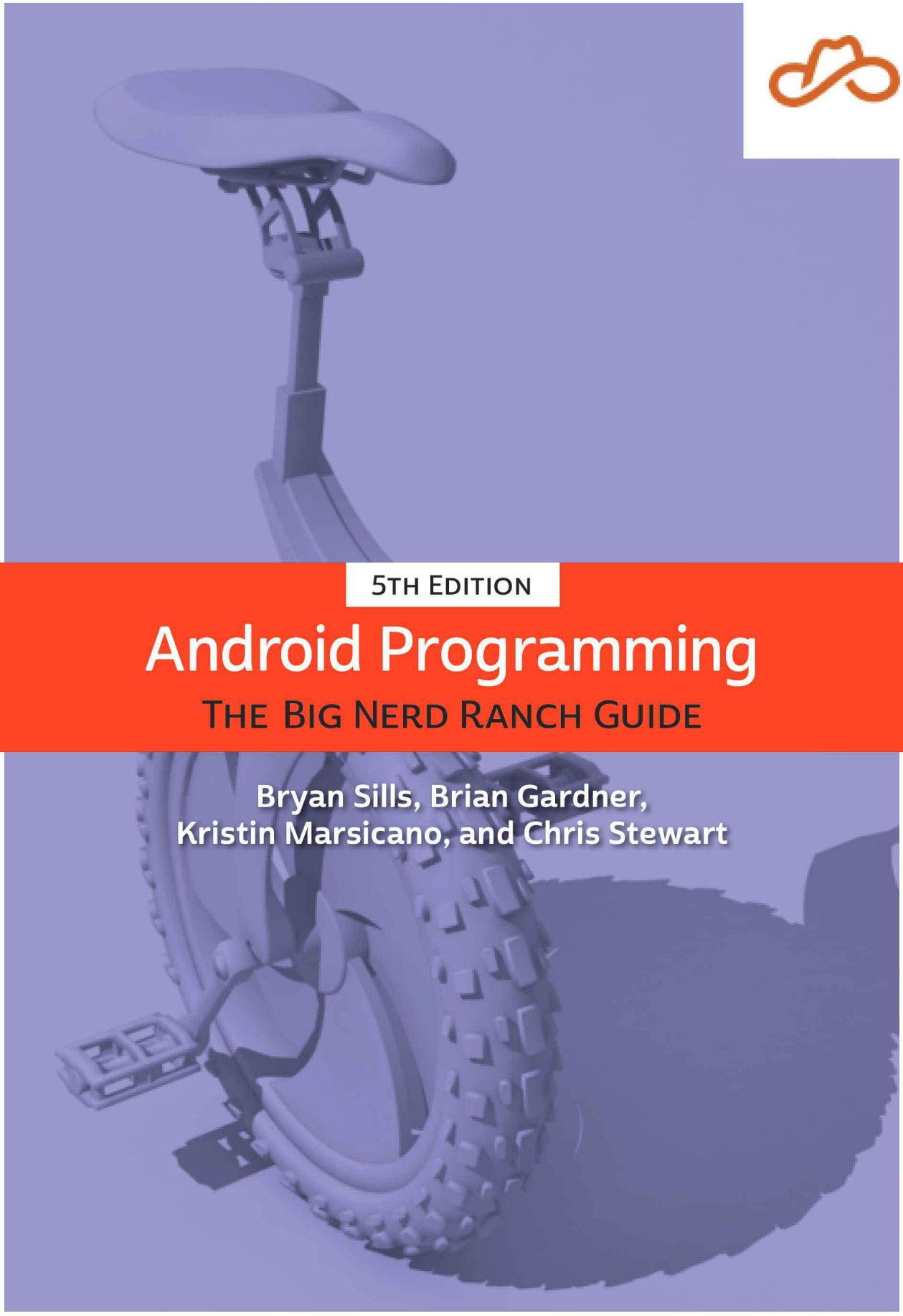


5TH EDITION

# Android Programming

## THE BIG NERD RANCH GUIDE

Bryan Sills, Brian Gardner,
Kristin Marsicano, and Chris Stewart

# Android Programming: The Big Nerd Ranch Guide

by Bryan Sills, Brian Gardner, Kristin Marsicano and Chris Stewart

Copyright © 2022 Big Nerd Ranch, LLC.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, contact

Big Nerd Ranch, LLC.
750 Glenwood Ave SE, Suite 200
Atlanta, GA 30316
(770) 817-6373
https://www.bignerdranch.com/
book-comments@bignerdranch.com

The 10-gallon hat logo is a trademark of Big Nerd Ranch, Inc.

Exclusive worldwide distribution of the English edition of this book by

Pearson Technology Group
800 East 96th Street
Indianapolis, IN 46240 USA
http://www.informit.com

The authors and publisher have taken care in writing and printing this book but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

ISBN-10 0137645635
ISBN-13 978-0137645633

Fifth edition, second printing, February 2023
Release D.5.2.1

# Dedication

*To my friends and family, for all of the love, support, and encouragement throughout my life.*

— B.S.

*To my wife, Carley, for supporting me in all that I do and reminding me of what's important along the way.*

— B.G.

*To Phil, Noah, and Sam for loving and supporting me through multiple editions of this book.*

— K.M.

*To my dad, David, for teaching me the value of hard work. To my mom, Lisa, for pushing me to always do the right thing.*

— C.S.

# Acknowledgments

With this being our fifth edition, we find ourselves used to saying this. It always needs to be said, though: Books are not created by authors alone. They are shepherded into existence by a community of collaborators, risk-takers, and other supporters, without whom the burden of comprehending and writing all this material would be overwhelming.

- Brian Hardy, who, along with Bill Phillips, had the gusto to bring the very first edition of this book into the world.
- Andrew Bailey, the most intelligent rubber duck we have ever met. You deserve a spot on the list of authors. In addition to writing all the chapters on Jetpack Compose, you were always the first person we reached out to to talk through tough conceptual decisions and figure out the most effective way to explain new concepts. Your architecture and teaching expertise was invaluable in crafting the solutions presented in this book. Your impact is felt throughout its pages.
- Mark Duran, for proposing the updated solution for PhotoGallery and for feedback throughout the process.
- Dave Severns, for his eagle eyes and for providing updated screenshots and solutions for all the projects.
- Anthony Kiniyalocts, Bryan Lindsey, Brandon Himes, Ben Zweber, Ben Bradley, Daniel Cook, Donovan LaDuke, Christian Keur, Michael Yotive, Tony Kazanjian, and Max McKinley. All y'all lent your time and expertise to make this the best resource for anyone to learn Android development.
- Desirée Johnson, one of our fantastically talented Big Nerd Ranch designers, who whipped together the nifty cheat sheet attached to this book.
- Kar Loong Wang, another of our amazing designers, for his illustrations of pizza and toppings for Coda Pizza and the handcuffs icon for CriminalIntent.
- Eric Maxwell, David Greenhalgh, Josh Skeen, Jeremy Sherman, Jamie Lee, Andrew Marshall, Zack Simon, Jeremy Kliphouse, Lixin Wang, Brett McCormick, and everyone else who has made a contribution to this book over its many editions.
- Our editor, Elizabeth Holaday. The famous beat author William S. Burroughs sometimes wrote by cutting up his work into little pieces, throwing them in the air, and publishing the rearrangement. Without a strong editor like Liz, our confusion and simpleminded excitement may have caused us to resort to such techniques. We are thankful that she was there to impose focus, precision, and clarity on our drafts.
- Ellie Volckhausen, who designed our cover.
- Simone Payment, our proofreader. Thank you for sanding away the remaining rough edges of this book.
- Chris Loper at IntelligentEnglish.com, who designed and produced the print and eBook versions of the book. His DocBook toolchain made life much easier, too.

- Thanks to Aaron Hillegass and the leadership team at Big Nerd Ranch. As a practical matter, it is not possible to do this work without Big Nerd Ranch, the company Aaron founded. Thank you.

Finally, thanks to our students. There is a feedback loop between us and our students: We teach them out of these materials, and they respond to them. Without that loop, this book could never have existed, nor could it be maintained. If Big Nerd Ranch books are special (and we hope they are), it is that feedback loop that makes them so. Thank you.

# Table of Contents


Learning Android ..... xv
Prerequisites ..... xv
What's New in the Fifth Edition? ..... xvi
Kotlin vs Java ..... xvi
How to Use This Book ..... xvii
How This Book Is Organized ..... xvii
Challenges ..... xviii
Are you more curious? ..... xviii
Typographical Conventions ..... xviii
Android Versions ..... xviii
The Necessary Tools ..... xix
Downloading and Installing Android Studio ..... xix
Downloading Earlier SDK Versions ..... xix
A Hardware Device ..... xx
1. Your First Android Application ..... 1
App Basics ..... 3
Creating an Android Project ..... 4
Navigating in Android Studio ..... 7
Laying Out the UI ..... 9
The view hierarchy ..... 14
View attributes ..... 15
Creating string resources ..... 16
Previewing the layout ..... 17
From Layout XML to View Objects ..... 19
Resources and resource IDs ..... 20
Wiring Up Views ..... 21
Getting references to views ..... 22
Setting listeners ..... 23
Making Toasts ..... 24
Running on the Emulator ..... 26
For the More Curious: The Android Build Process ..... 30
Android build tools ..... 31
Challenges ..... 32
Challenge: Switching Your Toast for a Snackbar ..... 32
2. Interactive User Interfaces ..... 33
Creating a New Class ..... 34
Updating the Layout ..... 36
Wiring Up the User Interface ..... 39
Adding an Icon ..... 45
Referencing resources in XML ..... 47
Screen Pixel Densities ..... 48
Running on a Device ..... 50
Challenge: Add a Listener to the TextView ..... 52
Challenge: Add a Previous Button ..... 52
3. The Activity Lifecycle ..... 53

Rotating GeoQuiz ..... 53
Activity States and Lifecycle Callbacks ..... 55
Logging the Activity Lifecycle ..... 57
Making log messages ..... 57
Using Logcat ..... 59
Exploring How the Activity Lifecycle Responds to User Actions ..... 60
Temporarily leaving an activity ..... 60
Finishing an activity ..... 63
Rotating an activity ..... 64
Device Configuration Changes and the Activity Lifecycle ..... 65
For the More Curious: Creating a Landscape Layout ..... 66
For the More Curious: UI Updates and Multi-Window Mode ..... 67
For the More Curious: Log Levels ..... 68
Challenge: Preventing Repeat Answers ..... 68
Challenge: Graded Quiz ..... 68
4. Persisting UI State ..... 69
Including the ViewModel Dependency ..... 70
Adding a ViewModel ..... 71
ViewModel lifecycle ..... 72
Add data to your ViewModel ..... 75
Saving Data Across Process Death ..... 78
For the More Curious: Jetpack, AndroidX, and Architecture Components ..... 81
For the More Curious: Avoiding a Half-Baked Solution ..... 81
For the More Curious: Activity and Instance State ..... 82
5. Debugging Android Apps ..... 83
Exceptions and Stack Traces ..... 85
Diagnosing misbehaviors ..... 86
Logging stack traces ..... 87
Setting breakpoints ..... 88
Android-Specific Debugging ..... 93
Using Android Lint ..... 93
Build issues ..... 96
Challenge: Using Conditional Breakpoints ..... 97
Challenge: Exploring the Layout Inspector ..... 97
Challenge: Exploring the Profiler ..... 97
6. Testing ..... 99
Two Types of Tests ..... 99
JVM Tests ..... 101
Instrumented Tests with Espresso and ActivityScenario ..... 106
Challenge: Asserting Yourself ..... 112
7. Your Second Activity ..... 113
Setting Up a Second Activity ..... 115
Creating a new activity ..... 116
A new activity subclass ..... 119
Declaring activities in the manifest ..... 120
Adding a cheat button to MainActivity ..... 121
Starting an Activity ..... 122
Communicating with intents ..... 122

