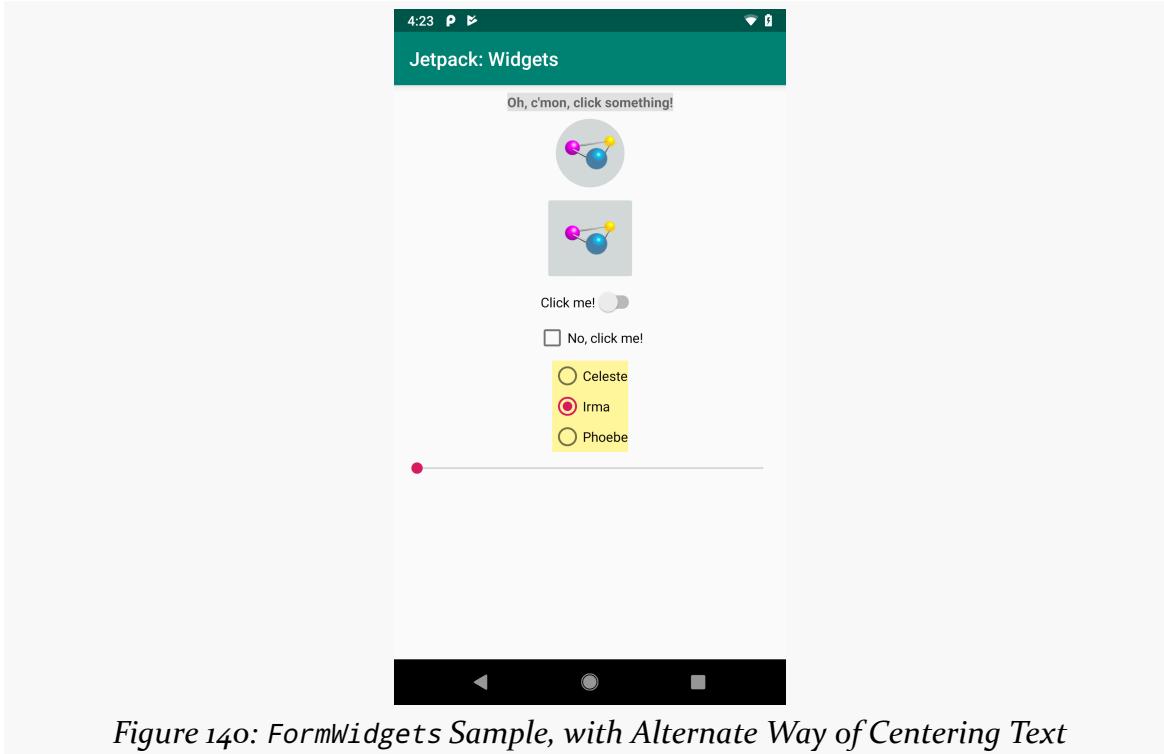


Figure 140: *FormWidgets* Sample, with Alternate Way of Centering Text

`log()`

That `TextView` is the `log` widget. Our `log()` function is set up to:

- Display a message in that `TextView`, and
- Log that same message to Logcat

So, we have a `log()` method in Java:

```
private void log(@StringRes int msg) {
    log.setText(msg);
    Log.d(TAG, getString(msg));
}
```

(from [FormWidgets/src/main/java/com/commonsware/jetpack/samplerj/formwidgets/MainActivity.java](#))

...and a corresponding `log()` function in Kotlin:

INTEGRATING COMMON FORM WIDGETS

```
private fun log(@StringRes msg: Int) {  
    log.setText(msg)  
    Log.d(TAG, getString(msg))  
}
```

(from [FormWidgets/src/main/java/com/commonsware/jetpack/sampler/formwidgets/MainActivity.kt](#))

The parameter to `log()` has a `@StringRes` annotation. This has no effect at runtime. However, it will help the build tools avoid some compile-time problems akin to the bug from [the chapter on debugging](#). There, we passed an arbitrary `Int` to `setText()` on a `TextView`. That function expects a string resource, and so it crashes if you pass some `Int` that does not represent a string resource. The `@StringRes` annotation on an `Int` says “hey! this `Int` should point to a string resource!”. If we attempt to call `log()` with an arbitrary `Int`, we will get an error in Android Studio:



This error message is being provided by a tool called Lint. The IDE uses Lint and its set of rules to identify possible problem spots in the code, such as calling `log()` with a value that is not a string resource ID.

INTEGRATING COMMON FORM WIDGETS

We also want to log messages from our SeekBar. There, though, we want to show the current value of the SeekBar, based on where the user has positioned the thumb:

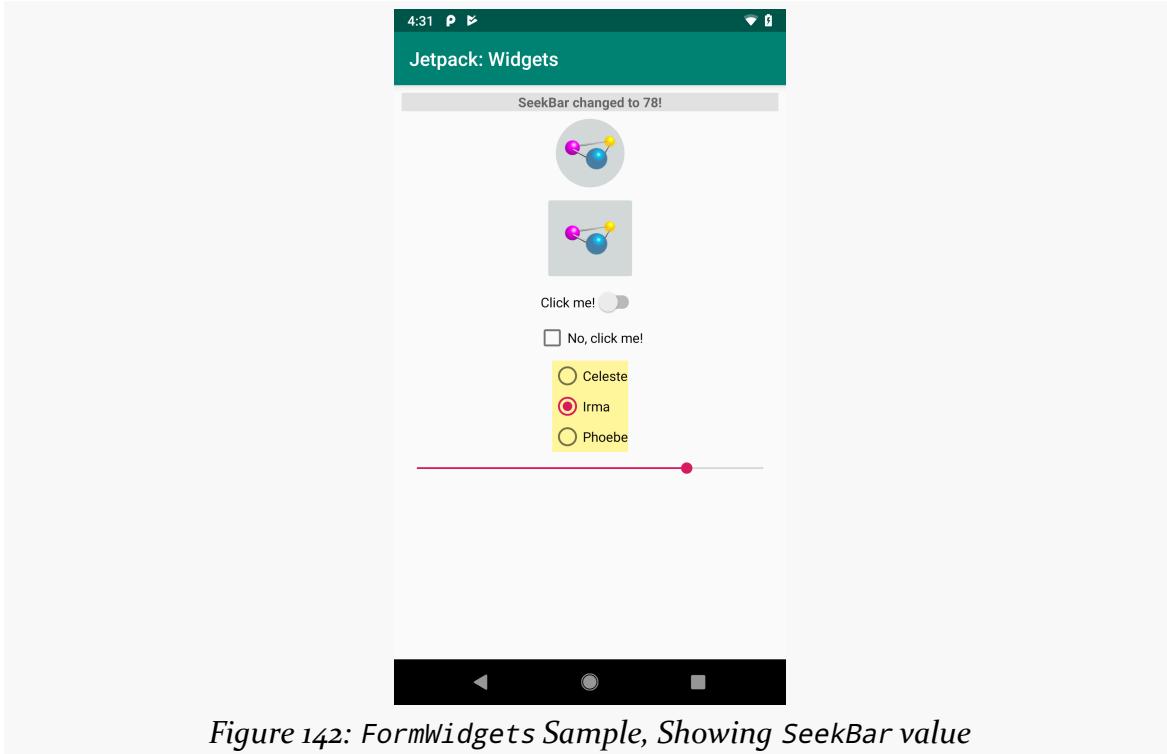


Figure 142: FormWidgets Sample, Showing SeekBar value

Our `log()` function is not set up for that, so we instead just do that bit of logging directly from our `OnSeekBarChangeListener`, whether that is implemented in Java:

```
((SeekBar) findViewById(R.id.seekbar))
.setOnSeekBarChangeListener(new SeekBar.OnSeekBarChangeListener() {
    @Override
    public void onProgressChanged(SeekBar seekBar, int progress,
        boolean fromUser) {
        String msg = getString(R.string.seekbar_changed, progress);

        log.setText(msg);
        Log.d(TAG, msg);
    }

    @Override
    public void onStartTrackingTouch(SeekBar seekBar) {
        // ignored
    }
})
```

INTEGRATING COMMON FORM WIDGETS

```
@Override  
public void onStopTrackingTouch(SeekBar seekBar) {  
    // ignored  
}  
});
```

(from [FormWidgets/src/main/java/com/commonsware/jetpack/samplerj/formwidgets/MainActivity.java](#))

...or Kotlin:

```
seekbar.setOnSeekBarChangeListener(object :  
    SeekBar.OnSeekBarChangeListener {  
        override fun onProgressChanged(  
            seekBar: SeekBar,  
            progress: Int,  
            fromUser: Boolean  
        ) {  
            val msg = getString(R.string.seekbar_changed, progress)  
  
            log.text = msg  
            Log.d(TAG, msg)  
        }  
  
        override fun onStartTrackingTouch(seekBar: SeekBar) {  
            // unused  
        }  
  
        override fun onStopTrackingTouch(seekBar: SeekBar) {  
            // unused  
        }  
    })
```

(from [FormWidgets/src/main/java/com/commonsware/jetpack/samplerj/formwidgets/MainActivity.kt](#))

Our string resource has a placeholder in it:

```
<string name="seekbar_changed">SeekBar changed to %d!</string>
```

(from [FormWidgets/src/main/res/values/strings.xml](#))

Here, %d in our string resource means “we will supply some Int at runtime to fill in here”. `getString()` not only takes a string resource ID, but it optionally takes objects to use for those placeholders.

The primary function of an `OnSeekBarChangeListener` is `onProgressChanged()`. This will be called when the `SeekBar` value is adjusted. You are passed:

INTEGRATING COMMON FORM WIDGETS

- The Seekbar itself,
- The new value of the SeekBar, and
- A Boolean indicating if the user changed the SeekBar value or if you did programmatically

So, our `getString(R.string.seekbar_changed, progress)` call takes the `progress` value and uses that to replace the `%d` in the string resource, giving us the text to display in the `TextView` and Logcat.

`OnSeekBarChangeListener` also has two other methods:

- `onStartTrackingTouch()`, which will be called when the user starts dragging the thumb
- `onStopTrackingTouch()`, which will be called when the user stops dragging the thumb

Most uses of `SeekBar` do not need to worry about those events, but we have to have those methods to satisfy the compiler, so the `FormWidgets` sample just has them as empty functions.

Licensed solely for use by Patrocinio Rodriguez

Contemplating Contexts

When we call `getString()` to convert a string resource into its actual `String` value, we have been calling it on our `MainActivity`. However, in reality, that function is implemented on `Context`. `MainActivity` extends `AppCompatActivity`, which extends `Activity`, which implements the `Context` interface.

We use `Context` objects a *lot* in Android.

So, in this chapter, let's quickly review what a `Context` is, where to get one, and how we use it.

It's Not an OMG Object, But It's Close

Back when Android was first created, in the mid-2000's, the dominant form of Java programming was for server-side Web development, using servlets, WARs, and so on. In Web development, there is often some form of "session" object. At minimum, this object represents the user's session with the server and represents a place where the servlet can stash values from one request that might be used in a subsequent request from the same user.

Computer programming has lots of "anti-patterns": things that we really should not do. One of those anti-patterns is "[the God object](#)", where we have one class that tries to do far too much. Session objects in Web development have a tendency to become God objects, because it is simple to say, "oh, well, we'll just have the session handle that", even for things that have little to do with maintaining user state across requests.

In Android, the God object is `Context`. A lot of functionality routes through a `Context`, either because:

CONTEMPLATING CONTEXTS

- Our only ways of accomplishing a particular thing involve functions on a Context, or
- Our only way to get at some other object that accomplishes that thing is by calling a function on the Context

For lightweight app development, such as most of this book's samples, this does not pose much of a problem. Any code in an activity, for example, has easy access to a Context, as the activity itself *is* a Context. In larger apps, though, quite a bit of time is spent trying to figure out how best to write the code to reduce the number of places where we need a Context.

The Major Types of Context

Context is an interface, so in theory anything could be a Context. However, there are only a few Context implementations that actually "do the heavy lifting" required of a fully-working Context.

Context from Components

Primarily, your components will give you a Context.

The one component that we have worked with so far is `Activity`. An `Activity` is itself a Context, which is why we can call `getString()` on it.

There are three other types of components:

- `Service`, which, like `Activity`, implements Context itself
- `BroadcastReceiver`, where you get a Context via its `onReceive()` function that you implement or inherit
- `ContentProvider`, where you get a Context by calling `getContext()` when you need one

The *vast* majority of the time, using a Context that you get from a "nearby" component is the right answer. For example, for code that is in an `Activity`, most of the time you will want to use the `Activity` itself as your Context.

Application

There is one other "root" Context besides your components: `Application`.

CONTEMPLATING CONTEXTS

Each Android app process has an `Application` singleton. This object is created when the process is created, and it lives through the life of that process. In your regular app code, you get access to this singleton... by calling `getApplicationContext()` on some other Context.

As a result, `Application` may seem somewhat pointless. If you already have a Context, why would you need `getApplicationContext()` to get a *different* Context?

The answer lies in lifespan. Other types of Context — particularly an `Activity` — have short lifespans. As we will see [later in the book](#), an `Activity` will come and go from the screen, even in some cases where the user might not think that there has been any change. Sometimes, we will need a Context that lives longer than this, and for that, we have `Application`.

That being said, [only use `Application` when you have a clear reason why you need it](#).

Instrumented Tests

Note that in testing we get a Context in other ways.

For example, back in [the chapter introducing testing](#), we had an `ExampleInstrumentedTest` class, both in Kotlin:

```
package com.commonsware.jetpack.hello

import androidx.test.platform.app.InstrumentationRegistry
import androidx.test.ext.junit.runners.AndroidJUnit4

import org.junit.Test
import org.junit.runner.RunWith

import org.junit.Assert.*

/**
 * Instrumented test, which will execute on an Android device.
 *
 * See [testing documentation](http://d.android.com/tools/testing).
 */
@RunWith(AndroidJUnit4::class)
class ExampleInstrumentedTest {
    @Test
    fun useAppContext() {
        // Context of the app under test.
        val applicationContext = InstrumentationRegistry.getInstrumentation().targetContext
        assertEquals("com.commonsware.jetpack.hello", applicationContext.packageName)
    }
}
```

CONTEMPLATING CONTEXTS

...and in Java:

```
package com.commonsware.jetpack.hello;

import android.content.Context;
import androidx.test.platform.app.InstrumentationRegistry;
import androidx.test.ext.junit.runners.AndroidJUnit4;

import org.junit.Test;
import org.junit.runner.RunWith;

import static org.junit.Assert.*;

/**
 * Instrumented test, which will execute on an Android device.
 *
 * @see <a href="http://d.android.com/tools/testing">Testing documentation</a>
 */
@RunWith(AndroidJUnit4.class)
public class ExampleInstrumentedTest {
    @Test
    public void useApplicationContext() {
        // Context of the app under test.
        Context applicationContext =
            InstrumentationRegistry.getInstrumentation().getTargetContext();

        assertEquals("com.commonsware.jetpack.hello", applicationContext.getPackageName());
    }
}
```

InstrumentationRegistry.getTargetContext() returns a Context from the app that we are testing.

Key Context Features

Many things take a Context. Constructors for all of our widgets and containers, for example, take a Context as a parameter.

In terms of directly calling functions on a Context, though, there are a few categories into which those functions fall.

Access to Resources and Assets

If you want to get the value or otherwise use a resource, you will wind up using a

CONTEMPLATING CONTEXTS

Context.

Sometimes, that comes from passing the Context to something else as a parameter. We will see examples of that later in the book, such as [data binding](#).

Sometimes, that comes from calling convenience functions on Context itself, such as `getString()`.

In general, though, access to resources comes from a Resources object, and you get one of those by calling `getResources()` on a Context. Resources has functions to retrieve most types of resources: strings, dimensions, colors, and so on. The convenience functions on Context simply delegate to the Resources object — for example, `getString()` on Context just forwards your request to `getString()` on Resources.

In addition to resources, an Android app can also have assets, found in an `assets/` directory that sits alongside directories like `src/` and `res/`. If you package assets this way with your app, calling `getAssets()` on a Context gives you an `AssetManager`. From there, you can work with the assets, such as by calling `open()` to get an `InputStream` from which you can read an asset's content.

Access to Root Directories

Resources and assets are packaged with the app. Beyond those, we have the ability to write to files on the local filesystem and read them back later. On the whole, this is standard I/O using Java classes like `File`, `InputStream`, `OutputWriter`, and so on. The big limitation is *where* we can read and write.

For that, Context has a family of functions, like `getFilesDir()`, that return `File` objects representing directories that we can read from and (usually) write to. We will explore those functions in detail [later in the book](#).

Access to System Services

Lots of stuff used by your app is actually managed outside of your app.

For example, suppose that you want to find out the device's location. Your app code does not work with GPS hardware directly. Instead, it will work with a `LocationManager` or something that wraps around a `LocationManager`. `LocationManager` is a "system service", and it in turn will talk to other parts of the OS that, eventually, work with the GPS hardware.

CONTEMPLATING CONTEXTS

Whenever you need access to a “system service”, the typical approach to get one is to call `getSystemService()` on a Context. Occasionally, there will be other approaches:

- Some system services have convenience functions on Context, such as `getPackageManager()` to get a PackageManager instance
- Some system services have a function that you can use to get an instance by way of a Context, such as `LayoutInflater.from()`

Access to Other Components

As we start to explore Android’s components more, you will see that if we want to work with other components from an existing component, we will need a Context.

Specifically, we use a Context to:

- Start an Activity
- Start or bind to a Service
- Send a system broadcast to be picked up by a BroadcastReceiver
- Work with a ContentProvider

Know Your Context

As your app grows, you will start having classes that are not Android components and do not have direct access to some Context. Instead, they will be talking with your Web services, or storing things in a local database, or performing any number of other tasks that are important for your app.

In an ideal world, few of those classes will need a Context. For those that do, you will need to decide *what* Context they will use:

- Do they take a Context as a function parameter for a particular operation, so the caller chooses the Context?
- Do they take a Context as a constructor parameter, so the choice of Context has to take into account how long this object is needed?
- Do they do something more elaborate, such as use dependency injection?

Context Anti-Patterns

The reason why you need to know what Context you are using in different circumstances is because there are several anti-patterns when working with Context.

CONTEMPLATING CONTEXTS

Some of these will result in warnings in the IDE, but not always, as sometimes the IDE cannot identify that you are implementing an anti-pattern.

We will explore these anti-patterns later in the book:

- [Avoiding static references to component Context objects](#)
- [Trying to use Application for everything](#)
- [Assuming that a Context is always of some specific type](#)

Licensed solely for use by Patrocinio Rodriguez

Icons

Each app has an icon. For most apps, the user will see that icon in their launcher, for whatever activity (or activities) are advertised as belonging in the launcher. In addition, this icon can appear in other places, such as in the Settings app.

App icons are `drawable` resources. Except when they are `mipmap` resources. And except when they are multiple resources, combined together at runtime to create an image with a “squircle” background. And...

Are you confused yet?

In theory, setting up app icons would be quite simple, and for years that was the case. Nowadays, setting up an app icon is unnecessarily complex, though at least Android Studio has an Asset Studio tool to try to make it a bit simpler.

In this chapter, we will explore what it takes to set up one of these app icons.

App Icons... And Everything Else

Google has been making app icons increasingly complicated over the past few years:

- App icons are `mipmap` resources, while everything else is a `drawable` resource
- Android 7.1 introduced the concept of a separate “round icon” that an app can have, which would be used in place of the regular app icon on certain Android 7.1 devices... then dropped this feature with Android 8.0
- Android 8.0 introduced the concept of “adaptive icons”, where you have to provide separate “foreground” and “background” images, mostly so that certain home screen launchers can shape the background image as they want (square, round, “squircle”, etc.)

Creating an App Icon with the Asset Studio

If you right-click over pretty much any directory in the Android Studio tree, the context menu will have an “Image Asset” option. This is also available from File > New > Image Asset. This brings up the Asset Studio, to help you to assemble icons.

By default, the Asset Studio has its “Icon Type” drop-down set for “Launcher Icons (Adaptive and Legacy)”, which is how you set up an app icon nowadays:

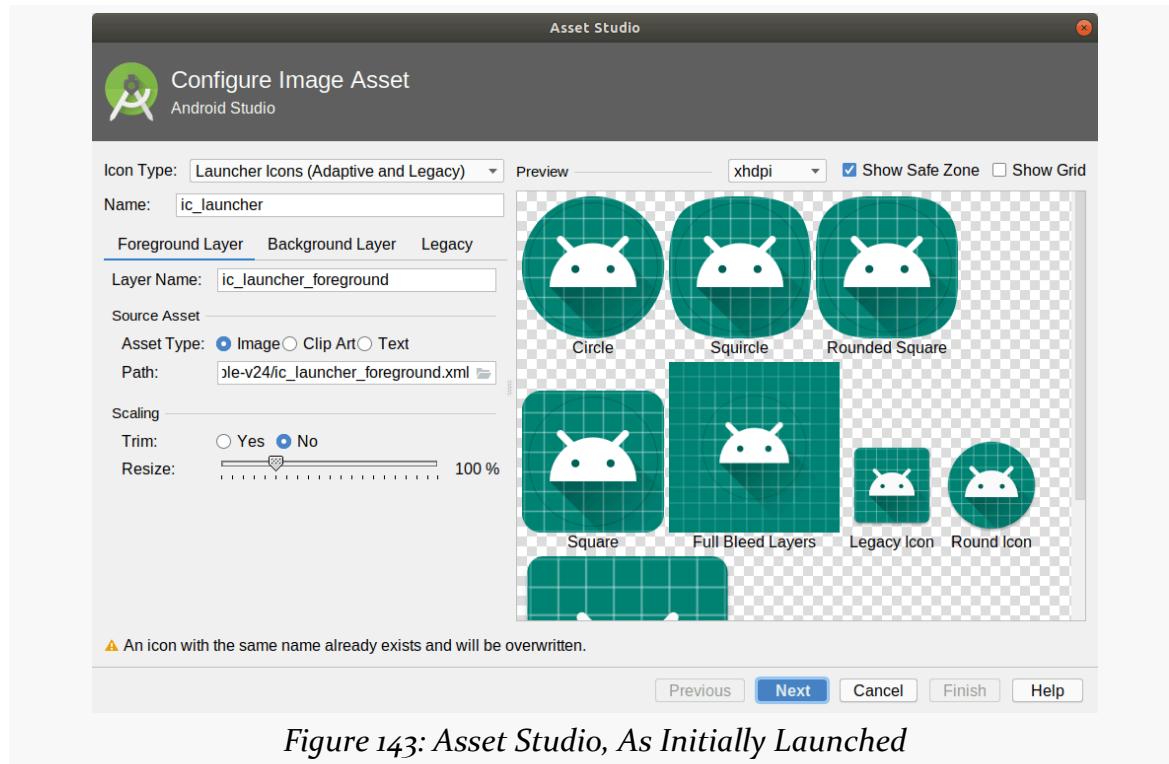


Figure 143: Asset Studio, As Initially Launched

The overall name for your launcher icon is found in the Name field, above the tabs. The default is `ic_launcher`, and unless you have a good reason to change it, you are best served by leaving it alone.

Foreground Layer

What you would ordinarily think of as your icon is what Android Studio and Android 8.0+ refer to as the “Foreground Layer”. The Asset Studio starts on the Foreground Layer tab for you to configure this layer.

ICONS

Your icon can come from three main sources:

- Some image of your own design, which you would load into the Foreground Layer tab by selecting the Image radio button, then clicking the “...” button next to the Path field, to browse your development machine and find that image
- A piece of canned clip art, which you would choose by clicking on the “Clip Art” radio button, clicking the “Clip Art” button to choose the image, and clicking the Color button to choose a color to apply to that image:

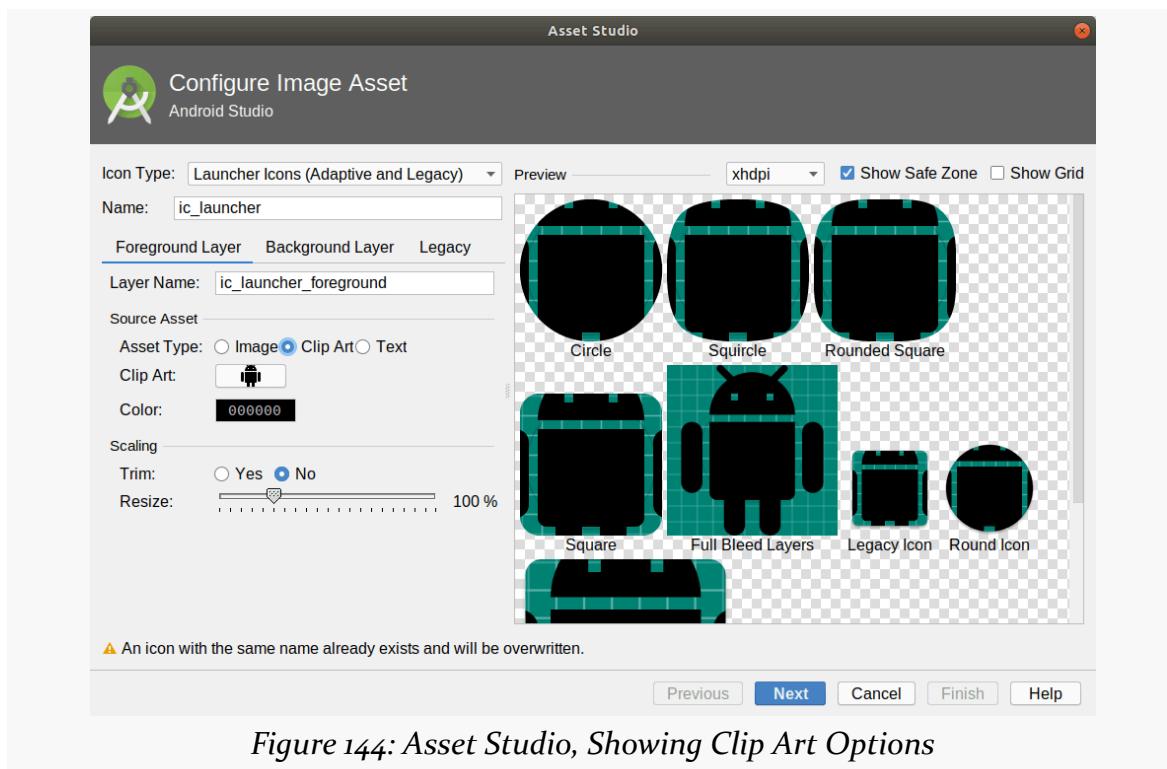


Figure 144: Asset Studio, Showing Clip Art Options

ICONS

- A letter or two, which you would choose by clicking on the Text radio button, filling in 1-2 letters in the Text field, choosing a font from your development machine in the adjacent drop-down, and clicking the Color button to choose a color to apply to the font:

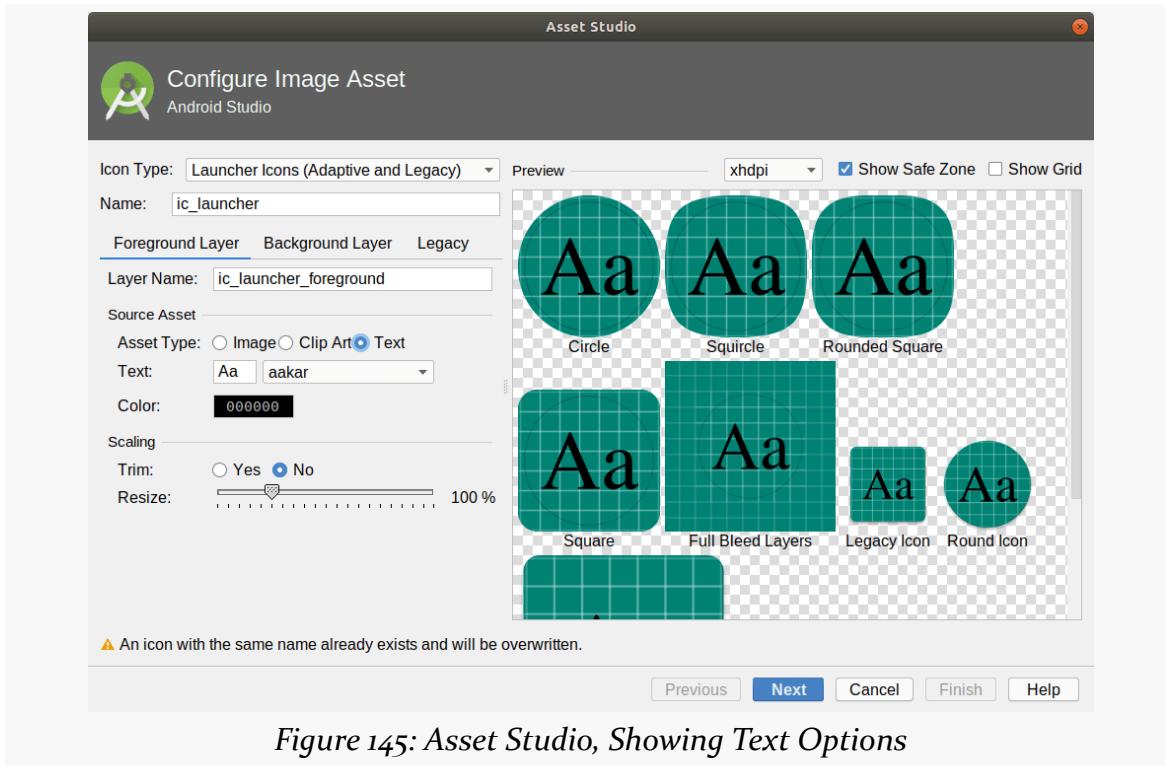


Figure 145: Asset Studio, Showing Text Options

For all three of these, you can:

- Choose the name to use for this image, in the “Layer Name” field
- Choose, via the Trim radio buttons, whether to remove all transparent space from the edges of the image (“Yes”) or not (“No”)
- Resize the image from its default size

That latter choice is particularly important, as you need to keep your foreground layer content within the “safe zone”. That shows up in the previews as a dark gray circle. So long as your foreground layer is in the safe zone, you should not have to worry about anything accidentally cutting off part of the layer when it renders your app icon.

ICONS

Background Layer

In the preview area, by default, you will see a blue-green grid behind your foreground layer. That is the default background layer. If you would like to use something else, click the “Background Layer” tab:

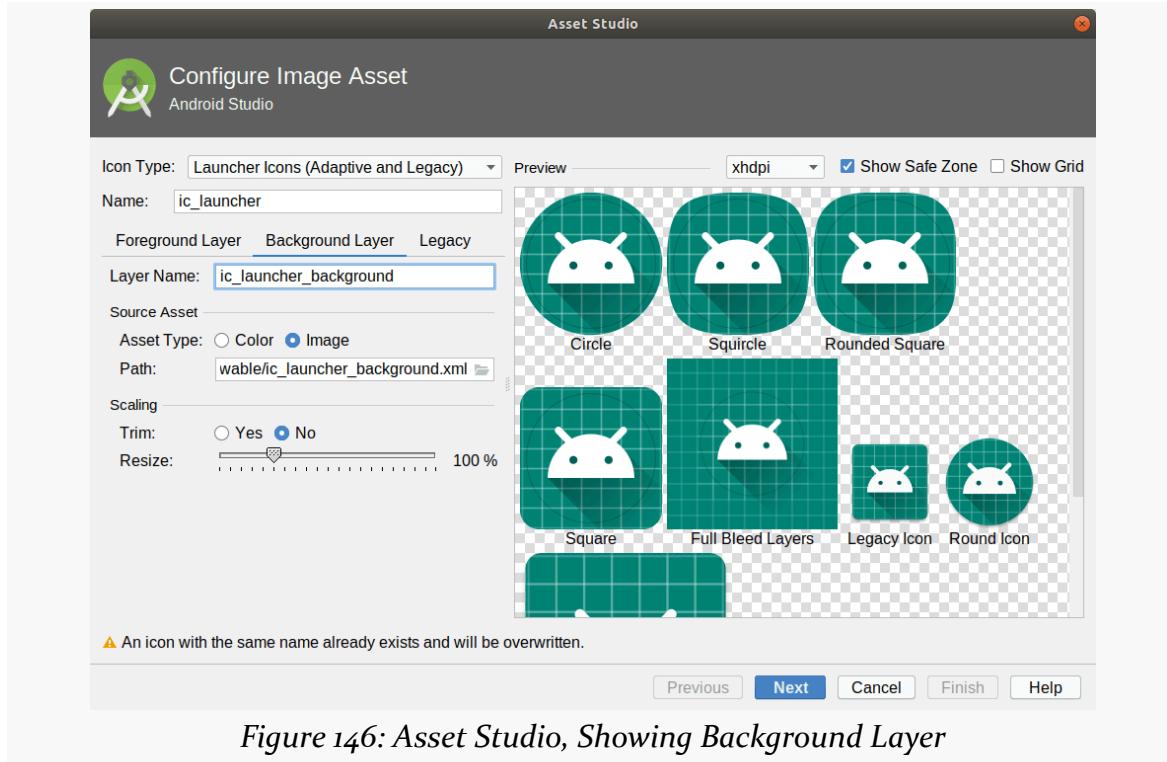


Figure 146: Asset Studio, Showing Background Layer

Your two main options are:

- a flat color
- an image of your choosing (akin to the foreground, where you click the “...” button next to the Path field to pick the background image)

The background needs to be designed to be cropped into a variety of shapes, such as those shown in the preview (circle, various rounded forms of squares, etc.). Hence, outside of flat colors, typical backgrounds will be gradients or simple patterns, such as the default grid.

Legacy

On Android 8.0+ devices, the foreground layer, background layer, and device-chosen

ICONS

shape (e.g., squircle) combine to create your app icon.

On older devices, the app icon is whatever you choose it to be, where the Legacy tab helps you decide what that is:

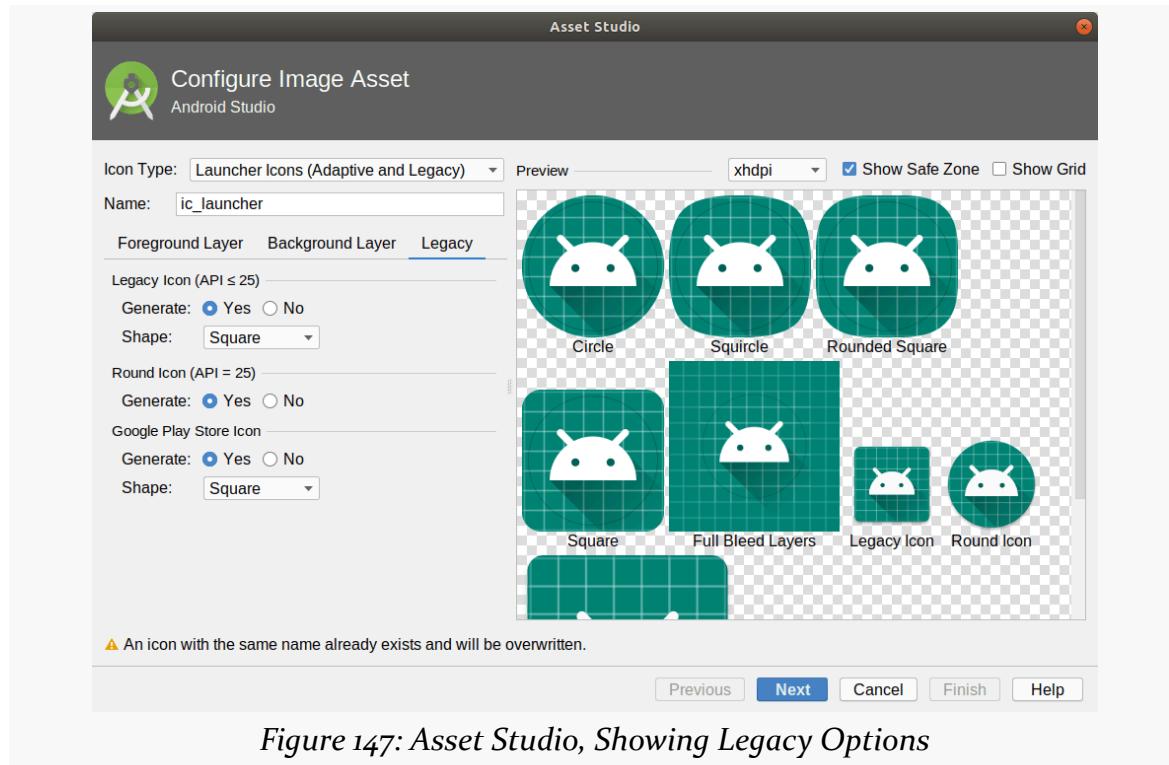


Figure 147: Asset Studio, Showing Legacy Options

The most important legacy option is the “Legacy Icon (API <= 25)” section. If your `minSdkVersion` is below 26, this will be the icon rendition that will be used as your app icon by default. The Asset Studio will merge your foreground and background layers itself, then apply your selected shape from the drop-down (e.g., square).

For Android 7.1 devices, you can also opt to have the Asset Studio create a separate round icon, that you can declare in your manifest, as will be seen later in this chapter.

If you are going to be distributing your app through the Play Store, you can also generate a Play Store rendition of your icon. This is reminiscent of the legacy icon, but at a higher resolution.

ICONS

Generating the Icon

Once you have adjusted your app icon via the three tabs, click Next at the bottom of the Asset Studio wizard. This will bring up the final wizard page, showing you what will be generated for you by the wizard:

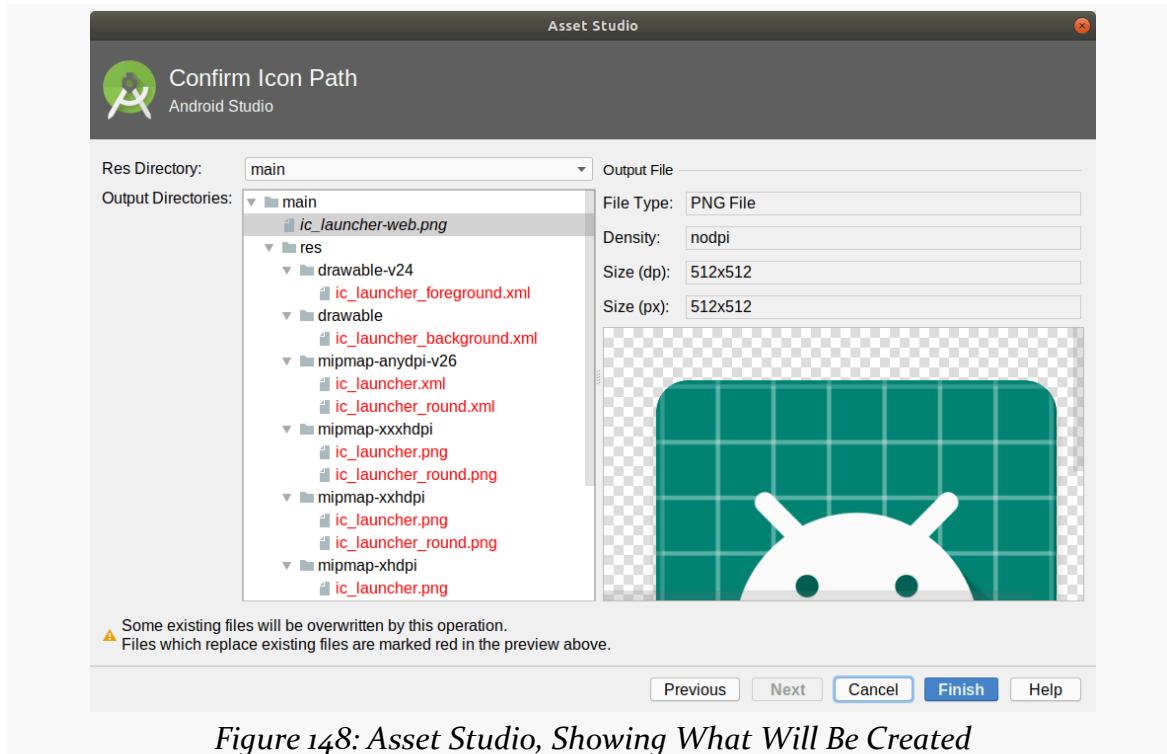


Figure 148: Asset Studio, Showing What Will Be Created

The Output Directories tree shows you each file that will be created or replaced. Those that show up in red are ones that will be replaced; ones that show up in black are new files. Typically, unless you changed the icon or layer names, most of the files will be replacements for files that exist already.

Some of these files will be typical bitmap-style resources. Some are vector drawables, a concept that we will explore [in an upcoming chapter](#).

Using In Your Manifest

Then, in your manifest, you can have an `android:icon` attribute on the `<application>` element to associate your icon with your app. If you did not change the names of the icon, then your manifest should already have the appropriate attribute values.

ICONS

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

  <application
    android:allowBackup="true"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:supportsRtl="true"
    android:theme="@style/AppTheme">
    <activity android:name=".MainActivity">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />

        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
  </application>

</manifest>
```

You can also have an `android:roundIcon` attribute. This is for the Android 7.1-specific scenario of having a dedicated round icon resource. If you elected to have Android Studio create a round icon for you, ensure that your `<application>` element has an `android:roundIcon` attribute that points to the round icon resource (e.g., `@mipmap/ic_launcher_round`).

Creating Other Icons with the Asset Studio

In principle, you can use the Asset Studio to create other types of icons, by choosing another type of icon from the drop down, such as “Action Bar and Tab Icons”.

In practice, the Asset Studio does not do much for you here, other than create multiple versions of your icon for different screen densities.

For most Android app developers, there are two other options:

1. Get icons at the right resolutions for different densities from your graphic designer, perhaps exported from Adobe Photoshop
2. Create vector icons, as will be covered in [an upcoming chapter](#)

Adding Libraries

While you are writing some code for an app, the vast majority of the code that *is* the app comes from other developers. Some might be teammates on your development team, but far more comes from outsiders: Google and other Android developers.

You have seen some of this already. You did not write `Activity`, `TextView`, and similar classes. Instead, they came from the Android SDK, written (primarily) by Google.

Beyond the Android SDK, though, there are thousands of libraries that developers have access to, including many from Google itself. We add these as dependencies in our projects, to use their code alongside ours.

You have seen some of that already too, in the form of libraries for things like `ConstraintLayout`.

In this chapter, we will explore a bit more about these libraries, how you find them, and how you integrate them.

Depending on a Local JAR

Going back to the `HelloWorld` and `HelloWorldJava` samples from the beginning of the book, one of our `implementation` statements to pull in a library looked like this:

```
implementation fileTree(dir: 'libs', include: ['*.jar'])
```

This pulls in any JAR files that happen to be in the `libs/` directory of this module.

JARs, as you probably know, are libraries containing Java code, as created by

ADDING LIBRARIES

standard Java build tools (javac, jar, etc.). For the first decade-plus of Java's existence, we distributed reusable bits of code in the form of JAR files. You would download a JAR from a Web site, drop it into your project, and through something like this implementation statement, say that your project should use the JAR.

In Android, the contents of these JARs are packaged into your APK. Whatever public classes happen to be in those JARs are available to you at compile time.

However, in general, using plain JARs nowadays is considered to be a bad idea. There is no information in a JAR about:

- What version of the library the JAR represents — while this could be part of the filename, files can be renamed far too easily
- What other libraries this JAR requires — at best, you find that out from documentation, then need to hunt down those JARs and see what *they* require, and so on

Instead, nowadays, you should try to use artifacts, rather than bare JAR files.

Artifacts and Repositories

An artifact is usually represented in the form of two files:

- The actual content, such as a JAR
- A metadata file, paired with the JAR, that has information about “transitive dependencies” (i.e., the other artifacts that this artifact depends upon)

Artifacts are housed in artifact repositories. Those repositories not only contain the artifacts, but they organize the artifacts for easy access. This includes organizing them by version, so you can request a specific version of an artifact and get it, instead of an older (or possibly newer) version of that same artifact.

Some artifact repositories are public. A typical Android project will use two such repositories:

- JCenter, a popular place for open source artifacts
- Google's repository, for things like the Android Gradle Plugin

There are other general-purpose artifact repositories, such as Maven Central or jitpack.io. And there can be private repositories, such as ones used by organizations for their own private artifacts, used by their private projects.

ADDING LIBRARIES

So, we have dependencies like these:

```
implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlin_version"
implementation 'androidx.appcompat:appcompat:1.0.2'
implementation 'androidx.core:core-ktx:1.0.2'
implementation 'androidx.constraintlayout:constraintlayout:1.1.3'
testImplementation 'junit:junit:4.12'
androidTestImplementation 'androidx.test:runner:1.2.0'
```

For those, Gradle checks with our registered repositories and says “hey, do you have this particular artifact?”. This is based on the group (e.g., `org.jetbrains.kotlin`), the artifact ID (e.g., `kotlin-stdlib-jdk7`), and the desired artifact version. If a match is found, Gradle downloads it and caches it for later use. If no match is found, you get a build error pointing out the difficulty.

Requesting Dependencies

With all that as background, let’s explore a bit more about how those `implementation` statements are working and how you can add your own for other dependencies that you may want to use.

Find What You Need

First, you need to identify the libraries that address whatever problems you need to solve.

Many libraries will be covered in this book. Yet more are covered in other CommonsWare books. A few more are in the developer documentation.

The biggest catalog of open source libraries is [the Android Arsenal](#). Here you can browse and search for libraries. Each listing will contain links to where you can find out more about that particular library, typically in the form of a GitHub repository.

Beyond that, you will find blog posts, Stack Overflow answers, and other resources pointing out candidate libraries to use.

Configure the Repositories

If you are using one of Google’s libraries, a project created in Android Studio should be set up with the proper artifact repository already. Your project should also come pre-configured to pull from JCenter, where many open source libraries are from. As a

ADDING LIBRARIES

result, for the vast majority of libraries, you do not need to configure an additional artifact repository.

But, sometimes you do.

For example, you may find some libraries that advise you to configure support for jitpack.io:

```
repositories {  
    maven {  
        url "https://jitpack.io"  
    }  
}
```

This is because they are using JitPack to publish their libraries, instead of JCenter.

You can do one of two things:

1. Add that `maven {}` closure to the `repositories` closure in the `allprojects` closure in your top-level `build.gradle` file
2. Add the entire sample `repositories` closure to your module's `build.gradle` file

Either of these will make this artifact repository available to you for that module. The first approach makes it available for *all* modules in your app, for if you create a more complex app that involves multiple modules.

So, when you review the documentation for a dependency, it should indicate what artifact repository to use, and you need to ensure that you are set up to use that repository. If the library does not state what artifact repository it uses, try adding the dependency, and if it works, then (hopefully) they published the artifact on JCenter.

Identify the Version That You Want

As noted above, each artifact has an identifier made up of three pieces: a group ID, an artifact ID within the group, and a version number.

Technically, the version number can contain wildcards. For example, while `1.0.2` indicates a specific version, `1.0.+` says “pick the latest among all versions that start with `1.0`”. This is convenient for getting patches, but it means that on some random day, all of a sudden, you are using a new version of the library, and that might cause problems. Typically, we do not use wildcards, but instead just keep tabs on when the

ADDING LIBRARIES

artifacts get new versions. Android Studio will help with this, highlighting artifacts that have newer versions available.

For Google's artifacts, you can browse [a Web page for their Maven repository](#) to find the available versions for an artifact given its group ID and artifact ID.

For other artifacts, you usually go to the support site for that artifact, which for many artifacts is a GitHub project. A well-maintained library will have details of the latest version of that artifact, so you know what version to request in your module's `build.gradle` file.

Add the Dependencies

Then, you just need to add the appropriate `implementation` lines for whatever dependencies that you wish to add, alongside the existing ones in your module's `build.gradle` file.

Note that many libraries showing sample code for adding them to your `build.gradle` file will show a slightly different syntax, with a `compile` keyword instead of `implementation`:

```
compile 'com.whatever:something:1.2.3'
```

That is because Android Studio 3.0 and Gradle 4.1 switched to a new syntax for specifying dependencies. `compile` was replaced by `implementation`. If you use `compile`, your Gradle build script will still work... for now. Eventually, support for `compile` will be dropped, and so you should aim to use `implementation` instead of `compile` going forward.

We will see lots of dependencies closures throughout the rest of this book, showing different artifacts that we can depend upon.

Licensed solely for use by Patrocinio Rodriguez

Employing RecyclerView

If you have spent much time with Android devices and apps, no doubt that you will have seen apps that show lists or grids of stuff: contacts, songs, movies, books, to-do items, etc.

If the app was written in 2015 or later, there is a very good chance that it uses `RecyclerView` for those lists and grids. In this chapter, we will explore what `RecyclerView` is and how you can use it for your own collections of data.

Recap: Layouts vs. Adapter-Based Containers

As we saw back in [the chapter on ConstraintLayout](#), there are two major types of `ViewGroup` implementations:

- There are those that organize a set of children known at the time that you are writing the app
- There are those that organize a set of children that cannot be known until the app is run

Mostly, `ConstraintLayout` is for the first scenario. You know that you will have a certain set of widgets that you want to appear, and you need to size and position them. `ConstraintLayout` — and the legacy containers, like `LinearLayout` — excel at this role.

However, quite frequently, we have collections of stuff to display. We might know what sorts of stuff would be in the collection (contacts, songs, movies, books, to-do items, etc.). But we do not know *how big* the collection will be. Perhaps when we first run the app, the collection will be empty. Over time, the collection might grow. If we are reusing data from other places, such as the user's contact manager, there

EMPLOYING RECYCLERVIEW

might be lots of data from the outset. We just do not know.

ConstraintLayout would be useful for displaying an individual item out of that collection of stuff. However, trying to use ConstraintLayout and ScrollView to handle an arbitrary-sized collection would be really painful.

Fortunately, we have options.

The original solution for this sort of problem was the AdapterView family of classes, notably ListView. AdapterView worked but was fairly inflexible. RecyclerView was created as an alternative, and it is the dominant solution for this problem today. We will discuss ListView and the rest of AdapterView [towards the end of the chapter](#), but outside of certain scenarios, your focus nowadays should be on RecyclerView.

The Challenge: Memory

Suppose that our UI is supposed to look like this:

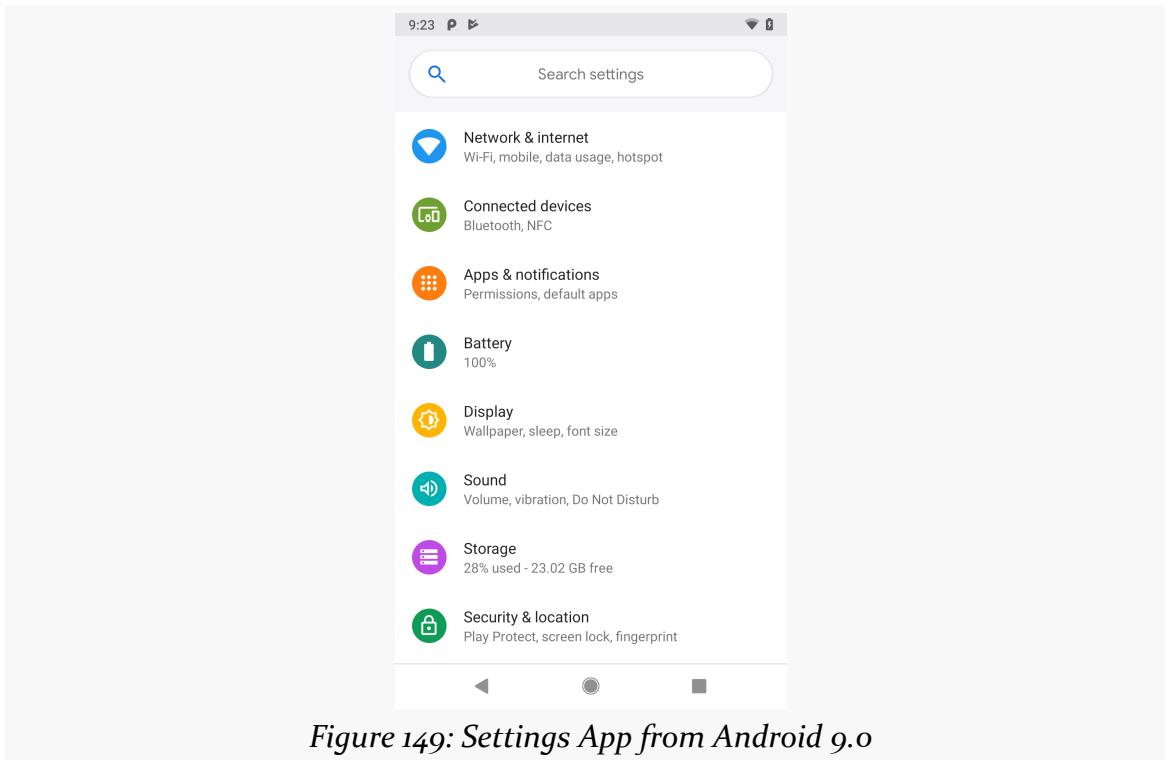


Figure 149: Settings App from Android 9.0

Here we have a vertically-scrolling list of items. Each item appears to have an

EMPLOYING RECYCLERVIEW

ImageView and a pair of TextView widgets.

This happens to be from the Settings app on an Android 9.0 device. If you have spent much time with Settings, you know that while it has a bunch of top-level options like these, there are only so many. We could, if needed, have one ConstraintLayout in a ScrollView that managed them.

But what happens if our list needs to have thousands of items in it?

Each widget — TextView, Button, ImageView, etc. — takes up around 1KB of memory, not counting its content (text, image, whatever). Several thousand widgets means several MB of memory.

On Android, the amount of memory your app can use is limited. On very old or very cheap devices, it might be as little as 16MB, though higher values (e.g., 32MB, 48MB) are a bit more common today. Still, there is always a cap. Having *one* screen use several MB of memory will be a problem for apps that have lots of screens, particularly if many of those screens will have their own long lists.

AdapterView and RecyclerView are designed around recycling. If you look at that screenshot again, no matter how many rows there might be in the list overall, the user can only see 8 of them on this particular screen at once. View recycling takes advantage of this, so we can limit our memory usage. Rather than having widgets for each item in the list, we have widgets for each *visible* item, plus a few more to help with reacting to scrolling events. As rows in the list scroll off the screen, they become eligible for reuse (recycling). This helps keep our memory usage to a more reasonable level, despite having a potentially very long list.

Enter RecyclerView

RecyclerView, on its own, does very little other than help manage view recycling (e.g., row recycling of a vertical list). It delegates almost everything else to other classes, such as:

- a layout manager, responsible for organizing the views into various structures (vertical list, grid, staggered grid, etc.)
- an item decorator, responsible for applying effects and light positioning to the views, such as adding divider lines between rows in a vertical list
- an item animator, responsible for animated effects as the model data changes

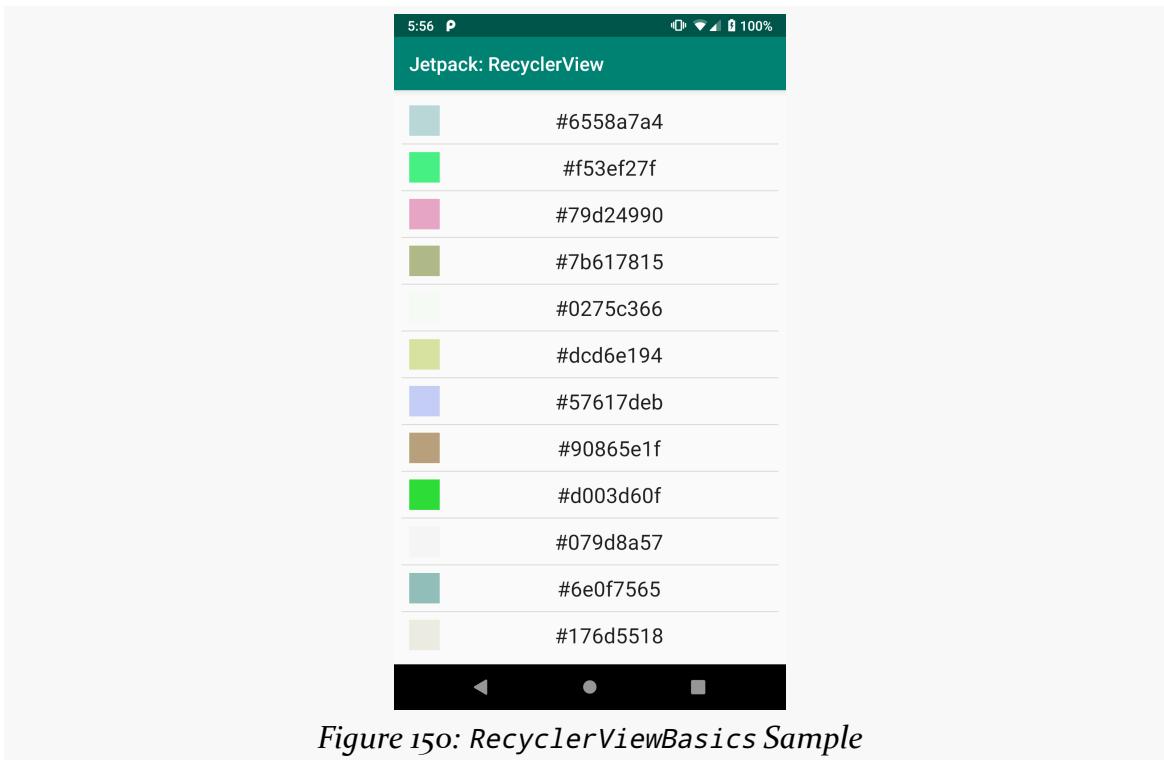
EMPLOYING RECYCLERVIEW

- and so on

Through “adapter” and “view-holder” classes, we teach RecyclerView what should go into the list rows, grid cells, or whatever. RecyclerView then handles the scrolling and recycling for us.

A Trivial List

The RecyclerViewBasics sample module in the [Sampler](#) and [SamplerJ](#) projects demonstrate a fairly simple list. The items in the list will be a set of randomly-generated integers, where we will show the values both as a hexadecimal number and as a color:



The Dependency

RecyclerView comes from a library, not the framework portion of the Android SDK. As a result, we need a new dependency, one for
androidx.recyclerview:recyclerview:

EMPLOYING RECYCLERVIEW

```
dependencies {  
    implementation 'androidx.appcompat:appcompat:1.0.2'  
    implementation 'androidx.recyclerview:recyclerview:1.0.0'  
    implementation 'androidx.constraintlayout:constraintlayout:1.1.3'  
    implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlin_version"  
}
```

(from [RecyclerViewBasics/build.gradle](#))

Elsewhere, you may see references to com.android.support:recyclerview-v7. That library offers the same RecyclerView, but is part of the older Android Support Library. Jetpack projects should use the androidx edition of RecyclerView.

The Layouts

This project has two layout resources: activity_main and row.

The Activity Layout

The activity_main layout resource contains a RecyclerView inside of a ConstraintLayout:

```
<?xml version="1.0" encoding="utf-8"?>  
<androidx.constraintlayout.widget.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"  
    android:padding="@dimen/content_padding"  
    tools:context=".MainActivity">  
  
<androidx.recyclerview.widget.RecyclerView  
    android:id="@+id/items"  
    android:layout_width="0dp"  
    android:layout_height="0dp"  
    app:layout_constraintBottom_toBottomOf="parent"  
    app:layout_constraintStart_toStartOf="parent"  
    app:layout_constraintEnd_toEndOf="parent"  
    app:layout_constraintTop_toTopOf="parent" />  
  
</androidx.constraintlayout.widget.ConstraintLayout>
```

(from [RecyclerViewBasics/src/main/res/layout/activity_main.xml](#))

EMPLOYING RECYCLERVIEW

As with ConstraintLayout itself, RecyclerView needs to have its fully-qualified class name in the layout resource XML, which is why we have an `<androidx.recyclerview.widget.RecyclerView>` element instead of simply a `<RecyclerView>` element. Otherwise, RecyclerView is just an ordinary thing that we can size and position as needed. Here, we have it set to fill the ConstraintLayout, excluding 8dp of padding that the ConstraintLayout applies.

Note that RecyclerView knows how to scroll, so we do not need to wrap RecyclerView in a ScrollView.

When constructing a new UI, if you wish to use the drag-and-drop GUI builder in Android Studio, you can find RecyclerView in the “Containers” category of the Palette:

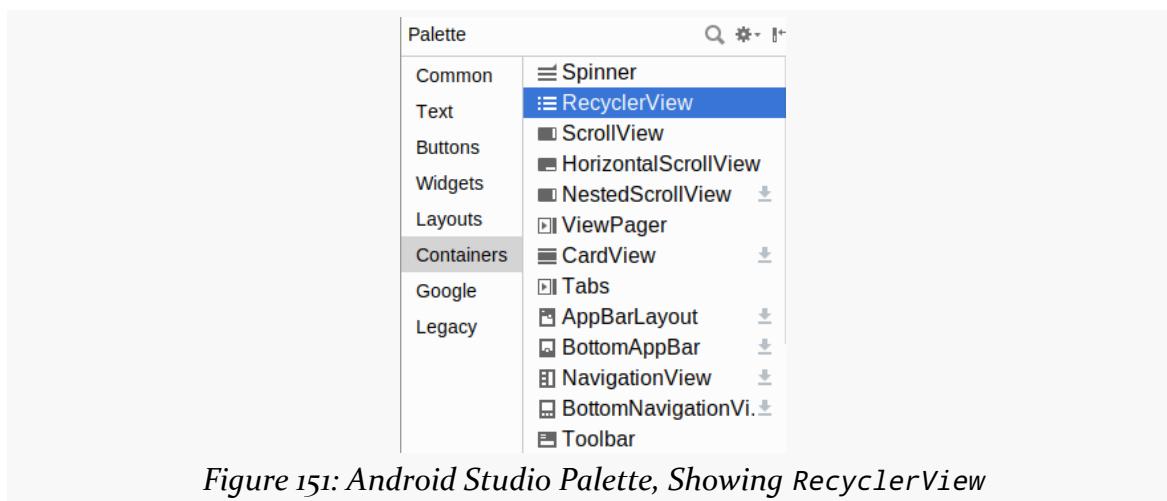


Figure 151: Android Studio Palette, Showing RecyclerView

The Row Layout

This project has a second layout resource, named `row`. Each row in our RecyclerView will be created from this `row` layout resource as a template:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:background="?android:attr/selectableItemBackground"
    android:clickable="true"
    android:focusable="true"
```

EMPLOYING RECYCLERVIEW

```
    android:padding="@dimen/content_padding">

    <View
        android:id="@+id/swatch"
        android:layout_width="@dimen/swatch_size"
        android:layout_height="@dimen/swatch_size"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <TextView
        android:id="@+id/label"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginStart="@dimen/label_start_margin"
        android:textAppearance="?android:attr/textAppearanceLarge"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toEndOf="@+id/swatch"
        app:layout_constraintTop_toTopOf="parent" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

(from [RecyclerViewBasics/src/main/res/layout/row.xml](#))

Once again, our root is a `ConstraintLayout`, this time with two children: a `TextView`... and a `View`.

Here, `View` is literally referring to `android.view.View`, the root class of all widgets and containers. `View` is not used all that often, as it does not render anything in the foreground: no text, no image, etc. However, it can have a background color, so the primary usage of `View` is for lines, boxes, and other shaded areas that have no children. In this case, we are using it for the color swatch seen on the left side of the rows, so we give it an ID of `swatch`.

The label `TextView` will be used to show the hexadecimal number that we randomly generate and are using for the color. Here, we use `android:textAppearance` as an attribute. That sets a bunch of things at once: text size, text style, etc. In particular, our value (`?android:attr/textAppearanceLarge`) is a reference to a standard text appearance from our activity's theme. We will explore themes more [later in the book](#). For now, take it on faith that `?android:attr/textAppearanceLarge` will give us a standard "large" text appearance.

The `ConstraintLayout` has three attributes that we have not used before:

- `android:background="?android:attr/selectableItemBackground"`

EMPLOYING RECYCLERVIEW

- android:clickable="true"
- android:focusable="true"

The android:clickable and android:focusable attributes control whether the user can interact with this element. By default, container classes like ConstraintLayout are non-interactive, so they ignore any touch or key events. By setting android:clickable="true" and android:focusable="true", we are indicating that this particular container *is* interactive, and so we should pay attention to touch and key events. This is typical of a container that serves as the root for an individual item in a RecyclerView.

In particular, those tie into android:background="?android:attr/selectableItemBackground". android:background says “this is what the background of this thing should be”. The default background for a container like ConstraintLayout is transparent. In this case, we are replacing that with something that we are pulling from our theme (?android:attr/selectableItemBackground), which will apply a standard “selectable” background. If you run the sample apps, and you tap on a row in the list, you will see a ripple effect — this comes from the “selectable” background.

The LayoutManager

As with our other samples, our activity is MainActivity, whether that is written in Java:

```
package com.commonsware.jetpack.samplerj.recyclerview;

import android.os.Bundle;
import java.util.ArrayList;
import java.util.List;
import java.util.Random;
import androidx.appcompat.app.AppCompatActivity;
import androidx.recyclerview.widget.DividerItemDecoration;
import androidx.recyclerview.widget.LinearLayoutManager;
import androidx.recyclerview.widget.RecyclerView;

public class MainActivity extends AppCompatActivity {
    private final Random random = new Random();

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

EMPLOYING RECYCLERVIEW

```
RecyclerView items = findViewById(R.id.items);
ColorAdapter adapter = new ColorAdapter(LayoutInflater.from(this));

adapter.submitList(buildItems());
items.setLayoutManager(new LinearLayoutManager(this));
items.addItemDecoration(
    new DividerItemDecoration(this, DividerItemDecoration.VERTICAL));
items.setAdapter(adapter);
}

private List<Integer> buildItems() {
    ArrayList<Integer> result = new ArrayList<>(25);

    for (int i = 0; i < 25; i++) {
        result.add(random.nextInt());
    }

    return result;
}
}
```

(from [RecyclerViewBasics/src/main/java/com/commonsware/jetpack/samplerj/recyclerview/MainActivity.java](#))

... or Kotlin:

```
package com.commonsware.jetpack.sampler.recyclerview

import android.os.Bundle
import androidx.appcompat.app.AppCompatActivity
import androidx.recyclerview.widget.DividerItemDecoration
import androidx.recyclerview.widget.LinearLayoutManager
import kotlinx.android.synthetic.main.activity_main.*
import java.util.*

class MainActivity : AppCompatActivity() {
    private val random = Random()

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        items.apply {
            layoutManager = LinearLayoutManager(this@MainActivity)
            addItemDecoration(
                DividerItemDecoration(this@MainActivity, DividerItemDecoration.VERTICAL)
            )
            adapter = ColorAdapter(LayoutInflater.from(this)).apply {
                submitList(buildItems())
            }
        }
    }
}
```

EMPLOYING RECYCLERVIEW

```
    private fun buildItems() = List(25) { random.nextInt() }
```

(from [RecyclerViewBasics/src/main/java/com/commonsware/jetpack/sampler/recyclerview/MainActivity.kt](#))

The UI is set up in `onCreate()`. Here, we configure the RecyclerView.

One thing that we need to teach the RecyclerView is what sort of look we want it to have overall. In this case, we want a vertically-scrolling list. For that, we can use a LinearLayoutManager.

RecyclerView uses a LayoutManager to control the overall layout of items within its scrollable area. LinearLayoutManager lays those items out in a list. By default, that list is vertical.

(despite the name, LinearLayoutManager is not related to LinearLayout, except in the most general sense)

So, we create an instance of LinearLayoutManager and associate it with the RecyclerView via `setLayoutManager()`.

If you do not want a vertically-scrolling list, you could use:

- GridLayoutManager, which implements a two-dimensional vertically-scrolling list, with rows consisting of multiple cells
- StaggeredGridLayoutManager, which implements a “staggered grid”, which has columns of cells, but where the cells do not have to all have the same size

In addition, it is possible to create your own RecyclerView.LayoutManager, or use ones from third-party libraries.

The Divider

By default, a RecyclerView just shows its items, and nothing else. However, with a vertically-scrolling list, frequently we want some sort of divider between the rows in the list. Sometimes, that is not necessary, as we have stuff in the items themselves that visually distinguish one from another. The rest of the time, we will want the help of RecyclerView to handle this.

RecyclerView delegates that sort of work to one or more ItemDecoration objects. The primary one offered by the RecyclerView library is DividerItemDecoration,

EMPLOYING RECYCLERVIEW

which draws a thin line for you.

DividerItemDecoration can be used both for the normal vertically-scrolling list and for the less-common horizontally-scrolling list. Hence, we need to tell DividerItemDecoration which way to draw the divider line. For a vertically-scrolling list, we need a divider configured for such a list, so we pass DividerItemDecoration.VERTICAL to the DividerItemDecoration constructor when we set it up.

You can then add these ItemDecoration objects to a RecyclerView via addItemDecoration() calls.

The Data

The data that we want to depict in the list is a roster of 25 random numbers. To that end, we have a buildItems() Java method:

```
private List<Integer> buildItems() {  
    ArrayList<Integer> result = new ArrayList<>(25);  
  
    for (int i = 0; i < 25; i++) {  
        result.add(random.nextInt());  
    }  
  
    return result;  
}
```

(from [RecyclerViewBasics/src/main/java/com/commonsware/jetpack/samplerj/recyclerview/MainActivity.java](#))

... or Kotlin function:

```
private fun buildItems() = List(25) { random.nextInt() }
```

(from [RecyclerViewBasics/src/main/java/com/commonsware/jetpack/sampler/recyclerview/MainActivity.kt](#))

In each case, we create a list of 25 random numbers, using Random.nextInt() as the source of the number. buildItems() is called from onCreate() and will be used to populate our RecyclerView.Adapter.

The ViewHolder

Each RecyclerView is associated with one or more implementations of RecyclerView.ViewHolder. Each instance of a ViewHolder class wraps one piece of

EMPLOYING RECYCLERVIEW

data (in our case, one random number) and ties it to one visual representation of that data. However, courtesy of recycling, a ViewHolder will get reused. As the user scrolls, and items scroll off the screen, their associated ViewHolder objects will need to be associated with new pieces of data that are to be scrolled onto the screen.

Our particular implementation is ColorViewHolder, both in Java:

```
package com.commonsware.jetpack.samplerj.recyclerview;

import android.view.View;
import android.widget.TextView;
import android.widget.Toast;
import androidx.recyclerview.widget.RecyclerView;

class ColorViewHolder extends RecyclerView.ViewHolder {
    private final View swatch;
    private final TextView label;

    ColorViewHolder(View row) {
        super(row);

        swatch = row.findViewById(R.id.swatch);
        label = row.findViewById(R.id.label);
        row.setOnClickListener(
            v -> Toast.makeText(label.getContext(), label.getText(),
                Toast.LENGTH_LONG).show());
    }

    void bindTo(Integer color) {
        label.setText(label.getContext().getString(R.string.label_template, color));
        swatch.setBackgroundColor(color);
    }
}
```

(from [RecyclerViewBasics/src/main/java/com/commonsware/jetpack/samplerj/recyclerview/ColorViewHolder.java](#))

... and Kotlin:

```
package com.commonsware.jetpack.sampler.recyclerview

import android.view.View
import android.widget.TextView
import android.widget.Toast
import androidx.recyclerview.widget.RecyclerView
import kotlinx.android.synthetic.main.row.view.*

class ColorViewHolder(val row: View) : RecyclerView.ViewHolder(row) {
    private val swatch: View = row.swatch
    private val label: TextView = row.label

    init {
        row.setOnClickListener { _ ->
```

EMPLOYING RECYCLERVIEW

```
        Toast.makeText(label.context, label.text, Toast.LENGTH_LONG).show()
    }
}

fun bindTo(color: Int) {
    label.text = label.context.getString(R.string.label_template, color)
    swatch.setBackgroundColor(color)
}
}
```

(from [RecyclerViewBasics/src/main/java/com/commonsware/jetpack/sampler/recyclerview/ColorViewHolder.kt](#))

The Constructor

Something outside of the ViewHolder is responsible for setting up the visual representation and providing the View for it to our constructor. We will see how that works [a bit later in this chapter](#). Our ColorViewHolder is expecting this View to be the root of our row layout resource, so we can work with its contents.

The job of our constructor is to find the individual widgets that we need and configure them for our use.

The Java code uses `findViewById()` to access the swatch and label widgets. The Kotlin code uses Kotlin synthetic accessors (`import kotlinx.android.synthetic.main.row.view.*`) to access the contents of the row and assign them to individual properties.

The Toast

We also call `setOnClickListener()` on the row itself, to find out when it is clicked. There, to provide feedback to the user, we show a Toast.

A Toast is a transient message, meaning that it displays and disappears on its own without user interaction. Moreover, it does not take focus away from the currently-active Activity, so if the user is busy writing the next Great Programming Guide, they will not have keystrokes be “eaten” by the message.

Since a Toast is transient, you have no way of knowing if the user even notices it. You get no acknowledgment from them, nor does the message stick around for a long time to pester the user. Hence, the Toast is mostly for advisory messages, ones that if the user misses them, no harm will come.

EMPLOYING RECYCLERVIEW

Making a Toast is fairly easy. The `Toast` class offers a static `makeText()` method that accepts a String (or string resource ID) and returns a `Toast` instance. The `makeText()` method also needs the Activity (or other Context) plus a duration. The duration is expressed in the form of the `LENGTH_SHORT` or `LENGTH_LONG` constants to indicate, on a relative basis, how long the message should remain visible. Once your `Toast` is configured, call its `show()` method, and the message will be displayed.

The Binding

Other than needing to use the base class of `RecyclerView.ViewHolder`, there is no other particular protocol that is mandated between the adapter and the view holder. However, at some point, our `ViewHolder` needs to be handed the data that it is supposed to represent in these widgets.

For that, `ColorViewHolder` has `bindTo()`. It takes a color integer and pours it into the label and swatch widgets.

For the label, we use `getString()`, where our string resource has a placeholder to format the number:

```
<string name="label_template">#%1$08x</string>
```

(from [RecyclerViewBasics/src/main/res/values/strings.xml](#))

That cryptic placeholder (`#%1$08x`) means that we want to format the first parameter (`%1`) as an eight-digit hexadecimal value (`08x`).

For the color swatch, we call `setBackgroundColor()`. This is a method available on any `View` that can be used to set its background color to a particular value. In our case, we are using a randomly-generated color, not a color resource, so we can just pass in the color integer.

The Adapter

We still need to tell the `RecyclerView` what to display. That is handled by an implementation of `RecyclerView.Adapter`. Here, “adapter” refers to the adapter pattern: an adapter takes data and adapts it for some other role. In our case, an `Adapter` takes some collection of data and uses it to define the visual representation of that data, in the form of `View` and `ViewGroup` objects.

Our Adapter — named `ColorAdapter` — will use the row layout that we defined

EMPLOYING RECYCLERVIEW

earlier to define the visual representation of each piece of data. ColorAdapter uses our ColorViewHolder to manage the actual widgets.

We have two implementations of ColorAdapter, in Java:

```
package com.commonsware.jetpack.samplerj.recyclerview;

import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import androidx.annotation.NonNull;
import androidx.recyclerview.widget.DiffUtil;
import androidx.recyclerview.widget.ListAdapter;

class ColorAdapter extends ListAdapter<Integer, ColorViewHolder> {
    private final LayoutInflater inflater;

    ColorAdapter(LayoutInflater inflater) {
        super(DIFF_CALLBACK);
        this.inflater = inflater;
    }

    @NonNull
    @Override
    public ColorViewHolder onCreateViewHolder(@NonNull ViewGroup parent,
                                              int viewType) {
        View row = inflater.inflate(R.layout.row, parent, false);

        return new ColorViewHolder(row);
    }

    @Override
    public void onBindViewHolder(@NonNull ColorViewHolder holder, int position) {
        holder.bindTo(getItem(position));
    }

    private static final DiffUtil.ItemCallback<Integer> DIFF_CALLBACK =
        new DiffUtil.ItemCallback<Integer>() {
            @Override
            public boolean areItemsTheSame(@NonNull Integer oldColor, @NonNull Integer newColor) {
                return oldColor.equals(newColor);
            }

            @Override
            public boolean areContentsTheSame(@NonNull Integer oldColor, @NonNull Integer newColor) {
                return areItemsTheSame(oldColor, newColor);
            }
        };
}
```

(from [RecyclerViewBasics/src/main/java/com/commonsware/jetpack/samplerj/recyclerview/ColorAdapter.java](#))

... and Kotlin:

```
package com.commonsware.jetpack.sampler.recyclerview
```

EMPLOYING RECYCLERVIEW

```
import android.view.LayoutInflater
import android.view.ViewGroup
import androidx.recyclerview.widget.DiffUtil
import androidx.recyclerview.widget.ListAdapter

class ColorAdapter(private val inflater: LayoutInflater) :
    ListAdapter<Int, ColorViewHolder>(ColorDiffer) {

    override fun onCreateViewHolder(
        parent: ViewGroup,
        viewType: Int
    ): ColorViewHolder {
        return ColorViewHolder(inflater.inflate(R.layout.row, parent, false))
    }

    override fun onBindViewHolder(holder: ColorViewHolder, position: Int) {
        holder.bindTo(getItem(position))
    }

    private object ColorDiffer : DiffUtil.ItemCallback<Int>() {
        override fun areItemsTheSame(oldColor: Int, newColor: Int): Boolean {
            return oldColor == newColor
        }

        override fun areContentsTheSame(oldColor: Int, newColor: Int): Boolean {
            return areItemsTheSame(oldColor, newColor)
        }
    }
}
```

(from [RecyclerViewBasics/src/main/java/com/commonsware/jetpack/sampler/recyclerview/ColorAdapter.kt](#))

The Base Class

When we created `ColorViewHolder`, we directly extended `RecyclerView.ViewHolder`. You can do the same thing with `RecyclerView.Adapter`, having your class (e.g., `ColorAdapter`) extend it directly. In this case, we are using `ListAdapter`, a supplied partial implementation of `RecyclerView.Adapter` that knows how to work with lists of data.

NOTE: There are *two* things named `ListAdapter` in the Android SDK. We are using `androidx.recyclerview.widget.ListAdapter`, which works with `RecyclerView`. The `android.widget.ListAdapter` interface is designed for use with [the older AdapterView family of widgets](#), such as `ListView`. Make sure that you use the right one, as otherwise you will get lots of strange compile errors.

EMPLOYING RECYCLERVIEW

ListAdapter uses generics and takes two data types:

- The type of data for individual elements in the list to be shown (in our case, a Java Integer or Kotlin Int)
- The RecyclerView.ViewHolder class that we want to use for the items in the list (in our case, ColorViewHolder)

The Constructor and the “Differ”

The ListAdapter constructor requires an implementation of a DiffUtil.ItemCallback object.

Part of what ListAdapter does for us is help deal with changes to our list of data. This app, as it stands, only shows one set of random numbers in the list. But, suppose we had a button that allowed the user to add more numbers. We would need our RecyclerView.Adapter to be able to show both the old numbers and the new numbers.

ListAdapter has all of the smarts to handle that for us as efficiently as possible. However, ListAdapter knows nothing about our data and our visual representation of that data, and knowing more about those things helps with efficiency. The DiffUtil.ItemCallback is our way of teaching ListAdapter more about our data. Specifically, a DiffUtil.ItemCallback has two Java methods or Kotlin functions:

- areItemsTheSame() takes in two pieces of data from our lists (old and changed) and needs to return true if they are the actual same thing
- areContentsTheSame() takes in two pieces of data and returns true if their visual representation will be the same when the data is rendered on the screen

If our ListAdapter were adapting the items in an online shopping cart, we would return true from areItemsTheSame() if both objects are really the same underlying cart entry. But, suppose the user had more than one of some particular product in the cart, such as three boxes of laundry detergent. areItemsTheSame() might return false for some pair, as the first box is not the same box as the second box. However, areContentsTheSame() might return true, as the visual representation might be the same (e.g., a thumbnail image of the laundry detergent box).

In our case, not only is the data not changing, but it is very simple and distinct, so we can use content equality for both areItemsTheSame() and areContentsTheSame(). So, we have a ColorDiffer that does just that, either in the

EMPLOYING RECYCLERVIEW

form of a static Java class or a singleton Kotlin object. We use `ColorDiffer` in our `ColorAdapter` constructor, so now `ListAdapter` knows about how to compare our colors.

`onCreateViewHolder()`

Any `RecyclerView.Adapter` needs to know how to do two other things:

1. Create the `ViewHolder` that we want, including setting up its UI; and
2. Bind data from our collection to `ViewHolder` instances as they are displayed on the screen

`onCreateViewHolder()` handles the first of these. It creates instances of `ColorViewHolder` and returns them.

However, `ColorViewHolder` wants the `View` that represents the UI for the row in our list. That UI is defined in the `row` layout resource. We need some way to get a `View` for a layout resource. All our prior uses of layout resources were for activities, and we just passed the resource ID to `setContentView()`. That will not work here.

So, our `ColorAdapter` constructor takes in a `LayoutInflater` object. `LayoutInflater` knows how to “inflate” layout resources. In Android, “inflate” means:

- Walk a tree of XML elements in an XML resource
- Create Java objects for each element in that tree
- Stitch those objects together into their own tree structure, mirroring the tree defined in the XML
- Return the Java object representing the root of the tree

Given that our `row` layout is:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:background="?android:attr/selectableItemBackground"
    android:clickable="true"
    android:focusable="true"
    android:padding="@dimen/content_padding">

    <View
```

EMPLOYING RECYCLERVIEW

```
    android:id="@+id/swatch"
    android:layout_width="@dimen/swatch_size"
    android:layout_height="@dimen/swatch_size"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />

<TextView
    android:id="@+id/label"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginStart="@dimen/label_start_margin"
    android:textAppearance="?android:attr/textAppearanceLarge"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toEndOf="@+id/swatch"
    app:layout_constraintTop_toTopOf="parent" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

(from [RecyclerViewBasics/src/main/res/layout/row.xml](#))

...then a call to `inflate()` on a `LayoutInflater`, supplying `R.layout.row` as the first parameter, should return a `ConstraintLayout` object that holds onto a `TextView` and a `View`.

That is what `onCreateViewHolder()` does: calls `inflate()`, gets the root `View`, passes that to the `ColorViewHolder()` constructor, and returns the `ColorViewHolder` instance.

`LayoutInflater` has a few `inflate()` methods, taking different parameters. They all do the same basic thing: “inflate” the layout into a `View` hierarchy. The particular `inflate()` that you will use most often takes three parameters:

- The resource ID of the layout that you want to inflate
- The `ViewGroup` that the widgets in that layout will eventually be added to
- `false` to say “but please do not add them to the parent right away”
(something else will when appropriate, `RecyclerView` and `LinearLayoutManager` in this case)

onBindViewHolder()

`onBindViewHolder()` will be called when `RecylerView` wants to show one of our pieces of data. We are passed the o-based index into our collection of data representing what `RecyclerView` wants, and we are passed a `ViewHolder` (created by

EMPLOYING RECYCLERVIEW

onCreateViewHolder() previously) to use for the visual representation.

ListAdapter gives us a getItem() method that we can use to get our color for a given position. We then just call bindTo() on the ColorViewHolder, and ColorViewHolder takes it from there.

Applying the ColorAdapter

To make use of ColorAdapter, we first need to give it some colors to display. Subclasses of ListAdapter have a submitList() method that you can use for that. You provide a List of your data, such as the results of our buildItems() call.

Then, to have the colors show up, you call setAdapter() on the RecyclerView, handing it your RecyclerView.Adapter instance.

We do both of these things in onCreate() of MainActivity as part of the overall setup:

```
@Override  
protected void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.activity_main);  
  
    RecyclerView items = findViewById(R.id.items);  
    ColorAdapter adapter = new ColorAdapter(LayoutInflater.from(this));  
  
    adapter.submitList(buildItems());  
    items.setLayoutManager(new LinearLayoutManager(this));  
    items.addItemDecoration(  
        new DividerItemDecoration(this, DividerItemDecoration.VERTICAL));  
    items.setAdapter(adapter);  
}
```

(from [RecyclerViewBasics/src/main/java/com/commonsware/jetpack/samplerj/recyclerview/MainActivity.java](#))

```
override fun onCreate(savedInstanceState: Bundle?) {  
    super.onCreate(savedInstanceState)  
    setContentView(R.layout.activity_main)  
  
    items.apply {  
        layoutManager = LinearLayoutManager(this@MainActivity)  
        addItemDecoration(  
            DividerItemDecoration(this@MainActivity, DividerItemDecoration.VERTICAL)  
        )  
        adapter = ColorAdapter(layoutInflater).apply {  
            submitList(buildItems())  
        }  
    }  
}
```

EMPLOYING RECYCLERVIEW

```
}
```

(from [RecyclerViewBasics/src/main/java/com/commonsware/jetpack/sampler/recyclerview/MainActivity.kt](#))

Hey, What About ListView?

If you read about Android from older sources, you will see a lot of discussion of `ListView`.

`ListView` is a subclass of `AdapterView`. The `AdapterView` family of classes is the precursor to `RecyclerView`. Like `RecyclerView`, they display collections of data. And, like `RecyclerView`, they use an adapter to convert your data into visual representations to display. However, on the whole, the `AdapterView` widgets were less configurable and less powerful than is `RecyclerView`.

Almost every `AdapterView` subclass that is part of the Android SDK could be replaced by `RecyclerView`. `ListView` and `GridView` can be replaced very easily. `Gallery` would take a bit more work. Many third-party libraries exist for implementing tree structures — parents with children, like a directory tree — in `RecyclerView`, to replace `ExpandableListView`.

The main exception is `Spinner`, which implements a drop-down list. `RecyclerView` cannot reproduce this sort of widget directly, though it could be used as part of some larger replacement.

Licensed solely for use by Patrocinio Rodriguez

Coping with Configurations

Devices sometimes change while users are using them, in ways that our application will care about:

- The user might rotate the screen from portrait to landscape, or vice versa
- The user might switch to a different language via the Settings application, returning to our running application afterwards
- It might become dark, suggesting that we should be in some sort of “night mode”
- And so on

In all of these cases, it is likely that we will want to change what resources we use. For example, our layout for a portrait screen may be too tall to use in landscape mode, so we would want to substitute in some other layout.

This chapter will explore how to provide alternative resources for these different scenarios — called “configuration changes” — and will explain what happens to our activities when the user changes the configuration while we are in the foreground.

What's a Configuration? And How Do They Change?

Different pieces of Android hardware can have different capabilities, such as:

- Different screen sizes
- Different screen densities (dots per inch)
- Different number and capabilities of cameras
- Different mix of radios (GSM? CDMA? GPS? Bluetooth? WiFi? NFC?)

COPING WITH CONFIGURATIONS

- something else?)
- And so on

Some of these, in the eyes of the core Android team, might drive the selection of resources, like layouts or drawables. Different screen sizes might drive the choice of layout. Different screen densities might drive the choice of drawable (using a higher-resolution image on a higher-density device). These are considered part of the device’s “configuration”.

Other differences — ones that do not drive the selection of resources — are not part of the device’s configuration but merely are “features” that some devices have and other devices do not. For example, cameras and Bluetooth and WiFi are features. The core Android team does not expect that you will want different resources based on whether or not the device has a front-facing camera.

Most of the hardware features that drive a configuration might change on the fly. For example, the user can rotate the device and switch from portrait to landscape, while our app is still running. Somehow, though, we may need to switch to landscape-friendly resources from the portrait-friendly resources that we started with. When a configuration switches to something else on the fly, that is a “configuration change”, and Android provides special support for such events to help developers adjust their applications to match the new configuration.

Configurations and Resource Sets

Somehow, we need to be able to tell Android:

- Some resources are to be used for a certain type of configuration, such as “large screens” or “high-density screens” or “uses English”
- Some resources are for some other configuration, for as many different configuration aspects as we wish to support
- Some resources are the default and are valid for any configuration

The way Android currently handles this is by having multiple resource directories, with the criteria for each embedded in their names.

For example, suppose that you want to support multiple languages. You will need to choose some default language, one that will be used if your app winds up on a device whose locale is set to a language that you do not support. For example, you might set your default language to be US English. In that case, your US English

COPING WITH CONFIGURATIONS

strings would go into `res/values/strings.xml`. If you also wanted to offer a translation in Spanish, you would add a `res/values-es/` directory, where es is the [ISO 639-1](#) two-letter code for Spanish. In `res/values-es/`, you would put your Spanish strings. At runtime, Android will choose which string to use, for a given string resource ID (e.g., `@string/app_name`), based on the device's locale(s) and the various translations of that string that you provide.

This, therefore, is the bedrock resource set strategy: have a complete set of resources in the default directory (e.g., `res/values/`), and override those resources in other resource sets tied to specific configurations as needed (e.g., `res/values-es/`).

Implementing Resource Sets

Ideally, that strategy would be sufficient. Unfortunately, things are not quite that simple. So, let's walk through some common configurations and associated resource sets, to see how we can set them up.

Language

As noted above, you can have different resource sets for different languages, by adding the language code as a suffix on the resource directory name. Usually, this is just applied to `res/values/` (e.g., `res/values-es/`), as usually the only things that change in an app based on language are string resources. In principle, though, you could use such language suffixes on any resource directory (e.g., `res/raw/` for English-language audio clips and `res/raw-es/` for Spanish-language audio clips).

However, languages often vary by region:

- US English has much in common with UK English, but not everything (e.g., “color” versus “colour”)
- French as spoken in France differs somewhat from French as spoken in the Quebec province of Canada
- Spanish as spoken in Spain differs somewhat from Spanish as spoken in Mexico
- And so on

To address this, Android supports two different systems for regional localization.

The classic approach was to add an [ISO-3166-1 alpha-2](#) code after the language code and prefixed by r (for “region”). For example, US English is `res/values-en-rUS/`,

COPING WITH CONFIGURATIONS

while UK English is `res/values-en-rGB/` (“GB” for “Great Britain”).

Android 7.0 and higher support [BCP 47 language and locale values](#) as an alternative system. Those get an overall `b+` prefix, to distinguish them from other resource set qualifiers. In the BCP 47 approach, English is `res/values-b+en/`, while US English is `res/values-b+en+US/` and UK English is `res/values-b+en+GB/`. However, this system has not been as popular, in part because we have so much history with the original system.

In either case, string resource sets are additive. You do not need to have a full set of US English strings *and* a full set of UK English strings to support full localization, because many of those strings will be in common. Plus, you can choose one of the regions to be the default for a particular language, so you only need resource sets and overrides for the specific strings that you need for the specific locales that concern you.

So, for example, you could have:

- US English strings in `res/values/` as the overall default for your app
- UK English strings in `res/values-en-rGB/`, for the handful of strings that you need where the British spelling or term differs from the US version
- Mexican Spanish strings in `res/values-es/`
- Spain Spanish strings in `res/values-es-rES/`, for the handful of strings that you need where the Spain localization differs from the Mexican localization
- And so on

Note that Android Studio has a translations editor to help you manage your string resources for your default language and whatever translations you are going to include in your app.

Screen Size and Orientation

The resource sets that get the most attention are those for screen size (and, secondarily, screen orientation).

Android “borrows a page” from Web development. Web content nowadays uses CSS media queries to have different layout rules based on screen size, typically screen width. Similarly, Android allows you to set up different resource sets to use for screens with varying sizes, where you choose the size. And, like CSS media queries, you can choose the dividing lines between different rules — you are not limited to some fixed set of size buckets.

COPING WITH CONFIGURATIONS

There are three families of qualifiers that you can use:

- `-wNNNdp`, for some number NNN, means “use these resources when the current width of the screen is NNN density-independent pixels (`dp`) or larger”
- `-hNNNdp`, for some number NNN, means “use these resources when the current height of the screen is NNN `dp` or larger”
- `-swNNNdp` means “use these resources when the *smallest width* of the screen, in any orientation, is NNN `dp` or larger”

The difference between `-wNNNdp` and `-swNNNdp` is that `-wNNNdp` is based on the *current width*, and the current width of a device depends on its orientation. `-swNNNdp`, by contrast, cares only what the length of the shorter screen dimension is (“smallest width”) and therefore is independent of orientation.

So, you could have:

- `res/layout/` for layout resources aimed at smaller screens
- `res/layout-w640dp/` for layout resources aimed at screens whose current width is 4" (640`dp`) or larger
- `res/layout-w1120dp/` for layout resources aimed at screens whose current width is 7" (1120`dp`) or larger

Hey, What About `res/layout-large`?

You will see references to other size-related resource set qualifiers: `-small`, `-normal`, `-large`, and `-xlarge`. This was the original system used by Android. However, it was not very flexible. Android 3.2 added the system described above, and so most modern Android app development should use that system.

Hey, What About `res/layout-land`?

Similarly, you will see references to a `-land` suffix to be used for landscape resources. So `res/layout/` would be used for portrait layouts and `res/layout-land/` would be used for landscape layouts.

Frequently, though, we do not care about the actual orientation, only the width (or *maybe* the height). For example, many Web sites that use CSS media queries apply different CSS rules based on the viewport width. While most computer monitors are landscape, some can be rotated to portrait — the CSS media queries do not care and only differentiate based on width.

COPING WITH CONFIGURATIONS

As a result, while `-1` exists and can be used, you will not see it quite as much as you might have in the early days of Android app development.

API Level

Earlier in the book, we saw `res/drawable-v24/` as a resource directory. This follows the same resource set system, where `-vNN`, for some value of `NN`, means “use these resources on devices running API Level `NN` or higher”. So, `res/drawable-v24/` will be used on API Level `24+` devices and will be ignored on older devices.

Screen Density

Similarly, we saw various `mipmap` directories, like `res/mipmap-hdpi/` and `res/mipmap-xhdpi/`. This too follows the same resource set system, where `-hdpi` says “these resources were optimized for HDPI screens”, while `-xhdpi` says “these resources were optimized for XHDPI screens”.

The Full Roster

There are *lots* of different resource set qualifiers.

The full roster can be found in [the Android developer documentation](#). In particular, “Table 2” in that section lists all of the current candidates.

Note that resource set qualifiers get added from time to time. For example, the `-wNNNdp` family of screen size qualifiers was added in API Level 13 (Android 3.2). The table points out what version of Android added support for those qualifiers. Older devices will not crash if they encounter those qualifiers, simply ignoring them instead.

Resource Set Rules

Where things start to get complicated is when you need to use multiple disparate criteria for your resources.

For example, suppose that you have drawable resources that are locale-dependent, because they tie into local iconography (e.g., roadside traffic signs). You might want to have resource sets of drawables tied to language, so you can substitute in different images for different locales. However, you might also want to have those images vary by density, using higher-resolution images on higher-density devices, so the images

COPING WITH CONFIGURATIONS

all come out around the same physical size.

To do that, you would wind up with directories with multiple resource set qualifiers, such as:

- res/drawable-ldpi/
- res/drawable-mdpi/
- res/drawable-hdpi/
- res/drawable-xhdpi/
- res/drawable-en-rUK-ldpi/
- res/drawable-en-rUK-mdpi/
- res/drawable-en-rUK-hdpi/
- res/drawable-en-rUK-xhdpi/
- And so on

(with the default language being, say, US English, using an American set of icons)

Once you get into these sorts of situations, though, a few rules come into play, such as:

- The configuration options (e.g., -en) have a particular order of precedence, and they must appear in the directory name in that order. The [Android documentation](#) outlines the specific order in which these options can appear. For the purposes of this example, screen density is more important than language.
- There can only be one value of each configuration option category per directory.

Given that you can have N different definitions of a resource, how does Android choose the one to use?

First, Android tosses out ones that are specifically invalid. So, for example, if the language of the device is -ru, Android will ignore resource sets that specify other languages (e.g., -zh). The exceptions to this are density qualifiers and screen size qualifiers — we will get to those exceptions later.

Then, Android chooses the resource set that has the desired resource and has *the most important distinct qualifier*. Here, by “most important”, we mean the one that appears left-most in the directory name, based upon the directory naming rules discussed above. And, by “distinct”, we mean where no other resource set has that qualifier.

COPING WITH CONFIGURATIONS

If there is no specific resource set that matches, Android chooses the default set — the one with no suffixes on the directory name (e.g., `res/layout/`).

With those rules in mind, let's look at some scenarios, to cover the base case plus the aforementioned exceptions.

Scenario #1: Something Simple

Let's suppose that we have a `main.xml` layout resource in:

- `res/layout-w640dp/`
- `res/layout/`

When we call `setContentView(R.layout.main)`, Android will choose the `main.xml` in `res/layout-w640dp/` if the device's current width is 640dp or larger. That particular resource set is valid in that case, and it has the most important distinct qualifier (`-w640dp`). If the device has a smaller current width, though, the `res/layout-w640dp/` resource set does not qualify, and so it is ignored. That leaves us with `res/layout/`, so Android uses that `main.xml` version.

Scenario #2: Disparate Resource Set Categories

It is possible, though bizarre, for you to have a project with `main.xml` in:

- `res/layout-en/`
- `res/layout-w640dp/`
- `res/layout/`

In this case, if the device's locale is set to be English, Android will choose `res/layout-en/`, regardless of the orientation of the device. That is because `-en` is a more important resource set qualifier — “Language and region” appears higher in the “Table 2. Configuration qualifier names” from the Android documentation than does “Available width” (for `-w640dp`). If the device is not set for English, though, Android will toss out that resource set, at which point the decision-making process is the same as in Scenario #1 above.

Scenario #3: Multiple Qualifiers

Now let's envision a project with `main.xml` in:

- `res/layout-en/`

COPING WITH CONFIGURATIONS

- `res/layout-w640dp-v21/`
- `res/layout/`

You might think that `res/layout-w640dp-v21/` would be a higher-priority choice than `res/layout-en/`, as it is more specific, matching on two resource set qualifiers versus the one or none from the other resource sets.

(in fact, the author of this book thought this was the choice for many years)

In this case, though, language is more important than either screen orientation or Android API level, so the decision-making process is similar to Scenario #2 above: Android chooses `res/layout-en/` for English-language devices, `res/layout-w640dp-v21/` for 4"-wide API Level 21+ devices, or `res/layout/` for everything else.

Scenario #4: Multiple Qualifiers, Revisited

Let's change the resource mix, so now we have a project with `main.xml` in:

- `res/layout-w640dp-night/`
- `res/layout-w640dp-v21/`
- `res/layout/`

Here, while `-w640dp` is the most important resource set qualifier, it is not distinct — we have more than one resource set with `-w640dp`. Hence, we need to check which is the next-most-important resource set qualifier. In this case, that is `-night`, as night mode is a more important category than is Android API level, and so Android will choose `res/layout-land-night/` if the device is in night mode. Otherwise, it will choose `res/layout-w640dp-v21/` if the device is running API Level 21 or higher. If the device is not in night mode and is not running API Level 21 or higher — or if the device is less than 4" wide at present — Android will go with `res/layout/`.

Scenario #5: Screen Density

Now, let's look at the main exception to the rules: screen density.

Android will *always* accept a resource set that contains a screen density, *even if it does not match the density of the device*. If there is an exact density match, of course, Android uses it. Otherwise, it will use what it feels is the next-best match, based upon how far off it is from the device's actual density and whether the other density is higher or lower than the device's actual density.

COPING WITH CONFIGURATIONS

The reason for this is that for drawable and mipmap resources, Android will downsample or upsample the image automatically, so the drawable will appear to be the right size, even though you did not provide an image in that specific density.