Passing Data Between Activities ..... 123
Using intent extras ..... 124
Getting a result back from a child activity ..... 127
How Android Sees Your Activities ..... 132
For the More Curious: startActivityForResult ..... 136
For the More Curious: The Back Button and the Activity Lifecycle ..... 137
Challenge: Closing Loopholes for Cheaters ..... 138
Challenge: Tracking Cheat Status by Question ..... 138
8. Android SDK Versions and Compatibility ..... 139
Android SDK Versions ..... 139
A sane minimum ..... 140
Minimum SDK version ..... 141
Target SDK version ..... 142
Compile SDK version ..... 142
Compatibility and Android Programming ..... 142
Jetpack libraries ..... 143
Safely adding code from later APIs ..... 143
Using the Android Developer Documentation ..... 147
Challenge: Reporting the Device's Android Version ..... 149
Challenge: Limited Cheats ..... 149
9. Fragments ..... 151
The Need for UI Flexibility ..... 152
Introducing Fragments ..... 153
Starting CriminalIntent ..... 154
Creating a new project ..... 156
Creating a Data Class ..... 159
Creating a Fragment ..... 160
Defining CrimeDetailFragment's layout ..... 160
Creating the CrimeDetailFragment class ..... 162
Hosting a Fragment ..... 169
Defining a FragmentContainerView ..... 169
The FragmentManager ..... 171
The fragment lifecycle ..... 173
Fragments and memory management ..... 174
Challenge: Testing with FragmentScenario ..... 176
10. Displaying Lists with RecyclerView ..... 177
Adding a New Fragment and ViewModel ..... 178
ViewModel lifecycle with fragments ..... 179
Adding a RecyclerView ..... 181
Implementing a LayoutManager ..... 183
Creating an Item View Layout ..... 184
Implementing a ViewHolder ..... 186
Implementing an Adapter to Populate the RecyclerView ..... 188
Setting the RecyclerView's adapter ..... 191
Recycling Views ..... 193
Cleaning Up Binding List Items ..... 194
Responding to Presses ..... 195
Lists and Grids: Past, Present, and Future ..... 196

For the More Curious: A Smarter Adapter with ListAdapter ..... 197
Challenge: RecyclerView View Types ..... 198
11. Creating User Interfaces with Layouts and Views ..... 199
Introducing ConstraintLayout ..... 201
Introducing the Layout Editor ..... 202
Using ConstraintLayout ..... 206
Making room ..... 207
Adding views ..... 209
ConstraintLayout's inner workings ..... 214
Editing properties ..... 215
Making list items dynamic ..... 221
Styles, Themes, and Theme Attributes ..... 223
For the More Curious: Margins vs Padding ..... 226
For the More Curious: Advanced Features in ConstraintLayout ..... 227
Challenge: Formatting the Date ..... 227
12. Coroutines and Databases ..... 229
An Introduction to Asynchronous Code on Android ..... 230
Using coroutines ..... 231
Consuming data from coroutines ..... 235
Creating a Database ..... 240
Room architecture component library ..... 240
Defining entities ..... 242
Creating a database class ..... 243
Creating a type converter ..... 243
Defining a Data Access Object ..... 245
Accessing the Database Using the Repository Pattern ..... 247
Importing Prepopulated Data ..... 250
Querying the Database ..... 252
Keeping the Changes Flowing ..... 253
Challenge: Addressing the Schema Warning ..... 257
For the More Curious: Singletons ..... 258
13. Fragment Navigation ..... 259
Performing Navigation ..... 260
Implementing the Navigation component library ..... 261
Navigating to the detail screen ..... 266
Passing data to a fragment ..... 273
Unidirectional Data Flow ..... 279
Updating the Database ..... 285
For the More Curious: A Better List Preview ..... 288
Challenge: No Untitled Crimes ..... 289
14. Dialogs and DialogFragment ..... 291
Creating a DialogFragment ..... 292
Showing a DialogFragment ..... 293
Passing Data Between Two Fragments ..... 297
Passing data to DatePickerFragment ..... 298
Returning data to CrimeDetailFragment ..... 300
Challenge: More Dialogs ..... 302
15. The App Bar ..... 303

The Default App Bar ..... 304
Menus ..... 305
Defining a menu in XML ..... 307
Creating the menu ..... 309
Responding to menu selections ..... 312
For the More Curious: App Bar vs Action Bar vs Toolbar ..... 318
For the More Curious: Accessing the AppCompat App Bar ..... 319
Challenge: An Empty View for the RecyclerView ..... 320
Challenge: Deleting Crimes ..... 320
16. Implicit Intents ..... 321
Adding Buttons ..... 322
Adding a Suspect Property ..... 323
Using a Format String ..... 325
Using Implicit Intents ..... 327
Parts of an implicit intent ..... 327
Sending a crime report ..... 329
Asking Android for a contact ..... 334
Checking for responding activities ..... 340
Challenge: Another Implicit Intent ..... 344
17. Taking Pictures with Intents ..... 345
A Place for Your Photo ..... 346
File Storage ..... 350
Using FileProvider ..... 351
Designating a picture location ..... 353
Using a Camera Intent ..... 354
Scaling and Displaying Bitmaps ..... 358
Declaring Features ..... 362
Challenge: Detail Display ..... 362
18. Localization ..... 363
Localizing Resources ..... 364
Default resources ..... 367
Checking string coverage using the Translations Editor ..... 368
Targeting a region ..... 369
Configuration Qualifiers ..... 370
Prioritizing alternative resources ..... 371
Multiple qualifiers ..... 373
Finding the best-matching resources ..... 374
Testing Alternative Resources ..... 375
For the More Curious: More on Determining Device Size ..... 376
Challenge: Localizing Dates ..... 376
19. Accessibility ..... 377
TalkBack ..... 378
Explore by Touch ..... 382
Linear navigation by swiping ..... 382
Making Non-Text Elements Readable by TalkBack ..... 385
Adding content descriptions ..... 385
Making a view focusable ..... 388
Creating a Comparable Experience ..... 389

For the More Curious: Using TalkBack with an Emulator ..... 392
For the More Curious: Using Accessibility Scanner ..... 394
Challenge: Improving the List ..... 399
Challenge: Providing Enough Context for Data Entry ..... 399
Challenge: Announcing Events ..... 400
20. Making Network Requests and Displaying Images ..... 401
Creating PhotoGallery ..... 403
Networking Basics with Retrofit ..... 406
Defining an API interface ..... 407
Building the Retrofit object and creating an API instance ..... 408
Executing a web request ..... 410
Asking permission to network ..... 411
Moving toward the repository pattern ..... 412
Fetching JSON from Flickr ..... 414
Deserializing JSON text into model objects ..... 418
Handling errors ..... 424
Networking Across Configuration Changes ..... 426
Displaying Results in RecyclerView ..... 429
Displaying images ..... 431
For the More Curious: Managing Dependencies ..... 435
Challenge: Paging ..... 436
21. SearchView and DataStore ..... 437
Searching Flickr ..... 438
Using SearchView ..... 442
Responding to SearchView user interactions ..... 445
Simple Persistence with DataStore ..... 448
Defining UI State ..... 454
Challenge: Polishing Your App Some More ..... 458
22. WorkManager ..... 459
Creating a Worker ..... 461
Scheduling Work ..... 462
Checking for New Photos ..... 465
Notifying the User ..... 468
Providing User Control over Polling ..... 474
23. Browsing the Web and WebView ..... 481
One Last Bit of Flickr Data ..... 482
The Easy Way: Implicit Intents ..... 484
The Harder Way: WebView ..... 486
WebChromeClient ..... 491
WebView vs a Custom UI ..... 495
For the More Curious: WebView Updates ..... 495
For the More Curious: Chrome Custom Tabs (Another Easy Way) ..... 496
Challenge: Using the Back Button for Browser History ..... 498
24. Custom Views and Touch Events ..... 499
Setting Up the DragAndDraw Project ..... 500
Creating a Custom View ..... 500
Creating BoxDrawingView ..... 501
Handling Touch Events ..... 502

Tracking across motion events ..... 504
Rendering Inside onDraw(Canvas) ..... 506
For the More Curious: Detecting Gestures ..... 509
Challenge: Saving State ..... 509
Challenge: Rotating Boxes ..... 510
Challenge: Accessibility Support ..... 510
25. Property Animation ..... 511
Building the Scene ..... 511
Simple Property Animation ..... 515
View transformation properties ..... 518
Using different interpolators ..... 520
Color evaluation ..... 521
Playing Animators Together ..... 523
For the More Curious: Other Animation APIs ..... 525
Legacy animation tools ..... 525
Transitions ..... 525
Challenges ..... 526
26. Introduction to Jetpack Compose ..... 527
Creating a Compose Project ..... 529
Composing Your First UI ..... 532
Layouts in Compose ..... 534
Composable Functions ..... 536
Previewing Composables ..... 539
Customizing Composables ..... 542
Declaring inputs on a composable function ..... 542
Aligning elements in a row ..... 548
Specifying text styles ..... 550
The Compose Modifier ..... 552
The padding modifier ..... 552
Chaining modifiers and modifier ordering ..... 553
The clickable modifier ..... 557
Sizing composables ..... 559
Specifying a modifier parameter ..... 561
Building Screens with Composables ..... 561
Scrollable Lists with LazyColumn ..... 565
For the More Curious: Live Literals ..... 568
27. UI State in Jetpack Compose ..... 569
Philosophies of State ..... 570
Defining Your UI State ..... 571
Updating UIs with MutableState ..... 572
Recomposition ..... 578
remember ..... 580
State Hoisting ..... 581
State and Configuration Changes ..... 586
Parcelable and Parcelize ..... 587
For the More Curious: Coroutines, Flow, and Compose ..... 589
For the More Curious: Scrolling State ..... 591
For the More Curious: Inspecting Compose Layouts ..... 592

28. Showing Dialogs with Jetpack Compose ..... 593
Your First Dialog in Compose ..... 595
Dismissing the Dialog ..... 598
Setting the Dialog's Content ..... 600
Sending Results from a Dialog ..... 606
Challenge: Pizza Sizes and Drop-Down Menus ..... 609
29. Theming Compose UIs ..... 611
Images ..... 613
Image's contentDescription ..... 615
Adding more images ..... 616
Customizing the Image composable ..... 618
Adding a header to LazyColumn ..... 627
MaterialTheme ..... 628
Scaffold and TopAppBar ..... 633
CompositionLocal ..... 635
Removing AppCompat ..... 640
For the More Curious: Accompanist ..... 644
For the More Curious: Creating Your Own CompositionLocals ..... 645
Challenge: Animations ..... 646
30. Afterword ..... 647
The Final Challenge ..... 647
Shameless Plugs ..... 647
Thank You ..... 648
Index ..... 649

# Learning Android

As a beginning Android programmer, you face a steep learning curve. Learning Android is like moving to a foreign city: Even if you speak the language, it will not feel like home at first. Everyone around you seems to understand things that you are missing. Things you already knew turn out to be dead wrong in this new context.

Android has a culture. That culture speaks Kotlin or Java (or a bit of both), but knowing Kotlin or Java is not enough. Getting your head around Android requires learning many new ideas and techniques. It helps to have a guide through unfamiliar territory.

That is where we come in. At Big Nerd Ranch, we believe that to be an Android programmer, you must:

- *write* Android applications
- *understand* what you are writing

This guide will help you do both. We have trained thousands of professional Android programmers using it. We will lead you through writing several Android applications, introducing concepts and techniques as needed. When there are rough spots, or when some things are tricky or obscure, you will face them head on, and we will do our best to explain why things are the way they are.

This approach allows you to put what you have learned into practice in a working app right away rather than learning a lot of theory and then having to figure out how to apply it all later. You will come away with the experience and understanding you need to get going as an Android developer.

## Prerequisites

To use this book, you need to be familiar with Kotlin, including classes and objects, interfaces, listeners, packages, inner classes, object expressions, and generic classes.

If these concepts do not ring a bell, you will be in the weeds by page 2. Start instead with an introductory Kotlin book and return to this book afterward. There are many excellent introductory books available, so you can choose one based on your programming experience and learning style. May we recommend *Kotlin Programming: The Big Nerd Ranch Guide*?