The catch is two-fold:

1. Android applies this logic to all resources, not just drawables and mipmaps, so even if there is no exact density match on, say, a layout, Android will still choose a resource from another density bucket for the layout
2. As a side-effect of the previous bullet, if you include a density resource set qualifier, Android will ignore any lower-priority resource set qualifiers (unless there are multiple directories with the same density resource set qualifier, in which case the lower-priority qualifiers serve as the “tiebreaker”)

So, now let's pretend that our project has `main.xml` in:

- `res/layout-mdpi/`
- `res/layout-nonav/`
- `res/layout/`

Android will choose `res/layout-mdpi/`, even for `-hdpi` devices that do not have a “non-touch navigation method”. While `-mdpi` does not match `-hdpi`, Android will still choose `-mdpi`. If we were dealing with drawable or mipmap resources, Android would upsample the `-mdpi` image.

Activity Lifecycles

The user taps a launcher icon to start our activity. Then, the user rotates the screen, causing a configuration change. Later, the user presses BACK to return to the launcher.

While those things were going on, Android was calling lifecycle methods on our activity, to let us know what is going on.

An activity, generally speaking, is in one of four states at any point in time:

1. *Active*: the activity was started by the user, is running, and is in the foreground. This is what you are used to thinking of in terms of your activity's operation.
2. *Paused*: the activity was started by the user, is running, and is visible, but

COPING WITH CONFIGURATIONS

another activity is overlaying part of the screen. During this time, the user can see parts of your activity but may not be able to interact with it.

3. *Stopped*: the activity was started by the user, is running, but it is completely hidden by other activities that have been launched or switched to.
4. *Destroyed*: the activity was destroyed, perhaps due to the user pressing the BACK button.

Android will call so-called “lifecycle methods” on your activity as the activity transitions between these four states.

The [Activity developer documentation](#) usually provides some variation of this diagram:

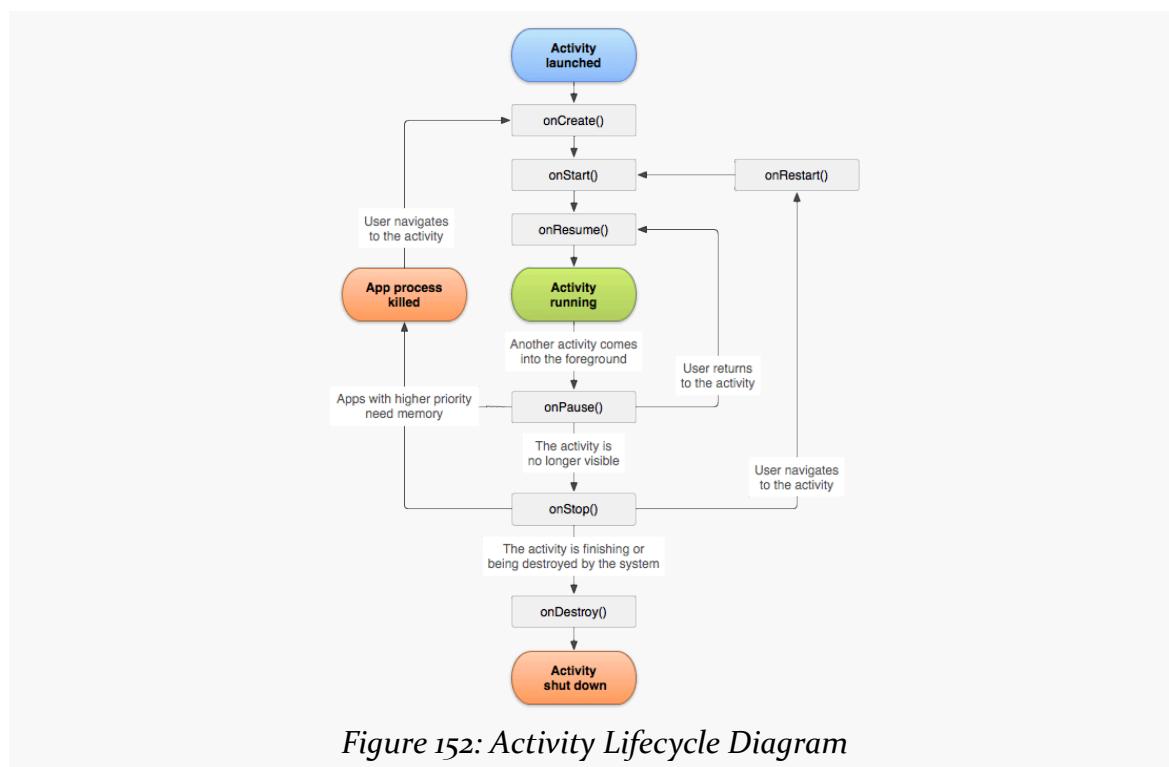


Figure 152: Activity Lifecycle Diagram

(the above image is reproduced from work created and [shared by the Android Open Source Project](#) and used according to terms described in [the Creative Commons 2.5 Attribution License](#))

This diagram shows the various lifecycle methods and the trigger events that cause them.

COPING WITH CONFIGURATIONS

The following sections outline these lifecycle methods and their roles. We will see them in use starting in [the next chapter](#).

onCreate() and onDestroy()

We have been implementing `onCreate()` in all of our `Activity` subclasses in all the examples. This will get called in two primary situations:

- When the activity is launched by the user, such as from a launcher icon, `onCreate()` will be invoked with a null parameter
- If the activity undergoes a configuration change, by default your activity will be re-created and `onCreate()` will be called

In general, `onCreate()` is where you initialize your user interface and set up anything that needs to be done once, regardless of how the activity gets used.

On the other end of the lifecycle, `onDestroy()` may be called when the activity is shutting down, such as because the activity called `finish()` (which “finishes” the activity) or the user presses the BACK button. Hence, `onDestroy()` is mostly for cleanly releasing resources you obtained in `onCreate()` (if any), plus making sure that anything you started up outside of lifecycle methods gets stopped, such as background threads.

Bear in mind, though, that `onDestroy()` may not be called. This would occur in a few circumstances:

- You crash with an unhandled exception
- The user force-stops your application, such as through the Settings app
- Android has an urgent need to free up RAM (e.g., to handle an incoming phone call), wants to terminate your process, and cannot take the time to call all the lifecycle methods

Hence, `onDestroy()` is very likely to be called, but it is not guaranteed.

Also, bear in mind that it may take a long time for `onDestroy()` to be called. It is called quickly if the user presses BACK to finish the foreground activity. If, however, the user presses HOME to bring up the home screen, your activity is not immediately destroyed. `onDestroy()` will not be called until Android does decide to gracefully terminate your process, and that could be seconds, minutes, or hours later.

COPING WITH CONFIGURATIONS

onStart(), onRestart(), and onStop()

An activity can come to the foreground either because it is first being launched, or because it is being brought back to the foreground after having been hidden (e.g., by another app's activity).

The `onStart()` method is called in either of those cases. The `onRestart()` method is called in the case where the activity had been stopped and is now restarting.

Conversely, `onStop()` is called when the activity is about to be stopped. Primarily, in `onStop()`, you clean up anything you set up in `onStart()`.

Once started, your activity is visible, at least partially. Anything that should be happening while your activity is visible should be set up in `onCreate()` or `onStart()` and cleaned up in `onStop()` (for `onStart()`) or `onDestroy()` (for `onCreate()`).

onPause() and onResume()

The `onResume()` method is called just before your activity comes to the foreground, either after being initially launched, being restarted from a stopped state, or after a pop-up dialog (e.g., incoming call) is cleared. When your activity is resumed and is now fully in the foreground, the user can interact with it:

- They can tap on your widgets
- Navigation button clicks, such as BACK, affect your activity
- Hardware input, such as from a keyboard, is sent to your activity

Conversely, anything that takes over user input — the activation of another activity — will result in your `onPause()` being called. Here, you should undo anything you did in `onResume()`.

Once `onPause()` is called, Android reserves the right to kill off your activity's process at any point. Hence, you should not be relying upon receiving any further events.

Stick to the Pairs

If you initialize something in `onCreate()`, clean it up in `onDestroy()`.

If you initialize something in `onStart()`, clean it up in `onStop()`.

If you initialize something in `onResume()`, clean it up in `onPause()`.

COPING WITH CONFIGURATIONS

In other words, stick to the pairs. For example, do not initialize something in `onStart()` and try to clean it up in `onPause()`, as there are scenarios where `onPause()` may be called multiple times in succession (i.e., user brings up a non-full-screen activity, which triggers `onPause()` but not `onStop()`, and hence not `onStart()`).

Which pairs of lifecycle methods you choose is up to you, depending upon your needs. You may decide that you need two pairs (e.g., `onCreate()/onDestroy()` and `onStart()/onStop()`). Just do not mix and match between them.

Making the Superclass Happy

If you override a lifecycle method, you need to chain to the superclass' implementation of the method. Otherwise, you will crash at runtime with a `SuperNotCalledException`. Android Studio will warn you if you implement a lifecycle method and fail to chain to the superclass.

In practice, *when* you chain to the superclass' implementation is up to you, so long as it is in the same method (e.g., chaining to `super.onCreate()` from `onCreate()`). In theory, though, if you are relying on things that you inherit from `Activity`, it is safest to:

- Chain to the superclass before doing your own work for the creation set of methods (`onCreate()`, `onStart()`, `onResume()`)
- Chain to the superclass after doing your own work for the destruction set of methods (`onPause()`, `onStop()`, `onDestroy()`)

When Activities Die

So, what gets rid of an activity? What can trigger the chain of events that results in `onDestroy()` being called?

First and foremost, when the user presses the BACK button, the foreground activity will be destroyed, and control will return to the previous activity in the user's navigation flow (i.e., whatever activity they were on before the now-destroyed activity came to the foreground).

You can accomplish the same thing by calling `finish()` from your activity. This is mostly for cases where some other UI action would indicate that the user is done with the activity (e.g., the activity presents a list for the user to choose from —

COPING WITH CONFIGURATIONS

clicking on a list item might close the activity). However, please do not artificially add your own “exit”, “quit”, or other menu items or buttons to your activity — just allow the user to use normal Android navigation options, such as the BACK button.

If none of your activities are in the foreground any more, your application’s process is a candidate to be terminated to free up RAM. As noted earlier, depending on circumstances, Android may or may not call `onDestroy()` in these cases (`onPause()` and `onStop()` would have been called when your activities left the foreground).

If the user causes the device to go through a configuration change, such as switching between portrait and landscape, Android’s default behavior is to destroy your current foreground activity and create a brand new one in its place. We will cover this more [in the next chapter](#).

And, if your activity has an unhandled exception, your activity will be destroyed, though Android will not call any more lifecycle methods on it, as it assumes your activity is in an unstable state.

Context Anti-Pattern: Outliving It

Suppose that we had an activity that looked like this:

```
private lateinit var doNotDoThis: View

class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        setContentView(R.layout.activity_main)

        doNotDoThis = showElapsed // from the SimpleBoom layout, showElapsed is a widget
    }
}
```

We take a widget from our activity’s layout and we assign it to a global property.

On the surface, this may not seem all that bad. However, when the activity is destroyed — due to a configuration change, BACK button press, etc. — we now have a memory leak.

Each widget holds a reference to the activity that created it. In this case, `doNotDoThis` has a reference back to our `MainActivity`. Since `doNotDoThis` is global

COPING WITH CONFIGURATIONS

in scope, that widget cannot be garbage collected. And since the `doNotDoThis` reference prevents the widget from being garbage collected, it prevents the destroyed activity from being garbage collected.

Fortunately, Android Studio will complain if you try to do this sort of thing.

In general, be *very* careful about having objects that are tied to some Context outlive the Context itself. This can include forking a background thread that has a reference to a Context, as that Context cannot be garbage collected until the thread terminates.

Integrating ViewModel

If you happened to install and run [the RecyclerViewBasics sample app](#), try the following:

1. Run the app
2. Make note of the top color
3. Rotate the device

Most likely, you will find that the top color is different than before. This implies that we generated a new random set of colors. And, if you keep switching the device between portrait and landscape, you would keep getting new colors.

That is not great.

It would be worse if our data were coming from disk, or from the network, instead of being randomly generated. If we have to re-load our data every time that the user rotates the screen, we would wind up having to make many extra requests of our database (local or remote), wasting time and battery.

In this chapter, we will focus on how to avoid this problem, through a viewmodel.

Configuration Changes

When you call methods in the Android SDK that load a resource (e.g., `setContentView(R.layout.activity_main)`), Android will walk through your resource sets, find the right resource for the given request, and use it.

But what happens if the configuration changes *after* we asked for the resource? For example, what if the user was holding their device in portrait mode, then rotates the

screen to landscape? We might want a version of our layouts that can take advantage of the wider screen, if such versions exist. And, since we already requested the resources, Android has no good way of handing us revised resources on the fly... except by forcing us to re-request those resources. So, this is what Android does, by default, to our foreground activity, when the [configuration](#) changes on the fly.

The biggest thing that Android does on a configuration change is destroy and recreate our activity. In other words:

- Android calls `onPause()`, `onStop()`, and `onDestroy()` on our original instance of the activity
- Android creates a brand new instance of the same activity class, using the same Intent that was used to create the original instance
- Android calls `onCreate()`, `onStart()`, and `onResume()` of the new activity instance
- The new activity appears on the screen

This may seem... invasive. You might not expect that Android would wipe out a perfectly good activity, just because the user flicked her wrist and rotated the screen of her phone. However, this is the only way Android has that guarantees that we will re-request all our resources.

What We Want... and What We Do Not Want

We want to present a user interface that fits well with the current configuration. For example, we want to take advantage of the screen space that is given to us.

Conversely, we do not want to have a UI that fits poorly in the given space. For example, in landscape mode, our layout might be too tall. Even if we use a `ScrollView` to make it vertically scrollable, so the user can take steps to see the entire layout, that may not be very user-friendly.

However, despite the UI change, we want the user to think that nothing much unusual happened when the user triggered the configuration.

We do **not** want the user to regret that configuration change, such as by losing the data that we were showing (or, worse, that the user had entered or changed).

Over the years, we have spent a fair amount of time dealing with this challenge. There is a two-tier solution that seems to work best. In this chapter, we will focus on

one of those tiers: retaining data using `ViewModel`.

Enter the `ViewModel`

So, a configuration change destroys and recreates our activity, by default. As a result, anything that is held onto uniquely by an activity instance — such as our random numbers — gets lost when we switch to the new activity instance, by default.

What would be nice is if we could separate our activity data into two groups:

- Part of our data, such as references to widgets, can be discarded on a configuration change, as we will need to get fresh data in the new activity
- Part of our data, such as our random numbers, could be passed from the old activity instance to the new activity instance, so we can present that data again to the user

The Jetpack solution for this is `ViewModel`. We can create a subclass of `ViewModel` that holds onto the second category of activity data: the stuff that we want to reuse after the configuration change. Jetpack will take the steps necessary to get that data — indeed, the entire `ViewModel` object — from the old activity instance to the new one. All that we need to do is make intelligent choices about what goes into the `ViewModel` and what does not.

Applying `ViewModel`

The `ViewModel` sample module in the [Sampler](#) and [SamplerJ](#) projects start with the same code that we used in `RecyclerViewBasics`. So, we have a `RecyclerView` in an activity that is set up to display 25 random numbers as colors. The difference is that this time, we will use a `ViewModel` to retain those random numbers across a configuration change. Also, we will log the lifecycle methods, so we can see them as they are executed.

The Dependencies

You will need to add a dependency on `androidx.lifecycle:lifecycle-extensions` to your project, in order to be able to create `ViewModel` instances in your activities.

Kotlin developers will also want to add `androidx.lifecycle:lifecycle-viewmodel-ktx` as a dependency. This adds a few Kotlin-specific extension functions associated

INTEGRATING VIEWMODEL

with `ViewModel` and related classes. They make using `ViewModel` a little bit easier in Kotlin than if you were to use the Java API alone. This is a common pattern in Jetpack: have a Java API with a separate library containing Kotlin extension functions. The “Android KTX” brand is used for these Kotlin extension function libraries.

So, in our Kotlin project, we have:

```
implementation 'androidx.lifecycle:lifecycle-extensions:2.0.0'  
implementation 'androidx.lifecycle:lifecycle-viewmodel-ktx:2.0.0'
```

(from [ViewModel/build.gradle](#))

The Java project just has:

```
implementation 'androidx.lifecycle:lifecycle-extensions:2.0.0'
```

(from [ViewModel/build.gradle](#))

The ViewModel

Our project now has a `ColorViewModel` class that inherits from Jetpack’s `ViewModel`, both in Java:

```
package com.commonsware.jetpack.samplerj.viewmodel;  
  
import java.util.ArrayList;  
import java.util.List;  
import java.util.Random;  
import androidx.lifecycle.ViewModel;  
  
public class ColorViewModel extends ViewModel {  
    private final Random random = new Random();  
    final List<Integer> numbers = buildItems();  
  
    private List<Integer> buildItems() {  
        ArrayList<Integer> result = new ArrayList<>(25);  
  
        for (int i = 0; i < 25; i++) {  
            result.add(random.nextInt());  
        }  
  
        return result;  
    }  
}
```

INTEGRATING VIEWMODEL

(from [ViewModel/src/main/java/com/commonsware/jetpack/samplerj/viewmodel/ColorViewModel.java](#))

...and Kotlin:

```
package com.commonsware.jetpack.sampler.viewmodel

import androidx.lifecycle.ViewModel
import java.util.*

class ColorViewModel : ViewModel() {
    private val random = Random()
    val numbers = List(25) { random.nextInt() }
}
```

(from [ViewModel/src/main/java/com/commonsware/jetpack/sampler/viewmodel/ColorViewModel.kt](#))

Our random numbers are now created in the `ColorViewModel`, via property initializers. Otherwise, our `ColorViewModel` itself is fairly unremarkable.

Using the ViewModel

Our activity can get its instance of `ColorViewModel` from `ViewModelProviders`. `ViewModelProviders` has an `of()` static method that will return a `ViewModelProvider` for our activity. And, on that, we can call `get()` to retrieve our `ColorViewModel`.

In Java, we need to pass the Java Class object for our `ColorViewModel` as a parameter to `get()`:

```
ColorViewModel vm = ViewModelProviders.of(this).get(ColorViewModel.class);
```

(from [ViewModel/src/main/java/com/commonsware/jetpack/samplerj/viewmodel/MainActivity.java](#))

In Kotlin, the `androidx.lifecycle:lifecycle-viewmodel-ktx` gives us a `get()` function that does not need a parameter, so long as we are assigning the result to a `ColorViewModel` variable:

```
val vm: ColorViewModel = ViewModelProviders.of(this).get()
```

(from [ViewModel/src/main/java/com/commonsware/jetpack/samplerj/viewmodel/MainActivity.kt](#))

We can then use the `numbers` property of our `ColorViewModel` to populate our `RecyclerView`:

```
setContentView(R.layout.activity_main);
```

```
RecyclerView items = findViewById(R.id.items);
ColorAdapter adapter = new ColorAdapter(getLayoutInflater());
ColorViewModel vm = ViewModelProviders.of(this).get(ColorViewModel.class);

adapter.submitList(vm.numbers);
items.setLayoutManager(new LinearLayoutManager(this));
items.addItemDecoration(
    new DividerItemDecoration(this, DividerItemDecoration.VERTICAL));
items.setAdapter(adapter);
```

(from [ViewModel/src/main/java/com/commonsware/jetpack/samplerj/viewmodel/MainActivity.java](#))

```
setContentView(R.layout.activity_main)

val vm: ColorViewModel = ViewModelProviders.of(this).get()

items.apply {
    layoutManager = LinearLayoutManager(this@MainActivity)
    addItemDecoration(
        DividerItemDecoration(
            this@MainActivity,
            DividerItemDecoration.VERTICAL
        )
    )
    adapter = ColorAdapter(layoutInflater).apply {
        submitList(vm.numbers)
    }
}
```

(from [ViewModel/src/main/java/com/commonsware/jetpack/samplerj/viewmodel/MainActivity.kt](#))

The Results

If you run this sample app, and you rotate the screen, the random numbers are the same.

The reason this works is that we have the *same* ColorViewModel instance in both the old activity (before the configuration change) and the new activity (after the configuration change). ViewModelProviders and the rest of the ViewModel portion of Jetpack will ensure that we have the same ColorViewModel in the old and new activity instances. Since our ColorViewModel holds our data, our data is retained across the configuration change, so we have the same numbers in both the old and the new activity.

ViewModel and the Lifecycle

Our MainActivity also has calls to Log.d() in the main lifecycle functions, both in Java:

```
package com.commonsware.jetpack.samplerj.viewmodel;

import android.os.Bundle;
import android.util.Log;
import androidx.appcompat.app.AppCompatActivity;
import androidx.lifecycle.ViewModelProviders;
import androidx.recyclerview.widget.DividerItemDecoration;
import androidx.recyclerview.widget.LinearLayoutManager;
import androidx.recyclerview.widget.RecyclerView;

public class MainActivity extends AppCompatActivity {
    private static final String TAG = "ViewModel";

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

        RecyclerView items = findViewById(R.id.items);
        ColorAdapter adapter = new ColorAdapter(getApplicationContext());
        ColorViewModel vm = ViewModelProviders.of(this).get(ColorViewModel.class);

        adapter.submitList(vm.numbers);
        items.setLayoutManager(new LinearLayoutManager(this));
        items.addItemDecoration(
            new DividerItemDecoration(this, DividerItemDecoration.VERTICAL));
        items.setAdapter(adapter);

        Log.d(TAG, "onCreate() called!");
    }

    @Override
    protected void onStart() {
        super.onStart();

        Log.d(TAG, "onStart() called!");
    }

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

INTEGRATING VIEWMODEL

```
    Log.d(TAG, "onResume() called!");
}

@Override
protected void onPause() {
    Log.d(TAG, "onPause() called!");

    super.onPause();
}

@Override
protected void onStop() {
    Log.d(TAG, "onStop() called!");

    super.onStop();
}

@Override
protected void onDestroy() {
    Log.d(TAG, "onDestroy() called!");

    super.onDestroy();
}
}
```

(from [ViewModel/src/main/java/com/commonsware/jetpack/samplerj/viewmodel/MainActivity.java](#))

...and in Kotlin:

```
package com.commonsware.jetpack.sampler.viewmodel

import android.os.Bundle
import android.util.Log
import androidx.appcompat.app.AppCompatActivity
import androidx.lifecycle.ViewModelProviders
import androidx.lifecycle.get
import androidx.recyclerview.widget.DividerItemDecoration
import androidx.recyclerview.widget.LinearLayoutManager
import kotlinx.android.synthetic.main.activity_main.*

private const val TAG = "ViewModel"

class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
```

INTEGRATING VIEWMODEL

```
val vm: ColorViewModel = ViewModelProviders.of(this).get()

items.apply {
    layoutManager = LinearLayoutManager(this@MainActivity)
    addItemDecoration(
        DividerItemDecoration(
            this@MainActivity,
            DividerItemDecoration.VERTICAL
        )
    )
    adapter = ColorAdapter(layoutInflater).apply {
        submitList(vm.numbers)
    }
}

Log.d(TAG, "onCreate() called!")
}

override fun onStart() {
    super.onStart()

    Log.d(TAG, "onStart() called!")
}

override fun onResume() {
    super.onResume()

    Log.d(TAG, "onResume() called!")
}

override fun onPause() {
    Log.d(TAG, "onPause() called!")

    super.onPause()
}

override fun onStop() {
    Log.d(TAG, "onStop() called!")

    super.onStop()
}

override fun onDestroy() {
    Log.d(TAG, "onDestroy() called!")

    super.onDestroy()
}
}
```

INTEGRATING VIEWMODEL

(from [ViewModel/src/main/java/com/commonsware/jetpack/sampler/viewmodel/MainActivity.kt](#))

As you run the app, and as you undergo configuration changes, you will see the lifecycle methods get logged, by looking for these messages in the Logcat view in Android Studio. For example, starting the app will result in Logcat messages like:

```
com.commonsware.jetpack.sampler.viewmodel D/ViewModel: onCreate() called!
com.commonsware.jetpack.sampler.viewmodel D/ViewModel: onStart() called!
com.commonsware.jetpack.sampler.viewmodel D/ViewModel: onResume() called!
```

If you rotate the screen, the list of messages becomes:

```
com.commonsware.jetpack.sampler.viewmodel D/ViewModel: onCreate() called!
com.commonsware.jetpack.sampler.viewmodel D/ViewModel: onStart() called!
com.commonsware.jetpack.sampler.viewmodel D/ViewModel: onResume() called!
com.commonsware.jetpack.sampler.viewmodel D/ViewModel: onPause() called!
com.commonsware.jetpack.sampler.viewmodel D/ViewModel: onStop() called!
com.commonsware.jetpack.sampler.viewmodel D/ViewModel: onDestroy() called!
com.commonsware.jetpack.sampler.viewmodel D/ViewModel: onCreate() called!
com.commonsware.jetpack.sampler.viewmodel D/ViewModel: onStart() called!
com.commonsware.jetpack.sampler.viewmodel D/ViewModel: onResume() called!
```

If you press BACK to exit the rotated activity, the list of messages becomes:

```
com.commonsware.jetpack.sampler.viewmodel D/ViewModel: onCreate() called!
com.commonsware.jetpack.sampler.viewmodel D/ViewModel: onStart() called!
com.commonsware.jetpack.sampler.viewmodel D/ViewModel: onResume() called!
com.commonsware.jetpack.sampler.viewmodel D/ViewModel: onPause() called!
com.commonsware.jetpack.sampler.viewmodel D/ViewModel: onStop() called!
com.commonsware.jetpack.sampler.viewmodel D/ViewModel: onDestroy() called!
com.commonsware.jetpack.sampler.viewmodel D/ViewModel: onCreate() called!
com.commonsware.jetpack.sampler.viewmodel D/ViewModel: onStart() called!
com.commonsware.jetpack.sampler.viewmodel D/ViewModel: onResume() called!
com.commonsware.jetpack.sampler.viewmodel D/ViewModel: onPause() called!
com.commonsware.jetpack.sampler.viewmodel D/ViewModel: onStop() called!
com.commonsware.jetpack.sampler.viewmodel D/ViewModel: onDestroy() called!
```

We are seeing our original activity instance come onto the screen, then get destroyed and a replacement created, before it too gets destroyed as the user exits the activity.

Our `ColorViewModel` is oblivious to most of this. It will get created as part of the original `onCreate()` call. When the new activity instance is in its `onCreate()`, and it attempts to retrieve the `ColorViewModel`, Jetpack realizes that we already have a `ColorViewModel` from the previous activity instance, so it reuses it.

INTEGRATING VIEWMODEL

If we wanted, we could override `onCleared()` in `ColorViewModel`. This will be called when our activity is “really” destroyed:

- `onDestroy()` is called, and
- We are not undergoing a configuration change

In `onCleared()`, we could clean up anything that we needed to clean up. Often, there is nothing that you need to clean up specifically — after all, the `ViewModel` will get garbage-collected shortly. However, sometimes your `ViewModel` might hold onto things that should get cleaned up, such as:

- References to threads, Kotlin coroutines, or the like
- Open Internet connections to some server
- Open references to local databases
- And so on

To the extent that your `ViewModel` has those, `onCleared()` is where you would release those things, such as disconnecting from the server.

Changing Data in the ViewModel

In our 25-random-numbers example, the data never changed. That is what we wanted for that example, so our random numbers would be consistent through configuration changes.

However, it is not very realistic. Most of the time, our data changes when the user uses the app, and we need our viewmodel to handle that.

INTEGRATING VIEWMODEL

The `LifecycleList` sample module in the [Sampler](#) and [SamplerJ](#) projects blend our lifecycle-logging logic with our show-a-list logic. Now, instead of showing 25 random colors, we will show a list of lifecycle events, collected as the user uses the app:

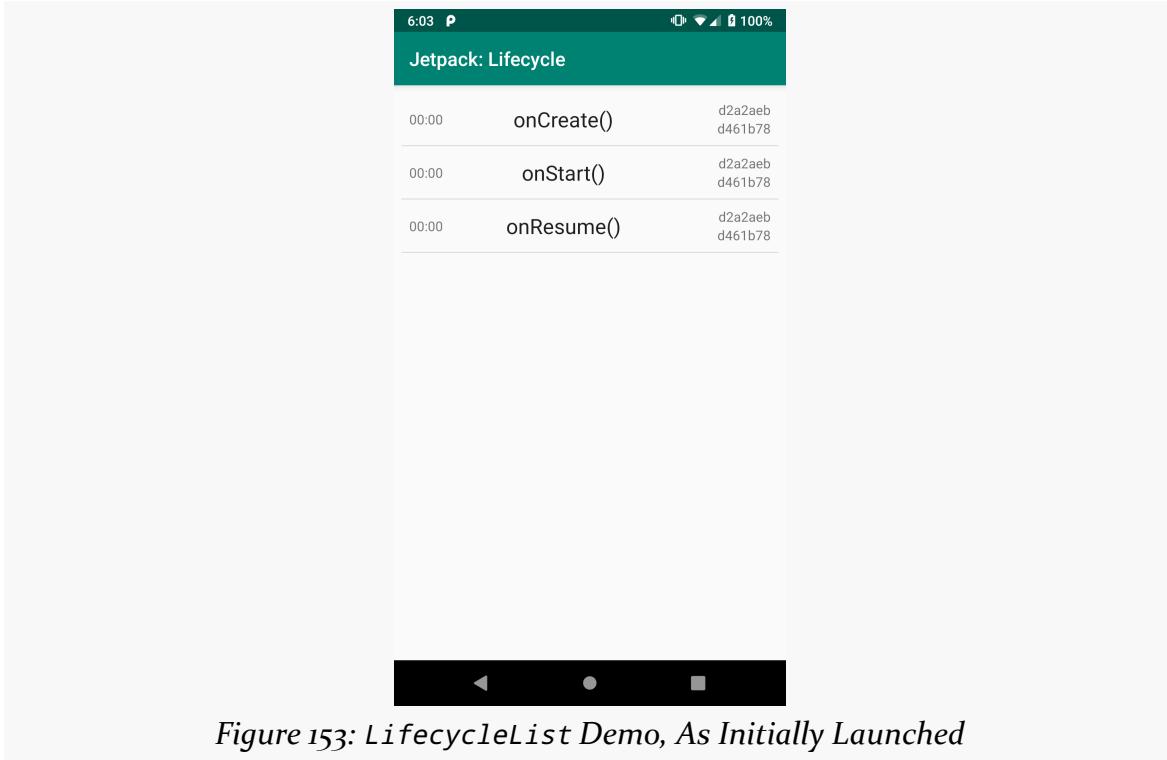


Figure 153: `LifecycleList` Demo, As Initially Launched

Each row in the list now shows four pieces of data:

- The lifecycle method that was called (e.g., `onCreate()`)
- The number of seconds since the app was started when the event occurred (e.g., 0:00 for initial events)
- A random number identifying the activity (in the upper right corner)
- A random number identifying the.viewmodel (in the lower right corner)

These latter two values will help show us when we wind up with different instances of those objects.

The Event Model

We need some object to hold that data to be shown in each of our `RecyclerView` rows. So, this project has an `Event` model class, in Java:

INTEGRATING VIEWMODEL

```
package com.commonsware.jetpack.samplerj.lifecycle;

import android.os.SystemClock;

class Event {
    final long timestamp = SystemClock.elapsedRealtime();
    final String message;
    final int activityHash;
    final int viewmodelHash;

    Event(String message, int activityHash, int viewmodelHash) {
        this.message = message;
        this.activityHash = activityHash;
        this.viewmodelHash = viewmodelHash;
    }
}
```

(from [LifecycleList/src/main/java/com/commonsware/jetpack/samplerj/lifecycle/Event.java](#))

...or Kotlin:

```
package com.commonsware.jetpack.sampler.lifecycle

import android.os.SystemClock

data class Event(
    val message: String,
    val activityHash: Int,
    val viewmodelHash: Int,
    val timestamp: Long = SystemClock.elapsedRealtime()
)
```

(from [LifecycleList/src/main/java/com/commonsware/jetpack/sampler/lifecycle/Event.kt](#))

The two hashCode() values are Int properties. The lifecycle method is a String referred to as the message. And we have a timestamp property that will track when this Event was created. For that, we use SystemClock.elapsedRealtime(), which returns the number of milliseconds since the device was powered on.

The New RecyclerView Bits

Our row layout now needs widgets for those four pieces of data that we wish to display:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
```

INTEGRATING VIEWMODEL

```
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="wrap_content"
android:padding="@dimen/content_padding">

<TextView
    android:id="@+id/activityHash"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    tools:text="0x12345678" />

<TextView
    android:id="@+id/viewmodelHash"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/activityHash"
    tools:text="0x90ABCDEF" />

<androidx.constraintlayout.widget.Barrier
    android:id="@+id/barrier"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginEnd="4dp"
    android:layout_marginStart="4dp"
    app:barrierDirection="start"
    app:constraint_referenced_ids="activityHash,viewmodelHash" />

<TextView
    android:id="@+id/timestamp"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    tools:text="01:23" />

<TextView
    android:id="@+id/message"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textAppearance="?android:attr/textAppearanceLarge"
    app:layout_constraintBottom_toBottomOf="parent"
```

INTEGRATING VIEWMODEL

```
    app:layout_constraintEnd_toStartOf="@+id/barrier"
    app:layout_constraintStart_toEndOf="@+id/timestamp"
    app:layout_constraintTop_toTopOf="parent"
    tools:text="onDestroy()" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

(from [LifecycleList/src/main/res/layout/row.xml](#))

Each of the four pieces of data is represented by a `TextView`. The two hash values are anchored to the “end” side of the `ConstraintLayout`, with the `activityHash` on the top and the `viewmodelHash` on the bottom. The `timestamp` widget is anchored on the “start” side of the `ConstraintLayout`, centered between the top and the bottom. And the `message` widget is centered in the remaining space, using a `Barrier` to determine where the hashes start.

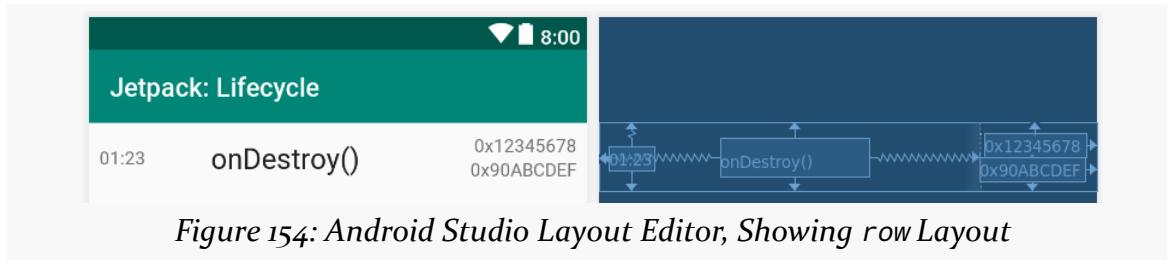


Figure 154: Android Studio Layout Editor, Showing row Layout

Note that we dropped the `android:background`, `android:clickable`, and `android:focusable` attributes from the `ConstraintLayout`, as this particular sample is not going to respond to click events on the rows. That allows our new view-holder class, `EventViewHolder`, to just focus on populating the widgets for its row:

```
package com.commonsware.jetpack.samplerj.lifecycle;

import android.text.format.DateUtils;
import android.util.TimeUtils;
import android.view.View;
import android.widget.TextView;
import androidx.recyclerview.widget.RecyclerView;

class EventViewHolder extends RecyclerView.ViewHolder {
    private final TextView timestamp, message, activityHash,.viewmodelHash;
    private final long startTime;

    EventViewHolder(View row, long startTime) {
        super(row);

        this.startTime = startTime;
```

INTEGRATING VIEWMODEL

```
    timestamp = row.findViewById(R.id.timestamp);
    message = row.findViewById(R.id.message);
    activityHash = row.findViewById(R.id.activityHash);
    viewmodelHash = row.findViewById(R.id.viewmodelHash);
}

void bindTo(Event event) {
    long elapsedSeconds = (event.timestamp - startTime)/1000;

    timestamp.setText(DateUtils.formatElapsedTime(elapsedSeconds));
    message.setText(event.message);
    activityHash.setText(Integer.toHexString(event.activityHash));
    viewmodelHash.setText(Integer.toHexString(event.viewmodelHash));
}
}
```

(from [LifecycleList/src/main/java/com/commonsware/jetpack/samplerj/lifecycle/EventViewHolder.java](#))

```
package com.commonsware.jetpack.sampler.lifecycle

import android.text.format.DateUtils
import android.view.View
import androidx.recyclerview.widget.RecyclerView
import kotlinx.android.synthetic.main.row.view.*

class EventViewHolder(row: View, private val startTime: Long) :
    RecyclerView.ViewHolder(row) {
    private val timestamp = row.timestamp
    private val message = row.message
    private val activityHash = row.activityHash
    private val viewmodelHash = row.viewmodelHash

    fun bindTo(event: Event) {
        val elapsedSeconds = (event.timestamp - startTime) / 1000

        timestamp.text = DateUtils.formatElapsedTime(elapsedSeconds)
        message.text = event.message
        activityHash.text = Integer.toHexString(event.activityHash)
        viewmodelHash.text = Integer.toHexString(event.viewmodelHash)
    }
}
```

(from [LifecycleList/src/main/java/com/commonsware/jetpack/sampler/lifecycle/EventViewHolder.kt](#))

For the `timestamp`, we use `DateUtils.formatElapsedTime()`. This is a utility method provided by Android that formats a number of seconds into an HH:MM:SS format to show the elapsed time in hours, minutes, and seconds. Note that the `Event` value for the `timestamp` is in milliseconds, as is the `startTime` value that is being passed into

INTEGRATING VIEWMODEL

EventViewHolder, so we need to divide our net time by 1000 to convert the milliseconds to seconds.

Our new adapter — EventAdapter — wraps a List of Event objects and pours them into EventViewHolder objects as needed:

```
package com.commonsware.jetpack.samplerj.lifecycle;

import android.os.SystemClock;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import androidx.annotation.NonNull;
import androidx.recyclerview.widget.DiffUtil;
import androidx.recyclerview.widget.ListAdapter;

class EventAdapter extends ListAdapter<Event, EventViewHolder> {
    private final LayoutInflater inflater;
    private final long startTime;

    EventAdapter(LayoutInflater inflater, long startTime) {
        super(DIFF_CALLBACK);
        this.inflater = inflater;
        this.startTime = startTime;
    }

    @NonNull
    @Override
    public EventViewHolder onCreateViewHolder(@NonNull ViewGroup parent,
                                              int viewType) {
        View row = inflater.inflate(R.layout.row, parent, false);

        return new EventViewHolder(row, startTime);
    }

    @Override
    public void onBindViewHolder(@NonNull EventViewHolder holder, int position) {
        holder.bindTo(getItem(position));
    }

    private static final DiffUtil.ItemCallback<Event> DIFF_CALLBACK =
        new DiffUtil.ItemCallback<Event>() {
            @Override
            public boolean areItemsTheSame(@NonNull Event oldEvent, @NonNull Event newEvent) {
                return oldEvent == newEvent;
            }

            @Override
            public boolean areContentsTheSame(@NonNull Event oldEvent, @NonNull Event newEvent) {
                return oldEvent.timestamp == newEvent.timestamp &&
                    oldEvent.message.equals(newEvent.message) &&
                    oldEvent.activityHash == newEvent.activityHash &&
                    oldEvent.viewmodelHash == newEvent.viewmodelHash;
            }
        };
}
```

INTEGRATING VIEWMODEL

(from [LifecycleList/src/main/java/com/commonsware/jetpack/samplerj/lifecycle/EventAdapter.java](#))

```
package com.commonsware.jetpack.sampler.lifecycle

import android.view.LayoutInflater
import android.view.ViewGroup
import androidx.recyclerview.widget.DiffUtil
import androidx.recyclerview.widget.ListAdapter

internal class EventAdapter(
    private val inflater: LayoutInflater,
    private val startTime: Long
) : ListAdapter<Event, EventViewHolder>(EventDiffer) {

    override fun onCreateViewHolder(
        parent: ViewGroup,
        viewType: Int
    ): EventViewHolder {
        val row = inflater.inflate(R.layout.row, parent, false)

        return EventViewHolder(row, startTime)
    }

    override fun onBindViewHolder(holder: EventViewHolder, position: Int) {
        holder.bindTo(getItem(position))
    }

    private object EventDiffer : DiffUtil.ItemCallback<Event>() {
        override fun areItemsTheSame(oldEvent: Event, newEvent: Event) =
            oldEvent === newEvent

        override fun areContentsTheSame(oldEvent: Event, newEvent: Event) =
            oldEvent == newEvent
    }
}
```

(from [LifecycleList/src/main/java/com/commonsware/jetpack/sampler/lifecycle/EventAdapter.kt](#))

For our `DiffUtil.ItemCallback` implementation, we use identity equality to determine whether the items are the same and content equality to determine if the contents are the same. Here, though, we have more substantial language differences:

- In Java, identity equality is via `==`, and we have to examine each one of the `Event` fields ourselves in `areContentsTheSame()`
- In Kotlin, identity is via `===`, and we can use object equality (`==`) for our `Event`, since it is a data class and generates an `equals()` function that compares each property for us

The EventViewModel

Our ViewModel is now called EventViewModel, and it has two properties:

- events, which is a list of the events that we have had to date
- startTime, which is the time when the EventViewModel is created, once again obtained via SystemClock.elapsedRealtime()

```
package com.commonsware.jetpack.samplerj.lifecycle;

import android.os.SystemClock;
import java.util.ArrayList;
import java.util.List;
import java.util.Random;
import androidx.lifecycle.ViewModel;

public class EventViewModel extends ViewModel {
    final List<Event> events = new ArrayList<>();
    final long startTime = SystemClock.elapsedRealtime();
    private final int id = new Random().nextInt();

    void addEvent(String message, int activityHash) {
        events.add(new Event(message, activityHash, id));
    }

    @Override
    protected void onCleared() {
        events.clear();
    }
}
```

(from [LifecycleList/src/main/java/com/commonsware/jetpack/samplerj/lifecycle/EventViewModel.java](#))

```
package com.commonsware.jetpack.sampler.lifecycle

import android.os.SystemClock
import androidx.lifecycle.ViewModel
import java.util.*

class EventViewModel : ViewModel() {
    val events: MutableList<Event> = mutableListOf()
    val startTime = SystemClock.elapsedRealtime()
    private val id = Random().nextInt()

    fun addEvent(message: String, activityHash: Int) {
        events.add(Event(message, activityHash, id))
    }
}
```

INTEGRATING VIEWMODEL

```
    override fun onCleared() {
        events.clear()
    }
}
```

(from [LifecycleList/src/main/java/com/commonsware/jetpack/sampler/lifecycle/EventViewModel.kt](#))

We also have:

- `addEvent()`, to record an event when it occurs
- `onCleared()`, to clear the events list when the `EventViewModel` is no longer being used

The `onCleared()` implementation is unnecessary, as the events list would be garbage-collected when the `EventViewModel` is. We have it here just to show you what overriding that method looks like.

Updating the EventViewModel

`MainActivity` now has an `addEvent()` function to update the `EventViewModel` when lifecycle events occur:

```
package com.commonsware.jetpack.samplerj.lifecycle;

import android.os.Bundle;
import java.util.ArrayList;
import java.util.Random;
import androidx.appcompat.app.AppCompatActivity;
import androidx.lifecycle.ViewModelProviders;
import androidx.recyclerview.widget.DividerItemDecoration;
import androidx.recyclerview.widget.LinearLayoutManager;
import androidx.recyclerview.widget.RecyclerView;

public class MainActivity extends AppCompatActivity {
    private EventAdapter adapter;
    private EventViewModel vm;
    private final int id = new Random().nextInt();

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

        RecyclerView items = findViewById(R.id.items);
        adapter = new EventAdapter();
        items.setAdapter(adapter);
        vm = ViewModelProviders.of(this).get(EventViewModel.class);
        vm.getEvents().observe(this, events -> adapter.setItems(events));
    }

    public void addEvent(Event event) {
        vm.addEvent(event);
    }
}
```

INTEGRATING VIEWMODEL

```
vm = ViewModelProviders.of(this).get(EventViewModel.class);
adapter = new EventAdapter(getLayoutInflater(), vm.startTime);
addEvent("onCreate()");

items.setLayoutManager(new LinearLayoutManager(this));
items.addItemDecoration(
    new DividerItemDecoration(this, DividerItemDecoration.VERTICAL));
items.setAdapter(adapter);
}

@Override
protected void onStart() {
    super.onStart();

    addEvent("onStart()");
}

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

    addEvent("onResume()");
}

@Override
protected void onPause() {
    addEvent("onPause()");

    super.onPause();
}

@Override
protected void onStop() {
    addEvent("onStop()");

    super.onStop();
}

@Override
protected void onDestroy() {
    addEvent("onDestroy()");

    super.onDestroy();
}

private void addEvent(String message) {
    vm.addEvent(message, id);
```

INTEGRATING VIEWMODEL

```
    adapter.submitList(new ArrayList<>(vm.events));
}
}
```

(from [LifecycleList/src/main/java/com/commonsware/jetpack/samplerj/lifecycle/MainActivity.java](#))

```
package com.commonsware.jetpack.sampler.lifecycle

import android.os.Bundle
import androidx.appcompat.app.AppCompatActivity
import androidx.lifecycle.ViewModelProviders
import androidx.lifecycle.get
import androidx.recyclerview.widget.DividerItemDecoration
import androidx.recyclerview.widget.LinearLayoutManager
import androidx.recyclerview.widget.RecyclerView
import kotlinx.android.synthetic.main.activity_main.*
import java.util.*
import kotlin.collections.ArrayList

class MainActivity : AppCompatActivity() {
    private lateinit var adapter: EventAdapter
    private lateinit var vm: EventViewModel
    private val id = Random().nextInt()

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        vm = ViewModelProviders.of(this).get()
        adapter = EventAdapter(layoutInflater, vm.startTime)
        addEvent("onCreate()")

        items.layoutManager = LinearLayoutManager(this)
        items.addItemDecoration(
            DividerItemDecoration(this, DividerItemDecoration.VERTICAL)
        )
        items.adapter = adapter
    }

    override fun onStart() {
        super.onStart()

        addEvent("onStart()")
    }

    override fun onResume() {
        super.onResume()
    }
}
```

INTEGRATING VIEWMODEL

```
    addEvent("onResume()")  
}  
  
override fun onPause() {  
    addEvent("onPause()")  
  
    super.onPause()  
}  
  
override fun onStop() {  
    addEvent("onStop()")  
  
    super.onStop()  
}  
  
override fun onDestroy() {  
    addEvent("onDestroy()")  
  
    super.onDestroy()  
}  
  
private fun addEvent(message: String) {  
    vm.addEvent(message, id)  
    adapter.submitList(ArrayList(vm.events))  
}  
}
```

(from [LifecycleList/src/main/java/com/commonsware/jetpack/sampler/lifecycle/MainActivity.kt](#))

`addEvent()` also calls `submitList()` on our `ListAdapter`, to update the `RecyclerView` when we add events to the list. This requires us to hold onto the `ListAdapter` and the `EventViewModel` in properties of the activity, rather than just using them as local variables in `onCreate()`.

When `addEvent()` calls `submitList()`, it creates a new `ArrayList` from the events. That is because `ListAdapter` and `submitList()` will “short-circuit” the update logic if you pass the same `List` to `submitList()` that you did the previous call. `ListAdapter` assumes that nothing needs to be done in that case, as `ListAdapter` assumes that the list *contents* did not change. To get past this, we have to pass in a brand-new `ArrayList` object each time.

The Results

When you run the app, lifecycle events show up in the list. If you rotate the screen a couple of times, the events start to pile up:

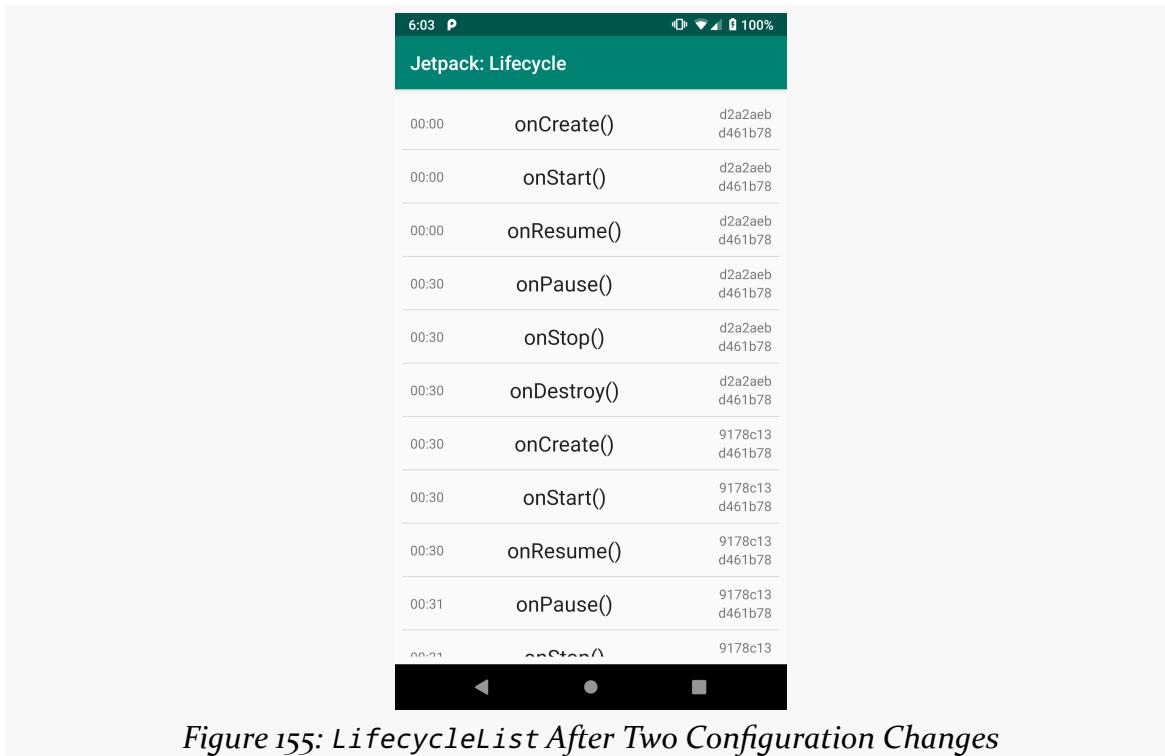


Figure 155: LifecycleList After Two Configuration Changes

The activity `hashCode()` value in the upper right changes with each configuration change, as we get a fresh instance of `MainActivity` each time. However, the viewmodel `hashCode()` value in the lower right remains the same, as we are using the same `EventViewModel` instance for each of our `MainActivity` instances.

ViewModel and AndroidViewModel

Sometimes, your `ViewModel` could really use a `Context`. For example, you might need to format some data using a string resource, and the only way to get a string resource is through a `Context`.

If you need a `Context` in your `ViewModel`, Google's official solution is `AndroidViewModel`. This offers a `getApplication()` function that you can call to retrieve the `Application` singleton. `Application` implements `Context`, so you can

INTEGRATING VIEWMODEL

look up string resources and stuff from there. Your activity or fragment does not need to worry about this: just use `ViewModelProviders.of()` and request the `ViewModel`.

Some of the examples in the upcoming chapters will use `ViewModel`, while others will use `AndroidViewModel`.

Licensed solely for use by Patrocinio Rodriguez

Understanding Processes

So far, we have been treating our activity like it is our entire application. Soon, we will start to get into more complex scenarios, involving multiple activities and other types of components, like services and content providers.

But, before we get into a lot of that, it is useful to understand how all of this ties into the actual OS itself. Android is based on Linux, and Linux applications run in OS processes. Understanding a bit about how Android and Linux processes inter-relate will be useful in understanding how our mixed bag of components work within these processes.

And, along the way, we will answer nagging questions like, “what *is* that Bundle thing that we are passed in `onCreate()` that we have been ignoring?”.

When Processes Are Created

A user installs your app, goes to their home screen’s launcher, and taps on an icon representing your activity. Your activity dutifully appears on the screen.

At this point, we know that we have a process for our app, because it is running.

Frequently, the trigger for Android to fork a process for you is that the user tapped on your launcher icon and launched your activity. However, your process might already exist for other reasons. In that case, by default, Android will reuse your existing process, and have it display your activity. By default, your app will have at most one process.

What Is In Your Process

When Android needs a process for your app — such as to show the launcher activity — Android forks a copy of a process known as the zygote. As a result of the way your process is forked from the zygote, your process contains:

- A copy of a virtual machine for running your app, shared among all such processes via Linux copy-on-write memory sharing
- A copy of the Android framework classes, like `Activity` and `Button`, also shared via copy-on-write memory
- A copy of shared native libraries, such as for SSL encryption and local database access, also shared via copy-on-write memory
- A copy of your own classes, loaded out of your APK
- A copy of classes from libraries in your app, loaded out of your APK
- Any native libraries (e.g., written in C/C++) that you linked into your app, loaded out of your APK
- Any objects created by you or the framework classes, such as the instance of your `Activity` subclass

So, our process has a lot of things in it. Much of it is shared among all other Android SDK apps forked from the zygote. The unique elements will be those things that we use from our APK.

BACK, HOME, and Your Process

So far, our sample apps have had just one activity. So, imagine this scenario: from the home screen's launcher, the user taps on the icon associated with your app's activity. Then, with your activity in the foreground, the user presses BACK.

At this point, the user is telling the OS that she is done with your activity. Our activity is destroyed. Control will return to whatever preceded that activity — in this case, the home screen's launcher.

You might think that this would cause your process to be terminated. After all, that is how most desktop operating systems work. Once the user closes the last window of the application, the process hosting that application is terminated.

However, that is not how Android works. Android will *keep your process around*, for a little while at least. This is done for speed and power: if the user happens to want to return to your app sooner rather than later, it is more efficient to simply bring up

UNDERSTANDING PROCESSES

another copy of your activity again in the existing process than it is to go set up a completely new copy of the process. This does not mean that your process will live forever; we will discuss when your process will go away later in this chapter.

Now, instead of the user pressing BACK, let's say that the user pressed HOME instead. Visually, frequently there is little difference: the home screen re-appears.

The difference is what happens to your activity.

When the user presses BACK, your foreground activity is destroyed. It will be called with `onDestroy()` among the other teardown lifecycle methods. And the activity itself — the instance of your subclass of `Activity` — will never be used again, and hopefully is garbage collected.

When the user presses HOME, your foreground activity is *not* destroyed... at least, not immediately. It remains in memory. If the user launches your app again from the home screen launcher, and if your process is still around, Android will simply bring your existing activity instance back to the foreground, rather than having to create a brand-new one (as is the case if the user pressed BACK and destroyed your activity).

What HOME literally is doing is bringing the home screen activity back to the foreground. Otherwise, it does not affect your process very much.

Termination

Processes cannot live forever. They take up a chunk of RAM, for your classes and objects, and these mobile devices only have so much RAM to work with. Eventually, therefore, Android has to get rid of your process, to free up memory for other apps and their processes.

How long your process will stick around depends on a variety of factors, including:

- What else the device is doing, either in the foreground (user using apps) or in the background (e.g., automated checks for new email)
- How much memory the device has
- What is still running inside your process

Going back to the scenario from above, we have an application with a single activity launched from the home screen, where the user can return to the home screen either by pressing BACK or by pressing HOME. You might think that this makes no

UNDERSTANDING PROCESSES

difference at all on when the process would be terminated, but that would be incorrect. Pressing HOME would keep the process around perhaps a bit longer than would pressing BACK.

Why?

When the user presses BACK, your one and only activity is destroyed. When the user presses HOME, your activity is not destroyed. Android will tend to keep processes around longer if they have active (i.e., not destroyed) components in them.

The key word there is “tend”.

There is an element of the Android OS called the “out-of-memory killer”. Its job is to ensure that there is a reasonable amount of free system RAM available at all times, so Android can fork new processes for apps when needed. When free system RAM drops below a certain level, the out-of-memory killer will terminate a process. The out-of-memory killer will tend to terminate empty processes more readily than it will terminate process with 1+ running components. However, the out-of-memory killer also takes into account things like:

- Process age, where older processes will be somewhat more likely to be terminated
- Memory usage, where bigger processes will be somewhat more likely to be terminated

However, in general, processes with active (not destroyed) components will stick around a bit longer than processes without such components.

Foreground Means “I Love You”

Just because Android terminates processes to free up memory does not mean that it will terminate just any process to free up memory. A foreground process — the most common of which is a process that has an activity in the foreground — is the least likely of all to be terminated. In fact, you can pretty much assume that if Android has to kill off the foreground process, that the phone is very sick and will crash in a matter of moments.

(and, fortunately, that does not happen very often)

So, if you are in the foreground, you are safe. It is only when you are not in the

UNDERSTANDING PROCESSES

foreground that you are at risk of having the process be terminated.

Tasks and Your App

If you have used an Android device for a reasonable length of time, most likely you will have seen the "overview screen, sometimes called the "recent-tasks list". The look has varied widely over the years and across the range of devices and manufacturers. They all give you a way to ["get back to where you once belonged"](#) — you can rapidly return to some app that you had been in recently.

So, on Android 9.0 you might see:

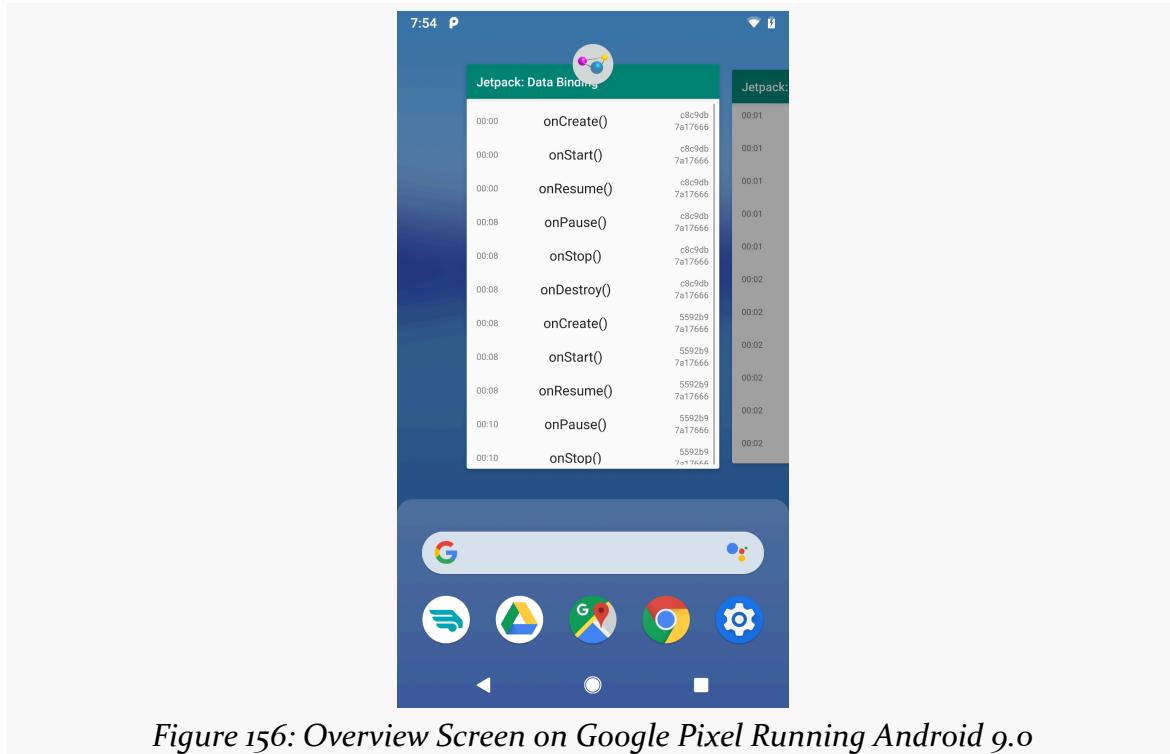


Figure 156: Overview Screen on Google Pixel Running Android 9.0

UNDERSTANDING PROCESSES

...while on Android 5.0, you might see:

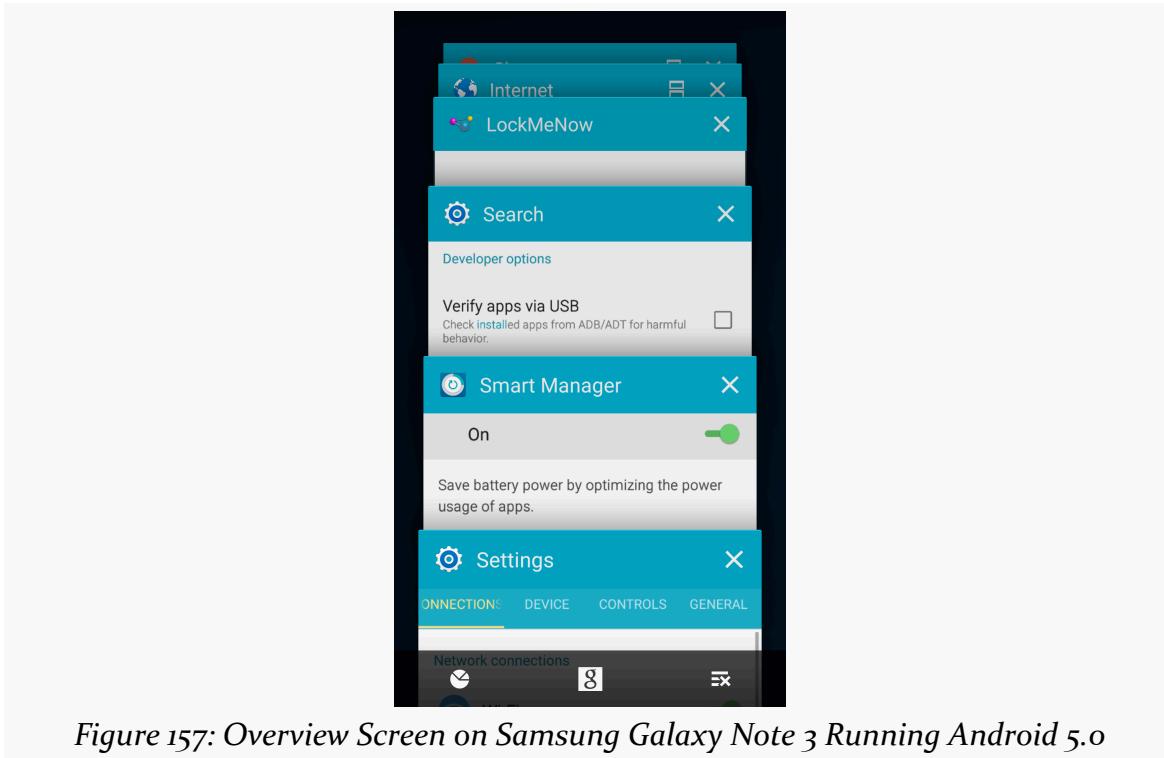


Figure 157: Overview Screen on Samsung Galaxy Note 3 Running Android 5.0

Tasks are the sort of thing that developers often ignore early on. However, many apps eventually do need to address task management.

What is a Task?

Tasks are a bit like tabs in a Web browser.

In a tabbed Web browser, each tab operates fairly independently from other tabs. In particular, each tab has its own “back stack”: pressing the BACK button affects your current tab, not other tabs that happen to be open.

Similarly, a task in Android represents a stack of activities. One activity can open other activities, in the same app or even in different apps, as we will explore [in an upcoming chapter](#). By default, those activities will all belong to the same task. Each time the user presses the BACK button, it destroys the current foreground activity and returns control to whatever activity preceded it. If the user presses BACK from the first (“root”) activity of the task, the task itself is destroyed and the user returns to the home screen.

OK, So Why Do I Care?

The point of tasks is to allow users to rapidly move between apps that have been used recently. In principle, this is the same as a desktop operating system allowing the user to move between opened apps.

However, there is one practical difference: mobile devices usually have a lot less RAM. While newer high-end Android devices might have a lot of RAM, older or less-expensive Android devices might not have very much. As a result, we cannot have lots of running processes — the out-of-memory killer will eventually have to start killing off those processes to free up memory for yet other processes.

However, once an app's process is terminated, everything it held onto in memory is now gone. It has to reload data from disk or from the network. Moreover, it loses all “context” by default: the app does not necessarily remember where the user had been in the app. Instead, by default, if the user goes back to the app, the app will start over from scratch.

To help deal with this, Android keeps track of some amount of state associated with your app. If:

- The user navigates away from your app,
- Android terminates your app's process, and
- The user attempts to return to your app within 30 minutes of having left it

...then Android will try to restore your app and its task to the state it had been in when the user left. In particular, for the purposes of this chapter, Android will hand some “instance state” back to your activity, and you can try to use that to pretend to the user that your activity had been around all along, even though in reality this is a brand-new activity instance in a brand-new process.

Instance State

So, with that in mind, let's talk a bit about “instance state”.

Why Are We Passed a Bundle in onCreate()?

You may have noticed that all of our activities in the sample apps so far have an `onCreate()` function that takes a `Bundle` as a parameter. Each of the samples has done work in `onCreate()`... and each of the samples has ignored this `Bundle`.

UNDERSTANDING PROCESSES

This Bundle is known as the “saved instance state”.

A Bundle is a key-value store, using strings for keys and a limited set of data types for the values. Mostly, a Bundle is designed to hold simple bits of data: Boolean, Int, Long, String, etc.

Normally, when our activity is created, the Bundle parameter to `onCreate()` is null. That is why in Kotlin it is typed as `Bundle?` instead of `Bundle`.

If that Bundle is not null, it represents some state from some previous instance of our activity, where that previous instance either:

- Had been destroyed as part of a configuration change, or
- Had been nuked from orbit as part of Android terminating the app’s process, but the user happened to try returning to that app within 30 minutes of having left it

Our job is to use that Bundle to help set up our user interface.

Nowadays, for the configuration-change scenario, we usually use a `ViewModel` or something like that. However, a `ViewModel` also gets nuked from orbit when the process is terminated, so that does not help us for the “return in 30 minutes” scenario.

When Do We Fill In the Instance State Bundle?

Your activity can optionally override an `onSaveInstanceState()` function. This will be passed a Bundle, and you can fill data into that Bundle.

Typically, you chain to the superclass, as `Activity` will put things into the Bundle related to your current on-screen widgets. In particular, `Activity` will save some user-mutable data from those widgets, such as:

- The text that the user has typed into an `EditText`
- The checked states of `CompoundButton` widgets like `Switch` or `CheckBox`
- The position of a `SeekBar`

When you chain to the superclass implementation of `onSaveInstanceState()`, `Activity` will save all that data for you. In addition, you can put your own data into the Bundle.

When Do We Get the Instance State Bundle?

Your activity will get a copy of that Bundle back in two places:

- `onCreate()`, and
- `onRestoreInstanceState()`

`onCreate()` is always called for a new activity instance. However, there may not be any instance state to restore, so its Bundle parameter may be null.

`onRestoreInstanceState()` is only called if there is instance state to restore, so its Bundle will never be null.

Your job is to look for this Bundle and apply any data from it that you put into the Bundle in `onSaveInstanceState()`.

What Are the Limits on the Bundle?

As noted above, Bundle does not support arbitrary data types. For your own custom classes, you can address this limitation via `Parcelable`, as we will see [in the next section](#). But you should not assume that you can put any sort of data into the Bundle.

There is also a size limitation. The details are rather complicated, but the general rule of thumb is “keep your Bundle well under 1MB”.

The instance state Bundle is not designed to be a replacement for a local database or a server. Use it for in-flight data that you cannot easily load again and that should fit the data type and size limitations. Use your `ViewModel` for data that you have loaded from disk or the network that you would prefer not to load again after a configuration change, and live with the fact that you *will* need to load that data again after your process has been terminated.

Pondering `Parcelable`

In the `LifecycleList` sample app from [the previous chapter](#), our `EventViewModel` held onto two pieces of data:

- The time when the `EventViewModel` was created, and
- The list of `Event` objects representing the lifecycle events

UNDERSTANDING PROCESSES

If our process is terminated, but the user returns to us with 30 minutes, ideally we would still have this data. But our `EventViewModel` is gone and will be replaced by a brand-new instance in the brand-new process. We could store this data in a file or database, but unless the list of `Event` objects is really long, we could put this data into the saved instance state `Bundle`.

Except for one problem: `Bundle` does not know anything about `Event`.

In order to be able to put our `Event` objects into the `Bundle`, we need to make `Event` be `Parcelable`.

`Parcelable` is a marker interface, reminiscent of Java's `Serializable`, that shows up in many places in the Android SDK. A `Parcelable` object is one that can be placed into a `Parcel`. A `Parcel` is the primary vehicle for passing data between processes in Android's inter-process communication (IPC) framework. And, of note, `Bundle` supports `Parcelable` objects.

You have three major approaches for adding `Parcelable` capabilities to your classes in Android:

1. Use an annotation processor that will add in the appropriate bits of magic for you
2. Use a code generator site or tool that will take your existing class as input and give you the `Parcelable`-enabled rendition as output
3. Just do it yourself

Parcelable by Annotation

Annotation processors take `@Things @That @LookLike @These("snippets")` and augment your code. In particular, they can generate code for you to save you typing in a bunch of boilerplate.

Kotlin users have easy access to a `@Parcelize` annotation that can make simple classes — such as `Event` — be `Parcelable`.

This is considered an experimental feature, so you need to opt into it by adding this closure to the `android` closure of your module's `build.gradle` file:

```
androidExtensions {  
    experimental = true  
}
```

UNDERSTANDING PROCESSES

(from [InstanceState/build.gradle](#))

Then, to add Parcelable support to a Kotlin class, just:

- Have that class declare that it implements the Parcelable interface, though you do not need to override any of its methods, and
- Add the @Parcelize annotation to the class:

```
package com.commonware.jetpack.sampler.state

import android.os.Parcelable
import android.os.SystemClock
import kotlinx.android.parcel.Parcelize

@Parcelize
data class Event(
    val message: String,
    val activityHash: Int,
    val viewmodelHash: Int,
    val timestamp: Long = SystemClock.elapsedRealtime()
) : Parcelable
```

(from [InstanceState/src/main/java/com/commonware/jetpack/sampler/state/Event.kt](#))

Here we have a revised version of the Event class that adds the @Parcelize annotation and the Parcelable interface. The actual code to support the Parcelable interface will be code-generated, and so we can skip it.

There are [many third-party libraries](#) that support Parcelable, including many annotation processors for Java that can accomplish a similar thing to what we get with Kotlin.

Parcelable by Code Generation

[The Parcelabler Web site](#) is a code generator. You paste in a simple Java class, with the class declaration and data members:

```
class Event {
    final long timestamp = SystemClock.elapsedRealtime();
    final String message;
    final int activityHash;
    final int viewmodelHash;

    Event(String message, int activityHash, int viewmodelHash) {
        this.message = message;
```

UNDERSTANDING PROCESSES

```
    this.activityHash = activityHash;
    this.viewmodelHash = viewmodelHash;
}
}
```

and it gives you an output class that adds the Parcelable logic:

```
package com.commonsware.jetpack.samplerj.state;

import android.os.Parcel;
import android.os.Parcelable;
import android.os.SystemClock;

class Event implements Parcelable {
    final long timestamp;
    final String message;
    final int activityHash;
    final int.viewmodelHash;

    Event(String message, int activityHash, int.viewmodelHash) {
        this.message = message;
        this.activityHash = activityHash;
        this.viewmodelHash =.viewmodelHash;
        this.timestamp = SystemClock.elapsedRealtime();
    }

    protected Event(Parcel in) {
        timestamp = in.readLong();
        message = in.readString();
        activityHash = in.readInt();
       .viewmodelHash = in.readInt();
    }

    @Override
    public int describeContents() {
        return 0;
    }

    @Override
    public void writeToParcel(Parcel dest, int flags) {
        dest.writeLong(timestamp);
        dest.writeString(message);
        dest.writeInt(activityHash);
        dest.writeInt.viewmodelHash;
    }

    @SuppressWarnings("unused")
    public static final Parcelable.Creator<Event> CREATOR = new Parcelable.Creator<Event>() {
        @Override
        public Event createFromParcel(Parcel in) {
            return new Event(in);
        }

        @Override
        public Event[] newArray(int size) {
            return new Event[size];
        }
    };
}
```

UNDERSTANDING PROCESSES

```
}
```

(from [InstanceState/src/main/java/com/commonsware/jetpack/samplerj/state/Event.java](#))

We will see in the next section what all of that code does for us, as part of understanding how to build it by hand.

However, the Parcelabler Web site has some limitations in its Java parsing, and so the more complex your Java class, the more likely it is that the Parcelabler site will have difficulty understanding it and blending in the Parcelable logic.

Parcelable by Hand

Adding Parcelable support yourself is not especially difficult, though it is a bit tedious.

The Parcelable Interface

The first steps is to add the Parcelable interface to the class. Immediately, your IDE should start complaining that you need to implement two methods to satisfy the Parcelable interface.

The easier of the two methods is `describeContents()`, where you will return 0, most likely.

The other method you will need to implement is `writeToParcel()`. You are passed in two parameters: a very important `Parcel`, and a usually-ignored `int` named `flags`.

Your job, in `writeToParcel()`, is to call a series of `write...()` methods on the `Parcel` to write out all data members of this object that should be considered part of the object as it is passed across process boundaries. There are dozens of type-safe methods for writing data into the `Parcel`, including:

- methods that write individual primitives (e.g., `.writeInt()`) or Java arrays of primitives (e.g., `writeStringArray()`)
- `writeBundle()`, for writing out a `Bundle`
- `writeParcelable()` and `writeParcelableArray()`, for writing out other objects that implement Parcelable
- various specialized methods for particular data types (e.g., `writeSizeF()`) or interfaces (e.g., `writeSerializable()`)

In the case of the generated Event code shown earlier in this chapter,

UNDERSTANDING PROCESSES

`writeToParcel()` writes out each of our four fields:

```
@Override  
public void writeToParcel(Parcel dest, int flags) {  
    dest.writeLong(timestamp);  
    dest.writeString(message);  
    dest.writeInt(activityHash);  
    dest.writeInt(viewmodelHash);  
}
```

(from [InstanceState/src/main/java/com/commonsware/jetpack/samplerj/state/Event.java](#))

The CREATOR

When Android tries reading objects in from a `Parcel`, and it encounters an instance of your `Parcelable` class, it will retrieve a static `CREATOR` object that must be defined on that class. The `CREATOR` is an instance of `Parcelable.Creator`, using generics to tie it to the type of your class:

```
@SuppressWarnings("unused")  
public static final Parcelable.Creator<Event> CREATOR = new Parcelable.Creator<Event>() {  
    @Override  
    public Event createFromParcel(Parcel in) {  
        return new Event(in);  
    }  
  
    @Override  
    public Event[] newArray(int size) {  
        return new Event[size];  
    }  
};
```

(from [InstanceState/src/main/java/com/commonsware/jetpack/samplerj/state/Event.java](#))

The `@SuppressWarnings("unused")` annotation is because the IDE will think that this `CREATOR` instance is not referred to anywhere. That is because it will only be used via Java reflection.

The `CREATOR` will need two methods. `createFromParcel()`, given a `Parcel`, needs to return an instance of your class populated from that `Parcel`. `newArray()`, given a size, needs to return a type-safe array of your class.

The typical implementation of `createFromParcel()` will delegate the actual work to a protected or private constructor on your class that takes the `Parcel` as input:

```
protected Event(Parcel in) {  
    timestamp = in.readLong();
```

UNDERSTANDING PROCESSES

```
    message = in.readString();
    activityHash = in.readInt();
    viewmodelHash = in.readInt();
}
```

(from [InstanceState/src/main/java/com/commonsware/jetpack/samplerj/state/Event.java](#))

You need to read in the *same* values that you wrote out to the `Parcel`, and in the same order.

A State-Aware ViewModel

With all that in mind, let's look at a version of the `LifecycleList` sample that takes instance state into account.

The `InstanceState` sample module in the [Sampler](#) and [SamplerJ](#) projects use an `EventViewModel` that still holds our events and start time. However, in this case, we also store that data in the saved instance state Bundle and use that data when creating a new `EventViewModel` instance in a new process.

The `ViewModelProvider.Factory`

We get our `EventViewModel` in `onCreate()` of our activity. We are handed our saved instance state Bundle in `onCreate()` of our activity. So, we should be able to get the Bundle to the `EventViewModel` in `onCreate()` of our activity.

There are two approaches for doing this.

One is to just have some sort of `onRestoreInstanceState()` method on the `EventViewModel`, where we pass in the Bundle. This works, and it is a fairly simple approach. And for `EventViewModel`, this would work.

However, even better would be to pass the Bundle into our `EventViewModel` constructor. That way, we could initialize the `EventViewModel` instance as it is being created. For example, in Kotlin, we might be able to use `val` instead of `var` for our properties.

However, we do not create instances of `EventViewModel` in the original version of this app. That is handled for us when we call `get()` on the `ViewModelProvider` supplied to us via the call to `ViewModelProviders.of()`.

UNDERSTANDING PROCESSES

The way to get around that is to create and use a `ViewModelProvider.Factory`. This is a class that knows how to create instances of our `ViewModel` as needed, and we can register it with `ViewModelProviders` to have that Factory create our `EventViewModel` and call a custom constructor.

So, to that end, we have `EventViewModelFactory`, both in Java:

```
package com.commonsware.jetpack.samplerj.state;

import android.os.Bundle;
import androidx.annotation.NonNull;
import androidx.lifecycle.ViewModel;
import androidx.lifecycle.ViewModelProvider;

class EventViewModelFactory implements ViewModelProvider.Factory {
    private final Bundle state;

    EventViewModelFactory(Bundle state) {
        this.state = state;
    }

    @NonNull
    @Override
    public <T extends ViewModel> T create(@NonNull Class<T> modelClass) {
        return (T) new EventViewModel(state);
    }
}
```

(from [InstanceState/src/main/java/com/commonsware/jetpack/samplerj/state/EventViewModelFactory.java](#))

...and Kotlin:

```
package com.commonsware.jetpack.sampler.state

import android.os.Bundle
import androidx.lifecycle.ViewModel
import androidx.lifecycle.ViewModelProvider

class EventViewModelFactory(private val state: Bundle?) :
    ViewModelProvider.Factory {

    override fun <T : ViewModel> create(modelClass: Class<T>): T {
        return EventViewModel(state) as T
    }
}
```

(from [InstanceState/src/main/java/com/commonsware/jetpack/sampler/state/EventViewModelFactory.kt](#))

UNDERSTANDING PROCESSES

The EventViewModelFactory takes our saved instance state Bundle as a constructor parameter. In its create() function, it creates an instance of EventViewModel, passing that Bundle to the EventViewModel constructor.

The ViewModelProvider.Factory class is designed to support creating an arbitrary number of arbitrary types of ViewModel classes, which is why create() is passed the Class of ViewModel that we need to create, and why we need to cast the return value to the generic T. In our case, we only support creating EventViewModel.

EventViewModel, in turn, uses the supplied Bundle to set up its properties:

```
package com.commonsware.jetpack.samplerj.state;

import android.os.Bundle;
import android.os.SystemClock;
import java.util.ArrayList;
import java.util.Random;
import androidx.lifecycle.ViewModel;

public class EventViewModel extends ViewModel {
    private static final String STATE_EVENTS = "events";
    private static final String STATE_START_TIME = "startTime";

    final ArrayList<Event> events;
    final long startTime;
    private final int id = new Random().nextInt();

    EventViewModel(Bundle state) {
        if (state != null) {
            events = state.getParcelableArrayList(STATE_EVENTS);
            startTime = state.getLong(STATE_START_TIME);
        } else {
            events = new ArrayList<>();
            startTime = SystemClock.elapsedRealtime();
        }
    }

    void addEvent(String message, int activityHash) {
        events.add(new Event(message, activityHash, id));
    }

    void onSaveInstanceState(Bundle state) {
        state.putParcelableArrayList(STATE_EVENTS, events);
        state.putLong(STATE_START_TIME, startTime);
    }
}
```

UNDERSTANDING PROCESSES

```
@Override  
protected void onCleared() {  
    events.clear();  
}  
}
```

(from [InstanceState/src/main/java/com/commonsware/jetpack/samplerj/state/EventViewModel.java](#))

```
package com.commonsware.jetpack.sampler.state  
  
import android.os.Bundle  
import android.os.SystemClock  
import androidx.lifecycle.ViewModel  
import java.util.*  
  
private const val STATE_EVENTS = "events"  
private const val STATE_START_TIME = "startTime"  
  
class EventViewModel(state: Bundle?) : ViewModel() {  
    val events =  
        state?.getParcelableArrayList<Event>(STATE_EVENTS) ?: arrayListOf()  
    val startTime =  
        state?.getLong(STATE_START_TIME) ?: SystemClock.elapsedRealtime()  
    private val id = Random().nextInt()  
  
    fun addEvent(message: String, activityId: Int) {  
        events.add(Event(message, activityId, id))  
    }  
  
    fun onSaveInstanceState(state: Bundle) {  
        state.putParcelableArrayList(STATE_EVENTS, events)  
        state.putLong(STATE_START_TIME, startTime)  
    }  
  
    override fun onCleared() {  
        events.clear()  
    }  
}
```

(from [InstanceState/src/main/java/com/commonsware/jetpack/sampler/state/EventViewModel.kt](#))

EventViewModel also now has an `onSaveInstanceState()` function that fills in the supplied Bundle with the current data. This is our own function, not one from the `ViewModel` system, so we will need to call it ourselves.

In terms of the Bundle contents, our `startTime` can be easily put into the Bundle as a

UNDERSTANDING PROCESSES

Long. Our events require us to make Event be Parcelable — the code from earlier in this chapter showing various Parcelable approaches shows the Event implementation from this module. We can then use getParcelableArrayList() and setParcelableArrayList(), as our events is an ArrayList of a Parcelable class (Event).

Applying the Factory

In onCreate() of MainActivity, we now need to use a two-parameter version of of() that takes an instance of our ViewModelProvider.Factory as the second parameter:

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

    RecyclerView items = findViewById(R.id.items);

    vm = ViewModelProviders
        .of(this, new EventViewModelFactory(savedInstanceState))
        .get(EventViewModel.class);
    adapter = new EventAdapter(getLayoutInflater(), vm.startTime);
    addEvent("onCreate()");

    items.setLayoutManager(new LinearLayoutManager(this));
    items.addItemDecoration(
        new DividerItemDecoration(this, DividerItemDecoration.VERTICAL));
    items.setAdapter(adapter);
}
```

(from [InstanceState/src/main/java/com/commonsware/jetpack/samplerj/state/MainActivity.java](#))

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

    val items = findViewById<RecyclerView>(R.id.items)

    vm = ViewModelProviders
        .of(this, EventViewModelFactory(savedInstanceState))
        .get()
    adapter = EventAdapter(layoutInflater, vm.startTime)
    addEvent("onCreate()")

    items.layoutManager = LinearLayoutManager(this)
```

UNDERSTANDING PROCESSES

```
    items.addItemDecoration(  
        DividerItemDecoration(this, DividerItemDecoration.VERTICAL)  
    )  
    items.adapter = adapter  
}
```

(from [InstanceState/src/main/java/com/commonsware/jetpack/sampler/state/MainActivity.kt](#))

We use our `EventViewModelFactory`, so if we need a new `EventViewModel`, the factory creates it using our `Bundle`. If we already have an `EventViewModel`, though, we still just get handed the existing instance.

In addition, we override the `onSaveInstanceState()` function in `MainActivity` and pass that call along to the `EventViewModel`:

```
@Override  
protected void onSaveInstanceState(Bundle outState) {  
    super.onSaveInstanceState(outState);  
  
    vm.onSaveInstanceState(outState);  
}
```

(from [InstanceState/src/main/java/com/commonsware/jetpack/samplerj/state/MainActivity.java](#))

```
override fun onSaveInstanceState(outState: Bundle) {  
    super.onSaveInstanceState(outState)  
  
    vm.onSaveInstanceState(outState)  
}
```

(from [InstanceState/src/main/java/com/commonsware/jetpack/sampler/state/MainActivity.kt](#))

This ensures that our `EventViewModel` can add its data to the `Bundle`, to get it back in some future process if appropriate.

The Results

Seeing this work is tricky. We need to have our process be terminated while leaving the task alone. That means we cannot swipe our task off of the overview screen, as that terminates the task. Similarly, the stop-process buttons in Android Studio toolbars (with a red square icon) seem to terminate the task as well.

One way to see this work is to use the command line.

UNDERSTANDING PROCESSES

The Android SDK ships with an adb command-line utility. We can use this to communicate with a running emulator or debuggable device. You will find adb in the platform-tools/ directory of your Android SDK installation.

In particular:

- adb shell says “give me access to a Linux-style shell inside of Android”
- adb shell am says “run the am command in that shell”, where am is the “activity manager” tool
- adb shell am kill ... says “kill the process identified by the supplied application ID” (shown here as ...)

To experiment with instance states:

- Run the sample app
- Undergo configuration changes, if desired
- Press HOME, to move the app to the background and trigger an onSaveInstanceState() call
- Run adb shell am kill com.commonsware.jetpack.sampler.state (for the Kotlin edition) or adb shell am kill com.commonsware.jetpack.samplerj.state (for the Java edition) to terminate the background process
- Bring up the overview screen and tap on the entry for this sample app

UNDERSTANDING PROCESSES

In the original `LifecycleList` sample, configuration changes would show new instance hash codes for the activity, but the `EventViewModel` instance hash code would be the same, as we reuse the existing viewmodel on a configuration change. Now, though, we are terminating the process, so we should see a new viewmodel hash code for the newer events:



Figure 158: InstanceState Sample App, Following Above Script

However, we still have the original events and the original start time — we did not start over with an empty `EventViewModel`, because we populated it from the saved instance state Bundle.

Binding Your Data

So far, to update our UI, we have been pushing data into widgets from our Java or Kotlin code. For example, we have been updating the `text` property of a `TextView`.

Data binding, in general, refers to frameworks or libraries designed to *pull* data into the UI. UI definitions — such as Android layout resources — contain information about how to populate widgets from supplied model objects.

This chapter explores Android’s data binding support and how to use it to perhaps simplify your Android app development. And, we will also see why the word “perhaps” is in that previous sentence.

The Basic Steps

The `.DataBind` sample module in the [Sampler](#) and [SamplerJ](#) projects adds to the `InstanceState` sample from the preceding chapter. The primary difference is that we will fill in the `RecyclerView` rows using data binding.

Enabling Data Binding

Data binding is an opt-in feature, in part because it can slow down the build process. For small projects like the ones in this book, you will not notice the overhead of the data binding tools, but that overhead becomes more annoying as projects get larger.

To opt into data binding, we need to enable it, by adding another closure to our module’s `build.gradle` file’s `android` closure:

```
dataBinding {  
    enabled = true
```

BINDING YOUR DATA

```
}
```

(from [DataBind/build.gradle](#))

Also, Kotlin projects using data binding should add the `kotlin-kapt` plugin:

```
apply plugin: 'kotlin-kapt'
```

(from [DataBind/build.gradle](#))

This plugin enables annotation processing for Kotlin source files. Using annotations is not absolutely required in a data binding project, but it is somewhat common, and this plugin is needed so the data binding code can process those annotations.

Augmenting the Layout

The real fun begins with the layout for our `RecyclerView` row. The original edition of this layout resource was a typical `ConstraintLayout` with our `TextView` widgets:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.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="wrap_content"
    android:padding="@dimen/content_padding">

    <TextView
        android:id="@+id/activityHash"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        tools:text="0x12345678" />

    <TextView
        android:id="@+id/viewmodelHash"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/activityHash"
        tools:text="0x90ABCDEF" />
```

BINDING YOUR DATA

```
<androidx.constraintlayout.widget.Barrier
    android:id="@+id/barrier"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginEnd="4dp"
    android:layout_marginStart="4dp"
    app:barrierDirection="start"
    app:constraint_referenced_ids="activityHash,viewmodelHash" />

<TextView
    android:id="@+id/timestamp"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    tools:text="01:23" />

<TextView
    android:id="@+id/message"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textAppearance="?android:attr/textAppearanceLarge"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toStartOf="@+id/barrier"
    app:layout_constraintStart_toEndOf="@+id/timestamp"
    app:layout_constraintTop_toTopOf="parent"
    tools:text="onDestroy()" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

(from [InstanceState/src/main/res/layout/row.xml](#))

We need to make some changes to that in order to leverage data binding:

```
<?xml version="1.0" encoding="utf-8"?>
<layout>

    <data>

        <import type="android.text.format.DateUtils" />

        <variable
            name="event"
            type="com.commonsware.jetpack.sampler.databind.Event" />

        <variable
            name="elapsedSeconds"
```

BINDING YOUR DATA

```
    type="Long" />
</data>

<androidx.constraintlayout.widget.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="wrap_content"
    android:padding="@dimen/content_padding">

    <TextView
        android:id="@+id/activityHash"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@{Integer.toHexString(event.activityHash)}"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        tools:text="0x12345678" />

    <TextView
        android:id="@+id/viewmodelHash"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@{Integer.toHexString(event.viewmodelHash)}"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/activityHash"
        tools:text="0x90ABCDEF" />

    <androidx.constraintlayout.widget.Barrier
        android:id="@+id/barrier"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginEnd="4dp"
        android:layout_marginStart="4dp"
        app:barrierDirection="start"
        app:constraint_referenced_ids="activityHash,viewmodelHash" />

    <TextView
        android:id="@+id/timestamp"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@{DateUtils.formatElapsedTime(elapsedSeconds)}"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        tools:text="01:23" />
```

BINDING YOUR DATA

```
<TextView  
    android:id="@+id/message"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:text="@{event.message}"  
    android:textAppearance="?android:attr/textAppearanceLarge"  
    app:layout_constraintBottom_toBottomOf="parent"  
    app:layout_constraintEnd_toStartOf="@+id/barrier"  
    app:layout_constraintStart_toEndOf="@+id/timestamp"  
    app:layout_constraintTop_toTopOf="parent"  
    tools:text="onDestroy()" />  
  
</androidx.constraintlayout.widget.ConstraintLayout>  
</layout>
```

(from [DataBind/src/main/res/layout/row.xml](#))

First, the entire resource file gets wrapped in a `<layout>` element, on which we can place the android namespace declaration. Only layout resources with a root `<layout>` element are processed by the data binding portion of the build system.

That `<layout>` element then has two children. The second child is our `ConstraintLayout`, representing the root View or ViewGroup for the resource. The first child is a `<data>` element, and that is where we configure how data binding should proceed when this layout resource gets used.

Inside of `<data>` we have three elements.

Two are `<variable>` elements. These identify and describe the objects that we can pull data from. In our case, we have an event variable that is an instance of our Event model class, and we have an `elapsedSeconds` value that represents the number of seconds that have elapsed between the start of the app and this event. As we will see, our Java or Kotlin code will now “bind” those objects into our layout, rather than update widgets directly.

The other element is `<import>`. This works like `import` statements in Java or Kotlin, indicating a particular class that we would like to reference. In the world of data binding, mostly we use `<import>` for classes where we wish to refer to static methods or fields (and their Kotlin equivalents).

Our four `TextView` widgets now have `android:text` attributes, where they had none before. That is because the earlier version of this sample relied upon Java or Kotlin code to push data into the widgets, and now we are going to pull data in using data

binding.

Those `android:text` attributes use “binding expressions”. A binding expression is identified in a data binding-enhanced layout through `@{}` syntax, with the actual expression between the braces. Those expressions use a language syntax that looks a lot like Java or Kotlin expressions, and they can reference:

- Our `<variable>` values
- Global functions and properties on the classes listed in `<import>` elements
- Global functions and properties on any `java.lang` classes
- A few other magic values, such as `context` to get a Context, should we need one
- Properties of other widgets in this same layout resource

The `activityHash` `TextView` has

`android:text="@{Integer.toHexString(event.activityHash)}`". The binding expression works like its Java or Kotlin counterparts, taking our `activityHash` value out of our event and formatting it as a hex string. The text of this `TextView` will then contain that hex string.

The `android:text` attributes of the other `TextView` widgets work the same:

- `viewmodelHash` uses `Integer.toHexString(event.viewmodelHash)` to fill in the text with the hex string of that hash code
- `timestamp` uses `DateUtils.formatElapsedTime(elapsedSeconds)` to format the `elapsedSeconds` value as an elapsed time
- `message` just uses `event.message` to bind from the message from the Event

While we happen to use `android:text` for all four widgets, binding expressions can be applied to just about any attribute, such as `android:checked` for the checked state of a `CompoundButton`.

Updating the Model

Data binding has its limits. One limit is that it can only access public functions and properties. So, we need to ensure that everything we need is now `public` in our `Event` class, both in Java:

```
package com.commonsware.jetpack.samplerj.databind;

import android.os.Parcel;
import android.os.Parcelable;
```

BINDING YOUR DATA

```
import android.os.SystemClock;

public class Event implements Parcelable {
    public final long timestamp;
    public final String message;
    public final int activityHash;
    public final int viewmodelHash;

    Event(String message, int activityHash, int viewmodelHash) {
        this.message = message;
        this.activityHash = activityHash;
        this.viewmodelHash = viewmodelHash;
        this.timestamp = SystemClock.elapsedRealtime();
    }

    protected Event(Parcel in) {
        timestamp = in.readLong();
        message = in.readString();
        activityHash = in.readInt();
        viewmodelHash = in.readInt();
    }

    @Override
    public int describeContents() {
        return 0;
    }

    @Override
    public void writeToParcel(Parcel dest, int flags) {
        dest.writeLong(timestamp);
        dest.writeString(message);
        dest.writeInt(activityHash);
        dest.writeInt(viewmodelHash);
    }

    @SuppressWarnings("unused")
    public static final Parcelable.Creator<Event> CREATOR = new Parcelable.Creator<Event>() {
        @Override
        public Event createFromParcel(Parcel in) {
            return new Event(in);
        }

        @Override
        public Event[] newArray(int size) {
            return new Event[size];
        }
    };
}
```

(from [DataBind/src/main/java/com/commonsware/jetpack/samplerj/databind/Event.java](#))

...and Kotlin:

```
package com.commonsware.jetpack.sampler.databind

import android.os.Parcelable
import android.os.SystemClock
```

BINDING YOUR DATA

```
import kotlinx.android.parcel.Parcelize

@Parcelize
data class Event(
    val message: String,
    val activityHash: Int,
    val viewmodelHash: Int,
    val timestamp: Long = SystemClock.elapsedRealtime()
) : Parcelable
```

(from [DataBind/src/main/java/com/commonsware/jetpack/sampler/databind/Event.kt](#))

Applying the Binding

The layout configures one side of the binding: pulling data into widgets. We still need to do some work to configure the other side of the binding: supplying the source of that data. In the case of this example, we need to provide the Event object for this layout resource.

That is handled via some modifications to our `EventAdapter` and `EventViewHolder`, to get at a “binding object” for an inflated layout and then call functions on it to supply our variables.

Creating the Binding

When we use `<layout>` in a layout resource and set up the layout side of the data binding system, the build system code-generates a Java class associated with that layout file. The class name is derived from the layout name, where `names_like_this` get converted into `NamesLikeThis` and have `Binding` appended. So, since our layout resource was `row.xml`, we get `RowBinding`. This is code-generated into a databinding Java sub-package of the package name from the manifest. Hence, the fully-qualified import statement for this class is:

```
import com.commonsware.jetpack.sampler.databinding.RowBinding;
```

(normally, projects would not have `databind` in their own application IDs, so the near-duplication that you see here is peculiar to this project)

This is a subclass of `ViewDataBinding`, supplied by the `androidx.databinding` libraries that are added to your project by enabling data binding in your `build.gradle` file.

Creating an instance of the binding also inflates the associated layout. Your binding

BINDING YOUR DATA

class has a number of factory methods for inflating the layout and creating the binding. These mirror other methods that you have used elsewhere:

- `setContentView()`, taking an `Activity` and the layout resource ID as parameters, inflates the layout, passes the result to `setContentView()` on the `Activity`, and creates the binding
- `inflate()`, with a variety of parameter list options, just inflates the layout using a `LayoutInflater`, and creates the binding

Our revised version of `EventAdapter` uses the three-parameter flavor of `inflate()`, which takes a `LayoutInflater` (obtained from the hosting activity), the parent container, and `false`. This mirrors the `inflate()` one would use on `LayoutInflater` itself, except that it also gives us our binding. We use this in `onCreateViewHolder()` and pass the `RowBinding` into our `EventViewHolder`:

```
@NonNull  
@Override  
public EventViewHolder onCreateViewHolder(@NonNull ViewGroup parent,  
                                         int viewType) {  
    RowBinding binding = RowBinding.inflate(inflater, parent, false);  
  
    return new EventViewHolder(binding, startTime);  
}
```

(from [DataBind/src/main/java/com/commonsware/jetpack/samplerj/databind/EventAdapter.java](#))

```
override fun onCreateViewHolder(  
    parent: ViewGroup,  
    viewType: Int  
): EventViewHolder {  
    val binding = RowBinding.inflate(inflater, parent, false)  
  
    return EventViewHolder(binding, startTime)  
}
```

(from [DataBind/src/main/java/com/commonsware/jetpack/sampler/databind/EventAdapter.kt](#))

Pouring the Model into the Binding

The generated binding class will have setters for each `<variable>` in our `<data>` element in the layout. Setter names are generated from the variable names using standard JavaBean conventions, so our `event` variable becomes `setEvent()` and our `elapsedSeconds` variable becomes `setElapsedSeconds()`. When we call `setEvent()` and `setElapsedSeconds()`, the generated code will use those objects to populate our

BINDING YOUR DATA

TextView widgets, applying the binding expression from our android:text attributes.

To accomplish this, our revised EventViewHolder holds onto the RowBinding and uses it in bindTo():

```
package com.commonsware.jetpack.samplerj.databind;

import com.commonsware.jetpack.samplerj.databind.databinding.RowBinding;
import androidx.recyclerview.widget.RecyclerView;

class EventViewHolder extends RecyclerView.ViewHolder {
    private final long startTime;
    private final RowBinding row;

    EventViewHolder(RowBinding row, long startTime) {
        super(row.getRoot());

        this.row = row;
        this.startTime = startTime;
    }

    void bindTo(Event event) {
        row.setElapsedSeconds((event.timestamp - startTime)/1000);
        row.setEvent(event);
        row.executePendingBindings();
    }
}
```

(from [DataBind/src/main/java/com/commonsware/jetpack/samplerj/databind/EventViewHolder.java](#))

```
package com.commonsware.jetpack.sampler.databind

import androidx.recyclerview.widget.RecyclerView
import com.commonsware.jetpack.sampler.databind.databinding.RowBinding

class EventViewHolder(val row: RowBinding, private val startTime: Long) :
    RecyclerView.ViewHolder(row.root) {
    fun bindTo(event: Event) {
        row.elapsedSeconds = (event.timestamp - startTime) / 1000
        row.event = event
        row.executePendingBindings()
    }
}
```

(from [DataBind/src/main/java/com/commonsware/jetpack/sampler/databind/EventViewHolder.kt](#))

BINDING YOUR DATA

We also call `executePendingBindings()` on the `RowBinding`. This is needed when populating items in a `RecyclerView`, to ensure that those binding expressions are evaluated and applied immediately rather than a bit later. In cases not involving `RecyclerView`, though, it is usually safe to skip the `executePendingBindings()` call.

Getting the Root View

When we chain to the superclass constructor in a `RecyclerView.ViewHolder`, we need to pass the `View` that is the UI for that particular bit of UI that the `ViewHolder` is managing. To get the root view of a layout associated with a binding object, call `getRoot()` on the binding object. That's what `EventViewHolder` does, passing the `getRoot()` results to the `RecyclerView.ViewHolder` constructor.

Retrieving Widgets from the Binding

The module's `activity_main` layout is also wrapped in a `<layout>` element, even though we have no `<data>` element and are not performing data binding:

```
<?xml version="1.0" encoding="utf-8"?>
<layout>

    <androidx.constraintlayout.widget.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"
        android:padding="@dimen/content_padding"
        tools:context=".MainActivity">

        <androidx.recyclerview.widget.RecyclerView
            android:id="@+id/items"
            android:layout_width="0dp"
            android:layout_height="0dp"
            app:layout_constraintBottom_toBottomOf="parent"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintTop_toTopOf="parent" />

    </androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```

(from [DataBind/src/main/res/layout/activity_main.xml](#))

BINDING YOUR DATA

This is here to demonstrate another feature of data binding: providing direct accessors to widgets.

In earlier versions of this sample, our Java edition of `MainActivity` used `setContentView()` to load `activity_main`, then used `findViewById()` to retrieve the `RecyclerView`. The DataBind edition of `MainActivity` gets a `ActivityMainBinding` instead and can get access to the `RecyclerView` directly from it:

```
@Override  
protected void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
  
    ActivityMainBinding binding =  
        DataBindingUtil.setContentView(this, R.layout.activity_main);  
  
    vm = ViewModelProviders  
        .of(this, new EventViewModelFactory(savedInstanceState))  
        .get(EventViewModel.class);  
    adapter = new EventAdapter(getLayoutInflator(), vm.startTime);  
    addEvent("onCreate()");  
  
    binding.items.setLayoutManager(new LinearLayoutManager(this));  
    binding.items.addItemDecoration(  
        new DividerItemDecoration(this, DividerItemDecoration.VERTICAL));  
    binding.items.setAdapter(adapter);  
}
```

(from [DataBind/src/main/java/com/commonsware/jetpack/samplerj/databind/MainActivity.java](#))

To use a data binding layout as the content view of an activity, use `DataBindingUtil.setContentView()`. This takes the activity and the layout resource ID as parameters, and it returns the binding object. Along the way, it does the same thing that `setContentView()` on `Activity`, setting up our layout for use for our activity.

Each widget with an `android:id` attribute in the layout resource gets a corresponding field in the binding object. Our `RecyclerView` is called `items`, so we have an `items` field in `ActivityMainBinding`. That allows us to use `binding.items` to get at the `RecyclerView`, rather than calling `findViewById()` ourselves.

The Kotlin edition of `MainActivity` also uses `DataBindingUtil` and `ActivityMainBinding`. However, the result is not that different than what we had with Kotlin synthetic accessors. If anything, it adds a bit more code:

```
override fun onCreate(savedInstanceState: Bundle?) {
```

BINDING YOUR DATA

```
super.onCreate(savedInstanceState)
val binding =
    DataBindingUtil.setContentView<ActivityMainBinding>(this, R.layout.activity_main)

vm = ViewModelProviders
    .of(this, EventViewModelFactory(savedInstanceState))
    .get()
adapter = EventAdapter(layoutInflater, vm.startTime)
addEvent("onCreate()")

binding.apply {
    items.layoutManager = LinearLayoutManager(this@MainActivity)
    items.addItemDecoration(
        DividerItemDecoration(this@MainActivity, DividerItemDecoration.VERTICAL)
    )
    items.adapter = adapter
}
}
```

(from [DataBind/src/main/java/com/commonsware/jetpack/sampler/databind/MainActivity.kt](#))

Results

Visually, this app is the same as before. Functionally, the app is the same as before. And, from a code complexity standpoint, the app is probably *worse* than before, as we went through a lot of work just to avoid calling `findViewById()` and `setText()` a few times.

Why Bother?

Some Android experts love data binding. Others hate it. Compared to a lot of things in Android app development, opinions of data binding are wide and varied.

Developers that love data binding seem to focus a lot on the “separation of concerns” that data binding helps to enforce. Your Java/Kotlin code can stop thinking about widget details quite so much, with a lot of that code moving to the layout resources.

Detractors point out that:

- Data binding slows down the build process, as there is more code generation that needs to go on
- Data binding can be difficult to debug, as more code is hidden from view, buried in data binding expressions and generated classes
- Data binding adds a bit more bloat to a project, adding three libraries and these generated classes, where the user gains very little from the results

Google promotes data binding as part of the Jetpack. However, whereas some

BINDING YOUR DATA

aspects of Jetpack are nearly unavoidable, data binding is optional. You can use it if you like or skip it if you like.

Defining and Using Styles

Android offers styles and themes, filling the same sort of role that CSS does in Web development. We have seen a little bit about themes, such as having a custom theme that defines the core colors to be used by your app.

On the whole, styles and themes are powerful yet confusing tools in the Android developer's toolbox. The "confusing" aspect stems from documentation that has ranged from "simply not existing" to "limited and mystifying" over the years. As such, most Android app development seems to avoid styles and themes to the extent possible, settling instead for directly configuring widgets in layout files. This works, and for smaller projects it is perfectly reasonable. The larger the project, the greater the likelihood is that you will benefit from using styles and themes, just as a larger Web app is more likely than a smaller one to benefit from a well-designed set of CSS stylesheets.

In this chapter, we will take a slightly "deeper dive" into styles and themes, exploring how you can create your own and apply them to your app's UI.

Styles: DIY DRY

The purpose of styles is to encapsulate a set of attributes that you intend to use repeatedly, conditionally, or otherwise wish to keep separate from your layouts. The primary use case is "don't repeat yourself" (DRY) — if you have a bunch of widgets that look the same, use a style to use a single definition for "look the same", rather than copying the look from widget to widget.

The `CustomStyle` sample module in the [Sampler](#) and [SamplerJ](#) projects is based off of the previous `InstanceState` sample, where we are showing a list of lifecycle events. In this edition of the sample, we will define and apply a style resource to

DEFINING AND USING STYLES

some of the `TextView` widgets in the list rows.

Styles and themes are defined using a style resource. By convention, style resources go in `res/values/styles.xml`, though the actual filename does not matter, so long as it is in a `values` resource directory.

Our `res/values/styles.xml` defines two style resources. One, `AppTheme`, is our custom application theme, and we will revisit it [later in the chapter](#).

The other is `CustomText`:

```
<style name="CustomText">
    <item name="android:textColor">?colorAccent</item>
    <item name="android:textStyle">italic</item>
    <item name="android:fontFamily">monospace</item>
</style>
```

(from [CustomStyle/src/main/res/values/styles.xml](#))

In the next sections, we will explore more about what this style does and how we are using it.

Elements of Style

There are a few elements to consider when working with style resources:

- Where do you put the style attributes to say you want to apply a style?
- What attributes can you define via a style?
- What values can those attributes have in a style definition?

The Locations Where We Use Styles

To indicate that we want to use a `<style>` to style a widget, we apply the `style` attribute to the widget, with a reference to our style resource (e.g., `@style/CustomText`). So, for example, our `res/layout/row.xml` resource uses `style="@style/CustomText"` on the `activityHash` and `viewmodelHash` widgets, but not the others:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
```

DEFINING AND USING STYLES

```
xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:padding="@dimen/content_padding">

<TextView
    android:id="@+id/activityHash"
    style="@style/CustomText"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    tools:text="0x12345678" />

<TextView
    android:id="@+id/viewmodelHash"
    style="@style/CustomText"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/activityHash"
    tools:text="0x90ABCDEF" />

<androidx.constraintlayout.widget.Barrier
    android:id="@+id/barrier"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginEnd="4dp"
    android:layout_marginStart="4dp"
    app:barrierDirection="start"
    app:constraint_referenced_ids="activityHash,viewmodelHash" />

<TextView
    android:id="@+id/timestamp"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    tools:text="01:23" />

<TextView
    android:id="@+id/message"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textAppearance="?android:attr/textAppearanceLarge"
    app:layout_constraintBottom_toBottomOf="parent"
```

DEFINING AND USING STYLES

```
    app:layout_constraintEnd_toStartOf="@+id/barrier"
    app:layout_constraintStart_toEndOf="@+id/timestamp"
    app:layout_constraintTop_toTopOf="parent"
    tools:text="onDestroy()" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

(from [CustomStyle/src/main/res/layout/row.xml](#))

If you apply the `style` attribute to a widget, it affects only that widget.

The `style` attribute can be applied to a container, to affect that container. However, doing this does *not* automatically style widgets that reside inside of the container.

You can also apply a style to an activity or an application as a whole, though then it is referred to as a “theme”, which will be covered [a bit later in this chapter](#).

The Available Attributes

When styling a widget or container, you can apply any of that widget’s or container’s attributes in the style itself. So, if it shows up in the “XML Attributes” or “Inherited XML Attributes” portions of the Android JavaDocs, or it shows up in the Attributes pane of the Android Studio graphical layout designer, you can put it in a style.

If we go back to `CustomText`, we will see that our style has three `<item>` elements, identifying three attributes that we wish to control:

```
<style name="CustomText">
    <item name="android:textColor">?colorAccent</item>
    <item name="android:textStyle">italic</item>
    <item name="android:fontFamily">monospace</item>
</style>
```

(from [CustomStyle/src/main/res/values/styles.xml](#))

Note that Android will ignore invalid styles. If you put an attribute in a style that is unused by something that you apply the style to, the attribute is ignored. It does not crash the app. It so happens that `TextView` has `android:textColor`, `android:textStyle`, and `android:fontFamily` attributes, to control the text color, style (bold, italic, etc.) and font (e.g., `monospace`) respectively. If we applied this style to the `ConstraintLayout` at the root of `row.xml`, all three style attribute values would be ignored, as a `ConstraintLayout` does not have a text color, style, or font.

Also, while layout directives, such as `android:layout_width`, can be put in a style,

DEFINING AND USING STYLES

usually they are not. Styles tend to be limited to other aspects of look-and-feel: colors, fonts, etc.

The Possible Values

Sometimes, the value that you will give those attributes in the style will be some constant, like 30sp or #FFFF0000.

Sometimes, the value will appear to be string, but in reality it is one of a limited number of possible “enumerated” values. For example, android:textStyle supports italic as a value, as the sample project is using in CustomText. It does not support aeroasdfasdf as a value.

Sometimes, the value will be a reference to a resource, such as @dimen/standard_padding or @color/really_red.

Sometimes, though, you want to perform a bit of indirection — you want to apply some other attribute value from the theme you are inheriting from. In that case, you will wind up using the somewhat cryptic ? and ?android:attr/ syntax.

For example, the android:textColor attribute in the CustomText style does not have a value of #D81B60 or @color/colorAccent, even though both of those are the color that we actually wind up applying. Instead, android:textColor has ?colorAccent. This says “go find the colorAccent value defined for our theme and use it here too”. Our theme is AppTheme, and it has a colorAccent value:

```
<style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar">
    <!-- Customize your theme here. -->
    <item name="colorPrimary">@color/colorPrimary</item>
    <item name="colorPrimaryDark">@color/colorPrimaryDark</item>
    <item name="colorAccent">@color/colorAccent</item>
</style>
```

(from [CustomStyle/src/main/res/values/styles.xml](#))

This way, if we change the theme’s colorAccent, we *also* change the android:textColor of CustomText to match.

The Results

We applied the `CustomText` style to two of our four `TextView` widgets, so those two are now red italic monospace:

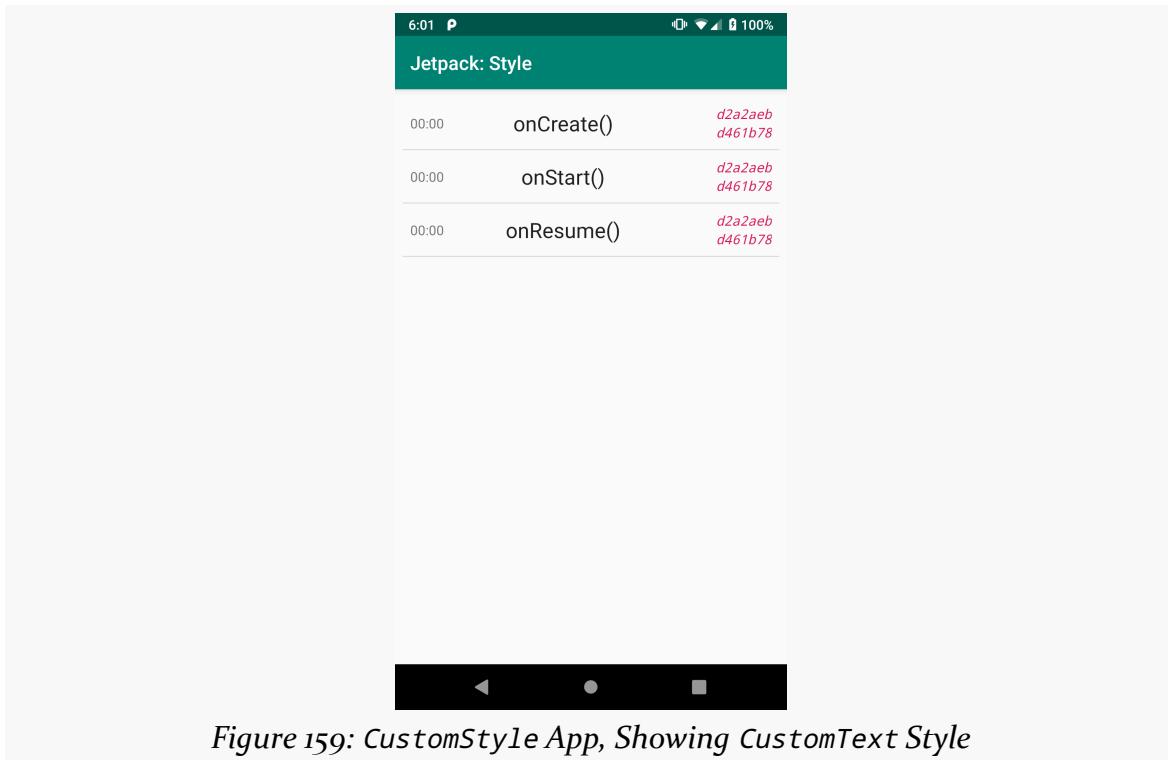


Figure 159: *CustomStyle App, Showing CustomText Style*

Themes: Would a Style By Any Other Name...

Themes are styles applied to an activity or application, via an `android:theme` attribute on the `<activity>` or `<application>` element. They affect properties that apply to the entire activity, such as the default colors to use. They also can define default attributes to use for widgets used by that activity, without manually applying `style` attributes to those widgets.

The `CustomTheme` sample module in the [Sampler](#) and [SamplerJ](#) projects is a variation of the preceding sample, where we want to make `TextView` widgets display red italic monospace text. This time, though, we are going to work with our `AppTheme` style resource as a theme.

The Locations Where We Apply a Theme

A theme usually is applied via an `android:theme` attribute in the manifest. If you apply it to an `<activity>`, that activity will use the designated theme. If you apply it to an `<application>`, that theme will be used for *all* activities in the app... except for any that override it with their own `android:theme` attribute.

If the theme you are applying is your own, just reference it as `@style/....`, just as you would in a `style` attribute of a widget (e.g., `@style/AppTheme`). This includes themes defined in libraries that you are using (`@style/Theme.AppCompat.Dialog`). If the theme you are applying, though, comes from Android, typically you will use a value with `@android:style/` as the prefix, such as `@android:style/Theme.Material.Dialog` or `@android:style/Theme.Material.Light`.

So, our sample app uses `android:theme` on the `<application>`, pointing to `AppTheme`:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest package="com.commonsware.jetpack.sampler.theme"
  xmlns:android="http://schemas.android.com/apk/res/android">

  <application
    android:allowBackup="true"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:supportsRtl="true"
    android:theme="@style/AppTheme">
    <activity android:name=".MainActivity">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />

        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
  </application>

</manifest>
```

(from [CustomTheme/src/main/AndroidManifest.xml](#))

The Theme Declaration

Themes are just style resources.

DEFINING AND USING STYLES

So, our AppTheme appears in `res/values/styles.xml` alongside `CustomText`, though with slight alterations from what we used in the previous sample:

```
<resources>

    <!-- Base application theme. -->
    <style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar">
        <!-- Customize your theme here. -->
        <item name="colorPrimary">@color/colorPrimary</item>
        <item name="colorPrimaryDark">@color/colorPrimaryDark</item>
        <item name="colorAccent">@color/colorAccent</item>
        <item name="android:textViewStyle">@style/CustomText</item>
    </style>

    <style name="CustomText" parent="android:Widget.Material.TextView">
        <item name="android:textColor">?colorAccent</item>
        <item name="android:textStyle">italic</item>
        <item name="android:fontFamily">monospace</item>
    </style>

</resources>
```

(from [CustomTheme/src/main/res/values/styles.xml](#))

The Parent, and an AppCompat Recap

Any style resource can have a `parent` attribute. This is an inheritance model: anything using the style resource gets the attributes defined both directly in that style resource plus any defined by its parent.

A theme always has a parent, and that parent should be another theme. There are hundreds, if not thousands, of attributes to be configured on a theme, so you want to inherit the vast majority of them. `AppTheme` is declared to have `Theme.AppCompat.Light.DarkActionBar` as its parent, so the activity in this app will not only get the attributes defined directly in `AppTheme` but all the attributes defined by `Theme.AppCompat.Light.DarkActionBar`.

`Theme.AppCompat.Light.DarkActionBar` is part of the AppCompat system. If you have an activity that extends `AppCompatActivity` — as all of the ones in this book do — that activity *must* use a theme that inherits from `Theme.AppCompat`. `Theme.AppCompat.Light.DarkActionBar` itself inherits from `Theme.AppCompat`, so `AppTheme` indirectly inherits from `Theme.AppCompat`.

The Style Override

A theme can have attributes that override ones defined in parents. `Theme.AppCompat` declares `colorPrimary`, `colorPrimaryDark`, and `colorAccent` attributes — `AppTheme` overrides them and provides the values that should be used in the app.

One particular class of attributes defined in parents are references to style resources that should be used to style particular types of widgets. For example, `android:textViewStyle` says “this is the style resource that should be used for all `TextView` widgets”. `AppTheme` has this attribute and points it to `CustomText`, so now the `CustomText` defines the default look for all `TextView` widgets used in activities that use `AppTheme`.

However, we now have to consider the parent of a regular (non-theme) style. When we use a `style` attribute to apply a style resource to a widget, the attributes defined in the style are used to override those from the default style for the type of widget, such as `TextView`. The `CustomStyle` sample module did not have a parent for `CustomText`, and the three attributes defined by `CustomText` would override whatever the defaults are for `TextView`. However, in the `CustomTheme` project, `CustomText` is the default, courtesy of it being applied via `android:textViewStyle`. As a result, we need `CustomText` to have values for all relevant attributes, mostly through inheritance from some parent. There are three major possibilities for the `parent` value for an `AppCompat` project like this one:

- If there is an `AppCompat` style for that widget (e.g., `Widget.AppCompat.Button`), use it as the parent
- If there is no `AppCompat` style, and the project has a `minSdkVersion` of 21 or higher, look for a `Material` style for that widget, such as the `android:Widget.Material.TextView` parent used here (the `android:` prefix is because `Widget.Material.TextView` comes from the framework, while `Widget.AppCompat.Button` or `Theme.AppCompat.Light.DarkActionBar` comes from a library)
- If there is no `AppCompat` style, and the project has a lower `minSdkVersion` than 21... the story gets very complicated and is well outside the scope of this book

The Result

Since our `CustomText` is now being applied by default for all `TextView` widgets, our

DEFINING AND USING STYLES

row can go back to having no style attributes:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.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="wrap_content"
    android:padding="@dimen/content_padding">

    <TextView
        android:id="@+id/activityHash"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        tools:text="0x12345678" />

    <TextView
        android:id="@+id/viewmodelHash"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/activityHash"
        tools:text="0x90ABCDEF" />

    <androidx.constraintlayout.widget.Barrier
        android:id="@+id/barrier"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginEnd="4dp"
        android:layout_marginStart="4dp"
        app:barrierDirection="start"
        app:constraint_referenced_ids="activityHash,viewmodelHash" />

    <TextView
        android:id="@+id/timestamp"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        tools:text="01:23" />

    <TextView
```

DEFINING AND USING STYLES

```
    android:id="@+id/message"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textAppearance="?android:attr/textAppearanceLarge"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toStartOf="@+id/barrier"
    app:layout_constraintStart_toEndOf="@+id/timestamp"
    app:layout_constraintTop_toTopOf="parent"
    tools:text="onDestroy()"/> />

</androidx.constraintlayout.widget.ConstraintLayout>
```

(from [CustomTheme/src/main/res/layout/row.xml](#))

Moreover, our red italic monospace look will be applied to all four `TextView` widgets in the row, not just the two that we manually specified in the `CustomStyle` sample:



Figure 16o: *CustomTheme Sample, As Initially Launched*

Context Anti-Pattern: Using Application Everywhere

We covered the `Application` implementation of Context [earlier in the book](#). There,

DEFINING AND USING STYLES

we saw that an Application is a process-wide singleton, so we cannot somehow leak it by holding some reference to it. In effect, it is pre-leaked for us.

Some might wonder why we would ever need any other type of Context. If we need Context for so many things, and we have this one Context that is available all the time... why not use it for everything?

The details are a bit complicated. The fundamental rule is fairly simple, though: never use Application for anything involving the UI, as it may not apply styles and themes correctly.

When it comes time to inflate layouts, we need a LayoutInflator that is aware of the styles and themes to use — otherwise, whatever we requested in those styles and themes will be ignored. But the theme might be specified on a per-activity basis. Right now, we have been looking at apps with a single activity, but it is possible — perhaps even likely — that your app will have [more than one activity](#). Those activities might have different themes. If we get our LayoutInflator from an Activity, it will take the theme into account. If we get our LayoutInflator from the Application, it will not, as the Application has no idea where and how the LayoutInflator will be used.

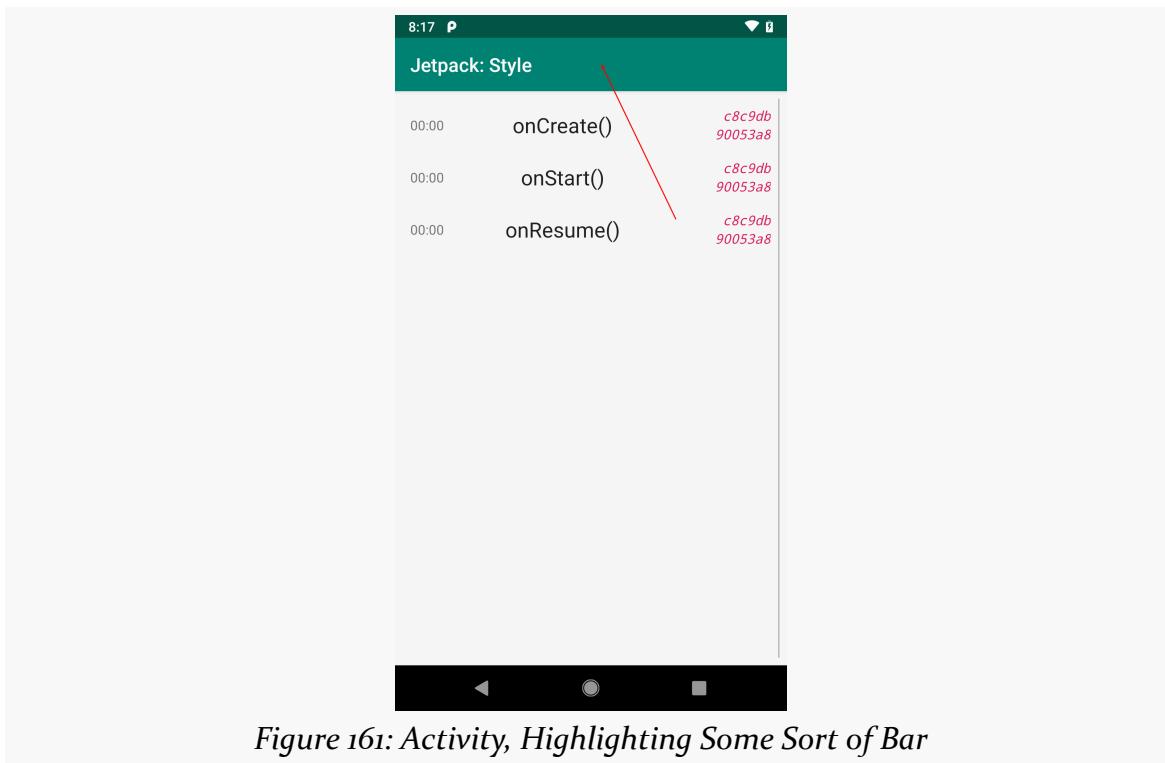
Using Application for non-UI concerns, such as for working with [files](#) or [databases](#), is reasonable, as there Application is as good as Activity or any other kind of Context. You do not have to go out of your way to use Application in general, but if you have your own singletons that need a Context, Applications will help to avoid memory leaks.

So, use Application:

- If you need a long-lived Context, and
- That Context is not going to be used for setting up your UI

Configuring the App Bar

Each of our apps' activities has had this green bar across the top:



To date, all it has done is show our title (`app_name` string resource). However, there is more that we can do with this bar, and we will explore some of that in this chapter.

Our first problem, though, in deciding what this thing is called.

So. Many. Bars.

Many aspects of Android have changed over the years. For example, few Android 1.x/2.x apps had this sort of bar at the top.

Sometimes, when Android changes, Google simply adds some new capability or design feature, and that's it. Sometimes, Google does one thing, then changes course and replaces it with something else. And, on occasion, Google makes so many changes that the result is quite a mess.

This bar is quite a mess.

Part of it is simply the name. Depending on who you ask, this bar could be referred to as:

- the action bar
- a toolbar
- an app bar
- a string of profanities, though usually these describe the developer's relationship with this bar and do not actually name the bar itself

Action Bar

In the beginning, we referred to this as the “action bar”. Activity and other classes had support for showing an action bar and doing things with it, such as dynamically changing the title shown in the bar.

Nowadays, “action bar” is more of a role than an actual thing. We can opt into using the action bar APIs, or we can achieve similar functionality without them.

Toolbar

The “action bar” terminology dominated Android 3.x and 4.x.

Android 5.0 debuted a Toolbar class. Initially, it was thought that the Toolbar was simply a bit of refactoring, giving us a widget that looked and worked like the action bar. The nice thing about Toolbar was that you could put one anywhere you wanted in your UI, making it simpler to add dedicated toolbars, such as for a rich-text editor.

CONFIGURING THE APP BAR

Over time, Toolbar became the stock implementation of this sort of top-of-the-activity bar.

From a terminology standpoint, we have used “toolbar” as a descriptive term for quite some time, as “toolbar” is a widely-used term in desktop apps. A desktop app toolbar served a very similar role to the action bar in Android, and then the Toolbar in Android. So, you might see references to “toolbar buttons” in Android, which really refer to that sort of UI pattern, regardless of whether those buttons appear on an actual Toolbar or not.

App Bar

In 2015, Google debuted [“Material Design”](#). This is Google’s “design language” for mobile, Web, and desktop apps. It provides Google’s recommendations for what things should look and work like.

In Material Design, this bar is called the [app bar](#), because apparently Google likes coming up with new names for this.

The Material Design team has also created [“Material Components for Android”](#), which is a library that implements many of the UI patterns seen in Material Design that go beyond what modern versions of Android support directly. There, they have classes like AppBarLayout and CollapsingToolbarLayout, for trying to implement some of the specific Material Design recommendations regarding app bars, such as how they behave with respect to scrolling content in an activity.

Bars and This Book

From this point forward, the book will tend to use:

- Toolbar, in monospace, when referring to the actual Toolbar class
- “App bar”, when referring to the concept of this bar
- “Action bar”, when referring to that specific role
- “toolbar buttons”, when referring to the icons that can appear in this bar that the user can tap on to perform actions

(the book will use few profanities, no matter how appropriate they may be for aspects of Android app development)

CONFIGURING THE APP BAR

Bars Beyond These Bars

There are other bars in the Android UI that are unrelated to the app bar:

- The status bar is the thin strip across the top containing the time and various icons for device status
- The navigation bar is the strip across the bottom that provides the HOME button and — depending on Android version and situation — buttons for BACK and RECENTS (the latter of which brings up the overview screen)

Vector Drawables

Frequently, the app bar contains toolbar buttons, such as the refresh one shown below:

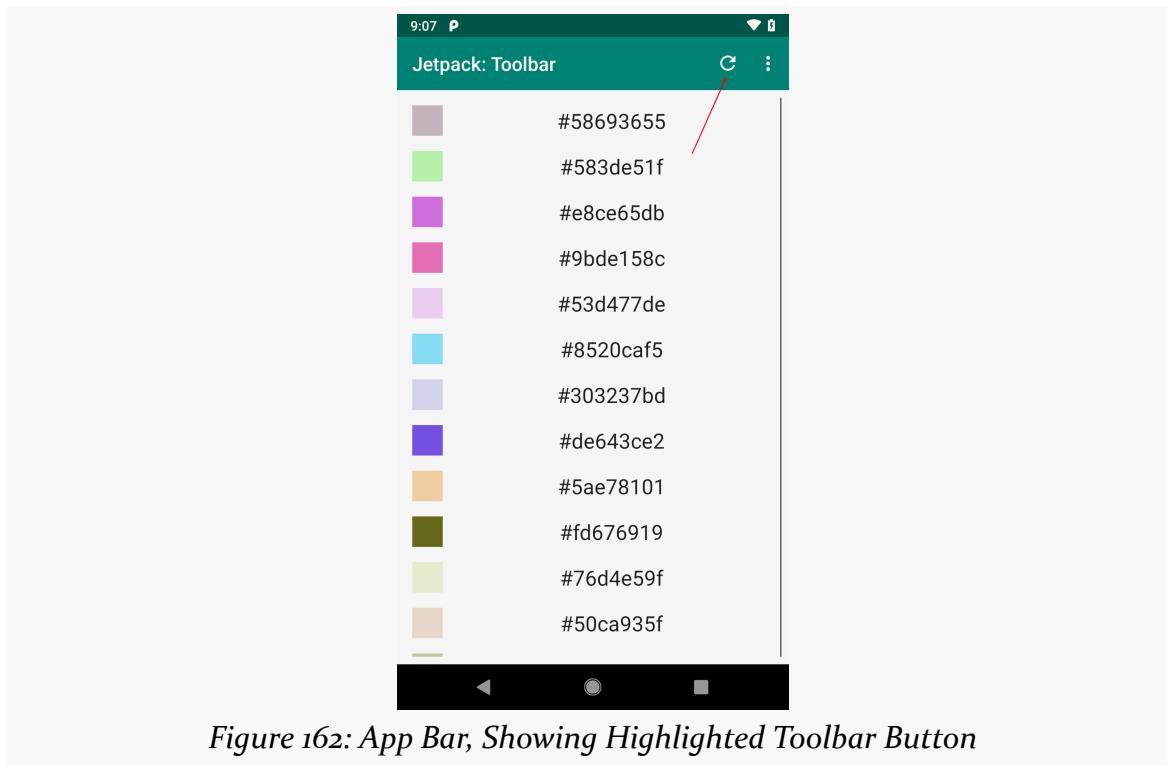


Figure 162: App Bar, Showing Highlighted Toolbar Button

We could use PNG images for these. However, typically, icons like this one are implemented using vector drawables. These are drawn dynamically, so we do not need different versions of vector drawables for different screen densities, the way that we do with PNGs. Plus, Google supplies us with a library of existing vector

CONFIGURING THE APP BAR

artwork that we can use, in addition to importing SVG files from elsewhere, such as from a graphic designer.

Starting the Vector Asset Wizard

To add a vector drawable to your app — either from Google-supplied artwork or SVGs — you can use the Vector Asset Wizard. You can start this by right-clicking over anything in your module's portion of the project tree in Android Studio, then choosing “New” > “Vector Asset” from the context menu.

This will bring up the Vector Asset Wizard on its first page:

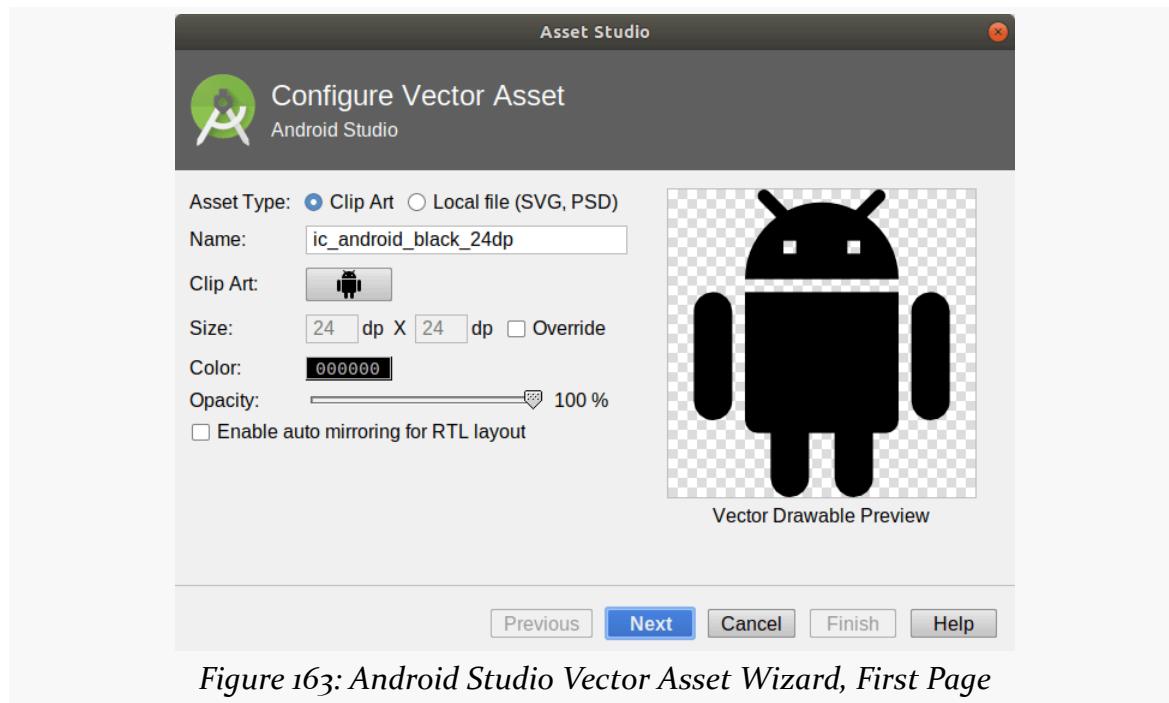


Figure 163: Android Studio Vector Asset Wizard, First Page

CONFIGURING THE APP BAR

Using Built-In Vector Artwork

By default, the wizard starts off with the “Asset Type” radio buttons set to “Clip Art”, allowing you to choose an icon from a Google-supplied library. The default icon is the Android mascot (“Bugdroid”), but you can click the button next to “Clip Art” to bring up the catalog of available artwork:

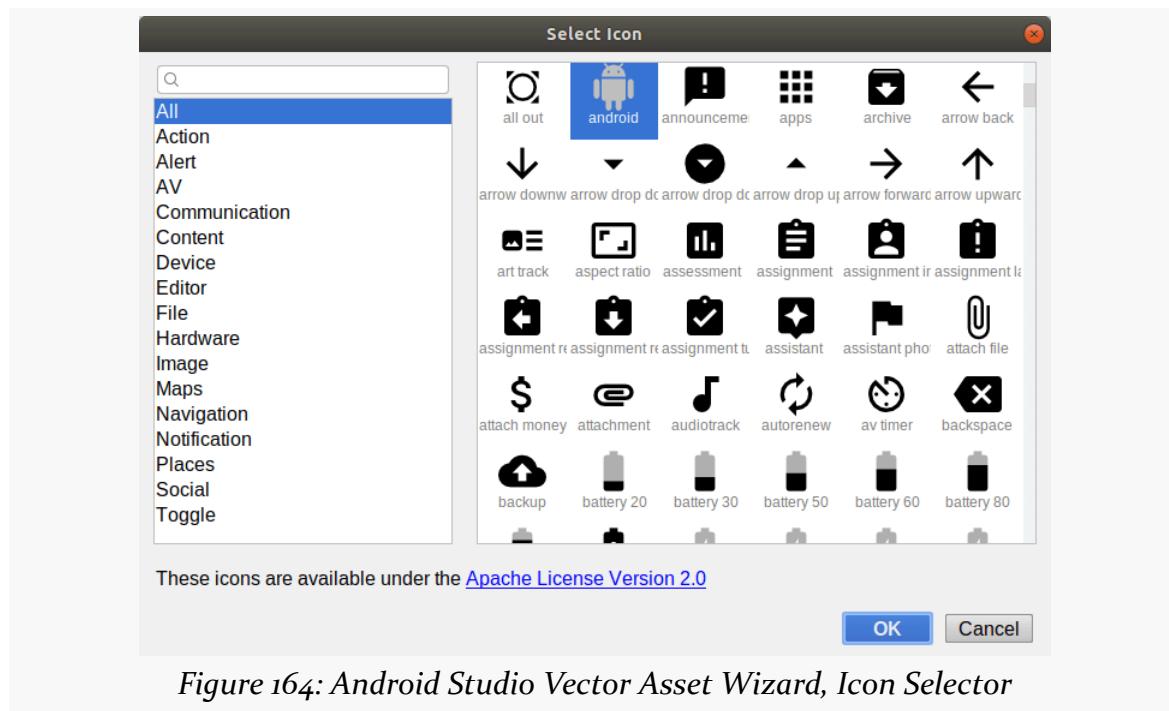


Figure 164: Android Studio Vector Asset Wizard, Icon Selector

Here you can:

- Search via the search field in the upper left;
- Browse all of the icons, by choosing “All” in the list on the left and scrolling through the icons; or
- Browse a category of the icons, by choosing anything other than “All” in the list on the left and scrolling through the chosen category of icons

CONFIGURING THE APP BAR

Once you have identified the icon that you want, you can click OK to return to the wizard, with your icon selected:

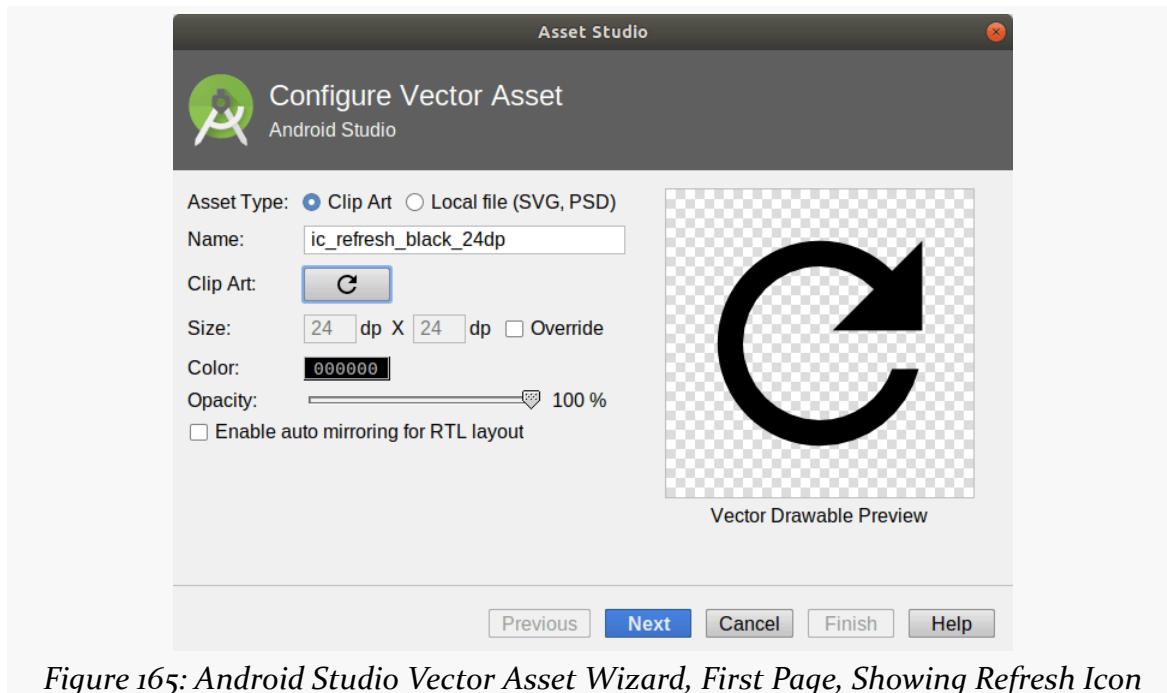


Figure 165: Android Studio Vector Asset Wizard, First Page, Showing Refresh Icon

An icon name will be filled in for you in the “Name” field, though you can change that if you wish. Similarly, you can override:

- The size of the icon (default is 24dp by 24dp)
- The color of the icon (default is black)
- The opacity of the icon, to control the color’s alpha channel (default is opaque)
- Whether the icon should be flipped on RTL screens or should remain unchanged

You can then click Next, followed by “Finish”, to add the vector drawable to the `res/drawable/` directory of your module.

Importing SVGs

If you have a vector image in SVG or PSD format from a graphic designer, you can try to use it. Android’s vector drawables implement a subset of SVG, and some SVGs will be too complex for Android to support.

CONFIGURING THE APP BAR

To use an existing SVG file, rather than browse the existing icons, choose “Local file (SVG, PSD)” in the “Asset Type” radio buttons in the first page of the Vector Asset Wizard:

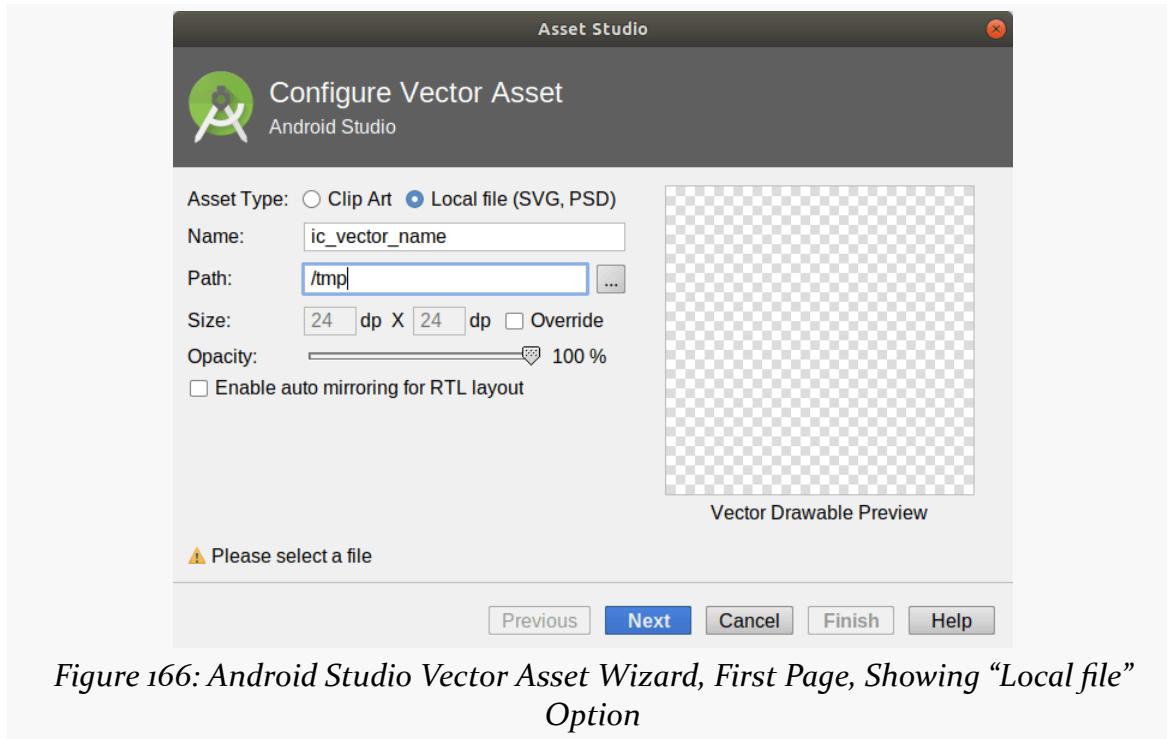


Figure 166: Android Studio Vector Asset Wizard, First Page, Showing “Local file” Option

CONFIGURING THE APP BAR

You can click the “...” button next to the “Path” field to browse for your SVG or PSD file. After choosing the file, the Vector Asset Wizard will load it and give you some idea of whether or not it will work. Specifically, if the preview does not look promising, or you get warnings, that particular file may not work well as a vector drawable:

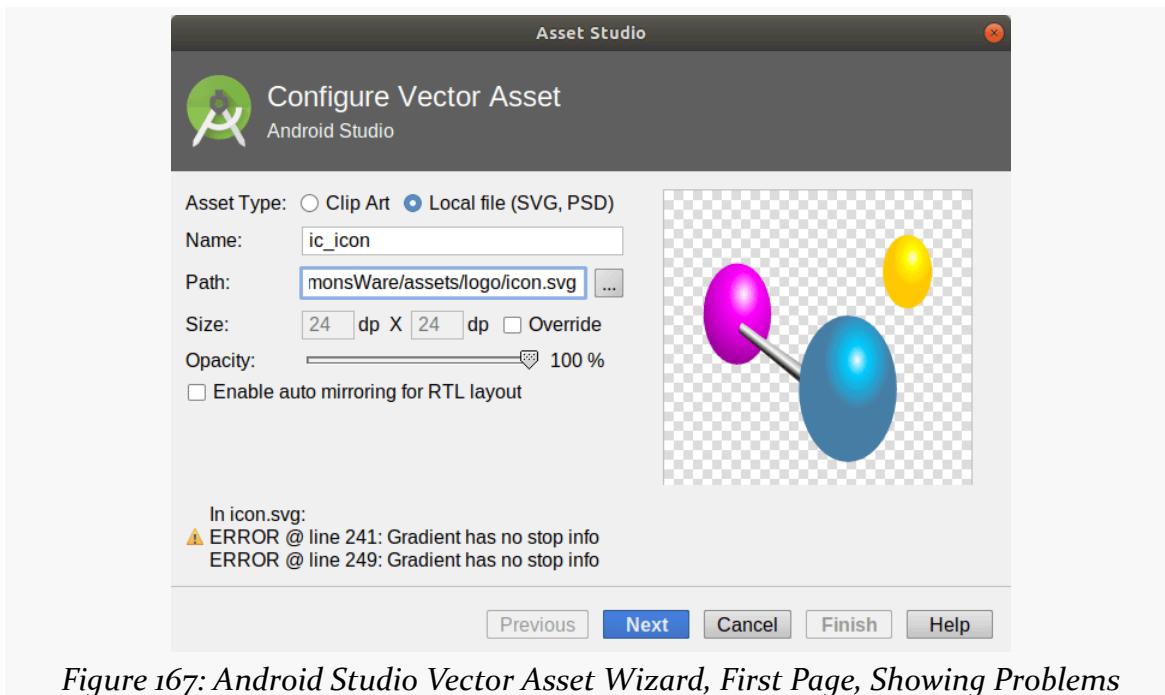


Figure 167: Android Studio Vector Asset Wizard, First Page, Showing Problems

Beyond that, though, you can override the size, opacity, and RTL settings, then proceed through the rest of the wizard to create the vector drawable.

Menu Resources

The next step in configuring the app bar is to set up a menu resource to reflect the clickable items that should appear in the app bar, such as toolbar buttons.

Why “Menu”?

Back in the dawn of Android time, referred to by some as “2007”, we had options menus. These would rise up from the bottom of the screen based on the user pressing a MENU key:



Figure 168: Legacy Options Menu

When Google introduced the action bar pattern with Android 3.0 in 2011, they used the old options menu system and simply changed its UI. So, you will see references to an “options menu” in our work with the action bar, as this is all based on that original 2011 implementation of the action bar.

Similarly, the options menu was populated using a menu resource. The action bar adopted the menu resource and extended it for its own purposes. Today, Toolbar continues to use menu resources, even though we might not think of our work involving a “menu” anymore.

Defining Menu Resources

Menu resources are “first-class” resources, like layouts, drawables, and mipmaps. They get their own dedicated `res/menu/` directory and have their own dedicated

CONFIGURING THE APP BAR

Android Studio editor.

Creating the res/menu/ Directory

However, when you create a brand-new Android Studio project, you may not have a `res/menu/` directory in your module.

Since this is an ordinary directory in an ordinary filesystem, you can create this directory by any means that you like, such as your development machine’s “file manager”, or via a `mkdir` command on the command line.

Inside of Android Studio, you have two main options for creating this directory. The simple one is to right-click over the `res/` directory of your module, then choose “New” > “Directory” from the context menu. This will pop up a dialog where you can provide your directory name — if you fill in `menu` and accept the dialog, it will create the `menu/` subdirectory under `res/` for you.

The more elaborate solution is to right-click over any directory in your module and choose “New” > “Android Resource Directory” from the context menu. This brings up a “New Resource Directory” dialog:

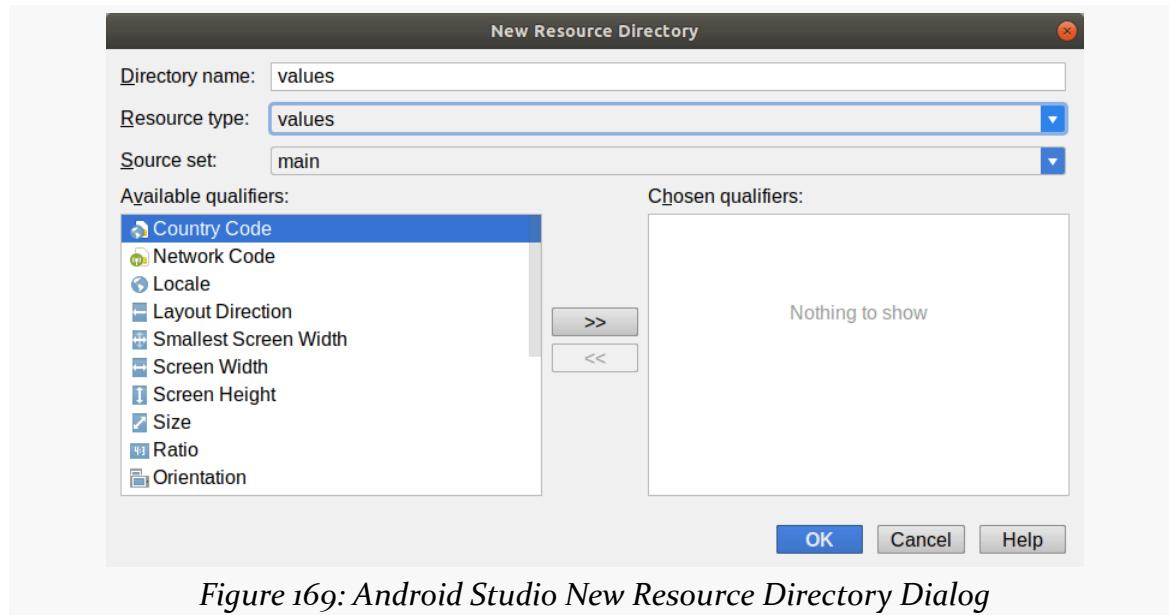


Figure 169: Android Studio New Resource Directory Dialog

There, you can choose “menu” in the “Resource type” drop-down, then click “OK” to create the `res/menu/` directory. This dialog also allows you to create directories for

CONFIGURING THE APP BAR

resource sets via the “Available qualifiers” list, and it has a few other “bells and whistles”, but in the end, it just creates a directory.

Creating the Menu Resource

Once you have the `res/menu/` directory, you can create an empty menu resource.

In Android Studio, that is a matter of:

1. Right-clicking over your `res/menu/` directory
2. Choosing “New” > “Menu resource file” from the context menu, to bring up a simple “New Menu Resource File” dialog
3. Entering the base name of the resource (e.g., `actions`) into the dialog
4. Clicking “OK” to close the dialog and create the nearly-empty resource

You will wind up with a menu resource based on your chosen name (e.g., `actions.xml`) with an empty `<menu>` root element:

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

</menu>
```

CONFIGURING THE APP BAR

Adding Menu Items

The menu resource editor works like the graphical layout editor. It has two sub-tabs, one with the raw XML (“Text”) and one with a drag-and-drop GUI builder (“Design”). The drag-and-drop GUI builder has the same basic structure as does the layout editor, with a palette, a component tree, a preview area, and an attributes pane:

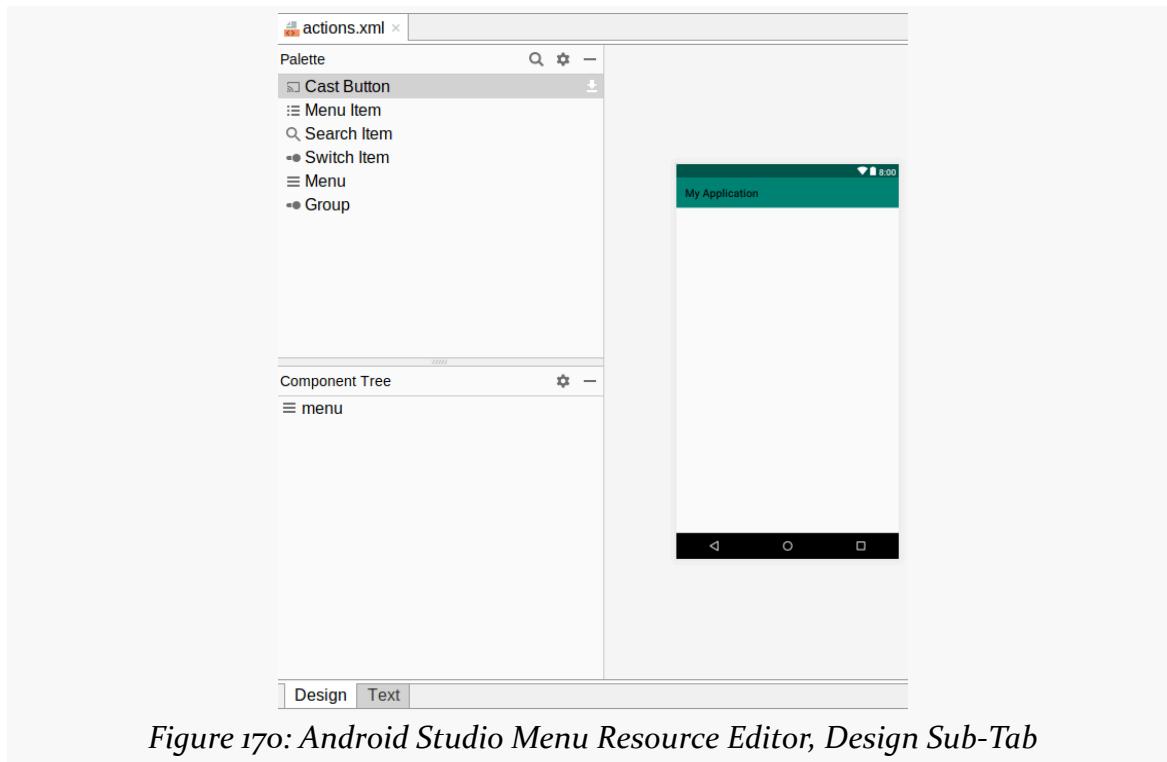


Figure 170: Android Studio Menu Resource Editor, Design Sub-Tab

CONFIGURING THE APP BAR

While there are several items in the “Palette” tool, mostly you will be working with “Menu Item” elements. You can drag them from the “Palette” into the “Component Tree” or preview area to add them to the menu resource:

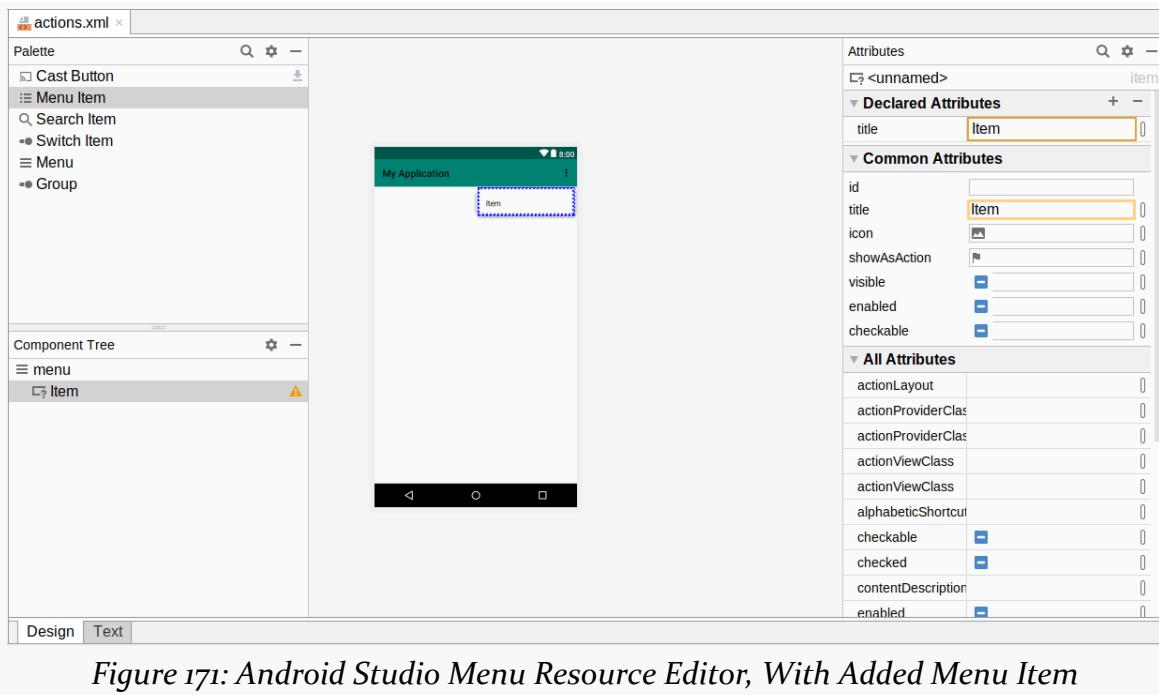


Figure 171: Android Studio Menu Resource Editor, With Added Menu Item

The “Attributes” pane will then allow you to manipulate the menu item’s attributes — we will explore this more [in an upcoming section](#).

Using Toolbar Directly

So, let’s see how we can use vector drawables and menu resources to add interactive elements to the app bar. First, we will look at how to do this using Toolbar directly as a widget.

The Toolbar sample module in the [Sampler](#) and [SamplerJ](#) projects are a variation on the show-a-list-of-random-colors ViewModel sample from earlier in the book. Most of the changes are tied to using Toolbar, but if you examine the full modules, you will see that this sample also blends in the instance state management — using `ViewModelProvider.Factory` — that we saw in some of show-a-list-of-lifecycle-events samples.

CONFIGURING THE APP BAR

Adding the Widget

We can add a Toolbar to our layout, positioning it wherever we want. Typically, it appears at the top of the activity, but that is not a requirement.

So, the Toolbar project edition of the `activity_main` layout has a Toolbar above the RecyclerView, wrapped in the ConstraintLayout:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.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">

    <androidx.appcompat.widget.Toolbar
        android:id="@+id/toolbar"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:background="?attr/colorPrimary"
        app:theme="?attr/actionBarPopupTheme"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <androidx.recyclerview.widget.RecyclerView
        android:id="@+id/items"
        android:layout_width="0dp"
        android:layout_height="0dp"
        android:padding="@dimen/content_padding"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/toolbar" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

(from [Toolbar/src/main/res/layout/activity_main.xml](#))

Usually, the app bar is set to be flush with the status bar and the sides of the screen. So, whereas the ViewModel sample had 8dp of padding in the ConstraintLayout, this sample moves that padding to the RecyclerView, so the Toolbar is not inset from the edges.

CONFIGURING THE APP BAR

The Toolbar has attributes that mostly set up the size and position. A simple Toolbar can use `wrap_content` for the height, and it will be sized appropriately for toolbar buttons and such. `android:background` allows you to specify the background color, and here we defer to the theme and use whatever we have set there as `colorPrimary` (`?attr/colorPrimary`). We will explore the `app:theme` attribute more in the next section.

Tailoring the Theme

The theme used by prior sample projects in this book was based on `Theme.AppCompat.Light.DarkActionBar`. As the name suggests, this adds an action bar to the top of the activity, with an eye towards it having a dark color, so text will be shown in white for a good contrast.

The problem is that we do not want an action bar. We have our own Toolbar that we want to show instead.

The theme has no way of knowing this, and so by default, we would wind up with two app bars, stacked on top of each other:

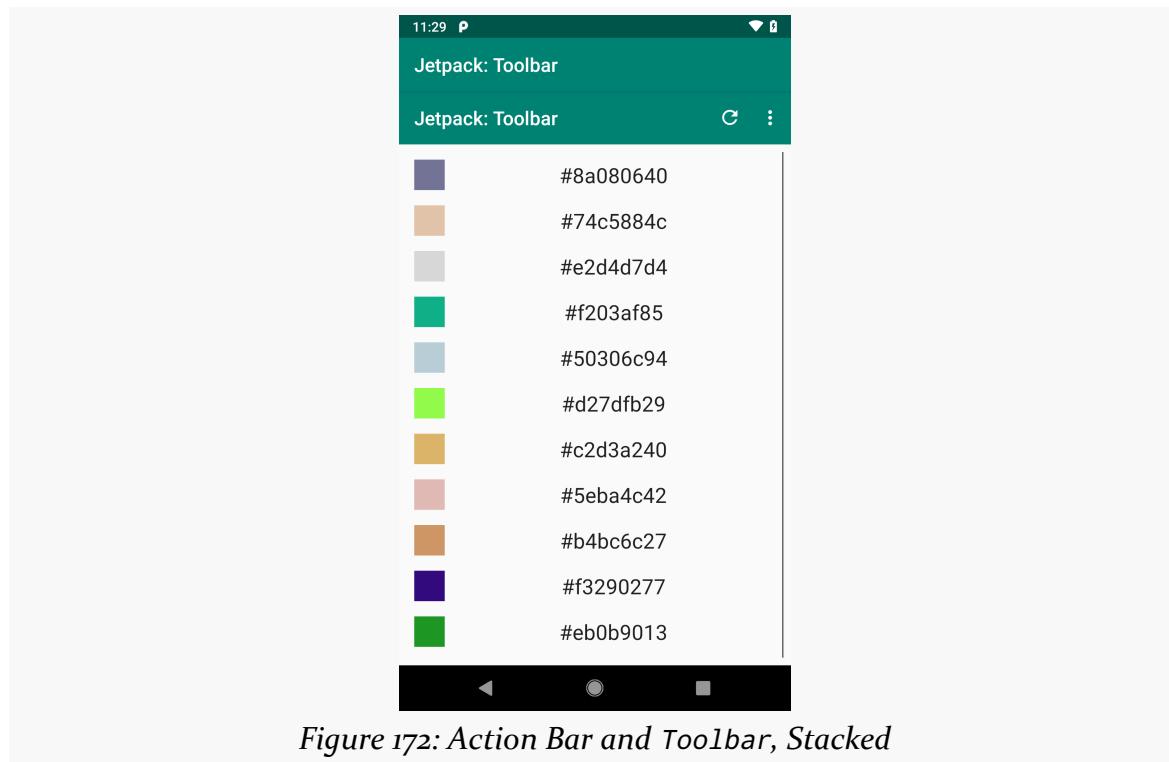


Figure 172: Action Bar and Toolbar, Stacked

CONFIGURING THE APP BAR

This is... not good.

There is another base theme, `Theme.AppCompat.Light.NoActionBar`, which skips the theme-supplied action bar, leaving it up to us. So, the Toolbar sample modules use that base theme instead:

```
<resources>

    <!-- Base application theme. -->
    <style name="AppTheme" parent="Theme.AppCompat.Light.NoActionBar">
        <!-- Customize your theme here. -->
        <item name="colorPrimary">@color/colorPrimary</item>
        <item name="colorPrimaryDark">@color/colorPrimaryDark</item>
        <item name="colorAccent">@color/colorAccent</item>
        <item name=" actionBarPopupTheme ">@style/PopupOverlay</item>
    </style>

    <style name="PopupOverlay" parent="ThemeOverlay.AppCompat.Dark.ActionBar">
        <item name="iconTint">@android:color/white</item>
    </style>

</resources>
```

(from [Toolbar/src/main/res/values/styles.xml](#))

The next problem is that the stock look of a Toolbar assumes that the background color will be light, and so we want dark text and icons on it. In our case, the background color is relatively dark, and so we would prefer light text and icons. This is a bit tricky to set up. One recipe is:

- Define a style resource that inherits from `ThemeOverlay.AppCompat.Dark.ActionBar` (here called `PopupOverlay`)
- In that style resource, define the `iconTint` to be whatever color you want — in this case, we are using `@android:color/white` to pull in a framework-defined white color
- In the theme for the activity (`AppTheme` in our case), define an `actionBarPopupTheme` attribute and have it point to the style resource that you just created
- Give the Toolbar an `app:theme` attribute that delegates to the theme's `actionBarPopupTheme` (`app:theme="?attr/actionBarPopupTheme"`)

`ThemeOverlay.AppCompat.Dark.ActionBar` will give us light text, and white for the `iconTint` will give us light icons.

CONFIGURING THE APP BAR

Defining the Menu Resource

The sample app has a menu resource, `res/menu/actions.xml`, that contain interactive elements that will go into the Toolbar:

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

    <item
        android:id="@+id/refresh"
        android:title="@string/menu_refresh"
        android:icon="@drawable/ic_refresh_black_24dp"
        app:showAsAction="ifRoom" />
    <item
        android:id="@+id/about"
        android:title="@string/menu_about"
        app:showAsAction="never" />
</menu>
```

(from [Toolbar/src/main/res/menu/actions.xml](#))

CONFIGURING THE APP BAR

In the graphical menu editor, we see two children — refresh and about — of the root menu in the “Component Tree”:

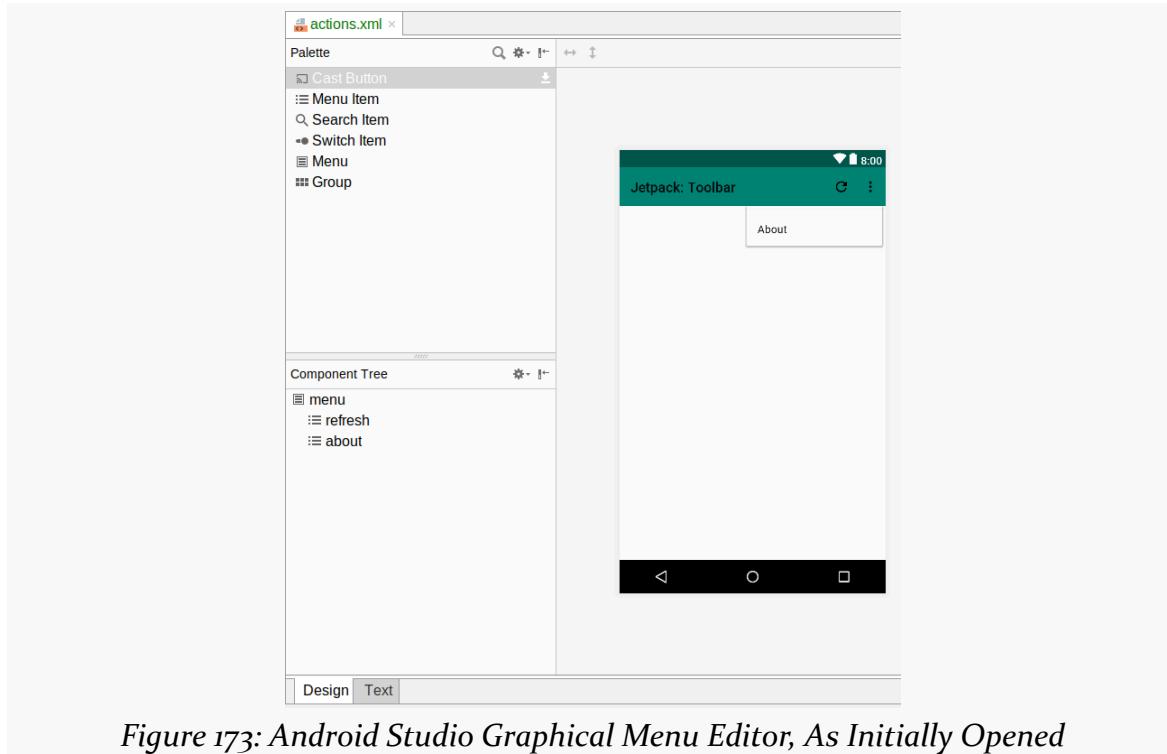


Figure 173: Android Studio Graphical Menu Editor, As Initially Opened

The preview area shows these items as well, though they appear differently due to the way that each of those items is configured, as we will see.

Hey, Why Is the Preview Showing Dark Text and Icons?

In the above screenshot, the title (“Jetpack: Toolbar”) and the icons in the Toolbar show up dark. The graphical menu editor attempts to use our app’s theme, but it has limits, and in this case it is not picking up some of the changes that [we made to the theme](#).

In general, consider the Android Studio preview options to be approximations of what you will see when you run the app on an emulator or device.

Refresh

One of our two menu items is called `refresh`, and we will use it to allow the user to

CONFIGURING THE APP BAR

come up with a new random list of colors:

```
<item  
    android:id="@+id/refresh"  
    android:title="@string/menu_refresh"  
    android:icon="@drawable/ic_refresh_black_24dp"  
    app:showAsAction="ifRoom" />
```

(from [Toolbar/src/main/res/menu/actions.xml](#))

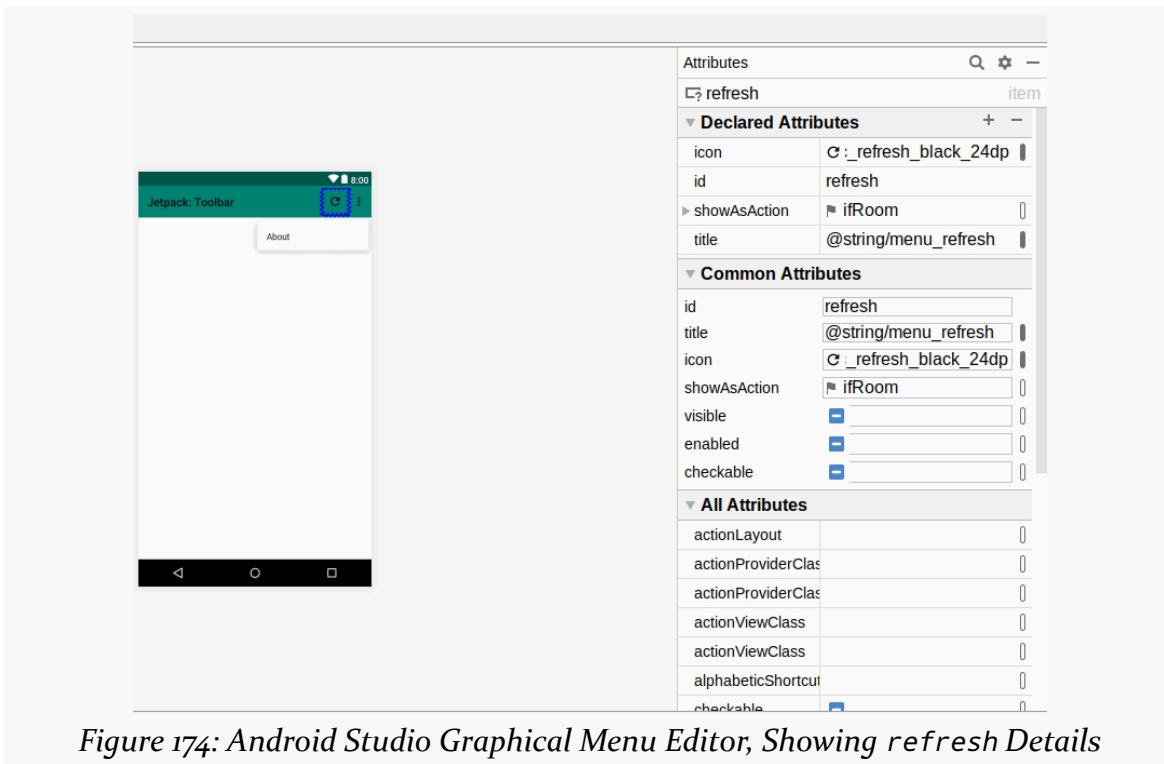


Figure 174: Android Studio Graphical Menu Editor, Showing refresh Details

Each interactive element in our Toolbar will have an `<item>` element inside of the root `<menu>` element of our menu resource. Items always have:

- An ID, using the same `android:id` system that we use for widget IDs
- A title, via the `android:title` attribute, usually pointing to a string resource

Items *usually* have a `showAsAction` attribute. For AppCompat-based activities, that should be `app:showAsAction` — if you see code with `android:showAsAction`, that is a menu resource for use with the native framework action bar or Toolbar, not the AppCompat-compatible Toolbar that we use from AndroidX. `showAsAction` has three major options:

CONFIGURING THE APP BAR

- always, to say that we *really* want this item to be shown as a toolbar button
- ifRoom, to say that we would prefer it be shown as a toolbar button, but that is not essential
- never, to say that this item is unimportant and does not need a dedicated button in the Toolbar

Items that go into the Toolbar that do not wind up with toolbar buttons go into “the overflow menu”. If you have used Android apps that have a vertical ellipsis (“...”) icon in their app bars, where a menu pops open when you tap it... that’s the overflow menu. In our case, the refresh item has ifRoom, so if there is space for it to have a toolbar button, it will have one, otherwise it will go into the overflow.

For items that might be toolbar buttons, usually you will want to provide an android:icon attribute with a pointer to a drawable resource. Here, we are using a refresh icon from the Vector Asset Wizard, using a stock refresh vector asset that is supplied by Google.

About

Our about menu item is not nearly as important, so we set it to have never for showAsAction and skip the android:icon attribute:

```
<item  
    android:id="@+id/about"  
    android:title="@string/menu_about"  
    app:showAsAction="never" />
```

CONFIGURING THE APP BAR

(from [Toolbar/src/main/res/menu/actions.xml](#))

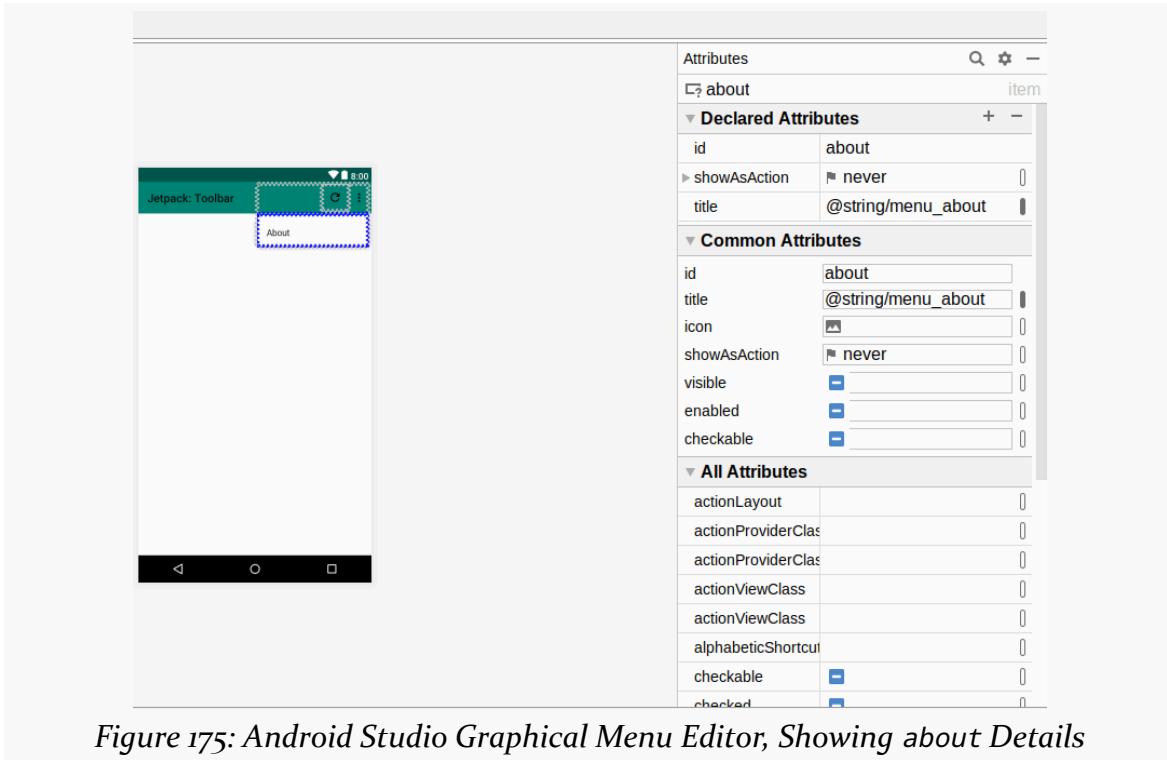


Figure 175: Android Studio Graphical Menu Editor, Showing about Details

Populating the Toolbar

We have a Toolbar in our layout. We have a menu resource. We now need to tie the two together. That requires a bit of code in `onCreate()` of our `MainActivity`, whether that is in Java:

```
Toolbar tb = findViewById(R.id.toolbar);

tb.setTitle(R.string.app_name);
tb.inflateMenu(R.menu.actions);
```

(from [Toolbar/src/main/java/com/commonsware/jetpack/samplerj/toolbar/MainActivity.java](#))

...or Kotlin:

```
toolbar.apply {
    setTitle(R.string.app_name)
    inflateMenu(R.menu.actions)
```

(from [Toolbar/src/main/java/com/commonsware/jetpack/samplerj/toolbar/MainActivity.kt](#))

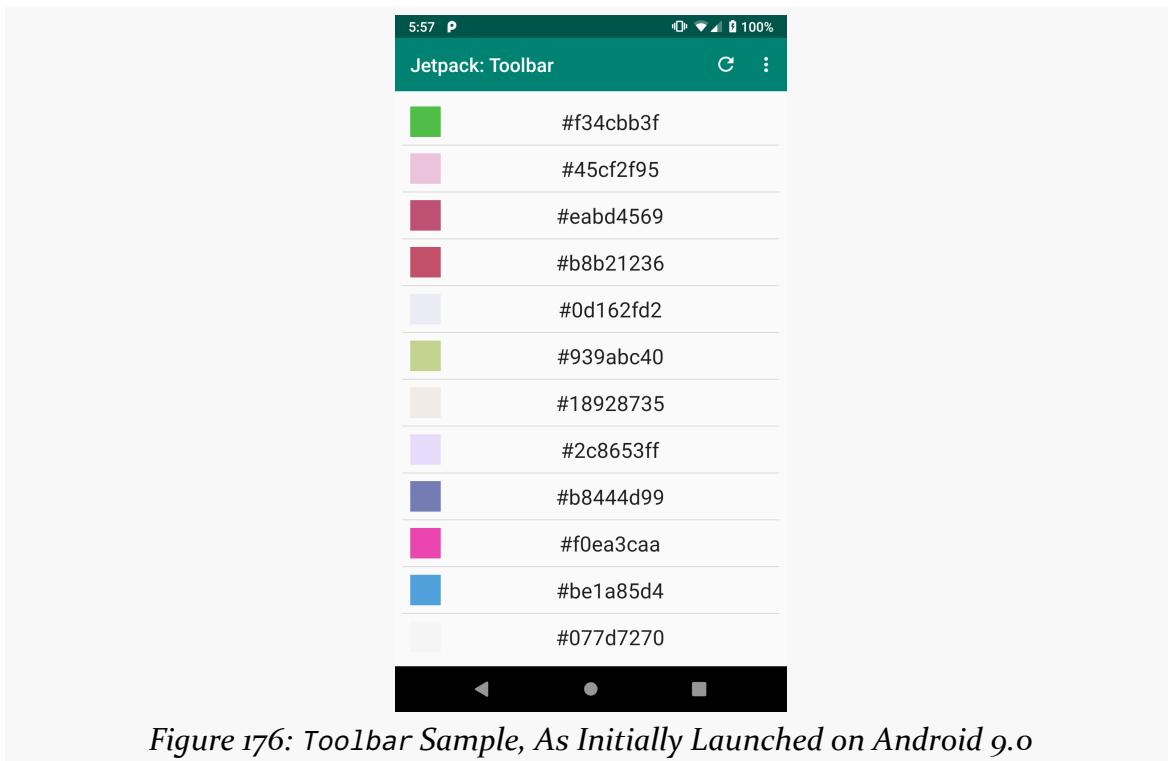
CONFIGURING THE APP BAR

(the Kotlin will appear to be missing a closing brace, but that is because there is more code in this `apply()` that we will see in the next section)

First, we get access to the Toolbar, either via `findViewById()` in Java or by using synthetic accessors in Kotlin. Then, we call two functions on the Toolbar:

- `setTitle()` sets the title text — when we use a Toolbar manually like this, we need to provide that value ourselves
- `inflateMenu()`, where we pass it an ID of our menu resource (`R.menu.actions`)

Under the covers, `inflateMenu()` uses a `MenuItemInflater` to convert our menu resource XML into `Menu` and `MenuItem` objects, which Toolbar will use to configure the Toolbar contents and show our toolbar button and overflow menu:



CONFIGURING THE APP BAR

The overflow menu appears when you tap the “...” button in the Toolbar:

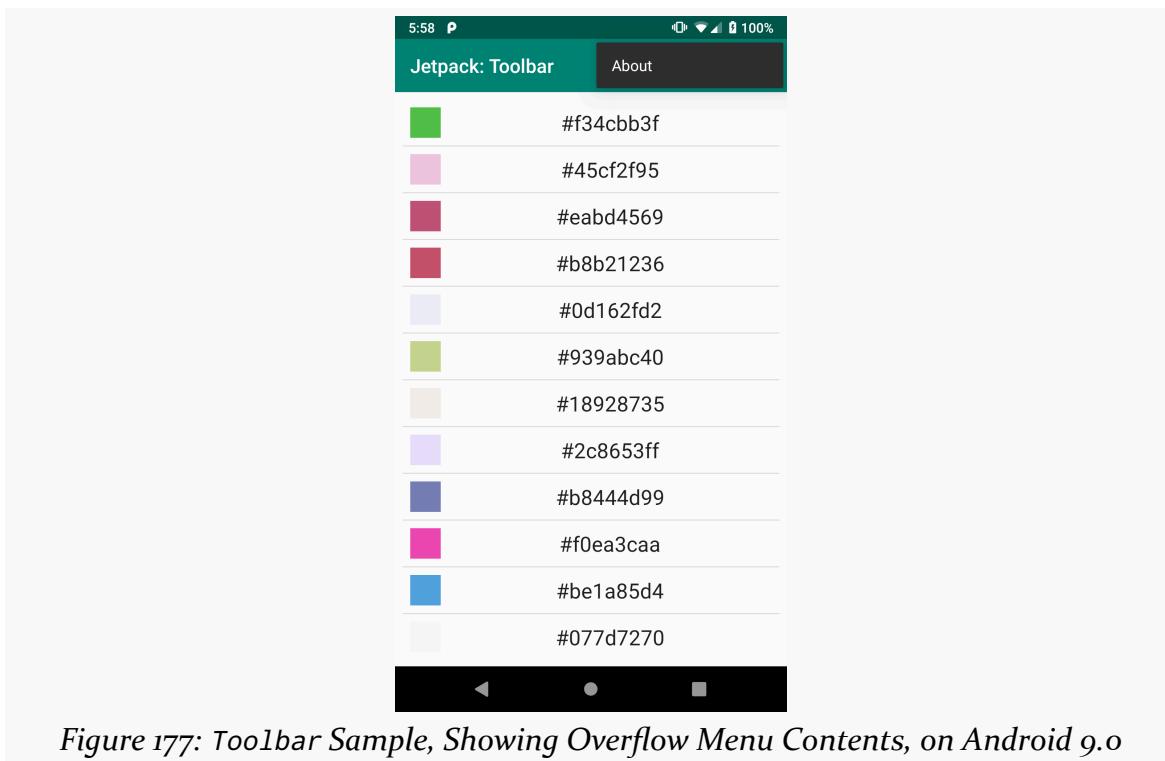


Figure 177: Toolbar Sample, Showing Overflow Menu Contents, on Android 9.0

Responding to Events

While our toolbar button and overflow menu are pretty, they are useless unless we arrange to find out when the user clicks on our refresh and about items.

For that, we call `setOnMenuItemClickListener()` on the Toolbar:

```
tb.setOnMenuItemClickListener(item -> {
    if (item.getItemId() == R.id.refresh) {
        vm.refresh();
        adapter.submitList(vm.numbers);
        return true;
    }
    else if (item.getItemId() == R.id.about) {
        Toast.makeText(MainActivity.this, R.string.msg_toast,
            Toast.LENGTH_LONG).show();
        return true;
    }
    else {
        return false;
    }
})
```

CONFIGURING THE APP BAR

```
    }  
});
```

(from [Toolbar/src/main/java/com/commonsware/jetpack/samplerj/toolbar/MainActivity.java](#))

```
toolbar.apply {  
    setTitle(R.string.app_name)  
    inflateMenu(R.menu.actions)  
  
    setOnMenuItemClickListener { item ->  
        when (item.itemId) {  
            R.id.refresh -> {  
                vm.refresh()  
                colorAdapter.submitList(vm.numbers)  
                true  
            }  
            R.id.about -> {  
                Toast.makeText(  
                    this@MainActivity,  
                    R.string.msg_toast,  
                    Toast.LENGTH_LONG  
                ).show()  
                true  
            }  
            else -> false  
        }  
    }  
}
```

(from [Toolbar/src/main/java/com/commonsware/jetpack/samplerj/toolbar/MainActivity.kt](#))

Our lambda expression is passed the `MenuItem` object corresponding to the `<item>` from our menu resource that the user tapped on. We can call `getitemId()` to determine which `<item>` it was.

If they clicked `refresh`, we call a `refresh()` function on the revised `ColorViewModel` that generates a fresh set of colors:

```
package com.commonsware.jetpack.samplerj.toolbar;  
  
import android.os.Bundle;  
import java.util.ArrayList;  
import java.util.Random;  
import androidx.lifecycle.ViewModel;  
  
public class ColorViewModel extends ViewModel {  
    private static final String STATE_NUMBERS = "numbers";
```

CONFIGURING THE APP BAR

```
ArrayList<Integer> numbers;

ColorViewModel(Bundle state) {
    if (state == null) {
        numbers = buildItems();
    }
    else {
        numbers = state.getIntegerArrayList(STATE_NUMBERS);
    }
}

void onSaveInstanceState(Bundle state) {
    state.putIntegerArrayList(STATE_NUMBERS, numbers);
}

void refresh() {
    numbers = buildItems();
}

private ArrayList<Integer> buildItems() {
    Random random = new Random();

    ArrayList<Integer> result = new ArrayList<>(25);

    for (int i = 0; i < 25; i++) {
        result.add(random.nextInt());
    }

    return result;
}
}
```

(from [Toolbar/src/main/java/com/commonsware/jetpack/samplerj/toolbar/ColorViewModel.java](#))

```
package com.commonsware.jetpack.sampler.toolbar

import android.os.Bundle
import androidx.lifecycle.ViewModel
import java.util.*

private const val STATE_NUMBERS = "numbers"

class ColorViewModel(state: Bundle?) : ViewModel() {
    private val random = Random()
    var numbers = state?.getIntegerArrayList(STATE_NUMBERS) ?: buildItems()

    fun onSaveInstanceState(state: Bundle) {
        state.putIntegerArrayList(STATE_NUMBERS, ArrayList(numbers))
    }
}
```

CONFIGURING THE APP BAR

```
}

fun refresh() {
    numbers = buildItems()
}

private fun buildItems() = List(25) { random.nextInt() }
}
```

(from [Toolbar/src/main/java/com/commonsware/jetpack/sampler/toolbar/ColorViewModel.kt](#))

We then update the `ColorAdapter` with the new colors, and it updates the list.

If the user clicked `About`, we show a `Toast` — in a real app, an “About” item would bring up some screen that contains a copyright notice, license terms, version information, and so on.

The lambda expression needs to return `true` if the item click was handled by the lambda, or `false` if the item was not recognized for one reason or another.

Using Toolbar as the Action Bar

There is also an option for us to take the `Toolbar` from our layout and tell `AppCompatActivity` to use it as the action bar. This does not add much value, but there may be need for it for some use cases.

The `ActionBar` modules in the [Sampler](#) and [SamplerJ](#) projects are almost the same as the `Toolbar` modules. We use the same layout, the same menu resource, and the same theme. The difference lies in the Java/Kotlin code for setting up the `Toolbar` and responding to events.

Registering the Toolbar

To indicate that a `Toolbar` should serve in the role of the action bar, call `setSupportActionBar()` on your `AppCompatActivity`, supplying the `Toolbar`:

```
setSupportActionBar(findViewById(R.id.toolbar));
```

(from [ActionBar/src/main/java/com/commonsware/jetpack/samplerj/actionbar/MainActivity.java](#))

```
setSupportActionBar(toolbar)
```

(from [ActionBar/src/main/java/com/commonsware/jetpack/sampler/actionbar/MainActivity.kt](#))

CONFIGURING THE APP BAR

The action bar automatically gets the title from its activity's `android:label` manifest attribute, so you may not need to call `setTitle()` yourself on the Toolbar.

Populating the Action Bar

Rather than call `inflateMenu()` on the Toolbar, the action bar reuses the old `onCreateOptionsMenu()` callback function from the days of options menus. Your `AppCompatActivity` subclass — such as `MainActivity` in the `ActionBar` samples — will need to override this:

```
@Override  
public boolean onCreateOptionsMenu(Menu menu) {  
    getMenuInflater().inflate(R.menu.actions, menu);  
  
    return super.onCreateOptionsMenu(menu);  
}
```

(from [ActionBar/src/main/java/com/commonsware/jetpack/samplerj/actionbar/MainActivity.java](#))

```
override fun onCreateOptionsMenu(menu: Menu): Boolean {  
    menuInflater.inflate(R.menu.actions, menu)  
  
    return super.onCreateOptionsMenu(menu)  
}
```

(from [ActionBar/src/main/java/com/commonsware/jetpack/samplerj/actionbar/MainActivity.kt](#))

In `onCreateOptionsMenu()`, you:

- Get a `MenuItem` by calling `getMenuInflater()` on the activity
- Call `inflate()` on the `MenuItem`, passing in your menu resource ID (`R.menu.actions`) and the `Menu` object supplied to `onCreateOptionsMenu()`
- Chain to the superclass' implementation of `onCreateOptionsMenu()` and return its results

Responding to Events

Similarly, instead of calling `setOnMenuItemClickListener()` on the Toolbar to find out when the user clicks on items, you override the legacy `onOptionsItemSelected()` function:

```
@Override  
public boolean onOptionsItemSelected(MenuItem item) {  
    if (item.getItemId() == R.id.refresh) {
```

CONFIGURING THE APP BAR

```
        vm.refresh();
        adapter.submitList(vm.numbers);
        return true;
    }
    else if (item.getItemId() == R.id.about) {
        Toast.makeText(MainActivity.this, R.string.msg_toast,
            Toast.LENGTH_LONG).show();
        return true;
    }
    else {
        return super.onOptionsItemSelected(item);
    }
}
```

(from [ActionBar/src/main/java/com/commonsware/jetpack/samplerj/actionbar/MainActivity.java](#))

```
override fun onOptionsItemSelected(item: MenuItem): Boolean {
    return when(item.itemId) {
        R.id.refresh -> {
            vm.refresh()
            colorAdapter.submitList(vm.numbers)
            true
        }
        R.id.about -> {
            Toast.makeText(
                this@MainActivity,
                R.string.msg_toast,
                Toast.LENGTH_LONG
            ).show()
            true
        }
        else -> super.onOptionsItemSelected(item)
    }
}
```

(from [ActionBar/src/main/java/com/commonsware/jetpack/sampler/ActionBar/MainActivity.kt](#))

This function does the same basic thing as the lambda expression supplied to `setOnMenuItemClickListener()`:

- Get the item ID from the `MenuItem`
- Compare it to expected values (e.g., a `when` in Kotlin)
- Handle the events for those expected values (e.g., showing a `Toast`)
- Return `true` if the event was handled

In the unrecognized-item scenario, since we chained to the superclass in `onCreateOptionsMenu()`, we chain to the superclass in `onOptionsItemSelected()`.

CONFIGURING THE APP BAR

That way, if our superclass is contributing items to the action bar, we will both add those items (`onCreateOptionsMenu()`) and handle their click events (`onOptionsItemSelected()`).

Visually, this sample is indistinguishable from the Toolbar one, other than using a different `app_name` string resource. Functionally, it is merely a matter of where and how you apply the menu resource and deal with the results.

Having Fun at Bars

There are many other things that you can do with an app bar, just with the Android SDK, such as:

- Have checkable menu items, particularly for the overflow, that the user can check and uncheck
- Add other sorts of widgets to it, such as search fields and custom menus
- Have a “contextual action bar” that overlays the regular app bar with options tied to the UI state, such as providing actions for manipulating the current selected item(s) in a list
- Hide and show the app bar, so it does not take up screen space all the time (e.g., in a video player)

Other libraries can offer other features. For example, Google publishes the [Material Components for Android](#) library, which offers extensions like:

- Bottom app bars (i.e., an app bar that appears on the bottom, rather than the top)
- Collapsing toolbars (i.e., ones that appear super-sized at the outset, then collapse as the user scrolls through content)

Implementing Multiple Activities

All of the apps that we have seen so far in this book have had a single screen's worth of UI. Many apps are somewhat more complicated than that, where we need to have lots of screens, such as:

- Showing a list of stuff
- Showing the details of a particular item out of that list
- Editing the details of a particular item, or adding a new one
- Settings to configure how the app behaves
- And so on

In the world of Jetpack, there are two main approaches for adding multiple screens. In this chapter, we will look at one: having more than one activity. In [the next chapter](#), we will explore the other option: using fragments.

Multiple Activities, and Your App

There are at least two scenarios where your app will have to deal with multiple activities.

The first is the one outlined above: you want to have different screens for handling different bits of app functionality. In that case, you are the one writing the additional activities, and you will be the one to choose when those activities get displayed (e.g., bring up the details when the user taps on an item in the list).

The second is when you have some piece of data that you want some other app to process. For example, you might have a URL to a Web page that you obtained from somewhere, and you want the user to view that Web page in their browser. In this case, you are not writing the Web browser (probably), but you are still deciding

when that Web browser activity gets displayed.

A less-common scenario is the inverse of the previous one. Suppose you *are* writing a Web browser. You want other apps to be able to hand you URLs, so you can display those Web pages to the user. In this case, you are writing the activity, but you may not be the one deciding when that activity gets displayed.

This chapter will focus on the first two of those scenarios, as they are the most common, though we will briefly cover how you support the third scenario.

Creating Your Second (and Third and...) Activity

Unfortunately, activities do not create themselves. On the positive side, this does help keep Android developers gainfully employed.

Given a module with one activity, if you want a second activity, you will need to add it yourself. The same holds true for the third activity, the fourth activity, and so on.

Defining the Class and Resources

To create your second (or third or whatever) activity, you first need to create the Java or Kotlin class. You need to create a new source file, containing a public class that extends `Activity` (or `AppCompatActivity`, etc.). You have two basic ways of doing this:

- Just create the class yourself
- Use the Android Studio new-activity wizard

To use the Android Studio new-activity wizard, right-click a package (e.g., `com.commonsware.jetpack.sampler.activities`) in the project tree, and go into the “New” > “Activity” portion of the context menu. This will give you a submenu of available activity templates.

IMPLEMENTING MULTIPLE ACTIVITIES

If you choose one of those templates, you will be presented with a one-page wizard in which to provide the details for this activity:

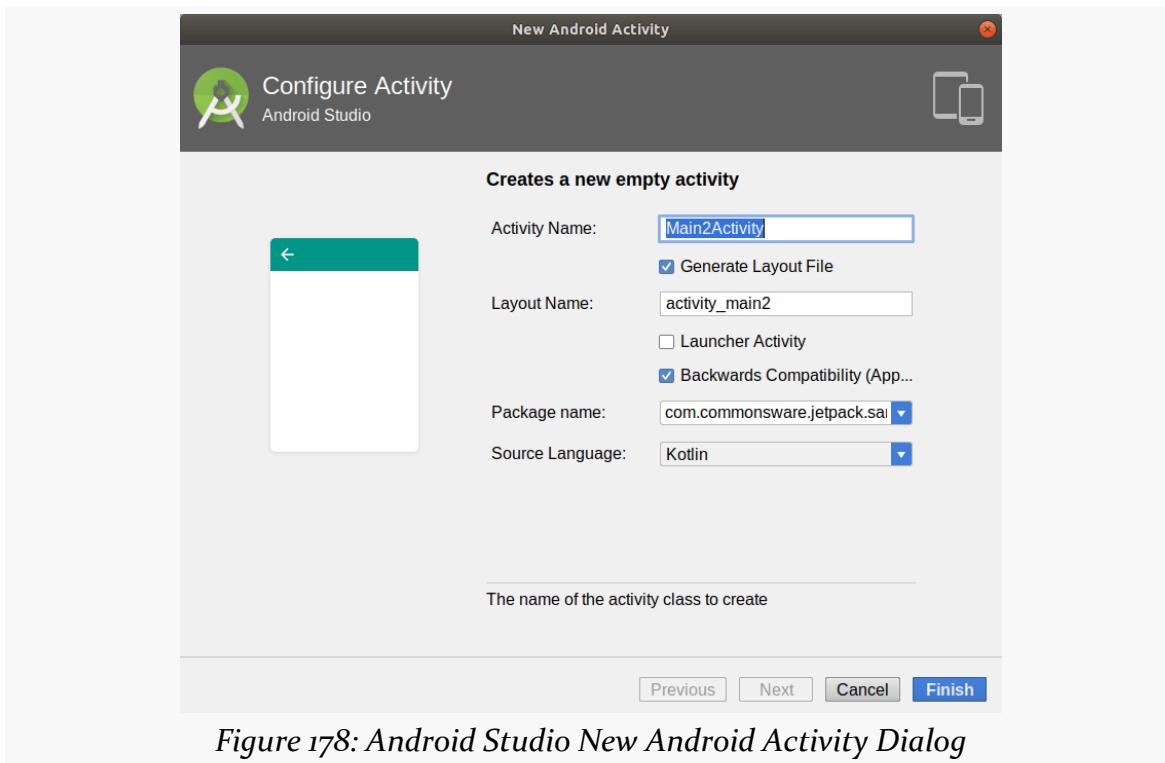


Figure 178: Android Studio New Android Activity Dialog

What you see here will be based upon the template you chose. This happens to be the wizard screen for the “Empty Activity” template; other templates will have forms with other data to collect.