If you are comfortable with object-oriented programming concepts, but your Kotlin is a little shaky, you will probably be OK. We will provide some brief explanations about Kotlin specifics throughout the book. But keep a Kotlin reference handy in case you need more support.

## What's New in the Fifth Edition?

The last edition of *Android Programming: The Big Nerd Ranch Guide* was released in October 2019. A lot has changed since then.

In 2019, reactive programming was gaining popularity as a way to architect Android code into maintainable and extensible structures. With the release of Jetpack Compose in 2021, Google poured gasoline onto that flame, supercharging the reactive programming movement. Reactive programming and Jetpack Compose's declarative framework fit together seamlessly and provide an excellent foundation to build modern Android apps.

Jetpack Compose is the future of Android development, and the fifth edition of this book is intended to prepare readers for that future. In addition to four new chapters introducing readers to Jetpack Compose, changes throughout the book are intended to ease the transition from developing apps with Android's existing UI toolkit to developing apps with Jetpack Compose. For example, there are many ways to write asynchronous code on Android, but this book exclusively uses Kotlin coroutines to perform asynchronous operations. Coroutines are baked directly into Jetpack Compose's API as well as being excellent tools to interact with UI written with Android's existing UI toolkit. We also reworked many of our projects to follow the unidirectional data flow architecture pattern. The unidirectional data flow pattern is essential to building apps with Jetpack Compose – and it also helps organize code when building apps with Android's existing UI toolkit.

Other changes in this fifth edition go beyond Jetpack Compose. For example, testing is an integral part of building modern Android apps, and we have rewritten the content around testing from the ground up with practical examples. Also, to reflect how modern Android applications are developed, this book now leans on libraries from Google and third parties. Apps in this book now use the Navigation component library to manage navigation between screens and libraries like Retrofit, Moshi, and Coil – as well as the Jetpack libraries – to handle other core features. We use libraries like these daily in our lives as Android developers.

For the second printing of this edition, we addressed some typos in Chapter 9, including the name of the variable used to inflate **`Fragment`** layouts. Also, in Chapter 17, we changed an argument passed to the **`createIntent`** function in Listing 17.12 from null to an empty **`Uri`**. Passing null works with certain versions of the Jetpack libraries, but is technically incorrect and causes crashes with newer library versions. The **`createIntent`** function needs some non-null input, even if the input is not used for any functionality.

## Kotlin vs Java

Official support for Kotlin for Android development was announced at Google I/O in 2017. Before that, there was an underground movement of Android developers using Kotlin even though it was not officially supported. Since 2017, Kotlin has become widely adopted, and it is most developers' preferred language for Android development. At Big Nerd Ranch, we use Kotlin for all our app development projects – even legacy projects that are mostly Java.

In the years following Google's announcement, Kotlin has become even more essential as a tool in the modern Android developer's toolbox. Going beyond mere compatibility with the existing platform, there are now tools and features on the Android platform that can only be used with Kotlin – including Jetpack Compose. You cannot write apps in Jetpack Compose with Java.

The Android framework was originally written in Java. This means most of the Android classes you interact with are Java. Luckily, Kotlin is interoperable with Java, so you should not run into any issues.

But even though you can still write apps in Java, the future of Android as a platform is with Kotlin. Google and the entire Android developer ecosystem are investing heavily into making Kotlin development easier and more useful on Android. It is not a fad, and it is not going away. Join the party; we think it is quite nice here.

## How to Use This Book

This book is not a reference book. Its goal is to get you over the initial hump to where you can get the most out of the reference and recipe books available. It is based on our five-day class at Big Nerd Ranch. As such, it is meant to be worked through from the beginning. Chapters build on each other, and skipping around is unproductive.

In our classes, students work through these materials, but they also benefit from the right environment – a dedicated classroom, good food and comfortable board, a group of motivated peers, and an instructor to answer questions.

As a reader, you want your environment to be similar. That means getting a good night's rest and finding a quiet place to work. These things can help, too:

- Start a reading group with your friends or coworkers.
- Arrange to have blocks of focused time to work on chapters.
- Participate in the forum for this book at `forums.bignerdranch.com`.
- Find someone who knows Android to help you out.

## How This Book Is Organized

As you work through this book, you will write six Android apps. A couple are very simple and take only a chapter to create. Others are more complex. The longest app spans 11 chapters. All are designed to teach you important concepts and techniques and give you direct experience using them.

| | |
|---|---|
| *GeoQuiz* | In your first app, you will explore the fundamentals of Android projects, activities, layouts, and explicit intents. You will also learn how to handle configuration changes seamlessly. |
| *CriminalIntent* | The largest app in the book, CriminalIntent lets you keep a record of your colleagues' lapses around the office. You will learn to use fragments, list-backed interfaces, databases, menus, the camera, implicit intents, and more. |
| *PhotoGallery* | A Flickr client that downloads and displays photos from Flickr's public feed, this app will take you through scheduling background work, multi-threading, accessing web services, and more. |
| *DragAndDraw* | In this simple drawing app, you will learn about handling touch events and creating custom views. |
| *Sunset* | In this toy app, you will create a beautiful representation of a sunset over open water while learning about animations. |
| *Coda Pizza* | This app will introduce you to Jetpack Compose, the newest way to create Android UIs. You will learn how to manage application state and how to use the declarative framework to describe how your UI should render itself. |

### Challenges

Most chapters have a section at the end with exercises for you to work through. This is your opportunity to use what you have learned, explore the documentation, and do some problem-solving on your own. We strongly recommend that you do the challenges. Going off the beaten path and finding your way will solidify your learning and give you confidence with your own projects. If you get lost, you can always visit `forums.bignerdranch.com` for some assistance.

### Are you more curious?

Many chapters also have a section at the end labeled "For the More Curious." These sections offer deeper explanations or additional information about topics presented in the chapter. The information in these sections is not absolutely essential, but we hope you will find it interesting and useful.

## Typographical Conventions

All code and XML listings are in a fixed-width font. Code or XML that you need to type in is always bold. Code or XML that should be deleted is struck through. For example, in the following function implementation, you are deleting the call to **`Toast.makeText(…).show()`** and adding the call to **`checkAnswer(true)`**.

```
trueButton.setOnClickListener { view: View ->
    Toast.makeText(
        this,
        R.string.correct_toast,
        Toast.LENGTH_SHORT
    )
        .show()
    checkAnswer(true)
}
```

## Android Versions

This book teaches Android development for the versions of Android in wide use at the time of writing. For this edition, that is Android 7.0 Nougat (N, API level 24) – Android 12L (Sv2, API level 32). That being said, since Google has invested heavily in providing backward-compatible solutions for Android, most of the code in this book will still work on older versions of Android – often supporting versions as old as Android 5.0 Lollipop (L, API level 21).

While there is still limited use of older versions of Android, we find that for many developers the amount of effort required to support those versions is not worth the reward. You will learn more about Android versions and how to pick the right version for your application in Chapter 8.

As new versions of Android and Android Studio are released, the techniques you learn in this book will continue to work, thanks to Android's backward compatibility support (discussed in Chapter 8). We will keep track of changes at `forums.bignerdranch.com` and offer notes on using this book with the latest versions. We may also make minor changes to this book in subsequent printings to account for any changes, such as updating screenshots or button names.

# The Necessary Tools

To get started with this book, you will need Android Studio. Android Studio is an integrated development environment used for Android development that is based on the popular IntelliJ IDEA.

An install of Android Studio includes:

*Android SDK*

: the latest version of the Android SDK

*Android SDK tools and platform tools*

: tools for debugging and testing your apps

*A system image for the Android emulator*

: a tool for creating and testing your apps on different virtual devices

As of this writing, Android Studio is under active development and is frequently updated. Be aware that you may find differences between your version of Android Studio and what you see in this book. Visit `forums.bignerdranch.com` for help with these differences.

## Downloading and Installing Android Studio

Android Studio is available from Android's developer site at `developer.android.com/studio`. It includes everything you need to build and run Android applications, including a built-in installation of the Java Development Kit.

If you want to build and compile Android apps from somewhere outside Android Studio, such as from the command line, you need to have a local installation of the Java Development Kit. The latest version of the Android Gradle plugin, which is the tool that builds and compiles Android apps, requires Java 11. More recent versions of the Java Development Kit should also work just fine. If you are having problems, return to `developer.android.com/studio` for more information.

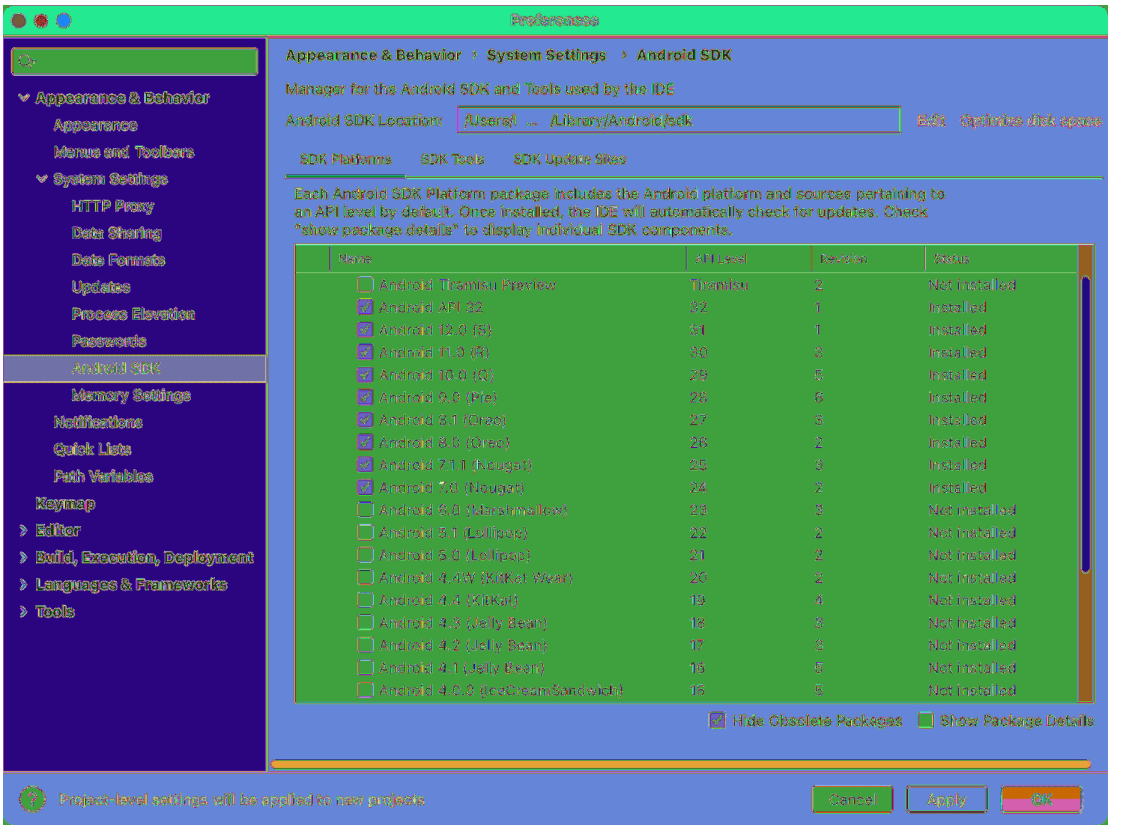
## Downloading Earlier SDK Versions

Android Studio provides the SDK and the emulator system image from the latest platform. However, you may want to test your apps on earlier versions of Android.

You can get components for each platform using the Android SDK Manager. In Android Studio, select Tools → SDK Manager. (You will only see the Tools menu if you have a project open. If you have not created a project yet, you can instead access the SDK Manager from the Android Welcome dialog. Click the three-dot overflow menu in the dialog's toolbar and select SDK Manager.)

The SDK Manager is shown in Figure 1.

Figure 1  Android SDK Manager




Select and install each version of Android that you want to test with. Note that downloading these components may take a while.

The Android SDK Manager is also how you can get Android's latest releases, like a new platform or an update of the tools.

## A Hardware Device

The emulator is useful for testing apps. However, it is no substitute for an actual Android device when measuring performance. If you have a hardware device, we recommend using it at times when working through this book. You will learn how to connect your device in Chapter 2.

# 1

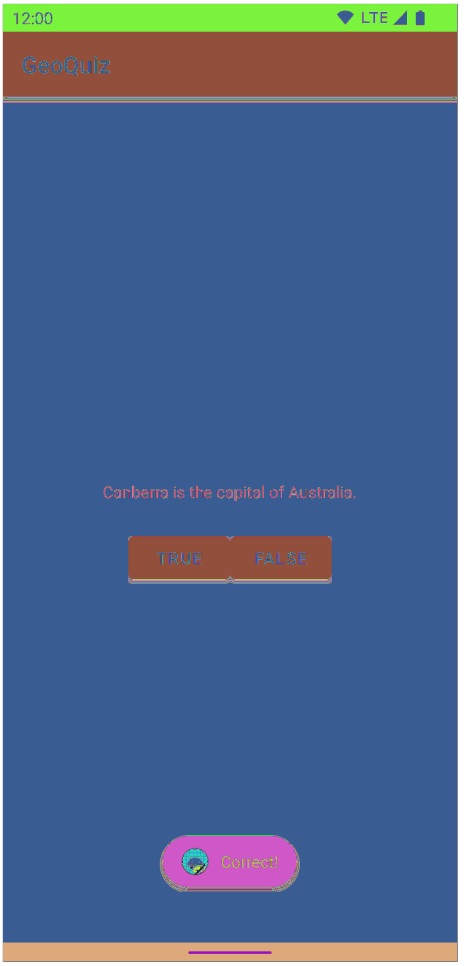
# Your First Android Application

This first chapter is full of the new concepts and moving parts required to build an Android application. It is OK if you do not understand everything by the end of this chapter. You will be revisiting these ideas in greater detail as you proceed through the book.

The application you are going to create is called GeoQuiz. GeoQuiz tests the user's knowledge of geography. The user presses TRUE or FALSE to answer the question onscreen, and GeoQuiz provides instant feedback.

Figure 1.1 shows the result of a user pressing the TRUE button.

Figure 1.1 Do you come from a land down under?



# App Basics

Your GeoQuiz application will consist of an *activity* and a *layout*:

- An activity is an instance of **`Activity`**, a class in the Android SDK. An activity is an entry point into your application and is responsible for managing user interaction with a screen of information.

  You write subclasses of **`Activity`** to implement the functionality your app requires. A simple application may need only one subclass; a complex application may have many.

  GeoQuiz is a simple app and will start off with a single **`Activity`** subclass named **`MainActivity`**. **`MainActivity`** will manage the user interface, or UI, shown in Figure 1.1.

- A layout defines a set of UI objects and the objects' positions on the screen. A layout is made up of definitions written in XML. Each definition is used to create an object that appears onscreen, like a button or some text.

  GeoQuiz will include a layout file named `activity_main.xml`. The XML in this file will define the UI shown in Figure 1.1.

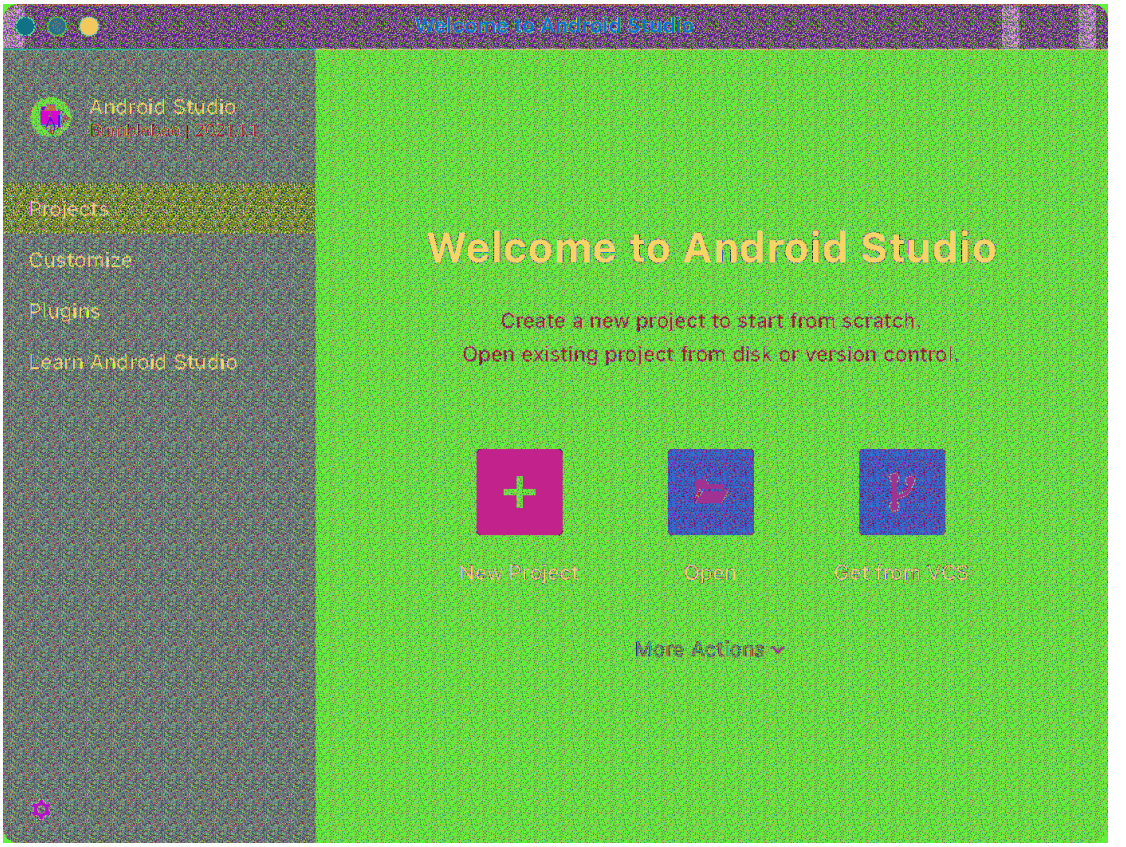
With those ideas in mind, let's build an app.

## Creating an Android Project

The first step is to create an Android *project*. An Android project contains the files that make up an application. To create a new project, first open Android Studio.

If this is your first time running Android Studio, you should see the Welcome dialog, as in Figure 1.2. If this is not the first time you have opened Android Studio since installing it, you may be brought directly to the last project you had open. To get back to the Welcome screen, close the project using File → Close Project.

Figure 1.2 Welcome to Android Studio



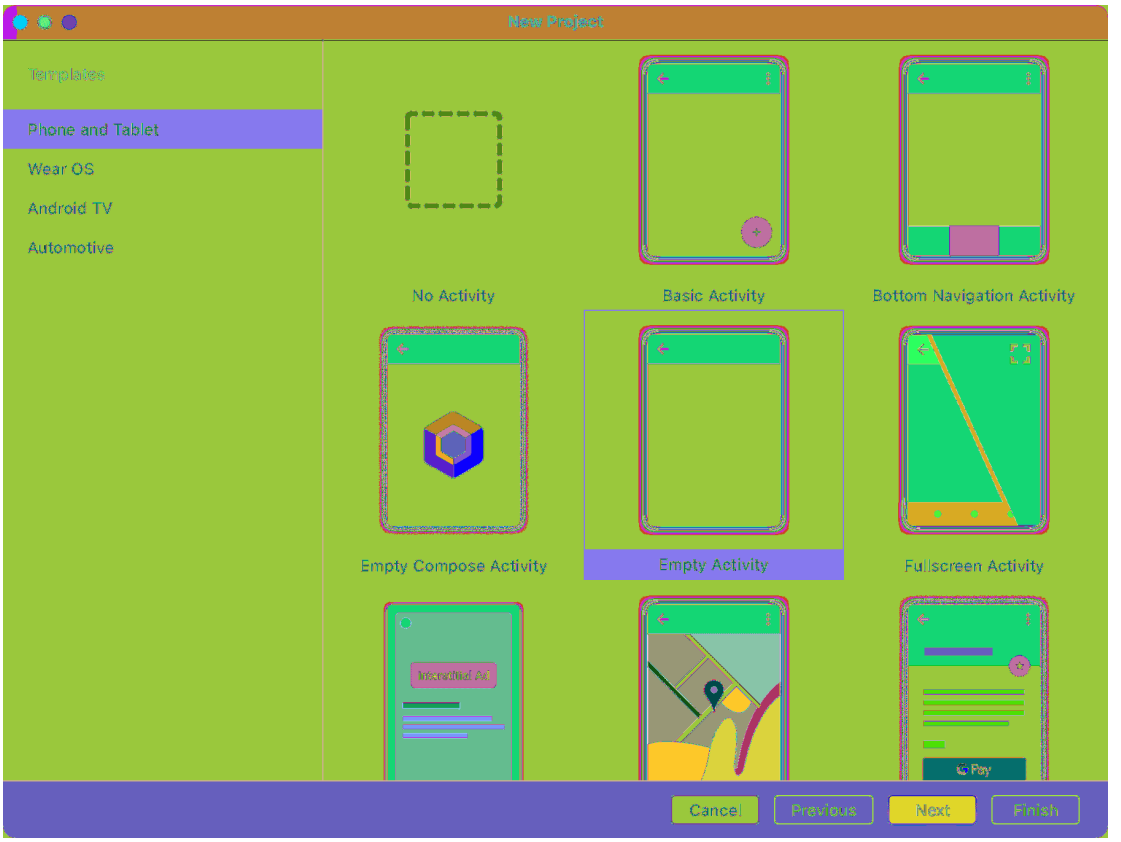

From the welcome screen, you can create your first Android Studio project. A quick disclaimer: We have written these instructions for Android Studio 2021.1.1 Patch 2 (Bumblebee). As Android Studio updates, the steps to create a project tend to change slightly (and sometimes they change a lot). Google often tweaks the New Project wizard and the templates it uses to generate new projects.

If you are using a newer version of Android Studio and are finding that these steps no longer match the latest wizards, check out our forums at `forums.bignerdranch.com`. We will post updates if there are different steps you need to follow to create your projects.

Now, back to business. In the Welcome dialog, choose New Project. If you do not see the dialog, choose File → New → New Project....

Welcome to the New Project wizard (Figure 1.3). Make sure Phone and Tablet is selected on the left. Pick Empty Activity and click Next.