Clicking “Finish” will then create the activity’s Java or Kotlin class, related resources (if any), and manifest entry.

Populating the Class and Resources

Once you have your stub activity set up, you can then add an `onCreate()` function to it (or edit an existing one created by the wizard), filling in all the details (e.g., `setContentView()`), just like you did with your first activity. Your new activity may need a new layout XML resource or other resources, which you would also have to create (or edit those created for you by the wizard).

Augmenting the Manifest

Simply having an activity implementation is not enough. We also need to add it to our `AndroidManifest.xml` file. If you used the new-activity wizard, this entry will be added for you. However, if you created the activity “by hand”, you will need to add its manifest element, and over time you will need to edit this element in many cases.

Adding an activity to the manifest is a matter of adding another `<activity>` element to the `<application>` element:

```
<activity android:name=".BigSwatchActivity"></activity>
```

(from [TwoActivities/src/main/AndroidManifest.xml](#))

You need the `android:name` attribute at minimum, identifying the Java/Kotlin class that is the implementation of the activity.

As we have seen previously, `<activity>` elements can be more complex:

```
<activity android:name=".MainActivity">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />

        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

(from [TwoActivities/src/main/AndroidManifest.xml](#))

We will explore that `<intent-filter>` a bit more [later in this chapter](#).

Starting Your Own Activity

To start an activity, we call `startActivity()` on some Context, typically on our Activity or a Context obtained from a View.

To identify what activity to start, we pass an Intent object to `startActivity()`. When starting an activity from your own project, the particular type of Intent that we will use is an “explicit” Intent. This is where we identify the specific class that implements the activity that we want to start.

The explicit form of the Intent constructor takes two parameters:

IMPLEMENTING MULTIPLE ACTIVITIES

- A Context, typically the activity that is asking to start another activity
- A Java Class object

That second parameter type means that in Kotlin, you will use `::class.java`, rather than just `::class`, to get to the proper object.

Extra! Extra!

Sometimes, we may wish to pass some data from one activity to the next. For example, we might have a RecyclerView in one activity showing a collection of stuff, and we might have a separate activity to show details of one of those items in the collection. We want to start the detail activity when the user clicks on an item in the RecyclerView. However, somehow, the detail activity needs to know *which* item is the one for which it is to show the details. Unless we tell it which one the user clicked, the detail activity has no way to know.

One way to accomplish this is via Intent extras.

There is a series of `putExtra()` methods on Intent to allow you to supply key/value pairs of data to be bundled into the Intent. The keys are strings. While you cannot use arbitrary objects for the values, most primitive data types are supported, as are strings and some types of lists. Also, anything implementing Parcelable can go in an Intent extra.

Any activity can call `getIntent()` to retrieve the Intent used to start it up, and then can call various forms of `get... Extra()` (with the ... indicating a data type) to retrieve any bundled extras.

Seeing This In Action

As the name suggests, the TwoActivities sample module in the [Sampler](#) and [SamplerJ](#) projects has two activities. The MainActivity is very similar to the Toolbar one from [the previous chapter](#). However, now when the user clicks on a color, we want to launch a second activity that shows that color in a larger form.

The Second Activity

The second activity is named BigSwatchActivity, and it was created using the “Empty Activity” template in the Android Studio new-activity wizard. As a result, the

IMPLEMENTING MULTIPLE ACTIVITIES

wizard added a manifest entry for our activity... the same one shown previously in this chapter:

```
<activity android:name=".BigSwatchActivity"></activity>
```

(from [TwoActivities/src/main/AndroidManifest.xml](#))

The new-activity wizard created an `activity_big_swatch` layout resource, which we modified to have a Toolbar at the top and a View taking up all the remaining space:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.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=".BigSwatchActivity">

    <androidx.appcompat.widget.Toolbar
        android:id="@+id/toolbar"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:background="?attr/colorPrimary"
        android:theme="?attr/actionBarPopupTheme"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar" />

    <View
        android:id="@+id/swatch"
        android:layout_width="0dp"
        android:layout_height="0dp"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/toolbar" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

(from [TwoActivities/src/main/res/layout/activity_big_swatch.xml](#))

And, the new-activity wizard created the `BigSwatchActivity` class itself, which we then augmented with actual app logic. That class could be written in Java:

```
package com.commonsware.jetpack.samplerj.activities;
```

IMPLEMENTING MULTIPLE ACTIVITIES

```
import androidx.appcompat.app.AppCompatActivity;
import androidx.appcompat.widget.Toolbar;
import android.os.Bundle;

public class BigSwatchActivity extends AppCompatActivity {
    static final String EXTRA_COLOR = "color";

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

        int color = getIntent().getIntExtra(EXTRA_COLOR, 0x7FFF0000);
        Toolbar toolbar = findViewById(R.id.toolbar);

        toolbar.setTitle("#" + Integer.toHexString(color));
        findViewById(R.id.swatch).setBackgroundColor(color);
    }
}
```

(from [TwoActivities/src/main/java/com/commonsware/jetpack/samplerj/activities/BigSwatchActivity.java](#))

...or Kotlin:

```
package com.commonsware.jetpack.sampler.activities

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import kotlinx.android.synthetic.main.activity_big_swatch.*

const val EXTRA_COLOR = "color"

class BigSwatchActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_big_swatch)

        val color = intent.getIntExtra(EXTRA_COLOR, 0x7FFF0000)

        toolbar.title = "#${Integer.toHexString(color)}"
        swatch.setBackgroundColor(color)
    }
}
```

(from [TwoActivities/src/main/java/com/commonsware/jetpack/samplerj/activities/BigSwatchActivity.kt](#))

IMPLEMENTING MULTIPLE ACTIVITIES

First, we call `setContentView(R.layout.activity_big_swatch)` to load up our layout resource.

Then, we need to know what color to display. This activity expects to receive that color in the form of an Intent extra. We have an `EXTRA_COLOR` constant defined to use as the key. We call `getIntent()` to retrieve the Intent that created the activity, and on that we call `getIntExtra()` to retrieve the `EXTRA_COLOR` value. The `get...Extra()` functions that return primitives, like an `Int`, take two parameters: the key to use to find the extra, and the default value to return if the extra is not found. In our case, we use a hard-coded gray value as the default.

Then, we:

- Retrieve the Toolbar widget from the layout
- Set the title in the Toolbar to be the color, with a # at the front
- Set the background color of the swatch widget, so our activity shows up almost entirely in that color (other than the Toolbar)

Starting the Activity

However, this activity will not ever be shown to the user unless we call `startActivity()` at some point to show it, ideally passing the desired color as the `EXTRA_COLOR` extra.

The Toolbar sample app — building on previous ones — handles row clicks in `ColorViewHolder`. So, this app just changes `ColorViewHolder` to start `BigSwatchActivity` instead of showing a `Toast`:

```
package com.commonsware.jetpack.samplerj.activities;

import android.content.Context;
import android.content.Intent;
import android.view.View;
import android.widget.TextView;
import androidx.recyclerview.widget.RecyclerView;

class ColorViewHolder extends RecyclerView.ViewHolder {
    private final View swatch;
    private final TextView label;
    private int color;

    ColorViewHolder(View row) {
        super(row);

        swatch = row.findViewById(R.id.swatch);
        label = row.findViewById(R.id.label);
        row.setOnClickListener(this::showBigSwatch);
    }
}
```

IMPLEMENTING MULTIPLE ACTIVITIES

```
}

void bindTo(Integer color) {
    this.color = color;

    label.setText(label.getContext().getString(R.string.label_template, color));
    swatch.setBackgroundColor(color);
}

private void showBigSwatch(View v) {
    Context context = v.getContext();

    context.startActivity(new Intent(context, BigSwatchActivity.class)
        .putExtra(BigSwatchActivity.EXTRA_COLOR, color));
}
}
```

(from [TwoActivities/src/main/java/com/commonsware/jetpack/samplerj/activities/ColorViewHolder.java](#))

```
package com.commonsware.jetpack.sampler.activities

import android.content.Intent
import android.view.View
import android.widget.TextView
import android.widget.Toast
import androidx.recyclerview.widget.RecyclerView
import kotlinx.android.synthetic.main.row.view.*

class ColorViewHolder(val row: View) : RecyclerView.ViewHolder(row) {
    private val swatch: View = row.swatch
    private val label: TextView = row.label
    private var color: Int = 0xFFFFFFFF

    init {
        row.setOnClickListener(this::showBigSwatch)
    }

    fun bindTo(color: Int) {
        this.color = color
        label.text = label.context.getString(R.string.label_template, color)
        swatch.setBackgroundColor(color)
    }

    private fun showBigSwatch(v: View) {
        val context = v.context

        context.startActivity(
            Intent(context, BigSwatchActivity::class.java)
                .putExtra(EXTRA_COLOR, color)
        )
    }
}
```

IMPLEMENTING MULTIPLE ACTIVITIES

```
}
```

(from [TwoActivities/src/main/java/com/commonsware/jetpack/sampler/activities/ColorViewHolder.kt](#))

We use a method reference to tie our row clicks to a `showBigSwatch()` method. There, we:

- Get a Context from the row View
- Create an explicit Intent, identifying `BigSwatchActivity`
- Add our `EXTRA_COLOR` extra to that Intent with the user's chosen color
- Call `startActivity()` on the Context, passing in our Intent

Now, when the user taps on a color in the list, we see the `BigSwatchActivity`, showing a very large color swatch that probably does not match our Toolbar color very well:

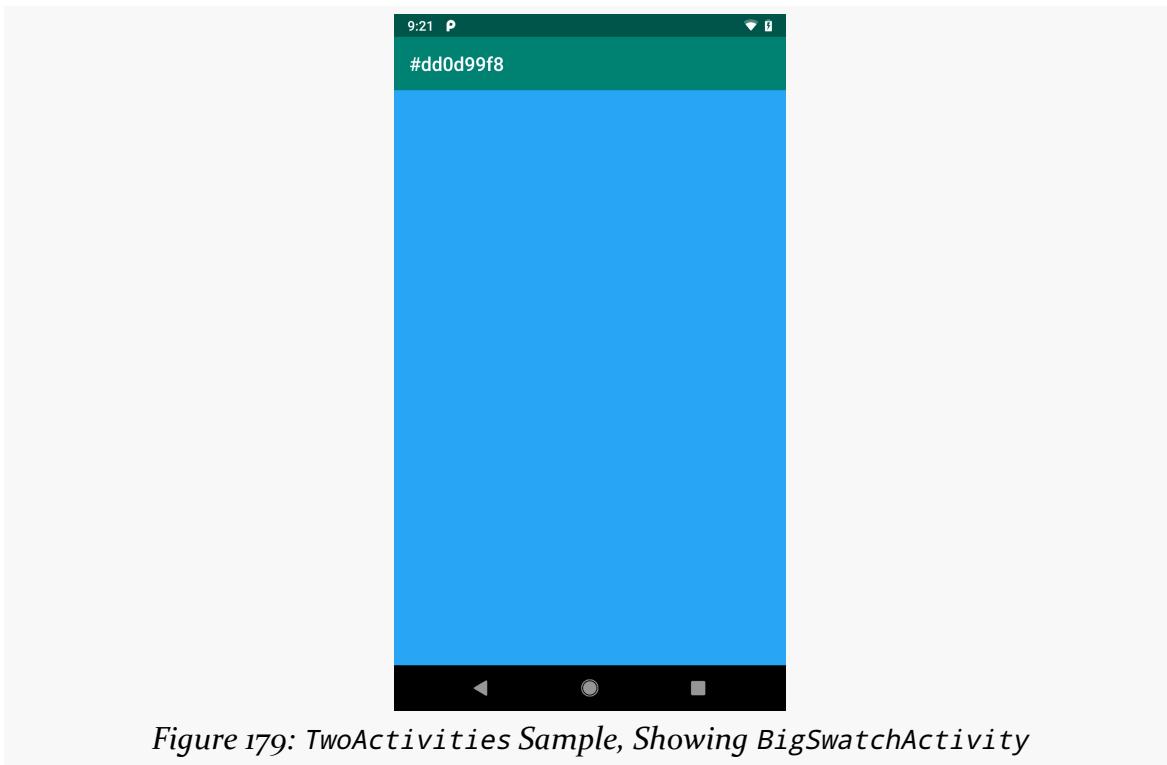


Figure 179: TwoActivities Sample, Showing BigSwatchActivity

Using Implicit Intents

The explicit Intent approach works fine when the activity to be started is one of yours. If you are going to start an activity from some other app, such as a Web

IMPLEMENTING MULTIPLE ACTIVITIES

browser to view a URL, an explicit Intent will be a problem. After all, you have no idea what Web browser will handle the request (Chrome? Firefox? Brave? Dolphin? something else?). Plus, you did not write the Web browser and do not know what classes are in it. And, even if you used tools to peek inside the other app and find out its class structure, the developers of that other app could change their implementation at any point.

Instead, you will use what are referred as the “implicit” Intent structure, which looks an *awful* lot like how the Web works.

If you have done any work on Web apps, you are aware that HTTP is based on verbs applied to URIs:

- We want to GET this image
- We want to POST to this script or controller
- We want to PUT to this REST resource
- Etc.

Android’s implicit Intent model works much the same way, just with a *lot* more verbs.

An implicit Intent is made up of two key pieces:

- A Uri object indicating what we want to act upon, such as a Uri representation of a Web site URL, and
- An action string, identifying the particular action that we want

There are hundreds of action strings that are part of the Android framework, such as:

- ACTION_VIEW, to bring up something that can view whatever the Uri refers to
- ACTION_PICK, to pick something from a collection of somethings
- ACTION_GET_CONTENT, to pick something based on a MIME type (e.g., pick an image)
- ACTION_SEND, to share some text or content with another app, often used for sending an SMS
- And so on

For example, to try to view a Web page, you can use:

```
startActivity(Intent(Intent.ACTION_VIEW, Uri.parse("https://commonsware.com")))
```

Asynchronicity and Results

`startActivity()` is asynchronous. The other activity will not show up until sometime later, particularly after you return from whatever callback you were in when you called `startActivity()` (e.g., `onClick()` of some `View.OnClickListener`).

Normally, this is not much of a problem. However, sometimes one activity might start another, where the first activity would like to know some “results” from the second. For example, the second activity might be some sort of “chooser”, to allow the user to pick a file or contact or song or something, and the first activity needs to know what the user chose. With `startActivity()` being asynchronous, it is clear that we are not going to get that sort of result as a return value from `startActivity()` itself.

To handle this scenario, there is a separate `startActivityForResult()` method. While it too is asynchronous, it allows the newly-started activity to supply a result (via a `setResult()` method) that is delivered to the original activity via an `onActivityResult()` method.

The `ContactPicker` sample module in the [Sampler](#) and [SamplerJ](#) projects demonstrates `startActivityForResult()` and implicit Intent objects.

The Scenario

The `MainActivity` UI is pretty simple: two really big buttons, one labeled “Pick” and one labeled “View”. The business rules are:

- The “View” button should be disabled initially
- The “Pick” button, when clicked, should allow the user to pick a contact from the list of contacts on the device
- Once the user picks a contact, the “View” button should be enabled
- The “View” button, when clicked, should show the user details of that particular contact

There are two ways of going about implementing the pick-a-contact and view-a-contact logic:

1. Write a UI ourselves. This is rather complex. In addition, it would require our app to have access to personally-identifying information (PII) about the user’s contacts. That requires us to [ask for permission](#), and the user might

IMPLEMENTING MULTIPLE ACTIVITIES

- not want to grant us permission, as letting us scan through their contacts may seem scary.
2. We ask some other app — one that already knows how to work with contacts — to let the user pick a contact and view a contact. On many devices, there will be a built-in “Contacts” app that can do those things on our behalf.

ContactPicker takes the second approach.

The Layout

We have two editions of the `activity_main` layout resource. One is in the traditional `res/layout/` directory, and it has two buttons:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="@dimen/container_padding">

    <Button
        android:id="@+id/pick"
        android:layout_width="0dp"
        android:layout_height="0dp"
        android:text="@string/pick_caption"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintHeight_percent="0.5"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <Button
        android:id="@+id/view"
        android:layout_width="0dp"
        android:layout_height="0dp"
        android:enabled="false"
        android:text="@string/view_caption"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintHeight_percent="0.5"
        app:layout_constraintStart_toStartOf="parent" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

(from [ContactPicker/src/main/res/layout/activity_main.xml](#))

IMPLEMENTING MULTIPLE ACTIVITIES

The other is in `res/layout-w640dp/`, and it has two buttons:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="@dimen/container_padding">

    <Button
        android:id="@+id/pick"
        android:layout_width="0dp"
        android:layout_height="0dp"
        android:text="@string/pick_caption"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintWidth_percent="0.5" />

    <Button
        android:id="@+id/view"
        android:layout_width="0dp"
        android:layout_height="0dp"
        android:enabled="false"
        android:text="@string/view_caption"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintWidth_percent="0.5" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

(from [ContactPicker/src/main/res/layout-w640dp/activity_main.xml](#))

IMPLEMENTING MULTIPLE ACTIVITIES

The difference is subtle, but the way we have the constraints set up, the res/layout/ edition of the layout has the two buttons be vertically stacked:

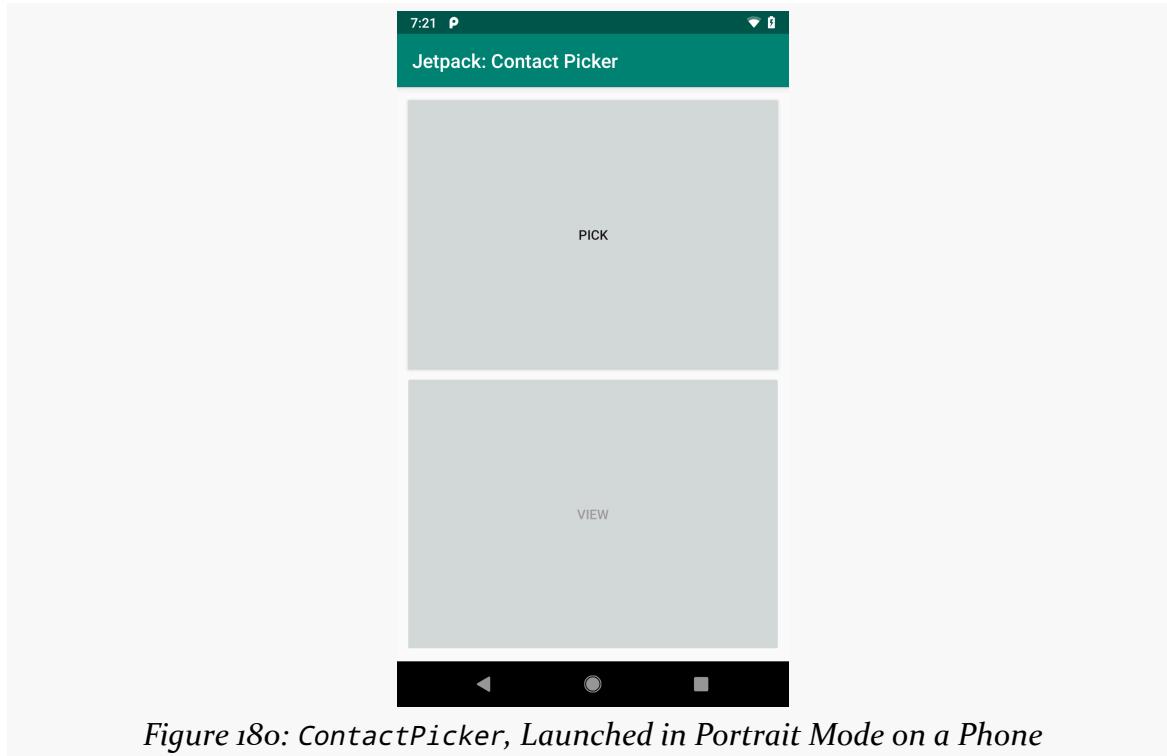


Figure 18o: ContactPicker, Launched in Portrait Mode on a Phone

IMPLEMENTING MULTIPLE ACTIVITIES

But, on devices with screens wider than 4", the res/layout-w640dp/ edition of the layout will be used, and it has the constraints set up for the buttons to be side-by-side:

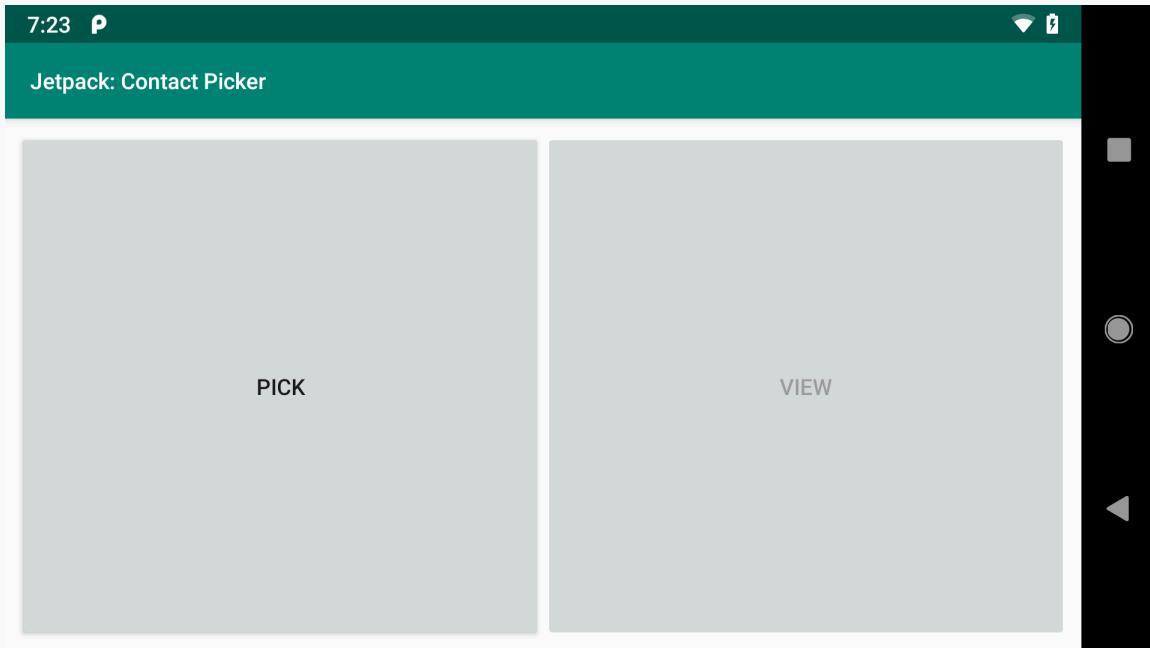


Figure 181: ContactPicker, Launched in Landscape Mode on a Phone

Picking a Contact

In MainActivity, we set up a View.OnClickListener for the pick button, to allow the user to pick a contact:

```
findViewById(R.id.pick).setOnClickListener(v -> {
    try {
        startActivityForResult(new Intent(Intent.ACTION_PICK,
            ContactsContract.Contacts.CONTENT_URI), REQUEST_PICK);
    }
    catch (Exception e) {
        Toast.makeText(this, R.string.msg_pick_error,
            Toast.LENGTH_LONG).show();
    }
});
```

(from [ContactPicker/src/main/java/com/commonsware/jetpack/samplerj/contact/MainActivity.java](#))

IMPLEMENTING MULTIPLE ACTIVITIES

```
pick.setOnClickListener {
    try {
        startActivityForResult(
            Intent(
                Intent.ACTION_PICK,
                ContactsContract.Contacts.CONTENT_URI
            ), REQUEST_PICK
        )
    } catch (e: Exception) {
        Toast.makeText(this, R.string.msg_pick_error, Toast.LENGTH_LONG).show()
    }
}
```

(from [ContactPicker/src/main/java/com/commonsware/jetpack/sampler/contact/MainActivity.kt](#))

The specific implicit Intent that we are going to try is ACTION_PICK. This allows the user to pick an item out of some collection of items. The collection is represented by a Uri, just as the URL of the Web page for ACTION_VIEW is represented by a Uri.

The specific Uri that we use is ContactsContract.Contacts.CONTENT_URI. This is a Uri provided by the framework that points to a system-supplied ContentProvider that is the standard location on the device for storing the user's contacts.

We wrap that Uri and ACTION_PICK in an Intent and pass that to startActivityForResult(). ACTION_PICK is designed for use with startActivityForResult(), since the point of having the user pick something is for us to be able to find out the “result” (i.e., what the user picked).

startActivityForResult() takes a second parameter: an Int that identifies this startActivityForResult() call from any others that we might be making in this activity. Here, we have it defined as a constant, REQUEST_PICK.

The whole startActivityForResult() call is wrapped in a try/catch block. When you use an implicit Intent to start an activity, there is a chance that there is no activity to handle the request. In our case, there are a few reasons why we might not be able to pick a contact, including:

- The app is running on a device that does not track contacts, such as perhaps some tablets or TVs
- The user is restricted from accessing contacts, due to work profiles or similar limitations imposed by the owner of the device
- The user might have disabled all apps that support our implicit Intent in the Settings app, if the user did not think that she would be using those apps

IMPLEMENTING MULTIPLE ACTIVITIES

With an explicit Intent to one of our activities, we *know* that it should succeed, barring some bug in the app. With an implicit Intent, our request might not succeed. If it fails, the most likely failure is an `ActivityNotFoundException`, indicating that there was no activity identified that could handle our Intent.

So, we use `try/catch` to catch any such exceptions, showing an apology `Toast` in response.

Getting and Retaining the Contact

If you call `startActivityForResult()` on an activity, as we do here, you need to implement the corresponding `onActivityResult()` callback function on the same activity. In our case, for an `ACTION_PICK` Intent, this will be called if either:

- The user picks something from the collection, and the other activity returns control to us with an indication of what the user picked
- The user presses the BACK button to exit the activity that we started, returning control to us that way

`onActivityResult()` gets three parameters:

- The `Int` that we passed as the second parameter to `startActivityForResult()` (the “request code”)
- An `Int` that holds `RESULT_OK` if the user picked something or `RESULT_CANCELED` if the user did not (the “response code”)
- An `Intent` object

In the particular case of `ACTION_PICK`, if we get a `RESULT_OK` response code, then the `Uri` in the `Intent` that we are given will be a `Uri` identifying what the user picked. In our case, that would be the `Uri` of a specific contact.

So, `MainActivity` has `onActivityResult()` collect that data:

```
@Override  
protected void onActivityResult(int requestCode, int resultCode,  
    @Nullable Intent data) {  
    if (requestCode == REQUEST_PICK && resultCode == RESULT_OK &&  
        data != null) {  
        vm.contact = data.getData();  
        updateViewButton();  
    }  
}
```

IMPLEMENTING MULTIPLE ACTIVITIES

(from [ContactPicker/src/main/java/com/commonsware/jetpack/samplerj/contact/MainActivity.java](#))

```
override fun onActivityResult(
    requestCode: Int,
    resultCode: Int,
    data: Intent?
) {
    if (requestCode == REQUEST_PICK && resultCode == Activity.RESULT_OK &&
        data != null
    ) {
        vm.contact = data.data
        updateViewButton()
    }
}
```

(from [ContactPicker/src/main/java/com/commonsware/jetpack/samplerj/contact/MainActivity.kt](#))

If we got a contact Uri, we do two things.

First, we update a ContactViewModel that is serving as the viewmodel for this activity. ContactViewModel has a contact field to hold our chosen contact (or null if there has not been a choice yet):

```
package com.commonsware.jetpack.samplerj.contact;

import android.net.Uri;
import android.os.Bundle;
import androidx.lifecycle.ViewModel;

public class ContactViewModel extends ViewModel {
    private static final String STATE_CONTACT = "contact";
    Uri contact;

    ContactViewModel(Bundle state) {
        if (state != null) {
            contact = state.getParcelable(STATE_CONTACT);
        }
    }

    void onSaveInstanceState(Bundle state) {
        state.putParcelable(STATE_CONTACT, contact);
    }
}
```

(from [ContactPicker/src/main/java/com/commonsware/jetpack/samplerj/contact/ContactViewModel.java](#))

```
package com.commonsware.jetpack.samplerj.contact
```

IMPLEMENTING MULTIPLE ACTIVITIES

```
import android.net.Uri
import android.os.Bundle
import androidx.lifecycle.ViewModel

private const val STATE_CONTACT = "contact"

class ContactViewModel(state: Bundle?) : ViewModel() {
    var contact: Uri? = state?.getParcelable(STATE_CONTACT)

    fun onSaveInstanceState(state: Bundle) {
        state.putParcelable(STATE_CONTACT, contact)
    }
}
```

(from [ContactPicker/src/main/java/com/commonsware/jetpack/sampler/contact/ContactViewModel.kt](#))

We also call an `updateViewButton()` function that marks the view button as being enabled if we happen to have a contact now:

```
private void updateViewButton() {
    if (vm.contact != null) {
        viewButton.setEnabled(true);
    }
}
```

(from [ContactPicker/src/main/java/com/commonsware/jetpack/samplerj/contact/MainActivity.java](#))

```
private fun updateViewButton() {
    if (vm.contact != null) {
        view.isEnabled = true
    }
}
```

(from [ContactPicker/src/main/java/com/commonsware/jetpack/sampler/contact/MainActivity.kt](#))

In addition, we call `updateViewButton()` in `onCreate()`, after getting our `ContactViewModel`, so we update the view button to be enabled after a configuration change, if appropriate:

```
setContentView(R.layout.activity_main);

viewButton = findViewById(R.id.view);

vm = ViewModelProviders
    .of(this, new ContactViewModelFactory(savedInstanceState))
    .get(ContactViewModel.class);
```

IMPLEMENTING MULTIPLE ACTIVITIES

```
updateViewButton();
```

(from [ContactPicker/src/main/java/com/commonsware/jetpack/samplerj/contact/MainActivity.java](#))

```
setContentView(R.layout.activity_main)

vm = ViewModelProviders
    .of(this, ContactViewModelFactory(savedInstanceState))
    .get(ContactViewModel::class.java)
updateViewButton()
```

(from [ContactPicker/src/main/java/com/commonsware/jetpack/samplerj/contact/MainActivity.kt](#))

So, the net is that once the user picks the contact, the view button becomes enabled, and we hold onto the contact Uri across configuration changes, so we do not lose track of who the user picked.

Viewing the Contact

Our View.OnClickListener for the view button wraps our contact Uri in an ACTION_VIEW Intent and tries to start an activity to view that contact:

```
viewButton.setOnClickListener(
    v -> {
        try {
            startActivity(new Intent(Intent.ACTION_VIEW, vm.contact));
        }
        catch (Exception e) {
            Toast.makeText(this, R.string.msg_view_error,
                Toast.LENGTH_LONG).show();
        }
    });
});
```

(from [ContactPicker/src/main/java/com/commonsware/jetpack/samplerj/contact/MainActivity.java](#))

```
view.setOnClickListener {
    try {
        startActivity(Intent(Intent.ACTION_VIEW, vm.contact))
    } catch (e: Exception) {
        Toast.makeText(this, R.string.msg_view_error, Toast.LENGTH_LONG).show()
    }
}
```

(from [ContactPicker/src/main/java/com/commonsware/jetpack/samplerj/contact/MainActivity.kt](#))

Once again, there might not be an activity available to the user to handle this ACTION_VIEW request, we wrap our startActivity() call in a try/catch block to

IMPLEMENTING MULTIPLE ACTIVITIES

deal with any possible exceptions.

If you run this on a device or emulator that has a working “contacts” app with 1+ contacts in it, clicking the “Pick” button lets you pick a contact, and then clicking the “View” button views details of the picked contact. In both cases, the contacts app is supplying the UI for picking and viewing contacts.

The Inverse: <intent-filter>

When we use an explicit Intent with `startActivity()`, it is fairly clear how Android determines what activity to display: it is the one whose class is listed in the Intent.

When we use an implicit Intent — one with an action string and maybe a Uri, but no class — somehow Android needs to find out what activity (or activities) can handle that.

This is where the `<intent-filter>` comes in.

In our manifest, inside of an `<activity>` element, you can optionally have one or more `<intent-filter>` elements. These describe certain implicit Intent structures that our activity can handle. If some code tries starting an implicit Intent with a matching structure, our activity will be considered a candidate. So, in the case of our `ACTION_PICK` and `ACTION_VIEW` requests in the preceding sections, we are hoping that there are 1+ activities with an `<intent-filter>` that matches our Intent.

Most apps will have an activity with one particular `<intent-filter>`:

```
<intent-filter>
    <action android:name="android.intent.action.MAIN" />

    <category android:name="android.intent.category.LAUNCHER" />
</intent-filter>
```

(from [ContactPicker/src/main/AndroidManifest.xml](#))

This would match an Intent like:

```
Intent(Intent.ACTION_MAIN).addCategory(Intent.CATEGORY_LAUNCHER)
```

A “category” is simply an identifier of related Intent structures and is one facet of an implicit Intent, along with things like the action string and “data” Uri.

IMPLEMENTING MULTIPLE ACTIVITIES

A launcher can ask Android “what activities support this Intent?”. Android will return a list of matches, and the launcher can use this to provide options for the user. When the user clicks on a *specific* activity, then the launcher has the details to create an explicit Intent, which it can use to start the activity. Since this is the standard behavior of launchers, and since most Android apps want to have 1+ activities appear in the launcher, most apps will have 1+ activities with the MAIN/LAUNCHER `<intent-filter>`.

Most apps will not need `<intent-filter>` on an `<activity>` beyond this one, though. However, they might *use* other apps that have activities with other `<intent-filter>` options, such as the ACTION_PICK and ACTION_VIEW scenarios.

We will see an example of this sort of structure in [an upcoming chapter](#), where we will examine an app that supports ACTION_SEND, the standard Intent action for “share” options in apps.

Licensed solely for use by Patrocinio Rodriguez

Adopting Fragments

Activities are fine, but they are fairly inflexible. Development is moving away from using activities as the core foundation of our UI. Activities will always exist, but many times we use an activity mostly as a simple container for other UI logic.

And, frequently, that UI logic is held in fragments.

The original vision for fragments was to make it easier to support early Android tablets, allowing you to assemble tablet-sized UIs by snapping together a bunch of phone-sized UIs that you use individually on phones.

Over time, fragments were used in more places and for more reasons. While fragments are not required, Google strongly encourages their use. Jetpack specifically advocates having an app be a single activity, with fragments for each “screen” of information.

This chapter will cover basic uses of fragments.

The Six Questions

In the world of journalism, the basics of any news story consist of six questions, [the Five Ws and How](#). Here, we will apply those six questions to help frame what we are talking about with respect to fragments.

What?

Fragments are not activities, though they can be used by activities.

Fragments are not containers (i.e., subclasses of `ViewGroup`), though typically they

create a ViewGroup.

Rather, you should think of fragments as being units of UI reuse. You define a fragment, much like you might define an activity, with layouts and lifecycle methods and so on.

However, at that point, you can use fragments in different ways, with two main patterns being:

- Having one activity switch between fragments based on user input
- Having multiple fragments on the screen at once, perhaps to take better advantage of larger screen sizes on tablets, Chromebooks, etc.

Functionally, fragments are Java/Kotlin classes, extending from a base Fragment class.

Where??

Fragments will appear on the screen where you tell them to appear. There are two main approaches for this:

- Having <fragment> elements in layout resources
- Having FrameLayout containers in layout resources, where you then supply the fragments to put in those containers at runtime

The first approach is for cases where the fragment will always be shown (“static fragments”) by the activity. The second approach is for cases where the fragment might be swapped out for another fragment, or whether the fragment might be conditionally shown based on user input, screen size, or other criteria (“dynamic fragments”).

Who?!?

Many fragments you will write yourself, just as you write your activities and other application code.

However, libraries can also contribute fragment implementations. We will see an example of this [in the next chapter](#), with a Google-supplied Jetpack library offering a fragment that we use directly. Third party libraries might offer fragments as part of their API, for you to use or subclass as appropriate.

When?!!?

In modern, Jetpack-centric app development, often we define our fragments at the outset, as part of building our UI.

It is certainly possible to take an app that has several activities and rewrite it such that it has one activity and several fragments. This is a bit tedious, but it might prove necessary at some point for legacy projects.

WHY?!?!?

Ah, this is the big question. If we have managed to make it this far through the book without fragments, and we do not necessarily need fragments to create Android applications, what is the point? Why would we bother?

There are many reasons for using fragments, some of which were hinted at above.

Break the Intent Barrier

Sharing data between activities is a problem. We can use extras on Intents, but they are limited in terms of size and data types. Other than that, we are limited to using singletons or other global objects, and those run the risk of memory leaks and related problems.

However, a single activity can have several fragments and switch between them at will. Plus, an activity can work directly with its fragments and share things, such as having a shared ViewModel. So, fragments give us ways of having different “screens” while still allowing everything to work using more-or-less normal Java/Kotlin object interactions.

To an extent, you can draw a parallel to Web development. Early Web development, and even a lot of modern development, has each “screen” be a separate Web page at a separate URL, fetched from the server. This works, but it has a similar problem to multiple activities in Android: one Web page cannot directly work with the DOM or JavaScript objects of another Web page. In the past decade, there has been a lot of movement towards “single-page applications”, where a single URL loading a single Web page has multiple screens’ worth of content, courtesy of lots of DOM rewriting. While this has its own set of problems, it gets past the separate-pages limitations, allowing multiple screens to share a common set of JavaScript objects.

Decomposition

Technically, one does not need fragments to allow for a single activity to represent multiple screens. We have used `setContentView()` to populate the activity's UI, and you can call that as many times as needed. Each `setContentView()` call replaces whatever the previous "content view" was with a new one. Or, you can do other things with the activity's view hierarchy to change the UI: hide and show widgets, add and remove widgets, etc.

However, if you go this route, you run the risk of having a single monster activity trying to manage all of this logic. For ease of maintenance and testing, having some decomposition is useful, so a set of Java/Kotlin classes can manage an individual screen, while the activity orchestrates which screen is seen at any point in time.

Fragments offer a foundation for this sort of decomposition. It is not the only option, and it may not be the absolute best option. However, it is *Google's* best option, and Jetpack reflects a fragment-centric view of Android app development.

Screen Sizes

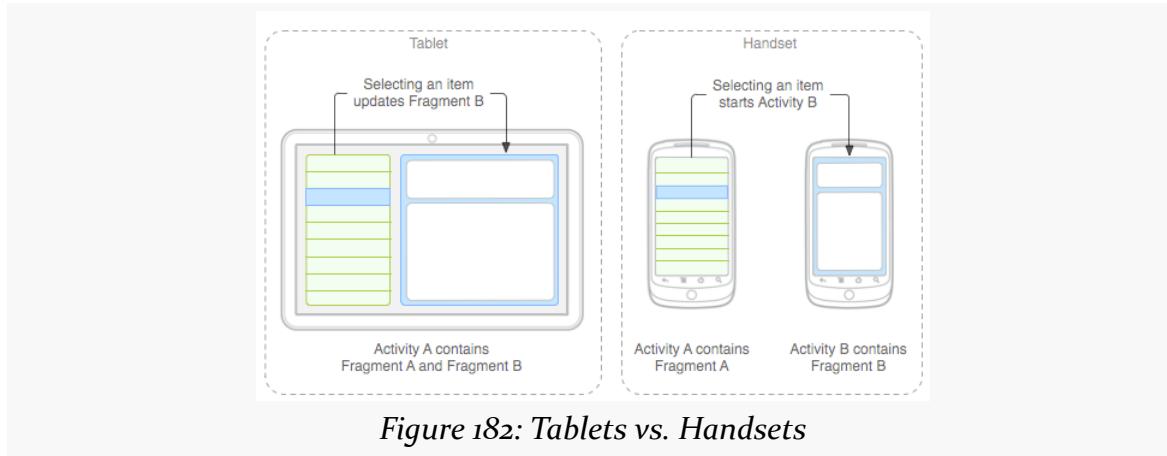
The original rationale for fragments was to make it easier to support multiple screen sizes.

Android started out supporting phones. Phones may vary in size, from tiny ones with less than 3" diagonal screen size, to monsters that are well over 5". However, those variations in screen size pale in comparison to the differences between phones and tablets, or phones and Chromebooks.

Some applications will simply expand to fill larger screen sizes. Many games will take this approach, simply providing the user with bigger interactive elements, bigger game boards, etc.

ADOPTING FRAGMENTS

Part of the original vision for fragments was that one could assemble a tablet UI from a collection of phone screens, side by side.



(the above image is reproduced from work created and [shared by the Android Open Source Project](#) and used according to terms described in [the Creative Commons 2.5 Attribution License](#))

The user can access all of that functionality at once on a tablet, whereas they would have to flip back and forth between separate screens on a phone.

For applications that can fit this design pattern, fragments allow you to support phones and tablets from one code base. The fragments can be used by individual activities on a phone, or they can be stitched together by a single activity for a tablet.

OMGOMGOMG, HOW?!?!??

Well, answering that question is what the rest of this chapter is for!

Where You Get Your Fragments From

We use a `Fragment` class as the basis for our fragments.

However, if you rummage through the Android SDK JavaDocs, you will find that there are *three* classes named `Fragment`... and they are all incompatible with each other.

This sucks.

The Jetpack edition of fragments is in the `androidx` namespace, specifically `androidx.fragment.app.Fragment`. That is the Fragment class that we will be using in this book.

The Jetpack Fragment class is based on the one from the Android Support Library. If you see references to `android.support.v4.app.Fragment`, that is the Android Support Library edition.

You will also find `android.app.Fragment`. This is the framework implementation of fragments. It has been officially deprecated, with Google steering developers towards a library-based implementation. The problem is that fragments have had quite a few bugs over the years. Framework classes are only updated when the OS is updated, and for many users that means the framework classes are rarely updated and are usually out of date. By contrast, *you* as the app developer control which version of the libraries that you use, so you can ensure that you are using up-to-date versions that contain relevant bug fixes.

Fortunately, Java and Kotlin are strongly-typed languages, so it will be difficult for you to accidentally use the wrong Fragment... though “difficult” is not “impossible”.

Static vs. Dynamic Fragments

Sometimes, you will have a fragment that should be around as long as your activity is around. For that, you can use “static fragments”, where you use a `<fragment>` element in a layout resource to specify where the fragment should go and how big it should be. We will explore this more in [the next chapter](#).

More often, though, your fragments will come and go, based on user input:

- You start by showing a list of stuff
- The user clicks on an item in the list, and you show details about the item that the user clicked on
- The user clicks an “edit” option in the toolbar, and so you show an edit form to allow the user to modify the item
- The user clicks BACK to return to the details, then BACK again to return to the list

In all of these cases, in a fragment-based UI, you will use dynamic fragments.

Roughly speaking, there are two ways of employing dynamic fragments:

1. Manually, using `FragmentManager` and `FragmentTransaction` classes, as we will explore in this chapter
2. Using the Jetpack Navigation component, which we will see in [the next chapter](#)

Fragments, and What You Have Seen Already

Fragments can do most everything of what we have seen so far that activities can do:

- Fragments can use layout resources to show a UI
- Fragments can have a `ViewModel` — in fact, they can not only have their own, but they can share a `ViewModel` with the activity that hosts them
- Fragments have [lifecycle methods](#), including both ones that you see in activities (e.g., `onCreate()` and `onDestroy()`) and ones that are distinct for fragments (e.g., `onAttach()` and `onDetach()`)

ToDo, or Not ToDo? That Is the Question

To see dynamic fragments in action, let's turn to the `FragmentManual` sample module in the [Sampler](#) and [SamplerJ](#) projects.

This sample app is relatively complicated. We have:

- One activity
- Two fragments
- Two viewmodels
- A `RecyclerView`
- A few layout resources
- And other stuff besides

What We're Building

The sample app implements a to-do list, showing a list of to-do items along with details of an individual item.

This code mimics some of the code from [Exploring Android](#), though with a number of differences, to illustrate the manual use of dynamic fragments.

ADOPTING FRAGMENTS

From the user's standpoint, launching the app brings up a list of to-do items, where the list of things to do is hard-coded in the app:

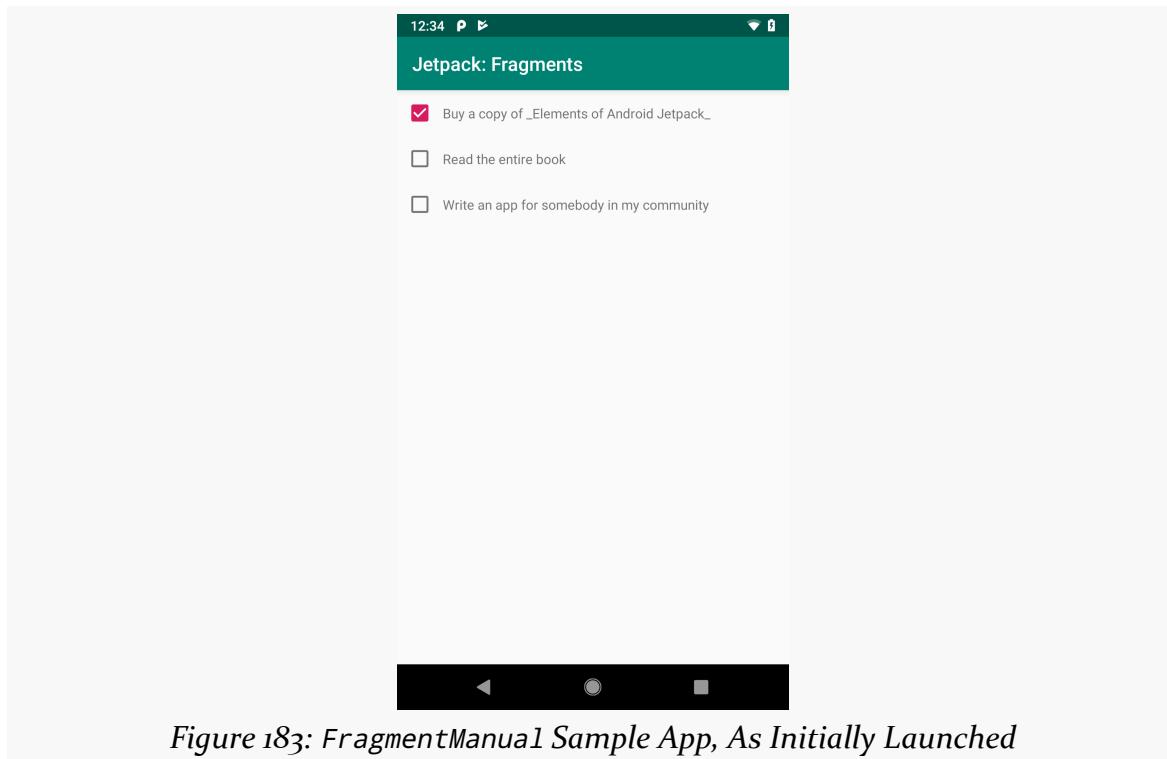


Figure 183: FragmentManual1 Sample App, As Initially Launched

ADOPTING FRAGMENTS

Tapping on the description of an item brings up additional details, including the creation date and, in some cases, some notes:

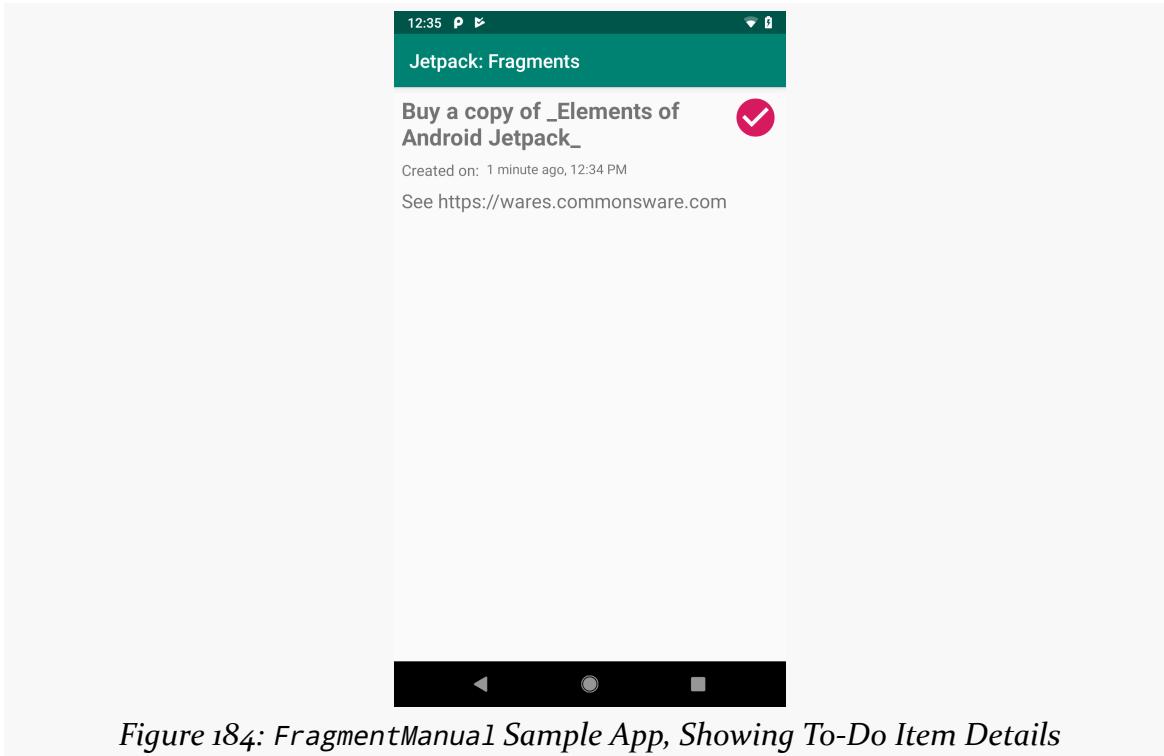


Figure 184: FragmentManual Sample App, Showing To-Do Item Details

This version of the app is read-only, though readers of *Exploring Android* will create a version of this app that allows users to add, edit, and remove to-do items.

The Model

Objects that represent our “business data” are usually called “model objects”. In a to-do list app, we need objects that represent to-do items, and those would be considered model objects.

So the sample app has a `ToDoModel` class that holds onto things like the description (e.g., “Write an app for somebody in my community”), the creation time, and so forth.

There are five pieces of data that we want to track:

- The description, as mentioned above
- Whether or not the item is completed, which will control things like

ADOPTING FRAGMENTS

whether the CheckBox is checked in the list and whether the checkmark icon shows up in the detail screen

- Some optional additional notes about the item (e.g., "Talk to some people at non-profit organizations to see what they need!")
- The time this item was created
- Some sort of unique identifier — while this is not important now, it will be later when [we start persisting this data in a database](#)

Functionally, the Java and Kotlin editions of this class are identical. The code, though, winds up being a fair bit different.

Kotlin

In Kotlin, we can use a simple data class, with individual `val` properties for the five pieces of data mentioned above:

```
data class ToDoModel(  
    val id: String,  
    val description: String,  
    val isCompleted: Boolean = false,  
    val notes: String? = null,  
    val createdOn: Calendar = Calendar.getInstance()  
)
```

(from [FragmentManual/src/main/java/com/commonsware/jetpack/sampler/fragments/ToDoModel.kt](#))

Of particular interest:

- `isCompleted` is defaulted to `false`
- `notes` can be `null` and is defaulted to `null`
- `createdOn` is defaulted to the current date and time

Because this is a data class, we get things like a `copy()` function, `toString()`, `equals()`, and so forth “for free”, courtesy of the Kotlin compiler.



You can learn more about data classes in the "Data Class" chapter of [Elements of Kotlin!](#)

ADOPTING FRAGMENTS

Java

We want the same basic characteristics in Java. It requires a bit more code:

```
public class ToDoModel {
    @NonNull
    public final String id;
    @NonNull
    public final String description;
    public final boolean isCompleted;
    @Nullable
    public final String notes;
    @NonNull
    public final Calendar createdOn;

    ToDoModel(@NonNull String id, @NonNull String description,
              boolean isCompleted, @Nullable String notes,
              @NonNull Calendar createdOn) {
        this.id = id;
        this.description = description;
        this.isCompleted = isCompleted;
        this.notes = notes;
        this.createdOn = createdOn;
    }
}
```

(from [FragmentManual/src/main/java/com/commonsware/jetpack/samplerj/fragments/ToDoModel.java](#))

We have fields for each of the five pieces of data. Those that are objects are marked with `@Nullable` and `@NotNull` annotations, so that Android Studio can help ensure that we use them properly with respect to null values. The fields are marked as `final`, which is as close as Java comes to the immutability that you get with `val` properties in a Kotlin data class.

We also have a constructor to fill in those fields. However, we do not have default parameter values, so the caller will need to supply all five pieces of data whenever it creates an instance of `ToDoModel`.

The Repository

Repositories are a common pattern in modern Android app development.

The idea of a repository is to isolate details of how data is stored from the UI that is using that data. The UI should neither know nor care whether the data is stored in a database, in some other type of file, or on a Web service. Similarly, the code that

ADOPTING FRAGMENTS

handles the data storage should not care whether the data is represented visually in fields, checkboxes, or other sorts of widgets. Having a clean boundary between “the stuff that stores data” and “the stuff that uses data” can also help with testing. We will explore the repository pattern in greater detail [later in the book](#).

For now, though, we need *something* that can hold a few fake ToDoModel instances, so we may as well set up a ToDoRepository for that.

A real repository can be very complicated. This sample app has no actual data storage, so our ToDoRepository is fairly trivial:

```
package com.commonsware.jetpack.samplerj.fragments;

import java.util.ArrayList;
import java.util.Calendar;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.UUID;
import androidx.annotation.NonNull;
import androidx.annotation.Nullable;

class ToDoRepository {
    static final ToDoRepository INSTANCE = new ToDoRepository();
    private final Map<String, ToDoModel> items = new HashMap<>();

    private ToDoRepository() {
        add(new ToDoModel(UUID.randomUUID().toString(),
            "Buy a copy of _Elements of Android Jetpack_", true,
            "See https://wares.commonsware.com",
            Calendar.getInstance()));
        add(
            new ToDoModel(UUID.randomUUID().toString(), "Read the entire book",
                false, null,
                Calendar.getInstance()));
        add(new ToDoModel(UUID.randomUUID().toString(),
            "Write an app for somebody in my community", false,
            "Talk to some people at non-profit organizations to see what they need!",
            Calendar.getInstance()));
    }

    @NonNull
    List<ToDoModel> getItems() {
        return new ArrayList<>(items.values());
    }
}
```

ADOPTING FRAGMENTS

```
@Nullable
ToDoModel findItemById(String id) {
    return items.get(id);
}

private void add(ToDoModel model) {
    items.put(model.id, model);
}
}
```

(from [FragmentManual/src/main/java/com/commonsware/jetpack/samplerj/fragments/ToDoRepository.java](#))

```
package com.commonsware.jetpack.sampler.fragments

import java.util.*

object ToDoRepository {
    private val items = listOf(
        ToDoModel(
            id = UUID.randomUUID().toString(),
            description = "Buy a copy of Elements of Android Jetpack",
            isCompleted = true,
            notes = "See https://wares.commonsware.com"
        ), ToDoModel(
            id = UUID.randomUUID().toString(),
            description = "Read the entire book"
        ), ToDoModel(
            id = UUID.randomUUID().toString(),
            description = "Write an app for somebody in my community",
            notes = "Talk to some people at non-profit organizations to see what they need!"
        )
    ).associateBy { it.id }

    fun getItems(): List<ToDoModel> = items.values.toList()

    fun findItemById(id: String) = items[id]
}
```

(from [FragmentManual/src/main/java/com/commonsware/jetpack/sampler/fragments/ToDoRepository.kt](#))

The repository holds a Map of ToDoModel objects, keyed by their id value. It has functions to retrieve all models or a single model by ID.

The repository is set up as a singleton, either using a static field in Java or simply having the repository be an object in Kotlin. That works, but it is not very flexible. [Later in the book](#), we will explore ways in which our app can use the repository like a singleton, yet we would have the ability to swap out implementations of the repository to use in different scenarios, through what is known as “dependency injection”.

The DisplayFragment

We need to display a `ToDoModel` to the user. And, in this sample app, we want to use fragments for that. So, we have a `DisplayFragment` that will fill that role. That `DisplayFragment` has its own layout and viewmodel, just like an activity might. The Java/Kotlin code for the fragment is a bit different than what you would see in an activity, but not *that* different.

So, let's look at each of the pieces individually.

The Layout

The `todo_display` layout resource represents the UI that we want to use to show a `ToDoModel` to the user:

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

    <data>

        <variable
            name="model"
            type="com.commonware.jetpack.sampler.fragments.ToDoModel" />

        <variable
            name="createdOnFormatted"
            type="java.lang.CharSequence" />

        <import type="android.view.View" />
    </data>

    <androidx.constraintlayout.widget.ConstraintLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent">

        <ImageView
            android:id="@+id/completed"
            android:layout_width="@dimen/checked_icon_size"
            android:layout_height="@dimen/checked_icon_size"
            android:layout_marginEnd="8dp"
            android:layout_marginTop="8dp"
            android:contentDescription="@string/is_completed"
            android:src="@drawable/ic_check_circle_black_24dp"
            android:tint="@color/colorAccent"
```

ADOPTING FRAGMENTS

```
        android:visibility="@{model.isCompleted ? View.VISIBLE : View.GONE}"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

<TextView
    android:id="@+id/desc"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_marginEnd="8dp"
    android:layout_marginStart="8dp"
    android:layout_marginTop="8dp"
    android:text="@{model.description}"
    android:textSize="@dimen/desc_view_size"
    android:textStyle="bold"
    app:layout_constraintEnd_toStartOf="@+id/completed"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />

<TextView
    android:id="@+id/label_created"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginStart="8dp"
    android:layout_marginTop="8dp"
    android:text="@string/created_on"
    android:textSize="@dimen/created_on_size"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/desc" />

<TextView
    android:id="@+id/created_on"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_marginEnd="8dp"
    android:layout_marginStart="8dp"
    android:layout_marginTop="8dp"
    android:text="@{createdOnFormatted}"
    app:layout_constraintEnd_toStartOf="@+id/completed"
    app:layout_constraintStart_toEndOf="@+id/label_created"
    app:layout_constraintTop_toBottomOf="@+id/desc" />

<TextView
    android:id="@+id/notes"
    android:layout_width="0dp"
    android:layout_height="0dp"
    android:layout_marginBottom="8dp"
    android:layout_marginEnd="8dp"
    android:layout_marginStart="8dp"
```

ADOPTING FRAGMENTS

```
    android:layout_marginTop="8dp"
    android:text="@{model.notes}"
    android:textSize="@dimen/notes_size"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/label_created" />

</androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```

(from [FragmentManual/src/main/res/layout/todo_display.xml](#))

On the whole, this is a basic ConstraintLayout-based layout resource, akin to those that we have seen previously in the book. It uses [data binding](#) to populate the core widgets.

The most interesting widget of the lot, though, is the ImageView:

```
<ImageView
    android:id="@+id/completed"
    android:layout_width="@dimen/checked_icon_size"
    android:layout_height="@dimen/checked_icon_size"
    android:layout_marginEnd="8dp"
    android:layout_marginTop="8dp"
    android:contentDescription="@string/is_completed"
    android:src="@drawable/ic_check_circle_black_24dp"
    android:tint="@color/colorAccent"
    android:visibility="@{model.isCompleted ? View.VISIBLE : View.GONE}"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
```

(from [FragmentManual/src/main/res/layout/todo_display.xml](#))

The icon that we use — ic_check_circle_black_24dp — is one imported from clip art through the Vector Asset wizard, the same way that we setup Toolbar icons in [an earlier chapter](#). However, we want it to show up in a different color. We could hand-modify the XML of the resource. A simpler approach is used here, where we use android:tint to apply our colorAccent color to the widget. All black pixels will be replaced with pixels based on our accent color. This way, if we elect to change the accent color, changing the color resource affects this ImageView without having to modify the drawable itself.

Also, the ImageView uses a slightly complicated data binding expression:

```
    android:visibility="@{model.isCompleted ? View.VISIBLE : View.GONE}"
```

ADOPTING FRAGMENTS

(from [FragmentManual/src/main/res/layout/todo_display.xml](#))

We want the icon to be visible if the `isCompleted` property is `true`. The `android:visibility` attribute handles this, with three possible values:

- `visible`
- `invisible`: the widget's pixels are not drawn, but it still takes up space on the screen
- `gone`: the widget is ignored entirely

We want to toggle the visibility to be `visible` or `gone` based on whether the to-do item has been completed.

Here, we use a “ternary expression”. This mimics the ternary expressions available in Java (though not in Kotlin). A ternary expression is made up of three pieces:

- To the left of the `?` is some boolean value — in our case, the value of the `isCompleted` property
- To the left of the `:` is the value to use if the boolean value is `true`
- To the right of the `:` is the value to use if the boolean value is `false`

The end result is that the user sees a tinted checkmark if the to-do item is completed.

The ViewModel

`DisplayFragment` has a `DisplayViewModel` to hold its data across configuration changes:

```
package com.commonsware.jetpack.samplerj.fragments;

import androidx.lifecycle.ViewModel;

public class DisplayViewModel extends ViewModel {
    private ToDoModel model;

    ToDoModel getModel(String id) {
        if (model == null) {
            model = ToDoRepository.INSTANCE.findItemById(id);
        }

        return model;
    }
}
```

ADOPTING FRAGMENTS

```
}
```

(from [FragmentManual/src/main/java/com/commonsware/jetpack/samplerj/fragments/DisplayViewModel.java](#))

```
package com.commonsware.jetpack.sampler.fragments

import androidx.lifecycle.ViewModel

class DisplayViewModel : ViewModel() {
    private var model: ToDoModel? = null

    fun getModel(id: String) =
        model ?: ToDoRepository.findItemById(id).also { model = it }
}
```

(from [FragmentManual/src/main/java/com/commonsware/jetpack/sampler/fragments/DisplayViewModel.kt](#))

Each instance of our `DisplayFragment` will display a single `ToDoModel` instance. The fragment starts out with just the ID of the model, though, as we will see in the next section. Our `DisplayViewModel` just asks our `ToDoRepository` for the `ToDoModel` associated with that ID.

The Class Declaration

As noted earlier in the chapter, we are using `androidx.fragment.app.Fragment` as our `Fragment` implementation, so `DisplayFragment` extends from that class.

Fragments should have a public zero-argument constructor. That is because the fragment code — such as the AndroidX fragment code — will create instances of our fragments for us after configuration changes. The only constructor that this code knows how to use is a public zero-argument constructor.

So, in Kotlin, we can skip the constructor in our declaration and just chain to the zero-argument constructor we inherit from `Fragment`:

```
class DisplayFragment : Fragment()
```

(from [FragmentManual/src/main/java/com/commonsware/jetpack/sampler/fragments/DisplayFragment.kt](#))

In Java, we similarly skip any constructor and just extend from `Fragment`:

```
public class DisplayFragment extends Fragment {
```

(from [FragmentManual/src/main/java/com/commonsware/jetpack/samplerj/fragments/DisplayFragment.java](#))

The Factory Function

Frequently, our fragments need to know what they are supposed to do. In the case of `DisplayFragment`, it needs to know what `ToDoModel` to display. Given `DisplayViewModel`, we can get a `ToDoModel` given its ID... but we still need for `DisplayFragment` to get that ID. Since we are using zero-argument constructors, we need another way to get that ID over to the fragment.

With manual fragments, that typically involves a factory function that can create instances of our `DisplayFragment` for us.

In Java, that would be a `static` method that returns a `DisplayFragment`:

```
static DisplayFragment newInstance(String modelId) {
    DisplayFragment result = new DisplayFragment();
    Bundle args = new Bundle();

    args.putString(ARG_MODEL_ID, modelId);
    result.setArguments(args);

    return result;
}
```

(from [FragmentManual/src/main/java/com/commonsware/jetpack/samplerj/fragments/DisplayFragment.java](#))

...while in Kotlin, it would be a function on a companion object:

```
companion object {
    fun newInstance(modelId: String) = DisplayFragment().apply {
        arguments = Bundle().apply { putString(ARG_MODEL_ID, modelId) }
    }
}
```

(from [FragmentManual/src/main/java/com/commonsware/jetpack/samplerj/fragments/DisplayFragment.kt](#))

The Arguments Bundle

These functions take a `modelId` as input and return a `DisplayFragment` as output. What happens in between is that the function attaches an “arguments Bundle” to the fragment, putting that model ID String into the Bundle.

We do this to handle the case where our process is terminated while in the background, but the user returns to our activity and its fragments quickly. The arguments Bundle for the fragment forms part of the saved instance state of our

ADOPTING FRAGMENTS

activity. This way, we can hold onto the model ID and be able to show that model again... in theory.

In reality, we generate fresh model objects with fresh IDs whenever we get a fresh process, so the ID from the old process will be wrong. That is because this example is very fake and does not persist its model data in a database. We will correct that limitation [in a later chapter](#).

Both implementations use a constant named ARG_MODEL_ID as the key to the value for the Bundle:

```
private static final String ARG_MODEL_ID = "modelId";
```

(from [FragmentManual/src/main/java/com/commonsware/jetpack/samplerj/fragments/DisplayFragment.java](#))

```
private const val ARG_MODEL_ID = "modelId"
```

(from [FragmentManual/src/main/java/com/commonsware/jetpack/samplerj/fragments/DisplayFragment.kt](#))

The newInstance() factory function:

- Creates an instance of DisplayFragment
- Creates a Bundle
- Puts the model ID into the Bundle under the ARG_MODEL_ID key
- Attaches the Bundle to the fragment via setArguments()
- Returns the DisplayFragment with the attached Bundle

The onCreateView() Function

A fragment typically has an onCreateView() function, whether on its own or one that it inherits from some superclass. The job of onCreateView() is to return the View that represents the view hierarchy to be managed by this fragment. In the case of DisplayFragment, we want to use the view hierarchy defined by the todo_display layout resource.

We are using data binding in this sample, so our todo_display layout resource gives us a TodoDisplayBinding class, courtesy of the data binding code generator. So, as we did in previous data binding examples, we can use that binding class to inflate() our UI and return the getRoot() View from onCreateView():

```
@Nullable  
@Override  
public View onCreateView(@NonNull LayoutInflater inflater,
```

ADOPTING FRAGMENTS

```
        @Nullable ViewGroup container,
        @Nullable Bundle savedInstanceState) {
binding = TodoDisplayBinding.inflate(inflater, container, false);

return binding.getRoot();
}
```

(from [FragmentManual/src/main/java/com/commonsware/jetpack/samplerj/fragments/DisplayFragment.java](#))

```
override fun onCreateView(
    inflater: LayoutInflater,
    container: ViewGroup?,
    savedInstanceState: Bundle?
) = TodoDisplayBinding.inflate(inflater, container, false)
    .apply { binding = this }
    .root
```

(from [FragmentManual/src/main/java/com/commonsware/jetpack/samplerj/fragments/DisplayFragment.kt](#))

In both cases, we are also holding onto the binding in a binding property for later use.

The onViewCreated() Function

A fragment also typically has an onViewCreated() function. This will be called after onCreateView(), and it is where you configure and populate the widgets that are part of the UI that the fragment is managing.

In our case, we are using data binding, so this is where we can bind our ToDoModel into the layout, to invoke the binding expressions and fill in the widgets.

To do that, we:

- Get our DisplayViewModel from ViewModelProviders(), passing in the fragment itself to of() to get a ViewModelProvider tied to our fragment
- Get the model ID out of the arguments Bundle
- Pass that ID to getModel() on the DisplayViewModel() to retrieve our ToDoModel
- Put that model, and a formatted edition of the createdOn value, into the binding

```
@Override
public void onViewCreated(@NonNull View view,
    @Nullable Bundle savedInstanceState) {
```

ADOPTING FRAGMENTS

```
super.onViewCreated(view, savedInstanceState);

DisplayViewModel vm =
    ViewModelProviders.of(this).get(DisplayViewModel.class);
String modelId = getArguments().getString(ARG_MODEL_ID);

if (modelId == null) {
    throw new IllegalArgumentException("no modelId provided!");
}

ToDoModel model = vm.getModel(modelId);

if (model != null) {
    binding.setModel(model);
    binding.setCreatedOnFormatted(DateUtils.getRelativeDateTimeString(
        getActivity(),
        model.createdOn.getTimeInMillis(), DateUtils.MINUTE_IN_MILLIS,
        DateUtils.WEEK_IN_MILLIS, 0
    ));
}
}
```

(from [FragmentManual/src/main/java/com/commonsware/jetpack/samplerj/fragments/DisplayFragment.java](#))

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)

    val vm: DisplayViewModel = ViewModelProviders.of(this).get()
    val model = vm.getModel(
        arguments?.getString(ARG_MODEL_ID) ?: throw IllegalStateException("no modelId provided!")
    )

    model?.let {
        binding.model = model
        binding.createdOnFormatted = DateUtils.getRelativeDateTimeString(
            activity,
            model.createdOn.timeInMillis, DateUtils.MINUTE_IN_MILLIS,
            DateUtils.WEEK_IN_MILLIS, 0
        )
    }
}
```

(from [FragmentManual/src/main/java/com/commonsware/jetpack/samplerj/fragments/DisplayFragment.kt](#))

Note that we are using a `DateUtils` utility class supplied by Android for formatting our date and time. The big advantage of using `DateUtils` is that this class is aware of the user's settings for how they prefer to see the date and time (e.g., 12- versus 24-hour mode). Specifically, we are using the `getRelativeDateTimeString()` method on `DateUtils`, which will return a value that expresses the creation time relative to now using phrases like “5 minutes ago” or “2 days ago”.

ADOPTING FRAGMENTS

The net result is that after `onViewCreated()` returns, our widgets will have the desired contents.

Of course, so far, we have glossed over the issue of when and how any of this happens. We have this lovely factory function... but something needs to call it. That function creates a perfectly delightful fragment... but something needs to arrange to show it on the screen. We will explore those steps in the upcoming sections.

The ListFragment

We also have a `ListFragment` that is responsible for displaying the list of to-do items. When the user clicks on an item in the list, we want to then show the `DisplayFragment`.

There are several pieces that come together to show this list, with `ListFragment` acting as a central coordinator for all of them.

The Fragment Layout

As we have seen in other samples in this book, this sample app uses a `RecyclerView` for displaying the list of to-do items. Our `todo_roster` layout resource has that `RecyclerView`, and nothing else:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.recyclerview.widget.RecyclerView android:id="@+id/items"
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

(from [FragmentManual/src/main/res/layout/todo_roster.xml](#))

The Row Layout

We also need a layout resource for the rows to appear in the list. That layout — `todo_row` — is a bit more interesting:

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

    <data>

        <variable
```

ADOPTING FRAGMENTS

```
    name="model"
    type="com.commonsware.jetpack.sampler.fragments.ToDoModel" />
</data>

<androidx.constraintlayout.widget.ConstraintLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:background="?android:attr/selectableItemBackground"
    android:clickable="true"
    android:focusable="true">

<CheckBox
    android:id="@+id/checkbox"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginBottom="8dp"
    android:layout_marginEnd="8dp"
    android:layout_marginStart="8dp"
    android:layout_marginTop="8dp"
    android:checked="@{model.isCompleted}"
    android:textSize="@dimen/desc_size"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />

<TextView
    android:id="@+id/desc"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_marginBottom="8dp"
    android:layout_marginEnd="8dp"
    android:layout_marginStart="8dp"
    android:layout_marginTop="8dp"
    android:text="@{model.description}"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toEndOf="@+id/checkbox"
    app:layout_constraintTop_toTopOf="parent" />
</androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```

(from [FragmentManual/src/main/res/layout/todo_row.xml](#))

From a user experience standpoint, we want:

- The user to be able to mark the to-do item as completed by toggling the checkbox (though, in this version of the app, we will not try to save that)

- change)
- The user to be able to view details of the item, via `DisplayFragment`, by clicking elsewhere on the row

Normally, a `CheckBox` is both the actual “checkbox” square plus associated text. So, visually, we could say that each row is just a `CheckBox` widget. But then clicks on the text will wind up toggling the `CheckBox` state, as `CheckBox` interprets all clicks on it equally. In our case, we want to distinguish between clicks on the checkbox and clicks on the rest of the row, so we cannot just use `CheckBox` alone.

To that end, we have both a `CheckBox` and a `TextView` in our row, wrapped in a `ConstraintLayout`, to mimic the presentation of a regular `CheckBox` but not have clicks on the text trigger `CheckBox` state changes.

The root `ConstraintLayout` has its attributes set up such that it represents something that can be clicked upon:

- `android:background="?android:attr/selectableItemBackground"`
- `android:clickable="true"`
- `android:focusable="true"`

We are also using data binding here, to populate the `CheckBox` checked state and the `TextView` text based on the data in a particular `ToDoModel`.

The Viewmodel

Once again, we have a `ViewModel` to hold our data to be displayed. In this case, `ListViewModel` holds the list of items obtained from the `ToDoRepository`:

```
package com.commonsware.jetpack.samplerj.fragments;

import java.util.List;
import androidx.lifecycle.ViewModel;

public class ListViewModel extends ViewModel {
    final List<ToDoModel> items = ToDoRepository.INSTANCE.getItems();
}
```

(from [FragmentManual/src/main/java/com/commonsware/jetpack/samplerj/fragments/ListViewModel.java](#))

```
package com.commonsware.jetpack.sampler.fragments

import androidx.lifecycle.ViewModel
```

ADOPTING FRAGMENTS

```
class ListViewModel : ViewModel() {  
    val items = ToDoRepository.getItems()  
}
```

(from [FragmentManual/src/main/java/com/commonsware/jetpack/samplerj/fragments/ListViewModel.kt](#))

The Adapter and Row Holder

Since we are using a RecyclerView, we need a RecyclerView.Adapter and a RecyclerView.ViewHolder. And we have an interesting requirement: we want our ListFragment to get control when the user clicks on those rows, so we can navigate to the DisplayFragment.

In Kotlin, we can accomplish this by having our adapter and viewholder accept a function type as a parameter, so we can call that function when the user clicks on a row.



You can learn more about function types in the "Functional Programming" chapter of [*Elements of Kotlin*](#)!

In Java, we can create a custom listener interface and use instances of that listener in lieu of the Kotlin function type.

Our RecyclerView.ViewHolder is called ToDoListRowHolder:

```
package com.commonsware.jetpack.samplerj.fragments;  
  
import com.commonsware.jetpack.samplerj.fragments.databinding.TodoRowBinding;  
import androidx.recyclerview.widget.RecyclerView;  
  
class ToDoListRowHolder extends RecyclerView.ViewHolder {  
    interface OnRowClickListener {  
        void onRowClick(ToDoModel model);  
    }  
  
    private final TodoRowBinding binding;  
  
    ToDoListRowHolder(TodoRowBinding binding, OnRowClickListener listener) {  
        super(binding.getRoot());  
  
        this.binding = binding;  
    }  
}
```

ADOPTING FRAGMENTS

```
binding.getRoot().setOnClickListener(v -> {
    if (binding.getModel()!=null) {
        listener.onRowClick(binding.getModel());
    }
});
}

void bind(TodoModel model) {
    binding.setModel(model);
    binding.executePendingBindings();
}
}
```

(from [FragmentManual/src/main/java/com/commonsware/jetpack/samplerj/fragments/ToDoListRowHolder.java](#))

```
package com.commonsware.jetpack.sampler.fragments

import androidx.recyclerview.widget.RecyclerView
import com.commonsware.jetpack.sampler.fragments.databinding.TodoRowBinding

class ToDoListRowHolder(
    private val binding: TodoRowBinding,
    onRowClick: (ToDoModel) -> Unit
) : RecyclerView.ViewHolder(binding.root) {
    init {
        binding.root.setOnClickListener { binding.model?.let { onRowClick(it) } }
    }

    fun bind(model: ToDoModel) {
        binding.model = model
        binding.executePendingBindings()
    }
}
```

(from [FragmentManual/src/main/java/com/commonsware/jetpack/samplerj/fragments/ToDoListRowHolder.kt](#))

We receive a TodoRowBinding object in our constructor — that is the data binding generated binding class for our todo_row layout resource. We also receive the Kotlin function type or the Java OnRowClickListener to use to report when the user clicks on rows.

Then, we:

- Hold onto the binding in a property
- Set up a click listener on the ConstraintLayout (the root of our binding)

ADOPTING FRAGMENTS

and invoke the Kotlin function type or call the Java listener when the user clicks on a row

- Bind a model to the binding in bind() as requested

We can supply the ToDoModel to the function type or listener by asking our binding for its model via getModel(). This way, we can pass the ToDoModel upstream when the user clicks on its row.

Our RecyclerView.Adapter is called ToDoListAdapter:

```
package com.commonsware.jetpack.samplerj.fragments;

import android.view.LayoutInflater;
import android.view.ViewGroup;
import com.commonsware.jetpack.samplerj.fragments.databinding.TodoRowBinding;
import androidx.annotation.NonNull;
import androidx.recyclerview.widget.ListAdapter;

class ToDoListAdapter extends ListAdapter<ToDoModel, ToDoListRowHolder> {
    private final LayoutInflater inflater;
    private final ToDoListRowHolder.OnRowClickListener listener;

    protected ToDoListAdapter(LayoutInflater inflater,
                               ToDoListRowHolder.OnRowClickListener listener) {
        super(ToDoModel.DIFF_CALLBACK);
        this.inflater = inflater;
        this.listener = listener;
    }

    @NonNull
    @Override
    public ToDoListRowHolder onCreateViewHolder(@NonNull ViewGroup parent,
                                                int viewType) {
        return new ToDoListRowHolder(
            TodoRowBinding.inflate(inflater, parent, false), listener);
    }

    @Override
    public void onBindViewHolder(@NonNull ToDoListRowHolder holder,
                               int position) {
        holder.bind(getItem(position));
    }
}
```

(from [FragmentManual/src/main/java/com/commonsware/jetpack/samplerj/fragments/ToDoListAdapter.java](#))

```
package com.commonsware.jetpack.samplerj.fragments
```

ADOPTING FRAGMENTS

```
import android.view.LayoutInflater
import android.view.ViewGroup
import androidx.recyclerview.widget.ListAdapter
import com.commonsware.jetpack.sampler.fragments.databinding.TodoRowBinding

class ToDoListAdapter(private val inflator: LayoutInflater,
    private val onRowClick: (ToDoModel) -> Unit) :
    ListAdapter<ToDoModel, ToDoListRowHolder>(ToDoModelDiffCallback) {
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int) =
        ToDoListRowHolder(TodoRowBinding.inflate(inflator, parent, false), onRowClick)

    override fun onBindViewHolder(holder: ToDoListRowHolder, position: Int) {
        holder.bind(getItem(position))
    }
}
```

(from [FragmentManual/src/main/java/com/commonsware/jetpack/sampler/fragments/ToDoListAdapter.kt](#))

ToDoListAdapter is a ListAdapter, and so we need our DiffUtil.ItemCallback to help identify changes in our data model. That is implemented on ToDoModel, as our model class should know how to compare instances of itself:

```
static final DiffUtil.ItemCallback<ToDoModel> DIFF_CALLBACK =
    new DiffUtil.ItemCallback<ToDoModel>() {
        @Override
        public boolean areItemsTheSame(@NonNull ToDoModel oldItem,
                                       @NonNull ToDoModel newItem) {
            return oldItem == newItem;
        }

        @Override
        public boolean areContentsTheSame(@NonNull ToDoModel oldItem,
                                         @NonNull ToDoModel newItem) {
            return oldItem.isCompleted == newItem.isCompleted &&
                oldItem.description.equals(newItem.description);
        }
    };
```

(from [FragmentManual/src/main/java/com/commonsware/jetpack/samplerj/fragments/ToDoModel.java](#))

```
object ToDoModelDiffCallback : DiffUtil.ItemCallback<ToDoModel>() {
    override fun areItemsTheSame(oldItem: ToDoModel, newItem: ToDoModel) =
        oldItem === newItem

    override fun areContentsTheSame(oldItem: ToDoModel, newItem: ToDoModel) =
        oldItem.isCompleted == newItem.isCompleted &&
        oldItem.description == newItem.description
}
```

(from [FragmentManual/src/main/java/com/commonsware/jetpack/sampler/fragments/ToDoModel.kt](#))

ADOPTING FRAGMENTS

`ToDoListAdapter` also accepts our function type or listener in its constructor, to pass to the constructors of each of the `ToDoListRowHolder` instances.

Beyond that, `ToDoListAdapter` is a fairly simple `ListAdapter` implementation, creating instances of `ToDoListRowHolder` in `onCreateViewHolder()` and binding `ToDoModel` instances in `onBindViewHolder()`.

The Fragment

With all that done, our `ListFragment` has little to do, other than connect the remaining pieces:

```
package com.commonsware.jetpack.samplerj.fragments;

import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import androidx.annotation.NonNull;
import androidx.annotation.Nullable;
import androidx.fragment.app.Fragment;
import androidx.lifecycle.ViewModelProviders;
import androidx.recyclerview.widget.LinearLayoutManager;
import androidx.recyclerview.widget.RecyclerView;

public class ListFragment extends Fragment {
    @Nullable
    @Override
    public View onCreateView(@NonNull LayoutInflater inflater,
                           @Nullable ViewGroup container,
                           @Nullable Bundle savedInstanceState) {
        return inflater.inflate(R.layout.todo_roster, container, false);
    }

    @Override
    public void onViewCreated(@NonNull View view,
                           @Nullable Bundle savedInstanceState) {
        super.onViewCreated(view, savedInstanceState);

        ListViewModel vm = ViewModelProviders.of(this).get(ListViewModel.class);
        RecyclerView rv = view.findViewById(R.id.items);

        rv.setLayoutManager(new LinearLayoutManager(getContext()));

        ToDoListAdapter adapter = new ToDoListAdapter(LayoutInflater.from(getContext()),
            this::navTo);
    }
}
```

ADOPTING FRAGMENTS

```
        adapter.submitList(vm.items);
        rv.setAdapter(adapter);
    }

    private void navTo(TodoModel model) {
        getFragmentManager().beginTransaction()
            .replace(android.R.id.content, DisplayFragment.newInstance(model.id))
            .addToBackStack(null)
            .commit();
    }
}
```

(from [FragmentManual/src/main/java/com/commonsware/jetpack/samplerj/fragments/ListFragment.java](#))

```
package com.commonsware.jetpack.sampler.fragments

import android.os.Bundle
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import androidx.fragment.app.Fragment
import androidx.fragment.app.transaction
import androidx.lifecycle.ViewModelProviders
import androidx.lifecycle.get
import androidx.recyclerview.widget.LinearLayoutManager
import kotlinx.android.synthetic.main.todo_roster.view.*

class ListFragment : Fragment() {

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View = inflater.inflate(R.layout.todo_roster, container, false)

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        val vm: ListViewModel = ViewModelProviders.of(this).get()
        val rv = getView()!!.items

        rv.layoutManager = LinearLayoutManager(context)

        val adapter = TodoListAdapter(layoutInflater) {
            navTo(it)
        }
    }
}
```

ADOPTING FRAGMENTS

```
    rv.adapter = adapter.apply { submitList(vm.items) }
```

```
}
```

```
private fun navTo(model: ToDoModel) {
    fragmentManager?.transaction {
        replace(android.R.id.content, DisplayFragment.newInstance(model.id))
        addToBackStack(null)
    }
}
```

(from [FragmentManual/src/main/java/com/commonsware/jetpack/sampler/fragments/ListFragment.kt](#))

As with `DisplayFragment`, our `onCreateView()` function sets up the UI to be managed by this fragment. That is the `RecyclerView` defined in the `todo_roster` layout resource. That layout resource does not use data binding, so we can use `inflate()` on `LayoutInflater` directly to load that layout resource and create the `RecyclerView`.

As with `DisplayFragment`, our `onViewCreated()` function configures the UI that we inflated in `onCreateView()`. Here we:

- Get our `ViewModel` from `ViewModelProviders`
- Create a `LinearLayoutManager` and attach it to the `RecyclerView`
- Create a `ToDoListAdapter` and attach it to the `RecyclerView`
- Get our list of `ToDoModel` objects from the `ViewModel` and supply those to the `ToDoListAdapter` via `submitList()`

When we create the `ToDoListAdapter`, we need to supply our `onItemClick` function type or listener implementation. In Java, we use a Java 8 method reference to have a generated listener call our `navTo()` method. In Kotlin, we use a lambda expression to call our `navTo()` function.

So, when the user clicks a row, whether we are in Java or Kotlin, `navTo()` is called. There, we need to arrange to show the `DisplayFragment`.

The FragmentTransaction

`navTo()` uses a `FragmentTransaction` to display the `DisplayFragment`. The actual work done is the same in Java and Kotlin. However, the Kotlin edition of the project has added `androidx.fragment:fragment-ktx` as a dependency. This Android KTX library offers some simpler syntax for Kotlin's use of `FragmentTransaction`.

ADOPTING FRAGMENTS

Java

The Java edition of navTo() has a fairly classic recipe for performing a FragmentTransaction:

```
private void navTo(TodoModel model) {
    getFragmentManager().beginTransaction()
        .replace(android.R.id.content, DisplayFragment.newInstance(model.id))
        .addToBackStack(null)
        .commit();
}
```

(from [FragmentManual/src/main/java/com/commonsware/jetpack/samplerj/fragments/ListFragment.java](#))

First, to get a FragmentTransaction, we need a FragmentManager. A fragment can get its own FragmentManager by calling getFragmentManager().

Next, to get a FragmentTransaction for use, we can call beginTransaction() on the FragmentManager. This returns a FragmentTransaction ready for us to configure and commit.

In our case, we do two things to configure the FragmentTransaction:

1. We call replace() to say “please replace any fragment in the android.R.id.content container with this new fragment”. The new fragment is our DisplayFragment, created using the factory method. The container is android.R.id.content, which is the framework’s ID for the container for an activity’s UI. When we call setContentView() on an activity, the “content view” winds up as a child of the android.R.id.content container.
2. We call addToBackStack(null), indicating that when the user clicks the BACK button, we want the FragmentTransaction to be rolled back. In our case, that would revert the replace() call, which would cause android.R.id.content to hold whatever fragment preceded the DisplayFragment. As we will see shortly, the ListFragment itself is going in that container, so reverting that transaction will cause the ListFragment to be displayed again.

Finally, once our FragmentTransaction is configured, we call commit() to say “make it so”. Shortly, the DisplayFragment will be shown on the screen, replacing the ListFragment.

ADOPTING FRAGMENTS

Kotlin

The Kotlin code does the same work, just with slightly different syntax:

```
private fun navTo(model: ToDoModel) {
    fragmentManager?.transaction {
        replace(android.R.id.content, DisplayFragment.newInstance(model.id))
        addToBackStack(null)
    }
}
```

(from [FragmentManual/src/main/java/com/commonsware/jetpack/sampler/fragments/ListFragment.kt](#))

Part of that syntax change is just ordinary Kotlin, where `getFragmentManager()` can be replaced by reading the `fragmentManager` pseudo-property.

Part of that syntax change comes from Android KTX, in the form of that `androidx.fragment:fragment-ktx` library. It adds an extension function to `FragmentManager`, named `transaction()`. The `transaction()` function:

- Accepts a function type as a parameter, typically in the form of a lambda expression as shown here
- Begins a `FragmentTransaction`
- Invokes that function type, setting the `FragmentTransaction` to be the current object (i.e., `this`), so calls like `replace()` and `addToBackStack()` can be called directly
- Commits that `FragmentTransaction` once the function type returns

The Activity

All that remains is for our activity to set up the `ListFragment` to be shown. That too is handled via a `FragmentTransaction`, much like how we navigated from `ListFragment` to `DetailFragment`. However, there is one wrinkle: we only execute the `FragmentTransaction` if our `savedInstanceState` is `null`:

```
package com.commonsware.jetpack.samplerj.fragments;

import android.os.Bundle;
import androidx.annotation.Nullable;
import androidx.appcompat.app.AppCompatActivity;

public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(@Nullable Bundle savedInstanceState) {
```

ADOPTING FRAGMENTS

```
super.onCreate(savedInstanceState);

if (savedInstanceState == null) {
    getSupportFragmentManager().beginTransaction()
        .add(android.R.id.content, new ListFragment())
        .commit();
}
}
```

(from [FragmentManual/src/main/java/com/commonsware/jetpack/samplerj/fragments/MainActivity.java](#))

```
package com.commonsware.jetpack.sampler.fragments

import android.os.Bundle
import androidx.appcompat.app.AppCompatActivity
import androidx.fragment.app.transaction

class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        if ( savedInstanceState == null ) {
            supportFragmentManager.beginTransaction {
                add(android.R.id.content, ListFragment())
            }
        }
    }
}
```

(from [FragmentManual/src/main/java/com/commonsware/jetpack/sampler/fragments/MainActivity.kt](#))

When the device undergoes a configuration change, Android will automatically destroy and recreate the activity *and its fragments*. As a result, in the re-created activity, by the time `onCreate()` is called, any displayed fragments will already exist. In that case, we do not need to create them ourselves. So, to distinguish between the first time the activity is created and when the activity is re-created after a configuration change, we see if the passed-in state Bundle is `null`. If it is, then our fragment should not yet exist, so we create one and `add()` it to the container via a `FragmentTransaction`. If the Bundle is not `null`, though, then the fragment should already exist, so we can skip the step for creating and adding it.