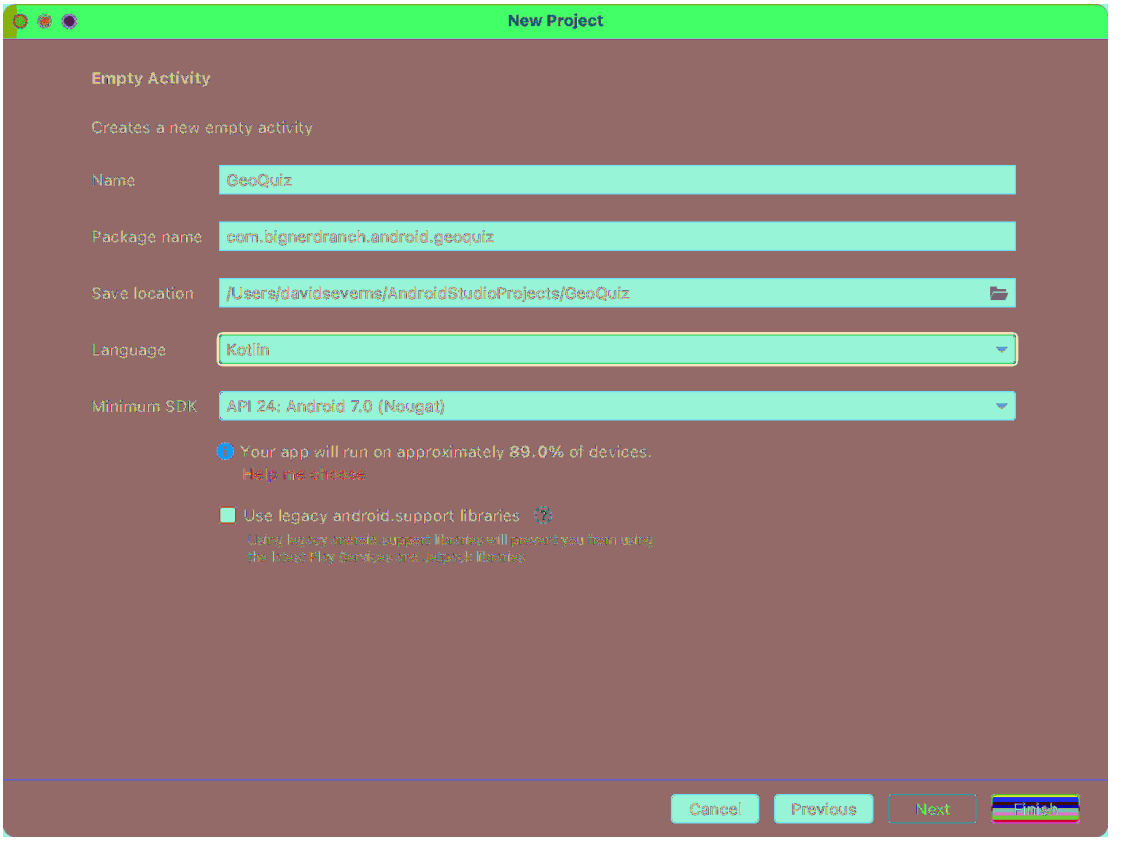
Figure 1.3 Choosing a project template




You should now see a dialog with fields for various project settings (Figure 1.4). Enter GeoQuiz as the application name. For the package name, enter com.bignerdranch.android.geoquiz. For the project location, use any location on your filesystem that you want.

Select Kotlin from the Language drop-down menu. Last, select a Minimum SDK of API 24: Android 7.0 (Nougat). You will learn about the different versions of Android in Chapter 8. Your screen should look like Figure 1.4.

Figure 1.4 Configuring your new project




Notice that the package name uses a "reverse DNS" convention: The domain name of your organization is reversed and suffixed with further identifiers. This convention keeps package names unique and distinguishes applications from each other on a device and in the Google Play Store.

Selecting Kotlin as the language tells Android Studio to include the dependencies necessary to write and build Kotlin code for your app. In the early years of Android development, Java was the only officially supported development language. But since 2017, the Android team has officially supported Kotlin for Android development. These days, Kotlin is preferred by most developers, ourselves included, which is why this book uses Kotlin.

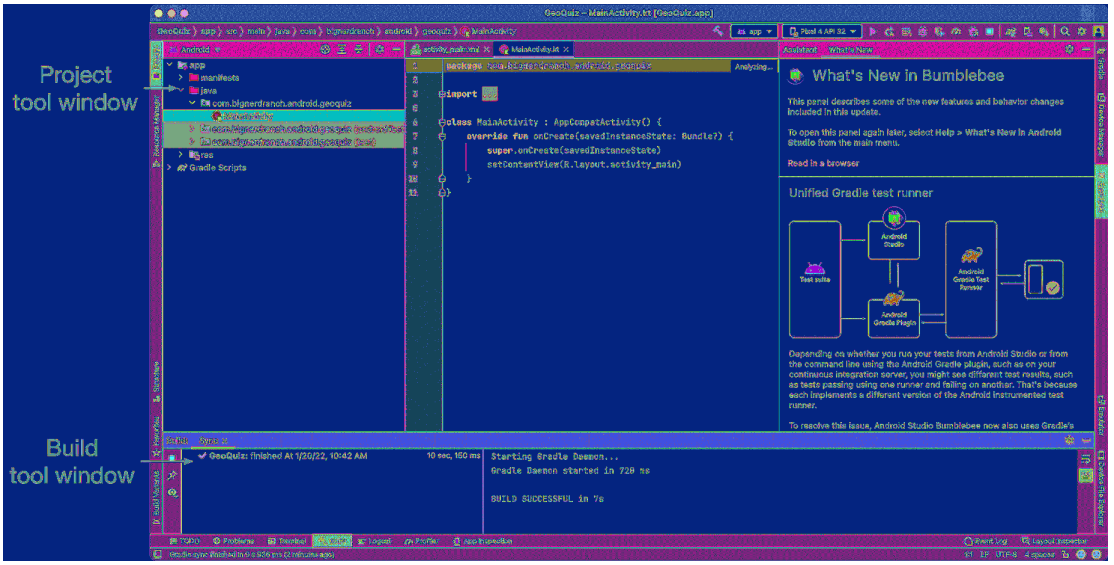
At this point, most of the Android platform is still in Java, and you can still choose to use Java in your projects outside of this book – but we do not recommend it for most projects. Most of the Android concepts and content you will learn here will still be applicable, but there are certain tools and libraries that only support Kotlin – such as Jetpack Compose, which you will learn about beginning in Chapter 26.

Click Finish. Android Studio will create and open your new project.

# Navigating in Android Studio

Android Studio opens your project in a window like the one shown in Figure 1.5. If you have launched Android Studio before, your window configuration might look a little different.

Figure 1.5 A fresh project window

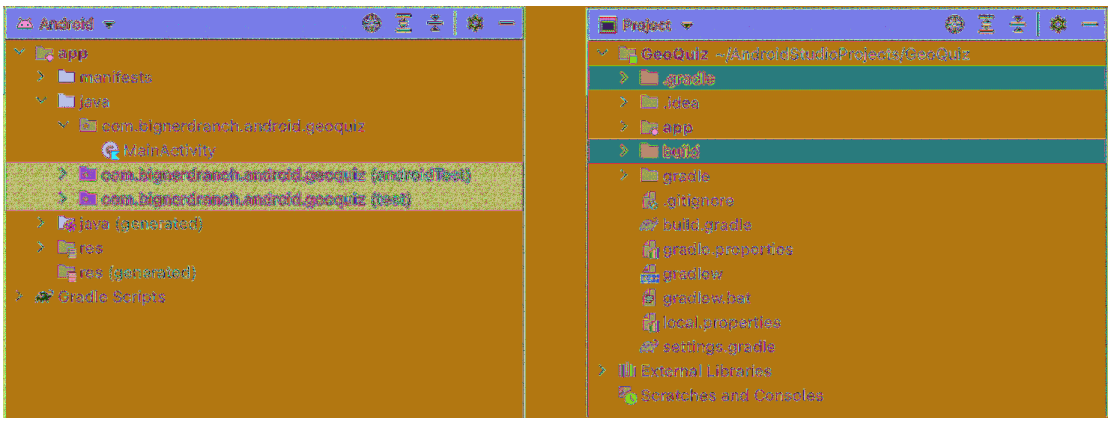



The different panes of the project window are called *tool windows*.

The lefthand view is the *project tool window*. From here, you can view and manage the files associated with your project.

By default, Android Studio displays the Android project view in the project tool window. This view hides the true directory structure of your Android project so that you can focus on the files and folders that you need most often. To see the files and folders in your project as they actually are, locate the dropdown at the top of the project tool window and click to expand it. Select the Project view to see the difference (Figure 1.6).

Figure 1.6 Project tool window: Android view vs Project view



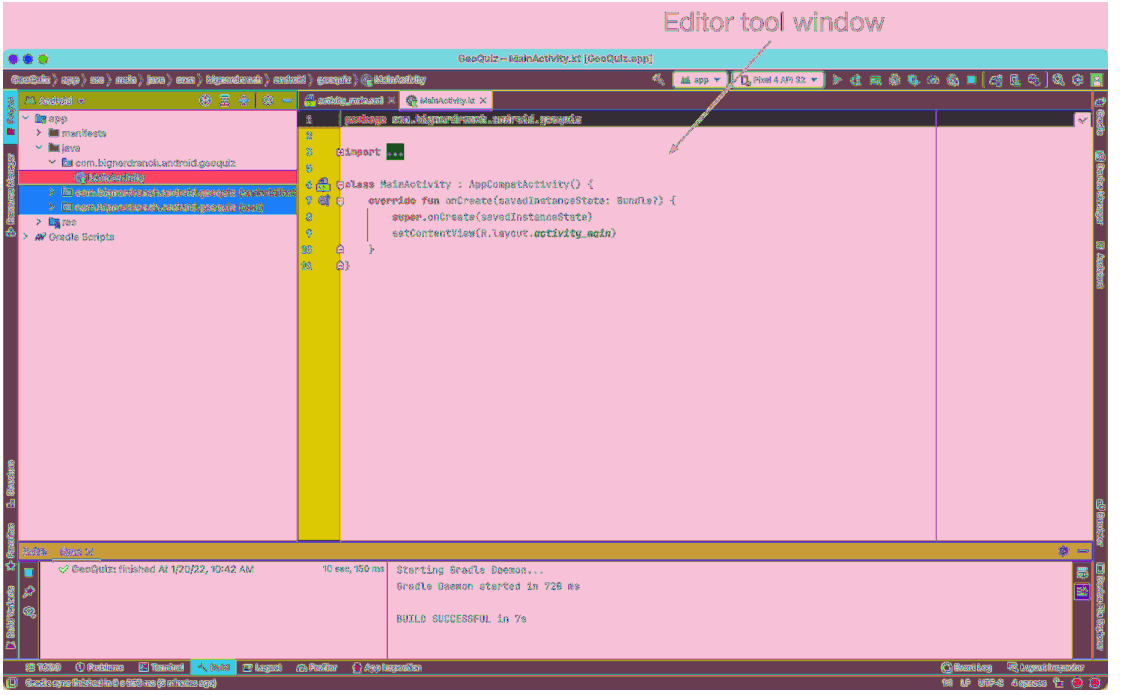
Feel free to use the Project view if it feels more natural to you, but throughout this book, we will use the Android view.

The view across the bottom of Android Studio is the *build tool window*. Here you can view details about the compilation process and the status of the build. When you created the project, Android Studio automatically built it. You should see in the build tool window that the process completed successfully. (If the build tool window did not open automatically, do not worry.)

On the righthand side of Android Studio, you might see the *assistant tool window* (shown in Figure 1.5). This view tells you about new features in Android Studio. If it is open, close it by clicking the hide button, which has a horizontal bar icon, in the top-right corner.

Android Studio automatically opens the files `activity_main.xml` and `MainActivity.kt` in the main view, called the *editor tool window* or just the *editor* (Figure 1.7). (If the editor is not visible or `MainActivity.kt` is not open, click the disclosure arrows to expand `app/java/com.bignerdranch.android.geoquiz/` in the project tool window. Double-click `MainActivity.kt` to open the file.)