Another approach that we could use is to see if the container already has a fragment in it, using code like:

ADOPTING FRAGMENTS

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)

    if (supportFragmentManager.findFragmentById(android.R.id.content) == null) {
        supportFragmentManager.beginTransaction {
            add(android.R.id.content, ListFragment())
        }
    }
}
```

Those approaches are subtly different. The original asks “did the activity get re-created?”. The revision asks “do we have a fragment in the container already?”. In the context of this app, those will be equivalent, so either approach can work. In more elaborate apps, where fragments might be coming and going with greater frequency and complexity, those approaches may yield different results, and you will want to choose the one that better matches the actual need of your code and situation.

The Recap

So, we have two fragments, each with their own viewmodel. The `MainActivity` sets up the first fragment (`ListFragment`). That fragment renders its own UI and switches to a different fragment (`DetailsFragment`) based on user input. The second fragment handles *its* own UI, independently from the first fragment, other than for data explicitly passed between them (the to-do item’s ID).

We could have implemented the same thing using two activities, instead of a single activity and two fragments. A lot of classic Android app development would have taken that approach. The Jetpack recommendations are to use fragments where possible and minimize your number of activities, for greater flexibility in implementing your UI.

The Fragment Lifecycle Methods

Fragments have lifecycle methods, just like activities do. In fact, they support most of the same lifecycle methods as activities:

- `onCreate()`
- `onStart()` (but not `onRestart()`)
- `onResume()`
- `onPause()`
- `onStop()`

ADOPTING FRAGMENTS

- `onDestroy()`

By and large, the same rules apply for fragments as do for activities with respect to these lifecycle methods (e.g., `onDestroy()` may not be called).

In addition to those and the `onCreateView()` method we examined earlier in this chapter, there are other lifecycle methods that you can elect to override if you so choose.

`onAttach()` will be called first, even before `onCreate()`, letting you know that your fragment has been attached to an activity. You are passed the `Activity` that will host your fragment.

`onViewCreated()` will be called after `onCreateView()`. This is particularly useful if you are inheriting the `onCreateView()` implementation but need to configure the resulting views. For example, you need to attach an adapter to a `ListFragment` — a common place to do that is in `onViewCreated()`, as you know that the `ListView` is set up at that time.

`onActivityCreated()` will be called to indicate that the activity's `onCreate()` has completed. If there is something that you need to initialize in your fragment that depends upon the activity's `onCreate()` having completed its work, you can use `onActivityCreated()` for that initialization work.

`onDestroyView()` is called before `onDestroy()`. This is the counterpart to `onCreateView()` where you set up your UI. If there are things that you need to clean up specific to your UI, you might put that logic in `onDestroyView()`.

`onDetach()` is called after `onDestroy()`, to let you know that your fragment has been disassociated from its hosting activity.

ADOPTING FRAGMENTS

Google offers this diagram, depicting the order in which these events will occur:

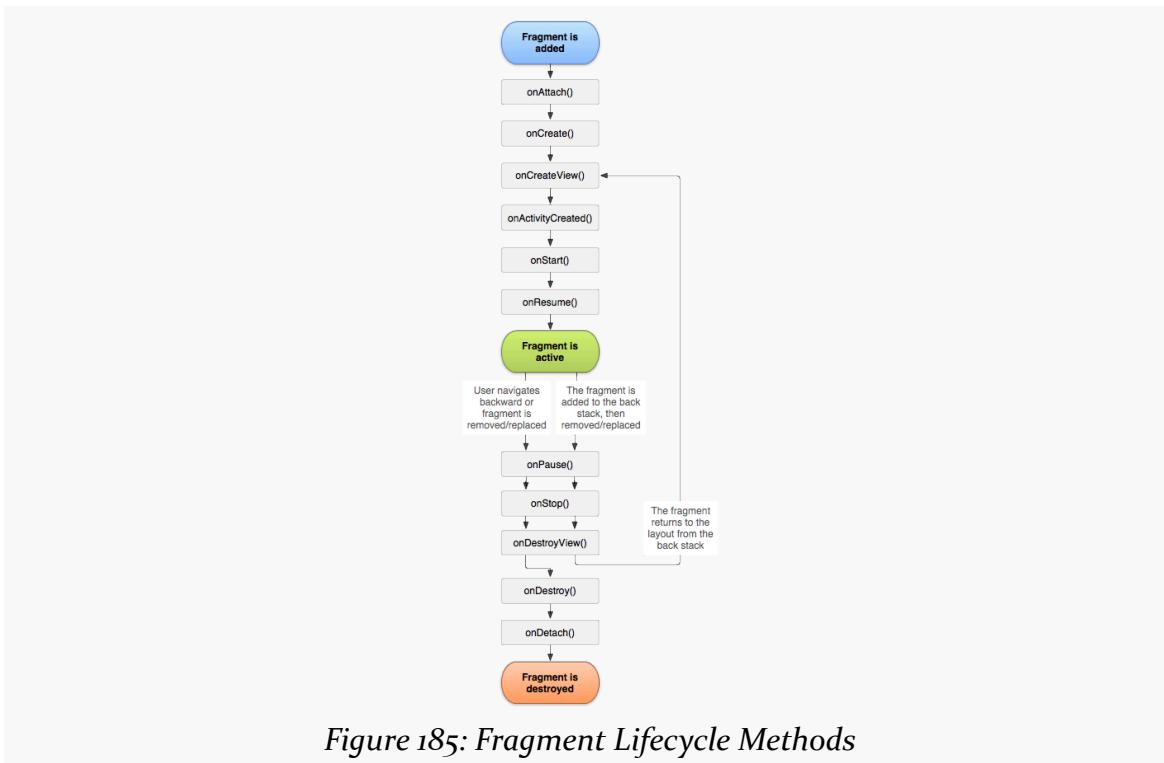


Figure 185: Fragment Lifecycle Methods

(the above image is reproduced from work created and [shared by the Android Open Source Project](#) and used according to terms described in [the Creative Commons 2.5 Attribution License](#))

onAttach() Versus onAttach()

If you set your project to have a `compileSdkVersion` of `23` or higher, and you attempt to override `onAttach()`, you may get a deprecation warning:

```
import android.app.Activity
import androidx.fragment.app.Fragment

class TestFragment : Fragment() {
    override fun onAttach(activity: Activity?) {
        super.onAttach(activity)
    }
}
```

Figure 186: Android Studio, Showing Deprecated onAttach()

That is because there are *two* versions of `onAttach()` (and `onDetach()`) starting with API Level `23`. One takes an `Activity` as a parameter, and the other takes a `Context` as a parameter. The one taking the `Activity` as a parameter has been deprecated, and deprecated things show up with strikethrough applied to their names in Android Studio.

The roles of `onAttach()` and `onDetach()` are the same with either parameter: let you know when the fragment has been attached to or detached from its host. However, now, the host could be anything that extends `Context`, not merely an `Activity`.

On API Level `22` and below, though, only the `Activity` flavor of `onAttach()` and `onDetach()` exists. This leads to a conundrum, as you try to determine exactly how to handle this for your app.

On the whole, if your `minSdkVersion` is below `23`, overriding just `onAttach(Activity)` is your best route. It will work on all Android devices that support fragments. Overriding only `onAttach(Context)` will not work, as older devices will ignore it (despite `Activity` being a subclass of `Context`). You could override both methods, but on API Level `23+` devices, both flavors will be called, which may or may not be a good idea for your Fragment subclass.

Context Anti-Pattern: Assuming Certain Types

Fragment has a `getContext()` method that you can call, to retrieve a `Context`, should

ADOPTING FRAGMENTS

you need one for retrieving resources, etc.

It is likely that this Context is the Activity that hosts the fragment. However you should not *assume* that this Context is the Activity that hosts the fragment. Google has been preparing for cases where fragments might be owned by something else, though what that “something else” is has yet to be determined.

If the documentation for the source of a Context stipulates that it is a particular type of Context, it should be safe to treat it as such, downcasting it as needed (e.g., `context as Activity` in Kotlin). Otherwise, limit your use of that Context to things that are available on Context and make no assumptions about its concrete type. That concrete type might vary from version to version of Android, or even between devices on the same Android version (due to manufacturer tweaks to Android).

Navigating Your App

In the beginning, if we wanted multiple screens of content, we would use multiple activities. To switch from screen to screen, we would use `startActivity()`.

As Android evolved, many developers started to use fragments. To switch from screen to screen, we would execute a `FragmentTransaction`.

All of that works and you are welcome to use it.

However, as part of Jetpack, Google is pushing a new Navigation component. This serves as a replacement for many direct uses of `FragmentTransaction` and `startActivity()`, while still allowing you to navigate from activities and fragments to other activities and fragments.

The Navigation component is very new... so new that, as of mid-February 2019, it is not yet fully released, as we use `1.0.0-rc01`. That should be cleared up by mid-March or so, when a `1.0.0` should be available, as Google is pretty good about advancing through the release candidate (`rc`) phase fairly quickly. However, even when `1.0.0` ships, this is still a fairly new capability in Android, and it remains to be seen whether it will “stand the test of time” or be dropped.

In this chapter, though, we will be optimistic about Navigation’s chances and we will see how to use it in your Android apps.

What We Get from the Navigation Component

The Navigation component does not add any new capabilities to Android. It can’t, after all — it comes from a library, and anything that can be done by Library A could be done by Library B or Hand-Written-Code C. Anything you can do with the

NAVIGATING YOUR APP

Navigation component you can do “the old-fashioned way” using `startActivity()`, `FragmentTransaction`, and related techniques.

So... why bother with the Navigation component?

Uniform API/Isolation of Details

When we call `startActivity()`, we have to provide an Intent that identifies the activity that we want to start. When we commit a `FragmentTransaction`, we have to configure that object with an instance of the fragment to be displayed. As a result:

- Our code is tied to a specific type of navigation (activity or fragment)
- Our code knows implementation details of the destination (the Java/Kotlin class that is the target)

A big thing in computer programming is “separation of concerns”. We want one chunk of code to know as little as possible about the next chunk of code, in case we need to radically change the implementation of either chunk. With classic manual navigation, our activities and fragments know more about the other activities and fragments than is ideal.

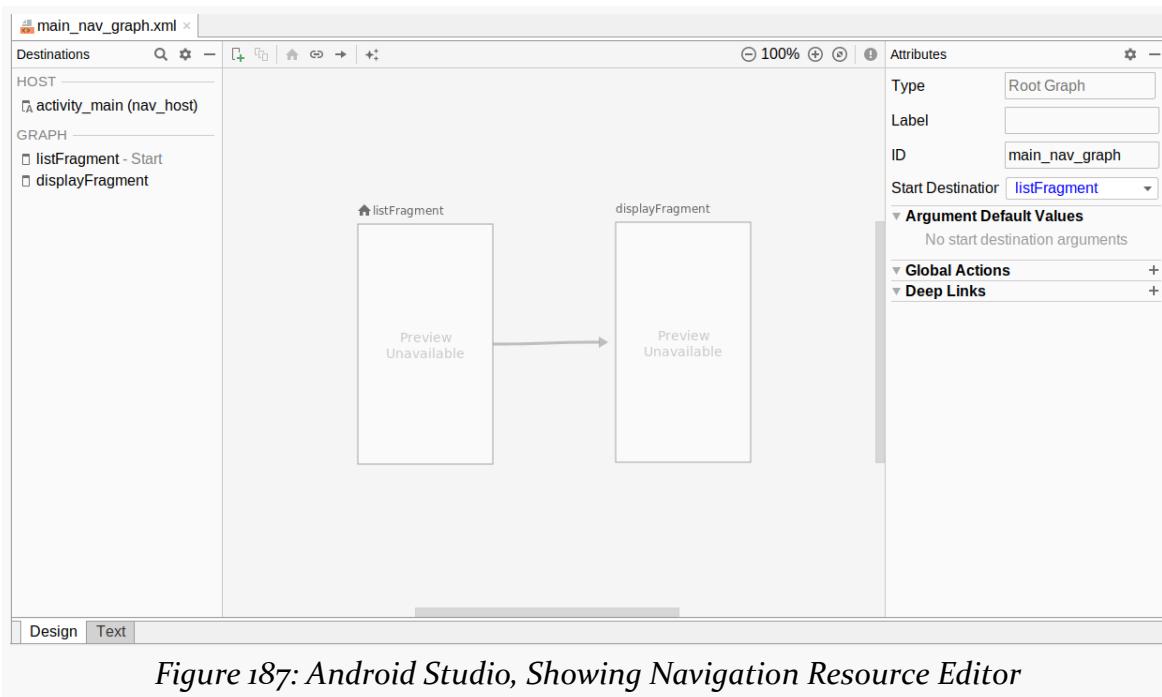
With the Navigation component, we start to get away from that.

One key aspect is that all navigation is the same with the Navigation component. When we want to switch to a different screen, we use the same API regardless of whether that screen is implemented by an activity or a fragment. Hence, we want to switch implementations someday (e.g., convert an activity to a fragment), the code that navigates to that changed destination may not itself need to change.

Another aspect is that the details of how different screens are implemented is contained in a new type of resource (navigation). The resource contains details about classes and whether they represent activities or fragments. The resource uses IDs — akin to widget IDs — to identify screens and how we get there. Our Java/Kotlin code uses those IDs to tell the Navigation component where we want to go. So, now not only do we use one API for navigating to both activities and fragments, but the details of *which* activities and fragments are hidden behind an ID. If we want to swap out one implementation of a screen with another, those navigating to that screen should not need to change.

Graphical Representation of Flows

The navigation resource type, like most of Android's resources, is an XML file with a particular set of elements. And, like layout and menu resources, Android Studio offers a graphical editor for navigation resources, allowing you to connect screens using a drag-and-drop interface:



We will explore the use of this editor later in this chapter.

“Safe Args”

In both [the chapter on activities](#) and [the chapter on fragments](#), we saw how we can pass data from one screen to another. In the case of activities, that is in the form of Intent extras. In the case of fragments, that is in the form of the arguments Bundle. As it turns out, the extras in an Intent are contained in a Bundle, so in both forms of manual navigation, your data is passed via a Bundle.

The problem with a Bundle is that while a Bundle supports a range of data types for values, any key could have any of those data types. This can lead to problems, if the sender specifies a String where the recipient is expecting an Int, or something like that. It is as if we got rid of the strong typing available in Java/Kotlin and just held

onto all of our data in Map objects.

An add-on to the Navigation component helps with this by offering “Safe Args”. Part of the details that we can put into the navigation resource is what data is expected by any given screen. Then, Android Studio will code-generate:

- A class to help code wishing to navigate to that screen provide the data, with the expected data types
- A class to help the implementation of the screen retrieve that data, once again with the expected data types

“Under the covers”, those generated classes work with Intent extras and the arguments Bundle, but those implementation details are hidden from us. This gives us type-safe navigation, so we do not accidentally provide data of the wrong type.

App Bar Up Integration

You may have noticed that in some apps, in some cases, a leftward-pointing arrow appears in the app bar:



Figure 188: Up Button in App Bar

This button allows the user to navigate “up” in the app. In most cases, this is the same as the user pressing the system BACK button in the navigation bar. Sometimes, in complicated apps with complicated navigation, up might lead somewhere else that represents “up” in some hierarchy of content.

The Navigation component allows you to tie your app bar (e.g., a Toolbar) into the navigation flow, such that the “up” button is shown when the Navigation component deems it to be appropriate.

Simpler Support of Advanced Features

Those reasons may be enough to warrant experimenting with the Navigation

component.

For veteran Android developers, the Navigation component also helps to simplify the use of a variety of advanced features, such as:

- “Deep links”, to help Google’s search engine and other Web content to launch an app (via a link) and drive straight to some inner portion of that app, instead of just opening the launcher activity
- Transitions, a means by which we can give the appearance of a widget from one screen “moving” to another screen, such as a thumbnail image in a list “zooming” into a larger image when the user clicks on the list item and goes to a detail screen

Elements of Navigation

There are four main pieces of the Navigation component:

- The navigation resources
- The Android Studio editor for those resources
- The NavHostFragment for intra-activity navigation between fragments
- The NavController that lets you navigate at runtime

Navigation Resources

Google only infrequently creates a new type of resource, and almost never do they create one that is not handled directly by the OS.

For the Navigation component, they did just that.

You can have a `res/navigation/` directory in your module that holds navigation resources. Those are XML files with a root `<navigation>` element and a series of child elements that describe the navigation graph of your app.

Principally, the Navigation component handles navigation between fragments within an activity. Hence, the primary children of the `<navigation>` element are `<fragment>` elements, identifying a particular fragment that is part of the navigation graph. However, if you want to navigate to a different activity — perhaps one that has its own navigation graph — you can have child `<activity>` elements off of `<navigation>` that identify those.

NAVIGATING YOUR APP

The `<fragment>` and `<activity>` elements — otherwise known as the “destinations” — have a number of possible child elements, of which two are the most important:

- `<action>` indicates a destination that is supported from this screen; the Navigation component helps you navigate based on these actions
- `<argument>` indicates a particular bit of data that this screen is expecting as input

The root `<navigation>` element will have an `app:startDestination` attribute that identifies which `<fragment>` should be displayed when this navigation graph is used by an activity.

We will explore all of these in greater detail when we work through a sample app [a bit later in the chapter](#).

Navigation Resource Editor

You are welcome to maintain navigation resources manually by editing the XML. Or, you can use Android Studio’s dedicated editor for those resources.

As with layout and menu resources, the navigation editor has “Design” and “Text” sub-tabs. The XML is available in the “Text” sub-tab, while the “Design” sub-tab has the visual editor.

Destinations

The main “preview” area shows the existing destinations in the navigation graph:

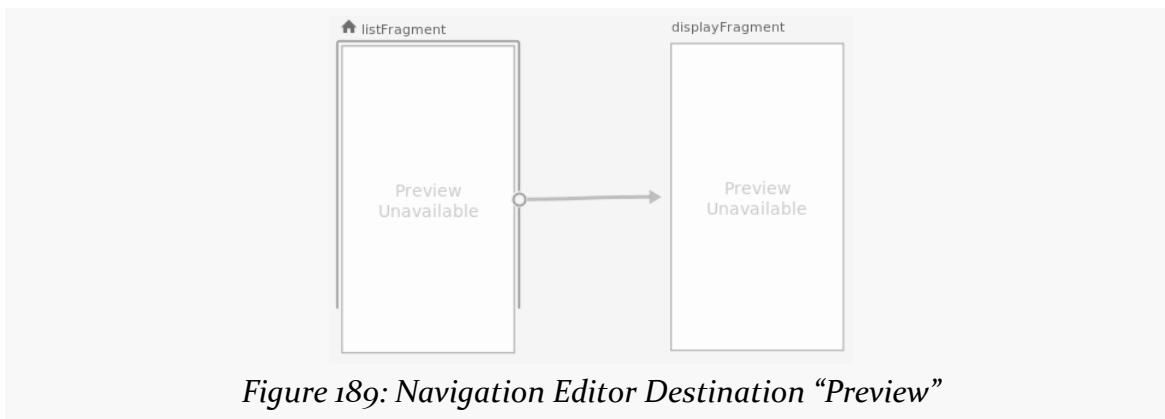


Figure 189: Navigation Editor Destination “Preview”

NAVIGATING YOUR APP

To add a new destination, you can click the left-most toolbar button in the preview area:



Figure 190: Navigation Editor Preview Area Toolbar

This will bring up a list of all of the activities and fragments in your module, along with options for a “placeholder” and “create new destination”. The latter brings up a new-fragment wizard to help you create a new fragment.

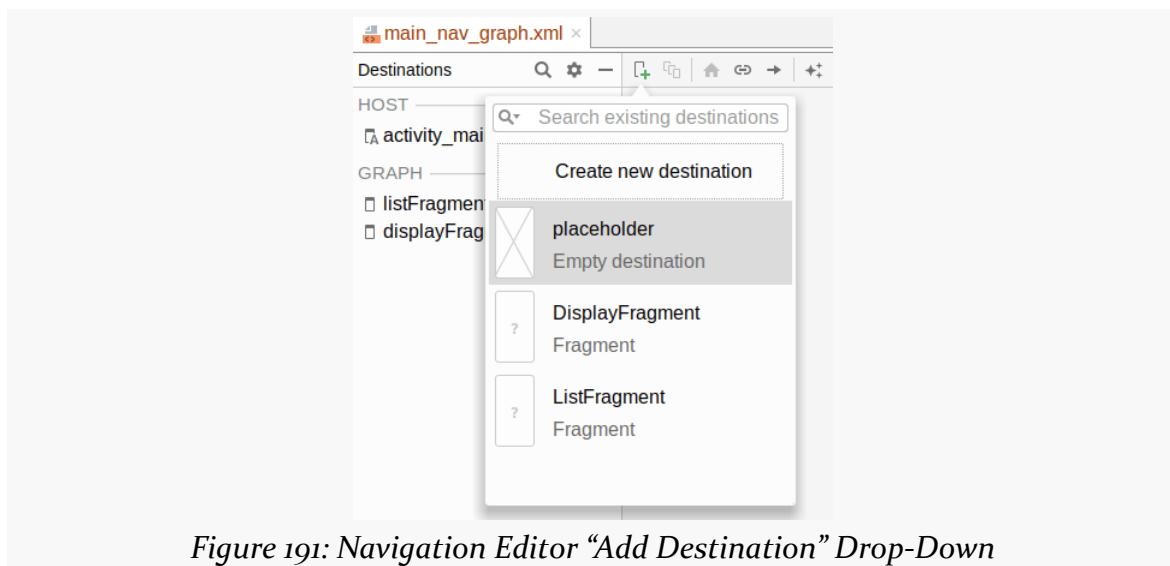


Figure 191: Navigation Editor “Add Destination” Drop-Down

If you click on a particular destination, the “Attributes” pane will show you key information about that destination, much of which is editable:

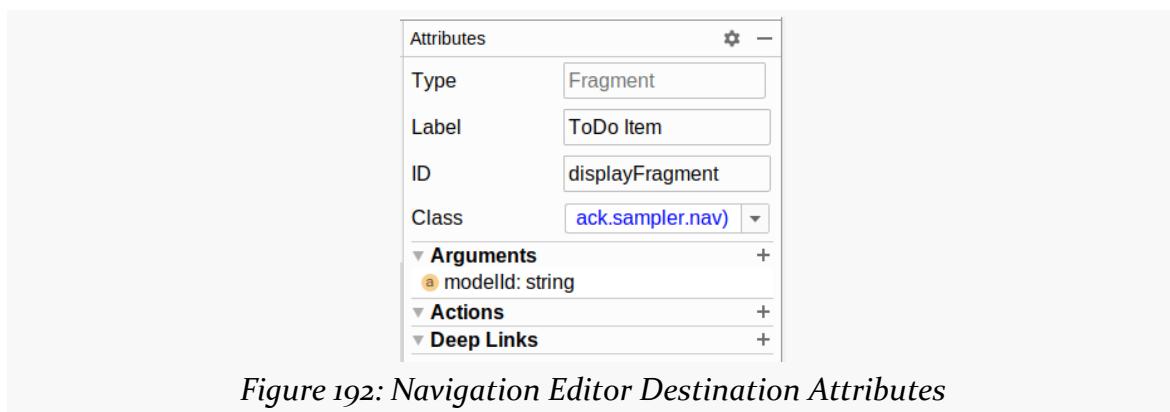


Figure 192: Navigation Editor Destination Attributes

NAVIGATING YOUR APP

Of particular note:

- “ID” holds the unique identifier of this destination. This uses the same @+id/ syntax as is used in layout and menu resources, though in the editor we drop off that prefix. So, while the editor may show displayFragment, the real identifier in the XML is @+id/displayFragment.
- “Class” is the Java/Kotlin class that implements this fragment or activity
- “Label” will be used in places where the Navigation component shows a user-readable name for the destination, such as in the app bar (e.g., a Toolbar)
- “Arguments” contains a list of the arguments that this particular destination expects to receive as input. The + button allows you to add a new argument:



Figure 193: Navigation Editor “Add Argument” Dialog

In the preview area toolbar, you can click the “home” icon to designate the selected destination as being the start destination for this navigation graph.

Actions

Arrows connecting destinations in the preview area represent actions. Reminiscent of constraining widgets in a ConstraintLayout, you can drag a circle on the right edge of a destination to a corresponding circle on the left edge of another destination to create an action. This indicates that you want to be able to navigate from the first destination to the second.

NAVIGATING YOUR APP

Clicking on an arrow highlights it in blue and brings up its own set of attributes:



Figure 194: Navigation Editor Action Attributes

Of particular note:

- Actions have their own “ID” value, once again using the same `@+id/` syntax that we have seen elsewhere
- “Destination” provides the ID of the destination that is the target of this action (i.e., the destination that will be navigated to if this action is performed)
- “Argument Default Values” allows you to optionally supply hard-coded values for arguments required by the chosen destination, instead of having to provide them in Java/Kotlin code

NavHostFragment

As noted earlier, the primary focus of the Navigation component is to navigate between fragments within an activity. This implies that the Navigation component knows where those fragments should go.

That is in the form of a `NavHostFragment`. This is a fragment that shows other fragments, the ones specified in your navigation graph. You tell it what navigation graph it should use, and it will be responsible for:

NAVIGATING YOUR APP

- Showing your start destination, and
- Switching to another fragment destination, when your code asks for that at runtime

NavController

Switching to another destination in your navigation graph at runtime involves a NavController. This will be associated with the activity or fragment that is hosting the NavHostFragment. There are utility functions for getting the NavController, and from there you can ask it to move forwards or backwards in the navigation graph. “Forwards” means “go to this destination”; “backwards” means “go back to where we came from to get to our current destination”.

A Navigation-ized To-Do List

The FragmentNav sample module in the [Sampler](#) and [SamplerJ](#) projects mostly is a clone of [last chapter's](#) FragmentManual module. In this edition of the app, though, we use the Navigation component to get between the ListFragment and the DisplayFragment. We also switch to using a Toolbar here for our app bar, so we can get the up button added for us when we are on the DisplayFragment.

The Dependencies

The Navigation component is relatively complex in terms of dependencies, as we have variations for Java and Kotlin plus additional setup for the Safe Args feature.

The Basics

There are two main dependencies that most apps will use to employ the Navigation component:

1. `android.arch.navigation:navigation-fragment` provides the core support for navigation, particularly using fragments
2. `android.arch.navigation:navigation-ui` offers the integration with the Toolbar and other forms of app bar

These dependencies are closely coupled and generally will need to have the same version. So, the overall Sampler/SamplerJ projects define a `nav_version` constant that contains the version to use:

NAVIGATING YOUR APP

```
buildscript {  
    ext.nav_version = "2.0.0"  
  
    repositories {  
        google()  
        jcenter()  
    }  
  
    dependencies {  
        classpath 'com.android.tools.build:gradle:3.5.0'  
        classpath "androidx.navigation:navigation-safe-args-gradle-plugin:$nav_version"  
    }  
}
```

(from [build.gradle](#))

That way, when we add the dependencies in our FragmentNav module, we can reference that nav_version constant and ensure that everything stays synchronized:

```
dependencies {  
    implementation 'androidx.appcompat:appcompat:1.0.2'  
    implementation 'androidx.recyclerview:recyclerview:1.0.0'  
    implementation 'androidx.lifecycle:lifecycle-extensions:2.0.0'  
    implementation 'androidx.constraintlayout:constraintlayout:1.1.3'  
    implementation "androidx.navigation:navigation-fragment:$nav_version"  
    implementation "androidx.navigation:navigation-ui:$nav_version"  
}
```

(from [FragmentNav/build.gradle](#))

Groovy — the language used for these Gradle scripts — supports string interpolation, using the same basic syntax as does Kotlin (\$name to add the value of name to the string). As a result, values like android.arch.navigation:navigation-fragment:\$nav_version get the \$nav_version portion replaced by whatever the value of nav_version is.

The KTX Bits

The dependencies closure shown above is from the SamplerJ project. Both Java and Kotlin projects are welcome to use those dependencies. Kotlin projects, though, are more likely to use the Android KTX variants of the dependencies:

```
dependencies {  
    implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlin_version"  
    implementation 'androidx.appcompat:appcompat:1.0.2'  
    implementation 'androidx.recyclerview:recyclerview:1.0.0'  
    implementation 'androidx.lifecycle:lifecycle-viewmodel-ktx:2.0.0'  
    implementation 'androidx.lifecycle:lifecycle-extensions:2.0.0'
```

NAVIGATING YOUR APP

```
implementation 'androidx.constraintlayout:constraintlayout:1.1.3'
implementation "androidx.navigation:navigation-fragment-ktx:$nav_version"
implementation "androidx.navigation:navigation-ui-ktx:$nav_version"
}
```

(from [FragmentNav/build.gradle](#))

These, like the rest of the Android KTX libraries, add some extension functions and similar Kotlin features that make using the Navigation component easier in Kotlin.

The Safe Args Code Generator

For basic use of the Navigation component, those are all that you need. If you want to use the Safe Args feature for type-safe data exchange as part of navigation, you have some additional setup to perform.

Let's review one of the earlier code snippets, where we defined nav_version:

```
buildscript {
    ext.nav_version = "2.0.0"

    repositories {
        google()
        jcenter()
    }

    dependencies {
        classpath 'com.android.tools.build:gradle:3.5.0'
        classpath "androidx.navigation:navigation-safe-args-gradle-plugin:$nav_version"
    }
}
```

(from [build.gradle](#))

If you look closely, this also adds another entry to the buildscript set of dependencies. Specifically, this adds a Gradle plugin for Safe Args. dependencies in a module usually refer to code added to that module for runtime use; dependencies in the buildscript closure usually refer to Gradle plugins that will be used by modules.

Simply having the classpath entry is insufficient, though. Modules need to opt into specific plugins offered by the library indicated by the classpath entry. So, our Java modules' build.gradle files have two apply plugin statements at the top: one for Android apps and one for Safe Args:

```
apply plugin: 'com.android.application'
apply plugin: 'androidx.navigation.safeargs'
```

NAVIGATING YOUR APP

(from [FragmentNav/build.gradle](#))

Kotlin modules will have additional plugins for Kotlin support, along with a Kotlin-specific version of the Safe Args plugin:

```
apply plugin: 'com.android.application'
apply plugin: 'kotlin-android'
apply plugin: 'kotlin-android-extensions'
apply plugin: 'androidx.navigation.safeargs.kotlin'
```

(from [FragmentNav/build.gradle](#))

The Navigation Resource

The module has a `res/navigation/` directory for navigation resources. In there, there is a `main_nav_graph.xml` resource that represents our tiny navigation graph:

```
<?xml version="1.0" encoding="utf-8"?>
<navigation android:id="@+id/main_nav_graph"
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    app:startDestination="@+id/listFragment">

    <fragment
        android:id="@+id/listFragment"
        android:name="com.commonsware.jetpack.sampler.nav.ListFragment"
        android:label="ToDo Items">
        <action
            android:id="@+id/displayModel"
            app:destination="@+id/displayFragment" />
    </fragment>
    <fragment
        android:id="@+id/displayFragment"
        android:name="com.commonsware.jetpack.sampler.nav.DisplayFragment"
        android:label="ToDo Item">
        <argument
            android:name="modelId"
            app:argType="string" />
    </fragment>
</navigation>
```

(from [FragmentNav/src/main/res/navigation/main_nav_graph.xml](#))

listFragment

There are two `<fragment>` elements set up as destinations in this resource. One is

NAVIGATING YOUR APP

listFragment, pointing to our ListFragment class:

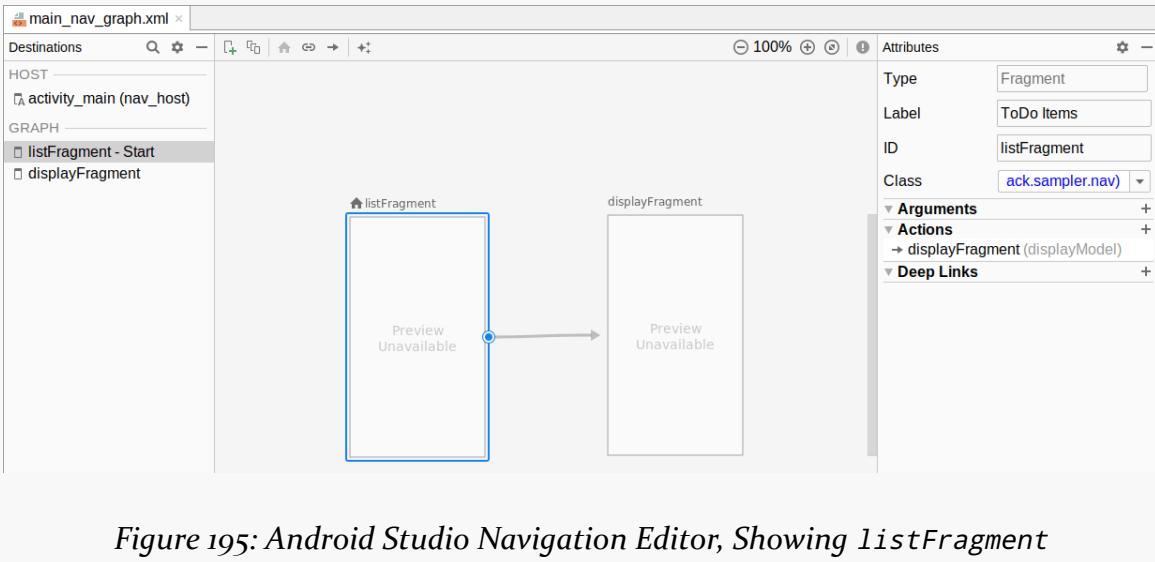


Figure 195: Android Studio Navigation Editor, Showing listFragment

It contains an <action> child element, identifying that this destination can navigate to the displayFragment destination. The action's ID is displayModel, and that is what we can use in our Java/Kotlin code to request to perform this action.

Clicking the action's arrow in the graphic designer shows the details of that action in the “Attributes” pane:

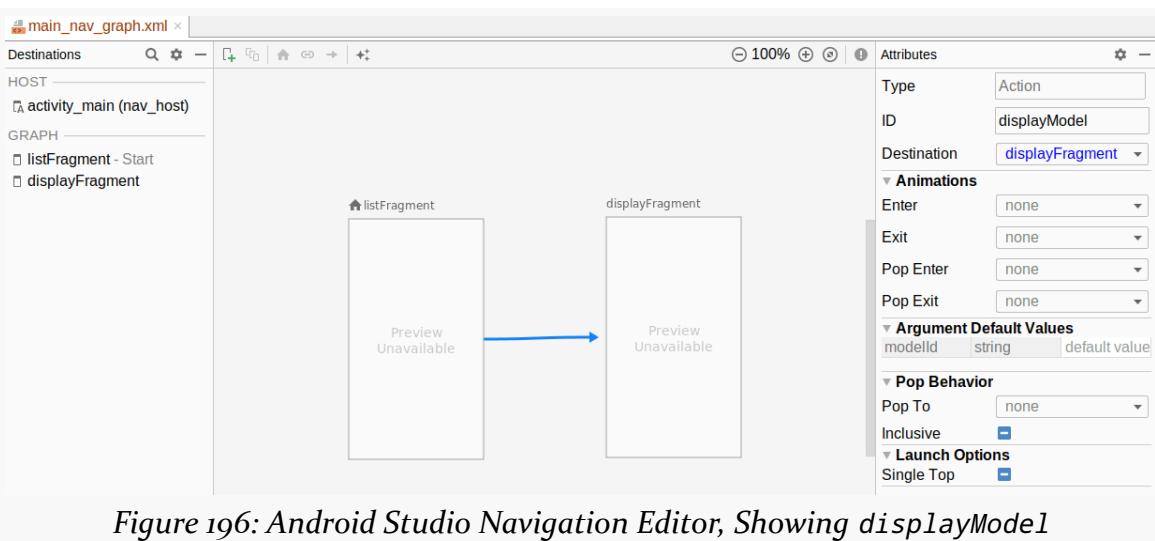


Figure 196: Android Studio Navigation Editor, Showing displayModel

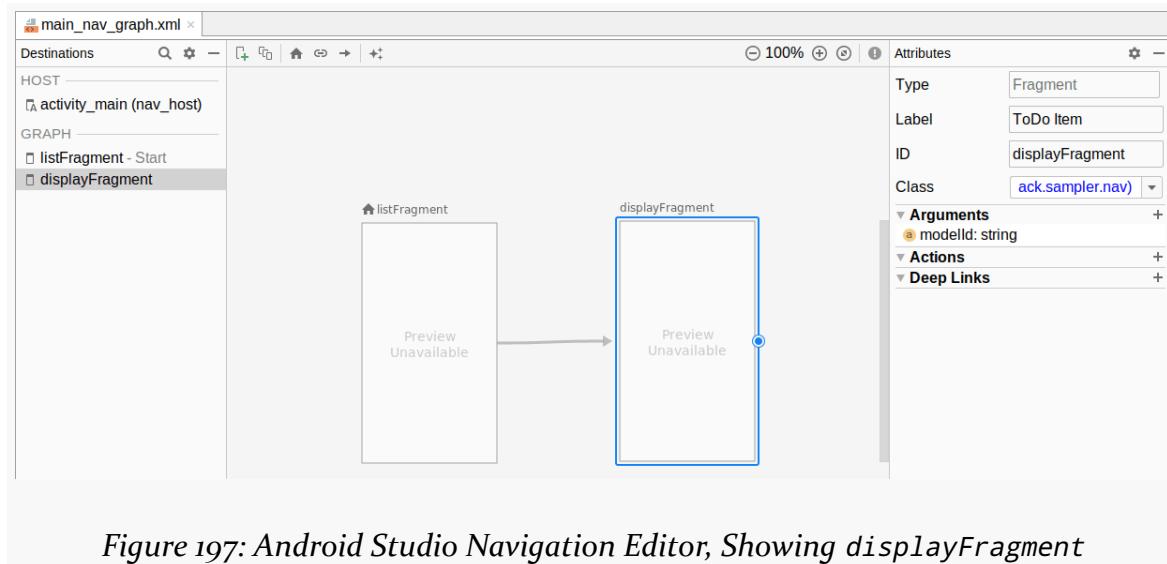
The app:startDestination attribute in the root <navigation> element points to

NAVIGATING YOUR APP

this `listFragment` destination, meaning that when we first use this navigation graph, the `ListFragment` is what should be displayed.

displayFragment

The other `<fragment>` element is for `displayFragment`:



This contains an `<argument>` child element, stating that this destination expects to receive a `modelId` String value.

The Activity Layout

In the `FragmentManual` sample, we did not need an activity layout. We just used the existing `android.R.id.content` container as a target for our `FragmentTransaction` requests.

With the Navigation component, usually you will need a layout resource, particularly one where you place the `NavHostFragment` to say where the navigation graph's `<fragment>` destinations should appear. So, `FragmentNav` has an `activity_main` layout resource that does just that:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
```

NAVIGATING YOUR APP

```
xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

<androidx.appcompat.widget.Toolbar
    android:id="@+id/toolbar"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:background="?attr/colorPrimary"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:theme="?attr/actionBarPopupTheme" />

<fragment
    android:id="@+id/nav_host"
    android:name="androidx.navigation.fragment.NavHostFragment"
    android:layout_width="0dp"
    android:layout_height="0dp"
    app:defaultNavHost="true"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/toolbar"
    app:navGraph="@navigation/main_nav_graph" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

(from [FragmentNav/src/main/res/layout/activity_main.xml](#))

The NavHostFragment is added using a <fragment> element in the layout XML. This is how you add “static fragments”, ones that will be used all the time and cannot themselves be removed. The android:name attribute identifies the particular Fragment subclass to use, using the fully-qualified class name. In this case, we are not using one of our fragments, but instead are using NavHostFragment (or, more formally, androidx.navigation.fragment.NavHostFragment).

The NavHostFragment takes two main custom attributes in addition to the standard ones for sizing and positioning the fragment. app:navGraph supplies a reference to the navigation graph resource that this host will be using for its contents. app:defaultNavHost basically says “have the system’s BACK button route through this fragment’s navigation graph”, so BACK presses navigate backwards through the graph.

The layout also has a Toolbar to use as our app bar, with the NavHostFragment

NAVIGATING YOUR APP

taking up the rest of the available space. As a result, this app uses `Theme.AppCompat.Light.NoActionBar` as a basis for its `AppTheme` style resource, to suppress the default action bar implementation, as we saw back in [the chapter on the app bar](#).

MainActivity

Most of the `FragmentNav` app's code is the same as its `FragmentManual` counterpart. We still have two fragments, each with its own viewmodel, with `ToDoModel` objects coming from a `ToDoRepository`. We also still have the same `RecyclerView` pieces for populating the list.

What differs is how our activity sets things up, how we navigate from the `ListFragment` to the `DisplayFragment`, and how we pass the `ToDoModel` ID between those fragments.

The `FragmentManual` edition of `MainActivity` conditionally executed a `FragmentTransaction` to bring up the `ListFragment`. The Navigation-enhanced `MainActivity` instead needs to initialize the Navigation component.

The Android KTX extensions for Kotlin cause the Java and Kotlin code to look somewhat different, even though in the end they do the same work:

- Obtain the `NavController` associated with our `NavHostFragment`
- Hook up that `NavController` to our `Toolbar`, to automatically manage things like the `Toolbar` title and up button

Java

```
package com.commonsware.jetpack.samplerj.nav;

import android.os.Bundle;
import androidx.annotation.Nullable;
import androidx.appcompat.app.AppCompatActivity;
import androidx.navigation.NavController;
import androidx.navigation.Navigation;
import androidx.navigation.ui.AppBarConfiguration;
import androidx.navigation.ui.NavigationUI;

public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
```

NAVIGATING YOUR APP

```
setContentView(R.layout.activity_main);

NavController nav = Navigation.findNavController(this, R.id.nav_host);
AppBarConfiguration appBarCfg =
    new AppBarConfiguration.Builder(nav.getGraph()).build();

NavigationUI.setupWithNavController(findViewById(R.id.toolbar), nav,
    appBarCfg);
}

}
```

(from [FragmentNav/src/main/java/com/commonsware/jetpack/samplerj/nav/MainActivity.java](#))

In Java, to obtain the NavController, we call `Navigation.findNavController()`, providing our activity and the ID given to our NavHostFragment in our layout (`R.id.nav_host`). That is sufficient to bring up the ListFragment, since it is the start destination indicated in our navigation graph resource.

After that, we:

- Ask the NavController for the NavGraph object that represents the parsed navigation resource (`getGraph()`)
- Pass that to an `AppBarConfiguration.Builder` and use that to build an `AppBarConfiguration`
- Pass that along with the NavController and the Toolbar to `NavigationUI.setupWithNavController()`, to have the Navigation component automatically manage the contents of the Toolbar

Now, the Navigation component will fill in the Toolbar title from our destination label and handle the up button, as we navigate through the graph that we have set up for this NavHostFragment.

Kotlin

The Kotlin code does the same work, with a lot less actual code:

```
package com.commonsware.jetpack.sampler.nav

import android.os.Bundle
import androidx.appcompat.app.AppCompatActivity
import androidx.navigation.findNavController
import androidx.navigation.ui.AppBarConfiguration
import androidx.navigation.ui.setupWithNavController
import kotlinx.android.synthetic.main.activity_main.*
```

NAVIGATING YOUR APP

```
class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        findNavController(R.id.nav_host).let { nav ->
            toolbar.setupWithNavController(nav, AppBarConfiguration(nav.graph))
        }
    }
}
```

(from [FragmentNav/src/main/java/com/commonsware/jetpack/sampler/nav/MainActivity.kt](#))

Android KTX gives us:

- A `findNavController()` extension function for `Activity` and `Fragment`, so we can get our `NavController` just by calling `findNavController()` and providing the ID of the `NavHostFragment`;
- A `setupWithNavController()` extension function on `Toolbar`, so we can add Navigation support for it more directly; and
- A global `AppBarConfiguration()` function that *looks* like a constructor, though it actually uses `AppBarConfiguration.Builder` “under the covers”

ListFragment

The change to `ListFragment` is in the `navTo()` function, as now we use the Navigation component instead of a `FragmentTransaction` to bring up the `DetailFragment`.

This app uses Safe Args, courtesy of the `<argument>` element in the navigation graph and the `androidx.navigation.safeargs` Gradle plugin. Hence, we get a `ListFragmentDirections` class code-generated for us. This class is named based on the initial destination (`ListFragment`) and has static methods for each of our actions that collect the arguments that we need to pass to the destination that we want to navigate to. Our action has `displayModel` as its ID, so `ListFragmentDirections` has a `displayModel()` method that we can call, either from Java:

```
private void navTo(TodoModel model) {
    NavHostFragment.findNavController(this)
        .navigate(ListFragmentDirections.displayModel(model.id));
}
```

NAVIGATING YOUR APP

(from [FragmentNav/src/main/java/com/commonsware/jetpack/samplerj/nav/ListFragment.java](#))

...or from Kotlin:

```
private fun navTo(model: ToDoModel) {
    findNavController().navigate(ListFragmentDirections.displayModel(model.id))
}
```

(from [FragmentNav/src/main/java/com/commonsware/jetpack/samplerj/nav/ListFragment.kt](#))

To navigate to a destination, we:

- Call `displayModel()`, passing it our `ToDoModel` ID as the argument, which gives us a `NavDirections` object that encapsulates our action ID (`displayModel`) and arguments
- Get our `NavController` using either `NavHostFragment.findNavController()` (in Java) or the `findNavController()` extension function in Kotlin
- Pass the `NavDirections` to the `navigate()` function on our `NavController`

The Navigation component will then do whatever is necessary to take us to that destination, updating the Toolbar as needed.

DisplayFragment

The only change in `DisplayFragment` is how we get our `ToDoModel` ID. In `FragmentManual`, we used the arguments Bundle directly. In `FragmentNav`, we are using Safe Args, which gives us a code-generated `DisplayFragmentArgs` class. This class is named after our destination (`DisplayFragment`), knows how to obtain its arguments, and supplies them to us in type-safe functions, such as `getModelId()` for our `modelId` argument.

In Java, we get a `DisplayFragmentArgs` by calling the static `fromBundle()` method, providing it our arguments Bundle:

```
String modelId = DisplayFragmentArgs.fromBundle(getArguments()).getModelId();
```

(from [FragmentNav/src/main/java/com/commonsware/jetpack/samplerj/nav/DisplayFragment.java](#))

From there, a call to `getModelId()` returns the `ToDoModel` ID that we supplied to `ListFragmentDirections.displayModel()`.

In Kotlin, while you could do the same thing, you also have the `navArgs()` property delegate from Android KTX:

NAVIGATING YOUR APP

```
private val args: DisplayFragmentArgs by navArgs()
```

(from [FragmentNav/src/main/java/com/commonsware/jetpack/sampler/nav/DisplayFragment.kt](#))

That knows how to get our arguments Bundle and set up our `DisplayFragmentArgs` instance, so we can just refer to its `modelId` property to get the `ToDoModel` ID:

```
val model = vm.getModel(args.modelId)
```

(from [FragmentNav/src/main/java/com/commonsware/jetpack/sampler/nav/DisplayFragment.kt](#))

So... Was It Worth It?

The Navigation component, particularly with the Safe Args plugin, has some nice features and characteristics. If your app has a fairly simple navigation graph, and the UI is not too elaborate, the Navigation component may well be worth using.

However, every time you choose a framework like the Navigation component, you sacrifice flexibility in favor of ease of development. There will be certain types of navigation or UI structures that the Navigation component simply will not support. If you need those things, you may not be able to use the Navigation component for some or all of your app.

Licensed solely for use by Patrocinio Rodriguez

Thinking About Threads and LiveData

Users like apps that run smoothly. Users do not like applications that feel sluggish.

Usually, the reason why apps feel sluggish is due to problems with thread management, particularly when performing some form of I/O (disk, database, network, etc.). We need to take care when performing I/O that we do not slow down the UI of the app, while still getting the data that we need to display.

Nowadays, working with background threads is often done in conjunction with some form of reactive programming. In reactive programming, we request for work to be done and arrange to get control when it is completed. That work can be performed on one thread, while we get the results on some other thread. In particular, reactive programming attempts to hide a lot of that thread complexity, making it seem like we are doing “normal” programming.

This chapter introduces the key threading issues with Android and explores the Jetpack solution for reactive programming: `LiveData`. Mostly, this chapter is to “set the stage” for exploring I/O in upcoming chapters.

The Main Application Thread

When you call `setText()` on a `TextView`, you probably think that the screen is updated with the text you supply, right then and there.

That’s not really what happens.

Rather, everything that modifies the widget-based UI goes through a message queue. Calls to `setText()` do not update the screen — they just place a message on a queue telling the operating system to update the screen. The operating system pops these

THINKING ABOUT THREADS AND LIVEDATA

messages off of this queue and does what the messages require.

The queue is processed by one thread, variously called the “main application thread” and the “UI thread”. So long as that thread can keep processing messages, the screen will update, user input will be handled, and so on.

However, the main application thread is also used for nearly all callbacks into your activity. Your `onCreate()`, `onClick()`, `onCreateView()`, and similar functions are all called on the main application thread. While your code is executing in these functions, Android is not processing messages on the queue, and so the screen does not update, user input is not handled, and so on.

This, of course, is bad. So bad, that if you take more than a few seconds to do work on the main application thread, Android may display the dreaded “Application Not Responding” dialog (ANR for short), and your activity may be killed off.

Nowadays, though, the bigger concern is jank.

“Jank”, as used in Android, refers to sluggish UI updates, particularly when something is animating. For example, you may have encountered some apps that when you scroll in the app, the content does not scroll smoothly. Rather, it scrolls jerkily, interleaving periods of rapid movement with periods where the scrolling animation is frozen. Most of the time, this is caused by the app’s author doing too much work on the main application thread.

Android widget-based UIs are updated at 60 frames per second. This means that any given frame needs to be assembled in less than 16ms. Since the framework and OS have work that needs to be done too, that means our application code must run in a lot less than 16ms per frame. And, since our app may be called on many times in a frame, this means that any given callback function, such as `onBindViewHolder()` of a `RecyclerView`, needs to be done *very* quickly, best measured in microseconds. If, instead, we take several milliseconds, we will “drop frames” (i.e., the screen does not get updated between two successive frames), and the user may perceive jank as a result.

Hence, you want to make sure that all of your work on the main application thread happens quickly. This means that anything slow should be done in a background thread, so as not to tie up the main application thread. This includes things like:

- Internet access, such as sending data to a Web service or downloading an image

- Significant file operations, since flash storage can be remarkably slow at times
- Any sort of complex calculations

The UI Thread is for UI

However, there is one big limitation: in general, you cannot modify the UI from a background thread. You can only modify the UI from the main application thread.

A few widgets have special support for being updated from a background thread. `ProgressBar` is one in particular that is set up this way. `ProgressBar` is designed to let you show progress of some background work, and so the Android engineers took the time to make `ProgressBar` support background UI updates.

However, outside of exceptions like that, any other UI updates have to be done from the main application thread.

This is a pain.

It means that we not only have to get our slow work off of the main application thread, but we have to update the UI with the results of that work on the main application thread. This sort of inter-thread communication is annoying.

Worse, this makes configuration changes that much trickier. Suppose we start some work on a background thread, and while that work is ongoing, the user rotates the screen. Our activity and its fragments get destroyed and recreated. Somehow, we not only need to arrange for our results to get processed on the main application thread, but they also need to be processed by whatever the now-current activity and fragments are.

This has been one of the major headaches that has plagued Android app developers since Android 1.0. Countless solutions have been offered by Google and by independent developers. The current “best practices” involve:

- Using a `ViewModel` (or similar construct) that is stable across configuration changes, so our code running on a background thread can hand the results to the `ViewModel` without having to worry as much about configuration changes
- Employing some sort of reactive programming solution for implementing that background work and getting the results over to the main application

thread

Introducing LiveData

However, even those “best practices” have a bit of a gap: how do we push data from a `ViewModel` to an activity or fragment? More importantly, how do we do so while taking into account configuration changes?

So far, our viewmodels have been simple state containers. The activities and fragments pull data from the viewmodels and push data into them.

Now, though, if we are saying that the viewmodel is somehow getting data delivered to it from the background, somehow we need the activity or fragment to find out that the data has arrived. So, for example, suppose that our fragment asks the viewmodel to retrieve some data from a Web site. That may take hundreds of milliseconds or longer, so the actual network I/O needs to be done on a background thread. When that I/O completes, the now-current fragment needs to find out about that, bearing in mind that it might be a different fragment instance by now, if a configuration change occurred while the I/O was ongoing.

The Jetpack solution to this problem is `LiveData`.

`LiveData` is itself a state container, designed to be held onto by a `ViewModel`. It has an `Observer` system, where interested parties can find out about changes in the data associated with the `LiveData`. So, if a background thread pushes new data into the `LiveData`, the `LiveData` will inform all of its observers.

More importantly, `LiveData` is aware of lifecycles, such as those from activities and fragments. When an activity or fragment registers an `Observer`, and later the activity or fragment is destroyed, the `LiveData` automatically removes the associated `Observer`. Hence, when we undergo a configuration change, we do not need to remember to remove any `Observer` objects that we may have registered with `LiveData` — those get cleaned up automatically.

`LiveData` also automatically pushes an existing value, if there is one, to any new registered `Observer` objects. When coupled with the automatic-removal feature, this makes `LiveData` very convenient for dealing with configuration changes. The activities and fragments can largely forget about configuration changes entirely, just reacting to whatever the `LiveData` pushes to it. Those “pushes” will either be:

THINKING ABOUT THREADS AND LIVEDATA

- Because some asynchronous work completed, or
- The device underwent a configuration change and the new activity or fragment is getting the data from before the change

Usually, the activity or fragment does not care about which of those scenarios occurred and can handle them identically.

Sources of LiveData

In many cases, you will get LiveData objects handed to you from something else. For example, if you use Room for working with on-device databases, you can have the results of your queries be in the form of LiveData. We will explore that more [later in the book](#).

Sometimes, you will use adapters to convert one form of reactive response into LiveData. For example, RxJava is the most popular reactive framework for Java. Jetpack offers a library that helps you convert RxJava reactive responses into LiveData, as RxJava itself knows nothing about Android and things like activity/fragment lifecycles.

Otherwise, you are on your own for creating LiveData objects. For that, there are two main approaches:

1. Create a subclass of LiveData that handles the background threading and delivery of updates
2. Use MutableLiveData (which does not require a subclass the way LiveData does) and have code outside of that object deal with data sources, threading, etc.

We will see both of those approaches in this chapter.

Active and Inactive States

If a LiveData was instantiated in a forest, and nobody was there to observe data changes, does the LiveData really exist?

The answer is: yes, but it hopefully is not consuming any resources.

A LiveData implementation will be called with `onActive()` when it receives its first active observer. Here, “active” means that the activity or fragment registering the observer is in the started or resumed state. Conversely, the LiveData will be called

THINKING ABOUT THREADS AND LIVEDATA

with `onInactive()` once it no longer has any active observers, either because all observers have been unregistered or none of them are active, as their lifecycles are all stopped or destroyed.

The idea is that a `Livedata` would only start consuming significant system resources — such as requesting GPS fixes — when there are active observers, releasing those resources when there are no more active observers. This works in many cases, though there are some that will require more finesse. For example, given that the GPS radio takes some time before it starts generating GPS fixes, a `Livedata` for GPS might want to wait some amount of time after losing its last active observer before releasing the GPS radio, in case a new observer pops up quickly, to avoid delays in getting those GPS fixes.

Subclasses of `Livedata`, therefore, should override `onActive()` (and perhaps `onInactive()`) and use those events to control resource consumption.

Colors... Live!

The `InLivingColor` sample module in the [Sampler](#) and [SamplerJ](#) projects implement the same basic UI as we saw in the `TwoActivities` sample [earlier in the book](#).

Part of the changes are to switch from two activities to two fragments and use the Navigation component. That largely mirrors what we saw [previously](#), so we will not focus on those changes here.

The change of importance for these samples are how we get our list of randomly-generated colors. In the earlier list-of-colors examples, we just generated colors directly in the `ColorViewModel`. This time, we are going to pretend that it takes “real work” to get these colors, such as having to call out to some random-color-generating Web service. To that end, we have a `ColorLiveData` subclass of `Livedata` that handles the threading aspects and generates the colors. `ColorViewModel` now exposes a `Livedata` of colors to observers, and a `ColorListFragment` pours those colors into the `RecyclerView` when they are ready.

ColorLiveData

Our `ColorLiveData` is responsible for creating a random set of colors, using a background thread. We only want to do this once we have one active observer — otherwise, the colors are pointless. So, our `Livedata` subclass overrides `onActive()`

THINKING ABOUT THREADS AND LIVEDATA

and handles the work there:

```
package com.commonsware.jetpack.samplerj.livedata;

import android.os.SystemClock;
import java.util.ArrayList;
import java.util.Random;
import java.util.concurrent.Executor;
import java.util.concurrent.Executors;
import androidx.lifecycle.LiveData;

class ColorLiveData extends LiveData<ArrayList<Integer>> {
    private final Random random = new Random();
    private final Executor executor = Executors.newSingleThreadExecutor();

    @Override
    public void onActive() {
        super.onActive();

        if (getValue() == null) {
            executor.execute(() -> {
                SystemClock.sleep(2000); // use only for book samples!
                postValue(buildItems());
            });
        }
    }

    private ArrayList<Integer> buildItems() {
        ArrayList<Integer> result = new ArrayList<>(25);

        for (int i = 0; i < 25; i++) {
            result.add(random.nextInt());
        }

        return result;
    }
}
```

(from [InLivingColor/src/main/java/com/commonsware/jetpack/samplerj/livedata/ColorLiveData.java](#))

```
package com.commonsware.jetpack.sampler.livedata

import android.os.SystemClock
import androidx.lifecycle.LiveData
import java.util.*
import java.util.concurrent.Executors

class ColorLiveData : LiveData<List<Int>>()
```

THINKING ABOUT THREADS AND LIVEDATA

```
private val random = Random()
private val executor = Executors.newSingleThreadExecutor()

override fun onActive() {
    super.onActive()

    if (value == null) {
        executor.execute {
            SystemClock.sleep(2000) // use only for book samples!
           .postValue(List(25) { random.nextInt() })
        }
    }
}
```

(from [InLivingColor/src/main/java/com/commonsware/jetpack/sampler/livedata/ColorLiveData.kt](#))

We use a Java Executor for performing our background work. For what we are doing now, a simple single-thread Executor is sufficient, but Executor gives us the flexibility to swap in a thread pool if we decided that we needed lots of random colors.

In `onActive()`, we see if we already have our colors. We do this by calling `getValue()`, which will return the value held by the `LiveData` or `null` if we have not yet generated any colors. If we do not already have colors, we use the Executor to run some code on a background thread to generate the colors.

We use `SystemClock.sleep()` to simulate two seconds worth of I/O to get these random colors, as our fictitious random-color-generating Web service is overloaded and our pretend Internet connection is poor. If you have used `Thread.sleep()` in Java before, `SystemClock.sleep()` works much the same way, blocking the current thread for the stated number of milliseconds. The biggest difference: `Thread.sleep()` throws an `InterruptedException`, which is unnecessary here and the resulting `try/catch` block just clutters up the example

To update the `LiveData` with the new colors, we call `postValue()`. This updates the `LiveData` and — back on the main application thread — lets each of the registered observers know about the new data.

This particular `LiveData` only ever calls `postValue()` once. That is all this sample needs. However, there is nothing stopping you from having a `LiveData` that delivers a stream of updates, such as a stream of sensor readings. There, you might call `postValue()` hundreds or thousands of times, with the `LiveData` informing its observers each time.

ColorViewModel Changes

The earlier example's ColorViewModel would obtain the random list of colors from the saved instance state Bundle. If the Bundle was null, then the ColorViewModel would generate the colors to use.

Now, the color-generation code is in ColorLiveData. Our ColorViewModel can hold a ColorLiveData, so our fragment can get the colors as soon as they are ready. However, we still need to deal with the saved instance state scenario... and now our API is a LiveData of colors, not just the colors themselves.

So, if we have our colors from the saved instance state Bundle, ColorViewModel uses a MutableLiveData instead of a ColorLiveData:

```
package com.commonsware.jetpack.samplerj.livedata;

import android.os.Bundle;
import java.util.ArrayList;
import androidx.lifecycle.LiveData;
import androidx.lifecycle.MutableLiveData;
import androidx.lifecycle.ViewModel;

class ColorViewModel extends ViewModel {
    private final String STATE_NUMBERS = "numbers";
    final LiveData<ArrayList<Integer>> numbers;

    ColorViewModel(Bundle state) {
        ArrayList<Integer> existing =
            state == null ? null : state.getIntegerArrayList(STATE_NUMBERS);

        if (existing == null) {
            numbers = new ColorLiveData();
        }
        else {
            MutableLiveData<ArrayList<Integer>> mutableNumbers =
                new MutableLiveData<>();

            mutableNumbers.setValue(existing);
            numbers = mutableNumbers;
        }
    }

    void onSaveInstanceState(Bundle state) {
        state.putIntegerArrayList(STATE_NUMBERS, numbers.getValue());
    }
}
```

THINKING ABOUT THREADS AND LIVEDATA

```
}
```

(from [InLivingColor/src/main/java/com/commonsware/jetpack/sampler/livedata/ColorViewModel.java](#))

```
package com.commonsware.jetpack.sampler.livedata

import android.os.Bundle
import androidx.lifecycle.LiveData
import androidx.lifecycle.MutableLiveData
import androidx.lifecycle.ViewModel

private const val STATE_NUMBERS = "numbers"

class ColorViewModel(state: Bundle?) : ViewModel() {
    val numbers =
        state?.getIntegerArrayList(STATE_NUMBERS)?.let { wrapInLiveData(it) } ?: ColorLiveData()

    fun onSaveInstanceState(state: Bundle) {
        numbers.value?.let {
            state.putIntegerArrayList(STATE_NUMBERS, ArrayList(it))
        }
    }

    private fun wrapInLiveData(values: ArrayList<Int>): LiveData<ArrayList<Int>> =
        MutableLiveData<ArrayList<Int>>().apply { setValue(values) }
}
```

(from [InLivingColor/src/main/java/com/commonsware/jetpack/sampler/livedata/ColorViewModel.kt](#))

LiveData offers `setValue()` and `postValue()` methods to supply it with a value. ColorLiveData uses `postValue()`, because you can call `postValue()` safely from any thread. `setData()` by comparison needs to be called on the main application thread. In LiveData, both `setValue()` and `postValue()` are protected and so are only available to subclasses. One such subclass is `MutableLiveData`, which turns around and makes those methods public. Hence, `MutableLiveData` can be used without subclassing. Mostly, it is used for cases where we do not need to override `onActive()` or `onInactive()`.

One such case is here, when we have our colors from the Bundle and do not need a ColorLiveData to generate them. The property that ColorViewModel exposes is typed as a `LiveData<ArrayList<Int>>`, so the hosting fragment does not care whether this is a ColorLiveData, a MutableLiveData, or something else. So, we use ColorLiveData and its background thread when we need to generate the colors, and we use MutableLiveData when we already have the colors and simply need something to fulfill the LiveData contract. We fill the existing colors into the MutableLiveData via `setValue()`, so by the time the fragment accesses the `numbers` property, the colors will already be in place.

Observing the Colors

In the earlier list-of-colors example, we would call `submitList()` on our `ColorAdapter` for our colors with the numbers that we get from the `ColorViewModel`. Now, we *observe* the `numbers` property, as that is a `LiveData`:

```
package com.commonsware.jetpack.samplerj.livedata;

import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import androidx.annotation.NonNull;
import androidx.fragment.app.Fragment;
import androidx.lifecycle.ViewModelProviders;
import androidx.navigation.fragment.NavHostFragment;
import androidx.recyclerview.widget.DividerItemDecoration;
import androidx.recyclerview.widget.LinearLayoutManager;
import androidx.recyclerview.widget.RecyclerView;

public class ColorListFragment extends Fragment {
    private ColorViewModel vm;

    @Override
    public View onCreateView(@NonNull LayoutInflater inflater,
                           ViewGroup container,
                           Bundle savedInstanceState) {
        return inflater.inflate(R.layout.fragment_list, container, false);
    }

    @Override
    public void onViewCreated(@NonNull View view, Bundle savedInstanceState) {
        vm = ViewModelProviders
            .of(this, new ColorViewModelFactory(savedInstanceState))
            .get(ColorViewModel.class);

        ColorAdapter colorAdapter = new ColorAdapter(getActivity(),
            color -> navTo(color));
        RecyclerView items = view.findViewById(R.id.items);

        items.setLayoutManager(new LinearLayoutManager(getActivity()));
        items.addItemDecoration(new DividerItemDecoration(getActivity(),
            DividerItemDecoration.VERTICAL));
        items.setAdapter(colorAdapter);

        vm.numbers.observe(getViewLifecycleOwner(), colorAdapter::submitList);
    }
}
```

THINKING ABOUT THREADS AND LIVEDATA

```
@Override  
public void onSaveInstanceState(@NonNull Bundle outState) {  
    super.onSaveInstanceState(outState);  
  
    vm.onSaveInstanceState(outState);  
}  
  
private void navTo(int color) {  
    NavHostFragment.findNavController(this)  
        .navigate(ColorListFragmentDirections.showColor(color));  
}  
}
```

(from [InLivingColor/src/main/java/com/commonsware/jetpack/samplerj/livedata/ColorListFragment.java](#))

```
package com.commonsware.jetpack.sampler.livedata  
  
import android.os.Bundle  
import android.view.LayoutInflater  
import android.view.View  
import android.view.ViewGroup  
import androidx.fragment.app.Fragment  
import androidx.lifecycle.Observer  
import androidx.lifecycle.ViewModelProviders  
import androidx.lifecycle.get  
import androidx.navigation.fragment.findNavController  
import androidx.recyclerview.widget.DividerItemDecoration  
import androidx.recyclerview.widget.LinearLayoutManager  
import kotlinx.android.synthetic.main.fragment_list.*  
  
class ColorListFragment : Fragment() {  
    private lateinit var vm: ColorViewModel  
  
    override fun onCreateView(  
        inflater: LayoutInflater,  
        container: ViewGroup?,  
        savedInstanceState: Bundle?  
    ): View? = inflater.inflate(R.layout.fragment_list, container, false)  
  
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {  
        vm = ViewModelProviders  
            .of(this, ColorViewModelFactory(savedInstanceState))  
            .get()  
  
        val colorAdapter = ColorAdapter(layoutInflater) { color ->  
            navTo(color)  
        }  
  
        items.apply {  
            layoutManager = LinearLayoutManager(activity)  
            addItemDecoration(  
                DividerItemDecoration(  
                    activity,  
                    DividerItemDecoration.VERTICAL  
                )  
        }  
    }  
}
```

THINKING ABOUT THREADS AND LIVEDATA

```
        )
    adapter = colorAdapter
}

vm.numbers.observe(viewLifecycleOwner, Observer { colorAdapter.submitList(it) })

override fun onSaveInstanceState(outState: Bundle) {
    super.onSaveInstanceState(outState)

    vm.onSaveInstanceState(outState)
}

private fun navTo(color: Int) {
    findNavController().navigate(ColorListFragmentDirections.showColor(color))
}
}
```

(from [InLivingColor/src/main/java/com/commonsware/jetpack/sampler/livedata/ColorListFragment.kt](#))

`observe()` on a `LiveData` takes two parameters:

1. A `LifecycleOwner`, usually supplied by the activity or fragment that is using the `LiveData`, so the `LiveData` knows when it is active or inactive
2. An `Observer` implementation

An `Observer` has a single method, `observe()`, that returns `void` and takes an instance of whatever type the `LiveData` holds. So, in our case, `observe()` will be passed an `ArrayList<Int>`. In the Java scenario, we use a Java 8 method reference to hand that `ArrayList` over to the `ColorAdapter`. In Kotlin, we use the SAM pattern to adapt a lambda expression into an instance of `Observer`.

We will look more at the `LifecycleOwner` parameter, and what `viewLifecycleOwner` is, [shortly](#).

The Results

When the fragment is first created, it will have a null saved instance state `Bundle`. When it creates the `ColorViewModel` (by way of `ViewModelProviders`), the `ColorViewModel` will see that it has no colors, so it creates a `ColorLiveData`. When the fragment then calls `observe()` on the `LiveData`, and the fragment is later shown on the screen, the `ColorLiveData` will be called with `onActive()`, as the fragment will now be started. At that point, the `ColorLiveData` will generate the colors (after a two-second delay). When `ColorLiveData` calls `postValue()`, the `Observer` registered by the fragment will be called, and the `ColorAdapter` will get its colors.

If the user rotates the screen, our original `ColorListFragment` will be destroyed and

THINKING ABOUT THREADS AND LIVEDATA

recreated. When the fresh `ColorListFragment` gets its `ColorViewModel`, it will be the already-existing instance. When the fragment calls `observe()`, its `Observer` will be called immediately if the `ColorLiveData` already has its colors — that is handled automatically by `LiveData`. Our fragment's code does not care whether it is the first or second instance of the `ColorListFragment` — it gets the colors the same way and consumes them the same way.

Suppose the user presses HOME and leaves the app for a bit. Android terminates the app's process to free up system RAM. 15 minutes later, the user returns to the app. Android forks a fresh process and spins up a new `ColorListFragment`. This time, when the `ColorListFragment` gets its `ColorViewModel`, the colors will exist in the saved instance state Bundle. So `ColorViewModel` skips the `ColorLiveData` and just wraps the colors in a `MutableLiveData`. When the fragment calls `observe()`, its observer will be handed the colors immediately. Once again, our fragment's code does not care whether it is the first or second process with a `ColorListFragment` — it gets the colors the same way and consumes them the same way.

Sources of Owners

`observe()` on a `LiveData` takes a `LifecycleOwner`. Technically, you could implement the `LifecycleOwner` interface on just about anything. In practice, there are three `LifecycleOwner` implementations that matter:

1. `FragmentActivity` and things that inherit from it, like `AppCompatActivity`
2. `Fragment`
3. The “view `LifecycleOwner`” of a `Fragment`

The first two are fairly straightforward. They let the `LiveData` know about the lifecycle of the activity and the fragment, respectively.

However, fragments are weird.

Fragments have, in effect, two lifecycles:

1. The lifecycle of the fragment itself
2. The lifecycle of a particular UI managed by that fragment

For example, let's consider the to-do app that we examined in previous chapters. We had two fragments: one showing the list of items, and another showing the details of a particular item. We show the list first, then we show the details if the

THINKING ABOUT THREADS AND LIVEDATA

user taps on an item. Suppose that the user launches the activity, taps on an item, then presses BACK to return to the list. If the user did not rotate the screen, the list fragment itself (`RosterListFragment` instance) is the same object both times it appears. However, its *UI* is different, as `onCreateView()` gets called twice, once for each time the fragment is displayed.

So, the “view LifecycleOwner” reflects the lifecycle of the *widgets* that a fragment manages, while the fragment itself is a `LifecycleOwner` for its overall lifecycle. You get the “view LifecycleOwner” by calling `getViewLifecycleOwner()` on the Fragment.

The general rule of thumb is:

- If you are observing a `LiveData` solely for populating a UI, such as we are doing here in `ColorListFragment`, use `getViewLifecycleOwner()` as the `LifecycleOwner` and start observing in `onViewCreated()` (or possibly in `onCreateView()`)
- If you are observing a `LiveData` and need to observe for the entire lifetime of the fragment, regardless of its UI, use the `Fragment` itself as the `LifecycleOwner` and start observing in `onCreate()`

Where Do Threads Come From? Um, Besides From Me?

In this sample, we created our own thread, by means of `Executors.newSingleThreadExecutor()`. This will be needed in some situations, where you are using a synchronous API, one that offers no sort of callback or other asynchronous option.

However, frequently, something else might create the threads for you.

Threads from Reactive Frameworks

As mentioned earlier, RxJava is the most popular reactive framework for Java development. With RxJava, you indicate what work you want done and what RxJava-supplied thread pool that you want that work to be done on (e.g., “use the thread pool dedicated for I/O”). RxJava takes care of allocating the threads and using them as appropriate.

In Kotlin, coroutines are gaining in popularity. Coroutines, along with other Kotlin

THINKING ABOUT THREADS AND LIVEDATA

language and standard library features, offer a lot of the same capabilities as does RxJava. Kotlin coroutines, though, can support Kotlin's array of platform targets (e.g., JavaScript) in addition to Android, while RxJava is tied to Java-based apps. On the other hand, coroutines are still somewhat of a work in progress and are fairly new to Kotlin overall, whereas RxJava has been around somewhat longer. But, if you use coroutines, once again you will not need to create threads yourself — you describe where you want work to be done thread-wise, and the coroutines engine handles the actual threading.

Threads from Data Sources

Many libraries that support database and network I/O offer their own thread pools for performing that I/O. They offer either a true reactive API or a classic callback-based system for getting the I/O results asynchronously.

Room, mentioned earlier in this chapter, is the Jetpack solution for on-device database access. Room wraps Android's SQLite support in an API that, among other things, supports both synchronous and reactive access. If you prefer, you can call synchronous APIs and handle the threading yourself. Or, you can have Room respond with `LiveData`, interact with RxJava, or support Kotlin coroutines. We will explore Room more [later in the book](#).

There are a massive number of third-party libraries available for Internet access. Some are general-purpose for accessing data over HTTPS, such as OkHttp. Others are optimized for specific types of Web access:

- Images (e.g., Glide, Picasso)
- REST Web services (Retrofit)
- GraphQL Web services (Apollo-Android)

All offer asynchronous APIs. In some cases, they offer integration with reactive frameworks like RxJava. In others, they just allow you to supply a callback function to be invoked when the I/O is completed.

Threads from Background Processing

Some work needs to be done in the background as a result of the user doing something in your UI. For example, a Twitter client needs to refresh the timeline when the user clicks a refresh button or performs a pull-to-refresh gesture. In those cases, using threads and `LiveData`, RxJava, or Kotlin coroutines are recommended.

THINKING ABOUT THREADS AND LIVEDATA

In other cases, the work may need to happen somewhat later, even if the user is no longer interacting with the app. For example, you may have a business requirement to check a server once per hour, every hour. Here, the user is not triggering the need for background work — the passage of time is.

WorkManager is the Jetpack solution for this problem. It handles threading for you; your work will be performed on a background thread automatically. We will explore WorkManager more [later in the book](#).

Coroutines and ViewModel

Most of the Kotlin samples in this book that need to do background processing will do so using coroutines. We will have suspend functions that either call other suspend functions or do work in a designated thread or thread pool:

```
suspend fun doSomethingCool(): AwesomeMix = withContext(Dispatchers.IO) {  
    // TODO something that returns an AwesomeMix  
}
```

A suspend function needs to be called inside of a CoroutineScope. While Kotlin offers a GlobalScope that can be used for anything, ideally you use a scope that is more closely tied to the work that is being done.

In Android UI development, that frequently will be a CoroutineScope associated with a ViewModel. There is a viewModelScope extension property available to ViewModel that provides a CoroutineScope tied to the lifetime of the ViewModel. In particular, if the ViewModel is no longer being used (i.e., it is being called with onCleared()), the viewModelScope is also cleaned up.

So, you wind up with code like this:

```
fun doSomethingCoolAndGetTheResultsOnTheMainApplicationThread() {  
    viewModelScope.launch(Dispatchers.Main) {  
        val mix = repo.doSomethingCool()  
  
        // TODO something with mix on the main application thread  
    }  
}
```

(where repo is an object of the class that has the aforementioned doSomethingCool() function)

THINKING ABOUT THREADS AND LIVEDATA

`Dispatchers.Main` will mean that while `doSomethingCool()` will run on a background thread, we will get the result delivered to us on the main application thread.

You can learn more about Kotlin coroutines in [*Elements of Kotlin Coroutines*](#).

Adding Some Architecture

MVC. MVP. MVVM. MVI. These abbreviations get tossed around a lot in app development discussions, and increasingly in Android app development discussions. Those using these abbreviations often think that:

- Everybody knows what they mean, and
- There is a single universal definition for each of those abbreviations, one that everybody holds

In reality, these MV* abbreviations are well-known in some circles and unknown in others. And, even among people who think they know these abbreviations, there is a fair bit of disagreement about what the abbreviations mean, particularly when it comes time to writing actual code. In this chapter, we will explore what these abbreviations mean, with a particular emphasis on the last of the four: MVI, which stands for Model-View-Intent. And, as an illustration of the problems inherent in applying these abbreviations:

- The “model” in Model-View-Intent may or may not be what you might think of as the model
- The “view” in Model-View-Intent is unlikely to be a View
- The “intent” in Model-View-Intent is *not* going to be an Intent

Along the way, we will review the architecture pattern used by most of the remaining sample apps in the book.

GUI Architectures

MVC, MVP, MVVM, MVI, and others are GUI architecture patterns. They describe different ways of organizing your code to update your UI based upon user input and

ADDING SOME ARCHITECTURE

other changes in data, such as the results of server requests or database operations. The abbreviations, as you might expect, abbreviate short phrases that are the formal names of these patterns:

- MVC = Model-View-Controller
- MVP = Model-View-Presenter
- MVVM = Model-View-Viewmodel
- MVI = Model-View-Intent

In these, “model” represents some data, and “view” represents some way of visualizing that data. The trailing portion of the GUI architecture name indicates another component that is involved in taking model changes and updating the view, and taking user input (e.g., button clicks, text entry) and updating the models... which in turn updates views.

Some people may find it surprising that there are so many different organization patterns for this work, and that developers spend time debating the merits of one pattern over another. Debates over GUI architectures are reminiscent of debates over text editors (`emacs` versus `vi`), in that they tend to be debates over minutiae and ignore other options (e.g., Sublime Text 3).

Why People Worry About GUI Architectures

Those participating in these debates over GUI architectures will tend to unite against architecture-agnostics, hammering home their belief that you should have *some* formal GUI architecture in your app — the debate merely is over which one. There are some reasons why formal GUI architecture adherents are strident in their beliefs that such architectures are universally beneficial.

Avoiding Known Problems

GUI development can be tricky. Android GUI development can be trickier than others, with things like configuration changes, tasks, and the like adding lots of edge and corner cases to worry about.

If a popular GUI architecture has been applied to Android — and if those results have been published in a blog post, conference presentation, etc. — part of the work will be to address those tricky bits. Some of the points of contention in the GUI architecture debates are how they handle things like configuration changes, where architectures that handle those things gracefully are considered to be better than

ADDING SOME ARCHITECTURE

those that ignore the problem.

This is not to say that using one of these formal GUI architectures is a requirement to address these problem areas in Android. However, *if* you implement an Android version of one of these GUI architectures, there is a decent chance that these problem areas will be covered as part of that work.

Consistency Between Team Members

Many Android apps are developed by teams, rather than by solo developers. Team members may switch between working on different parts of the code base at different times, and so while certain areas may have a specific “owner”, that owner may not be the only one to work in those areas. For example, if a critical bug is discovered while an “owner” is away from the office, somebody else may need to step in and fix the bug, so that users are not harmed any longer than is necessary.

Having team members be able to work in any portion of an app’s code base is the reason for overall standards, such as:

- Tabs versus spaces for indents
- Line lengths
- Naming conventions (as opposed to `NamingConventions` or `namingConventions` or `naming_conventions`)
- And so on

At a higher level, teams may elect to standardize on a GUI architecture so that different developers writing different portions of a UI (e.g., different activities) will create similar code. It will be easier for others to adopt and modify that code if it is similar to other areas of the code that were seen previously.

Consistency Over Time

Teams are rarely stable for long. Team members come and go, within the overall organization or departing for greener pastures elsewhere. As such, “onboarding” of team members is important, and having a standardized approach to UI construction, in the form of a specific GUI architecture, can help here.

However, even solo developers are victims of time. Code that might make sense today may make less sense in a year and no sense at all in three years. Developer experience and expertise change, even when the code does not. Having consistent code within an app reduces this problem, in part by making it more likely that the

ADDING SOME ARCHITECTURE

developer will have seen similar code recently. And, adopting a specific GUI architecture means that all the world's prose on that GUI architecture act as documentation for the project that the developer does not have to write.

Why Others Ignore GUI Architectures

It is quite likely that the vast majority of Android apps do not use any of these GUI architectures.

In some cases, this is unintentional, insofar as the developer(s) of the app do not know about such architectures or have not considered them. Not everybody has equal education and experience, and that will be reflected in the tactical and strategic development decisions that they make or participate in making.

However, in some cases, there are clear reasons why a classic MV* GUI architecture is unsuitable.

Atypical Apps

These GUI architectures have been refined for “typical” app structures:

- You have some data (local or remote)
- You want to display that data to the user (in collections or for individual items)
- Often, the user can manipulate that data, adding to it, modifying it, or perhaps deleting it
- Sometimes, the data can change on the fly from outside of the app (e.g., push messages, other forms of server-centered updates)
- You want those data changes to flow back to the data source, plus update the UI of the app as needed

Lots of apps fit that general description. At the same time, lots of apps do not fit that general description, such as:

- Many types of games
- System-integrated tools, like soft keyboards and VPNs
- Musical instrument simulators
- Terminal emulators
- Camera apps
- Calculators

ADDING SOME ARCHITECTURE

- And so on

Some of these will have their own architecture patterns, perhaps tied to libraries or frameworks. Game development, for example, has its own approaches, often embodied in toolkits like Unity3D. Those approaches may not resemble the MV* architectures that you might use in, say, a social network client.

It is up to you to decide how well these sorts of GUI architectures fit your particular type of app.

YAGNI and Overhead

The bigger the app, the more likely it is that you will gain benefit from a formal GUI architecture. Such apps are more likely to have more developers and be used for more time, where a formal GUI architecture can yield benefits.

Conversely, the smaller the app, the less likely it is that a formal GUI architecture is necessarily worth it. Or, as the saying goes, “you aren’t going to need it” (YAGNI).

Most of the previous sample apps in this book skip the formalities. That is because they are samples of how to use particular APIs and usually have little code beyond that.

No Obvious User Benefit

The counterpart to this issue is that the user rarely, if ever, benefits directly from the use of a formal GUI architecture. Developers may benefit, and their organization (where relevant) may benefit. However, the user is not necessarily going to see anything different. A note-taking app, or a chat client, or a video player, should look the same to the user whether the developer(s) used MVC, MVP, MVVM, MVI, some other GUI architecture... or no specific GUI architecture at all.

It is entirely possible that bugs will exist in an informally-developed app that would not exist had the developers chosen and implemented a formal GUI architecture. However, there is no guarantee that bugs will exist in an informally-developed app. Similarly, there is no guarantee that using an MV* architecture will eliminate all of your bugs by magic. A development team that doesn’t pay much attention to GUI architecture — instead choosing to invest that time in an awesome test suite — may have a better app in the end than a team that focused heavily on the architecture and did a sub-par job at testing.

ADDING SOME ARCHITECTURE

Choosing a GUI architecture does not give the users any additional features. There are no new marketing buzzwords to tout. The media will not praise the app for its adherence to some GUI architecture. If management thinks that investing in a GUI architecture is slowing down “work that matters”, management might steer developers towards other efforts and away from a formal architecture. Frequently, decisions made on this basis turn out to be bad ones... but management is free to make those decisions.

No Consistency In Definitions

Compounding that latter problem is that it may take a while to figure out how to implement a GUI architecture, as they are *very* ill-defined. We toss around terms like “model” and “view” without clear definitions, particularly with respect to concrete scenarios and Android apps. As a result, not only does the development team need to debate which architecture to use, but also what that architecture really means in practice.

Repositories

One aspect that all Android GUI architecture implementations largely agree upon is the use of the repository pattern. Rather than the GUI code dealing with things like disk and network I/O, we delegate that to a repository, which handles those details. The GUI code just works with the repository, one typically implemented as a singleton object (a global instance that all pieces of the app can use).

Objective: Isolation

We really want our GUI code to be separate from our I/O code.

Mostly, that is for testing purposes. When we want to test our GUI code, we may or may not want to actually go through that I/O:

- I/O is slow, and being able to use some sort of mock repository that is purely memory-backed would allow tests to run faster
- Server I/O can be a problem, as there may not be a server that you can use for automated testing
- We often need to test specific scenarios, and it is much easier to do that if we can have a mock repository that we can control, rather than have to set up files (or, worse, server entries) to get the real repository to respond the way our scenario requires

Repository Structures

Simple repositories — like the ones that will be shown in this book — work directly with files, databases, servers, and the like.

Some projects will add another layer of indirection, where repositories work with “data sources”. A data source would be responsible for the direct I/O with one particular set of files, or a particular database, or a particular Web service. The repository would interact with the data source.

Mostly, the data source abstraction is there for complex repositories:

- You want to add in-memory caching at the repository level, and therefore you want to separate out the I/O logic into another class
- You need to support both local I/O (for a persistent cache) and network I/O (the source of shared or updated data), and putting all of that in a single repository results in a huge class
- The actual data source itself is user-configurable, both in terms of details (e.g., email account information) and protocol (e.g., IMAP4, JMAP, POP3), where the repository does not know at compile-time what data source implementation will be needed

So, while the apps in this book largely will skip the repository/data source separation of concerns, that is just because the sample apps are pretty tiny.

Unidirectional Data Flow

One of the GUI architectures listed above — Model-View-Intent — focuses on a [unidirectional data flow](#) pattern. [Popularized by Redux](#) in the world of Web app development, adopting a unidirectional data flow can help simplify how different bits of your app work with that data.

While the details will differ somewhat by implementation, the general approach of a unidirectional data flow (UDF) is to minimize the number of places in the app that can *change* data. In particular, while a user interface may allow the user to change data, the GUI code itself does not change data, but instead works off of a stream of immutable objects as a source of details for what to render in the GUI itself.

ADDING SOME ARCHITECTURE

In particular, the UDF approach that will be used in most of the remaining book samples will look a bit like this:

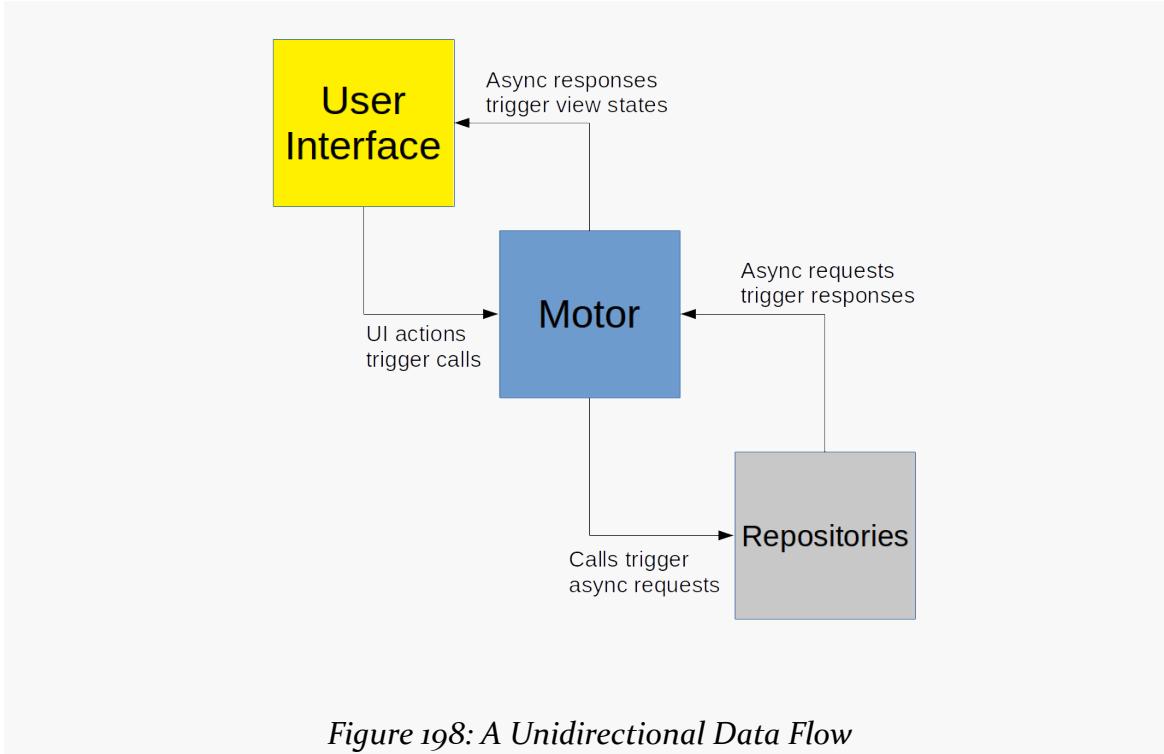


Figure 198: A Unidirectional Data Flow

In a nutshell:

- When a user does something in the UI, such as click a toolbar button, the UI calls a function on something called “the motor”
- In response to those calls, the motor asks the repository to read, update, or delete some data, using some sort of asynchronous mechanism
- The result of the repository work gets delivered to the motor via that asynchronous mechanism
- The motor emits a “view state” of details that get used by the UI to update what the user sees

A UDF Implementation

That UDF explanation may make more sense once we work through an example.

The DiceLight sample module in the [Sampler](#) and [SamplerJ](#) projects implement a

ADDING SOME ARCHITECTURE

“diceware” app. It allows the user to randomly generate a passphrase from a word list. Later, we will look at [a more complex Diceware sample](#) that allows the user to supply an alternative word list. For now, we will settle for having a word list packaged with the app.

The UI

When you launch the app, you are immediately greeted by a random passphrase:

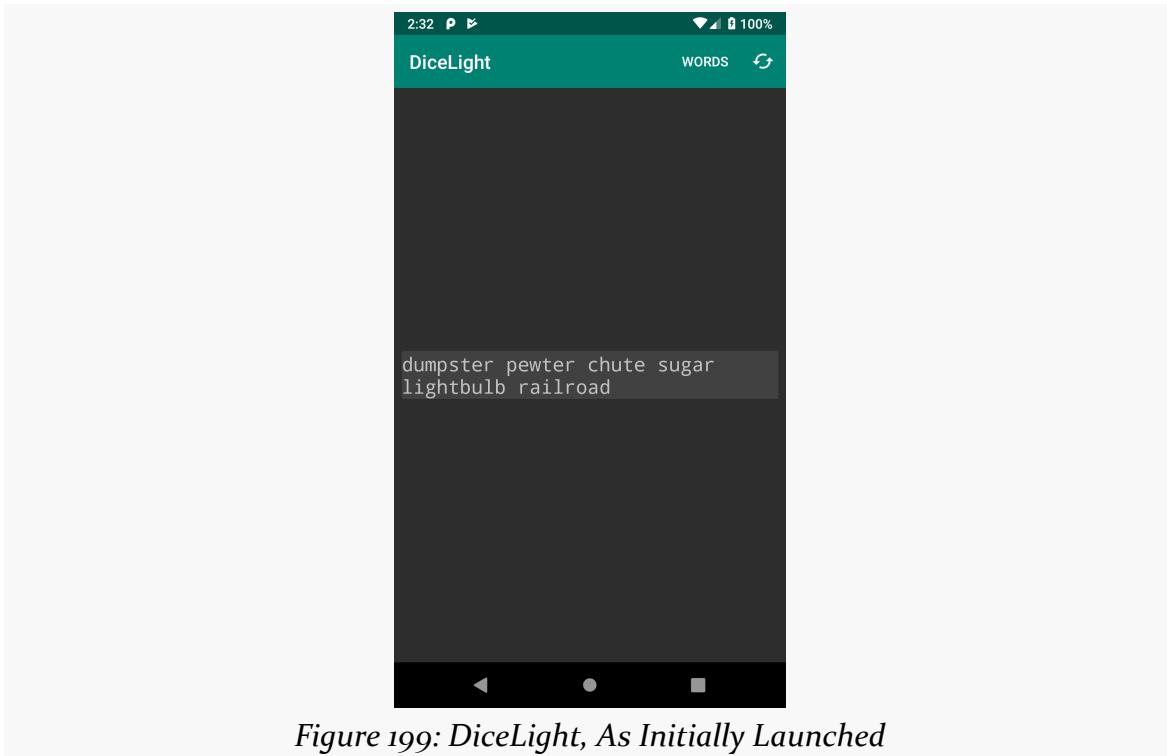


Figure 199: DiceLight, As Initially Launched

ADDING SOME ARCHITECTURE

The refresh toolbar button will generate a fresh passphrase, in case you do not like the original one. The “WORDS” item will display a list of word counts:

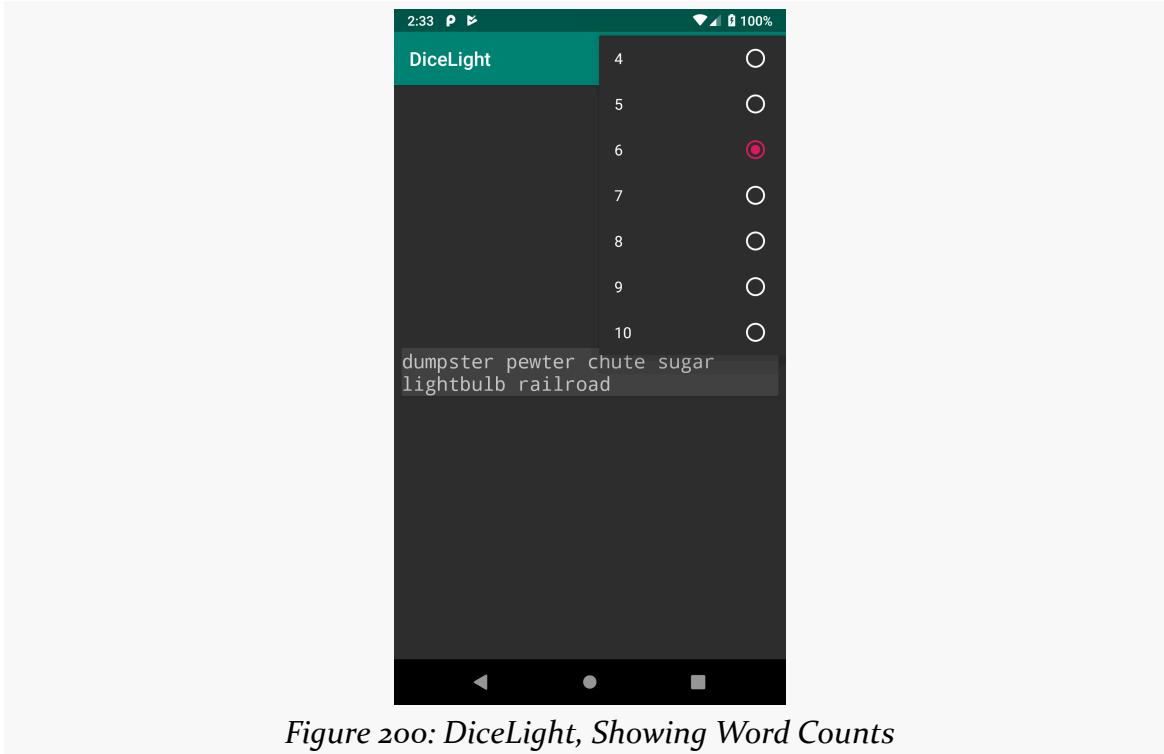


Figure 200: DiceLight, Showing Word Counts

Switching to a different word count will give you a fresh passphrase with that number of words, in case you want one that is shorter or longer.

The words come from [the EFF's short wordlist](#), where we randomly choose a few out of the 1,296 in the list. That word list is packaged in the assets/ directory, so it is part of the Android APK that is our compiled app.

The View-State

In order to render this UI, we need a randomly-generated passphrase. However, there are two other scenarios to consider:

- What happens while we are loading the words? Initially, they are not in memory, and until they are, we cannot randomly generate a passphrase from them. That disk I/O will be quick, but not instantaneous. The speed of network I/O is a lot worse, though in this case we do not have any of that. So, in theory, we should show something, such as a ProgressBar, while we

ADDING SOME ARCHITECTURE

are loading the words.

- What happens if we have some sort of problem loading the words? That should not happen here, as we know that our words are available in our APK. However, in general, disk I/O and network I/O can trigger exceptions. If we have such an exception, we really ought to tell the user about it.

So, now we have three pieces of data to track:

- Whether or not we are loading
- The current passphrase, if we have it
- The current error, if we have one

Our `MainViewState` encapsulates those three pieces of data... but how it does so varies by language.

Java

In Java, `MainViewState` is a simple class:

```
package com.commonsware.jetpack.diceware;

class MainViewState {
    final boolean isLoading;
    final String content;
    final Throwable error;

    MainViewState(boolean isLoading, String content, Throwable error) {
        this.isLoading = isLoading;
        this.content = content;
        this.error = error;
    }
}
```

(from [DiceLight/src/main/java/com/commonsware/jetpack/diceware/MainViewState.java](#))

We will use `null` to indicate that we do not have our passphrase (`content`) yet or do not have a current `error`.

Kotlin

That Java representation works, and it is simple, but it is not great.

Partly, the problem is that those three pieces of data are mutually exclusive to an

ADDING SOME ARCHITECTURE

extent:

- If we have our passphrase, we did not crash, so we will not have an error
- If we have an error, we will not have a passphrase
- If we are loading, we will not have either of those things

Plus, using `null` as a “we don’t have the data yet” value is annoying in Kotlin. You keep having to use safe calls (`?.`) and the like to deal with `null`.

In Kotlin, we take a substantially different approach, using a sealed class:

```
package com.commonsware.jetpack.diceware

sealed class MainViewState {
    object Loading : MainViewState()
    data class Content(val passphrase: String) : MainViewState()
    data class Error(val throwable: Throwable) : MainViewState()
}
```

(from [DiceLight/src/main/java/com/commonsware/jetpack/diceware/MainViewState.kt](#))

Now, any instance of `MainViewState` has just the data that is appropriate for it (e.g., the passphrase for `Content`). And, for a state that has no data — such as `Loading` — it is an `object`, rather than a `class`.

This loading/content/error pattern is increasingly common in Android, as a way of representing the three major states.

The Repository

So, we need to create those states and get them to the activity, so the activity can do something with that information to display in the UI. A lot of that work is going to be handled by our repository.

The `PassphraseRepository` is responsible for loading the words and generating a random subset for us on demand. The core logic is the same between the Java and Kotlin implementations, but the API is different, owing to different ways in handling background work and singletons.

Java

Our `PassphraseRepository` is set up as a classic Java singleton, using a `static`

ADDING SOME ARCHITECTURE

volatile field and a synchronized, lazy-initializing get() method:

```
package com.commonsware.jetpack.diceware;

import android.content.Context;
import android.content.res.AssetManager;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.security.SecureRandom;
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.atomic.AtomicReference;

class PassphraseRepository {
    private static final String ASSET_FILENAME = "eff_short_wordlist_2_0.txt";
    private static volatile PassphraseRepository INSTANCE;

    synchronized static PassphraseRepository get(Context context) {
        if (INSTANCE == null) {
            INSTANCE = new PassphraseRepository(context.getApplicationContext());
        }

        return INSTANCE;
    }

    private final AssetManager assets;
    private final AtomicReference<List<String>> wordsCache = new AtomicReference<>();
    private final SecureRandom random = new SecureRandom();

    private PassphraseRepository(Context context) {
        assets = context.getAssets();
    }

    List<String> generate(int wordCount) throws Exception {
        List<String> words = wordsCache.get();

        if (words == null) {
            InputStream in = assets.open(PassphraseRepository.ASSET_FILENAME);

            words = readWords(in);
            in.close();
        }

        synchronized (wordsCache) {
            wordsCache.set(words);
        }
    }

    return rollDemBones(words, wordCount);
}

private List<String> rollDemBones(List<String> words, int wordCount) {
    List<String> result = new ArrayList<>();
    int size = words.size();

    for (int i = 0; i < wordCount; i++) {
        result.add(words.get(random.nextInt(size)));
    }
}
```

ADDING SOME ARCHITECTURE

```
    return result;
}

private List<String> readWords(InputStream in) throws IOException {
    InputStreamReader isr = new InputStreamReader(in);
    BufferedReader reader = new BufferedReader(isr);
    String line;
    List<String> result = new ArrayList<>();

    while ((line = reader.readLine()) != null) {
        String[] pieces = line.split("\s");

        if (pieces.length == 2) {
            result.add(pieces[1]);
        }
    }

    return result;
}
}
```

(from [DiceLight/src/main/java/com/commonsware/jetpack/diceware/PassphraseRepository.java](#))

`get()` takes a Context as a parameter. We need a Context to be able to read in our word list stored in assets. However, we do not want to hold onto an arbitrary Context in this singleton — if the Context is an Activity, we will wind up with a memory leak. So, we hold onto the Application singleton edition of Context instead.

The singleton has a single visible method: `generate()`. Given a word count, `generate()` returns a list of randomly-chosen words, or it throws an exception if we run into some sort of I/O problem. As a result, `generate()` is a synchronous API, relying on its caller to manage a background thread.

The repository holds onto the read-in words in a cache, wrapped by an `AtomicReference` to ensure that we handle possible parallel access to the cache across multiple threads. `generate()` will:

- See if we have cached the words
- If not, read in the words and save them in the cache
- Randomly choose the requested number of words, returning that List

The word list contains a “die roll” value and a word for each line, separated by tabs, such as:

```
1154  again
1155  agency
1156  aggressor
1161  aghast
```

ADDING SOME ARCHITECTURE

```
1162 agitate
1163 agnostic
```

The `readWords()` method needs to split each line along the whitespace and take the second part.

Kotlin

The Kotlin implementation does the same work, with three key differences:

1. Kotlin offers an `object` keyword for creating singletons, so we use that rather than manage our own singleton
2. Since such singlettons cannot have a constructor, we pass the `Context` into `generate()`, whereas Java supplied it to the repository constructor
3. `generate()` is a suspend function, doing the work on a background thread, and we can still return our result list or throw an exception

```
package com.commonsware.jetpack.diceware

import android.content.Context
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.withContext
import java.io.BufferedReader
import java.io.InputStream
import java.io.InputStreamReader
import java.security.SecureRandom
import java.util.concurrent.atomic.AtomicReference

private const val ASSET_FILENAME = "eff_short_wordlist_2_0.txt"

object PassphraseRepository {
    private val wordsCache = AtomicReference<List<String>>()
    private val random = SecureRandom()

    suspend fun generate(context: Context, count: Int): List<String> {
        val words: List<String>? = wordsCache.get()

        return words?.let { rollDemBones(it, count) }
            ?: loadAndGenerate(context, count)
    }

    private suspend fun loadAndGenerate(
        context: Context,
        count: Int
    ): List<String> =
```

ADDING SOME ARCHITECTURE

```
withContext(Dispatchers.IO) {
    val inputStream = context.assets.open(ASSET_FILENAME)

    inputStream.use {
        val words = it.readLines()
            .map { line -> line.split("\t") }
            .filter { pieces -> pieces.size == 2 }
            .map { pieces -> pieces[1] }

        wordsCache.set(words)

        rollDemBones(words, count)
    }
}

private fun rollDemBones(words: List<String>, wordCount: Int) =
    List(wordCount) { words[random.nextInt(words.size)] }

private fun InputStream.readLines(): List<String> {
    val result = mutableListOf<String>()

    BufferedReader(InputStreamReader(this)).forEachLine { result.add(it); }

    return result
}
```

(from [DiceLight/src/main/java/com/commonsware/jetpack/diceware/PassphraseRepository.kt](#))

As a result, our repository is a bit simpler (no singleton code, plus lots of Kotlin functions to make it easier to read in and process the words). And, since we are handling the background thread in the repository, consumers of this repository can be a bit simpler as well.

The Motor

In Android, we have a problem with our UIs: they keep getting destroyed, courtesy of configuration changes. We saw how we can use a `ViewModel` for retaining objects across configuration changes, and we saw how we can have a `ViewModel` expose `LiveData` objects to the UI layer to deliver data updates.

A motor is simply a `ViewModel` exposing `LiveData`.

The reason for the “motor” name (e.g., `MainMotor`) instead of a “viewmodel” name (e.g., `ViewModel1`) comes from other GUI architectures. In MVVM (Model-View-

ADDING SOME ARCHITECTURE

Viewmodel) and MVP (Model-View-Presenter), what gets referred to as the “viewmodel” fills the sort of role that we have here as the view-state: it is the data to be displayed in the UI. However, in Android’s Architecture Components, just because we use a `ViewModel` (for configuration changes) does not mean we want that object to serve as a viewmodel (representing the data to be displayed). So, this book uses “motor” to identify a `ViewModel` that exposes a `LiveData` of view-state objects.

And, as with our repository, the motor is a bit different between Java and Kotlin, principally due to the way threading is handled.

Java

In Java, our repository has a synchronous API. Since that API involves I/O, we need to get its work onto a background thread. So, `MainMotor` uses a single-thread Executor for that in its `generatePassphrase()` method:

```
package com.commonsware.jetpack.diceware;

import android.app.Application;
import android.text.TextUtils;
import java.util.concurrent.Executor;
import java.util.concurrent.Executors;
import androidx.annotation.NonNull;
import androidx.lifecycle.AndroidViewModel;
import androidx.lifecycle.LiveData;
import androidx.lifecycle.MutableLiveData;

public class MainMotor extends AndroidViewModel {
    private final MutableLiveData<MainViewState> viewStates = new MutableLiveData<>();
    private final PassphraseRepository repo;
    private final Executor executor = Executors.newSingleThreadExecutor();

    public MainMotor(@NonNull Application application) {
        super(application);

        repo = PassphraseRepository.get(application);
    }

    LiveData<MainViewState> getViewStates() {
        return viewStates;
    }

    void generatePassphrase(int wordCount) {
        viewStates.setValue(new MainViewState(true, null, null));

        executor.execute(() -> {
            try {
                viewStates.postValue(new MainViewState(false, TextUtils.join(" ", repo.generate(wordCount)),
null));
            }
            catch (Exception e) {
                viewStates.postValue(new MainViewState(false, null, e));
            }
        });
    }
}
```

ADDING SOME ARCHITECTURE

```
    }
  });
}
```

(from [DiceLight/src/main/java/com/commonsware/jetpack/diceware/MainMotor.java](#))

MainMotor has a MutableLiveData for our MainViewState, exposing it to the UI layer via a getViewStates() method returning a LiveData. MainMotor also has reference to the PassphraseRepository singleton. And, since we need to supply a Context to the repository, MainMotor extends AndroidViewModel, so we have a getApplication() method to retrieve the Application singleton Context.

The generatePassphrase() method first emits a view-state indicating that we are loading. Then, in the Executor-supplied background thread, we:

- Call generate() on the repository, given the supplied word count
- Use Android's TextUtils.join() method to convert the list of words into a single space-delimited passphrase
- Emit a view-state with that result, or emit a view-state with the exception if we ran into a problem

Kotlin

In Kotlin, since our repository exposes a suspend function, we use viewModelScope to launch our coroutine, receiving the results on the main application thread:

```
package com.commonsware.jetpack.diceware

import android.app.Application
import androidx.lifecycle.AndroidViewModel
import androidx.lifecycle.LiveData
import androidx.lifecycle.MutableLiveData
import androidx.lifecycle.viewModelScope
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.launch

class MainMotor(application: Application) : AndroidViewModel(application) {
  private val _results = MutableLiveData<MainViewState>()
  val results: LiveData<MainViewState> = _results

  fun generatePassphrase(wordCount: Int) {
    _results.value = MainViewState.Loading

    viewModelScope.launch(Dispatchers.Main) {
```

ADDING SOME ARCHITECTURE

```
_results.value = try {
    val randomWords = PassphraseRepository.generate(
        getApplication(),
        wordCount
    )

    MainViewState.Content(randomWords.joinToString(" "))
} catch (t: Throwable) {
    MainViewState.Error(t)
}
}
```

(from [DiceLight/src/main/java/com/commonsware/jetpack/diceware/MainMotor.kt](#))

We also:

- Use the PassphraseRepository singleton supplied by the object declaration
- Use Kotlin's own `joinToString()` instead of Android's `TextUtils.join()` to convert the list of words into the passphrase
- Use our sealed class implementations, so we are emitting some sub-type of `MainViewState`

The Activity

Our layout is based on a `CardView`: a widget that is a simple rounded rectangle with a bit of a drop shadow. Inside of there, we have a `TextView` and a `ProgressBar`:

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="8dp">

    <androidx.cardview.widget.CardView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:padding="8dp">

        <TextView
            android:id="@+id/passphrase"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_gravity="center"
            android:freezesText="true"
            android:textSize="20sp" />
    
```

ADDING SOME ARCHITECTURE

```
        android:typeface="monospace" />

    <ProgressBar
        android:id="@+id/progress"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
    </androidx.cardview.widget.CardView>

</FrameLayout>
```

(from [DiceLight/src/main/res/layout/activity_main.xml](#))

Managing that layout is a simple activity. We could have used a fragment here, but there is only one screen. In general, the remaining sample apps will use fragments and the Navigation component when there is more than one screen, but just a single Activity otherwise.

In `onCreate()`, we get our `MainMotor`, start observing the `LiveData`, apply the `MainViewState` to our widgets, and call `generatePassphrase()` to start the work for getting our initial passphrase:

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

    motor = ViewModelProviders.of(this).get(MainMotor.class);

    final View progress = findViewById(R.id.progress);
    final TextView passphrase = findViewById(R.id.passphrase);

    motor.getViewStates().observe(this, viewState -> {
        progress.setVisibility(
            viewState.isLoading ? View.VISIBLE : View.GONE);

        if (viewState.content != null) {
            passphrase.setText(viewState.content);
        }
        else if (viewState.error != null) {
            passphrase.setText(viewState.error.getLocalizedMessage());
            Log.e("Diceware", "Exception generating passphrase",
                viewState.error);
        }
        else {
            passphrase.setText("");
        }
    });
}
```

ADDING SOME ARCHITECTURE

```
    motor.generatePassphrase(wordCount);
}
```

(from [DiceLight/src/main/java/com/commonsware/jetpack/diceware/MainActivity.java](#))

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

    motor = ViewModelProviders.of(this)[MainMotor::class.java]

    motor.results.observe(this, Observer { viewState ->
        when (viewState) {
            MainViewState.Loading -> {
                progress.visibility = View.VISIBLE
                passphrase.text = ""
            }
            is MainViewState.Content -> {
                progress.visibility = View.GONE
                passphrase.text = viewState.passphrase
            }
            is MainViewState.Error -> {
                progress.visibility = View.GONE
                passphrase.text = viewState.throwable.localizedMessage
                Log.e(
                    "Diceware",
                    "Exception generating passphrase",
                    viewState.throwable
                )
            }
        }
    })
}

motor.generatePassphrase(wordCount)
}
```

(from [DiceLight/src/main/java/com/commonsware/jetpack/diceware/MainActivity.kt](#))

We also have a menu resource for the word count options and the refresh button:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android"
      xmlns:app="http://schemas.android.com/apk/res-auto">
    <item
        android:id="@+id/word_count"
        app:showAsAction="ifRoom"
        android:title="@string/menu_words">
```

ADDING SOME ARCHITECTURE

```
<menu>
    <group android:checkableBehavior="single">
        <item
            android:id="@+id/word_count_4"
            android:title="4" />
        <item
            android:id="@+id/word_count_5"
            android:title="5" />
        <item
            android:id="@+id/word_count_6"
            android:checked="true"
            android:title="6" />
        <item
            android:id="@+id/word_count_7"
            android:title="7" />
        <item
            android:id="@+id/word_count_8"
            android:title="8" />
        <item
            android:id="@+id/word_count_9"
            android:title="9" />
        <item
            android:id="@+id/word_count_10"
            android:title="10" />
    </group>
</menu>
</item>
<item
    android:id="@+id/refresh"
    android:icon="@drawable/ic_cached_white_24dp"
    app:showAsAction="ifRoom"
    android:title="@string/menu_refresh" />

</menu>
```

(from [DiceLight/src/main/res/menu/actions.xml](#))

The word_count menu item is a bit unusual, following the Android recipe for creating such a selection menu:

- The `<item>` has a `<menu>` child...
- ...which in turn holds a `<group>` with `android:checkableBehavior="single"`...
- ... which wraps a set of `<item>` elements, one for each checkable menu item...
- ... with `android:checked="true"` on the `word_count_6` item, to pre-check that one

ADDING SOME ARCHITECTURE

In MainActivity, our wordCount is initialized to 6 to match the menu resource:

```
private int wordCount = 6;
```

(from [DiceLight/src/main/java/com/commonsware/jetpack/diceware/MainActivity.java](#))

```
private var wordCount = 6
```

(from [DiceLight/src/main/java/com/commonsware/jetpack/diceware/MainActivity.kt](#))

We then have code in onCreateOptionsMenu() and onOptionsItemSelected() to set up and handle that menu:

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.actions, menu);

    return super.onCreateOptionsMenu(menu);
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case R.id.refresh:
            motor.generatePassphrase(wordCount);
            return true;

        case R.id.word_count_4:
        case R.id.word_count_5:
        case R.id.word_count_6:
        case R.id.word_count_7:
        case R.id.word_count_8:
        case R.id.word_count_9:
        case R.id.word_count_10:
            item.setChecked(!item.isChecked());

            int temp = Integer.parseInt(item.getTitle().toString());

            if (temp != wordCount) {
                wordCount = temp;
                motor.generatePassphrase(wordCount);
            }

            return true;
    }

    return super.onOptionsItemSelected(item);
}
```

ADDING SOME ARCHITECTURE

```
}
```

(from [DiceLight/src/main/java/com/commonsware/jetpack/diceware/MainActivity.java](#))

```
override fun onCreateOptionsMenu(menu: Menu): Boolean {
    menuInflater.inflate(R.menu.actions, menu)

    return super.onCreateOptionsMenu(menu)
}

override fun onOptionsItemSelected(item: MenuItem): Boolean {
    when (item.itemId) {
        R.id.refresh -> {
            motor.generatePassphrase(wordCount)
            return true
        }

        R.id.word_count_4, R.id.word_count_5, R.id.word_count_6, R.id.word_count_7,
        R.id.word_count_8, R.id.word_count_9, R.id.word_count_10 -> {
            item.isChecked = !item.isChecked

            val temp = Integer.parseInt(item.title.toString())

            if (temp != wordCount) {
                wordCount = temp
                motor.generatePassphrase(wordCount)
            }
        }

        return true
    }
}

return super.onOptionsItemSelected(item)
}
```

(from [DiceLight/src/main/java/com/commonsware/jetpack/diceware/MainActivity.kt](#))

The refresh item is simple: we just call `generatePassphrase()` again, causing the motor to get a fresh set of words from the repository and emitting another `MainViewState` to update our UI.

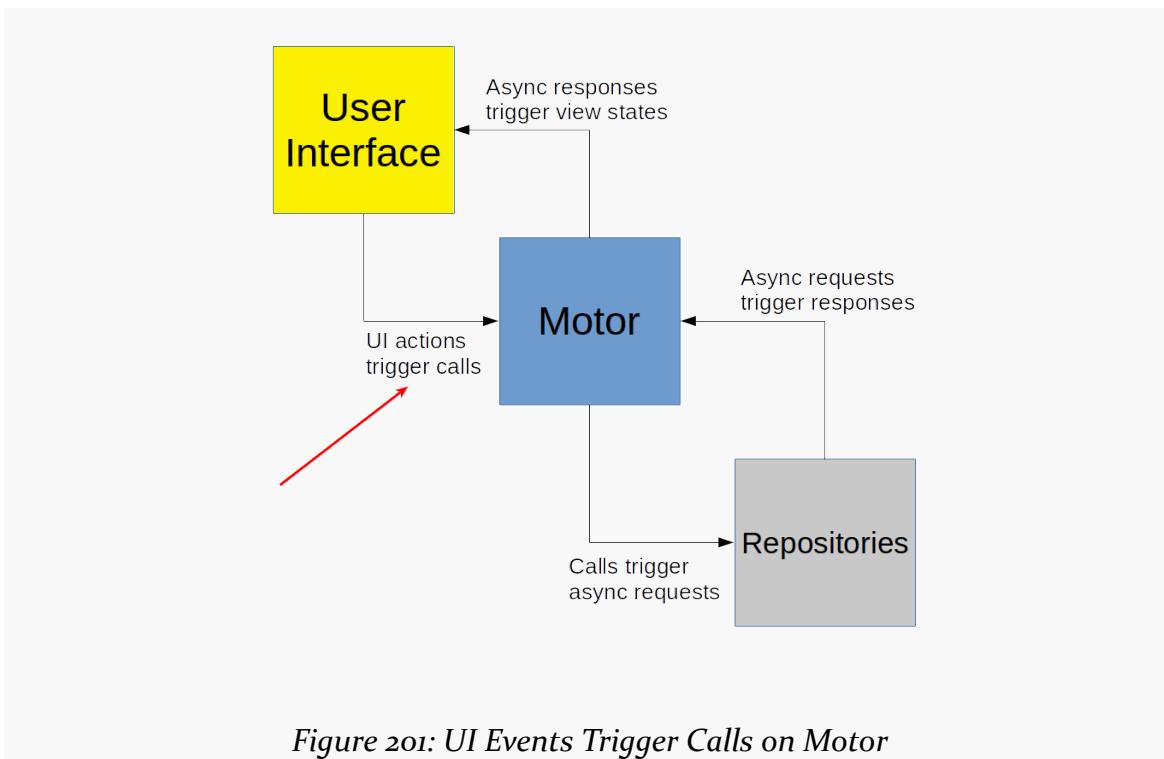
For the word count, we have more work to do. First, we need to toggle the `isChecked` state of the `MenuItem`, because Android (inexplicably) does not handle that for us when the user clicks a checkable menu item. Then, we cheat a bit and parse the actual text of the menu item as an `Integer` — this only works because this app is English-only, so we know that the menu item captions can be parsed by `Integer.parseInt()`. We then update the `wordCount` property and call `generatePassphrase()` to generate a passphrase with the new number of words.

Revisiting the Unidirectional Data Flow

Our activity takes user input (activity launch, refresh, word-count change) and

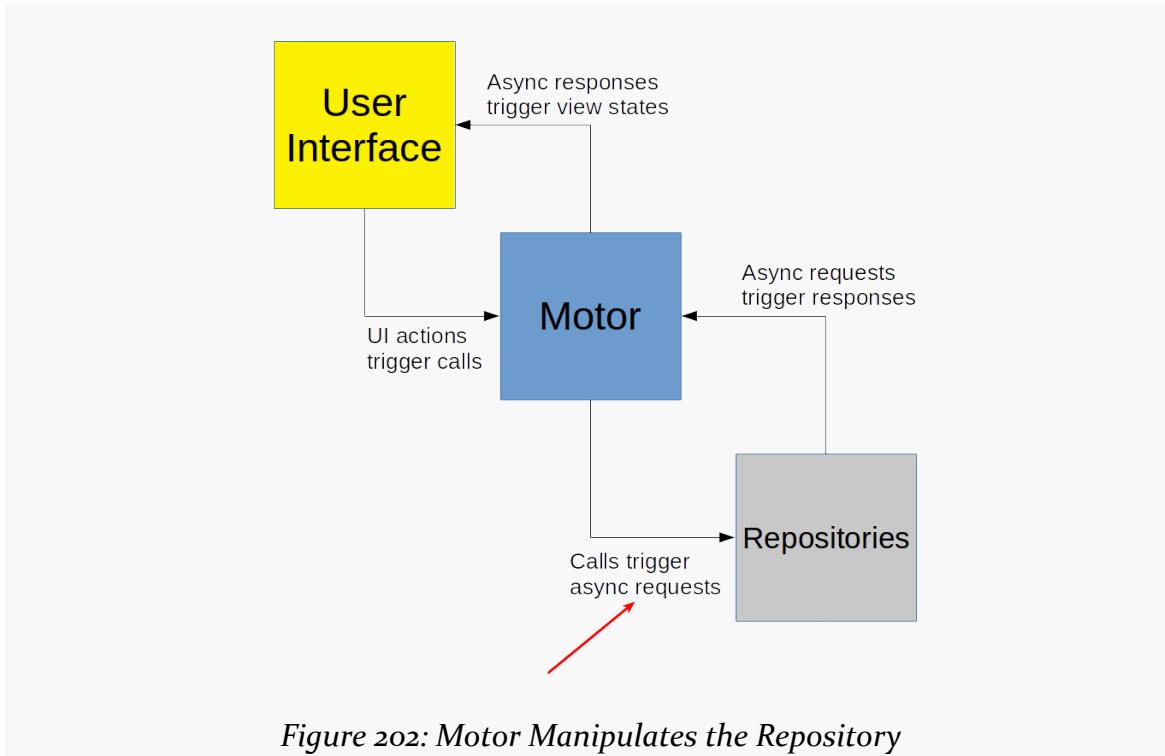
ADDING SOME ARCHITECTURE

converts those into function calls on the motor:



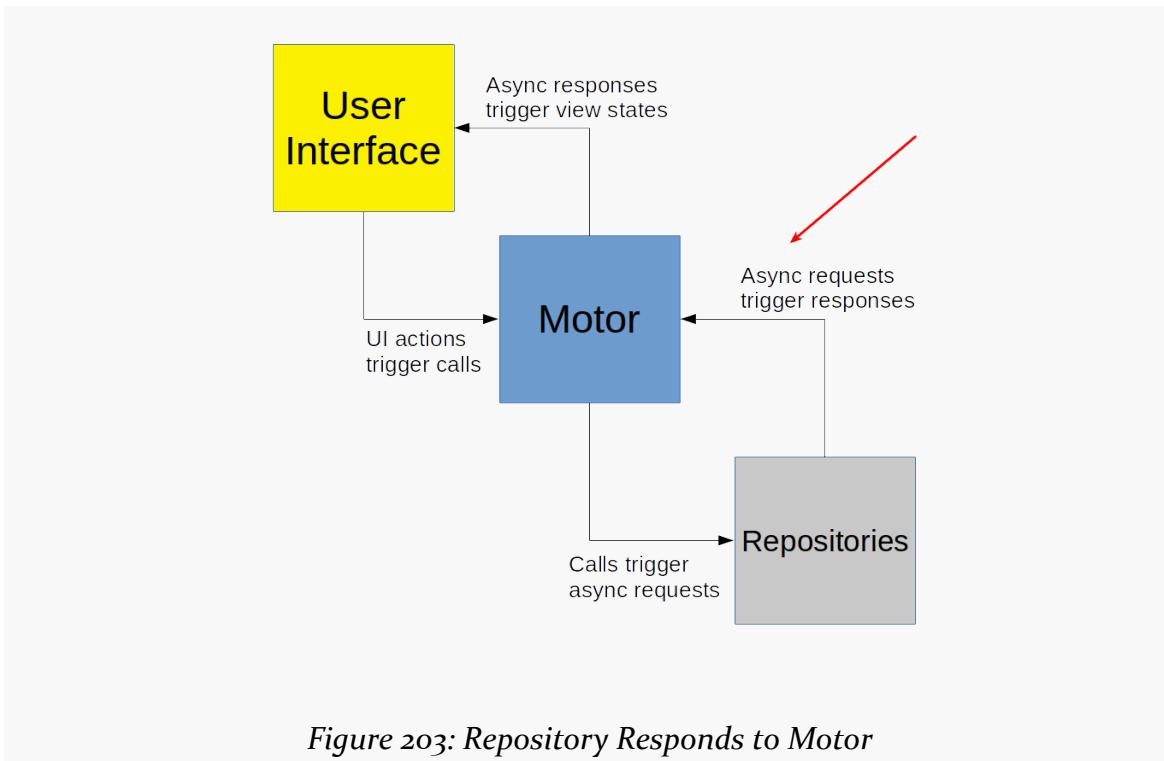
ADDING SOME ARCHITECTURE

The motor then converts those calls into operations to be performed on our repository, in an asynchronous fashion:



ADDING SOME ARCHITECTURE

The repository does the work and asynchronously delivers the result to the motor:



ADDING SOME ARCHITECTURE

The motor takes those results and crafts a revised view-state that the activity can use to render the UI:

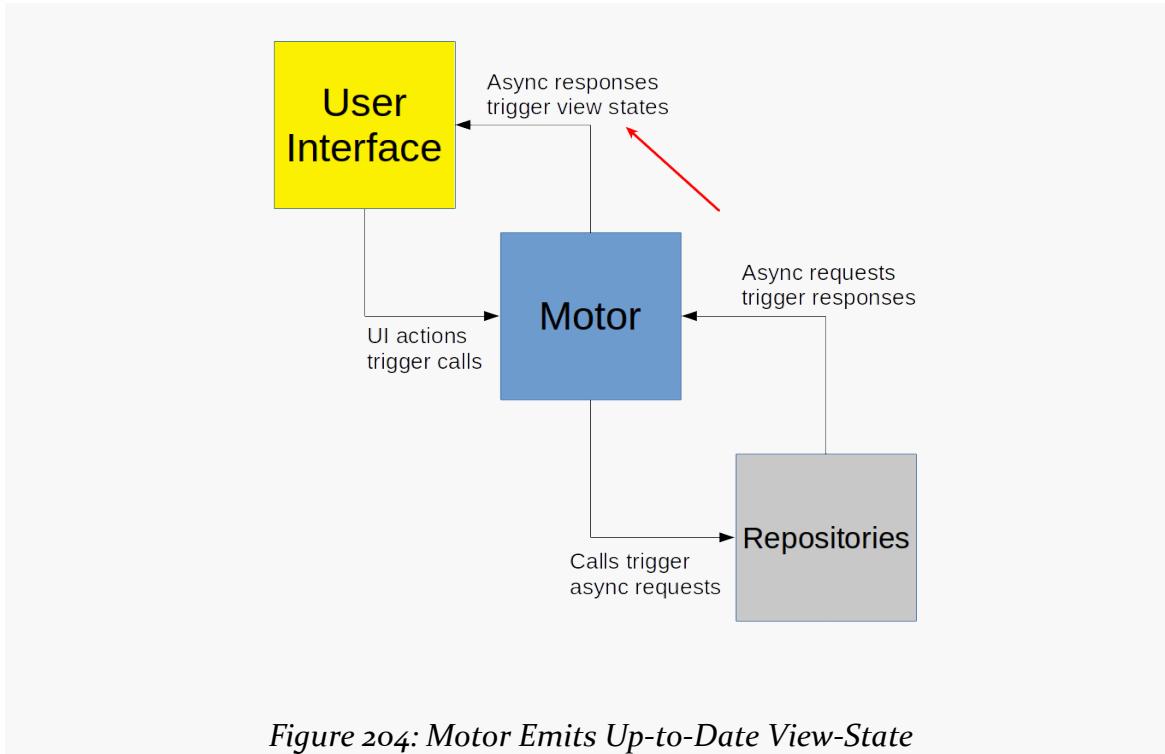


Figure 204: Motor Emits Up-to-Date View-State

Obviously, this is a *very simple* example. There are plenty of MVI frameworks, such as Spotify's [Mobius](#), that provide a scaffold around this sort of flow that you can plug into. However, in many cases, that level of sophistication (and corresponding complexity) is not needed, and a simple motor-based flow like the one shown here can suffice.

Working with Content

If you have written software for other platforms, you are used to working with files on the filesystem. Android has some support for that, but it is limited and awkward — we will explore that more [in an upcoming chapter](#).

Instead, Google would like for us to work with content more generally, whether that content comes from files, from services like Google Drive, or other places. To that end, Android comes with the Storage Access Framework for creating and consuming content.

The Storage Access Framework

Let's think about photos for a minute.

A person might have photos managed as:

- on-device photos, mediated by an app like a gallery
- photos stored online in a photo-specific service, like Instagram
- photos stored online in a generic file-storage service, like Google Drive or Dropbox

Now, let's suppose that person is in an app that allows the user to pick a photo, such as to attach to an email or to include in an MMS message.

Somehow, that email or text messaging client needs to allow the user to choose a photo. Some developers attempt to do this by looking for photo files on the filesystem, but that will miss lots of photos, particularly on newer versions of Android. Some developers will use a class called `MediaStore` to query for available photos. That is a reasonable choice, but `MediaStore` only allows you to query for

WORKING WITH CONTENT

certain types of content, such as photos. If the user wants to attach a PDF to the email, MediaStore is not a great solution.

The Storage Access Framework is designed to address these issues. It provides its own “picker” UI to allow users to find a file of interest that matches the MIME type that the client app wants. Document providers simply publish details about their available files — including those that may not be on the device but could be retrieved if needed. The picker UI allows for easy browsing and searching across all possible document providers, to streamline the process for the user. And, since Android is the one providing the picker, the picker should more reliably give a result to the client app based upon the user’s selection (if any).

More Dice!

The DiceWare modules of the [Java](#) and [Kotlin](#) projects are based on the DiceLight sample that we saw [previously](#). The difference is that in this version of the app, there is an overflow menu with “Open Word File”, where the user can choose a different word list to use as the source of words for the passphrases. To let the user choose the word list, we will use the Storage Access Framework.

The SAF Actions

From a programming standpoint, the Storage Access Framework feels like the “file open”, “file save-as”, and “choose directory” dialogs that you may be used to from other GUI environments. The biggest difference is that the Storage Access Framework is not limited to files.

These three bits of UI are tied to three Intent actions:

Action	Equivalent Role
ACTION_OPEN_DOCUMENT	file open
ACTION_CREATE_DOCUMENT	file save-as
ACTION_OPEN_DOCUMENT_TREE	choose directory

Those three Intent actions are designed for use with `startActivityForResult()`. In `onActivityResult()`, if we got a result, that will contain a `Uri` that points to a document (for `ACTION_OPEN_DOCUMENT` and `ACTION_CREATE_DOCUMENT`) or document

WORKING WITH CONTENT

tree (for ACTION_CREATE_DOCUMENT). We can then use that Uri to write content, read in existing content, or find child documents in a tree.

Opening a Document

Technically, we do not “open” a document using ACTION_OPEN_DOCUMENT. Instead, we are requesting a Uri pointing to some document that the user chooses.

To do that, create an Intent with:

- ACTION_OPEN_DOCUMENT as the action
- CATEGORY_OPENABLE as the category
- your desired MIME type, including wildcards if appropriate

Then, use that Intent with startActivityForResult().

In the Diceware sample, we call startActivityForResult() with that Intent when the user chooses the open menu item:

```
case R.id.open:  
    Intent i =  
        new Intent()  
            .setType("text/plain")  
            .setAction(Intent.ACTION_OPEN_DOCUMENT)  
            .addCategory(Intent.CATEGORY_OPENABLE);  
  
    startActivityForResult(i, REQUEST_OPEN);  
    return true;
```

(from [Diceware/src/main/java/com/commonsware/jetpack/diceware/MainActivity.java](#))

```
R.id.open -> {  
    val i = Intent()  
        .setType("text/plain")  
        .setAction(Intent.ACTION_OPEN_DOCUMENT)  
        .addCategory(Intent.CATEGORY_OPENABLE)  
  
    startActivityForResult(i, REQUEST_OPEN)  
    return true  
}
```

(from [Diceware/src/main/java/com/commonsware/jetpack/diceware/MainActivity.kt](#))

ACTION_OPEN_DOCUMENT should supply a Uri in the result Intent that points to the document the user chose, if the user actually chose one and we got RESULT_OK as the

WORKING WITH CONTENT

result code. In our case, we pass that to MainMotor, asking it to generate a passphrase for us, using that Uri and the user's current requested word count:

```
@Override  
protected void onActivityResult(int requestCode, int resultCode,  
    @Nullable Intent data) {  
    if (requestCode == REQUEST_OPEN) {  
        if (resultCode == RESULT_OK && data != null) {  
            motor.generatePassphrase(data.getData(), wordCount);  
        }  
    }  
}
```

(from [Diceware/src/main/java/com/commonsware/jetpack/diceware/MainActivity.java](#))

```
override fun onActivityResult(  
    requestCode: Int, resultCode: Int,  
    data: Intent?  
) {  
    if (requestCode == REQUEST_OPEN) {  
        if (resultCode == Activity.RESULT_OK && data != null) {  
            data.data?.let { motor.generatePassphrase(it, wordCount) }  
        }  
    }  
}
```

(from [Diceware/src/main/java/com/commonsware/jetpack/diceware/MainActivity.kt](#))

WORKING WITH CONTENT

The user is presented with the system's ACTION_OPEN_DOCUMENT UI to browse among various places and choose a document:

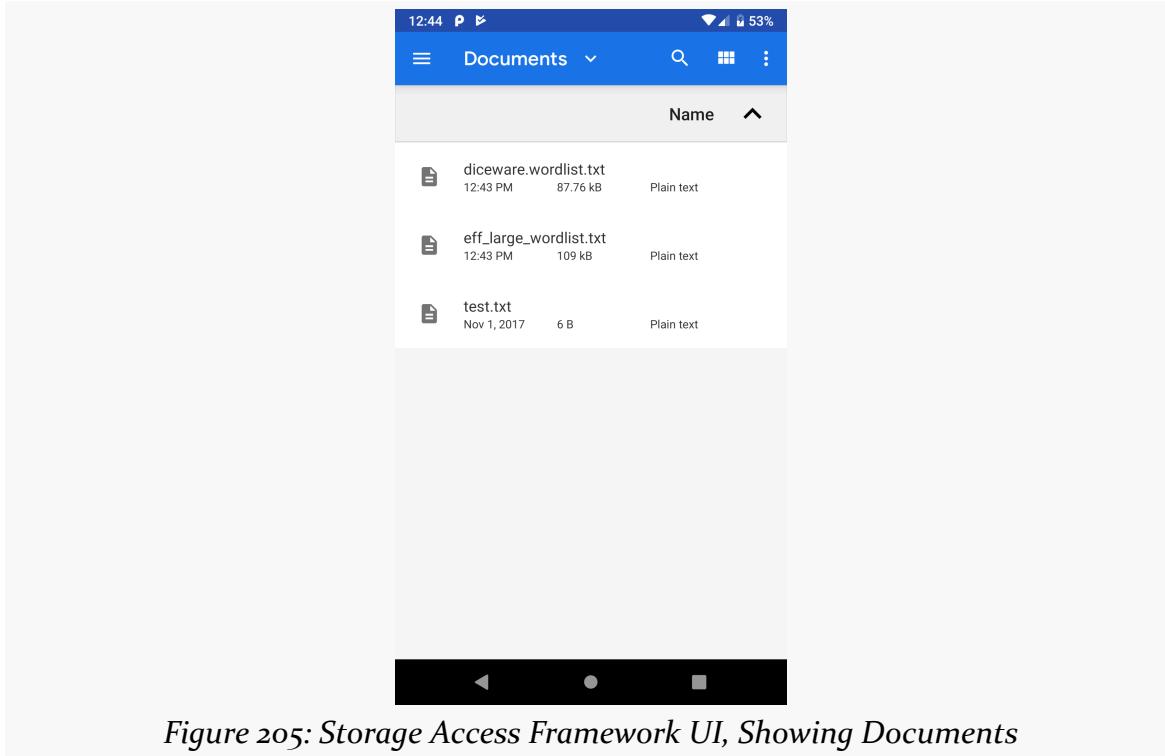


Figure 205: Storage Access Framework UI, Showing Documents

If you want to try this yourself, you can download the [diceware.wordlist.txt](#) or [eff_large_wordlist.txt](#) word list files, put them on your test device, then open them up within the Diceware app.

Why We Want Things To Be Openable

You will notice that the ACTION_OPEN_DOCUMENT Intent created in the Diceware sample has CATEGORY_OPENABLE applied to it. This is supposed to guarantee that we can actually consume the content represented by the Uri that we get. In particular, we should be able to use a ContentResolver and open streams on that content.

If we leave off CATEGORY_OPENABLE, it is possible that we will get a Uri that we cannot open ourselves.

The difference boils down to what use we intend to put the Uri toward:

- If all we plan to do is use that Uri for is to wrap it in an ACTION_VIEW Intent

WORKING WITH CONTENT

and start an activity on it, you could skip CATEGORY_OPENABLE and perhaps offer more choices to your user

- Otherwise, use CATEGORY_OPENABLE

Since we used CATEGORY_OPENABLE, we can try to open an InputStream on the content.

Consuming the Chosen Content

MainMotor is largely the same as before. However, in load(), it passes the Uri along to generate() on PassphraseRepository:

```
package com.commonsware.jetpack.diceware;

import android.app.Application;
import android.net.Uri;
import android.text.TextUtils;
import java.util.concurrent.Executor;
import java.util.concurrent.Executors;
import androidx.annotation.NonNull;
import androidx.lifecycle.AndroidViewModel;
import androidx.lifecycle.MutableLiveData;

public class MainMotor extends AndroidViewModel {
    private final PassphraseRepository repo;
    private final Executor executor = Executors.newSingleThreadExecutor();
    private Uri wordsDoc = PassphraseRepository.ASSET_URI;
    final MutableLiveData<MainViewState> viewStates =
        new MutableLiveData<>();

    public MainMotor(@NonNull Application application) {
        super(application);

        repo = PassphraseRepository.get(application);
    }

    void generatePassphrase(int wordCount) {
        viewStates.setValue(new MainViewState(true, null, null));

        executor.execute(() -> {
            try {
                viewStates.postValue(new MainViewState(false,
                    TextUtils.join(" ", repo.generate(wordsDoc, wordCount)), null));
            }
            catch (Exception e) {
                viewStates.postValue(new MainViewState(false, null, e));
            }
        });
    }
}
```

WORKING WITH CONTENT

```
    }
});

}

void generatePassphrase(Uri wordsDoc, int wordCount) {
    this.wordsDoc = wordsDoc;

    generatePassphrase(wordCount);
}
}
```

(from [Diceware/src/main/java/com/commonsware/jetpack/diceware/MainMotor.java](#))

```
package com.commonsware.jetpack.diceware

import android.app.Application
import android.net.Uri
import androidx.lifecycle.*
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.launch

class MainMotor(application: Application) : AndroidViewModel(application) {
    private val _results = MutableLiveData<MainViewState>()
    val results: LiveData<MainViewState> = _results
    private var wordsDoc = ASSET_URI

    fun generatePassphrase(wordCount: Int) {
        _results.value = MainViewState.Loading

        viewModelScope.launch(Dispatchers.Main) {
            _results.value = try {
                val randomWords = PassphraseRepository.generate(
                    getApplication(),
                    wordsDoc,
                    wordCount
                )

                MainViewState.Content(randomWords.joinToString(" "))
            } catch (t: Throwable) {
                MainViewState.Error(t)
            }
        }
    }

    fun generatePassphrase(wordsDoc: Uri, wordCount: Int) {
        this.wordsDoc = wordsDoc

        generatePassphrase(wordCount)
    }
}
```

WORKING WITH CONTENT

```
}
```

(from [Diceware/src/main/java/com/commonsware/jetpack/diceware/MainMotor.kt](#))

For our initial passphrase, though, we start off with a Uri that points to our asset:

```
static final Uri ASSET_URI =
Uri.parse("file:///android_asset/eff_short_wordlist_2_0.txt");
```

(from [Diceware/src/main/java/com/commonsware/jetpack/diceware/PassphraseRepository.java](#))

```
val ASSET_URI: Uri =
Uri.parse("file:///android_asset/eff_short_wordlist_2_0.txt")
```

(from [Diceware/src/main/java/com/commonsware/jetpack/diceware/PassphraseRepository.kt](#))

In PassphraseRepository, our words cache now is an LruCache. This is an Android SDK class, representing a thread-safe Map that caps its size to a certain number of entries. If we try putting more things in the cache than we have room for, the LruCache will evict the least-recently-used (“LRU”) entry. In our case, the cache is keyed by a Uri and is capped to at most four word lists:

```
private final LruCache<Uri, List<String>> wordsCache = new LruCache<>(4);
```

(from [Diceware/src/main/java/com/commonsware/jetpack/diceware/PassphraseRepository.java](#))

```
private val wordsCache = LruCache<Uri, List<String>>(4)
```

(from [Diceware/src/main/java/com/commonsware/jetpack/diceware/PassphraseRepository.kt](#))

Then, our revised generate() function gets the cached word list by examining our cache using the supplied Uri and proceeds from there:

```
List<String> generate(Uri wordsDoc, int wordCount) throws Exception {
    List<String> words;

    synchronized (wordsCache) {
        words = wordsCache.get(wordsDoc);
    }

    if (words == null) {
        InputStream in;

        if (wordsDoc.equals(ASSET_URI)) {
            in = assets.open(PassphraseRepository.ASSET_FILENAME);
        }
    }
}
```

WORKING WITH CONTENT

```
    else {
        in = resolver.openInputStream(wordsDoc);
    }

    words = readWords(in);
    in.close();

    synchronized (wordsCache) {
        wordsCache.put(wordsDoc, words);
    }
}

return rollDemBones(words, wordCount);
}
```

(from [Diceware/src/main/java/com/commonsware/jetpack/diceware/PassphraseRepository.java](#))

```
suspend fun generate(
    context: Context,
    wordsDoc: Uri,
    count: Int
): List<String> {
    var words: List<String>?

    synchronized(wordsCache) {
        words = wordsCache.get(wordsDoc)
    }

    return words?.let { rollDemBones(it, count, random) } ?: loadAndGenerate(context, wordsDoc, count)
}

private suspend fun loadAndGenerate(
    context: Context,
    wordsDoc: Uri,
    count: Int
): List<String> = withContext(Dispatchers.IO) {
    val inputStream: InputStream? = if (wordsDoc == ASSET_URI) {
        context.assets.open(ASSET_FILENAME)
    } else {
        context.contentResolver.openInputStream(wordsDoc)
    }

    inputStream?.use {
        val words = it.readLines()
            .map { line -> line.split("\t") }
            .filter { pieces -> pieces.size == 2 }
            .map { pieces -> pieces[1] }
    }
}
```

WORKING WITH CONTENT

```
synchronized(wordsCache) {  
    wordsCache.put(wordsDoc, words)  
}  
  
rollDemBones(words, count, random)  
} ?: throw IllegalStateException("could not open $wordsDoc")  
}
```

(from [Diceware/src/main/java/com/commonsware/jetpack/diceware/PassphraseRepository.kt](#))

If we do not have our words yet, we need to load them. In DiceLight, we would always open the asset. Now, we see if the asset is the magic ASSET_URI, and we only use AssetManager and its open() function if that is the case. Otherwise, we use a ContentResolver and openInputStream() to get the content identified by the Uri that we got from ACTION_OPEN_DOCUMENT. You can get a ContentResolver by calling getContentResolver() on a Context, such as the one supplied to the repository constructor (Java) or generate() function (Kotlin).

The rest of PassphraseRepository is the same, as the rest of the code for loading words does not care whether the InputStream came from an asset or came from the ContentResolver.

DocumentFile and the Rest of the CRUD

ACTION_OPEN_DOCUMENT will give you a Uri for a document that you can open for reading — the “R” in “CRUD”, as we saw in Diceware. The Storage Access Framework also supports the remaining operations: create, update, and delete.

To help you with these operations, the Jetpack offers a DocumentFile class, which provides convenience functions for finding out key details about the Uri that you received. For ACTION_OPEN_DOCUMENT and ACTION_CREATE_DOCUMENT, you can get a DocumentFile for a Uri by calling DocumentFile.fromSingleUri(), passing in that Uri. DocumentFile then has functions like getType() to tell you the MIME type associated with that particular piece of content.

Create

ACTION_CREATE_DOCUMENT will give you a Uri for a document that you can open for writing, as it is your document.

To do this, construct an Intent with:

WORKING WITH CONTENT

- an action of ACTION_CREATE_DOCUMENT
- a category of CATEGORY_OPENABLE
- the MIME type of the content you wish to write
- an extra, named EXTRA_TITLE, containing your desired filename or other “display name” — this does not have to be a classic filename with an extension

Then, invoke `startActivityForResult()` on that Intent, and use the Uri supplied in the result Intent delivered to `onActivityResult()`. For example, you can use `ContentResolver` and `openOutputStream()` to get an `OutputStream` that lets you write data to the user-chosen location.

Note, though, that the user has the right to replace your proposed title with something else. You can find out the title for a Uri by calling `getName()` on a `DocumentFile`. Again, bear in mind that this does not have to have a classic filename structure. In particular, it does not need to have a file extension.

Update

The Uri returned from an ACTION_OPEN_DOCUMENT request may be writable. You can find out by calling `canWrite()` on a `DocumentFile` for the Uri. If that returns true, you can use `openOutputStream()` on a `ContentResolver` to write to that document.

Delete

If you can write to the content, you can also delete it. To do that, call `delete()` on a `DocumentFile` for that Uri.

Getting Durable Access

By default, you will have the rights to read (and optionally write) to the document represented by the Uri until the activity that requested the document via ACTION_OPEN_DOCUMENT or ACTION_CREATE_DOCUMENT is destroyed.

If you pass the Uri to another component — such as another activity — you will need to add `FLAG_GRANT_READ_URI_PERMISSION` and/or `FLAG_GRANT_WRITE_URI_PERMISSION` to the Intent used to start that component. That extends your access until *that* component is destroyed. Note that fragments are all considered to be a part of the activity that created them, so you do not need to worry about extending rights from one fragment to another.

WORKING WITH CONTENT

If, however, you need the rights to survive your app restarting, you can call `takePersistableUriPermission()` on a `ContentResolver`, indicating the `Uri` of the document and the permissions (`FLAG_GRANT_READ_URI_PERMISSION` and/or `FLAG_GRANT_WRITE_URI_PERMISSION`) that you want persisted. Then, you can save the `Uri` somewhere — such as in `SharedPreferences`, which we will explore [in an upcoming chapter](#). Later, when your app runs again, you can get the `Uri` and probably still use it with `ContentResolver` and `DocumentFile`, even for a completely new activity or completely new process. Those rights even survive a reboot.

However, those rights will not survive the document being deleted or moved by some other app. You can call `exists()` on a `DocumentFile` to see if your `Uri` still points to a document that exists.

In addition, you can call `getPersistedUriPermissions()` to find out what persisted permissions your app has. This returns a `List` of `UriPermission` objects, where each one of those represents a `Uri`, what persisted permissions (read or write) you have, and when the permissions will expire.

Document Trees

`ACTION_OPEN_DOCUMENT` and `ACTION_CREATE_DOCUMENT` are sufficient for most apps.

However, there may be cases where you need the equivalent of a “choose directory” dialog, to allow the user to pick a location where you can create (or work with) several documents. For example, suppose that your app offers a report generator, taking data from the database and creating a report with tables and graphs and stuff. Some file formats, like PDF, might have the entire report in a single file — for that, use `ACTION_CREATE_DOCUMENT` to allow the user to choose where to put that report. Other file formats, like HTML, might require several files (e.g., the report body in HTML and embedded graphs in PNG format). For that, you really need a “directory”, into which you can create all of those individual bits of content.

For that, the Storage Access Framework offers document trees.

Getting a Tree

Instead of using `ACTION_OPEN_DOCUMENT`, you can use `ACTION_OPEN_DOCUMENT_TREE`. Once again, you will use `startActivityForResult()` to request access to the tree. In `onActivityResult()`, the result Intent has a `Uri` (`getData()`) that represents the tree. You should have full read/write access not only to this tree but to anything

inside of it.

Working in the Tree

The simplest approach for then working with the tree is to use the aforementioned `DocumentFile` wrapper. You can create one representing the tree by using the `fromTreeUri()` static method, passing in the `Uri` that you got from the `ACTION_OPEN_DOCUMENT_TREE` request.

From there, you can:

- Call `listFiles()` to get the immediate children of the root of this tree, getting back an array of `DocumentFile` objects representing those children
- Call `isDirectory()` to confirm that you do indeed have a tree (or, call it on a child to see if that child represents a sub-tree)
- For those existing children that are files (`isFile()` returns `true`), use `getUri()` to get the `Uri` for this child, so you can read its contents using a `ContentResolver` and `openInputStream()`
- Call `createDirectory()` or `createFile()` to add new content as an immediate child of this tree, getting a `DocumentFile` as a result
- For the `createFile()` scenario, call `getUri()` on the `DocumentFile` to get a `Uri` that you can use for writing out the content using `ContentResolver` and `openOutputStream()`
- and so on

Note that you can call `takePersistableUriPermission()` on a `ContentResolver` to try to have durable access to the document tree, just as you can for a `Uri` to an individual document.

Licensed solely for use by Patrocinio Rodriguez

Using Preferences

Android allows apps to keep preferences, in the form of key/value pairs (akin to a Map), that will persist between invocations of an activity. As the name suggests, the primary purpose is for you to store user-specified configuration details, such as the last feed the user looked at in your feed reader, or what sort order to use by default on a list, or whatever. Of course, you can store in the preferences whatever you like, so long as it is keyed by a String and has a rudimentary value (boolean, String, etc.)

The principal use of SharedPreferences, though, is for storing data that is collected by a preference UI. Android provides a system for producing a UI that looks a lot like the individual screens of the Settings app. That UI is fairly easy to set up and is a recommended solution for configuration options and related settings.

In this chapter, we will examine how to set up a preference UI and work with SharedPreferences

The Preferred Preferences

The good news is that there is only one SharedPreferences class.

The bad news is that there are a few implementations of the preference UI. The native one — classes in android.preference — has been deprecated. Instead, we are supposed to use the Support Library implementation or the AndroidX implementation.

The SimplePrefs sample module in the [Sampler](#) and [SamplerJ](#) projects use the AndroidX implementation, in the form of the androidx.preference:preference library. This sample app offers a preference UI (PrefsFragment) and a regular UI that

shows the current value of the user-supplied preferences (`HomeFragment`).

Collecting Preferences with `PreferenceFragmentCompat`

Some “preferences” will be collected as part of the natural use of your user interface. For example, if you have a `SeekBar` widget to control a zoom level, you might elect to record the `SeekBar` position in `SharedPreferences`, so you can restore the user’s last zoom level later on.

However, in many cases, we have various settings that we would like the user to be able to configure but are not something that the user would configure elsewhere in our UI. For that, typically we use preference XML resources and a `PreferenceFragmentCompat`.

Defining Your Preferences

First, you need to tell Android what preferences you are trying to collect from the user.

To do this, you will need to add a `res/xml/` directory to your project, if one does not already exist. Then, you will define an XML resource file that describes the preferences that you want. The root element of this XML file will be `<PreferenceScreen>`, and it will contain child elements, generally one per preference.

In the sample project, we have one such file, `res/xml/preferences.xml`:

```
<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android">

    <CheckBoxPreference
        android:key="checkbox"
        android:summary="@string/checkSummary"
        android:title="@string/checkTitle"/>

    <EditTextPreference
        android:dialogTitle="@string/dialogTitle"
        android:key="field"
        android:summary="@string/fieldSummary"
        android:title="@string/fieldTitle"/>

    <ListPreference
```

USING PREFERENCES

```
    android:dialogTitle="@string/listDialogTitle"
    android:entries="@array/cities"
    android:entryValues="@array/airportCodes"
    android:key="list"
    android:summary="@string/listSummary"
    android:title="@string/listTitle"/>

</PreferenceScreen>
```

(from [SimplePrefs/src/main/res/xml/preferences.xml](#))

If you open up that resource in Android Studio, you will be given an editor that is reminiscent of the layout resource editor. You will have two sub-tabs: “Text” with the XML and “Design” with a drag-and-drop UI:

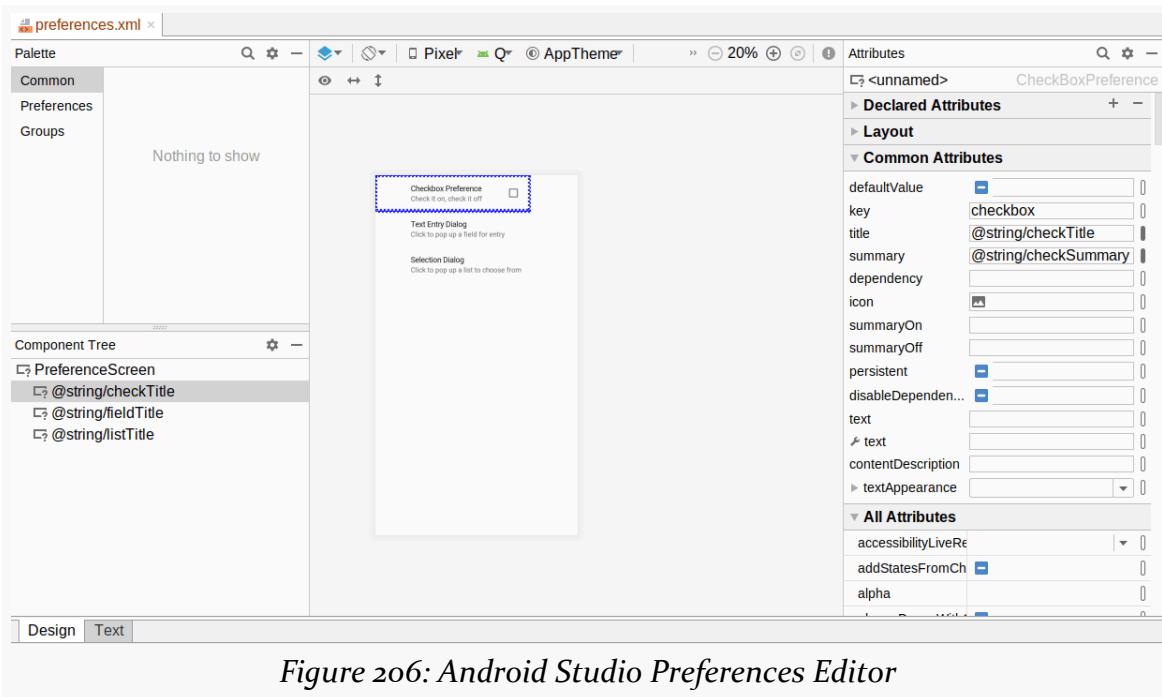


Figure 206: Android Studio Preferences Editor

The drag-and-drop editor UI works akin to its layout resource editor counterpart. You can drag a preference from the Palette into either the preview area or into the Component Tree to add it to the resource. For any selected preference, the Attributes pane allows you to modify attributes, either from the default short list of popular properties or the full list of properties that you get from clicking “View all properties”.

As with widgets in a layout resource, the element names of the preferences reflect a

USING PREFERENCES

Java class that is the implementation of that preference. Our preference XML has `CheckBoxPreference`, `EditTextPreference`, and `ListPreference` elements, so our UI will be constructed from those classes. Note that these are *not* widgets — they do not extend from `View` — so you cannot use them directly in a layout resource.

Each preference element has two attributes at minimum:

1. `android:key`, which is the key you use to look up the value in the `SharedPreferences` object via methods like `getInt()`
2. `android:title`, which is a few words identifying this preference to the user

You may also wish to consider having `android:summary`, which is a short sentence explaining what the user is to supply for this preference.

There are lots of other attributes that are common to all preference elements, and there are more types of preference elements than the ones that we used in the preference XML shown above. We will examine these preference elements and others like them [later in this chapter](#).

Creating Your Preference Fragment

We use preference XML resources with a `PreferenceFragmentCompat` class. This is a type of fragment that knows:

- How to load preference XML, inflating it into the actual Java objects
- How to render the UI for those preferences
- How to populate that UI with the current preference values
- How to update the preference values based on user input

Typically, a `PreferenceFragmentCompat` subclass just overrides `onCreatePreferences()` and calls `addPreferencesFromResource()`:

```
package com.commonsware.jetpack.simpleprefs;

import android.os.Bundle;
import androidx.preference.PreferenceFragmentCompat;

public class PrefsFragment extends PreferenceFragmentCompat {
    @Override
    public void onCreatePreferences(Bundle savedInstanceState, String rootKey) {
        addPreferencesFromResource(R.xml.preferences);
    }
}
```

USING PREFERENCES

(from [SimplePrefs/src/main/java/com/commonsware/jetpack/simpleprefs/PrefsFragment.java](#))

```
package com.commonsware.jetpack.simpleprefs

import android.os.Bundle
import androidx.preference.PreferenceFragmentCompat

class PrefsFragment : PreferenceFragmentCompat() {
    override fun onCreatePreferences(
        savedInstanceState: Bundle?,
        rootKey: String?
    ) {
        addPreferencesFromResource(R.xml.preferences)
    }
}
```

(from [SimplePrefs/src/main/java/com/commonsware/jetpack/simpleprefs/PrefsFragment.kt](#))

Otherwise, this is an ordinary Fragment. We can start it using a `FragmentTransaction` or the Navigation component, as we see fit. In this sample, we use the Navigation component, linking a `HomeFragment` to the `PrefsFragment`:

```
<?xml version="1.0" encoding="utf-8"?>
<navigation android:id="@+id/nav_graph"
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:label="@string/app_name"
    app:startDestination="@+id/homeFragment">

    <fragment
        android:id="@+id/homeFragment"
        android:name="com.commonsware.jetpack.simpleprefs.HomeFragment"
        android:label="@string/app_name">
        <action
            android:id="@+id/editPrefs"
            app:destination="@+id/prefsFragment" />
    </fragment>
    <fragment
        android:id="@+id/prefsFragment"
        android:name="com.commonsware.jetpack.simpleprefs.PrefsFragment"
        android:label="@string/app_name" />
</navigation>
```

(from [SimplePrefs/src/main/res/navigation/nav_graph.xml](#))

USING PREFERENCES

The UI

If you run the app and click the “Edit Preferences” button, you will be taken to `PrefsFragment` and the UI that it creates:

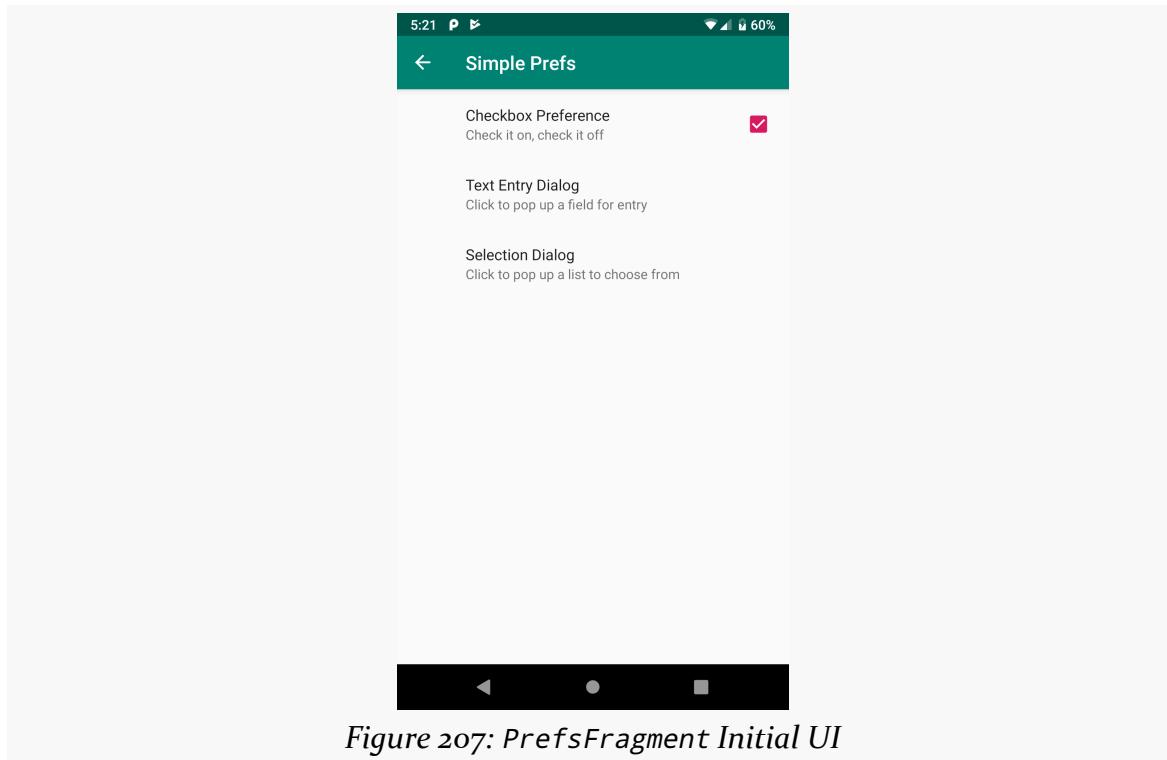


Figure 207: `PrefsFragment` Initial UI

`CheckBoxPreference` is an “inline” preference: the user can set the value from a widget right in the preference itself. In the case of `CheckBoxPreference`, that is in the form of a `CheckBox` widget.

USING PREFERENCES

Our other two preferences are dialog preferences, where the user taps on the preference to bring up a dialog where the user sets the value:

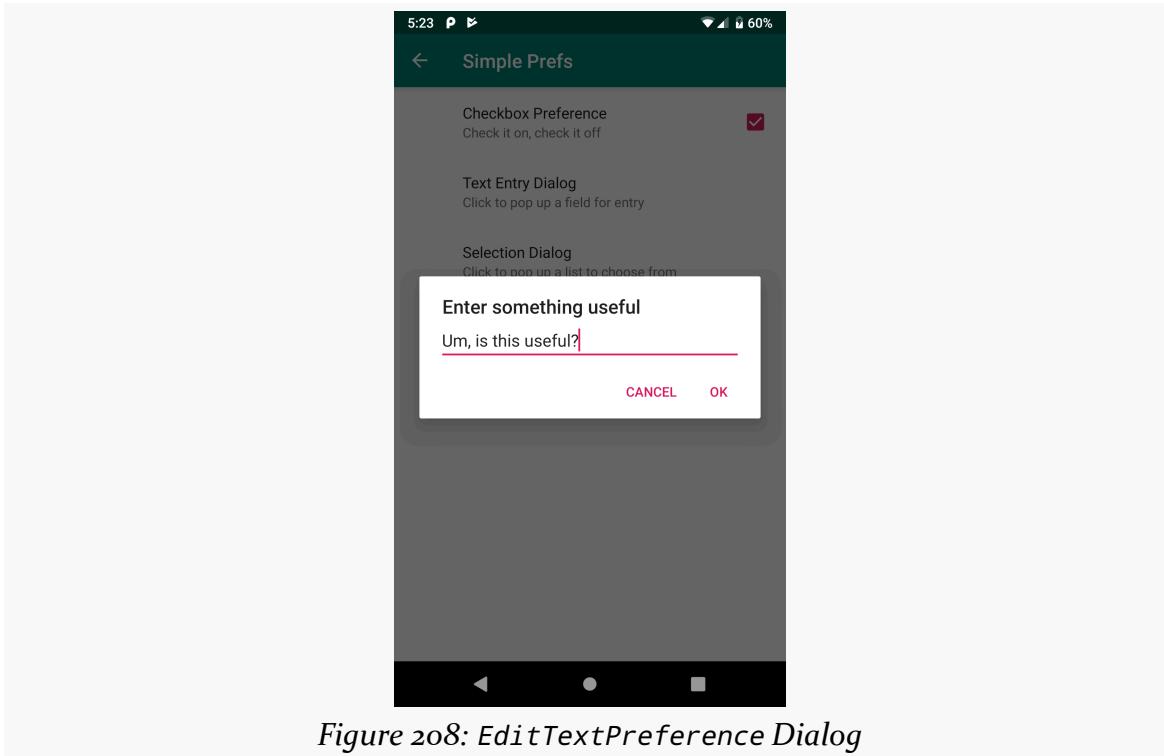


Figure 208: *EditTextPreference Dialog*

USING PREFERENCES

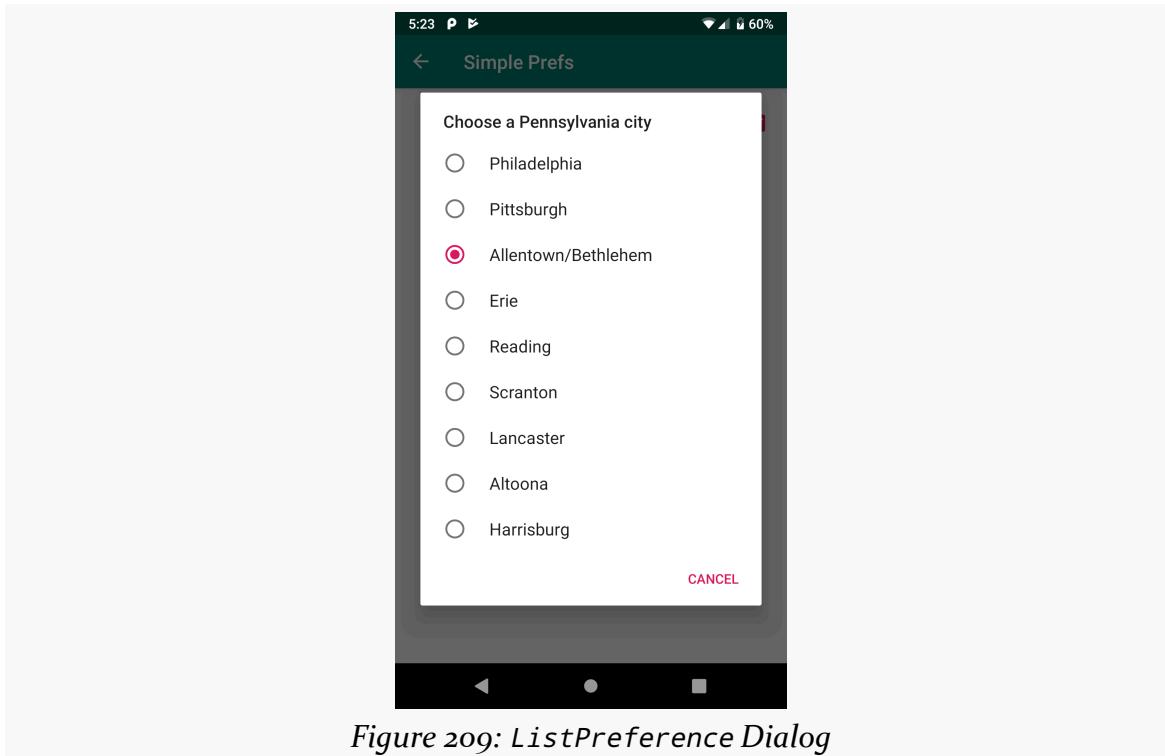


Figure 209: ListPreference Dialog

Each value is saved in the `SharedPreferences` as the user changes it. There is no “save” action that the user needs to do in order to save the changes.

Types of Preferences

The elements in the preference XML will refer to subclasses of `androidx.preference.Preference`. There are several of these in the current AndroidX Preference library, and most may be added in the future. You can also create your own, such as by extending `DialogPreference` and calling various methods to build up the content of the dialog (e.g., `setDialogTitle()`, `setDialogLayoutResource()`).

CheckBoxPreference and SwitchPreference

The sample application shown above a `CheckBoxPreference`. As noted, a `CheckBoxPreference` is an “inline” preference, in that the widget the user interacts with (in this case, a `CheckBox`) is part of the preference screen itself, rather than contained in a separate dialog.

USING PREFERENCES

`SwitchPreference` is functionally equivalent to `CheckBoxPreference`, insofar as both collect boolean values from the user. The difference is that `SwitchPreference` uses a `Switch` widget that the user slides left and right to toggle between “on” and “off” states.

The value that will be stored in the `SharedPreferences` is a boolean — we will explore how to read and manipulate these values from your own code [later in this chapter](#).

EditTextPreference

`EditTextPreference`, when tapped by the user, pops up a dialog that contains an `EditText` widget. You can configure this widget via attributes on the `<EditTextPreference>` element — in addition to standard preference attributes like `android:key`, you can include any attribute understood by `EditText`, such as `android:inputType`. Also, as the sample app shows, you can have `android:DialogTitle` to provide the title for the dialog that wraps the `EditText` widget.

The value stored in the `SharedPreferences` is a string.

ListPreference and MultiSelectListPreference

A `ListPreference` displays a dialog with your choice of entries. Each is accompanied by a `RadioButton`, with the checked `RadioButton` indicating the current value (if any). A `MultiSelectListPreference` has the same look, except it has checkboxes for each entry, and the user can choose multiple values, not just one.

To configure what appears in the list, you provide two attributes in the `ListPreference` or `MultiSelectListPreference` element:

- `android:entries` provides what the user sees
- `android:entryValues` provides the corresponding values that are saved in the `SharedPreferences`

In the preference XML, these attributes need to point to `string-array` resources. String resources hold individual strings; string array resources hold a collection of strings. Typically, you will find string array resources in `res/values/arrays.xml` and related resource sets for translation. The `<string-array>` element has the `name` attribute to identify the resource, along with child `<item>` elements for the individual strings in the array.

USING PREFERENCES

So, our sample app has a pair of <string-array> resources in `res/values/arrays.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string-array name="cities">
        <item>Philadelphia</item>
        <item>Pittsburgh</item>
        <item>Allentown/Bethlehem</item>
        <item>Erie</item>
        <item>Reading</item>
        <item>Scranton</item>
        <item>Lancaster</item>
        <item>Altoona</item>
        <item>Harrisburg</item>
    </string-array>
    <string-array name="airportCodes">
        <item>PHL</item>
        <item>PIT</item>
        <item>ABE</item>
        <item>ERI</item>
        <item>RDG</item>
        <item>AVP</item>
        <item>LNS</item>
        <item>A00</item>
        <item>MDT</item>
    </string-array>
</resources>
```

(from [SimplePrefs/src/main/res/values/arrays.xml](#))

Here, the actual strings are written in-line. They could just as easily be references to string resource (e.g., `[item]>@string/philly</item]`). For user-facing strings, like those in the `cities` array, having them as string resources may make it easier for you to manage your translations.

The sample app then uses those arrays in a `ListPreference`:

```
<ListPreference
    android:dialogTitle="@string/listDialogTitle"
    android:entries="@array/cities"
    android:entryValues="@array/airportCodes"
    android:key="list"
    android:summary="@string/listSummary"
    android:title="@string/listTitle"/>
```

(from [SimplePrefs/src/main/res/xml/preferences.xml](#))

USING PREFERENCES

The dialog will show the strings in the `android:entries` array. The value that matches, position-wise, from the `android:entryValues` array is what gets saved in the `SharedPreferences`. So, if the user chooses Pittsburgh as the city, PIT will be the value saved in the `SharedPreferences`. If you want the user-visible strings to be the same as what goes into the `SharedPreferences`, just use the same string-array resource for both `android:entries` and `android:entryValues`.

A `ListPreference` saves a string to `SharedPreferences`. A `MultiSelectListPreference` saves a Set of strings to the `SharedPreferences`.

There is also `DropDownPreference`, which works like a `ListPreference` but uses a drop-down list presentation, rather than a pop-up dialog.

SeekBarPreference

A `SeekBarPreference` shows a `SeekBar` widget, to allow the user to specify a value in a range. It saves its value as an `int` to the `SharedPreferences`.

Working with SharedPreferences

At some point, our app needs to use the values that the user provided via the preference UI. We might also want to save other data in `SharedPreferences` that is not part of the preference UI.

Reading Preferences

Call `getSharedPreferences()` on `android.preference.PreferenceManager` to get the `SharedPreferences` that is used by the preference UI for its values.

`SharedPreferences` offers a series of getters to access named preferences, returning a suitably-typed result (e.g., `getBoolean()` to return a boolean preference). The getters also take a default value, which is returned if there is no preference set under the specified key.

You can read values at any point. If you want to find out when the preference UI (or other code in your app) changes the preferences, call `registerOnSharedPreferenceChangeListener()` on the `SharedPreferences` to register an `OnSharedPreferenceChangeListener` to be notified of when values change.

USING PREFERENCES

The sample app has a ConstraintLayout that shows the values of our three preferences:

```
<?xml version="1.0" encoding="utf-8"?>
<layout 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">

    <data>

        <variable
            name="state"
            type="androidx.lifecycle.LiveData<com.commonsware.jetpack.simpleprefs.HomeViewState>" />
    </data>

    <androidx.constraintlayout.widget.ConstraintLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        tools:context=".HomeFragment">

        <TextView
            android:id="@+id/checkLabel"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_margin="8dp"
            android:text="@string/labelCheck"
            android:textAppearance="@style/TextAppearance.AppCompat.Large"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintTop_toTopOf="parent" />

        <TextView
            android:id="@+id/fieldLabel"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_margin="8dp"
            android:text="@string/fieldLabel"
            android:textAppearance="@style/TextAppearance.AppCompat.Large"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintTop_toBottomOf="@+id/checkLabel" />

        <TextView
            android:id="@+id/listLabel"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_margin="8dp"
            android:text="@string/listLabel"
            android:textAppearance="@style/TextAppearance.AppCompat.Large"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintTop_toBottomOf="@+id/fieldLabel" />

        <androidx.constraintlayout.widget.Barrier
            android:id="@+id/barrier"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            app:barrierDirection="end"
            app:constraint_referenced_ids="checkLabel,fieldLabel,listLabel" />

        <TextView

    
```

USING PREFERENCES

```
        android:id="@+id/checkValue"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_margin="8dp"
        android:text="@{state.isChecked ? @string/checked : @string/unchecked}"
        android:textAppearance="@style/TextAppearance.AppCompat.Large"
        app:layout_constraintBaseline_toBaselineOf="@+id/checkLabel"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toEndOf="@+id/barrier"
        tools:text="checked" />

    <TextView
        android:id="@+id/fieldValue"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_margin="8dp"
        android:text="@{state.fieldValue}"
        android:textAppearance="@style/TextAppearance.AppCompat.Large"
        app:layout_constraintBaseline_toBaselineOf="@+id/fieldLabel"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toEndOf="@+id/barrier"
        tools:text="Something" />

    <TextView
        android:id="@+id/listValue"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_margin="8dp"
        android:text="@{state.listValue}"
        android:textAppearance="@style/TextAppearance.AppCompat.Large"
        app:layout_constraintBaseline_toBaselineOf="@+id/listLabel"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toEndOf="@+id/barrier"
        tools:text="ABE" />

    <Button
        android:id="@+id/edit"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginEnd="8dp"
        android:layout_marginStart="8dp"
        android:layout_marginTop="8dp"
        android:text="@string/editPrefs"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/listValue" />
    </androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```

(from [SimplePrefs/src/main/res/layout/fragment_home.xml](#))

We are using data binding to populate three of the TextView widgets with certain values, pulled from a HomeViewState that we get via LiveData. That comes from HomeMotor implementation of AndroidViewModel:

```
package com.commonsware.jetpack.simpleprefs;
import android.app.Application;
```

USING PREFERENCES

```
import android.content.SharedPreferences;
import androidx.annotation.NonNull;
import androidx.lifecycle.AndroidViewModel;
import androidx.lifecycle.LiveData;
import androidx.lifecycle.MutableLiveData;
import androidx.preference.PreferenceManager;

public class HomeMotor extends AndroidViewModel {
    private final SharedPreferences prefs;
    private final MutableLiveData<HomeViewState> states = new MutableLiveData<>();

    public HomeMotor(@NonNull Application application) {
        super(application);

        prefs = PreferenceManager.getDefaultSharedPreferences(application);
        prefs.registerOnSharedPreferenceChangeListener(LISTENER);
        emitState();
    }

    @Override
    protected void onCleared() {
        prefs.unregisterOnSharedPreferenceChangeListener(LISTENER);
    }

    LiveData<HomeViewState> getStates() {
        return states;
    }

    private void emitState() {
        states.setValue(
            new HomeViewState(prefs.getBoolean("checkbox", false),
                prefs.getString("field", ""), prefs.getString("list", "")));
    }

    private SharedPreferences.OnSharedPreferenceChangeListener LISTENER =
        (prefs, key) -> emitState();
}
```

(from [SimplePrefs/src/main/java/com/commonware/jetpack/simpleprefs/HomeMotor.java](#))

```
package com.commonware.jetpack.simpleprefs

import android.app.Application
import android.content.SharedPreferences
import androidx.lifecycle.AndroidViewModel
import androidx.lifecycle.LiveData
import androidx.lifecycle.MutableLiveData
import androidx.preference.PreferenceManager

class HomeMotor(application: Application) : AndroidViewModel(application) {
    private val listener =
        SharedPreferences.OnSharedPreferenceChangeListener { _, _ -> emitState() }
    private val prefs = PreferenceManager.getDefaultSharedPreferences(application)
    private val _states = MutableLiveData<HomeViewState>()
    val states: LiveData<HomeViewState> = _states

    init {
        prefs.registerOnSharedPreferenceChangeListener(listener)
        emitState()
    }
}
```

USING PREFERENCES

```
override fun onCleared() {
    prefs.unregisterOnSharedPreferenceChangeListener(listener)
}

private fun emitState() {
    _states.value = HomeViewState(
        prefs.getBoolean("checkbox", false),
        prefs.getString("field", "") ?: "",
        prefs.getString("list", "") ?: ""
    )
}
```

(from [SimplePrefs/src/main/java/com/commonsware/jetpack/simpleprefs/HomeMotor.kt](#))

When the motor is created, we call `PreferenceManager.getDefaultSharedPreferences` to retrieve the `SharedPreferences` object. Then, we call `registerOnSharedPreferenceChangeListener()` to be notified of changes, then call `emitState()`. That reads the values of our three preferences (using `getBoolean()` and `getString()` methods) and puts them in the `HomeViewState` for the UI to use. Later on, when the `HomeMotor` is cleared, we call `unregisterOnSharedPreferenceChangeListener()`, so we no longer get updates (and so the motor can be garbage-collected).

USING PREFERENCES

The net result is that the HomeFragment will show the initial default values, then will reflect the results of any changes that you make:

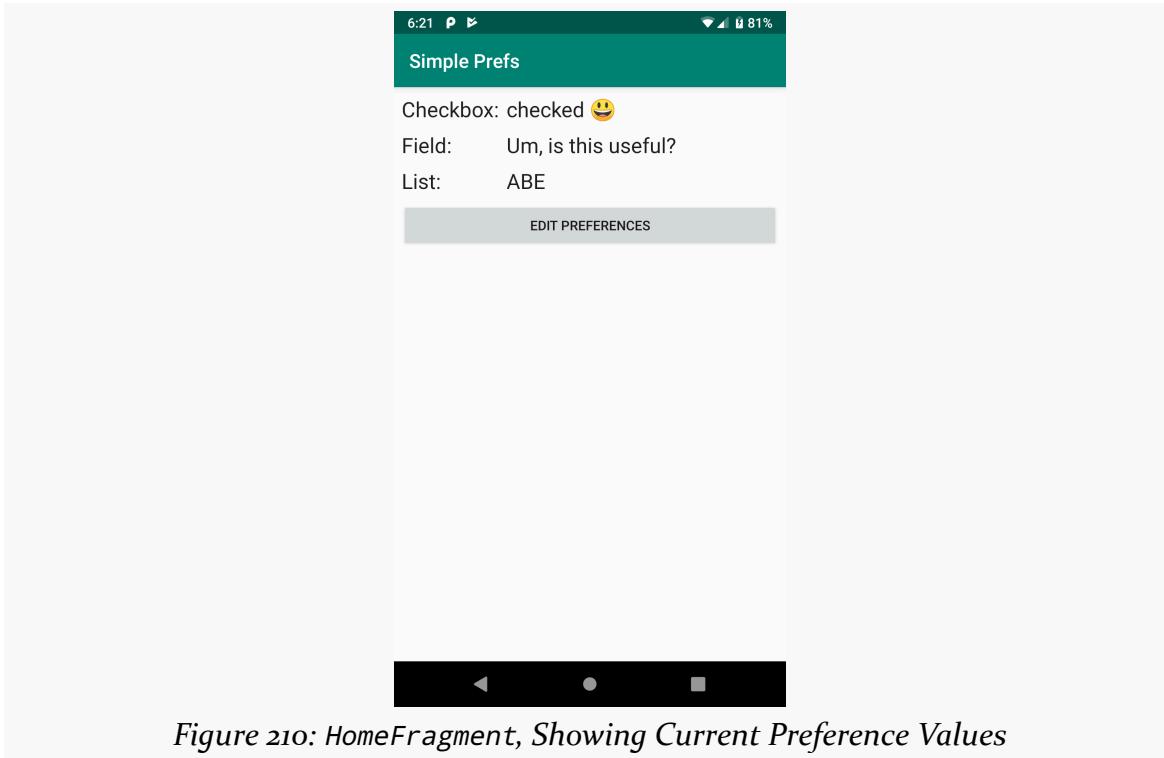


Figure 210: HomeFragment, Showing Current Preference Values

Modifying Preferences

Call `edit()` on the `SharedPreferences` object to get an “editor” for the preferences. This object has a set of setters that mirror the getters on the parent `SharedPreferences` object. It also has:

1. `remove()` to get rid of a single named preference
2. `clear()` to get rid of all preferences
3. `apply()` or `commit()` to persist your changes made via the editor

The last one is important — if you modify preferences via the editor and fail to save the changes, those changes will evaporate once the editor goes out of scope. `commit()` is a blocking call, while `apply()` works asynchronously. If you are on a background thread already, call `commit()`; otherwise, call `apply()`.

Requesting Permissions

Some things that your app might want to do — access the Internet, get a GPS fix, get personal information about the user’s contacts, etc. — will require permission from the user. Sometimes the user grants permission implicitly, just by installing your app. Sometimes, the user grants permission explicitly, for things that Google considers to be “dangerous”. And your app will need to deal with these permissions, including the possibility that the user decides not to grant you a permission.

In this chapter, we will explore the basics of Android’s permission system.

Frequently-Asked Questions About Permissions

Permissions are frequently a confusing topic in Android app development, particularly with respect to those “dangerous” permissions. So, here is a FAQ about Android’s permissions system.

What Is a Permission?

A permission is a way for Android (or, sometimes, a third-party app) to require an app developer to notify the user about something that the app will do that might raise concerns with the user. Only if an app holds a certain permission can the app do certain things that are defined by that permission.

Mechanically, permissions take the form of elements in the manifest, in the form of a `<uses-permission>` element.

When Will I Need a Permission?

Most permissions that you will deal with come from Android itself. Usually, the

REQUESTING PERMISSIONS

documentation will tell you when you need to request and hold one of these permissions.

However, occasionally the documentation has gaps.

If you are trying out some code and you crash with a `SecurityException` the description of the exception may tell you that you need to hold a certain permission — that means you need to add the corresponding `<uses-permission>` element to your manifest.

Third-party code, including Google's own Play Services SDK, may define their own custom permissions. Once again, ideally, you find out that you need to request a permission through documentation, and otherwise you find out through crashing during testing.

What Are Some Common Permissions, and What Do They Defend?

There are dozens upon dozens of permissions in Android. Here are some of the more common ones:

- `INTERNET`, if your application wishes to access the Internet through any means from your own process, using anything from raw Java sockets through the `WebView` widget
- `READ_EXTERNAL_STORAGE` and `WRITE_EXTERNAL_STORAGE`, for working with files directly on [external storage](#)
- `ACCESS_COARSE_LOCATION` and `ACCESS_FINE_LOCATION`, for determining where the device is, such as via GPS
- `READ_CONTACTS`, to get at personally-identifying information of arbitrary contacts that the user has in their Contacts app
- `BLUETOOTH` and `BLUETOOTH_ADMIN`, for communicating with other devices over Bluetooth
- `CAMERA`, for taking pictures directly using the camera APIs or wrapper libraries like the CameraX library
- `RECEIVE_BOOT_COMPLETED`, to get control when the device reboots

In this book and in casual conversation, we refer to the permissions using the unique portion of their name (e.g., `INTERNET`). Really, the full name of a framework permission will usually have `android.permission.` as a prefix (e.g., `android.permission.INTERNET`), for Android-defined permissions. Custom permissions from third-party apps should use a different prefix. You will need the

REQUESTING PERMISSIONS

full permission name, including the prefix, in your manifest entries.

What Are “Normal” and “Dangerous” Permissions?

In modern versions of Android, a normal permission is one that we have to request via the manifest, but the user does not need to approve explicitly. Just by installing the app, they implicitly grant us the permission. They can find out about these permissions (e.g., via the Play Store listing), but they cannot deny them, other than by uninstalling the app. Generally, normal permissions are ones that have limited user impact (e.g., RECEIVE_BOOT_COMPLETED, which only affects behavior on a reboot). The biggest exception to that rule is the INTERNET permission, which is a normal permission because a *lot* of apps want to be able to access the Internet.

By contrast, a dangerous permission is one that the user must approve explicitly. On Android 5.1 and older devices, this occurs at install time, either on the device or in the Play Store (for apps distributed through that channel). On Android 6.0+ devices, not only must our app have a `<uses-permission>` element in the manifest, but we also need to call methods to request that permission at runtime, at the point when we first need the permission. Mostly, dangerous permissions are ones that have privacy or security ramifications.

How Do I Request a Permission?

Put a `<uses-permission>` element in your manifest, as a direct child of the root `<manifest>` element (i.e., as a peer element of `<application>`), with an `android:name` attribute identifying the permission that you are interested in.

For example, here is a sample manifest, with a request to hold the `WRITE_EXTERNAL_STORAGE` permission:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest package="com.commonsware.jetpack.contenteditor"
  xmlns:android="http://schemas.android.com/apk/res/android">

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

  <application
    android:allowBackup="true"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:supportsRtl="true"
```

REQUESTING PERMISSIONS

```
    android:theme="@style/AppTheme">
    <activity android:name=".MainActivity">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />

            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
</application>

</manifest>
```

(from [ContentEditor/src/main/AndroidManifest.xml](#))

For normal permissions, this is all that you need. For dangerous permissions — such as `WRITE_EXTERNAL_STORAGE` — there is more work to be done, as we will explore [later in this chapter](#).

Note that you are welcome to have zero, one, or several such `<uses-permission>` elements. Also note that some libraries that you elect to use might add their own `<uses-permission>` elements to your manifest. If you look at the “Merged Manifest” sub-tab of the manifest editor, you will see all of the `<uses-permission>` elements that will be included in your app and where they come from.

Dangerous Permissions: Request at Runtime

In Android 6.0 and higher devices, permissions that are considered to be dangerous not only have to be requested via `<uses-permission>` elements, but you *also* have to ask the user to grant you those permissions at runtime. What you gain, though, is that users are not bothered with these permissions at install time, and you can elect to delay asking for certain permissions until such time as the user actually does something that needs them.

Let’s explore the runtime permissions system via a new series of questions.

Along the way, we will examine bits of the `ContentEditor` sample module in the [Sampler](#) and [SamplerJ](#) projects. This app implements a tiny text editor, where the user can edit text from various sources. We will explore the file I/O portions of this code in [an upcoming chapter](#). Here, we will focus on the `WRITE_EXTERNAL_STORAGE` permission that this app needs, as that is a dangerous permission and one that we have to request at runtime.

REQUESTING PERMISSIONS

What Permissions Are Affected By This?

Inside Android, permissions are organized into permission groups. As of Android 9.0, there are nine permission groups that contain dangerous permissions:

Permission Group	Permission
CALENDAR	READ_CALENDAR, WRITE_CALENDAR
CALL_LOG	PROCESS_OUTGOING_CALLS, READ_CALL_LOG, WRITE_CALL_LOG
CAMERA	CAMERA
CONTACTS	GET_ACCOUNTS, READ_CONTACTS, WRITE_CONTACTS
LOCATION	ACCESS_COARSE_LOCATION, ACCESS_FINE_LOCATION
MICROPHONE	RECORD_AUDIO
PHONE	ADD_VOICEMAIL, ANSWER_PHONE_CALLS, CALL_PHONE, READ_PHONE_NUMBERS, READ_PHONE_STATE, USE_SIP
SENSORS	BODY_SENSORS
SMS	READ_SMS, RECEIVE_SMS, RECEIVE_MMS, RECEIVE_WAP_PUSH, SEND_SMS
STORAGE	READ_EXTERNAL_STORAGE, WRITE_EXTERNAL_STORAGE

This roster changes over time. For example, in Android 6.0, the call log permissions were in the PHONE category.

REQUESTING PERMISSIONS

Users will be able to revoke permissions by group, through the Settings app. They can go into the page for your app, click on Permissions, and see a list of the permission groups for which you are requesting permissions:

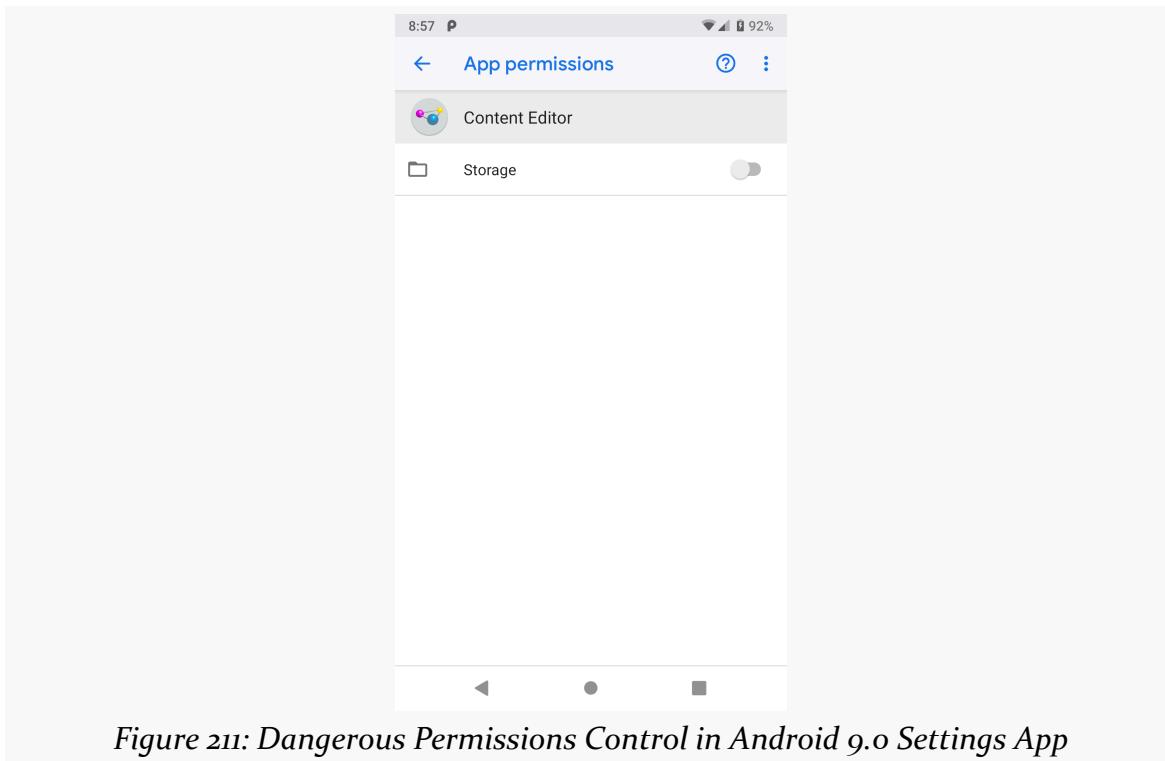


Figure 211: Dangerous Permissions Control in Android 9.0 Settings App

What Goes in the Manifest?

The same <uses-permission> elements as before. These declare the superset of all possible permissions that you can have. If you do not have a <uses-permission> element for a particular permission, you cannot ask for it at runtime, and the user cannot grant it to you.

How Do I Know If I Have Permission?

Use `ContextCompat.checkSelfPermission()`. This takes a Context and the full name of a permission as parameters, and it returns either `PERMISSION_GRANTED` or `PERMISSION_DENIED` to indicate the status of the permission:

```
private void loadFromExternalRoot() {
    if (ContextCompat.checkSelfPermission(this,
        Manifest.permission.WRITE_EXTERNAL_STORAGE) ==
```

REQUESTING PERMISSIONS

```
PackageManager.PERMISSION_GRANTED) {  
    loadFromDir(Environment.getExternalStorageDirectory());  
}  
else {  
    String[] perms = {Manifest.permission.WRITE_EXTERNAL_STORAGE};  
  
    ActivityCompat.requestPermissions(this, perms, REQUEST_PERMS);  
}  
}
```

(from [ContentEditor/src/main/java/com/commonsware/jetpack/contenteditor/MainActivity.java](#))

```
private fun loadFromExternalRoot() {  
    if (ContextCompat.checkSelfPermission(  
        this,  
        Manifest.permission.WRITE_EXTERNAL_STORAGE  
    ) == PackageManager.PERMISSION_GRANTED  
    ) {  
        loadFromDir(Environment.getExternalStorageDirectory())  
    } else {  
        val perms = arrayOf(Manifest.permission.WRITE_EXTERNAL_STORAGE)  
  
        ActivityCompat.requestPermissions(this, perms, REQUEST_PERMS)  
    }  
}
```

(from [ContentEditor/src/main/java/com/commonsware/jetpack/contenteditor/MainActivity.kt](#))

Here, we check to see if we hold the `WRITE_EXTERNAL_STORAGE` permission. For Java and Kotlin code, the `android.Manifest.permission` class has constants for all Android SDK permissions (e.g., `Manifest.permission.WRITE_EXTERNAL_STORAGE`) — it is better to use those than hard-code strings, as the compiler will help prevent you from introducing typos or other errors.

How Do I Ask the User For Permission?

To ask the user for one of the runtime permissions, call `ActivityCompat.requestPermissions()`. This takes your `Activity`, a `String` array of the permissions that you are requesting ,and a locally-unique integer to identify this request from any other similar requests that you may be making. This `int` serves in much the same role as does the `int` passed into `startActivityForResult()`, though you should keep the value to 8 bits (0 to 255) for maximum compatibility.

In the preceding code snippets, if `ContextCompat.checkSelfPermission()` indicates that we do not hold `Manifest.permission.WRITE_EXTERNAL_STORAGE`, we request

REQUESTING PERMISSIONS

that permission via `ActivityCompat.requestPermissions()`.

When we call `ActivityCompat.requestPermissions()`, the user will be presented with a system dialog with one “page” for each permission group, based on the array of permissions that we request:

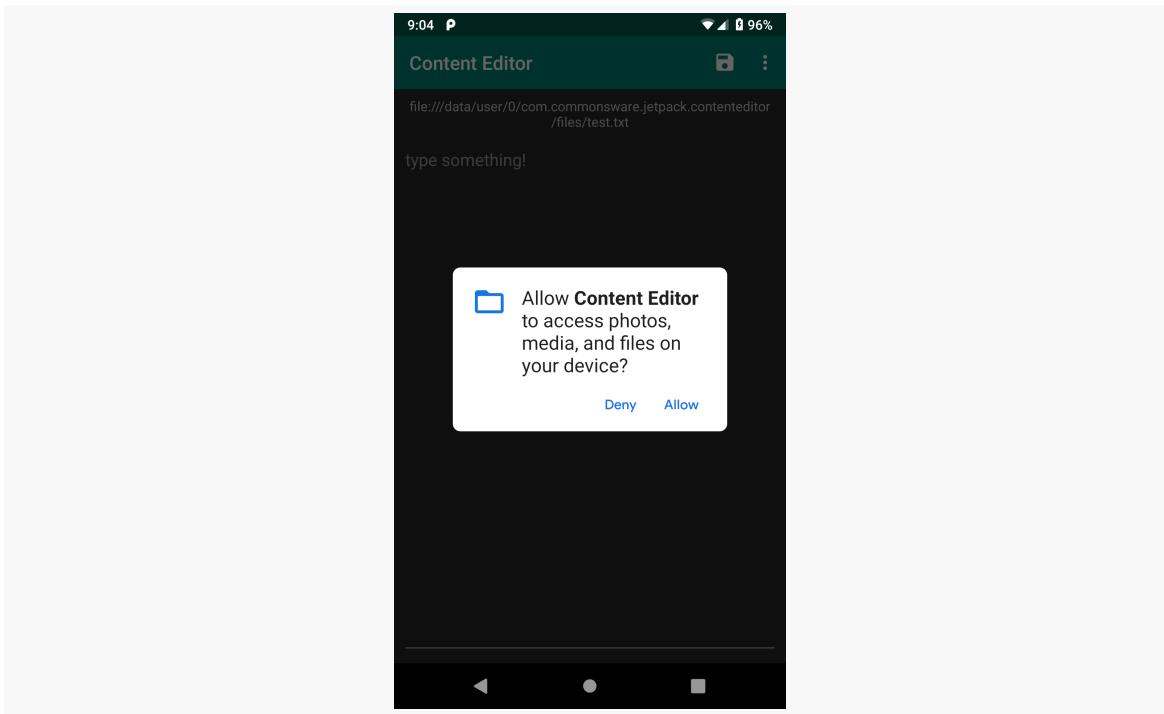


Figure 212: Runtime Permission Dialog, As Initially Displayed

The hope, of course, is that the user clicks “Allow” for each page, granting you all of the requested permissions.

When the user has proceeded through the dialog pages, your `Activity` will be called with `onRequestPermissionsResult()`. You are passed three parameters:

- the locally-unique integer from your `requestPermissions()` call, to identify which `requestPermissions()` call this is the result for
- a `String` array of the requested permissions
- an `int` array of the corresponding results (`PERMISSION_GRANTED` or `PERMISSION_DENIED`)

Whether you use those latter two parameters or simply call `ContextCompat.checkSelfPermission()` again is up to you. Regardless, at this point,

REQUESTING PERMISSIONS

you should determine what you got, so you know how to react, such as disabling things that the user cannot use given the lack of permission.

Frequently, you do the same thing in `onRequestPermissionsResult()` — if you were granted permission — that you do if your `ContextCompat.checkSelfPermission()` call had indicated that you had the permission already. In the case of the sample, we call a `loadFromDir()` function:

```
@Override
public void onRequestPermissionsResult(int requestCode,
                                       @NonNull String[] permissions,
                                       @NonNull int[] grantResults) {
    if (requestCode == REQUEST_PERMS) {
        if (grantResults.length == 1 && grantResults[0] == PackageManager.PERMISSION_GRANTED) {
            loadFromDir(Environment.getExternalStorageDirectory());
        } else {
            Toast.makeText(this, R.string.msg_sorry, Toast.LENGTH_LONG).show();
        }
    }
}
```

(from [ContentEditor/src/main/java/com/commonsware/jetpack/contenteditor/MainActivity.java](#))

```
override fun onRequestPermissionsResult(
    requestCode: Int,
    permissions: Array<String>,
    grantResults: IntArray
) {
    if (requestCode == REQUEST_PERMS) {
        if (grantResults.size == 1 && grantResults[0] == PackageManager.PERMISSION_GRANTED) {
            loadFromDir(Environment.getExternalStorageDirectory())
        } else {
            Toast.makeText(this, R.string.msg_sorry, Toast.LENGTH_LONG).show()
        }
    }
}
```

(from [ContentEditor/src/main/java/com/commonsware/jetpack/contenteditor/MainActivity.kt](#))

When Do I Ask the User For Permission?

That depends a bit on the nature of the permission.

In an ideal world, your app can function without any of the revocable permissions granted to you, albeit perhaps in a limited fashion. In that case, you might ask for permission only when the user tries to do something (e.g., taps on an action bar item) for which you definitely need the permission.

However, sometimes you will need permission to be at all useful to the user. In that case, you will need to ask for permission when the app opens.

REQUESTING PERMISSIONS

In either case, though, bear in mind that while the user will see the dialog asking for permission, the user may not understand *why* you are asking for this permission. You need to make sure that the user understands the cost/benefit trade-off in granting the permission — in other words, what does the user get out of the deal?

For permissions that you are requesting based on user input, you might pop your own dialog or other UI explaining what you want and why you want it, before calling `ActivityCompat.requestPermissions()`. For permissions that you would want to ask for when the app starts up, make sure that you clearly explain the need for the permissions and what the user gets in exchange as part of a one-time introductory tutorial, one that might also be accessed via an overflow item or nav drawer entry as part of your app's help facility.

When Do I Not Ask the User For Permission?

One limitation with the `ActivityCompat.requestPermissions()` implementation is that it is oblivious to configuration changes.

For example, suppose that in `onCreate()` of your activity, you check to see if you have been granted a runtime permission (via `ContextCompat.checkSelfPermission()`), and if you have not, you call `ActivityCompat.requestPermissions()` to request it from the user. This displays the dialog. Now the user rotates the screen. If the user denies the permission, by default, the user will immediately see the permission dialog again... because your activity will have been destroyed and recreated, and your `onCreate()` will see that you do not have the permission, and so you ask for it again.

In cases like this, you will need to track whether you are in the permission-request flow (e.g., via a boolean saved in the instance state Bundle) and skip requesting the permission if you have been recreated in the middle of that flow.

What Do I Do If the User Says “No”?

The user can click the “Deny” button for one or more pages in that system dialog and thereby reject granting you the requested permission(s).

If you were requesting permission as a direct response to some bit of user input (e.g., user tapped on an action bar item), and the user rejects the permission you need to do the work, obviously you cannot do the work. Depending on overall flow, showing a dialog or something to explain why you cannot do what the user asked for may be needed. In some cases, you may deem it to be obvious, by virtue of the fact that the

REQUESTING PERMISSIONS

user saw the permission-request dialog and said “deny”.

If you were requesting permission pre-emptively, such as when the activity starts, you will need to decide whether that decision needs to be reflected in the current UI (e.g., “no data available” messages, disabled action bar items).

One thing you can do to help here is to detect when this has occurred before you request permissions again. Before you call `ActivityCompat.requestPermissions()`, you can call `ActivityCompat.shouldShowRequestPermissionRationale()`, supplying the name of a permission. This will return true if the user had previously declined to grant you permission, in cases where Android thinks that the user might benefit from learning a bit more about *why* you need the permission. You can use this to determine whether you should show some explanatory UI of your own first, before continuing with the permission request, or if you should just go ahead and call `ActivityCompat.requestPermissions()`.

What Do I Do If the User Says “No, And Please Stop Asking”?

The second time you ask a user for a particular runtime permission, the user will have a “Don’t ask again” option:

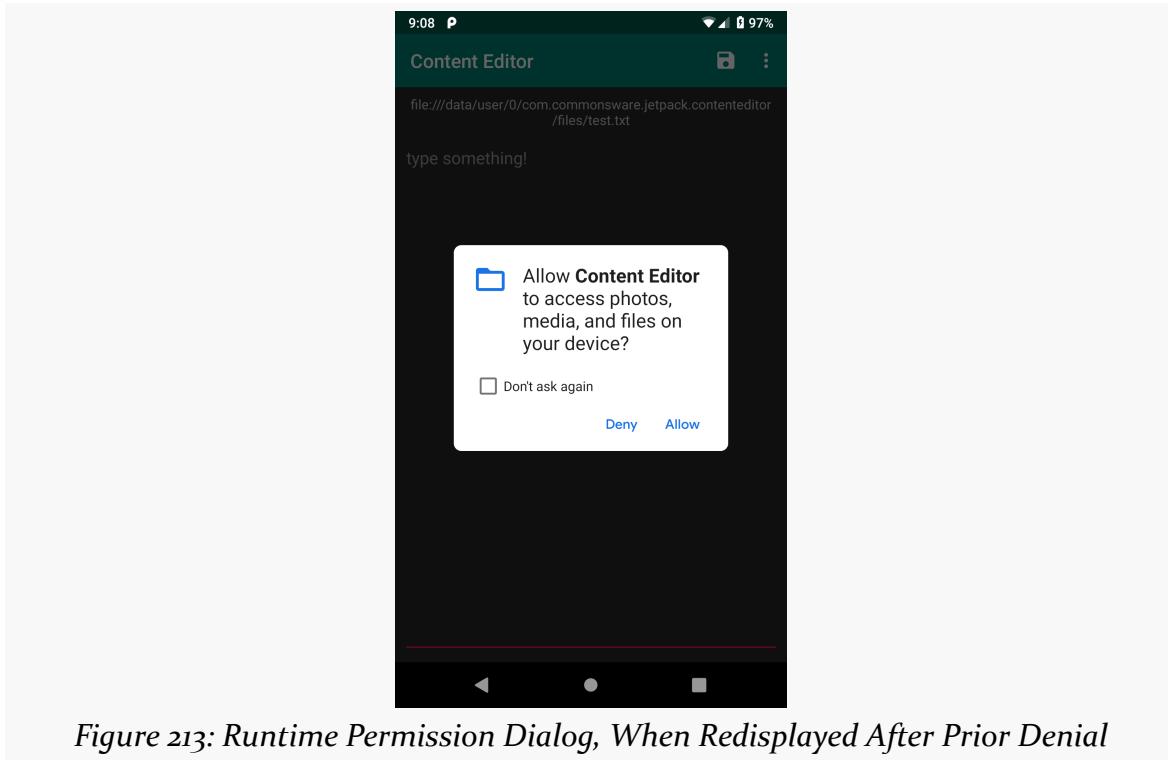


Figure 213: Runtime Permission Dialog, When Redisplayed After Prior Denial

REQUESTING PERMISSIONS

If the user chooses that, not only will you not get the runtime permission now, but all future requests will immediately call `onRequestPermissionsResult()` indicating that your request for permission was denied. The only way the user can now grant you this permission is via the Settings app.

You need to handle this situation with grace and aplomb.

Choices include:

- Disabling UI input (e.g., action bar items) that cannot be performed because you lack permission
- Display a dialog, explaining the situation, with a button that links the user over to your app's screen in Settings, so the user can grant you this permission
- Displaying inline messages about why you cannot show data (e.g., a list of contacts that you cannot show because the user did not grant you access), perhaps with a hyperlink that displays a screen with additional information about the situation

For permissions that, when denied, leave your app in a completely useless state, you may wind up just displaying a screen on app startup that says “sorry, but this app is useless to you”, with options for the user to uninstall the app or grant you the desired permissions.

Note that `ActivityCompat.shouldShowRequestPermissionRationale()` returns `false` if the user declined the permission and checked the checkbox to ask you to stop pestering the user.

How Do I Know If the User Takes Permissions Away From Me?

If the user grants you access to some permission group, the only way the user can revoke that is via the Settings app. If the user *does* revoke access to a permission group, your process is terminated.

Hence, while your code is running, you will have all permissions that you started with, plus any new ones that the user grants on the fly based upon your request. There should be no circumstance where your process is running yet you lose a permission.

That being said, your app is not specifically notified about losing the permission. You should be calling `ContextCompat.checkSelfPermission()` to determine what

REQUESTING PERMISSIONS

you can and cannot do, at least for every process invocation. And, since the call appears to be reasonably cheap, you should just call it whenever you need to know whether you can perform a particular operation.

What Happens When I Ship This to an Android 5.1 or Older Device?

Older devices behave as they always have. Since you still list the permissions in the manifest, those permissions will be granted to you if the user installs the app, and the user will be notified about those permissions as part of the installation process. If you are using ContextCompat and ActivityCompat as described above, your code should just work.

What Happens if the User Clears My App's Data?

If the user clears your app's data through the Settings app, the runtime permissions are cleared as well. Behavior at this point will be as if your app had been just installed — `ContextCompat.checkSelfPermission()` will return `PERMISSION_DENIED`, and you will need to request the permissions.

Licensed solely for use by Patrocinio Rodriguez

Handling Files

Most programmers, early on, learn how to read and write files. File I/O has been a staple of computer programming for decades. And, on Android, you can read and write files, much as you can with other operating systems.

The biggest difference in Android is *where* you can read and write files. That is quite a bit different than what you may be used to. And, increasingly, Google is pushing developers away from files entirely, steering us in the direction of the Storage Access Framework, as we saw in [the chapter on content](#).

In this chapter, we will explore where we can read and write files.

The Three Types of File Storage

File storage locations in Android break down into three main types: internal, external, and removable.

Internal Storage

A user will think that “internal storage” refers to what they get when they plug their phone into a desktop via a USB cable. From the standpoint of the Android SDK, though, that is really external storage.

Internal storage, from the Android SDK’s perspective, refers to portions of the on-board flash that are both:

- Private to your app, and
- Invisible to the user

HANDLING FILES

The only way that the user will interact with internal storage is through your app.

We have already worked a bit with internal storage: SharedPreferences are on internal storage. Later, when we [examine Room](#), its databases are stored on internal storage by default. In terms of standard Java/Kotlin file I/O code, you find your internal storage locations via methods on Context. The two most common of those methods are:

- `getFilesDir()`, which returns a place where you can read and write files, create subdirectories, etc.
- `getCacheDir()`, which returns a separate directory where you can also read and write files (and so on)... but where the system might delete your files to free up space for the user

External Storage

What the user thinks is “internal storage” is really “external storage” in the terms used by the Android SDK documentation. There are two main sets of locations on external storage that you can use: app-specific locations, and the overall shared portions of external storage.

App-Specific

There are methods on Context named `getExternalFilesDir()` and `getExternalCacheDir()`. These return locations on external storage that are unique for your app. And, as with their `getFilesDir()`/`getCacheDir()` counterparts, the “cache” ones are eligible to be cleared by the OS to free up disk space.

Unlike `getFilesDir()` and `getCacheDir()`, though, all locations on external storage can be accessed by the user and may be able to be accessed by other apps on the device. However, `getExternalFilesDir()` and `getExternalCacheDir()` are locations that are *unique* for your app — if you put things there, other apps should not accidentally overwrite or otherwise modify them.

Also, unlike `getFilesDir()` and `getCacheDir()`, `getExternalFilesDir()` and `getExternalCacheDir()` take a parameter. Typically, you pass in `null`.

Shared

Apps running on Android 9.0 and older can work with external storage overall via

HANDLING FILES

methods on the Environment class. In particular:

- `Environment.getExternalStorageDirectory()` returns the root of external storage
- `Environment.getExternalStoragePublicDirectory()` returns a specific common location on external storage, based on a supplied parameter (e.g., `Environment.DIRECTORY_DOWNLOADS` for a Downloads/ directory)

These locations are visible to the user and may be visible to other apps. And, since they are common and shared, other apps are more likely to manipulate files that you place here. At the same time, you may be able to manipulate the files of other apps, or files placed here by the user.

On newer versions of Android, these methods are deprecated, and you will not have access to them by default. Google is trying to steer you to the Storage Access Framework as an alternative.

Those location identifiers on Environment, such as `DIRECTORY_DOWNLOADS`, can also be passed to `getExternalFilesDir()` and `getExternalCacheDir()`, to get access to an app-specific directory on external storage for that type of material. Frequently, apps do not bother with this, just passing in `null` to get the main app-specific external storage directory.

Removable Storage

Removable storage refers to micro SD cards, USB thumb drives, or anything else that the device supports that can be physically removed by the user, without breaking their phone.

For several years, Android had no official support for removable storage. Nowadays, not only does the Storage Access Framework work with it, but there are methods on Context that let you get to app-specific directories on removable storage. Calling `getExternalFilesDirs()`, `getExternalCacheDirs()`, or `getExternalMediaDirs()` (note the plural method names) will return an array of `File` objects representing directories. The first element in that array will be a location on external storage – in the case of `getExternalFilesDirs()` and `getExternalCacheDirs()`, it should be the same location as you get from calling `getExternalFilesDir()` and `getExternalCacheDir()`. If the array has 2+ elements, all but the first will point to an app-specific directory on some removable medium. You can read and write files here, create subdirectories, etc.

What the User Sees

On Android devices:

- Users cannot see any of your files on internal storage, unless they root their device
- Users can see all of your files on external storage and removable storage

Anything a user can see, a user can read, write, or delete at will.

Note, though, that the user cannot see files on external or removable storage right away through some apps. Apps that are working with the filesystem or the Storage Access Framework will be able to see your new files immediately. Apps that rely on a central index of media called MediaStore will only find out about your new files when they get indexed. That will happen at some point automatically. You can use `MediaConnection.scanFile()` to force Android to index your new files more quickly — we will see this in action later in the chapter.

Storage, Permissions, and Access

Android 4.4 through 9.0 had a fairly stable set of rules for when you need permissions:

Storage Type	Storage Methods	Permissions Required
internal	methods on Context	none
external	methods on Context	none
external	methods on Environment	READ_EXTERNAL_STORAGE, WRITE_EXTERNAL_STORAGE
removable	methods on Context	none

So, only if you access shared locations on external storage do you need permission. As the names suggest, `READ_EXTERNAL_STORAGE` allows you to read those files, while `WRITE_EXTERNAL_STORAGE` allows you to write those files. These are dangerous permissions, so you need to request them both in the manifest and at runtime, as we

saw in [the chapter on permissions](#).

Android 10 has its own rule: you no longer need permissions to read and write using filesystem APIs. However, you can only read and write to your app-specific locations on internal, external, and removable storage. Shared locations are inaccessible, which is why methods like `Environment.getExternalStorageDirectory()` are deprecated.

Reading, Writing, and Debugging Storage

Once you have a `File` object that points to a location of interest to you, everything else works like standard Java/Kotlin disk I/O. You can use streams (e.g., `FileInputStream`), readers/writers (e.g., `OutputStreamWriter`), and so on. However, disk I/O may be slow, so you want to do that I/O on a background thread of some form.

For debuggable apps — the sort that you normally run from Android Studio — the IDE will give you somewhat greater ability to view files than ordinary users get. There is a “Device File Explorer” tool in Android Studio, by default docked on the right edge. If you have a device (or emulator) available, it will give you a file explorer for that device that will let you examine the major storage locations of interest to you.

Introducing the Sample App

In [the chapter on permissions](#), we saw a few code snippets from the `ContentEditor` sample module in the [Sampler](#) and [SamplerJ](#) projects. This app implements a tiny text editor, where the user can edit text from various sources.

HANDLING FILES

The app mostly is a large `EditText` widget for typing in some text. Above it is a `TextView` that we use to show a `Uri` representation of the content being shown in the `EditText`. By default, this will open up a file in the app's portion of internal storage:

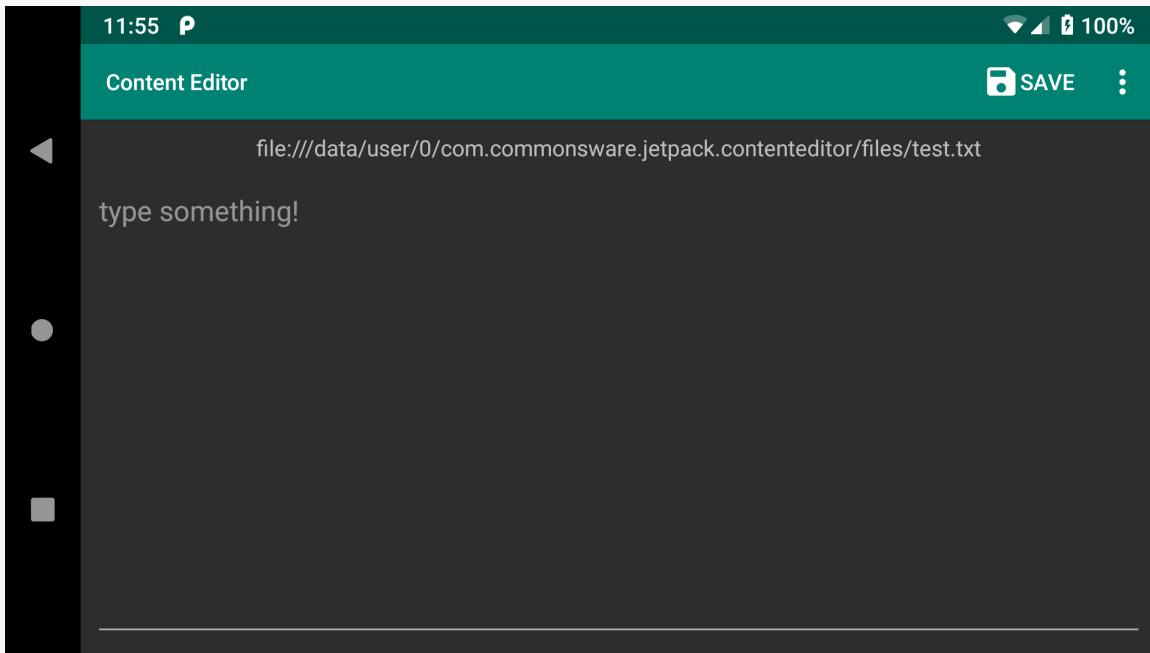


Figure 214: ContentEditor Sample, As Initially Opened

The Save button will let you save any changes that you made to the file. The overflow menu lets you switch the editor to different files or to content that you create using the Storage Access Framework.

Specifying the Location

Our `MainActivity` manages the options menu, including the “Save” item and the overflow. It, therefore, indicates where we should be loading our text from (or saving it to). That is triggered by `onOptionsItemSelected()`:

```
@Override  
public boolean onOptionsItemSelected(MenuItem item) {  
    switch (item.getItemId()) {  
        case R.id.loadInternal:  
            loadFromDir(getFilesDir());  
            return true;
```

HANDLING FILES

```
case R.id.loadExternal:
    loadFromDir(getExternalFilesDir(null));
    return true;

case R.id.loadExternalRoot:
    loadFromExternalRoot();
    return true;

case R.id.openDoc:
    startActivityForResult(
        new Intent(Intent.ACTION_OPEN_DOCUMENT)
            .setType("text/*")
            .addCategory(Intent.CATEGORY_OPENABLE), REQUEST_SAF);
    return true;

case R.id.newDoc:
    startActivityForResult(
        new Intent(Intent.ACTION_CREATE_DOCUMENT)
            .setType("text/plain")
            .addCategory(Intent.CATEGORY_OPENABLE), REQUEST_SAF);
    return true;

case R.id.save:
    motor.write(current, text.getText().toString());
    return true;
}
```

(from [ContentEditor/src/main/java/com/commonsware/jetpack/contenteditor/MainActivity.java](#))

```
override fun onOptionsItemSelected(item: MenuItem): Boolean {
    when (item.itemId) {
        R.id.loadInternal -> {
            loadFromDir(filesDir)
            return true
        }

        R.id.loadExternal -> {
            loadFromDir(getExternalFilesDir(null))
            return true
        }

        R.id.loadExternalRoot -> {
            loadFromExternalRoot()
            return true
        }

        R.id.openDoc -> {
            startActivityForResult(

```

HANDLING FILES

```
        Intent(Intent.ACTION_OPEN_DOCUMENT)
            .setType("text/*")
            .addCategory(Intent.CATEGORY_OPENABLE), REQUEST_SAF
    )
    return true
}

R.id.newDoc -> {
    startActivityForResult(
        Intent(Intent.ACTION_CREATE_DOCUMENT)
            .setType("text/plain")
            .addCategory(Intent.CATEGORY_OPENABLE), REQUEST_SAF
    )
    return true
}

R.id.save -> {
    current?.let { motor.write(it, text.text.toString()) }
    return true
}

return super.onOptionsItemSelected(item)
}
```

(from [ContentEditor/src/main/java/com/commonsware/jetpack/contenteditor/MainActivity.kt](#))

In the case of the loadInternal and loadExternal items, we call a loadFromDir() function providing getFilesDir() and getExternalFilesDir(null), respectively. loadFromDir() just clears out the EditText and passes the location to our MainMotor and its read() function:

```
private void loadFromDir(File dir) {
    text.setText("");
    motor.read(Uri.fromFile(new File(dir, FILENAME)));
}
```

(from [ContentEditor/src/main/java/com/commonsware/jetpack/contenteditor/MainActivity.java](#))

```
private fun loadFromDir(dir: File?) {
    text.setText("")
    motor.read(Uri.fromFile(File(dir, FILENAME)))
}
```

(from [ContentEditor/src/main/java/com/commonsware/jetpack/contenteditor/MainActivity.kt](#))

However, we pass the location to read() as a Uri, not a File. You can use

HANDLING FILES

`Uri.fromFile()` to create a `Uri` representation of a file. This is perfectly fine within an app. However, if you try passing such a `Uri` to another app, such as via an `ACTION_VIEW` Intent, you will get a `FileUriExposedException` on Android 7.0+. You can use `FileProvider` to get a safe `Uri` to pass to another app, as we will see [later in this chapter](#).

For the `loadExternalRoot` item, we will wind up calling `loadFromDir()` providing `Environment.getExternalStorageDirectory()` as the location. However, this requires permission from the user. So, in addition to having the `<uses-permission>` element in the manifest, we need to check for this permission at runtime, asking for it if we do not have it already, as we saw in [the chapter on permissions](#):

```
private void loadFromExternalRoot() {
    if (ContextCompat.checkSelfPermission(this,
        Manifest.permission.WRITE_EXTERNAL_STORAGE) ==
        PackageManager.PERMISSION_GRANTED) {
        loadFromDir(Environment.getExternalStorageDirectory());
    }
    else {
        String[] perms = {Manifest.permission.WRITE_EXTERNAL_STORAGE};

        ActivityCompat.requestPermissions(this, perms, REQUEST_PERMS);
    }
}
```

(from [ContentEditor/src/main/java/com/commonsware/jetpack/contenteditor/MainActivity.java](#))

```
private fun loadFromExternalRoot() {
    if (ContextCompat.checkSelfPermission(
        this,
        Manifest.permission.WRITE_EXTERNAL_STORAGE
    ) == PackageManager.PERMISSION_GRANTED
    ) {
        loadFromDir(Environment.getExternalStorageDirectory())
    } else {
        val perms = arrayOf(Manifest.permission.WRITE_EXTERNAL_STORAGE)

        ActivityCompat.requestPermissions(this, perms, REQUEST_PERMS)
    }
}
```

(from [ContentEditor/src/main/java/com/commonsware/jetpack/contenteditor/MainActivity.kt](#))

Then, in `onRequestPermissionsResult()`, if we were granted permission, we go ahead and call `loadFromDir()`:

HANDLING FILES

```
@Override
public void onRequestPermissionsResult(int requestCode,
                                       @NonNull String[] permissions,
                                       @NonNull int[] grantResults) {
    if (requestCode == REQUEST_PERMS) {
        if (grantResults.length == 1 && grantResults[0] == PackageManager.PERMISSION_GRANTED) {
            loadFromDir(Environment.getExternalStorageDirectory());
        } else {
            Toast.makeText(this, R.string.msg_sorry, Toast.LENGTH_LONG).show();
        }
    }
}
```

(from [ContentEditor/src/main/java/com/commonsware/jetpack/contenteditor/MainActivity.java](#))

```
override fun onRequestPermissionsResult(
    requestCode: Int,
    permissions: Array<String>,
    grantResults: IntArray
) {
    if (requestCode == REQUEST_PERMS) {
        if (grantResults.size == 1 && grantResults[0] == PackageManager.PERMISSION_GRANTED) {
            loadFromDir(Environment.getExternalStorageDirectory())
        } else {
            Toast.makeText(this, R.string.msg_sorry, Toast.LENGTH_LONG).show()
        }
    }
}
```

(from [ContentEditor/src/main/java/com/commonsware/jetpack/contenteditor/MainActivity.kt](#))

For the openDoc and newDoc items, we call `startActivityForResult()` with a Storage Access Framework action:

- ACTION_OPEN_DOCUMENT for openDoc
- ACTION_CREATE_DOCUMENT for newDoc

Both will trigger a call to `onActivityResult()`. And, in our case, we treat both the same: clear the `EditText` and pass the `Uri` to `read()`:

```
@Override
protected void onActivityResult(int requestCode, int resultCode,
                               @Nullable Intent data) {
    if (requestCode == REQUEST_SAF) {
        if (resultCode == RESULT_OK && data != null && data.getData() != null) {
            text.setText("");
            motor.read(data.getData());
        }
    }
    else {
        super.onActivityResult(requestCode, resultCode, data);
    }
}
```

HANDLING FILES

```
}
```

(from [ContentEditor/src/main/java/com/commonsware/jetpack/contenteditor/MainActivity.java](#))

```
override fun onActivityResult(
    requestCode: Int, resultCode: Int,
    data: Intent?
) {
    if (requestCode == REQUEST_SAF) {
        if (resultCode == Activity.RESULT_OK && data != null) {
            text.setText("")
            data.data?.let { motor.read(it) }
        }
    } else {
        super.onActivityResult(requestCode, resultCode, data)
    }
}
```

(from [ContentEditor/src/main/java/com/commonsware/jetpack/contenteditor/MainActivity.kt](#))

In these cases, we get a Uri from the Storage Access Framework, so we do not need to convert anything into a Uri.

In the end, no matter which of the options the user chooses, we pass a Uri to the MainMotor to read. Also, when the user clicks the save item, we pass that Uri and the current contents of the EditText to a write() function on the MainMotor.

Reading from the Location

Eventually, the read() call on MainMotor turns into a read() call on TextRepository.

Java

In Java, this creates and returns a LiveStreamReader implementation of LiveData that reads in the file and returns a StreamResult:

```
private static class LiveStreamReader extends LiveData<StreamResult> {
    private final Uri source;
    private final ContentResolver resolver;
    private final Executor executor;

    LiveStreamReader(Uri source, ContentResolver resolver, Executor executor) {
        this.source = source;
```

HANDLING FILES

```
this.resolver = resolver;
this.executor = executor;

postValue(new StreamResult(true, null, null, null));
}

@Override
protected void onActive() {
    super.onActive();

    executor.execute(() -> {
        try {
            postValue(new StreamResult(false, source,
                slurp(resolver.openInputStream(source)), null));
        }
        catch (FileNotFoundException e) {
            postValue(new StreamResult(false, source, "", null));
        }
        catch (Throwable t) {
            postValue(new StreamResult(false, null, null, t));
        }
    });
}

private String slurp(final InputStream is) throws IOException {
    final char[] buffer = new char[8192];
    final StringBuilder out = new StringBuilder();
    final Reader in = new InputStreamReader(is, StandardCharsets.UTF_8);
    int rsz = in.read(buffer, 0, buffer.length);

    while (rsz > 0) {
        out.append(buffer, 0, rsz);
        rsz = in.read(buffer, 0, buffer.length);
    }

    is.close();

    return out.toString();
}
}
```

(from [ContentEditor/src/main/java/com/commonsware/jetpack/contenteditor/TextRepository.java](#))

StreamResult is just an encapsulation of the state, including the text once we have it loaded:

```
package com.commonsware.jetpack.contenteditor;
```

HANDLING FILES

```
import android.net.Uri;

class StreamResult {
    final boolean isLoading;
    final Uri source;
    final String text;
    final Throwable error;

    StreamResult(boolean isLoading, Uri source, String text, Throwable error) {
        this.isLoading = isLoading;
        this.source = source;
        this.text = text;
        this.error = error;
    }
}
```

(from [ContentEditor/src/main/java/com/commonsware/jetpack/contenteditor/StreamResult.java](#))

Note that we use ContentResolver and openInputStream(). This not only works for a Storage Access Framework Uri but also one that we get from Uri.fromFile(). Most of TextRepository does not care what sort of Uri this is, so long as ContentResolver can open it.

Kotlin

The Kotlin version of TextRepository uses coroutines, returning a StreamResult directly from read():

```
try {
    resolver.openInputStream(source)?.use { stream ->
        StreamResult.Content(source, stream.readText())
    } ?: throw IllegalStateException("could not open $source")
} catch (e: FileNotFoundException) {
    StreamResult.Content(source, "")
} catch (t: Throwable) {
    StreamResult.Error(t)
}
```

(from [ContentEditor/src/main/java/com/commonsware/jetpack/contenteditor/TextRepository.kt](#))

Also, the Kotlin version of StreamResult is a sealed class representing the loading-content-error state:

```
package com.commonsware.jetpack.contenteditor
```

HANDLING FILES

```
import android.net.Uri

sealed class StreamResult {
    object Loading : StreamResult()
    data class Content(val source: Uri, val text: String) : StreamResult()
    data class Error(val throwable: Throwable) : StreamResult()
}
```

(from [ContentEditor/src/main/java/com/commonsware/jetpack/contenteditor/StreamResult.kt](#))

Writing to the Location

Similarly, when we call `write()` on the motor, it routes to `write()` on the `TextRepository`.

Java

The Java version of `TextRepository` creates and returns a `LiveStreamWriter` that wraps up the disk I/O in a `LiveData`:

```
private static class LiveStreamWriter extends LiveData<StreamResult> {
    private final Uri source;
    private final ContentResolver resolver;
    private final Executor executor;
    private final String text;
    private Context context;

    LiveStreamWriter(Uri source, ContentResolver resolver, Executor executor,
                    String text, Context context) {
        this.source = source;
        this.resolver = resolver;
        this.executor = executor;
        this.text = text;
        this.context = context;

        postValue(new StreamResult(true, null, null, null));
    }

    @Override
    protected void onActive() {
        super.onActive();

        executor.execute(() -> {
            try {
                OutputStream os = resolver.openOutputStream(source);
                PrintWriter out = new PrintWriter(new OutputStreamWriter(os));

```

HANDLING FILES

```
        out.print(text);
        out.flush();
        out.close();

    final String externalRoot =
        Environment.getExternalStorageDirectory().getAbsolutePath();

    if (source.getScheme().equals("file") &&
        source.getPath().startsWith(externalRoot)) {
        MediaScannerConnection
            .scanFile(context,
                new String[]{source.getPath()},
                new String[]{"text/plain"},
                null);
    }

   .postValue(new StreamResult(false, source, text, null));
}
catch (Throwable t) {
   .postValue(new StreamResult(false, null, null, t));
}
);
}
}
}
```

(from [ContentEditor/src/main/java/com/commonsware/jetpack/contenteditor/TextRepository.java](#))

We can just use `openOutputStream()` on `ContentResolver`, regardless of whether this is a Storage Access Framework `Uri` or one representing a file on the filesystem.

However, if the `Uri` does represent a file on the filesystem, and that file is on external storage, we want to tell the `MediaStore` about it. We detect this case by:

- Confirming that the scheme of the `Uri` is `file` — the scheme is the first portion of a URI, such as the `https` in `https://commonsware.com`
- Confirming that the path of the `Uri` starts with the root directory of external storage, to cover both our `external` and `externalRoot` scenarios

If both are true, we then call `scanFile()` on `MediaScannerConnection`, passing in:

- A Context
- An array of the paths to be indexed
- An array of the associated MIME types for each of those paths
- A callback object, or `null` if we do not need one (as is the case here)

HANDLING FILES

Our file may not be indexed immediately, but it should be indexed much more quickly than if we did not do this bit of work.

Kotlin

The Kotlin version of TextRepository does the same work, but once again it uses coroutines:

```
suspend fun write(context: Context, source: Uri, text: String) = withContext(Dispatchers.IO) {
    val resolver: ContentResolver = context.contentResolver

    try {
        resolver.openOutputStream(source)?.use { stream ->
            stream.writeText(text)

        val externalRoot =
            Environment.getExternalStorageDirectory().absolutePath

        if (source.scheme == "file" && source.path!!.startsWith(externalRoot)) {
            MediaScannerConnection
                .scanFile(
                    context,
                    arrayOf(source.path),
                    arrayOf("text/plain"),
                    null
                )
        }
    }

    StreamResult.Content(source, text)
} ?: throw IllegalStateException("could not open $source")
} catch (t: Throwable) {
    StreamResult.Error(t)
}
```

(from [ContentEditor/src/main/java/com/commonsware/jetpack/contenteditor/TextRepository.kt](#))

The Motor

Our motor's job is to work with TextRepository to load and save our text. As with most things, this is incrementally easier with Kotlin.

Java

Ideally, the activity would have a stable LiveData to observe for getting the content to display in the editor. However, our TextRepository returns a LiveData for each load request. Somehow, we need to “pour” all of those individual LiveData objects into a single LiveData that the activity observes.

HANDLING FILES

The solution for that is MediatorLiveData.

MediatorLiveData can observe one or several other LiveData objects. When values change in those LiveData objects, MediatorLiveData will invoke a lambda expression that you provide. There, you can convert the value to whatever you need and update the MediatorLiveData itself based on that change. You can add and remove LiveData objects from the MediatorLiveData whenever you need.

So, MainMotor uses a MediatorLiveData as the stable LiveData that the activity observes. As we change sources of text content, we swap in a new LiveData source for the MediatorLiveData:

```
package com.commonsware.jetpack.contenteditor;

import android.app.Application;
import android.net.Uri;
import androidx.annotation.NonNull;
import androidx.lifecycle.AndroidViewModel;
import androidx.lifecycle.LiveData;
import androidx.lifecycle.MediatorLiveData;

public class MainMotor extends AndroidViewModel {
    private final TextRepository repo;
    private final MediatorLiveData<StreamResult> results = new MediatorLiveData<>();
    private LiveData<StreamResult> lastResult;

    public MainMotor(@NonNull Application application) {
        super(application);

        repo = TextRepository.get(application);
    }

    MediatorLiveData<StreamResult> getResults() {
        return results;
    }

    void read(Uri source) {
        if (lastResult != null) {
            results.removeSource(lastResult);
        }

        lastResult = repo.read(source);
        results.addSource(lastResult, results::postValue);
    }

    void write(Uri source, String text) {
        if (lastResult != null) {
            results.removeSource(lastResult);
        }

        lastResult = repo.write(source, text);
        results.addSource(lastResult, results::postValue);
    }
}
```

HANDLING FILES

(from [ContentEditor/src/main/java/com/commonsware/jetpack/contenteditor/MainMotor.java](#))

Note that we do not have a separate view-state class — instead, we just pass StreamResult along to the activity. That works in this case because we had no conversions that we needed to perform on the data from the repository.

Kotlin

The Kotlin version is simpler, using MutableLiveData and viewModelScope for getting the coroutine results over to LiveData and from there to MainActivity:

```
package com.commonsware.jetpack.contenteditor

import android.app.Application
import android.net.Uri
import androidx.lifecycle.AndroidViewModel
import androidx.lifecycle.LiveData
import androidx.lifecycle.MutableLiveData
import androidx.lifecycle.viewModelScope
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.launch

class MainMotor(application: Application) : AndroidViewModel(application) {
    private val _results = MutableLiveData<StreamResult>()
    val results: LiveData<StreamResult> = _results

    fun read(source: Uri) {
        _results.value = StreamResult.Loading

        viewModelScope.launch(Dispatchers.Main) {
            _results.value = TextRepository.read(getApplicationContext(), source)
        }
    }

    fun write(source: Uri, text: String) {
        _results.value = StreamResult.Loading

        viewModelScope.launch(Dispatchers.Main) {
            _results.value = TextRepository.write(getApplicationContext(), source, text)
        }
    }
}
```

(from [ContentEditor/src/main/java/com/commonsware/jetpack/contenteditor/MainMotor.kt](#))

HANDLING FILES

The Results

The exact locations of internal, external, and removable storage may vary by device.

On a typical device, each app's portion of internal storage can be found in `/data/data/...`, where the `...` is replaced by the application ID of the app.

Unfortunately, this results in a very long list of apps, because all of the pre-installed apps get included:

The screenshot shows the Device File Explorer interface for a Google Pixel 2 running Android 9, API 28. The main pane displays a table of files under the path `/data/data`. The columns are Name, Permissions, Date, and Size. The table lists numerous system and application-specific files, many of which are empty or have small sizes. The permissions column shows various combinations of d, r, and w, with x indicating executability. The date column shows creation times ranging from 1971-08-15 to 2018-08-18. The size column shows most files are 0 B or 4 KB. The interface includes tabs for Gradle and Device File Explorer, and a toolbar with icons for file operations.

Name	Permissions	Date	Size
acct	dr-xr-xr-x	1971-08-15 05:57	0 B
bin	lrw-r--r--	2009-01-01 03:00	11 B
config	drwxr-xr-x	1969-12-31 19:00	0 B
d	lnw-r--r--	2009-01-01 03:00	17 B
data	drwxrwx--x	2018-08-18 09:48	4 KB
app	drwxrwx--x	2018-08-18 09:48	4 KB
data	drwxrwx--x	2018-08-18 09:48	4 KB
android	drwxrwx--x	2018-08-18 09:48	4 KB
android.auto_generated_rro_	drwxrwx--x	2018-08-18 09:48	4 KB
android.autoinstalls.config.google.nexus	drwxrwx--x	2018-08-18 09:48	4 KB
com.adobe.reader	drwxrwx--x	2018-08-18 09:48	4 KB
com.android.backupconfirm	drwxrwx--x	2018-08-18 09:48	4 KB
com.android.bips	drwxrwx--x	2018-08-18 09:48	4 KB
com.android.bluetooth	drwxrwx--x	2018-08-18 09:48	4 KB
com.android.bluetoothmidiservice	drwxrwx--x	2018-08-18 09:48	4 KB
com.android.bookmarkprovider	drwxrwx--x	2018-08-18 09:48	4 KB
com.android.calllogbackup	drwxrwx--x	2018-08-18 09:48	4 KB
com.android.captiveportallogin	drwxrwx--x	2018-08-18 09:48	4 KB
com.android.carrierdefaultapp	drwxrwx--x	2018-08-18 09:48	4 KB
com.android.cellbroadcastreceiver	drwxrwx--x	2018-08-18 09:48	4 KB
com.android.certinstaller	drwxrwx--x	2018-08-18 09:48	4 KB
com.android.chrome	drwxrwx--x	2018-08-18 09:48	4 KB
com.android.companionondevicemanager	drwxrwx--x	2018-08-18 09:48	4 KB
com.android.connectivity.metrics	drwxrwx--x	2018-08-18 09:48	4 KB
com.android.cts.ctsshim	drwxrwx--x	2018-08-18 09:48	4 KB
com.android.cts.priv.ctsshim	drwxrwx--x	2018-08-18 09:48	4 KB
com.android.defcontainer	drwxrwx--x	2018-08-18 09:48	4 KB
com.android.documentsui	drwxrwx--x	2018-08-18 09:48	4 KB

Figure 215: Device File Explorer, Showing (Some) Internal Storage Locations

HANDLING FILES

If you type something into the app after you initially launch it, then click “Save”, then scroll to the `/data/data/com.commonsware.jetpack.contenteditor/` directory in the Device File Explorer, you should see the `files/` subdirectory that maps to `getFilesDir()`, and in there you should see a `test.txt` file that contains what you wrote:

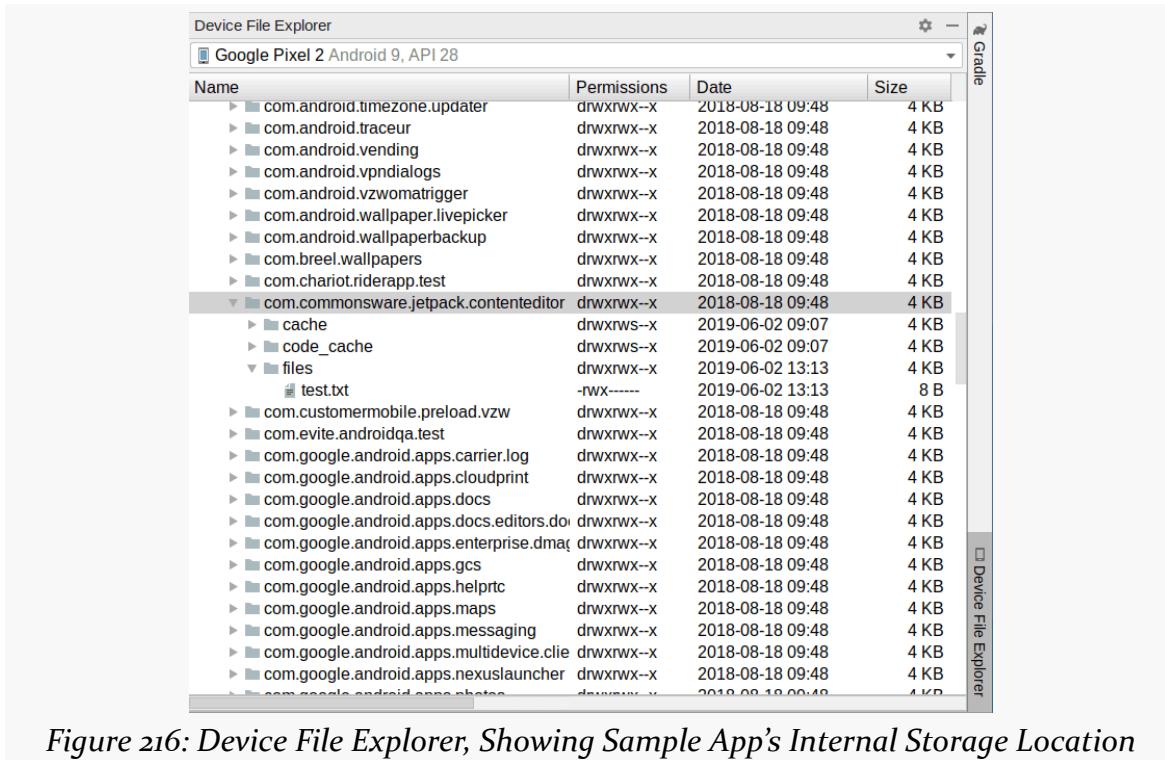


Figure 216: Device File Explorer, Showing Sample App’s Internal Storage Location

If you right-click over a file, a context menu gives you a few operations that you can perform on that file, such as “Save As” to download it to your development machine and “Delete” to remove it from the device. “Synchronize” updates the entire tree to reflect the current contents of the device.

HANDLING FILES

/sdcard in the Device File Explorer should give you a view of the root of external storage. This will include a test.txt file if you created it using the app by choosing “Load External Root” in the overflow menu, filling in some text, and clicking “Save”:

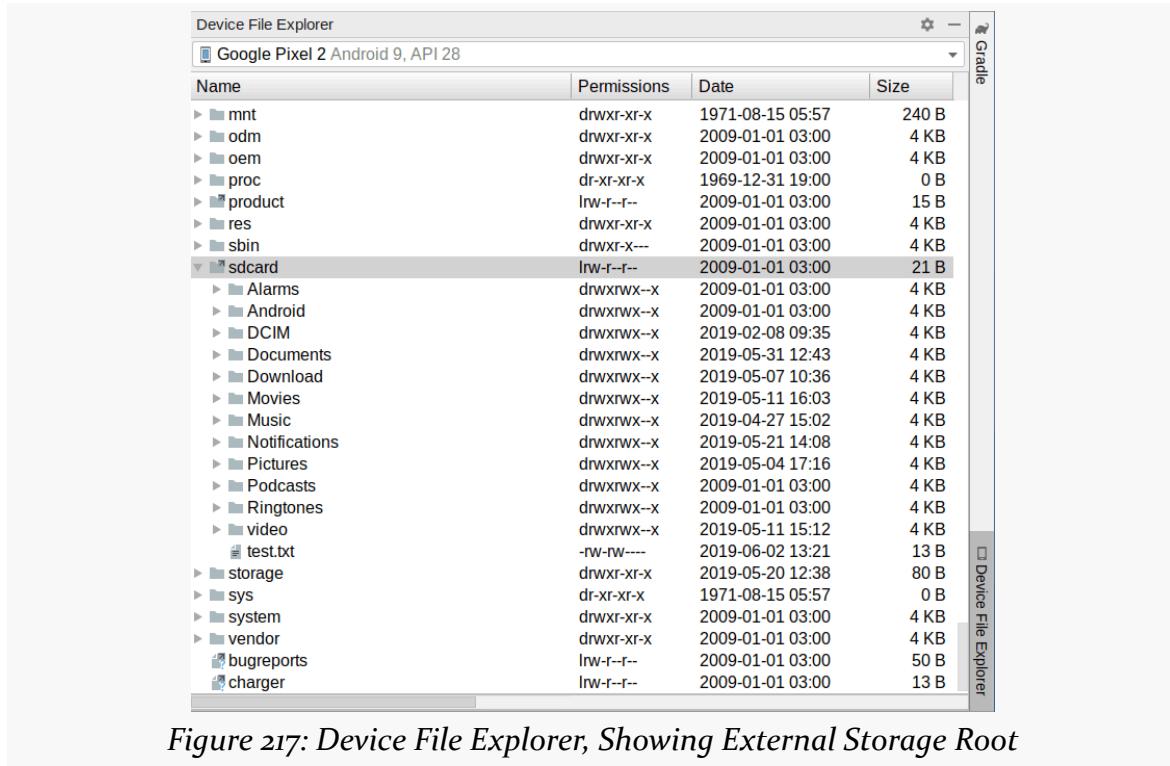


Figure 217: Device File Explorer, Showing External Storage Root

HANDLING FILES

There will be an `Android/` directory in the root of external storage, with a `data/` directory inside of it. That is akin to the `/data/data/` directory, except that it provides the list of app-specific external storage locations. And, fewer apps store data on external storage, so the list is more manageable, though it still can be rather long. If you use “Load External” in the app and save some text there, you will see a `test.txt` file in the app’s external storage location:

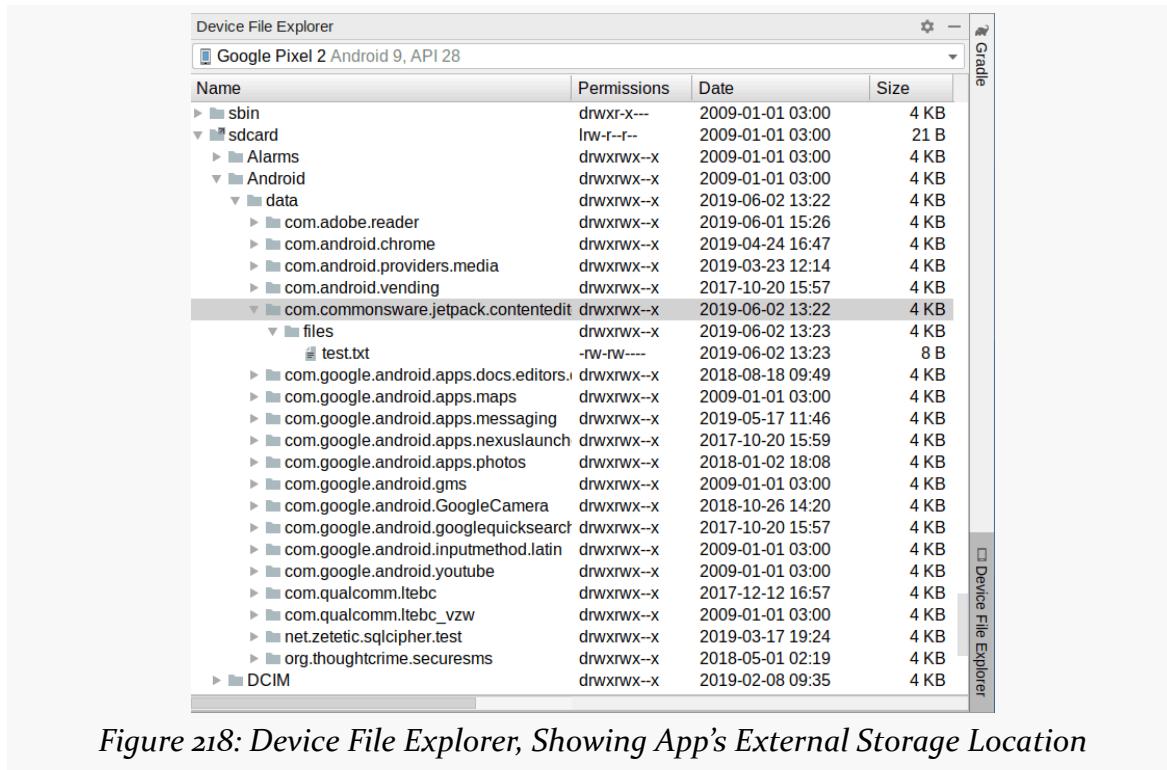


Figure 218: Device File Explorer, Showing App’s External Storage Location

Dealing with Android 10+

As noted earlier, methods like `Environment.getExternalStorageDirectory()` and their associated locations do not work on Android 10 and higher. As a result, we need to block access to the “Load External Root” item in our overflow menu on those devices.

To do that, we make use of version-specific resources.

In `res/values/`, each project has a `bools.xml` file. By convention, this contains `<bool>` resources that define a boolean value:

```
<?xml version="1.0" encoding="utf-8"?>
```

HANDLING FILES

```
<resources>
    <bool name="isPreQ">true</bool>
</resources>
```

(from [ContentEditor/src/main/res/values/bools.xml](#))

Here, we define `isPreQ` to be true.

Each project also has a `res/values-v29/` directory. This contains resources are only relevant on API Level 29+ devices, where API Level 29 corresponds to Android 10. In there, we have another `bools.xml` with its own definition of `isPreQ`, setting the value to false:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <bool name="isPreQ">false</bool>
</resources>
```

(from [ContentEditor/src/main/res/values-v29/bools.xml](#))

Then, in `res/menu/actions.xml`, for the “Load External Root” item, we use `android:enabled` and point it to `@bool/isPreQ`:

```
<item
    android:id="@+id/loadExternalRoot"
    android:title="@string/menu_external_root"
    android:enabled="@bool/isPreQ"
    app:showAsAction="never" />
```

(from [ContentEditor/src/main/res/menu/actions.xml](#))

As the name suggests, `android:enabled` controls whether the item is enabled or not. This, plus our dual definitions of `isPreQ`, means that:

- On Android 10+ devices, this item will be disabled, as Android will use the `res/values-v29/` edition of `isPreQ`
- On older devices, this item will be enabled, as Android will fall back to the `res/values/` edition of `isPreQ`

Our Java/Kotlin code can remain oblivious to this distinction, simply reacting to that item if it is chosen... even if on some devices, it cannot be chosen, because it is disabled.

Serving Files with FileProvider

We cannot assume that any other app has access to the files that we create, given Android's increasing restrictions on file access. Our app has access to the files, and those on external or removable storage are accessible by the user. Beyond that, though, nothing is guaranteed.

If we want to allow another app to have access to our files, we need to use something like `FileProvider`.

Scenarios for FileProvider

If we want to use `ACTION_VIEW` or various other Intent actions, we need to supply a `Uri` identifying what the action should be performed upon.

In early versions of Android, you might have written the file to external storage, then used `Uri.fromFile()` to get a `file:// Uri` that points to the file that you created. However, that has a few problems:

- It has been years since you could assume that every app requested `READ_EXTERNAL_STORAGE`
- You are cluttering up the user's external storage with files that the user may or may not want there
- On Android 7.0, `file:// Uri` values were mostly blocked by the OS, if you try to pass one in an Intent

If you used the Storage Access Framework and wrote your content to a location specified by one of its `Uri` values, you can use that `Uri` with `ACTION_VIEW`. However, this implies that the user has a use for this content independent of both apps (yours and whatever responds to the `ACTION_VIEW` Intent). That may not be the case. For example, your app might package a PDF file to serve as a user manual. Your app has the PDF, and a PDF viewer needs access to it, but the user does not necessarily need (or even want) that PDF to be stored somewhere public.

Those are the scenarios where `FileProvider` is useful: for content that your app has, that other apps need, but the user does not need independently of those apps.

Configuring FileProvider

The `PdfProvider` sample module in the [Sampler](#) and [SamplerJ](#) projects implements

HANDLING FILES

the scenario outlined above:

- We have a PDF packaged with the app as an asset
- We want to allow a PDF viewer to view that PDF, so the user can read it

One way to handle this is to copy the asset to a file, then use `FileProvider` to make it available to the PDF viewer.

Metadata XML Resource

We need to teach `FileProvider` what files we are willing to provide to other apps. To do that, we need to define an `xml` resource, in the same `res/xml/` directory where we put our preference screen configuration. `res/xml/` is a resource directory that can hold any XML that you want, including arbitrary XML that you create. Some parts of Android, like the preference system and `FileProvider`, will want a resource matching their desired XML structure.

In the case of `FileProvider`, that consists of a root `<paths>` element, followed by one or more child elements indicating where we want to serve from and what we want that location to be called:

```
<?xml version="1.0" encoding="utf-8"?>
<paths>
    <files-path name="stuff" path="/" />
</paths>
```

(from [PdfProvider/src/main/res/xml/provider_paths.xml](#))

`<files-path>` says “serve from `getFilesDir()` as a root directory”. That is one of a few possible element names, including `<cache-path>` (mapping to `getCacheDir()`) and `<external-files-path>` (mapping to `getExternalFilesDir(null)`).

The `path` attribute indicates where underneath the specified root location we want to serve. Here, by using `/`, we are saying that anything in `getFilesDir()` is fine. If there are files in `getFilesDir()` that you do not want to be available via `FileProvider`, set up a designated directory under `getFilesDir()` for the shareable files, then put that directory name in the `path` attribute.

The `name` attribute is a unique name for this location. No two locations in your resource can share the same name. This will form part of the `Uri` that `FileProvider` uses to map to your files.

Manifest Element

FileProvider is an implementation of ContentProvider. A ContentProvider, like an Activity, needs to be registered in the manifest. Instead of an <activity> element, it uses a <provider> element, but otherwise it fills the same basic role: tell Android that this class is an entry point for our app and how it can be used.

So, our manifest has a <provider> element, pointing to the AndroidX implementation of FileProvider:

```
<provider
    android:name="androidx.core.content.FileProvider"
    android:authorities="${applicationId}.provider"
    android:exported="false"
    android:grantUriPermissions="true">
    <meta-data
        android:name="android.support.FILE_PROVIDER_PATHS"
        android:resource="@xml/provider_paths" />
</provider>
```

(from [PdfProvider/src/main/AndroidManifest.xml](#))

As with <activity>, the android:name attribute on <provider> points to the class that is the ContentProvider implementation. Since this one is coming from a library, we need to provide the fully-qualified class name (androidx.core.content.FileProvider).

android:authorities is an identifier (or optionally several in a comma-delimited list, though you almost never see that done). This identifier has to be unique on the device; there cannot be two <provider> elements with the same authority installed at the same time. Fortunately, the Android build system lets us use the \${applicationId} macro, which expands into our application ID. This app's application ID is com.commonsware.jetpack.pdfprovider, so our authority turns into com.commonware.jetpack.pdfprovider.provider.

android:exported says whether or not third-party apps can initiate communications on their own with this ContentProvider. FileProvider requires this to be set to false. The only way another app will be able to work with our content is if we grant it temporary permission, on a case-by-case basis.

android:grantUriPermissions says whether or not we want to use that “case-by-case basis” approach or not. true indicates that we do.

HANDLING FILES

Finally, the nested <meta-data> element is a way that you can add configuration details to the manifest for arbitrary stuff that the build tools nor Android really know about. In this case, `FileProvider` expects a <meta-data> element for `android.support.FILE_PROVIDER_PATHS`, pointing to the `xml` resource that we saw in the preceding section.

With that resource and this manifest element, our `FileProvider` is ready for use.

Employing `FileProvider`

If you have a file in one of the locations listed in your <paths> resource, you can call `FileProvider.getUriForFile()` to retrieve the corresponding Uri:

```
Uri uri = FileProvider.getUriForFile(this, AUTHORITY, state.content);
Intent intent =
    new Intent(Intent.ACTION_VIEW)
    .setDataAndType(uri, "application/pdf")
    .addFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION);

startActivity(intent);
```

(from [PdfProvider/src/main/java/com/commonsware/jetpack/pdfprovider/MainActivity.java](#))

```
val uri = FileProvider.getUriForFile(this, AUTHORITY, state.pdf)
val intent = Intent(Intent.ACTION_VIEW)
    .setDataAndType(uri, "application/pdf")
    .addFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION)

startActivity(intent)
```

(from [PdfProvider/src/main/java/com/commonsware/jetpack/pdfprovider/MainActivity.kt](#))

Here, `state.pdf` is a `File` object, pointing to a PDF file that we have copied from assets to a file via some code in `MainMotor`. `this` is `MainActivity`. `AUTHORITY` is the authority that we used when declaring the `FileProvider` in the manifest:

```
private static final String AUTHORITY =
    BuildConfig.APPLICATION_ID + ".provider";
```

(from [PdfProvider/src/main/java/com/commonsware/jetpack/pdfprovider/MainActivity.java](#))

```
private const val AUTHORITY = "${BuildConfig.APPLICATION_ID}.provider"
```

(from [PdfProvider/src/main/java/com/commonsware/jetpack/pdfprovider/MainActivity.kt](#))

The equivalent of the `${applicationId}` macro in the manifest is to refer to

HANDLING FILES

BuildConfig.APPLICATION_ID, so we are assembling the authority string the same way.

Given the Uri from FileProvider, we can put it in an ACTION_VIEW Intent and pass that to startActivity(), to try to view the PDF file. However, by default, third-party apps have no rights to view the content identified by that Uri. By calling addFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION) on the Intent, we are telling Android to grant read access (but not write access) to that content.

Overall, the activity has a pair of buttons, one to copy the PDF from the asset to a file in getFilesDir():

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.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">

    <Button
        android:id="@+id/export"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginEnd="8dp"
        android:layout_marginStart="8dp"
        android:layout_marginTop="8dp"
        android:text="@string/btn_export"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <Button
        android:id="@+id/view"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginEnd="8dp"
        android:layout_marginStart="8dp"
        android:layout_marginTop="8dp"
        android:enabled="false"
        android:text="@string/btn_view"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/export" />
```

HANDLING FILES

```
<TextView
    android:id="@+id/error"
    android:layout_width="0dp"
    android:layout_height="0dp"
    android:layout_marginBottom="8dp"
    android:layout_marginEnd="8dp"
    android:layout_marginStart="8dp"
    android:layout_marginTop="8dp"
    android:typeface="monospace"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/view" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

(from [PdfProvider/src/main/res/layout/activity_main.xml](#))

The work to copy the asset to a file is handled by an `exportPdf()` function on `MainMotor`:

```
package com.commonsware.jetpack.pdfprovider;

import android.app.Application;
import android.content.res.AssetManager;
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.util.concurrent.Executor;
import java.util.concurrent.Executors;
import androidx.annotation.NonNull;
import androidx.lifecycle.AndroidViewModel;
import androidx.lifecycle.LiveData;
import androidx.lifecycle.MutableLiveData;

public class MainMotor extends AndroidViewModel {
    private static final String FILENAME = "test.pdf";
    private final MutableLiveData<MainViewState> states = new MutableLiveData<>();
    private final AssetManager assets;
    private final File dest;
    private final Executor executor = Executors.newSingleThreadExecutor();

    public MainMotor(@NonNull Application application) {
        super(application);

        assets = application.getAssets();
        dest = new File(application.getFilesDir(), FILENAME);

        if (dest.exists()) {
            states.setValue(new MainViewState(false, dest, null));
        }
    }

    LiveData<MainViewState> getStates() {
        return states;
```

HANDLING FILES

```
}

void exportPdf() {
    states.setValue(new MainViewState(true, null, null));

    executor.execute(() -> {
        try {
            copy(assets.open(FILENAME));
            states.postValue(new MainViewState(false, dest, null));
        }
        catch (IOException e) {
            states.postValue(new MainViewState(false, null, e));
        }
    });
}

private void copy(InputStream in) throws IOException {
    FileOutputStream out=new FileOutputStream(dest);
    byte[] buf=new byte[8192];
    int len;

    while ((len=in.read(buf)) > 0) {
        out.write(buf, 0, len);
    }

    in.close();
    out.getFD().sync();
    out.close();
}
}
```

(from [PdfProvider/src/main/java/com/commonsware/jetpack/pdfprovider/MainMotor.java](#))

```
package com.commonsware.jetpack.pdfprovider

import android.app.Application
import androidx.lifecycle.AndroidViewModel
import androidx.lifecycle.LiveData
import androidx.lifecycle.MutableLiveData
import androidx.lifecycle.viewModelScope
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.launch
import java.io.File
import java.io.FileOutputStream
import java.io.IOException

private const val FILENAME = "test.pdf"

class MainMotor(application: Application) : AndroidViewModel(application) {
    private val _states = MutableLiveData<MainViewState>()
    val states: LiveData<MainViewState> = _states
    private val assets = application.assets
    private val dest = File(application.filesDir, FILENAME)
```

HANDLING FILES

```
init {
    if (dest.exists()) {
        _states.value = MainViewState.Content(dest)
    }
}

fun exportPdf() {
    _states.value = MainViewState.Loading

    viewModelScope.launch(Dispatchers.IO) {
        try {
            assets.open(FILENAME).use { pdf ->
                FileOutputStream(dest).use { pdf.copyTo(it) }
            }
            _states.postValue(MainViewState.Content(dest))
        } catch (e: IOException) {
            _states.postValue(MainViewState.Error(e))
        }
    }
}
```

(from [PdfProvider/src/main/java/com/commonsware/jetpack/pdfprovider/MainMotor.kt](#))

MainMotor does that work on a background thread, since it might take a few moments. It emits MainViewState objects to report our loading/content/error status:

```
package com.commonsware.jetpack.pdfprovider;

import java.io.File;

class MainViewState {
    final boolean isLoading;
    final File content;
    final Throwable error;

    MainViewState(boolean isLoading, File content, Throwable error) {
        this.isLoading = isLoading;
        this.content = content;
        this.error = error;
    }
}
```

(from [PdfProvider/src/main/java/com/commonsware/jetpack/pdfprovider/MainViewState.java](#))

```
package com.commonsware.jetpack.pdfprovider
```

HANDLING FILES

```
import java.io.File

sealed class MainViewState {
    object Loading : MainViewState()
    class Content(val pdf: File) : MainViewState()
    class Error(val throwable: Throwable) : MainViewState()
}
```

(from [PdfProvider/src/main/java/com/commonsware/jetpack/pdfprovider/MainViewState.kt](#))

MainActivity observes the LiveData of MainViewState, and when the content is ready, enables the second button. When that button is clicked, we start the PDF viewer activity to view our content.

```
package com.commonsware.jetpack.pdfprovider;

import android.content.Intent;
import android.net.Uri;
import android.os.Bundle;
import android.view.View;
import android.widget.TextView;
import androidx.appcompat.app.AppCompatActivity;
import androidx.core.content.FileProvider;
import androidx.lifecycle.ViewModelProviders;

public class MainActivity extends AppCompatActivity {
    private static final String AUTHORITY =
        BuildConfig.APPLICATION_ID + ".provider";

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

        final MainMotor motor = ViewModelProviders.of(this).get(MainMotor.class);
        final View export = findViewById(R.id.export);
        final View view = findViewById(R.id.view);
        final TextView error = findViewById(R.id.error);

        motor.getStates().observe(this, state -> {
            export.setEnabled(!state.isLoading && state.content == null);
            view.setEnabled(!state.isLoading && state.content != null);

            if (view.isEnabled()) {
                view.setOnClickListener(v -> {
                    Uri uri = FileProvider.getUriForFile(this, AUTHORITY, state.content);
                    Intent intent =
```

HANDLING FILES

```
        new Intent(Intent.ACTION_VIEW)
            .setDataAndType(uri, "application/pdf")
            .addFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION);

        startActivity(intent);
    });
}

if (state.error != null) {
    error.setText(state.error.getLocalizedMessage());
}
});

export.setOnClickListener(v -> motor.exportPdf());

}
}
```

(from [PdfProvider/src/main/java/com/commonsware/jetpack/pdfprovider/MainActivity.java](#))

```
package com.commonsware.jetpack.pdfprovider

import android.content.Intent
import android.os.Bundle
import androidx.appcompat.app.AppCompatActivity
import androidx.core.content.FileProvider
import androidx.lifecycle.Observer
import androidx.lifecycle.ViewModelProviders
import kotlinx.android.synthetic.main.activity_main.*

private const val AUTHORITY = "${BuildConfig.APPLICATION_ID}.provider"

class MainActivity : AppCompatActivity {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        val motor = ViewModelProviders.of(this).get(MainMotor::class.java)

        motor.states.observe(this, Observer { state ->
            when (state) {
                MainViewState.Loading -> {
                    export.isEnabled = false
                    view.isEnabled = false
                }
                is MainViewState.Content -> {
                    export.isEnabled = false
                }
            }
        })
    }
}
```

HANDLING FILES

```
        view.isEnabled = true
        view.setOnClickListener {
            val uri = FileProvider.getUriForFile(this, AUTHORITY, state.pdf)
            val intent = Intent(Intent.ACTION_VIEW)
                .setDataAndType(uri, "application/pdf")
                .addFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION)

            startActivity(intent)
        }
    }
    is MainViewState.Error -> {
        export.isEnabled = false
        view.isEnabled = false
        error.text = state.throwable.localizedMessage
    }
}
}

export.setOnClickListener { motor.exportPdf() }
}
```

(from [PdfProvider/src/main/java/com/commonsware/jetpack/pdfprovider/MainActivity.kt](#))

What You Should Use

For data that your app needs, but that the user only needs through your app, use internal storage. This can include data that the user might consume with some other app, such as in the PDF FileProvider. With internal storage, your app needs to initiate giving the user the data, whether it is directly in your UI or by serving it up to other apps.

For data that the user will want independently of your app, use the Storage Access Framework, due to the long-term limitations of working with external and removable storage via the filesystem. External or removable storage should be limited to cases where you are using some third-party library that only accepts a `File` as input, or similar cases where you are unable to use a `Uri` from the Storage Access Framework.

Accessing the Internet

Nearly every Android device has Internet access. It is fairly likely that you will have interest in accessing the Internet from your Android app.

Android has a lot to offer here, both in terms of what is part of the OS and in terms of the wide range of add-on libraries to assist in Internet access.

This chapter covers some general Internet access options for Android apps, then walks through a sample using two popular libraries: Retrofit and Glide.

NOTE: This chapter will use “the Internet” to refer to any networking. The rules and options outlined here are not solely for accessing the public Internet but also for accessing private networks, such as an office network.

An API Roundup

There are lots and lots of ways that your app can access the Internet. Some APIs are part of the Android SDK, while some are third-party libraries layered atop the SDK. Here are some of the more popular and better-known options.

Socket

In the end, anything in Java/Kotlin code that is working with the Internet will wind up using a `Socket`. If you are trying to create a library that implements some obscure or new Internet protocol, you might find yourself using `Socket` to connect to some server, then implementing all of the communications yourself.

Most developers should not need to do this. Most major protocols have implementations available for Android, whether in the OS or in a library. All else

being equal, you are better served using an existing implementation rather than “rolling your own”.

HttpURLConnection

The original HTTP API in Java centers around `HttpURLConnection` and related classes. Android supports this, and it is the only supported direct HTTP API in the Android SDK at the present time. However, the `HttpURLConnection` API is *very old*, and there are better alternatives, such as OkHttp. In fact, Android’s `HttpURLConnection` is built on top of a forked copy of OkHttp.

Apache HttpClient

In early Android versions, the Android SDK also included a copy of Apache’s `HttpClient` library. However, that has been removed in more modern versions of Android. You can still use Apache `HttpClient`, but you would need to add [an independent copy of the library](#), rather than use one provided by the Android SDK.

Apache `HttpClient` has a very rich API, with hundreds of classes and thousands of methods. As such, it tends to be rather verbose.

WebView

If your objective is to display Web content to the user, most likely you will want to use `WebView`. This is a widget in the Android SDK that renders Web content, much like a browser. You can put a `WebView` in your activities or fragments and hand the `WebView` the content to display, in the form of HTML strings or URLs.

Conversely, if you are looking to talk to a Web service or otherwise engage in HTTPS communications without displaying HTML/CSS/JavaScript to the user, you will want to use something other than `WebView`.

DownloadManager

If your objective is to download a file from a publicly-accessible URL, you could use `DownloadManager`. This too is part of the Android SDK, and its job is to perform this sort of download. It handles some of the complexity for you, such as dealing with network disconnections and picking up the download later from where it left off.

However, `DownloadManager` has many limitations. For example, you have no means

ACCESSING THE INTERNET

of providing a session cookie or authentication header to the Web server — the URL must be one that can be used without additional HTTP stuff. You have limited places for storing the downloaded result. By default, the user will see a notification in their status bar, indicating the progress of the download. And so on.

For its narrow purpose, `DownloadManager` may be OK, but is not designed for working with arbitrary Web services and does not offer much flexibility.

Volley

Google offers its own add-on library for Internet access, called [Volley](#). Volley offers a cleaner API for accessing Web services and downloading images than you get with `HttpURLConnection`, even though “under the covers” Volley uses `HttpURLConnection` to do its work. Beyond that, Volley’s primary “claims to fame” are:

- It was created by Google
- It reportedly is used by the Play Store and some other Google apps

As such, some developers prefer to use this, over other options, as it is easier to get the library approved by decision-makers.

OkHttp

Perhaps the most popular option for generic HTTP requests is [OkHttp](#). Published by Square, OkHttp is frequently updated and has good developer and community support. OkHttp is a first-class HTTP client, working directly with `Socket` (as opposed to being layered atop `HttpURLConnection`). OkHttp has a clean yet rich API for making and monitoring requests and responses.



You can learn more about OkHttp in the “Contacting a Web Service” chapter of [Exploring Android!](#)

OkHttp also serves as the “plumbing” for various other libraries, including most of those listed below.

Retrofit

Square’s [Retrofit](#) is a library layered atop of OkHttp that aims to simplify making

ACCESSING THE INTERNET

REST-style Web service calls. You use annotations on a Java/Kotlin interface to describe the REST URLs that you wish to access, along with the HTTP operations to perform (e.g., GET, POST, PUT). If you combine Retrofit with a parsing library suitable for the Web service — such as a JSON parser for Web services serving JSON responses — Retrofit allows you to craft an API that feels like it is just a local function call yet returns an object tree representing the response.

We will examine the use of Retrofit in [the sample app](#).

Apollo-Android

If your server is using GraphQL instead of REST, [Apollo-Android](#) is basically “Retrofit for GraphQL”. You provide your GraphQL in the form of documents in a `graphql/` project directory. Apollo-Android plugins will generate Java/Kotlin classes that implement that API and represent the responses. You can then use that API at runtime, with OkHttp handling the HTTPS communications.

Image Loaders

A common need in Android apps is to display images from URLs. There are *lots* of image-loading libraries that handle that work for you with a simple API. The two most popular are [Glide](#) and [Picasso](#). Both can integrate with OkHttp, which is particularly important if you are using OkHttp-based libraries elsewhere and wish to have a common configuration for things like logging, proxy servers, and so on.

We will examine the use of Glide in [the sample app](#).

Specialized APIs

If you are looking to communicate with an existing third-party Web service, there might be dedicated libraries for that purpose. That third party might offer their own SDK, and independent developers may have created other libraries.

In general, if a Web service has an official SDK, you are better off using that. If nothing else, when it comes time to get support, the Web service developers may be expecting you to use their SDK and may have difficulty helping you debug your own manual HTTPS code.

Android's Restrictions

To a large extent, accessing the Internet on Android is not significantly different than is accessing the Internet on other platforms. For example, Apache HttpClient, OkHttp, and Retrofit work with ordinary Java/Kotlin for use on desktops and servers, in addition to working on Android. Similarly, WebView is backed by the same Web rendering engine that powers Chrome, not only for Android, but for all Chrome-supported platforms.

However, Android *does* have some specific restrictions that will impact how you work with the Internet. None should be big problems for you, but they will require some extra care to handle.

The INTERNET Permission

To do anything with the Internet (or a local network) from your app, you need to hold the INTERNET permission. This includes cases where you use things like WebView — if your process needs network access, you need the INTERNET permission.

Hence, the manifest for our sample project contains the requisite <uses-permission> declaration:

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

NetworkOnMainThreadException

If you attempt to perform network I/O on the main application thread, you will crash immediately with a NetworkOnMainThreadException. Network I/O can be slow — even if it is fast for you in the office, it may be slow for users due to differences in network connectivity. As such, doing network I/O on the main application thread is *very* likely to cause your UI to freeze for a bit while that I/O goes on.

Technically, there are ways to disable the code that enforces this exception (a class called StrictMode). In practice, the right answer is to do your network I/O on a background thread.

Many of the HTTP options presented above have built-in thread management options:

- WebView
- DownloadManager
- Volley
- OkHttp
- Retrofit
- Apollo-Android
- Glide
- Picasso

In some cases, everything is asynchronous (e.g., WebView, DownloadManager). In other cases, you have your choice between synchronous and asynchronous APIs (e.g., OkHttp, Retrofit) — you would use the synchronous APIs in cases where you are doing something else to put the work in the background, such as Kotlin coroutines.

Cleartext Restrictions

On Android 8.0+, if you attempt to use an http URL, you will crash, as “cleartext” traffic is not allowed by default.

The best answer is to use an https URL. In particular, if you are working with a server that you control, please secure it with an SSL/TLS certificate and use https.

If you are not in control over the schemes used in the URLs, though, you can update your [network security configuration](#) to support cleartext traffic from some or all domains. Consider warning your users, though, when you are about to use an http URL.

The Reality of Mobile Devices

Mobile devices have an interesting property: they are mobile. Users with mobile devices have this annoying habit of walking, jogging, running, driving, riding, dancing, kayaking, skydiving, and so on. When they do that, their network connection may change:

- They switch from their home or office WiFi to a mobile data connection
- They switch from one mobile tower to another tower while they are “on the road” (or in the sky, on the water, etc.)
- They return to their home or office and reconnect to the WiFi

In all these cases, when the network connection changes, any outstanding socket

connections get dropped. It does not matter whether you are working directly with Socket or are working with HTTP APIs. So, you might be in the middle of getting a Web service reply and, all of a sudden, your request gets abandoned, because you (briefly) lost communication with the server.

In casual apps, like this book's samples, you largely ignore this issue. In production-grade apps, typically you handle this by having suitable retry policies. So, if you get an IOException, you retry the request up to N times before giving up. That way, for a transient problem like a connectivity change, you get the work done, just a bit more slowly. Yet, at the same time, you do not retry indefinitely, as if there are lots of successive failures, it may require user intervention to fix (e.g., the user turned on airplane mode).

Forecasting the Weather

The Weather sample module in the [Sampler](#) and [SamplerJ](#) projects shows a weather forecast for New York City. It gets the forecast from the US National Weather Service, which offers a Web service for retrieving forecasts. The forecast elements that the app uses includes the time, projected temperature, and an icon representing the expected weather (e.g., sunny, partly cloudy, mostly cloudy, rain, snow, fog, alien invasion, kaiju attack, or zombie infestation).

(OK, perhaps not those last three)

The icon is supplied in the form of a URL from which we can download the icon.

To get the weather forecast itself, we will use Retrofit. To populate ImageView widgets in a RecyclerView with the weather icons, we will use Glide.

The Dependencies

Glide is simple. Most of the time, you can just use the com.github.bumptech.glide:glide library:

```
implementation 'com.github.bumptech.glide:glide:4.9.0'
```

(from [Weather/build.gradle](#))

Retrofit is a bit more complicated. You will usually need at least two libraries:

- com.squareup.retrofit2:retrofit, which is Retrofit itself

- A “converter” library that teaches Retrofit how to parse the sort of content that your Web service returns (JSON, XML, etc.)

The US National Weather Service Web service API supports returning JSON. There are a few JSON converter options for Retrofit, each of which use a different JSON parser. If you are using a JSON parser elsewhere in your app, you can try to use the corresponding Retrofit converter library — that way, you do not wind up bundling two JSON parsing libraries in your app. The three most popular JSON parsers for Android are Gson, Jackson, and Moshi, and there are Retrofit converters for each of those. [Moshi](#) happens to be written by Square, the same team that created Retrofit itself, so this sample app uses Moshi:

```
implementation "com.squareup.retrofit2:retrofit:2.6.1"
implementation "com.squareup.retrofit2:converter-moshi:2.6.1"
```

(from [Weather/build.gradle](#))

The Response Classes

The converters that Retrofit supports are designed to take a raw response — such as a JSON string — and convert them into instances of classes that are designed to reflect the response structure. The US National Weather Service has [some documentation](#) about their Web service API, and that information can be used to create classes that match the JSON that they will serve.

Usually, your classes can skip portions of the response that you do not care about. Moshi or other JSON parsers will skip anything that appears in the JSON but does not have a corresponding spot in your Java or Kotlin classes.

The JSON that we get back from the Web service looks like this (with the list of forecast periods truncated to save space in the book):

```
{
    "@context": [
        "https://raw.githubusercontent.com/geojson/geojson-ld/master-contexts/geojson-
base.jsonld",
        {
            "wx": "https://api.weather.gov/ontology#",
            "geo": "http://www.opengis.net/ont/geosparql#",
            "unit": "http://codes.wmo.int/common/unit/",
            "@vocab": "https://api.weather.gov/ontology#"
        }
    ],
}
```

ACCESSING THE INTERNET

```
"type": "Feature",
"geometry": {
    "type": "GeometryCollection",
    "geometries": [
        {
            "type": "Point",
            "coordinates": [
                -74.0130202,
                40.714515800000001
            ]
        },
        {
            "type": "Polygon",
            "coordinates": [
                [
                    [
                        [
                            [
                                [
                                    [
                                        [
                                            [
                                                [
                                                    [
                                                        [
                                                            [
                                                                [
                                                                    [
                                                                        [
                                                                            [
                                                                                [
                                                                                    [
                                                                                        [
                                                                                            [
                                                                                                [
                                                                                                 [
                                                                                                 [
                                                                                                 [
                                                                                                 [
                                                                                                 [
                                                                                                 [
                                                                                                 [
                                                                                                 [
                                                                                                 [
                                                                                                 [
                                                                                                 [
                                                                                                 [
................................................................
```

ACCESSING THE INTERNET

```
        "value": 2.133599999999999,
        "unitCode": "unit:m"
    },
    "periods": [
        {
            "number": 1,
            "name": "Tonight",
            "startTime": "2019-06-05T18:00:00-04:00",
            "endTime": "2019-06-06T06:00:00-04:00",
            "isDaytime": false,
            "temperature": 67,
            "temperatureUnit": "F",
            "temperatureTrend": null,
            "windSpeed": "10 to 14 mph",
            "windDirection": "SW",
            "icon": "https://api.weather.gov/icons/land/night/
tsra,80?size=medium",
            "shortForecast": "Showers And Thunderstorms",
            "detailedForecast": "Showers and thunderstorms. Cloudy, with a low
around 67. Southwest wind 10 to 14 mph, with gusts as high as 24 mph. Chance of
precipitation is 80%. New rainfall amounts between a quarter and half of an inch
possible."
        },
        {
            "number": 2,
            "name": "Thursday",
            "startTime": "2019-06-06T06:00:00-04:00",
            "endTime": "2019-06-06T18:00:00-04:00",
            "isDaytime": true,
            "temperature": 83,
            "temperatureUnit": "F",
            "temperatureTrend": null,
            "windSpeed": "10 mph",
            "windDirection": "W",
            "icon": "https://api.weather.gov/icons/land/day/rain_showers,50/
rain_showers,20?size=medium",
            "shortForecast": "Chance Rain Showers",
            "detailedForecast": "A chance of rain showers before 3pm. Partly
sunny, with a high near 83. West wind around 10 mph. Chance of precipitation is 50%.
New rainfall amounts between a tenth and quarter of an inch possible."
        },
        {
            "number": 3,
            "name": "Thursday Night",
            "startTime": "2019-06-06T18:00:00-04:00",
            "endTime": "2019-06-07T06:00:00-04:00",
            "isDaytime": false,
            "temperature": 66,
```

ACCESSING THE INTERNET

```
        "temperatureUnit": "F",
        "temperatureTrend": null,
        "windSpeed": "7 to 10 mph",
        "windDirection": "N",
        "icon": "https://api.weather.gov/icons/land/night/sct?size=medium",
        "shortForecast": "Partly Cloudy",
        "detailedForecast": "Partly cloudy, with a low around 66. North wind
7 to 10 mph."
    }
]
}
}
```

For the purposes of this app, we only need a small subset of this information. We need the periods list of objects from the properties property of the root JSON object. And, for each period, we need the name, temperature, temperatureUnit, and icon properties. A full-featured US weather app could use a lot more of the properties, but we will keep this simple.

So, we have these classes to model that response in Java:

```
package com.commonsware.jetpack.weather;

import java.util.List;

public class WeatherResponse {
    public final Properties properties=null;

    public static class Properties {
        public final List<Forecast> periods=null;
    }
}
```

(from [Weather/src/main/java/com/commonsware/jetpack/weather/WeatherResponse.java](#))

```
package com.commonsware.jetpack.weather;

public class Forecast {
    final String name;
    final int temperature;
    final String temperatureUnit;
    final String icon;

    public Forecast(String name, int temperature,
                   String temperatureUnit, String icon) {
        this.name = name;
        this.temperature = temperature;
```

ACCESSING THE INTERNET

```
this.temperatureUnit = temperatureUnit;
this.icon = icon;
}
```

(from [Weather/src/main/java/com/commonsware/jetpack/weather/Forecast.java](#))

...and Kotlin:

```
package com.commonsware.jetpack.weather

class WeatherResponse {
    val properties: Properties? = null

    class Properties {
        val periods: List<Forecast>? = null
    }
}
```

(from [Weather/src/main/java/com/commonsware/jetpack/weather/WeatherResponse.kt](#))

```
package com.commonsware.jetpack.weather

data class Forecast(
    val name: String,
    val temperature: Int,
    val temperatureUnit: String,
    val icon: String
)
```

(from [Weather/src/main/java/com/commonsware/jetpack/weather/Forecast.kt](#))

The overall Web service response is a `WeatherResponse`, which holds a `Properties` object, which in turn host a `List` of `Forecast` objects.

Moshi does not care about the package structure, so long as it can create instances of the class and fill in the fields or properties. Here, we have a mix of top-level classes (`WeatherResponse`, `Forecast`) and nested classes (`Properties`) — you can organize your classes as you see fit.

The Retrofit API Declaration

The next step for using Retrofit is to declare an interface that describes the API that we are invoking on the Web service and maps it to functions that we want to be able to call from our Java/Kotlin code.

ACCESSING THE INTERNET

In our case, we are only invoking a single Web service URL, where we tell it a location for a forecast, and we get back the corresponding forecast data. So, our interface has only a `getForecast()` function:

```
package com.commonsware.jetpack.weather;

import retrofit2.Call;
import retrofit2.http.GET;
import retrofit2.http.Headers;
import retrofit2.http.Path;

public interface NWSInterface {
    @Headers("Accept: application/geo+json")
    @GET("/gridpoints/{office}/{gridX},{gridY}/forecast")
    Call<WeatherResponse> getForecast(@Path("office") String office,
                                         @Path("gridX") int gridX,
                                         @Path("gridY") int gridY);
}
```

(from [Weather/src/main/java/com/commonsware/jetpack/weather/NWSInterface.java](#))

```
package com.commonsware.jetpack.weather

import retrofit2.http.GET
import retrofit2.http.Headers
import retrofit2.http.Path

interface NWSInterface {
    @Headers("Accept: application/geo+json")
    @GET("/gridpoints/{office}/{gridX},{gridY}/forecast")
    suspend fun getForecast(
        @Path("office") office: String,
        @Path("gridX") gridX: Int,
        @Path("gridY") gridY: Int
    ): WeatherResponse
}
```

(from [Weather/src/main/java/com/commonsware/jetpack/weather/NWSInterface.kt](#))

Every Retrofit interface function will have at least one annotation, one that indicates the HTTP operation to perform and a relative path on which to perform it. In our case, that is `@GET("/gridpoints/{office}/{gridX},{gridY}/forecast")`, saying that we want to perform an HTTP GET operation on... something.

The `@GET` annotation takes a relative path, where portions of that path can come from parameters to the annotated function. Our three parameters — `office`, `gridX`,

and `gridY` — are themselves annotated with `@Path`. `@Path` says “this function parameter can be used to help assemble the relative path”. The name given as a parameter to `@Path` can be used in the relative path, wrapped in braces. Retrofit will then replace that brace-wrapped name with the actual runtime value of the parameter when we call it. So,

`"/gridpoints/{office}/{gridX},{gridY}/forecast"` will turn into something like `"/gridpoints/OKX/32,32/forecast"`, if we call `getForecast("OKX", 32, 34)`.

The US National Weather Service Web service API divides its forecasts into gridded areas served by offices. We are supplying the ID of an office (`office`) plus the X/Y coordinates of a particular grid cell (`gridX` and `gridY`). We will receive a forecast for that particular cell of that specific office. There is a separate Web service URL that gives us the office and cell for a given latitude and longitude. An app that provided weather information for an arbitrary location — such as one that is retrieved from `LocationManager` and the GPS hardware on the device — would use that URL to get the office and grid, then use the URL associated with `getForecast()` to get the weather. Here, we will supply the office and cell from values hard-coded in our activity, to help simplify the example.

Retrofit has a variety of additional annotations that you can add as needed. In our case, we have a `@Headers` annotation to add an HTTP header to our Web service call, indicating that we want a [GeoJSON](#)-encoded response.

The Repository

We have a `WeatherRepository` that is responsible for working with Retrofit and obtaining our weather forecast. As with some of the previous samples, the Java and Kotlin implementations diverge a bit, as Kotlin uses coroutines, while Java returns a `LiveData` instead.

Java

Our Java implementation of `WeatherRepository` looks like this:

```
package com.commonsware.jetpack.weather;

import androidx.lifecycle.LiveData;
import androidx.lifecycle.MutableLiveData;
import retrofit2.Call;
import retrofit2.Callback;
import retrofit2.Response;
import retrofit2.Retrofit;
import retrofit2.converter.moshi.MoshiConverterFactory;
```

ACCESSING THE INTERNET

```
class WeatherRepository {
    private static volatile WeatherRepository INSTANCE;
    private final NWSInterface api;

    synchronized static WeatherRepository get() {
        if (INSTANCE == null) {
            INSTANCE = new WeatherRepository();
        }

        return INSTANCE;
    }

    private WeatherRepository() {
        Retrofit retrofit=
            new Retrofit.Builder()
                .baseUrl("https://api.weather.gov")
                .addConverterFactory(MoshiConverterFactory.create())
                .build();

        api = retrofit.create(NWSInterface.class);
    }

    LiveData<WeatherResult> load(String office, int gridX, int gridY) {
        final MutableLiveData<WeatherResult> result = new MutableLiveData<>();

        result.setValue(new WeatherResult(true, null, null));

        api.getForecast(office, gridX, gridY).enqueue(
            new Callback<WeatherResponse>() {
                @Override
                public void onResponse(Call<WeatherResponse> call,
                                      Response<WeatherResponse> response) {
                    result.postValue(new WeatherResult(false, response.body().properties.periods, null));
                }

                @Override
                public void onFailure(Call<WeatherResponse> call, Throwable t) {
                    result.postValue(new WeatherResult(false, null, t));
                }
            });
    }

    return result;
}
}
```

(from [Weather/src/main/java/com/commonsware/jetpack/weather/WeatherRepository.java](#))

In the constructor, we get an instance of our `NWSInterface` from Retrofit. Specifically, we:

- Create a `Retrofit.Builder` object
- Provide the base URL to it (`https://api.weather.gov`), which will combine with the paths on the individual interface functions to assemble the entire URL to use
- Teach the Builder that it can use Moshi for converting JSON into objects, via

ACCESSING THE INTERNET

- ```
addConverterFactory(MoshiConverterFactory.create())
• build() the resulting Retrofit object
• Call create() on the Retrofit object, to cause Retrofit to give us an instance
of some generated class that implements our NWSInterface
```

Retrofit can handle background threading for us. Our Java implementation of `NWSInterface` returns our `WeatherResponse` wrapped in a `Call` object. The two main methods on `Call` are `execute()` (to perform the Web service request synchronously) and `enqueue()` (to perform the Web service request asynchronously, on a Retrofit-supplied background thread). `load()` on `WeatherRepository` uses `enqueue()`, so Retrofit will handle our threading for us. We need to then supply a `Callback` to receive either our `WeatherResponse` or a `Throwable` if there is some problem (e.g., the Web service is down for maintenance). In our case, we wrap those results in a `WeatherResult` and update a `MutableLiveData` with that result. `WeatherResult` encapsulates an `isLoading` flag, along with our `WeatherResponse` and `Throwable` from the callback:

```
package com.commonsware.jetpack.weather;

import java.util.List;

public class WeatherResult {
 final boolean isLoading;
 final List<Forecast> forecasts;
 final Throwable error;

 WeatherResult(boolean isLoading, List<Forecast> forecasts, Throwable error) {
 this.isLoading = isLoading;
 this.forecasts = forecasts;
 this.error = error;
 }
}
```

(from [Weather/src/main/java/com/commonsware/jetpack/weather/WeatherResult.java](#))

Hence, if we call `load()` on our `WeatherRepository` singleton, we will get a `LiveData` that we can observe, where it will give us our `WeatherResult` as we progress from the loading state to either the success or failure states.

## Kotlin

The Kotlin code is far simpler, because Retrofit (as of 2.6.0) has built-in support for coroutines. So, our Kotlin `NWSInterface` returns a `WeatherResponse` directly

## ACCESSING THE INTERNET

---

(without the `Call` wrapper), and its `getForecast()` is marked with the `suspend` modifier. So, our `WeatherRepository` can just have its own `suspend` function that delegates to Retrofit and converts the `WeatherResponse` or caught exception into `WeatherResult` objects:

```
package com.commonsware.jetpack.weather

import retrofit2.Retrofit
import retrofit2.converter.moshi.MoshiConverterFactory

object WeatherRepository {
 private val api = Retrofit.Builder()
 .baseUrl("https://api.weather.gov")
 .addConverterFactory(MoshiConverterFactory.create())
 .build()
 .create(NWSInterface::class.java)

 suspend fun load(office: String, gridX: Int, gridY: Int) = try {
 val response = api.getForecast(office, gridX, gridY)

 WeatherResult.Content(response.properties?.periods ?: listOf())
 } catch (t: Throwable) {
 WeatherResult.Error(t)
 }
}
```

(from [Weather/src/main/java/com/commonsware/jetpack/weather/WeatherRepository.kt](#))

In our case, then, we only have `Content` and `Error` states — `load()` returns the end result of the API call, so it has no opportunity to return a `Loading` state:

```
package com.commonsware.jetpack.weather

sealed class WeatherResult {
 data class Content(val forecasts: List<Forecast>) : WeatherResult()
 data class Error(val throwable: Throwable) : WeatherResult()
}
```

(from [Weather/src/main/java/com/commonsware/jetpack/weather/WeatherResult.kt](#))

## The Motor and the View States

The `Forecast` objects in our `WeatherResult` have an integer `temperature` value plus a `temperatureUnit` string (e.g., `F` for Fahrenheit, `C` for Celsius). For display purposes, it would be nice to convert those into a single string, one that we can data bind into a `TextView`.

## ACCESSING THE INTERNET

---

So, our MainMotor implementations will have a LiveData of MainViewState objects. MainViewState, in turn, will have a List of RowState objects, where we have a single property for the visual representation of the temperature. MainMotor will get the WeatherResult from the Web service and convert it into a MainViewState object as they come in.

### Java

MainViewState and RowState look a lot like WeatherResult and Forecast, just with the single temperature field:

```
package com.commonsware.jetpack.weather;

import java.util.List;

class MainViewState {
 final boolean isLoading;
 final List<RowState> forecasts;
 final Throwable error;

 MainViewState(boolean isLoading, List<RowState> forecasts, Throwable error) {
 this.isLoading = isLoading;
 this.forecasts = forecasts;
 this.error = error;
 }
}
```

(from [Weather/src/main/java/com/commonsware/jetpack/weather/MainViewState.java](#))

```
package com.commonsware.jetpack.weather;

public class RowState {
 public final String name;
 public final String temp;
 public final String icon;

 RowState(String name, String temp, String icon) {
 this.name = name;
 this.temp = temp;
 this.icon = icon;
 }
}
```

(from [Weather/src/main/java/com/commonsware/jetpack/weather/RowState.java](#))

As with some of the previous examples, MainMotor uses MediatorLiveData, to fold

## ACCESSING THE INTERNET

---

one or more load() calls into a single results field that contains our LiveData of MainViewState objects:

```
package com.commonsware.jetpack.weather;

import android.app.Application;
import java.util.ArrayList;
import androidx.annotation.NonNull;
import androidx.lifecycle.AndroidViewModel;
import androidx.lifecycle.LiveData;
import androidx.lifecycle.MediatorLiveData;

public class MainMotor extends AndroidViewModel {
 final private WeatherRepository repo = WeatherRepository.get();
 final MediatorLiveData<MainViewState> results = new MediatorLiveData<>();
 private LiveData<WeatherResult> lastResult;

 public MainMotor(@NonNull Application application) {
 super(application);
 }

 void load(String office, int gridX, int gridY) {
 if (lastResult != null) {
 results.removeSource(lastResult);
 }

 lastResult = repo.load(office, gridX, gridY);
 results.addSource(lastResult, weather -> {
 ArrayList<RowState> rows = new ArrayList<>();

 if (weather.forecasts != null) {
 for (Forecast forecast : weather.forecasts) {
 String temp =
 getApplication().getString(R.string.temp, forecast.temperature,
 forecast.temperatureUnit);

 rows.add(new RowState(forecast.name, temp, forecast.icon));
 }
 }

 results.postValue(
 new MainViewState(weather.isLoading, rows, weather.error));
 });
 }
}
```

(from [Weather/src/main/java/com/commonsware/jetpack/weather/MainMotor.java](#))

## ACCESSING THE INTERNET

---

To create the temperature string, we use a string resource that contains placeholders for the number and unit:

```
<string name="temp">%d%s</string>
```

(from [Weather/src/main/res/values/strings.xml](#))

Then, we use `getString()` on a Context to retrieve that string resource and fill in those placeholders with our desired values.

## Kotlin

The Kotlin implementation is similar, just using the `MutableLiveData` and `viewModelScope` approach that we saw in previous examples:

```
package com.commonsware.jetpack.weather

sealed class MainViewState {
 object Loading : MainViewState()
 data class Content(val forecasts: List<RowState>) : MainViewState()
 data class Error(val throwable: Throwable) : MainViewState()
}
```

(from [Weather/src/main/java/com/commonsware/jetpack/weather/MainViewState.kt](#))

```
package com.commonsware.jetpack.weather

data class RowState(
 val name: String,
 val temp: String,
 val icon: String
)
```

(from [Weather/src/main/java/com/commonsware/jetpack/weather/RowState.kt](#))

```
package com.commonsware.jetpack.weather

import android.app.Application
import androidx.lifecycle.AndroidViewModel
import androidx.lifecycle.LiveData
import androidx.lifecycle.MutableLiveData
import androidx.lifecycle.viewModelScope
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.launch

class MainMotor(application: Application) : AndroidViewModel(application) {
```

## ACCESSING THE INTERNET

---

```
private val _results = MutableLiveData<MainViewState>()
val results: LiveData<MainViewState> = _results

fun load(office: String, gridX: Int, gridY: Int) {
 _results.value = MainViewState.Loading

 viewModelScope.launch(Dispatchers.Main) {
 val result = WeatherRepository.load(office, gridX, gridY)

 _results.value = when (result) {
 is WeatherResult.Content -> {
 val rows = result.forecasts.map { forecast ->
 val temp = getApplication<Application>()
 .getString(
 R.string.temp,
 forecast.temperature,
 forecast.temperatureUnit
)

 RowState(forecast.name, temp, forecast.icon)
 }
 MainViewState.Content(rows)
 }
 is WeatherResult.Error -> MainViewState.Error(result.throwable)
 }
 }
}
```

(from [Weather/src/main/java/com/commonsware/jetpack/weather/MainMotor.kt](#))

## The Image Loading

Eventually, our List of RowState objects makes it over to a WeatherAdapter. This is a ListAdapter that we use to fill in a RecyclerView that will show the list of forecasts.

We use data binding for the rows, where our row.xml layout has binding expressions to populate its widgets from a RowState:

```
<?xml version="1.0" encoding="utf-8"?>
<layout 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">

 <data>
```

## ACCESSING THE INTERNET

---

```
<variable
 name="state"
 type="com.commonsware.jetpack.weather.RowState" />
</data>

<androidx.constraintlayout.widget.ConstraintLayout
 android:layout_width="match_parent"
 android:layout_height="wrap_content"
 android:paddingBottom="8dp"
 android:paddingTop="8dp">

 <TextView
 android:id="@+id/name"
 android:layout_width="wrap_content"
 android:layout_height="wrap_content"
 android:text="@{state.name}"
 android:textAppearance="@style/TextAppearance.AppCompat.Large"
 app:layout_constraintBottom_toBottomOf="@+id/icon"
 app:layout_constraintEnd_toStartOf="@+id/temp"
 app:layout_constraintStart_toStartOf="parent"
 app:layout_constraintTop_toTopOf="@+id/icon"
 tools:text="Tonight" />

 <ImageView
 android:id="@+id/icon"
 android:layout_width="0dp"
 android:layout_height="64dp"
 android:contentDescription="@string/icon"
 app:imageUrl="@{state.icon}"
 app:layout_constraintDimensionRatio="1:1"
 app:layout_constraintEnd_toEndOf="parent"
 app:layout_constraintTop_toTopOf="parent" />

 <TextView
 android:id="@+id/temp"
 android:layout_width="wrap_content"
 android:layout_height="wrap_content"
 android:layout_marginEnd="16dp"
 android:layout_marginStart="16dp"
 android:text="@{state.temp}"
 android:textAppearance="@style/TextAppearance.AppCompat.Large"
 app:layout_constraintBottom_toBottomOf="@+id/icon"
 app:layout_constraintEnd_toStartOf="@+id/icon"
 app:layout_constraintTop_toTopOf="@+id/icon"
 tools:text="72F" />

</androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```

## ACCESSING THE INTERNET

---

(from [Weather/src/main/res/layout/row.xml](#))

In particular, our ImageView for the weather icon uses `app:imageUrl="@{state.icon}"` to pull the `icon` property out of the `RowState` and apply it to the `ImageView`. `ImageView` does not have an `app:imageUrl` attribute, though — we are using a `BindingAdapter` that in turn uses `Glide` to load the image:

```
package com.commonsware.jetpack.weather;

import android.widget.ImageView;
import com.bumptech.glide.Glide;
import androidx.databinding.BindingAdapter;

public class BindingAdapters {
 @BindingAdapter("imageUrl")
 public static void loadImage(ImageView view, String url) {
 if (url != null) {
 Glide.with(view.getContext())
 .load(url)
 .into(view);
 }
 }
}
```

(from [Weather/src/main/java/com/commonsware/jetpack/weather/BindingAdapters.java](#))

```
package com.commonsware.jetpack.weather

import android.widget.ImageView
import androidx.databinding.BindingAdapter
import com.bumptech.glide.Glide

@BindingAdapter("imageUrl")
fun ImageView.loadImage(url: String?) {
 url?.let {
 Glide.with(context)
 .load(it)
 .into(this)
 }
}
```

(from [Weather/src/main/java/com/commonsware/jetpack/weather/BindingAdapters.kt](#))

For simple cases like this one, `Glide` has a really simple API:

- Call `Glide.with()` to get a `RequestManager` object, passing in a Context
- Call `load()` on the `RequestManager` to ask it to load a URL — this returns a

### RequestBuilder

- Call `into()` on the `RequestBuilder` to tell it to show the resulting image in the supplied `ImageView`

And that's it. Glide will handle doing the network I/O and populating the `ImageView` with the resulting image, using a background thread for the I/O work. It also handles recycling — if we call `into()` with an `ImageView` that already has an outstanding request, Glide will cancel the old request for us automatically.

## The Results

Our `MainActivity` starts all of this off by calling `load()` on the `MainMotor`:

```
motor.load("OKX", 32, 34)
```

(from [Weather/src/main/java/com/commonsware/jetpack/weather/MainActivity.kt](#))

Here, `OKX`, `32`, and `34` were determined by manually invoking another Web service URL, providing the latitude and longitude for One World Trade Center in lower Manhattan.

`MainActivity` observes the `LiveData` from the motor, and for successful Web service calls forwards the `List` of `RowState` objects to the `WeatherAdapter` that it created and attached to a `RecyclerView`.

## ACCESSING THE INTERNET

---

If you run the app, you should see an upcoming forecast for New York City:

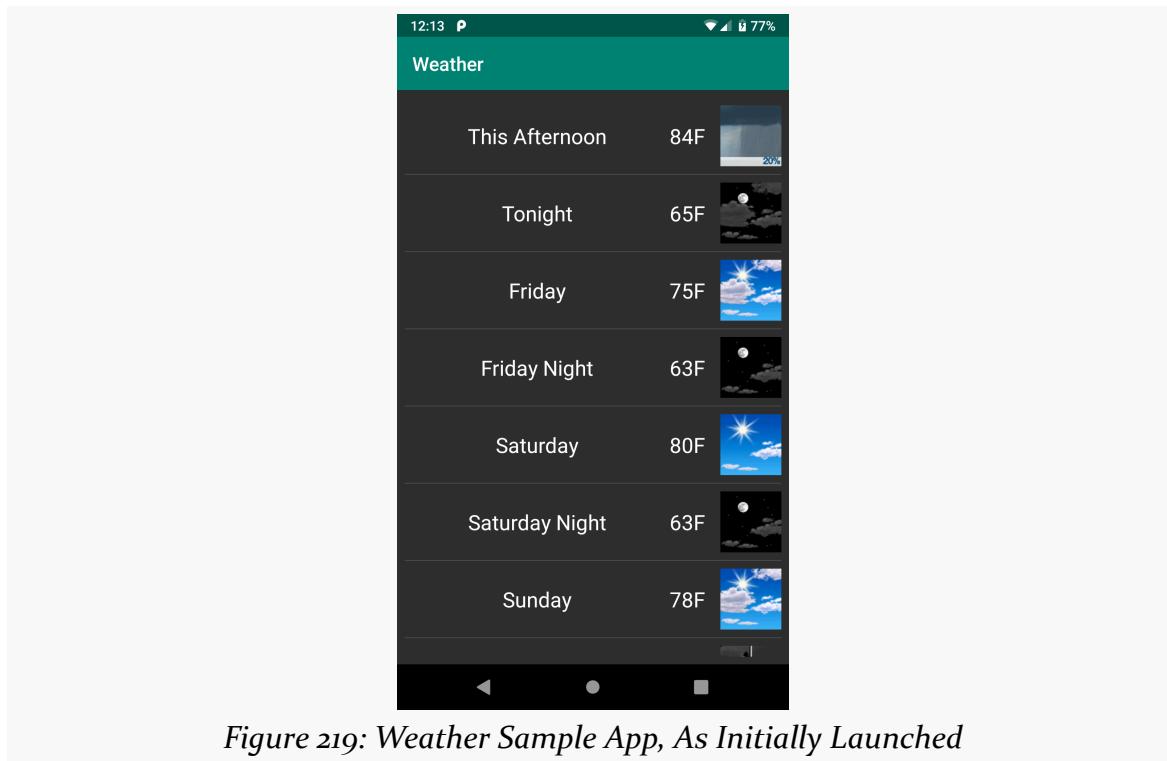


Figure 219: Weather Sample App, As Initially Launched

Here, we see that there will be a 20% chance of rain today, then clear skies for the next few days, with no signs of kaiju.

(then again, kaiju can be sneaky...)

Licensed solely for use by Patrocinio Rodriguez

# **Storing Data in a Room**

---

There are three main options for storing data locally in an Android device:

- `SharedPreferences`
- `SQLite databases`
- Arbitrary other files or content

In this chapter, we will look at the second of those: `SQLite databases`. `SQLite` is an embedded relational database, so you can use standard SQL, while the database is a simple set of files on the device, not some server.

The Jetpack solution for working with `SQLite databases` is called Room. Google describes Room as providing “an abstraction layer over `SQLite` to allow fluent database access while harnessing the full power of `SQLite`.”

In other words, Room aims to make your use of `SQLite` easier, through a lightweight annotation-based implementation of an [object-relational mapping \(ORM\) engine](#).

This chapter only scratches the surface of what Room has to offer, though it does include [a section on other Room features](#). Room is a fairly powerful library, and with that power comes a fair bit of complexity, once you get past the basics.

The Bookmarker sample module in the [Sampler](#) and [SamplerJ](#) projects is an app that tracks bookmarks. You will be able to share Web pages from your favorite Web browser (if it offers a “Share” option), and Bookmarker will save that URL in a Room database. It will also present the list of saved bookmarks to you, sorted by page title, and you can click on an entry in the list to view that Web page in your favorite browser.

## Room Requirements

To use Room, you need two dependencies in your module's build.gradle file:

1. The runtime library version, using the standard `implementation` directive
2. An annotation processor, using the `annotationProcessor` directive (in Java) or the `kapt` directive (in Kotlin)

The Java edition of Bookmarker pulls in both of those:

```
implementation "androidx.room:room-runtime:2.1.0"
annotationProcessor "androidx.room:room-compiler:2.1.0"
```

(from [Bookmarker/build.gradle](#))

The Kotlin edition pulls in both of those, along with an additional dependency:

```
implementation "androidx.room:room-runtime:2.1.0"
implementation "androidx.room:room-ktx:2.1.0"
kapt "androidx.room:room-compiler:2.1.0"
```

(from [Bookmarker/build.gradle](#))

`androidx.room:room-runtime` is the runtime library, while `androidx.room:room-compiler` is the annotation processor. The third dependency in the Kotlin edition is `androidx.room:room-ktx`, which gives Room the ability to work with Kotlin coroutines for threading purposes.

## Room Furnishings

Roughly speaking, your basic use of Room is divided into three sets of classes:

1. Entities, which are simple objects that model the data you are transferring into and out of the database
2. The data access object (DAO), that provides the description of the Java API that you want for working with certain entities
3. The database, which ties together all of the entities and DAOs for a single SQLite database

### Entities

In many ORM systems, the entity (or that system's equivalent) is a simple object

## STORING DATA IN A ROOM

---

that you happen to want to store in the database. It usually represents some part of your overall domain, so a payroll system might have entities representing departments, employees, and paychecks.

With Room, a better description of entities is that they are simple objects representing tables in your database, with one Java field/Kotlin property usually mapping to one column in that table.

From a coding standpoint, an entity is a class marked with the `@Entity` annotation, such as the `BookmarkEntity` class:

```
package com.commonsware.jetpack.bookmarker;

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

@Entity
public class BookmarkEntity {
 @PrimaryKey
 @NonNull
 public String pageTitle;
 public String title;
 public String iconUrl;
}
```

(from [Bookmarker/src/main/java/com/commonsware/jetpack/bookmarker/BookmarkEntity.java](#))

```
package com.commonsware.jetpack.bookmarker

import androidx.room.Entity
import androidx.room.PrimaryKey

@Entity
class BookmarkEntity {
 @PrimaryKey
 var pageTitle: String = ""
 var title: String? = null
 var iconUrl: String? = null
}
```

(from [Bookmarker/src/main/java/com/commonsware/jetpack/bookmarker/BookmarkEntity.kt](#))

There is no particular superclass required for entities. The expectation is that often they will be simple objects, as we see here, where `BookmarkEntity` is a plain class with no superclass.

## STORING DATA IN A ROOM

---

Each of the Kotlin properties (or Java fields) of the class will map to columns in the database. Usually, this is a 1:1 mapping (each property gets its own column), though there are ways to change that if needed. By default, the column names match the names of the properties or fields, so in this case, we have three columns:

- pageUrl
- title
- iconUrl

Besides the `@Entity` annotation, the only absolute requirement of an entity is that a column be designated as the primary key, usually via the `@PrimaryKey` annotation. For Java code, this can just go on a field. A primary key needs to be unique — in this case, we do not want duplicate entries for the same page URL, so we will take steps to avoid adding more than one when we add bookmarks to the database. Note that a `@PrimaryKey` needs to be non-nullable — that is a SQLite requirement that Room (and its associated Lint checks) helps to enforce.

Beyond these annotations, the rest of the code in your entity class is simply in support of the app — Room does not need anything else.

## DAO

“Data access object” (DAO) is a fancy way of saying “the API into the data”. The idea is that you have a DAO that provides methods for the database operations that you need: queries, inserts, updates, deletes, whatever.

In Room, the DAO is identified by the `@Dao` annotation, applied to either an abstract class or an interface. The actual concrete implementation will be code-generated for you by the Room annotation processor.

The primary role of the `@Dao`-annotated abstract class or interface is to have one or more methods, with their own Room annotations, identifying what you want to do with the database and your entities. In the case of Bookmarker, we have a `BookmarkStore` interface that serves in this role:

```
package com.commonsware.jetpack.bookmarker;

import java.util.List;
import androidx.lifecycle.LiveData;
import androidx.room.Dao;
import androidx.room.Insert;
import androidx.room.OnConflictStrategy;
```

## STORING DATA IN A ROOM

---

```
import androidx.room.Query;

@Dao
public interface BookmarkStore {
 @Query("SELECT * FROM BookmarkEntity ORDER BY title")
 LiveData<List<BookmarkEntity>> all();

 @Insert(onConflict = OnConflictStrategy.REPLACE)
 void save(BookmarkEntity entity);
}
```

(from [Bookmarker/src/main/java/com/commonsware/jetpack/bookmarker/BookmarkStore.java](#))

```
package com.commonsware.jetpack.bookmarker

import androidx.lifecycle.LiveData
import androidx.room.Dao
import androidx.room.Insert
import androidx.room.OnConflictStrategy
import androidx.room.Query

@Dao
interface BookmarkStore {
 @Query("SELECT * FROM BookmarkEntity ORDER BY title")
 fun all(): LiveData<List<BookmarkEntity>>

 @Insert(onConflict = OnConflictStrategy.REPLACE)
 suspend fun save(entity: BookmarkEntity)
}
```

(from [Bookmarker/src/main/java/com/commonsware/jetpack/bookmarker/BookmarkStore.kt](#))

Most, if not all, functions on a @Dao-annotated interface will have their own Room annotations. In the case of BookmarkStore, both of the functions have a Room annotation, indicating what sort of code Room should generate for us to serve as an implementation. We do not write the SQLite access code ourselves — instead, Room handles that, while we just use the API that we declare in the @Dao.

### @Query

One of our functions has a @Query annotation. Typically, these are for SQL SELECT statements, though in principle a @Query annotation can be used for any SQL. The property of the annotation contains the SQL statement to be executed. Note that this is a SQL statement and needs to use the table and column names. By default, the table name is the name of the entity class, and the column names are the names

## STORING DATA IN A ROOM

---

of the fields or properties, so it *looks* like you referencing the entity itself.

Room supports a wide variety of return types for a query:

- A query can return a single entity, for cases where there should be at most one result
- A query can return a list of entities, for cases where there may be many matches
- A query can return other types than entities, for cases where your SQL does not match an entity (e.g., you are using aggregation functions like SUM)

If a query returns those sorts of types directly, then the function will be synchronous, blocking until the database I/O is completed. If, however, the query returns the type wrapped in LiveData, the function will perform the database I/O asynchronously — you will get the results delivered to your LiveData observer when they are ready. Also, with LiveData, if Room thinks that the data may have been altered, any active observers will get new results delivered to them automatically, without you having to call the function again.

### **@Insert, @Update, and @Delete**

Our other function has an @Insert annotations. This performs a SQL INSERT statement, for the entity or entities provided as parameters to the function. There are also @Update and @Delete annotations, mapping to a SQL UPDATE or DELETE statement, but BookmarkStore is not using those.

Instead, the app uses the save() function for both inserts and updates. This works courtesy of the `onConflict = OnConflictStrategy.REPLACE` property on the @Insert annotation, which says “if there already is a row in the table for this primary key, replace it with the contents of the entity”. So, when you call save(), either it will insert a new row or overwrite the contents of an existing row, depending on whether the `pageUrl` value on the entity is already used in the table or not.

`save()` is marked with the `suspend` keyword. The Room annotation processor will detect this and will generate a coroutine for the implementations of `save()`. Room will handle setting up the background thread for you. Without the `suspend` keyword, these functions would be synchronous, blocking until the database I/O completed.

## STORING DATA IN A ROOM

---

### Database

In addition to entities and DAOs, you will have at least one @Database-annotated abstract class, extending a RoomDatabase base class. This class knits together the database file, the entities, and the DAOs. In the case of Bookmarker, BookmarkDatabase fills this role:

```
package com.commonsware.jetpack.bookmarker;

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

@Database(entities = {BookmarkEntity.class}, version = 1)
abstract class BookmarkDatabase extends RoomDatabase {
 private static final String DB_NAME = "bookmarks.db";
 private static volatile BookmarkDatabase INSTANCE;

 synchronized static BookmarkDatabase get(Context context) {
 if (INSTANCE == null) {
 INSTANCE =
 Room.databaseBuilder(context, BookmarkDatabase.class, DB_NAME).build();
 }

 return INSTANCE;
 }

 abstract BookmarkStore bookmarkStore();
}
```

(from [Bookmarker/src/main/java/com/commonsware/jetpack/bookmarker/BookmarkDatabase.java](#))

```
package com.commonsware.jetpack.bookmarker

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

private const val DB_NAME = "bookmarks.db"

@Database(entities = [BookmarkEntity::class], version = 1)
abstract class BookmarkDatabase : RoomDatabase() {

 abstract fun bookmarkStore(): BookmarkStore
```

## STORING DATA IN A ROOM

---

```
companion object {
 @Volatile
 private var INSTANCE: BookmarkDatabase? = null

 @Synchronized
 operator fun get(context: Context): BookmarkDatabase {
 if (INSTANCE == null) {
 INSTANCE =
 Room.databaseBuilder(context, BookmarkDatabase::class.java, DB_NAME)
 .build()
 }

 return INSTANCE!!
 }
}
```

(from [Bookmarker/src/main/java/com/commonsware/jetpack/bookmarker/BookmarkDatabase.kt](#))

There are two mandatory properties on a `@Database` annotation:

- `entities`, providing a list of all of the entity classes whose tables should go into this database
- `version`, providing a version number for the database schema (increment this number when you ship a newer app with a newer set of entities)

Just as the `@Database` annotation lists the entities, each associated `@Dao`-annotated class also needs to be tied into the `RoomDatabase` subclass. Specifically, you need an abstract function that returns an instance of the `@Dao`-annotated type. The name of the function can be whatever you want, as Room only cares about the return type. In a typical app, every entity and every DAO is handled by a single `RoomDatabase`, but you can have more than one if needed (e.g., one from a library and one for your own app's entities).

To retrieve an instance of the generated subclass of `BookmarkDatabase`, we need to call a function on the `Room` class:

- `databaseBuilder()` returns a `RoomDatabase.Builder` that will create a database in the file specified by the filename parameter (`DB_NAME` in the sample)
- `inMemoryDatabaseBuilder()` returns a `RoomDatabase.Builder` that will create an in-memory database, useful for your test code

`inMemoryDatabaseBuilder()` is great for testing, as it is fast and disposable. In our

## STORING DATA IN A ROOM

---

case, we are using `databaseBuilder()`, and using its result to set up a singleton instance of `BookmarkDatabase`.

### Tying It All Together

Given all of this, your code “simply” needs to:

- Obtain an instance of your `RoomDatabase` subclass
- Call the function(s) on it to obtain your DAO objects (e.g., `bookmarkStore()`)
- Call the functions(s) on the DAO to perform database operations, including observing any `LiveData` results or calling suspend functions inside a suitable coroutine scope

We have a `BookmarkRepository` which does all of that, and a bit more, as we will explore shortly.

### Other Fun Stuff in the App

Of course, this app has more to it than just a Room database, because that would not be sufficient to meet our needs. Plus, it would be really boring.

#### The Activity and ACTION\_SEND

If you launch the app from the launcher icon (or your IDE), you will see a list of bookmarks. Initially, that list will be empty, and there is no obvious way to add bookmarks to it.

## STORING DATA IN A ROOM

However, if you open up a Web browser on the device, browse to a page, and choose the browser's "Share" option (e.g., in an overflow menu), Bookmarker should show up as an option. If you choose it, our activity will pop up, and you should see your bookmark:

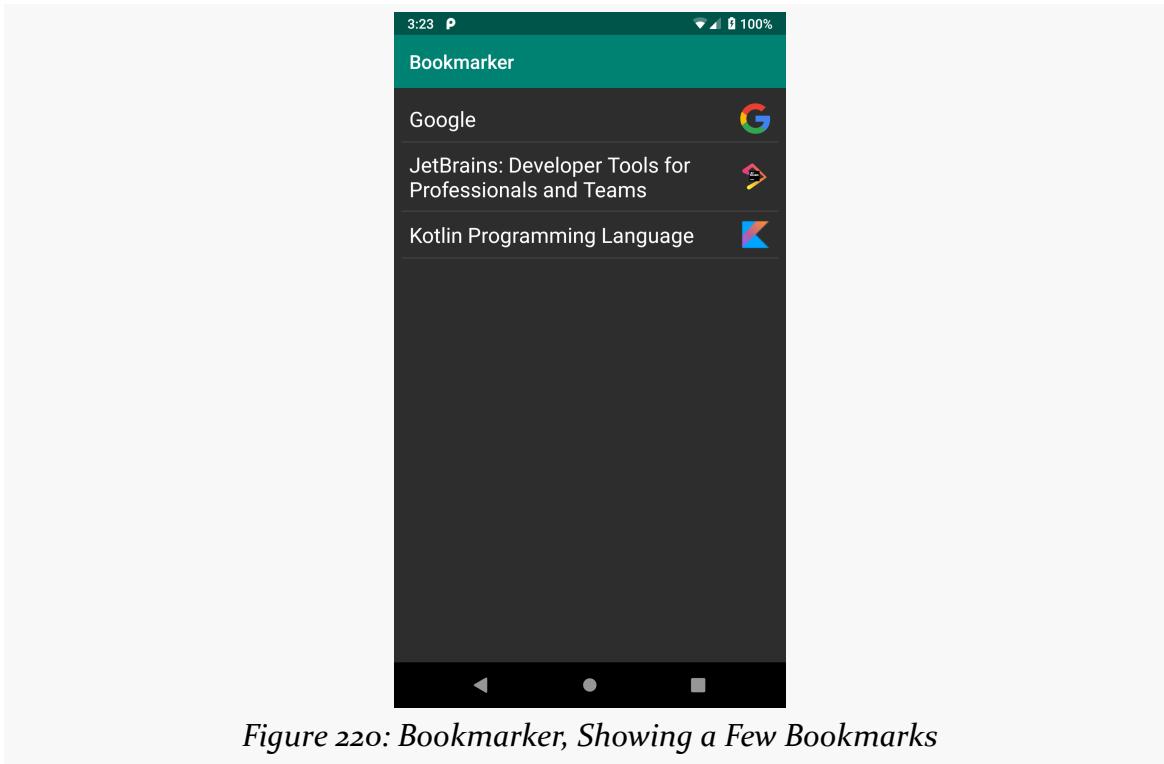


Figure 220: Bookmarker, Showing a Few Bookmarks

Our activity, in the manifest, has *two* <intent-filter> elements. One is the standard one to get the launcher icon. The other is for a different Intent action: ACTION\_SEND.