Figure 1.7 Editor engaged



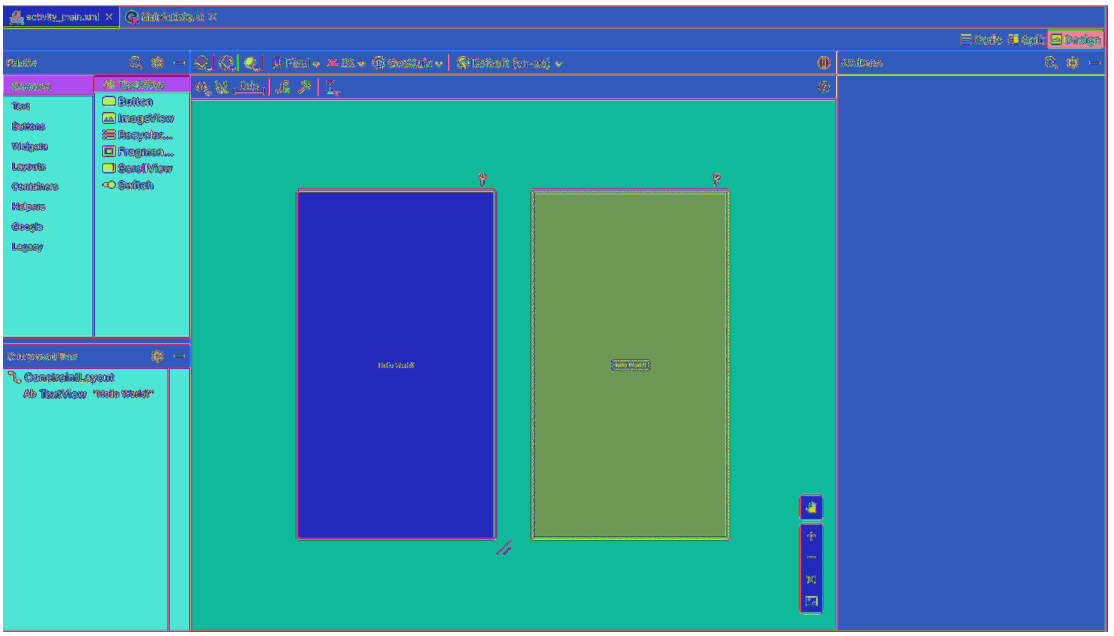

Notice the `Activity` suffix on the class name. This is not required, but it is an excellent convention to follow.

You can toggle the visibility of the various tool windows by clicking their names in the strips of tool window bars on the left, right, and bottom of the screen. There are keyboard shortcuts for many of these as well. If you do not see the tool window bars, click the gray square button in the lower-left corner of the main window or choose View → Appearance → Tool Window Bars. Tool windows and other panes can also be closed by clicking the hide button in their top-right corner.

## Laying Out the UI

Click the editor tab for the layout file, `activity_main.xml`. This will open the layout editor in the editor tool window (Figure 1.8). (If you do not see a tab for `activity_main.xml`, click the disclosure arrows to expand `app/res/layout/` in the project tool window. Double-click `activity_main.xml` to open the file. If `activity_main.xml` opens but shows XML instead of the layout editor, click the Design tab near the top-right corner of the editor tool window.)

Figure 1.8 Layout editor



By convention, a layout file is named based on the activity it is associated with: Its name begins with `activity_`, and the rest of the activity name follows in all lowercase, using underscores to separate words (a style called “snake_case”). So, for example, your layout file’s name is `activity_main.xml`, and the layout file for an activity called **`SplashScreenActivity`** would be named `activity_splash_screen`. This naming style is recommended for layouts as well as other resources that you will learn about later.

The layout editor shows a graphical preview of the file. Select the Code tab at the top right to see the backing XML.

Currently, `activity_main.xml` holds the default activity layout template. The template changes frequently, but the XML will look something 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>
```

The default activity layout defines two ***Views***: a **`ConstraintLayout`** and a **`TextView`**.

For most of this book, views will be the building blocks you will use to create a UI. (Starting in Chapter 26, you will see a new way to create UIs.) Some views show text. Some views show graphics. Others, like buttons, do things when touched. (You will sometimes see views that the user can see or interact with called "widgets," but we prefer to call them all "views.")

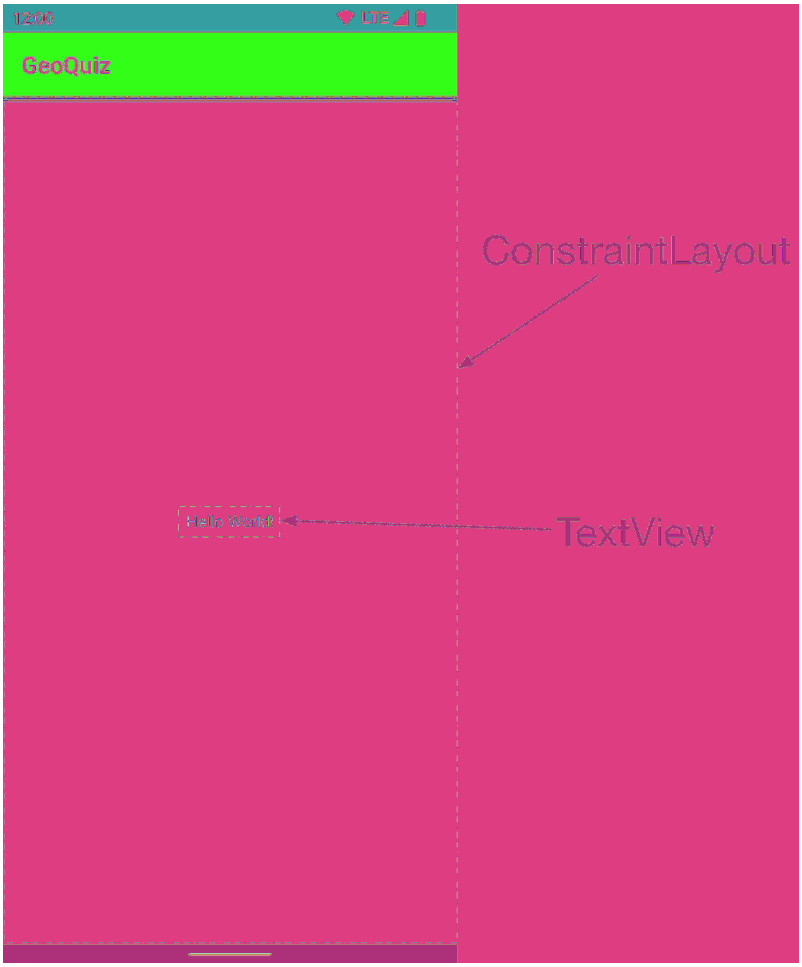
The Android SDK includes many views that you can configure to get the appearance and behavior you want. Each is an instance of the **`View`** class or one of its subclasses (such as **`TextView`** or **`Button`**).

Something has to tell the views where they belong onscreen. A **`ViewGroup`** is a kind of **`View`** that contains and arranges other views. A **`ViewGroup`** does not display content itself. Rather, it orchestrates where other views' content is displayed. **`ViewGroup`**s are often referred to as "layouts."

In the default activity layout, **`ConstraintLayout`** is the **`ViewGroup`** responsible for laying out its sole child, a **`TextView`**. You will learn more about layouts and views and about using **`ConstraintLayout`** in Chapter 11.

Figure 1.9 shows how the **`ConstraintLayout`** and **`TextView`** defined in the default XML would appear onscreen.

Figure 1.9 Default views as seen onscreen

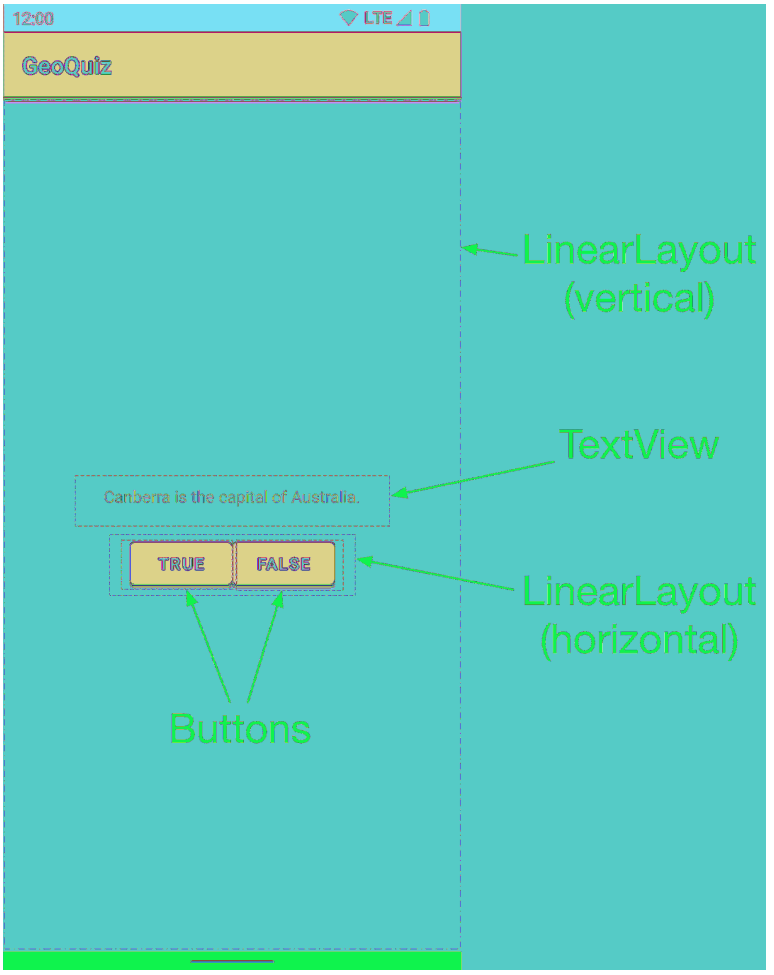



But these are not the views you are looking for. The interface for **`MainActivity`** requires five views:

- a vertical **`LinearLayout`**
- a **`TextView`**
- a horizontal **`LinearLayout`**
- two **`Button`**s

Figure 1.10 shows how these views compose **`MainActivity`**'s interface.

Figure 1.10 Planned views as seen onscreen




Now you need to define these views in your layout XML. Edit the text contents of `activity_main.xml` to match Listing 1.1. The XML that you need to delete is struck through, and the XML that you need to add is in bold font. This is the pattern we will use throughout this book.

Do not worry about understanding what you are typing; you will learn how it works next. However, do be careful. Layout XML is not validated, and typos will cause problems sooner or later.

You will see that the three `android:text` values are red, indicating that there is a problem with them. Ignore these errors for now; you will fix them soon.

Listing 1.1 Defining views in XML (`res/layout/activity_main.xml`)

```
<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>

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:gravity="center"
    android:orientation="vertical">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:padding="24dp"
        android:text="@string/question_text" />

    <LinearLayout
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:orientation="horizontal">