```
<activity android:name=".MainActivity">
 <intent-filter>
 <action android:name="android.intent.action.MAIN" />
 <category android:name="android.intent.category.LAUNCHER" />
 </intent-filter>
 <intent-filter>
 <action android:name="android.intent.action.SEND" />
 <category android:name="android.intent.category.DEFAULT" />
 <data android:mimeType="text/plain" />
 </intent-filter>
</activity>
```

(from [Bookmarker/src/main/AndroidManifest.xml](#))

## STORING DATA IN A ROOM

---

Here we are saying that we want our activity to be started either for the standard launcher icon Intent (ACTION\_MAIN and CATEGORY\_LAUNCHER), but also for an Intent that has:

- ACTION\_SEND as the action
- CATEGORY\_DEFAULT as the category
- text/plain for the MIME type

Browsers that implement a “share” option often use that particular Intent structure, and from there, we can get the URL of the active Web page.

In our MainActivity, as part of onCreate() processing, we call a saveBookmark() method:

```
saveBookmark(getIntent());
```

(from [Bookmarker/src/main/java/com/commonsware/jetpack/bookmarker/MainActivity.java](#))

```
saveBookmark(intent)
```

(from [Bookmarker/src/main/java/com/commonsware/jetpack/bookmarker/MainActivity.kt](#))

getIntent() is how we get the Intent that was used to create this activity instance — we pass that along to saveBookmark() for processing.

saveBookmark() will try to get the URL, and if it finds one, it will pass it along to save() on our MainMotor:

```
private void saveBookmark(Intent intent) {
 if (Intent.ACTION_SEND.equals(intent.getAction())) {
 String pageTitle = getIntent().getStringExtra(Intent.EXTRA_SUBJECT);
 String pageUrl = getIntent().getStringExtra(Intent.EXTRA_STREAM);

 if (pageTitle == null) {
 pageTitle = getIntent().getStringExtra(Intent.EXTRA_TEXT);
 }

 if (pageTitle != null &&
 Uri.parse(pageTitle).getScheme().startsWith("http")) {
 motor.save(pageTitle);
 }
 } else {
 Toast.makeText(this, R.string.msg_invalid_url,
 Toast.LENGTH_LONG).show();
 finish();
 }
}
```

## STORING DATA IN A ROOM

---

```
}
```

(from [Bookmarker/src/main/java/com/commonsware/jetpack/bookmarker/MainActivity.java](#))

```
private fun saveBookmark(intent: Intent) {
 if (Intent.ACTION_SEND == intent.action) {
 val pageTitle = getIntent().getStringExtra(Intent.EXTRA_STREAM)
 ?: getIntent().getStringExtra(Intent.EXTRA_TEXT)

 if (pageTitle != null && Uri.parse(pageTitle).scheme!! .startsWith("http")) {
 motor.save(pageTitle)
 } else {
 Toast.makeText(this, R.string.msg_invalid_url, Toast.LENGTH_LONG).show()
 finish()
 }
 }
}
```

(from [Bookmarker/src/main/java/com/commonsware/jetpack/bookmarker/MainActivity.kt](#))

First, we check the action of the Intent and see if it is ACTION\_SEND. If it is, we look in two “extras” of the Intent to try to find the URL:

- First, we check EXTRA\_STREAM, which if it exists is always supposed to be a string representation of a Uri
- If we did not find one there, we then check EXTRA\_TEXT, which could be any sort of text

In principle, for a text/plain ACTION\_SEND request, we should get one of those. We then see if it looks plausible as a URL (starts with http) — if it is, we forward it to the MainMotor via save(), which in turn forwards it to a BookmarkRepository and its save() function.

## The Repository

We seem to have a slight gap in our data, though. What we get from ACTION\_SEND is a URL. What we have in our BookmarkEntity is a pageTitle, but also a title and an iconUrl. We need to derive values for title and iconUrl given the page URL.

## Making Some (J)Soup

The way a Web browser gets the title and icon for a Web page is by parsing the HTML and looking for things like a <title> element in the <head> element. We will need to do the same thing. Fortunately, JSoup can help.

## STORING DATA IN A ROOM

---

[JSoup](#) is an HTML parser for Java (and, by extension, for Kotlin/JVM). It deals with a lot of the quirks with HTML and gives us a way to navigate the document contents to find things of interest.

This app pulls in JSoup as a dependency:

```
implementation 'org.jsoup:jsoup:1.12.1'
```

(from [Bookmarker/build.gradle](#))

We then use JSoup to attempt to read in this Web page and find our title and icon. In the case of the Java implementation of BookmarkRepository, that is part of a SaveLiveData that does the work on a background thread supplied by an Executor:

```
private static class SaveLiveData extends LiveData<BookmarkResult> {
 private final String pageTitle;
 private final Executor executor;
 private final BookmarkStore store;

 SaveLiveData(String pageTitle, Executor executor, BookmarkStore store) {
 this.pageTitle = pageTitle;
 this.executor = executor;
 this.store = store;
 }

 @Override
 protected void onActive() {
 super.onActive();

 executor.execute(() -> {
 try {
 BookmarkEntity entity = new BookmarkEntity();
 Document doc = Jsoup.connect(pageTitle).get();

 entity.pageTitle = pageTitle;
 entity.title = doc.title();

 // based on https://www.mkyong.com/java/jsoup-get-favicon-from-html-page/

 String iconUrl = null;
 Element candidate = doc.head().select("link[href~=.*\.(ico|png)]").first();

 if (candidate == null) {
 candidate = doc.head().select("meta[itemprop=image]").first();

 if (candidate != null) {
 iconUrl = candidate.attr("content");
 }
 } else {
 iconUrl = candidate.attr("href");
 }

 if (iconUrl != null) {
 URI uri = new URI(pageTitle);
```

## STORING DATA IN A ROOM

---

```
 entity.iconUrl = uri.resolve(iconUrl).toString();
 }

 store.save(entity);

 .postValue(new BookmarkResult(new BookmarkModel(entity), null));
}
catch (Throwable t) {
 .postValue(new BookmarkResult(null, t));
}
});
}
}
```

(from [Bookmarker/src/main/java/com/commonsware/jetpack/bookmarker/BookmarkRepository.java](#))

In the case of Kotlin, the `save()` function on `BookmarkRepository` handles it directly in a coroutine:

```
suspend fun save(context: Context, pageUrl: String) =
 withContext(Dispatchers.IO) {
 val db: BookmarkDatabase = BookmarkDatabase[context]
 val entity = BookmarkEntity()
 val doc = Jsoup.connect(pageUrl).get()

 entity.pageUrl = pageUrl
 entity.title = doc.title()

 // based on https://www.mkyong.com/java/jsoup-get-favicon-from-html-page/

 val iconUrl: String? =
 doc.head().select("link[href~=.*\.(ico|png)]").first()?.attr("href")
 ?: doc.head().select("meta[itemprop=image]").first().attr("content")

 if (iconUrl != null) {
 val uri = URI(pageUrl)

 entity.iconUrl = uri.resolve(iconUrl).toString()
 }

 db.bookmarkStore().save(entity)

 BookmarkModel(entity)
 }
```

(from [Bookmarker/src/main/java/com/commonsware/jetpack/bookmarker/BookmarkRepository.kt](#))

`JSoup.connect(pageUrl).get()` will synchronously retrieve the HTML page and parse it, throwing an exception if there is some sort of problem (e.g., cannot retrieve

## STORING DATA IN A ROOM

---

the page). The Document object that we get back has a `title()` function to get the page title, which is nice and easy. To get the icon URL, we have to use [some funky code, owing to multiple standards](#). In general, we use `head()` to get to the `<head>` section of the Web page, then use `select()` to use a CSS selector to find a particular element in that page, then use `attr()` to retrieve an attribute from the `first()` element matching that CSS selector. If we get a value, it could be an absolute URL (e.g., <https://somebody.com/favicon.png>) or a relative URL (e.g., `/favicon.png`). So we use `java.net.URI` to get an absolute URL given the page URL and the raw icon URL.

We then put the two URLs and the title into a `BookmarkEntity` and `save()` it using our `BookmarkStore`.

## What Else Does Room Offer?

Room has a wide array of other capabilities, besides the simple CRUD (create-read-update-delete) operations seen in these to-do classes, such as:

- You can add properties to the `@Entity` annotation to rename the table (`tableName`), define indexes (`indices`), and so on
- `@Query`-annotated functions that have parameters can inject those parameters into the SQL:

```
@Query("SELECT * FROM todos WHERE id = :modelId")
fun find(modelId: String): LiveData<ToDoEntity>
```

- `@Insert`, `@Update`, and `@Delete` functions can accept a single entity, a `List` of entities, or a variable argument list of entities (e.g., `vararg` in Kotlin)
- Using `@TypeConverter` and `@TypeConverters` annotations, you can teach Room how to save arbitrary data types into a single column, such as converting a `Calendar` or `LocalDate` into a `Long` or `string`
- Room supports `@ForeignKey` annotations to describe relations between two entity classes... but the entities cannot directly refer to the other entities via properties or fields. Room requires you to retrieve those related objects separately and tie them together on your own.
- When you do update your database schema by adding, changing, or removing entities, you may need to run some code to convert the older database contents to the new schema when the user upgrades your app. Room has a `Migration` interface and related code to help you set up those conversions.

## STORING DATA IN A ROOM

---

- In addition to LiveData and Kotlin coroutines, Room also has add-on libraries that add support for RxJava (e.g., queries returning an Observable or Single).
- By default, each SQL statement runs in its own transaction. However, through things like the @Transaction annotation, you can define your own custom transaction boundaries, where you have several SQL operations that should succeed or fail as a whole.
- Room, in conjunction with the Jetpack Paging library, supports progressive loading of large data sets, typically based on a user scrolling through a list, to minimize how much memory is needed at any one point in time.
- Room supports a pluggable database implementation. It defaults to the standard copy of SQLite that is available to any Android app, but you can replace that with some other implementation, such as [SQLCipher for Android](#) for an encrypted database.
- Room has support for SQLite's full-text search (FTS) engine and other SQL constructs, such as views.

# Inverting Your Dependencies

---

Android app development winds up using a fair number of singletons, such as the repositories that we have used in several preceding chapters. So long as those singletons do not result in an unexpected memory leak, and so long as they are part of some organized architecture, singletons can be fine.

The problems kick in when you need different “singletons” in different situations. For example, you might need a singleton that responds in certain ways for testing, compared to the “real” singleton that you need for your production app use.

So, even though singletons are largely unavoidable in modern Android app development, how we declare and get those singletons can vary. So far, our repositories have been static fields in Java or object classes in Kotlin. Those are simple but inflexible. Dependency inversion allows us to still use singletons, yet be able to replace them as needed, such as for testing.

## The Problem: Test Control

Let’s revisit the diceware samples from earlier in the book. In both `DiceLight` and `Diceware`, we have a `PassphraseRepository`. In Kotlin, it is declared as an object; in Java, it is declared as a class with a static field holding a singleton instance. And, both work, using a `SecureRandom` as the source of random numbers for use in generating the passphrase.

But now, suppose we want to test that code.

Random numbers are messy when it comes to tests. Truly random numbers are not repeatable, and so that makes our tests difficult to write. The way you test data backed by random numbers is to “seed” the random number generator, so that

## INVERTING YOUR DEPENDENCIES

---

while it generates random numbers, those random numbers are consistent across uses of the seed. Two random number generators started with the same seed will generate the same random numbers. Now, we have repeatable data, suitable for testing... but we do *not* want to use a fixed seed for the actual production use of the app. In the case of `SecureRandom`, it will seed itself from “sources of entropy” by default, thereby providing more truly random numbers.

So, for testing, we want a manually-seeded `SecureRandom`. For production, we want an automatically-seeded `SecureRandom`. Yet our singleton repository has just the one `SecureRandom`, and we do not have a great way in the `PassphraseRepository` to know whether we are running as part of a test or not.

We could say that the `PassphraseRepository` takes the `SecureRandom` as a parameter. Tests can pass in the seeded `SecureRandom`; production code could pass in a regular `SecureRandom`. For this limited scenario, we could make that work. However, this becomes unwieldy if we have hundreds of things that vary based on testing and need to pass those objects through several layers of app code.

## The Solution: Dependency Inversion

What we want is to be able to configure `PassphraseRepository` at runtime in two different ways:

- If we are running normally, use a regular `SecureRandom`
- If we are running in a test, use a seeded `SecureRandom`

The recommended approach for this sort of pattern is to use dependency inversion. Basically, rather than `PassphraseRepository` creating a `SecureRandom` or having one passed in on every call, `PassphraseRepository` gets *created* with the desired `SecureRandom` instance. We push dependencies (`SecureRandom`) into the things that use those dependencies (`PassphraseRepository`) as part of setting up the singletons.

Dependency inversion is usually implemented using “dependency injectors”, “service locators”, or hybrid solutions. The details of the differences between those specific approaches is beyond the scope of this book — see [this Stack Overflow discussion](#) as an example of the long debates that are had regarding them.

## Dependency Inversion in Android

When it comes to Android app development, there are a few dependency inversion options that have become popular.

### Java: Dagger 2

The *de facto* dependency injection standard for Java app development is [Dagger](#). It is very powerful, but that power comes with a lot of confusion and complexity.

Dagger uses annotations to indicate how the dependencies get connected. In our example, annotations would indicate that PassphraseRepository needs a SecureRandom and how that SecureRandom gets created, both for regular code and for tests. Dagger then generates the “glue code” needed to connect those dependencies.

### Kotlin: Koin and Kodein

You can use Dagger for Kotlin code as well, though the confusion and complexity increases. For Kotlin-centric projects, there are other options that are simpler to use, though they focus more on runtime dependency inversion rather than compile-time code generation.

The two most popular options for Kotlin appear to be [Kodein’s dependency injector](#) and [Koin](#). In this chapter, we will explore the use of Koin.

Koin can be used in Java projects, but it is definitely geared around use in Kotlin. As such, we will only look at Koin in a Kotlin project. The concepts of dependency inversion still hold true for Java, but the implementation in something like Dagger would look quite a bit different.

## Applying Koin

With all that in mind, let’s look at the DiceKoin sample module in the [Sampler](#) project, to see how Koin works and what problems it solves.

### The Dependency

As with most things, Koin comes from a library. Actually, Koin is made up of several

## INVERTING YOUR DEPENDENCIES

libraries, to handle different scenarios. For example, you can use Koin for non-Android projects using Kotlin.

In our case, we are using koin-androidx-viewmodel:

```
implementation "org.koin:koin-androidx-viewmodel:2.0.1"
```

(from [DiceKoin/build.gradle](#))

This not only pulls in Koin and Koin's support for Android, but it offers specific support for the Jetpack edition of ViewModel, as we will see.

## PassphraseRepository

PassphraseRepository is now a class, not an object as it was before. And, it gets a Context and a SecureRandom in its constructor. This allows us to avoid the Context parameter on functions like generate() and the locally-initialized SecureRandom instance:

```
package com.commonsware.jetpack.diceware

import android.content.Context
import android.net.Uri
import android.util.LruCache
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.withContext
import java.io.BufferedReader
import java.io.InputStream
import java.io.InputStreamReader
import java.security.SecureRandom
import java.util.*

val ASSET_URI: Uri =
 Uri.parse("file:///android_asset/eff_short_wordlist_2_0.txt")
private const val ASSET_FILENAME = "eff_short_wordlist_2_0.txt"

class PassphraseRepository(
 private val context: Context,
 private val random: SecureRandom
) {
 private val wordsCache = LruCache<Uri, List<String>>(4)

 suspend fun generate(wordsDoc: Uri, count: Int): List<String> {
 var words: List<String>?

 synchronized(wordsCache) {
 words = wordsCache.get(wordsDoc)
 }

 return words?.let { rollDemBones(it, count) }
 ?: loadAndGenerate(wordsDoc, count)
 }
}
```

## INVERTING YOUR DEPENDENCIES

```
private suspend fun loadAndGenerate(wordsDoc: Uri, count: Int): List<String> =
 withContext(Dispatchers.IO) {
 val inputStream: InputStream? = if (wordsDoc == ASSET_URI) {
 context.assets.open(ASSET_FILENAME)
 } else {
 context.contentResolver.openInputStream(wordsDoc)
 }

 inputStream?.use {
 val words = it.readLines()
 .map { line -> line.split("\t") }
 .filter { pieces -> pieces.size == 2 }
 .map { pieces -> pieces[1] }

 synchronized(wordsCache) {
 wordsCache.put(wordsDoc, words)
 }

 rollDemBones(words, count)
 } ?: throw IllegalStateException("could not open $wordsDoc")
 }

private fun rollDemBones(words: List<String>, wordCount: Int) =
 List(wordCount) {
 words[random.nextInt(words.size)]
 }

private fun InputStream.readLines(): List<String> {
 val result = ArrayList<String>()

 BufferedReader(InputStreamReader(this)).forEachLine { result.add(it); }

 return result
}
}
```

(from [DiceKoin/src/main/java/com/commonsware/jetpack/diceware/PassphraseRepository.kt](#))

Otherwise, PassphraseRepository is the same as it was, in terms of API and functionality.

## MainMotor

MainMotor has a couple of changes as well:

- It gets a PassphraseRepository via its constructor, rather than referencing the former object form of the repository
- It no longer needs to pass a Context in its calls to the repository, so it can be a regular ViewModel instead of an AndroidViewModel

```
package com.commonsware.jetpack.diceware
```

## INVERTING YOUR DEPENDENCIES

---

```
import android.net.Uri
import androidx.lifecycle.LiveData
import androidx.lifecycle.MutableLiveData
import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.launch

class MainMotor(private val repo: PassphraseRepository) : ViewModel() {
 private val _results = MutableLiveData<MainViewState>()
 val results: LiveData<MainViewState> = _results
 private var wordsDoc = ASSET_URI

 fun generatePassphrase(wordCount: Int) {
 _results.value = MainViewState.Loading

 viewModelScope.launch(Dispatchers.Main) {
 _results.value = try {
 val randomWords = repo.generate(wordsDoc, wordCount)

 MainViewState.Content(randomWords.joinToString(" "))
 } catch (t: Throwable) {
 MainViewState.Error(t)
 }
 }
 }

 fun generatePassphrase(wordsDoc: Uri, wordCount: Int) {
 this.wordsDoc = wordsDoc

 generatePassphrase(wordCount)
 }
}
```

(from [DiceKoin/src/main/java/com/commonsware/jetpack/diceware/MainMotor.kt](#))

Otherwise, it too is unchanged from before.

## KoinApp

At some point, though, *something* needs to be creating an instance of `PassphraseRepository`. `MainMotor` is not doing that — it is expecting something else to create the instance and supply it to the `MainMotor` constructor. Similarly, *something* needs to be creating that `SecureRandom` instance to pass to `PassphraseRepository`, whenever somebody gets around to making that repository instance.

## INVERTING YOUR DEPENDENCIES

The “something” is Koin.

Compared with the Diceware sample, DiceKoin has one new class: KoinApp:

```
package com.commonsware.jetpack.diceware

import android.app.Application
import org.koin.android.ext.koin.androidContext
import org.koin.android.ext.koin.androidLogger
import org.koin.androidx.viewmodel.dsl.viewModel
import org.koin.core.context.startKoin
import org.koin.dsl.module
import java.security.SecureRandom

private val MODULE = module {
 single { SecureRandom() }
 single { PassphraseRepository(androidContext(), get()) }
 viewModel { MainMotor(get()) }
}

class KoinApp : Application() {
 override fun onCreate() {
 super.onCreate()

 startKoin {
 androidLogger()
 androidContext(this@KoinApp)
 modules(MODULE)
 }
 }
}
```

(from [DiceKoin/src/main/java/com/commonsware/jetpack/diceware/KoinApp.kt](#))

### Subclass of Application

We have seen the Application object before. It is a process-wide singleton instance that we can access at various points, such as calling `getApplication()` on an Activity or AndroidViewModel. By default, it is an instance of `android.app.Application`.

However, you can create your own subclass of Application and use it instead.

The big reason to do this is to get control every time your process is forked. Application has an `onCreate()` function, much like how an Activity does.

## INVERTING YOUR DEPENDENCIES

`onCreate()` on an Activity is called when that Activity is created — similarly, `onCreate()` on an Application is called when that Application is created. Since the Application singleton is created when your process starts, your code in `onCreate()` will get called when your process starts.

As a result, we tend to use a custom Application subclass for cases where we need to do some process-wide initialization. Setting up dependency inversion, whether using Koin or something else, is usually done in a custom Application subclass.

### Referenced in the Manifest

To tell Android to create an instance of your Application subclass when your process starts, you need to add an `android:name` attribute to the `<application>` element itself in the manifest:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:tools="http://schemas.android.com/tools"
 package="com.commonsware.jetpack.diceware"
 xmlns:android="http://schemas.android.com/apk/res/android">

 <application
 android:name=".KoinApp"
 android:allowBackup="false"
 android:icon="@mipmap/ic_launcher"
 android:label="@string/app_name"
 android:roundIcon="@mipmap/ic_launcher_round"
 android:supportsRtl="true"
 android:theme="@style/AppTheme"
 tools:ignore="GoogleAppIndexingWarning">
 <activity android:name=".MainActivity">
 <intent-filter>
 <action android:name="android.intent.action.MAIN" />

 <category android:name="android.intent.category.LAUNCHER" />
 </intent-filter>
 </activity>
 </application>

</manifest>
```

(from [DiceKoin/src/main/AndroidManifest.xml](#))

Previously, we had only used `android:name` for things like `<activity>`. However, `android:name` fills the same role: identifying the Java or Kotlin class that Android should use for this component.

## INVERTING YOUR DEPENDENCIES

---

An `<application>` element without `android:name` defaults to using `android.app.Application`. In our case, we are telling Android to use this `KoinApp` class.

### Defines a Module

Part of what `KoinApp` does is define a Koin “module”. A module states what objects Koin can make available to parts of your app.

The simplest way to declare a Koin module is via the `module()` top-level function. This takes a lambda expression, in which you have statements that set up each of the Koin-managed objects (or factories of objects, as we will see).

Two of those statements are calls to `single()`:

```
single { SecureRandom() }
single { PassphraseRepository(androidContext(), get()) }
```

(from [DiceKoin/src/main/java/com/commonsware/jetpack/diceware/KoinApp.kt](#))

`single()` in a `module()` says “hey, Koin, when something asks for an object, here is one that you can use... but only have one of it for the entire app”. `single()` takes a lambda expression that creates an instance of some object, and `single()` knows what type of object that is. When something needs an object of that type, Koin can execute that lambda expression (if needed) and hand over that singleton instance.

The first `single()` call sets up a Koin-managed singleton instance of `SecureRandom`. The second `single()` call sets up a Koin-managed singleton instance of `PassphraseRepository`.

Our lambda expression for creating the `PassphraseRepository` instance needs to provide a Context and a `SecureRandom` to the `PassphraseRepository` constructor. For the Context, we use `androidContext()`, which says, “hey, Koin, you should have a Context that we can use here!” — we will see where that Context comes from shortly. For the `SecureRandom`, we use `get()`, which says “hey, Koin, search this module for a supplier of `SecureRandom` objects, and use that here!”. In our case, we set up a `SecureRandom` singleton on the preceding line. So, Koin will take that singleton `SecureRandom` and pass it to `PassphraseRepository`, when it is time to create our singleton instance of `PassphraseRepository`.

The third statement in our module is not a `single()`, but a `viewModel()`:

## INVERTING YOUR DEPENDENCIES

```
viewModel { MainMotor(get()) }
```

(from [DiceKoin/src/main/java/com/commonsware/jetpack/diceware/KoinApp.kt](#))

Courtesy of our `koin-androidx-viewmodel`, this ties into the Jetpack `ViewModel` system. Our lambda expression needs to return a `ViewModel`, and in this case it creates an instance of `MainMotor`. `MainMotor` takes a `PassphraseRepository` instance as a constructor parameter, and our `get()` call causes Koin to retrieve the `PassphraseRepository` singleton, creating it if it does not already exist.

Note that `viewModel` is not creating a singleton instance. Rather, it will defer to the Jetpack `ViewModel` system to return the proper `ViewModel` for the activity or fragment that might request one.

## Starts Koin

The `onCreate()` function in `KoinApp` chains to the superclass implementation of `onCreate()`, then calls `startKoin()`:

```
override fun onCreate() {
 super.onCreate()

 startKoin {
 androidLogger()
 androidContext(this@KoinApp)
 modules(MODULE)
 }
}
```

(from [DiceKoin/src/main/java/com/commonsware/jetpack/diceware/KoinApp.kt](#))

As the name suggests, `startKoin()` starts Koin. By doing that here in `onCreate()`, Koin will be ready for use in the rest of our app. Like `module()`, `startKoin()` takes a lambda expression where we can have a series of statements to describe our Koin configuration. Here, we have three:

- `androidLogger()`, telling Koin that if it has any messages to log, use Logcat
- `androidContext()`, telling Koin what Context to return from  
`androidContext()` calls in our module definition — in this case, we provide the `KoinApp` itself
- `modules()`, where we can provide one or more modules that we want Koin to support

## INVERTING YOUR DEPENDENCIES

### MainActivity

MainActivity now needs to use Koin to get its MainMotor. Previously, we had `motor` as a `lateinit` var, then used `ViewModelProviders` to get our instance. Now... we inject one:

```
private val motor: MainMotor by inject()
```

(from [DiceKoin/src/main/java/com/commonsware/jetpack/diceware/MainActivity.kt](#))

`inject()` is supplied by Koin. It returns a Kotlin property delegate for the type of the property on which we use `inject()`.



You can learn more about property delegates in the "Property Delegates" chapter of [\*Elements of Kotlin\*](#)!

The net effect is that the first time we try using `motor`, `inject()` will look in the Koin modules to see if it knows how to create an object of the desired type. In this case, we configured how to create a `MainMotor`.

### The Dependency Inversion Flow

Here's how this works in practice, when the user taps our launcher icon.

First, `onCreate()` of `KoinApp` gets called. There, we set up our Koin module. However, none of those Koin-managed objects are actually created yet. For example, the `PassphraseRepository` is not created right away, nor is any instance of `MainMotor`.

Eventually, `MainActivity` gets instantiated. When we initialize `motor` using `by inject()`, Koin sets up a property delegate to retrieve a `MainMotor` when needed.

We then reference `motor` in `onCreate()` of `MainActivity` as before:

```
override fun onCreate(savedInstanceState: Bundle?) {
 super.onCreate(savedInstanceState)
 setContentView(R.layout.activity_main)

 motor.results.observe(this, Observer { viewState ->
 when (viewState) {
```

## INVERTING YOUR DEPENDENCIES

---

```
 MainViewState.Loading -> {
 progress.visibility = View.VISIBLE
 passphrase.text = ""
 }
 is MainViewState.Content -> {
 progress.visibility = View.GONE
 passphrase.text = viewState.passphrase
 }
 is MainViewState.Error -> {
 progress.visibility = View.GONE
 passphrase.text = viewState.throwable.localizedMessage
 Log.e(
 "Diceware",
 "Exception generating passphrase",
 viewState.throwable
)
 }
}
}

motor.generatePassphrase(wordCount)
}
```

(from [DiceKoin/src/main/java/com/commonsware/jetpack/diceware/MainActivity.kt](#))

When we do that, Koin's property delegate tries to retrieve an existing `MainMotor` instance for this activity. That will fail, as this is the first time we are creating a `MainMotor`. So Koin executes the lambda expression that we provided to `viewModel()`, intending to return that object as our `MainMotor` from the property delegate.

That lambda expression has a `get()` call for a parameter that needs a `PassphraseRepository`. So, Koin looks around the module and finds the `single()` for `PassphraseRepository`. Since we have not created an instance of that yet, Koin executes our lambda expression that we provided to `single()`, intending to return that object as our `PassphraseRepository` from our `get()` call.

*That* lambda expression in turn has a `get()` call that needs a `SecureRandom`. Once again, Koin examines the module for a match and finds the `SecureRandom single()`. Since we have not needed this before, Koin executes that lambda expression, saving that `SecureRandom` for future `get()` or `inject()` calls, and returns it from the `get()` call. That in turn allows the `PassphraseRepository` to be created and cached by Koin for future `get()` or `inject()` calls. And, *that* allows us to create the `MainMotor` and use it as our `ViewModel`.

## INVERTING YOUR DEPENDENCIES

---

If we rotate the screen and undergo a configuration change, our new `MainActivity` instance will once again use Koin to get its `MainMotor`. This time, Koin will return the previous `MainMotor` instance, courtesy of Jetpack's `ViewModel` system.

Suppose the user presses BACK, but then immediately re-launches our app (e.g., the BACK click was by accident). Now, when `MainActivity` asks Koin for a `MainMotor`, since this is a completely new activity instance, we get a completely new `MainMotor` instance. However, most likely, this is the same process as before, since the user had not been gone very long. So, we use the same `PassphraseRepository` and `SecureRandom` as before, since Koin caches them and reuses those instances, as instructed by the `single()` rule.

## What This Buys Us

In the end, that's not really much additional code... but it *is* a bit more complex than what we had originally. And it may not be obvious what we gained by it, since we still wind up with all the same objects as before, just connected in a different fashion.

When it comes time to test `MainMotor` or `MainActivity`, though, that is where dependency inversion starts to become important. As noted before, we will want a controlled test instance of `SecureRandom`, one with a stable seed so we get a stable set of random numbers.

When our instrumented tests run, our `KoinApp` still gets instantiated and still configures Koin — that does not change just because we are testing. However, after that occurs, and before we test our classes, we can *change the Koin configuration*. In particular, we can replace the self-seeding `SecureRandom` with a manually-seeded `SecureRandom`, so our tests become predictable. That code resides solely in our tests. `PassphraseRepository`, in particular, does not know about this change... because so long as it gets *some* `SecureRandom` instance, `PassphraseRepository` can do its work. So, rather than having to teach `PassphraseRepository` different rules for creating a `SecureRandom` instance ("in tests, create it this way, otherwise create it this other way"), `PassphraseRepository` receives a `SecureRandom`. It is our Koin configuration code — in `KoinApp` and our tests — that determine which type of `SecureRandom` we use.

We will explore this in greater detail [in an upcoming chapter](#).

Licensed solely for use by Patrocinio Rodriguez

# **Testing Your Changes**

This chapter will be added in a future update to this book!

Licensed solely for use by Patrocinio Rodriguez

# **Working with WorkManager**

---

We have seen how to do work on the main application thread, and we have seen how to use background threads for things like disk and network I/O.

Sometimes, though, we may need to do work that happens completely independently of the UI

- The user requests to download a large file but asks for the download to be performed overnight
- We want to synchronize with a server periodically, such as every couple of hours
- The user asks us to perform some slow on-device task, such as converting a video between formats, and we want to do that work when the device is plugged in and is not being actively used by the user
- And so on

Your primary option for those scenarios is `WorkManager`.

## **The Role of WorkManager**

This is not to say that `WorkManager` is the only solution for background work. `WorkManager` is designed for “deferrable” work — work that you need to have done but does not have to happen right away. This includes the possibility that the work will be done sometime after your current process has terminated, because the user left your app and a lot of time passes before it is time to do your work.

However, there are scenarios for which `WorkManager` is optimized. Alternative scenarios might be better handled using other techniques:

## WORKING WITH WORKMANAGER

- WorkManager is designed for discrete, “transactional” tasks, not ongoing work. So, for example, WorkManager is not designed to play music continuously in the background. A “foreground service” is the solution to use for that, with background threads as needed (e.g., for disk I/O to read in the playlist details).
- WorkManager is designed for work that will happen sometime, but not at some specific time. If you need to get control at a specific time — such as to alert the user about an upcoming calendar event — use `setExactAndAllowWhileIdle()` on AlarmManager.
- WorkManager is designed for work that will happen eventually, but perhaps not immediately. If you have background work that has to be done in real time in response to user input (e.g., download the video that they just purchased), use a simple thread. If you are concerned that the work might take too long, and your process might be terminated while that work is ongoing, use a foreground service.
- WorkManager is designed for work that might happen completely asynchronously with respect to your current process. Hence, it is not useful for cases where the work that you are doing only affects the current process, particularly its UI. So, for example, downloading avatar icons to display in your app may not make sense once the UI is gone, as you may never need those icons. For that, use a thread pool, reactive solutions (e.g., RxJava, Kotlin coroutines), or libraries that in turn use those sorts of things.

Foreground services and AlarmManager are powerful techniques, but ones that we will not be exploring in this particular book. [\*The Busy Coder's Guide to Android Development\*](#), while older, offers material on those subjects.

## WorkManager Dependencies

The main artifact that you will use for adding WorkManager to your project is `androidx.work:work-runtime`. This contains WorkManager and its related classes.

There are a few additional artifacts that you can elect to use, of which two are general-purpose:

- `work-runtime-ktx` provides a Kotlin-specific WorkManager API
- `work-testing` is available for [testing your code that uses WorkManager](#)

## Workers: They Do Work

The work that you want to have done in the background needs to be wrapped in a `Worker` subclass. This is an abstract class with one abstract method: `dowork()`. In that method, you put your work to be done in the background.

Note that:

- `dowork()` will keep the device awake, so you do not have to worry about the CPU powering down while your work is ongoing
- `dowork()` is called on a background thread, so you do not need to fork one yourself
- `dowork()` needs to return a `ListenableWorker.Result` object indicating if the work succeeded or failed, so `dowork()` should not be starting other background threads (directly or through libraries)
- `dowork()` cannot run forever — at best, it might run for 10 minutes before it is terminated, and it is possible that it will have less time than that

As noted above, `dowork()` returns a `ListenableWorker.Result` object. There are static factory methods on `ListenableWorker.Result` that you use to create instances. Those factory methods represent three main result scenarios:

- `success()`, which is what you are hoping for
- `retry()`, which indicates that for one reason or another you could not do the work but would like `WorkManager` to retry the work in a little while
- `failure()`, which indicates that the work could not be done and a later retry is likely to fail as well, so you are giving up

Beyond the return value and the aforementioned limitations on what you can do in `dowork()`, the actual business logic is up to you.

The `DownloadWork` sample module in the [Sampler](#) and [SamplerJ](#) projects contains a `DownloadWorker` class that downloads a file using `OkHttp` and `Okio`:

```
package com.commonsware.jetpack.work.download;

import android.content.Context;
import android.util.Log;
import java.io.File;
import java.io.IOException;
import androidx.annotation.NonNull;
import androidx.work.ListenableWorker;
```

## WORKING WITH WORKMANAGER

```
import androidx.work.Worker;
import androidx.work.WorkerParameters;
import okhttp3.OkHttpClient;
import okhttp3.Request;
import okhttp3.Response;
import okio.BufferedSink;
import okio.Okio;

public class DownloadWorker extends Worker {
 public static final String KEY_URL="url";
 public static final String KEY_FILENAME="filename";

 public DownloadWorker(@NonNull Context context,
 @NonNull WorkerParameters workerParams) {
 super(context, workerParams);
 }

 @NonNull
 @Override
 public Result doWork() {
 OkHttpClient client=new OkHttpClient();
 Request request=new Request.Builder()
 .url(getInputData().getString(KEY_URL))
 .build();

 try (Response response=client.newCall(request).execute()) {
 File dir=getApplicationContext().getCacheDir();
 File downloadedFile=
 new File(dir, getInputData().getString(KEY_FILENAME));
 BufferedSink sink=Okio.buffer(Okio.sink(downloadedFile));

 sink.writeAll(response.body().source());
 sink.close();
 }
 catch (IOException e) {
 Log.e(getClass().getSimpleName(), "Exception downloading file", e);
 }

 return ListenableWorker.Result.failure();
 }

 return ListenableWorker.Result.success();
}
}
```

(from [DownloadWork/src/main/java/com/commonsware/jetpack/work/download/DownloadWorker.java](#))

```
package com.commonsware.jetpack.work.download
```

## WORKING WITH WORKMANAGER

```
import android.content.Context
import android.util.Log
import androidx.work.Worker
import androidx.work.WorkerParameters
import okhttp3.OkHttpClient
import okhttp3.Request
import okio.buffer
import okio.sink
import java.io.File
import java.io.IOException

class DownloadWorker(context: Context, workerParams: WorkerParameters) :
 Worker(context, workerParams) {

 override fun doWork(): Result {
 val client = OkHttpClient()
 val request = Request.Builder()
 .url(inputData.getString(KEY_URL)!!)
 .build()

 try {
 client.newCall(request).execute().use { response ->
 val dir = applicationContext.cacheDir
 val downloadedFile = File(dir, inputData.getString(KEY_FILENAME)!!)
 val sink = downloadedFile.sink().buffer()

 response.body?.let { sink.writeAll(it.source()) }
 sink.close()
 }
 } catch (e: IOException) {
 Log.e(javaClass.simpleName, "Exception downloading file", e)
 }

 return Result.failure()
 }

 return Result.success()
}

companion object {
 const val KEY_URL = "url"
 const val KEY_FILENAME = "filename"
}
}
```

(from [DownloadWork/src/main/java/com/commonsware/jetpack/work/download/DownloadWorker.kt](#))

Worker — from which DownloadWorker inherits — has a two-parameter constructor, taking a Context and a WorkerParameters object. In many cases, you can just chain

## WORKING WITH WORKMANAGER

to the superclass constructor, as `DownloadWorker` does.

Pretty much everything inside of `dowork()` is just application code that does the download and returns `success()` if the download succeeded or `failure()` if there was some exception during the download.

This `dowork()` method is using two methods that we get from `Worker`:

- `getApplicationContext()`, which works like the similarly-named method on `Context`, returning you the `Application` singleton, in case you need a `Context`
- `getInputData()`, which we will examine more closely [later in the chapter](#)

## Performing Simple Work

Having a `Worker` is part of the puzzle. We still need to tell `WorkManager` to actually use that class to do our work.

If we want to perform the work once — perhaps in response to user input — we can create a `OneTimeWorkRequest` object to describe that work, then `enqueue()` it with `WorkManager`, as in this Java snippet:

```
OneTimeWorkRequest downloadWork=
 new OneTimeWorkRequest.Builder(DownloadWorker.class)
 .build();

WorkManager.getInstance(getApplicationContext()).enqueue(downloadWork);
```

We create a `OneTimeWorkRequest` via its associated `Builder`, which takes the Java `Class` object for our `Worker` subclass its constructor. We `build()` the `Builder` and pass the `OneTimeWorkRequest` to `enqueue()` on the `WorkManager` singleton, which we get by calling `getInstance()` on `WorkManager`.

After this code executes, at some point in time, an instance of `DownloadWorker` will be created and `dowork()` will be called. Exactly when that will be is indeterminate. In this particular case, *probably* it will be called fairly quickly, with `dowork()` being executed on a thread in a `WorkManager`-managed thread pool. However, it is entirely possible that the user leaves our app and our process is terminated before this work can begin. If so, `WorkManager` will arrange to have the work done later.

## Work Inputs

However, `doWork()` would crash if we scheduled it this way. That comes back to those `getInputData()` calls from our `doWork()` method.

Often, our work needs data describing that work. In the case of `DownloadWorker`, we need to know:

- the HTTPS URL to download from
- the name of the file to download to (where the file will be placed in `getCacheDir()`)

`WorkManager` has a solution for this, via the `Data` class (perhaps named after [the android character in Star Trek properties](#)) (or perhaps not).

`Data` is a key-value store. The values are simple primitives plus arrays of simple primitives. We can package information into a `Data`, attach it to the work request, and then get that information from inside of `doWork()`.

Reading the `Data` is a matter of calling `getInputData()` inside of `doWork()`, then calling getter methods based on type (e.g., `getString()`). Those getter methods take the key under which the data is stored as a parameter. Getters for primitive types (e.g., `getInt()`, `getBoolean()`) also have a second parameter to use for the default response, if there is nothing associated with that key in the `Data`.

Putting `Data` into a request is a matter of creating a `Data` instance using a `Data.Builder`, which contains the corresponding setter methods (e.g., `putString()`), such as in this Java snippet:

```
OneTimeWorkRequest downloadWork=
 new OneTimeWorkRequest.Builder(DownloadWorker.class)
 .setInputData(new Data.Builder()
 .putString(DownloadWorker.KEY_URL,
 "https://commonsware.com/Android/Android-1_0-CC.pdf")
 .putString(DownloadWorker.KEY_FILENAME, "oldbook.pdf")
 .build())
 .build();

WorkManager.getInstance(getApplicationContext()).enqueue(downloadWork);
```

Here, we fill in the two values that the `DownloadWorker` is expecting, using `setInputData()` to attach the `Data` to our `OneTimeWorkRequest`.

## WORKING WITH WORKMANAGER

Note that there is a 10KB limit on the size of the Data. Data is there mostly to provide identifiers, such as the URL and filename that we are using here. Use the Data for unique information, stuff that the Worker subclass cannot obtain from other sources (e.g., your Room database, your SharedPreferences).

## Constrained Work

Frequently, the work that we want to do has some requirements. For example, in the case of DownloadWorker, it helps to have an Internet connection, as otherwise we may not be able to download the content.

WorkManager exposes a set of constraints. You can constrain your work based on:

- Whether the device has an Internet connection, or perhaps a particular type of Internet connection (e.g., an unmetered connection)
- Whether the device has a decent amount of battery life remaining, or perhaps is on a charger
- Whether the device has a decent amount of storage space available
- Whether the device is idle (so your work is less likely to interfere with the user)

To configure these, we:

- Create a Constraints.Builder,
- Call setter methods on that Builder to specify our constraints,
- build() the Builder, and
- Call setConstraints() on the request Builder to attach the constraints

This Java snippet illustrates what this might look like:

```
Constraints constraints=new Constraints.Builder()
 .setRequiredNetworkType(NetworkType.CONNECTED)
 .setRequiresBatteryNotLow(true)
 .build();

OneTimeWorkRequest downloadWork=
 new OneTimeWorkRequest.Builder(DownloadWorker.class)
 .setConstraints(constraints)
 .setInputData(new Data.Builder()
 .putString(DownloadWorker.KEY_URL,
 "https://commonsware.com/Android/Android-1_0-CC.pdf")
 .putString(DownloadWorker.KEY_FILENAME, "oldbook.pdf")
 .build())
```

## WORKING WITH WORKMANAGER

```
.build();

WorkManager.getInstance(getApplicationContext()).enqueue(downloadWork);
```

Here, we say that we need a network connection (of any type) and that the battery should not be low.

## Tagged Work

We can also associate one or more tags with our work requests. We can later get information about our outstanding work based on tags, or cancel work based on tags.

Tags are meant to be used as categories, to identify similar pieces of work that we might want to operate on in unison:

- All downloads
- All work associated with some particular database table
- All work associated with some account
- And so on

To add tags, just call `addTag()` one or more times on the request Builder:

```
Constraints constraints = new Constraints.Builder()
 .setRequiredNetworkType(NetworkType.CONNECTED)
 .setRequiresBatteryNotLow(true)
 .build();

OneTimeWorkRequest downloadWork =
 new OneTimeWorkRequest.Builder(DownloadWorker.class)
 .setConstraints(constraints)
 .setInputData(new Data.Builder()
 .putString(DownloadWorker.KEY_URL,
 "https://commonsware.com/Android/Android-1_0-CC.pdf")
 .putString(DownloadWorker.KEY_FILENAME, "oldbook.pdf")
 .build())
 .addTag("download")
 .build();

WorkManager.getInstance(getApplicationContext()).enqueue(downloadWork);
```

(from [DownloadWork/src/main/java/com/commonsware/jetpack/work/download/DownloadViewModel.java](#))

```
val constraints = Constraints.Builder()
 .setRequiredNetworkType(NetworkType.CONNECTED)
```

## WORKING WITH WORKMANAGER

```
.setRequiresBatteryNotLow(true)
.build()
val downloadWork = OneTimeWorkRequest.Builder(DownloadWorker::class.java)
.setConstraints(constraints)
.setInputData(
 Data.Builder()
 .putString(
 DownloadWorker.KEY_URL,
 "https://commonsware.com/Android/Android-1_0-CC.pdf"
)
 .putString(DownloadWorker.KEY_FILENAME, "oldbook.pdf")
 .build()
)
.addTag("download")
.build()

WorkManager.getInstance(getApplicationContext()).enqueue(downloadWork)
```

(from [DownloadWork/src/main/java/com/commonsware/jetpack/work/download/DownloadViewModel.kt](#))

Here, we use `addTag()` to tag this work as download.

## Monitoring Work

WorkManager does not provide a built-in means for you to monitor progress inside of an individual piece of work. It does, however, provide you with an API for monitoring the gross state changes of a piece of work: is it enqueued, is it running, is it completed, etc.

### Getting the Status Updates

To find out about the general state changes in the life of a piece of work, you can use `getWorkInfoByIdLiveData()`, available on `WorkManager`. Each request has an ID, generated by the `WorkManager` system, which you get by calling `getId()` on the request:

```
final LiveData<WorkInfo> liveOpStatus =
 WorkManager.getInstance(getApplicationContext()).getWorkInfoByIdLiveData(
 downloadWork.getId());
```

(from [DownloadWork/src/main/java/com/commonsware/jetpack/work/download/DownloadViewModel.java](#))

```
val liveOpStatus =
 WorkManager.getInstance(getApplicationContext())
 .getWorkInfoByIdLiveData(downloadWork.id)
```

## WORKING WITH WORKMANAGER

(from [DownloadWork/src/main/java/com/commonsware/jetpack/work/download/DownloadViewModel.kt](#))

The LiveData that we get back will emit WorkInfo updates for the work identified by this ID. A WorkInfo, in turn, holds a State enum, that indicates what phase of the WorkManager process this piece of work is in:

- ENQUEUED
- BLOCKED (for use with [chained work](#))
- RUNNING
- SUCCEEDED
- FAILED
- CANCELED (for use with [canceling work](#))

You can then arrange to observe the LiveData or otherwise make use of its updates.

## Consuming the Status Updates... In Code

The code shown in this chapter so far that created the OneTimeWorkRequest and enqueued the work is in a DownloadViewModel:

```
package com.commonsware.jetpack.work.download;

import android.app.Application;
import androidx.annotation.NonNull;
import androidx.lifecycle.AndroidViewModel;
import androidx.lifecycle.LiveData;
import androidx.lifecycle.MediatorLiveData;
import androidx.work.Constraints;
import androidx.work.Data;
import androidx.work.NetworkType;
import androidx.work.OneTimeWorkRequest;
import androidx.work.WorkInfo;
import androidx.work.WorkManager;

public class DownloadViewModel extends AndroidViewModel {
 public final MediatorLiveData<WorkInfo> liveWorkStatus =
 new MediatorLiveData<>();

 public DownloadViewModel(@NonNull Application application) {
 super(application);
 }

 public void doTheDownload() {
 Constraints constraints = new Constraints.Builder()
 .setRequiredNetworkType(NetworkType.CONNECTED)
```

## WORKING WITH WORKMANAGER

```
.setRequiresBatteryNotLow(true)
.build();

OneTimeWorkRequest downloadWork =
new OneTimeWorkRequest.Builder(DownloadWorker.class)
.setConstraints(constraints)
.setInputData(new Data.Builder()
.putString(DownloadWorker.KEY_URL,
"https://commonsware.com/Android/Android-1_0-CC.pdf")
.putString(DownloadWorker.KEY_FILENAME, "oldbook.pdf")
.build())
.addTag("download")
.build();

WorkManager.getInstance(getApplicationContext()).enqueue(downloadWork);

final LiveData<WorkInfo> liveOpStatus =
WorkManager.getInstance(getApplicationContext()).getWorkInfoByIdLiveData(
downloadWork.getId());

liveWorkStatus.addSource(liveOpStatus, workStatus -> {
 liveWorkStatus.setValue(workStatus);

 if (workStatus.getState().isFinished()) {
 liveWorkStatus.removeSource(liveOpStatus);
 }
});
}
}
```

(from [DownloadWork/src/main/java/com/commonsware/jetpack/work/download/DownloadViewModel.java](#))

```
package com.commonsware.jetpack.work.download

import android.app.Application
import androidx.lifecycle.AndroidViewModel
import androidx.lifecycle.MediatorLiveData
import androidx.work.*

class DownloadViewModel(application: Application) :
 AndroidViewModel(application) {
 val liveWorkStatus = MediatorLiveData<WorkInfo>()

 fun doTheDownload() {
 val constraints = Constraints.Builder()
 .setRequiredNetworkType(NetworkType.CONNECTED)
 .setRequiresBatteryNotLow(true)
 .build()
 val downloadWork = OneTimeWorkRequest.Builder(DownloadWorker::class.java)
```

## WORKING WITH WORKMANAGER

```
.setConstraints(constraints)
.setInputData(
 Data.Builder()
 .putString(
 DownloadWorker.KEY_URL,
 "https://commonsware.com/Android/Android-1_0-CC.pdf"
)
 .putString(DownloadWorker.KEY_FILENAME, "oldbook.pdf")
 .build()
)
.addTag("download")
.build()

WorkManager.getInstance(getApplicationContext()).enqueue(downloadWork)

val liveOpStatus =
 WorkManager.getInstance(getApplicationContext())
 .getWorkInfoByIdLiveData(downloadWork.id)

liveWorkStatus.addSource(liveOpStatus) { workStatus ->
 liveWorkStatus.value = workStatus

 if (workStatus.state.isFinished) {
 liveWorkStatus.removeSource(liveOpStatus)
 }
}
}
```

(from [DownloadWork/src/main/java/com/commonsware/jetpack/work/download/DownloadViewModel.kt](#))

The `doTheDownload()` method will be called when the user clicks a button in the UI of `MainActivity`. That triggers our creation of the work request.

`DownloadViewModel` takes the `MediatorLiveData` approach seen elsewhere in the book. Consumers of the `DownloadViewModel`, such as our `MainActivity`, have access to a `liveWorkStatus` field that represents the outbound stream of work status updates. For each `doTheDownload()` call, we chain the `LiveData` for this individual download onto the `MediatorLiveData`, removing it as a source once the State reaches a terminal condition (`isFinished()`, which will be true for a State of `SUCCEEDED`, `FAILED`, or `CANCELED`).

The result is that our `MainActivity` can observe `liveWorkStatus`, without having to worry about individual `LiveData` objects from individual download requests.

`MainActivity` observes `liveWorkStatus` and uses it to display a `Toast` when the

## WORKING WITH WORKMANAGER

download is finished:

```
package com.commonsware.jetpack.work.download;

import android.os.Bundle;
import android.view.View;
import android.widget.Toast;
import com.commonsware.jetpack.work.download.databinding.ActivityMainBinding;
import androidx.appcompat.app.AppCompatActivity;
import androidx.databinding.BindingAdapter;
import androidx.lifecycle.ViewModelProviders;
import androidx.work.WorkInfo;

public class MainActivity extends AppCompatActivity {
 @BindingAdapter("android:enabled")
 public static void setEnabled(View v, WorkInfo info) {
 if (info==null) {
 v.setEnabled(true);
 } else {
 v.setEnabled(info.getState().isFinished());
 }
 }

 private ActivityMainBinding binding;

 @Override
 protected void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);

 final DownloadViewModel vm= ViewModelProviders.of(this).get(DownloadViewModel.class);

 binding=ActivityMainBinding.inflate(getApplicationContext());
 binding.setViewModel(vm);
 binding.setLifecycleOwner(this);

 setContentView(binding.getRoot());

 vm.liveWorkStatus.observe(this, workStatus -> {
 if (workStatus!=null && workStatus.getState().isFinished()) {
 Toast.makeText(this, R.string.msg_done, Toast.LENGTH_LONG).show();
 }
 });
 }
}
```

(from [DownloadWork/src/main/java/com/commonsware/jetpack/work/download/MainActivity.java](#))

```
package com.commonsware.jetpack.work.download

import android.os.Bundle
import android.view.View
import android.widget.Toast
import androidx.appcompat.app.AppCompatActivity
import androidx.databinding.BindingAdapter
import androidx.lifecycle.Observer
```

## WORKING WITH WORKMANAGER

```
import androidx.lifecycle.ViewModelProviders
import androidx.work.WorkInfo
import com.commonsware.jetpack.work.download.databinding.ActivityMainBinding

@BindingAdapter("android:enabled")
fun View.setEnabled(info: WorkInfo?) {
 isEnabled = info?.state?.isFinished ?: true
}

class MainActivity : AppCompatActivity() {

 override fun onCreate(savedInstanceState: Bundle?) {
 super.onCreate(savedInstanceState)

 val vm = ViewModelProviders.of(this)[DownloadViewModel::class.java]
 val binding = ActivityMainBinding.inflate(layoutInflater)

 binding.viewModel = vm
 binding.lifecycleOwner = this

 setContentView(binding.root)

 vm.liveWorkStatus.observe(this, Observer { workStatus ->
 if (workStatus != null && workStatus.state.isFinished) {
 Toast.makeText(this, R.string.msg_done, Toast.LENGTH_LONG).show()
 }
 })
 }
}
```

(from [DownloadWork/src/main/java/com/commonsware/jetpack/work/download/MainActivity.kt](#))

## Consuming the Status Updates... In Data Binding

MainActivity — and its activity\_main layout resource — use data binding. Partially, this is to get control to DownloadViewModel when the user clicks a button. But we also want to disable the button while the download is going on, to reduce the likelihood of accidentally triggering multiple downloads.

Data binding has dedicated support for LiveData. If you have a LiveData available through a <variable>, you can reference the LiveData in a binding expression as if it were a simple variable representing the data. The data binding framework will take care of the details of observing the LiveData and updating your UI when the data changes.

## WORKING WITH WORKMANAGER

To make this work, our layout has a reference to the DownloadViewModel and has a binding expression on android:enabled that looks at the state:

```
<?xml version="1.0" encoding="utf-8"?>
<layout>

 <data>

 <variable
 name="viewModel"
 type="com.commonware.jetpack.work.download.DownloadViewModel" />
 </data>

 <androidx.constraintlayout.widget.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">

 <Button
 android:id="@+id/download"
 android:layout_width="0dp"
 android:layout_height="0dp"
 android:text="@string/btn_title"
 android:onClick="@{() -> viewModel.doTheDownload()}"
 android:enabled="@{viewModel.liveWorkStatus }"
 app:layout_constraintBottom_toBottomOf="parent"
 app:layout_constraintEnd_toEndOf="parent"
 app:layout_constraintStart_toStartOf="parent"
 app:layout_constraintTop_toTopOf="parent" />
 </androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```

(from [DownloadWork/src/main/res/layout/activity\\_main.xml](#))

Our MainActivity then does three things in support of all of this:

1. It has a BindingAdapter that can update the enabled status of a view given a WorkInfo, allowing us to use a binding expression on android:enabled
2. It binds the DownloadViewModel into the binding
3. It calls setLifecycleOwner() on the binding, which the data binding framework will use for observing the LiveData:

```
package com.commonware.jetpack.work.download;
```

## WORKING WITH WORKMANAGER

```
import android.os.Bundle;
import android.view.View;
import android.widget.Toast;
import com.commonsware.jetpack.work.download.databinding.ActivityMainBinding;
import androidx.appcompat.app.AppCompatActivity;
import androidx.databinding.BindingAdapter;
import androidx.lifecycle.ViewModelProviders;
import androidx.work.WorkInfo;

public class MainActivity extends AppCompatActivity {
 @BindingAdapter("android:enabled")
 public static void setEnabled(View v, WorkInfo info) {
 if (info==null) {
 v.setEnabled(true);
 } else {
 v.setEnabled(info.getState().isFinished());
 }
 }

 private ActivityMainBinding binding;

 @Override
 protected void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);

 final DownloadViewModel vm= ViewModelProviders.of(this).get(DownloadViewModel.class);

 binding=ActivityMainBinding.inflate(getApplicationContext());
 binding.setViewModel(vm);
 binding.setLifecycleOwner(this);

 setContentView(binding.getRoot());

 vm.liveWorkStatus.observe(this, workStatus -> {
 if (workStatus!=null && workStatus.getState().isFinished()) {
 Toast.makeText(this, R.string.msg_done, Toast.LENGTH_LONG).show();
 }
 });
 }
}
```

(from [DownloadWork/src/main/java/com/commonsware/jetpack/work/download/MainActivity.java](#))

```
package com.commonsware.jetpack.work.download

import android.os.Bundle
import android.view.View
import android.widget.Toast
import androidx.appcompat.app.AppCompatActivity
import androidx.databinding.BindingAdapter
import androidx.lifecycle.Observer
import androidx.lifecycle.ViewModelProviders
import androidx.work.WorkInfo
import com.commonsware.jetpack.work.download.databinding.ActivityMainBinding
```

## WORKING WITH WORKMANAGER

```
@BindingAdapter("android:enabled")
fun View.setEnabled(info: WorkInfo?) {
 isEnabled = info?.state?.isFinished ?: true
}

class MainActivity : AppCompatActivity() {

 override fun onCreate(savedInstanceState: Bundle?) {
 super.onCreate(savedInstanceState)

 val vm = ViewModelProviders.of(this)[DownloadViewModel::class.java]
 val binding = ActivityMainBinding.inflate(layoutInflater)

 binding.viewModel = vm
 binding.lifecycleOwner = this

 setContentView(binding.root)

 vm.liveWorkStatus.observe(this, Observer { workStatus ->
 if (workStatus != null && workStatus.state.isFinished) {
 Toast.makeText(this, R.string.msg_done, Toast.LENGTH_LONG).show()
 }
 })
 }
}
```

(from [DownloadWork/src/main/java/com/commonsware/jetpack/work/download/MainActivity.kt](#))

Here, we map the State from a WorkInfo to the boolean value to use for the android:enabled attribute. Basically, if the WorkInfo is null or is finished, the button is enabled, otherwise it is disabled. So, as the LiveData emits new WorkInfo objects, data binding takes each, calls this setEnabled() method, and uses that to update the enabled state of the Button.

## Cancelling Work

Frequently, the work that we enqueue into the WorkManager is “fire and forget”, where either the work succeeds or fails on its own. Occasionally, though, we may need to try to cancel a piece of enqueued work. For example, we might offer a cancel button in the UI to allow the user to abandon some enqueued request (e.g., do not download the thing that the user just requested to download).

For that, you can call cancelWorkById() or cancelAllWorkByTag() on WorkManager. The former takes the work’s ID (from getId() on the WorkRequest), while the latter

## WORKING WITH WORKMANAGER

---

takes a tag. Since IDs are unique, `cancelWorkById()` will only try to cancel that once piece of work, while `cancelAllWorkByTag()` will try to cancel all enqueued work associated with that tag. So, for example, if you associate the download tag with your download work requests, `cancelAllWorkByTag("download")` will try to cancel all of those requests.

Note, though, that cancellation is “best effort”. In particular, if the work has already begun, it might not be canceled. In some cases, this is fine. In other cases, you might want to both cancel the work in `WorkManager` and take steps to ensure that any affected running work finds out about the cancellation. For example, you might have some state field somewhere that the work can monitor to see if it should continue doing whatever it is doing.

## Delayed Work

Typically, we are happy to have our work begin right away, if the conditions allow it. Occasionally, we may want to intentionally delay that work for a bit.

For this, you can call `setInitialDelay()` on the `OneTimeWorkRequest.Builder` as part of configuring the work. There are two flavors of `setInitialDelay()`:

- one takes a `long` value and a `TimeUnit`, where the `TimeUnit` indicates what unit of measure the `long` is in (e.g., `TimeUnit.SECONDS`)
- one takes a `Duration`, which is part of Java 8 and only available on API Level 26 and higher

In either case, the work will be delayed by at least that amount of time. However, depending on circumstances (constraints, Doze mode, etc.), the work might happen substantially later than the delay period. Do not assume that your work will start immediately after your delay period.

## Parallel Work

Suppose you have N pieces of work to be done in (approximate) parallel. As it turns out, `enqueue()` on `WorkManager` takes either a varargs of `WorkRequest` or a `List` of `WorkRequest` objects. If you pass in more than one `WorkRequest`, all will be enqueued, and all will run when possible, based upon their constraints and the number of threads in the `WorkManager` thread pool.

`WorkManager` has a default thread pool, and in many cases it will be sufficient for

## WORKING WITH WORKMANAGER

---

your needs. If you wish to control that thread pool, call the static `initialize()` on `WorkManager` *once*, such as from `onCreate()` of a custom Application.

`initialize()` takes a Context and a Configuration. You get a Configuration through the builder pattern:

- Create an instance of `Configuration.Builder`
- Call `setExecutor()` on that Builder with an Executor that will serve as the thread pool for the `WorkManager`
- Call `build()` on the Builder to get the Configuration

## Chained Work

Where `WorkManager` shines is in its support for chained work. Chained work is where you set up work requests that in turn depend upon other work requests. Later work requests in the chain are only performed if the previous ones succeeded. And, work requests can supply data to the next request in the chain, akin to command-line pipelines or basic workflow systems.

### Why?

On the one hand, chained work may not seem necessary. In principle, what you do as a series of work requests could be done in one large work request.

The big benefit of splitting the work into separate requests comes with the application of constraints. For example, the sample app that we will examine demonstrates chained work by downloading a ZIP file, then unZIPping it. Downloading a ZIP file requires an Internet connection, but unZIPping it does not. By providing separate constraints for each work request, you can require a network connection for the download, yet not require it for the unZIP task, thereby allowing that work to proceed even if Internet connectivity is lost.

Also, smaller Worker classes can be made more reusable. One can imagine a library of common Worker classes. Rather than having to write your own `CompositeWorker` that used several Worker classes, you can simply set up a chain using existing APIs.

Chained work also helps to address the delivery of status updates as a larger task is being processed. Each `WorkRequest` in the chain has its own `WorkStatus` that can be monitored via `LiveData`. This way, you can at least get coarse-grained information about how the chain overall is proceeding.

### How Do We Chain Work?

To enqueue a `WorkRequest`, we used `enqueue()` on the `WorkManager` instance. In truth, that is a convenience method. This:

```
WorkManager.getInstance(getApplicationContext())
 .enqueue(request);
```

is really this:

```
WorkManager.getInstance(getApplicationContext())
 .beginWith(request)
 .enqueue();
```

`beginWith()` returns a `WorkContinuation`. This is an object that knows a `WorkRequest` to process and knows how to be chained.

To have a follow-on `WorkRequest` in a simple two-element chain, call `then()` on the `WorkContinuation` before the terminal `enqueue()` call:

```
WorkManager.getInstance(getApplicationContext())
 .beginWith(request)
 .then(otherRequest)
 .enqueue();
```

Now, `request` will be processed, and if it succeeds, then (and only then) will `otherRequest` be processed.

### How Do We Pass Data Along the Chain?

We provide input to a `WorkRequest` via its `Builder` and `setInputData()`. However, this is input that is created outside the processing of any individual request; it is input that is defined when the chain is defined.

In addition, a `Worker` can provide *output* data to factory methods like `success()` on `ListenableWorker.Result`. Those factory methods take the same sort of `Data` object that `setInputData()` does. The output data can be used in two places:

- If this request has another request chained after it, that later request receives the earlier request's output data as input.
- The output data is available from the `WorkInfo` once the work is finished, so consumers of the `LiveData` status stream can also see the output data.

### OK, Where's the Code?

The UnZIPWork sample module in the [Sampler](#) and [SamplerJ](#) projects are a variation on the previous example, this time where we have two requests in a chain.

DownloadWorker is largely the same as before, with two differences:

1. Rather than receiving a filename as input, it decides what the filename will be, as that will merely serve as a temporary file
2. It passes the path to that file to the next request in the chain via `setOutputData()`

```
package com.commonsware.jetpack.work.download;

import android.content.Context;
import android.util.Log;
import java.io.File;
import java.io.IOException;
import androidx.annotation.NonNull;
import androidx.work.Data;
import androidx.work.ListenableWorker;
import androidx.work.Worker;
import androidx.work.WorkerParameters;
import okhttp3.OkHttpClient;
import okhttp3.Request;
import okhttp3.Response;
import okio.BufferedSink;
import okio.Okio;

public class DownloadWorker extends Worker {
 public static final String KEY_URL="url";
 public static final String KEY_RESULTDIR="resultDir";

 public DownloadWorker(@NonNull Context context,
 @NonNull WorkerParameters workerParams) {
 super(context, workerParams);
 }

 @NonNull
 @Override
 public Result doWork() {
 OkHttpClient client=new OkHttpClient();
 Request request=new Request.Builder()
 .url(getInputData().getString(KEY_URL))
 .build();
```

## WORKING WITH WORKMANAGER

```
File dir=getApplicationContext().getCacheDir();
File downloadedFile=new File(dir, "temp.zip");

if (downloadedFile.exists()) {
 downloadedFile.delete();
}

try (Response response=client.newCall(request).execute()) {
 BufferedSink sink=Okio.buffer(Okio.sink(downloadedFile));

 sink.writeAll(response.body().source());
 sink.close();
}
catch (IOException e) {
 Log.e(getClass().getSimpleName(), "Exception downloading file", e);

 return ListenableWorker.Result.failure();
}

return ListenableWorker.Result.success(new Data.Builder()
 .putString(UnZIPWorker.KEY_ZIPFILE, downloadedFile.getAbsolutePath())
 .build());
}
```

(from [UnZIPWork/src/main/java/com/commonsware/jetpack/work/download/DownloadWorker.java](#))

```
package com.commonsware.jetpack.work.download

import android.content.Context
import android.util.Log
import androidx.work.Data
import androidx.work.Worker
import androidx.work.WorkerParameters
import okhttp3.OkHttpClient
import okhttp3.Request
import okio.buffer
import okio.sink
import java.io.File
import java.io.IOException

class DownloadWorker(context: Context, workerParams: WorkerParameters) :
 Worker(context, workerParams) {

 override fun doWork(): Result {
 val client = OkHttpClient()
 val request = Request.Builder()
 .url(inputData.getString(KEY_URL)!!)
 }
}
```

## WORKING WITH WORKMANAGER

```
.build()

val dir = applicationContext.cacheDir
val downloadedFile = File(dir, "temp.zip")

if (downloadedFile.exists()) {
 downloadedFile.delete()
}

try {
 client.newCall(request).execute().use { response ->
 val sink = downloadedFile.sink().buffer()

 response.body?.let { sink.writeAll(it.source()) }
 sink.close()
 }
} catch (e: IOException) {
 Log.e(javaClass.simpleName, "Exception downloading file", e)

 return Result.failure()
}

return Result.success(
 Data.Builder()
 .putString(UnZIPWorker.KEY_ZIPFILE, downloadedFile.absolutePath)
 .build()
)
}

companion object {
 const val KEY_URL = "url"
 const val KEY_RESULTDIR = "resultDir"
}
}
```

(from [UnZIPWork/src/main/java/com/commonsware/jetpack/work/download/DownloadWorker.kt](#))

We now also have an UnZIPWorker. This expects two pieces of input: the file to unZIP and the directory to unZIP it into. It uses [the CWAC-Security library](#) and its ZipUtils.unzip() method, as that safely handles possibly-malicious ZIP files (e.g., zip bombs):

```
package com.commonsware.jetpack.work.download;

import android.content.Context;
import android.util.Log;
import com.commonsware.cwac.security.ZipUtils;
import java.io.File;
import androidx.annotation.NonNull;
```

## WORKING WITH WORKMANAGER

```
import androidx.work.ListenableWorker;
import androidx.work.Worker;
import androidx.work.WorkerParameters;

public class UnZIPWorker extends Worker {
 public static final String KEY_ZIPFILE="zipFile";
 public static final String KEY_RESULTDIR="resultDir";

 public UnZIPWorker(@NonNull Context context,
 @NonNull WorkerParameters workerParams) {
 super(context, workerParams);
 }

 @NonNull
 @Override
 public Result doWork() {
 File downloadedFile=new File(getInputData().getString(KEY_ZIPFILE));
 File dir=getApplicationContext().getCacheDir();
 String resultDirData=getInputData().getString(KEY_RESULTDIR);
 File resultDir=new File(dir, resultDirData==null ? "results" : resultDirData);

 try {
 ZipUtils.unzip(downloadedFile, resultDir, 2048, 1024*1024*16);
 downloadedFile.delete();
 }
 catch (Exception e) {
 Log.e(getClass().getSimpleName(), "Exception unZIPing file", e);
 }

 return ListenableWorker.Result.failure();
 }

 return ListenableWorker.Result.success();
}
}
```

(from [UnZIPWork/src/main/java/com/commonsware/jetpack/work/download/UnZIPWorker.java](#))

```
package com.commonsware.jetpack.work.download

import android.content.Context
import android.util.Log
import androidx.work.ListenableWorker
import androidx.work.Worker
import androidx.work.WorkerParameters
import com.commonsware.cwac.security.ZipUtils
import java.io.File

class UnZIPWorker(context: Context, workerParams: WorkerParameters) :
 Worker(context, workerParams) {

 override fun doWork(): Result {
 val downloadedFile = File(inputData.getString(KEY_ZIPFILE)!!)
 val dir = applicationContext.cacheDir
 val resultDirData = inputData.getString(KEY_RESULTDIR)
 val resultDir = File(dir, resultDirData ?: "results")
 }
}
```

## WORKING WITH WORKMANAGER

```
try {
 ZipUtils.unzip(downloadedFile, resultDir, 2048, 1024 * 1024 * 16)
 downloadedFile.delete()
} catch (e: Exception) {
 Log.e(javaClass.simpleName, "Exception unZIPing file", e)

 return Result.failure()
}

return Result.success()
}

companion object {
 const val KEY_ZIPFILE = "zipFile"
 const val KEY_RESULTDIR = "resultDir"
}
}
```

(from [UnZIPWork/src/main/java/com/commonsware/jetpack/work/download/UnZIPWorker.kt](#))

DownloadViewModel now sets up a request chain using both worker classes:

```
package com.commonsware.jetpack.work.download;

import android.app.Application;
import androidx.annotation.NonNull;
import androidx.lifecycle.AndroidViewModel;
import androidx.lifecycle.LiveData;
import androidx.lifecycle.MediatorLiveData;
import androidx.work.Constraints;
import androidx.work.Data;
import androidx.work.NetworkType;
import androidx.work.OneTimeWorkRequest;
import androidx.work.WorkInfo;
import androidx.work.WorkManager;

public class DownloadViewModel extends AndroidViewModel {
 public final MediatorLiveData<WorkInfo> liveWorkStatus=new MediatorLiveData<>();

 public DownloadViewModel(@NonNull Application application) {
 super(application);
 }

 public void doTheDownload() {
 OneTimeWorkRequest downloadWork=
 new OneTimeWorkRequest.Builder(DownloadWorker.class)
 .setConstraints(new Constraints.Builder()
 .setRequiredNetworkType(NetworkType.CONNECTED)
 .setRequiresBatteryNotLow(true)
 .build())
 .setInputData(new Data.Builder()
 .putString(DownloadWorker.KEY_URL,
 "https://commonsware.com/Android/source_1_0.zip")

```

## WORKING WITH WORKMANAGER

---

```
.build())
.addTag("download")
.build();
OneTimeWorkRequest unZIPWork=
new OneTimeWorkRequest.Builder(UnZIPWorker.class)
.setConstraints(new Constraints.Builder()
.setRequiresStorageNotLow(true)
.setRequiresBatteryNotLow(true)
.build())
.setInputData(new Data.Builder()
.putString(DownloadWorker.KEY_RESULTDIR, "unzipped")
.build())
.addTag("unZIP")
.build();

WorkManager.getInstance(getApplicationContext())
.beginWith(downloadWork)
.then(unZIPWork)
.enqueue();

final LiveData<WorkInfo> liveOpStatus=
WorkManager.getInstance(getApplicationContext()).getWorkInfoByIdLiveData(unZIPWork.getId());

liveWorkStatus.addSource(liveOpStatus, workStatus -> {
 liveWorkStatus.setValue(workStatus);

 if (workStatus.getState().isFinished()) {
 liveWorkStatus.removeSource(liveOpStatus);
 }
});
}
}
```

(from [UnZIPWork/src/main/java/com/commonsware/jetpack/work/download/DownloadViewModel.java](#))

```
package com.commonsware.jetpack.work.download

import android.app.Application
import androidx.lifecycle.AndroidViewModel
import androidx.lifecycle.LiveData
import androidx.lifecycle.MediatorLiveData
import androidx.work.Constraints
import androidx.work.Data
import androidx.work.NetworkType
import androidx.work.OneTimeWorkRequest
import androidx.work.WorkInfo
import androidx.work.WorkManager

class DownloadViewModel(application: Application) :
 AndroidViewModel(application) {
 val liveWorkStatus = MediatorLiveData<WorkInfo>()

 fun doTheDownload() {
 val downloadWork = OneTimeWorkRequest.Builder(DownloadWorker::class.java)
 .setConstraints(
```

## WORKING WITH WORKMANAGER

---

```
Constraints.Builder()
 .setRequiredNetworkType(NetworkType.CONNECTED)
 .setRequiresBatteryNotLow(true)
 .build()
)
.setInputData(
 Data.Builder()
 .putString(
 DownloadWorker.KEY_URL,
 "https://commonsware.com/Android/source_1_0.zip"
)
 .build()
)
.addTag("download")
.build()
val unZIPWork = OneTimeWorkRequest.Builder(UnZIPWorker::class.java)
.setConstraints(
 Constraints.Builder()
 .setRequiresStorageNotLow(true)
 .setRequiresBatteryNotLow(true)
 .build()
)
.setInputData(
 Data.Builder()
 .putString(DownloadWorker.KEY_RESULTDIR, "unzipped")
 .build()
)
.addTag("unZIP")
.build()

WorkManager.getInstance(getApplicationContext())
.beginWith(downloadWork)
.then(unZIPWork)
.enqueue()

val liveOpStatus = WorkManager.getInstance(getApplicationContext())
.getWorkInfoByIdLiveData(unZIPWork.id)

liveWorkStatus.addSource(liveOpStatus) { workStatus ->
 liveWorkStatus.value = workStatus

 if (workStatus.state.isFinished) {
 liveWorkStatus.removeSource(liveOpStatus)
 }
}
}
```

## WORKING WITH WORKMANAGER

(from [UnZIPWork/src/main/java/com/commonsware/jetpack/work/download/DownloadViewModel.kt](#))

Of note:

- downloadWork is defined the same as before, except that we skip supplying the filename, and the URL now points to a ZIP file instead of a PDF
- unZIPWork does not require an Internet connection, but it does require that we have a reasonable amount of storage available
- unZIPWork gets the name of a directory to create in getCacheDir() to hold the unZIPped results
- We use beginWith() and then() to set up the chain, using enqueue() to enqueue the results
- We monitor the unZIPWork status for the purposes of re-enabling the button and showing the Toast

In principle, we should be monitoring *both* requests' status updates. If the first request fails for some reason (e.g., HTTP 404 error), the second request will never run. We could do that by calling getWorkInfosLiveData() on the WorkContinuation, which returns a LiveData of a *list* of WorkInfo objects, one for each request in the chain. That significantly increases the complexity of the sample (e.g., what do we do for data binding in this case?), and so we cheat for the sake of brevity.

## How Complex Can This Get?

It can get as complicated as you like:

- You can keep chaining work together by successive then() calls:

```
WorkManager.getInstance(getApplicationContext())
 .beginWith(lets)
 .then(go)
 .then(crazy)
 .enqueue();
```

- You can have parallel requests as part of a chain, by passing multiple WorkRequest objects to beginWith() or then()
- You can chain a WorkContinuation onto another WorkContinuation
- You can create InputMerger implementations to help coordinate out the output data from previous steps in the chain are merged together to form the input data for successive steps in the chain
- And so on

## WORKING WITH WORKMANAGER

---

However, while WorkManager is useful for deferrable tasks, it is not a full workflow system:

- There is limited ability to cancel work, as noted previously
- There is no ability to change enqueued work, except by trying to cancel it and then enqueueing its replacement
- There are no specifications for how long any individual request or an entire chain can take, in terms of time
- There are no specifications for how results are handled when it takes multiple process invocations to complete a chain (e.g., a long chain extending past the 10-minute limit)
- And so on

As a result, at least for the time being, be careful when trying to create complex WorkRequest chains.

## Periodic Work

So far, all of the work has been for single tasks, using OneTimeWorkRequest. The other WorkRequest implementation is PeriodicWorkRequest, and as the name suggests, it is for work that should repeat with a given interval.

The interval is provided via the PeriodicWorkRequest.Builder constructor, either as a Duration or as a long and TimeUnit pair. There is a minimum allowed period, defined as PeriodicWorkRequest.MIN\_PERIODIC\_INTERVAL\_MILLIS, presently defined as 15 minutes. Consider the supplied interval to be a suggestion, rather than a requirement.

## Unique Work

Sometimes, we want to avoid accidentally enqueueing the same work more than once.

For example, suppose that the app has a Worker that pulls data from a server. Normally, that is triggered by a push message (e.g., Firebase Cloud Messaging). However, you also have it set up that user actions can trigger that work to be done, such as via a manual refresh option (e.g., pull-to-refresh). What you want to avoid is trying to do two of this bit of work simultaneously, as that might confuse things on either the device side or the server side.

## WORKING WITH WORKMANAGER

---

To handle this, instead of `enqueue()` on `WorkManager`, you can use `beginUniqueWork()`. This takes a “name” that identifies this logical unit of work. If you previously used `beginUniqueWork()`, and you later call `beginUniqueWork()` with the same name, and the earlier work is still ongoing, you can specify what should happen (e.g., ignore the new work request).

Note that this does not support periodic work: you can only coordinate `OneTimeWorkRequest`, not `PeriodicWorkRequest`.

## Testing Work

The work-testing artifact offers a `WorkManagerTestInitHelper` utility class to help with instrumented testing.

First, it has `initializeTestWorkManager()`. This configures `WorkManager` to use a `SynchronousExecutor`. This amounts to a mock Executor, one that runs supplied `Runnable` objects immediately on the current thread. By using `SynchronousExecutor`, your `enqueue()` calls for `WorkManager` will happen immediately and synchronously, rather than asynchronously.

Also, `WorkManagerTestInitHelper` has a `getTestDriver()` method, which returns a `TestDriver`. This offers a `setAllConstraintsMet()` method, which takes a work request ID and tells `WorkManager` that all of the constraints for that work request are met. This makes your tests more deterministic, since constraints are normally there to test the environment, and that might change from run to run of your tests. However, it is very important to call `setAllConstraintsMet()` **after** you `enqueue()` the work:

```
WorkManager.getInstance(context).enqueue(work);
WorkManagerTestInitHelper.getTestDriver(context).setAllConstraintsMet(work.getId());
```

Note: calling `setAllConstraintsMet()` before calling `enqueue()` results in a crash.

The `Work/Download` sample app contains a `DownloadWorkerTest` class that shows the use of `WorkManagerTestInitHelper` and `TestDriver`:

```
package com.commonsware.jetpack.work.download;

import android.content.Context;
import com.commonsware.jetpack.work.download.DownloadWorker;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
```

## WORKING WITH WORKMANAGER

---

```
import java.io.File;
import androidx.test.platform.app.InstrumentationRegistry;
import androidx.test.ext.junit.runners.AndroidJUnit4;
import androidx.work.Constraints;
import androidx.work.Data;
import androidx.work.NetworkType;
import androidx.work.OneTimeWorkRequest;
import androidx.work.WorkManager;
import androidx.work.WorkRequest;
import androidx.work.testing.WorkManagerTestInitHelper;
import static org.junit.Assert.assertFalse;
import static org.junit.Assert.assertTrue;

@RunWith(AndroidJUnit4.class)
public class DownloadWorkerTest {
 private File expected;
 private Context context = InstrumentationRegistry.getInstrumentation().getTargetContext();

 @Before
 public void setUp() {
 WorkManagerTestInitHelper.initializeTestWorkManager(context);

 expected = new File(context.getCacheDir(),"oldbook.pdf");

 if (expected.exists()) {
 expected.delete();
 }
 }

 @Test
 public void download() {
 assertFalse(expected.exists());

 WorkManager.getInstance(context).enqueue(buildWorkRequest(null));

 assertTrue(expected.exists());
 }

 @Test
 public void downloadWithConstraints() {
 Constraints constraints=new Constraints.Builder()
 .setRequiredNetworkType(NetworkType.CONNECTED)
 .setRequiresBatteryNotLow(true)
 .build();
 WorkRequest work=buildWorkRequest(constraints);

 assertFalse(expected.exists());

 WorkManager.getInstance(context).enqueue(work);
 WorkManagerTestInitHelper.getTestDriver(context).setAllConstraintsMet(work.getId());

 assertTrue(expected.exists());
 }

 private WorkRequest buildWorkRequest(Constraints constraints) {
 OneTimeWorkRequest.Builder builder=
 new OneTimeWorkRequest.Builder(DownloadWorker.class)
 . setInputData(new Data.Builder()
 .putString(DownloadWorker.KEY_URL,
 "https://commonsware.com/Android/Android-1_0-CC.pdf")

```

## WORKING WITH WORKMANAGER

---

```
.putString(DownloadWorker.KEY_FILENAME, "oldbook.pdf")
.build()
.addTag("download");

if (constraints!=null) {
 builder.setConstraints(constraints);
}

return builder.build();
}
```

(from [DownloadWork/src/androidTest/java/com/commonsware/jetpack/work/download/DownloadWorkerTest.java](#))

```
package com.commonsware.jetpack.work.download

import androidx.test.ext.junit.runners.AndroidJUnit4
import androidx.test.platform.app.InstrumentationRegistry
import androidx.work.*
import androidx.work.testing.WorkManagerTestInitHelper
import org.junit.Assert.assertFalse
import org.junit.Assert.assertTrue
import org.junit.Before
import org.junit.Test
import org.junit.runner.RunWith
import java.io.File

@RunWith(AndroidJUnit4::class)
class DownloadWorkerTest {
 private lateinit var expected: File
 private val context =
 InstrumentationRegistry.getInstrumentation().targetContext

 @Before
 fun setUp() {
 WorkManagerTestInitHelper.initializeTestWorkManager(context)

 expected = File(context.cacheDir, "oldbook.pdf")

 if (expected.exists()) {
 expected.delete()
 }
 }

 @Test
 fun download() {
 assertFalse(expected.exists())

 WorkManager.getInstance(context).enqueue(buildWorkRequest(null))

 assertTrue(expected.exists())
 }

 @Test
 fun downloadWithConstraints() {
 val constraints = Constraints.Builder()
 .setRequiredNetworkType(NetworkType.CONNECTED)
 .setRequiresBatteryNotLow(true)
 .build()
 val work = buildWorkRequest(constraints)
 }
}
```

## WORKING WITH WORKMANAGER

```
assertFalse(expected.exists())

WorkManager.getInstance(context).enqueue(work)
WorkManagerTestInitHelper.getTestDriver(context)!!.setAllConstraintsMet(work.id)

assertTrue(expected.exists())
}

private fun buildWorkRequest(constraints: Constraints?): WorkRequest {
 val builder = OneTimeWorkRequest.Builder(DownloadWorker::class.java)
 .setInputData(
 Data.Builder()
 .putString(
 DownloadWorker.KEY_URL,
 "https://commonsware.com/Android/Android-1_0-CC.pdf"
)
 .putString(DownloadWorker.KEY_FILENAME, "oldbook.pdf")
 .build()
)
 .addTag("download")

 if (constraints != null) {
 builder.setConstraints(constraints)
 }

 return builder.build()
}
}
```

(from [DownloadWork/src/androidTest/java/com/commonsware/jetpack/work/download/DownloadWorkerTest.kt](#))

This class tests DownloadWorker both with and without constraints, validating that the output file exists after the work has been done. Since we are using the synchronous test configuration of WorkManager, we can test this work without having to resort to CountDownLatch or similar tricks for testing multithreaded code.

Independent of work-testing, note that Worker has some dependencies on Context, and it may be difficult to mock that Context since you are not the one providing it. It may be necessary to consider your Worker as something to be tested with instrumented tests, as we are doing here. If you wish to have deferred tasks be unit tested outside of Android, consider isolating that logic in another class that your Worker then happens to use.

## WorkManager and Side Effects

WorkManager has a fairly clean and easy API and hides a lot of the complexity of scheduling background work that is not time-sensitive.

However, it has side effects.

## WORKING WITH WORKMANAGER

---

To be able to restart your scheduled work after a reboot, WorkManager registers an ACTION\_BOOT\_COMPLETED receiver named androidx.work.impl.background.systemalarm.RescheduleReceiver. To be a good citizen, WorkManager only enables that receiver when you have relevant work and disables it otherwise. That way, your app does not unnecessarily slow down the boot process if there is no reason for your app to get control at boot time.

However, enabling and disabling a component, such as a receiver, triggers an ACTION\_PACKAGE\_CHANGED broadcast. Few apps directly have any code that watches for this broadcast, let alone would be harmed by having that broadcast be sent more times than might otherwise be necessary.

App widgets, though, *are* affected by ACTION\_PACKAGE\_CHANGED. Specifically, ACTION\_PACKAGE\_CHANGED triggers an onUpdate() call to your AppWidgetProvider.

That too may not be a problem for most app widgets. Ideally, your AppWidgetProvider makes no assumptions about when, or how frequently, it gets called with onUpdate(). However, there is one area where this *is* a problem: with an AppWidgetProvider scheduling work in onUpdate(). The flow then becomes:

- A “regular” onUpdate() call comes in
- Your AppWidgetProvider schedules some work with WorkManager
- WorkManager enables RescheduleReceiver
- That triggers ACTION\_PACKAGE\_CHANGED, which triggers an onUpdate() call
- Your AppWidgetProvider schedules some work with WorkManager again
- WorkManager eventually gets through those two pieces of work
- WorkManager disables RescheduleReceiver, since it is no longer needed
- That triggers ACTION\_PACKAGE\_CHANGED, which triggers an onUpdate() call
- Your AppWidgetProvider schedules some work with WorkManager
- WorkManager enables RescheduleReceiver
- And we’re in an infinite loop

The recommendation from Google is to avoid unconditionally scheduling work with WorkManager from onUpdate(). Instead, only do it if you know that the work is needed and that it is safe to do so, meaning that you will not get into the infinite loop.

That advice may be difficult for some to implement.

This, and any other possible side-effects of WorkManager, are not documented. So, you need to be a bit careful about your use of WorkManager:

## WORKING WITH WORKMANAGER

---

- If your app responds to ACTION\_PACKAGE\_CHANGED broadcasts, directly or indirectly, it may not be safe to schedule work there, lest you wind up in the infinite loop scenario described above.
- If your app responds to ACTION\_BATTERY\_OK, ACTION\_BATTERY\_LOW, ACTION\_POWER\_CONNECTED, ACTION\_POWER\_DISCONNECTED, ACTION\_DEVICE\_STORAGE\_LOW, ACTION\_DEVICE\_STORAGE\_OK, CONNECTIVITY\_CHANGE, ACTION\_TIME\_SET, or ACTION\_TIMEZONE\_CHANGED, bear in mind that WorkManager has receivers for those broadcasts in your app. These are all disabled at the outset, but presumably WorkManager has code to enable them based on certain conditions, such as certain constraints that you set in your work requests. Be careful about scheduling work with WorkManager on those broadcasts as well.
- If your app responds to ACTION\_BOOT\_COMPLETED broadcasts, bear in mind that WorkManager also depends on this broadcast. Your respective receivers might be invoked in any order. It may not be safe to schedule work here, as WorkManager might assume that its own ACTION\_BOOT\_COMPLETED receiver has completed its work by the time you try scheduling new work. While I would not expect an infinite loop scenario, this is the sort of edge case that requires a lot of testing to ensure everything will work as expected.
- WorkManager has a ContentProvider that it bakes into your app as well. While scheduling your own work from onCreate() of a ContentProvider would be rather odd, it's possible that somebody might want to do that. Be careful, as WorkManager may not be fully ready for operation at that point. Note that, last time I tested it, all ContentProvider instances are created before onCreate() of Application, so *probably* it is safe to schedule work there.

There may be other edge and corner cases beyond these. So, while WorkManager is nice, make sure that you thoroughly test your use of it.

# **Creating a New Project**

---

This chapter will be added in a future update to this book!

Licensed solely for use by Patrocinio Rodriguez

# **Shrinking Your App**

This chapter will be added in a future update to this book!

Licensed solely for use by Patrocinio Rodriguez

## **Using the SDK Manager**

This chapter will be added in a future update to this book!

Licensed solely for use by Patrocinio Rodriguez

# **Using the AVD Manager and the Emulator**

---

This chapter will be added in a future update to this book!

Licensed solely for use by Patrocinio Rodriguez

# **Configuring Your Project**

---

This chapter will be added in a future update to this book!

Licensed solely for use by Patrocinio Rodriguez

# **Configuring Android Studio**

---

This chapter will be added in a future update to this book!

Licensed solely for use by Patrocinio Rodriguez