        <Button
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/true_button" />

        <Button
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/false_button" />

    </LinearLayout>

</LinearLayout>
```

Compare your XML with the UI shown in Figure 1.10. Every view has a corresponding XML element, and the name of the element is the type of the view.

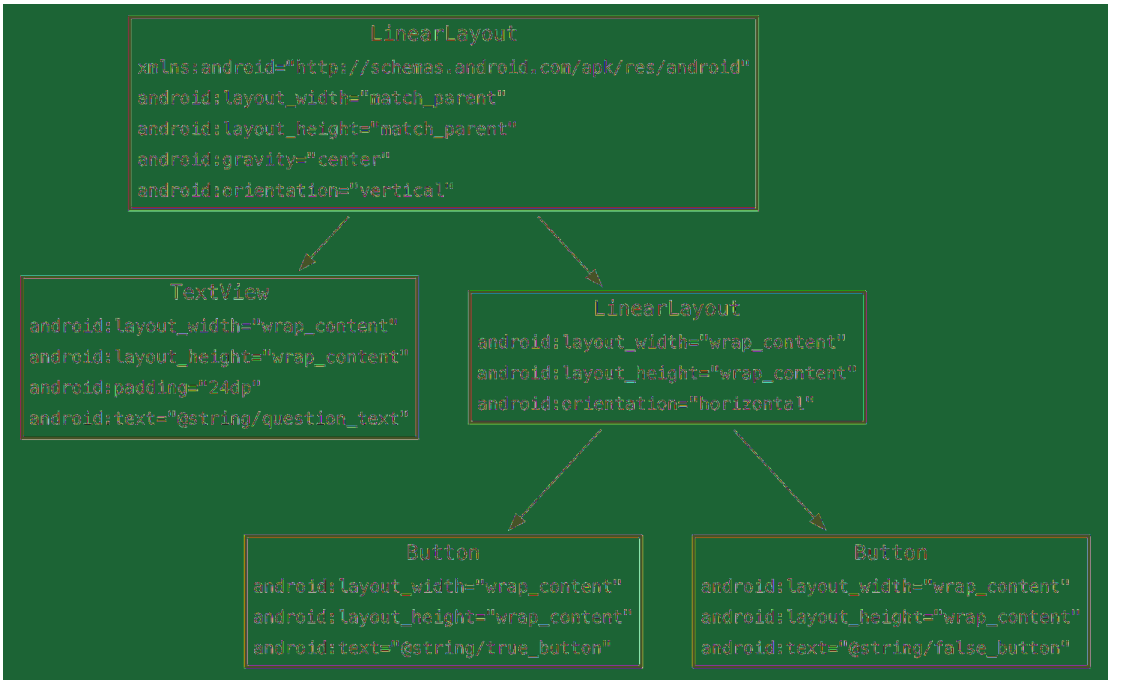
Each element has a set of XML *attributes*. Each attribute is an instruction about how the view should be configured.

To understand how the elements and attributes work, it helps to look at the layout from a hierarchical perspective.

## The view hierarchy

Your views exist in a hierarchy of **`View`** objects called the *view hierarchy*. Figure 1.11 shows the view hierarchy that corresponds to the XML in Listing 1.1.

Figure 1.11 Hierarchical layout of views and attributes




The root element of this layout's view hierarchy is a **`LinearLayout`**. As the root element, the **`LinearLayout`** must specify the Android resource XML namespace at `http://schemas.android.com/apk/res/android`.

**`LinearLayout`** inherits from **`ViewGroup`**, which, as we said earlier, is a subclass of **`View`** that contains and arranges other views. You use a **`LinearLayout`** when you want views arranged in a single column or row. Other **`ViewGroup`** subclasses that you will meet later include **`ConstraintLayout`** and **`FrameLayout`**.

When a view is contained by a **`ViewGroup`**, that view is said to be a *child* of the **`ViewGroup`**. The root **`LinearLayout`** has two children: a **`TextView`** and another **`LinearLayout`**. The child **`LinearLayout`** has two **`Button`** children of its own.

## View attributes

Let's go over some of the attributes you used to configure your views.

### android:layout_width and android:layout_height

The `android:layout_width` and `android:layout_height` attributes are required for almost every type of view. The most common values for these attributes are either `match_parent` or `wrap_content`. They behave the following ways:

`match_parent` view will be as big as its parent

`wrap_content` view will be as big as its contents require

For the root **`LinearLayout`**, the value of both the height and width attributes is `match_parent`. The **`LinearLayout`** is the root element, but it still has a parent – the view that Android provides for your app's view hierarchy to live in.

The other views in your layout have their widths and heights set to `wrap_content`. You can see in Figure 1.10 how this determines their sizes.

The **`TextView`** is slightly larger than the text it contains due to its `android:padding="24dp"` attribute. This attribute tells the view to add the specified amount of space to its contents when determining its size. You are using it to get a little breathing room between the question and the buttons. (Wondering about the `dp` units? These are density-independent pixels, which you will learn about in Chapter 11.)

### android:orientation

The `android:orientation` attribute on the two **`LinearLayout`** views determines whether their children will appear vertically or horizontally. The root **`LinearLayout`** is vertical; its child **`LinearLayout`** is horizontal.

The order in which children are defined determines the order they appear onscreen. In a vertical **`LinearLayout`**, the first child defined will appear topmost. In a horizontal **`LinearLayout`**, the first child defined will be leftmost. (Unless the device is set to a language that runs right to left, such as Arabic or Hebrew. In that case, the first child will be rightmost.)

### android:text

The **`TextView`** and **`Button`** views have `android:text` attributes. This attribute tells the view what text to display.

Notice that the values of these attributes are not literal strings. They are references to *string resources*, as denoted by the `@string/` syntax.

A string resource is a string that lives in a separate XML file called a *strings file*. You can give a view a hardcoded string, like `android:text="True"`, but it is usually not a good idea. Placing strings into a separate file and then referencing them is better because it makes localization (which you will learn about in Chapter 18) easy.

The string resources you are referencing in `activity_main.xml` do not exist yet, which is why you have errors in your project. Let's fix that.

## Creating string resources

Every project includes a default strings file named `res/values/strings.xml`. Expand `res/values` in the project tool window to locate this file, then double-click its name to open it.

The template has already added one string resource for you. Add the three new strings that your layout requires.

### Listing 1.2 Adding string resources (`res/values/strings.xml`)

```
<resources>
    <string name="app_name">GeoQuiz</string>
    <string name="question_text">Canberra is the capital of Australia.</string>
    <string name="true_button">True</string>
    <string name="false_button">False</string>
</resources>
```

(Depending on your version of Android Studio, you may have additional strings. Do not delete them. Deleting them could cause cascading errors in other files.)

Now, when you refer to `@string/false_button` in any XML file in the GeoQuiz project, you will get the literal string “False” at runtime.

The errors in `activity_main.xml` about the missing string resources should now be gone. (If you still have errors, check both files for typos.)

Although the default strings file is named `strings.xml`, you can name a strings file anything you want. You can also have multiple strings files in a project. As long as the file is located in `res/values/`, has a `resources` root element, and contains child `string` elements, your strings will be found and used.

## Previewing the layout

Your layout is now complete. Switch back to `activity_main.xml` and preview the layout in the design view by clicking the Design tab near the top-right corner of the editor (Figure 1.12).

Figure 1.12 Previewing `activity_main.xml` in the design view

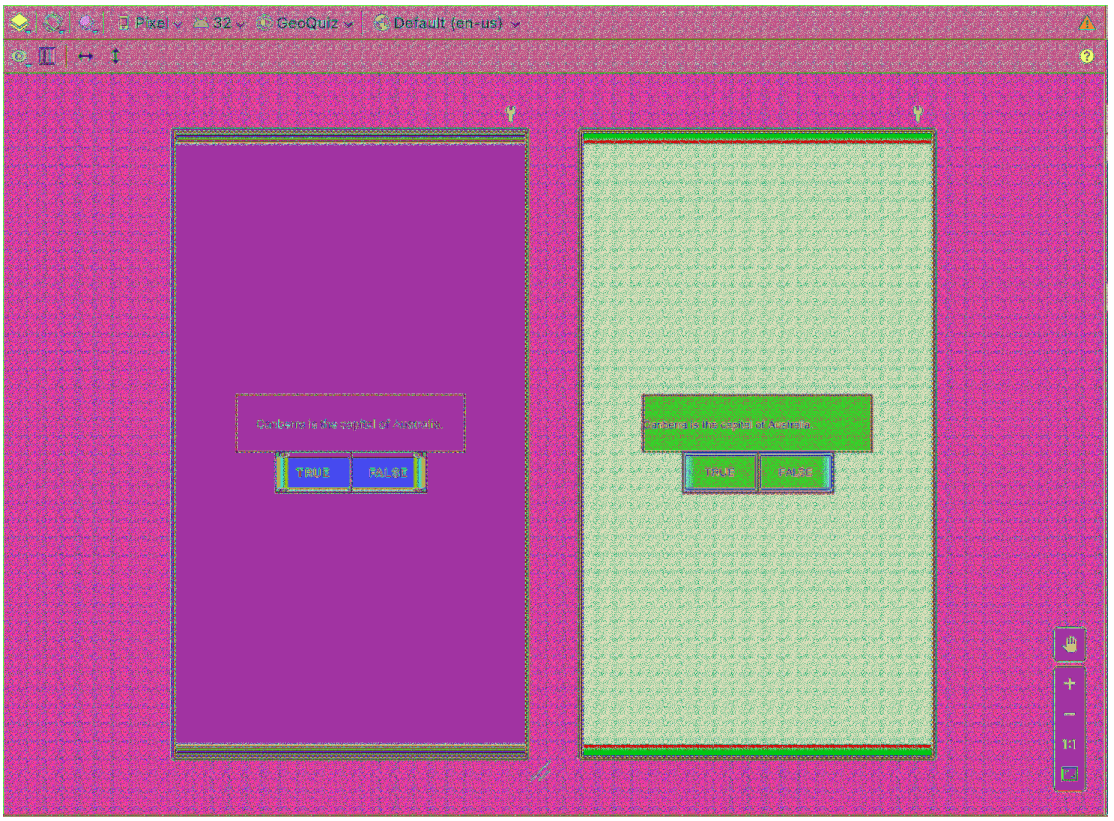



Figure 1.12 shows the two kinds of preview available. You can select from the preview types using a menu that drops down from the blue diamond button leftmost in the top toolbar. You can show either kind of preview individually or both together, as shown here.

The preview on the left is the *Design* preview. This shows how the layout would look on a device, including theming.

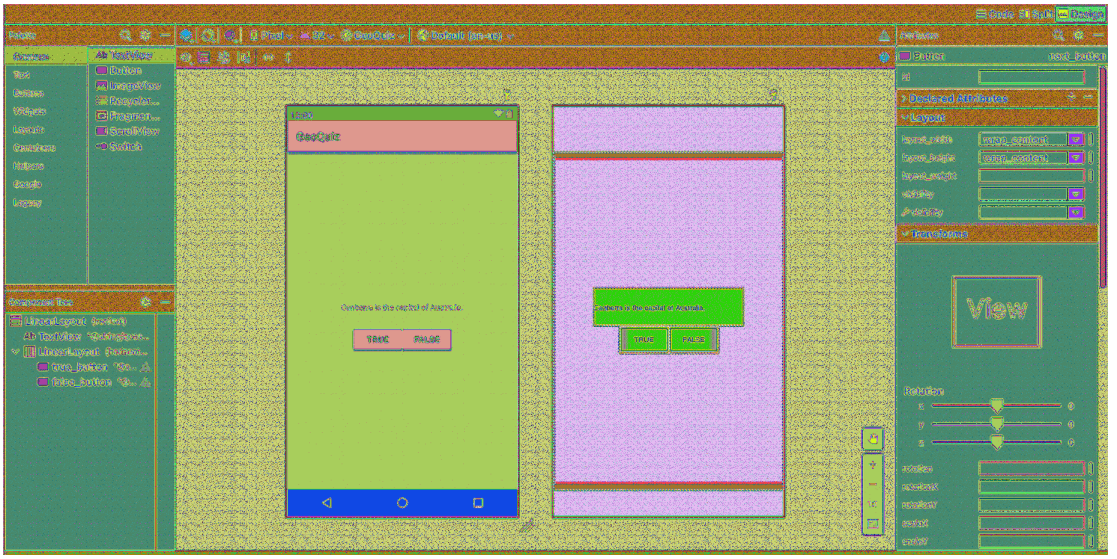
The preview on the right is the *Blueprint* preview. This preview focuses on the size of views and the relationships between them.

The design view also allows you to see how your layout looks on different device configurations. At the top of the preview window, you can specify the type of device, the version of Android to simulate, the device theme, and the locale to use when rendering your layout. You can even rotate the preview to see how a layout looks in landscape or pretend your current locale uses right-to-left text.

In addition to previewing, you can also build your layouts using the *layout editor*. In the top left of the design view, there is a palette that contains all the built-in views (Figure 1.13). You can drag these views from the palette and drop them into your view. You can also drop them into the component tree in the bottom left to have more control over where the view is placed.

Figure 1.13 shows the preview with *layout decorations* – the device status bar, app bar with GeoQuiz label, and virtual device button bar. To see these decorations, click the eye-shaped button in the toolbar just above the preview and select Show System UI.

Figure 1.13 Layout editor




You will find this graphical editor especially valuable when working with **`ConstraintLayout`**, as you will see in Chapter 11.

## From Layout XML to View Objects

How do XML elements in `activity_main.xml` become **`View`** objects? The answer starts in the **`MainActivity`** class.

When you created the GeoQuiz project, a subclass of **`Activity`** named **`MainActivity`** was created for you. The class file for **`MainActivity`** is in the `app/java` directory of your project.

A quick aside about the directory name before we get into how layouts become views: This directory is called `java` because Android originally supported only Java code. In your project, because you configured it to use Kotlin (and Kotlin is fully interoperable with Java), the `java` directory is where the Kotlin code lives. You could create a `kotlin` directory and place your Kotlin files there, but that would provide no real benefits and requires additional configuration, so most developers just place their Kotlin files in the `java` directory.

Return to the `MainActivity.kt` file and take a look at its contents:

```
package com.bignerdranch.android.geoquiz

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)
    }
}
```

(Wondering what **`AppCompatActivity`** is? It is a subclass of Android's **`Activity`** class that provides compatibility support for older versions of Android. You will learn much more about **`AppCompatActivity`** in Chapter 15.)

If you are not seeing all of the import statements, click the ... next to the word `import` to reveal them.

This file has one **`Activity`** function: **`onCreate(Bundle?)`**.

The **`onCreate(Bundle?)`** function is called when an instance of the activity subclass is created. When an activity is created, it needs a UI to manage. To give the activity its UI, you call **`Activity.setContentView(layoutResID: Int)`**.

This function *inflates* a layout and puts it onscreen. When a layout is inflated, each view in the layout file is instantiated as defined by its attributes. You specify which layout to inflate by passing in the layout's *resource ID*.

## Resources and resource IDs

A layout is a *resource*. A resource is a piece of your application that is not code – things like image files, audio files, and XML files.

Resources for your project live in a subdirectory of the `app/res` directory. In the project tool window, you can see that `activity_main.xml` lives in `res/layout/`. Your strings file, which contains string resources, lives in `res/values/`.

To access a resource in code, you use its resource ID. The resource ID for your layout is `R.layout.activity_main`.

You did not define `R.layout.activity_main` yourself. As a part of compiling your app, the build process automatically generates that resource ID for you. In fact, the build process generates resource IDs for all the resources in your project.

During the compilation and packaging of your app, the build tools generate a class known as the **`R`** class. This class contains a long list of IDs as integer constants, allowing you to access and use your resources in your application. When referencing `R.layout.activity_main`, you are actually referencing an integer constant named `activity_main` within the **`layout`** inner class of **`R`**.

As you add, remove, and change resources, the build process will automatically update the file in order to maintain the correct mapping between resources and their IDs. Here is an example of what your `R` class looks like in Java:

```
package com.bignerdranch.android.geoquiz;

public final class R {
    public static final class anim {
        ...
    }
    ...
    public static final class id {
        ...
    }
    public static final class layout {
        ...
        public static final Int activity_main=0x7f030017;
    }
    public static final class mipmap {
        public static final Int ic_launcher=0x7f030000;
    }
    public static final class string {
        ...
        public static final Int app_name=0x7f0a0010;
        public static final Int false_button=0x7f0a0012;
        public static final Int question_text=0x7f0a0014;
        public static final Int true_button=0x7f0a0015;
    }
}
```

Your strings also have resource IDs. You have not yet referred to a string in code, but if you did, it would look like this:

```
setTitle(R.string.app_name)
```

Android generated a resource ID for the entire layout and for each string, but it did not generate resource IDs for the individual views in `activity_main.xml`. Not every view needs a resource ID. In this chapter, you will only interact with the two buttons in code, so only they need resource IDs.

To generate a resource ID for a view, you include an `android:id` attribute in the view's definition. In `activity_main.xml`, add an `android:id` attribute to each button. (You will need to switch to the Code tab to do this.)

Listing 1.3 Adding IDs to **`Button`**s (`res/layout/activity_main.xml`)

```
<LinearLayout ... >

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:padding="24dp"
        android:text="@string/question_text" />

    <LinearLayout
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:orientation="horizontal">

        <Button
            android:id="@+id/true_button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/true_button" />

        <Button
            android:id="@+id/false_button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/false_button" />

    </LinearLayout>

</LinearLayout>
```

Notice that there is a + sign in the values for `android:id` but not in the values for `android:text`. This is because you are *creating* the resource IDs and only *referencing* the strings.

## Wiring Up Views

You are ready to wire up your button views. This is a two-step process:

- get references to the inflated **`View`** objects
- set listeners on those objects to respond to user actions

## Getting references to views

Now that the buttons have resource IDs, you can access them in **`MainActivity`**. Type the following code into `MainActivity.kt` (Listing 1.4). (Do not use code completion; type it in yourself.) After you enter each line, it will report an error.

### Listing 1.4 Accessing view objects by ID (`MainActivity.kt`)

```
class MainActivity : AppCompatActivity() {

    private lateinit var trueButton: Button
    private lateinit var falseButton: Button

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

        trueButton = findViewById(R.id.true_button)
        falseButton = findViewById(R.id.false_button)
    }
}
```

In an activity, you can get a reference to an inflated view by calling **`Activity.findViewById(Int)`**. This function returns the corresponding view. Specifically, rather than return it as a **`View`**, it casts the view to the expected subtype of **`View`**. Here, that type is **`Button`**.

In the code above, you use the resource IDs of your buttons to retrieve the inflated objects and assign them to your view properties. Since the view objects are not inflated into and available in memory until after **`setContentView(…)`** is called in **`onCreate(…)`**, you use `lateinit` on your property declarations to indicate to the compiler that you will provide a non-null **`View`** value before you attempt to use the contents of the property.

Then, in **`onCreate(…)`**, you look up and assign the view objects the appropriate properties. You will learn more about **`onCreate(…)`** and the activity lifecycle in Chapter 3.

Now let's get rid of those pesky errors. Mouse over one of the red **`Button`** type declarations. It reports: Unresolved reference: Button.

This error is telling you that you need to import the **`android.widget.Button`** class into `MainActivity.kt`. You could type the following import statement at the top of the file:

```
import android.widget.Button
```

Or you can do it the easy way and let Android Studio do it for you. Just press Option-Return (or Alt-Enter) to let the IntelliJ magic under the hood amaze you. The new import statement now appears with the others at the top of the file. This shortcut is generally useful when something is not correct with your code. Try it often!

This should get rid of all the errors, because the errors for **`findViewById`** had to do with not being able to locate the button instances. (If you still have errors, check for typos in your code and XML.) Once your code is error free, it is time to make your app interactive.

## Setting listeners

Android applications are typically *event driven*. Unlike command-line programs or scripts, event-driven applications start and then wait for an event, such as the user pressing a button. (Events can also be initiated by the OS or another application, but user-initiated events are the most obvious.)

When your application is waiting for a specific event, we say that it is "listening for" that event. The object that you create to respond to an event is called a *listener*, and the listener implements a *listener interface* for that event.

The Android SDK comes with listener interfaces for various events, so you do not have to write your own. In this case, the event you want to listen for is a button being pressed (or "clicked"), so your listener will implement the **`View.OnClickListener`** interface.

Start with the TRUE button. In `MainActivity.kt`, set a listener on the button in **`onCreate(Bundle?)`**, just after the variable assignments.

### Listing 1.5 Setting a listener for the TRUE button (`MainActivity.kt`)

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

    trueButton = findViewById(R.id.true_button)
    falseButton = findViewById(R.id.false_button)

    trueButton.setOnClickListener { view: View ->
        // Do something in response to the click here
    }
}
```

(If you have an Unresolved reference: View error, try using Option-Return [Alt-Enter] to import the **`View`** class.)

In Listing 1.5, you set a listener to inform you when the **`Button`** known as `trueButton` has been pressed. The Android framework defines **`View.OnClickListener`** as a Java interface with a single method, **`onClick(View)`**. Interfaces with a *single abstract method* are common enough in Java that the pattern has a pet name, *SAM*.

Kotlin has special support for this pattern as part of its Java interoperability layer. It lets you write a function literal, and it takes care of turning that into an object implementing the interface. This behind-the-scenes process is called *SAM conversion*.

Your on-click listener is implemented using a lambda expression. Set a similar listener for the FALSE button.

### Listing 1.6 Setting a listener for the FALSE button (`MainActivity.kt`)

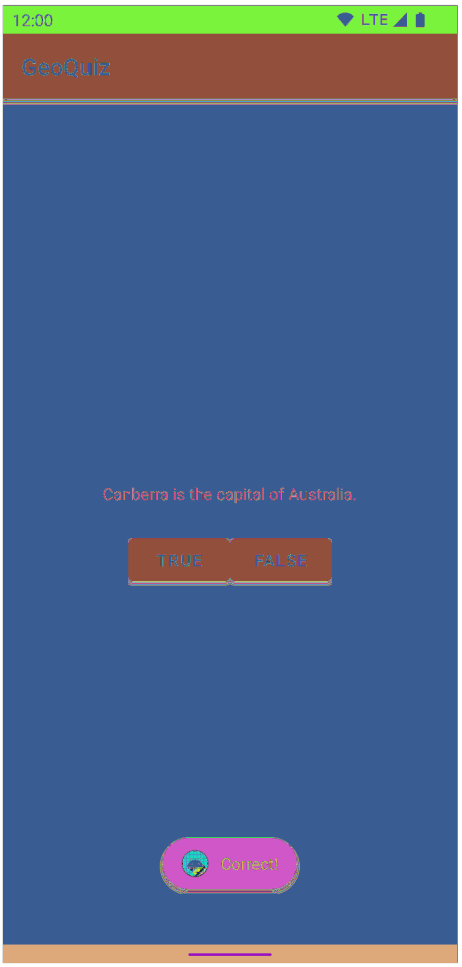
```
override fun onCreate(savedInstanceState: Bundle?) {
    ...
    trueButton.setOnClickListener { view: View ->
        // Do something in response to the click here
    }

    falseButton.setOnClickListener { view: View ->
        // Do something in response to the click here
    }
}
```

## Making Toasts

Now to make the buttons fully armed and operational. You are going to have a press of each button trigger a pop-up message called a *toast*. A toast is a short message that informs the user of something but does not require any input or action. You are going to make toasts that announce whether the user answered correctly or incorrectly (Figure 1.14).

Figure 1.14 A toast providing feedback




First, return to `strings.xml` and add the string resources that your toasts will display.

Listing 1.7 Adding toast strings (`res/values/strings.xml`)

```
<resources>
    <string name="app_name">GeoQuiz</string>
    <string name="question_text">Canberra is the capital of Australia.</string>
    <string name="true_button">True</string>
    <string name="false_button">False</string>
    <string name="correct_toast">Correct!</string>
    <string name="incorrect_toast">Incorrect!</string>
</resources>
```

Next, update your click listeners to create and show a toast. Use code completion to help you fill in the listener code. Code completion can save you a lot of time, so it is good to become familiar with it early.

Start typing the code shown in Listing 1.8 in `MainActivity.kt`. When you get to the period after the **`Toast`** class, a pop-up window will appear with a list of suggested functions and constants from the **`Toast`** class.

To choose one of the suggestions, use the up and down arrow keys to select it. (If you wanted to ignore code completion, you could just keep typing. It will not complete anything for you if you do not press the Tab key, press the Return key, or click the pop-up window.)

From the list of suggestions, select **`makeText(context: Context, resId: Int, duration: Int)`**. Code completion will add the function call for you.

Fill in the parameters for the **`makeText(…)`** function until you have added the code shown in Listing 1.8.

### Listing 1.8 Making toasts (`MainActivity.kt`)

```
override fun onCreate(savedInstanceState: Bundle?) {
    ...
    trueButton.setOnClickListener { view: View ->
        // Do something in response to the click here
        Toast.makeText(
            this,
            R.string.correct_toast,
            Toast.LENGTH_SHORT
        ).show()
    }

    falseButton.setOnClickListener { view: View ->
        // Do something in response to the click here
        Toast.makeText(
            this,
            R.string.incorrect_toast,
            Toast.LENGTH_SHORT
        ).show()
    }
}
```

To create a toast, you call the static function **`Toast.makeText(Context, Int, Int)`**. This function creates and configures a **`Toast`** object. The **`Context`** parameter is typically an instance of **`Activity`** (and **`Activity`** is a subclass of **`Context`**). Here you pass the instance of **`MainActivity`** as the **`Context`** argument.

The second parameter is the resource ID of the string that the toast should display. The **`Context`** is needed by the **`Toast`** class to be able to find and use the string's resource ID. The third parameter is one of two **`Toast`** constants that specify how long the toast should be visible.

After you have created a toast, you call **`Toast.show()`** on it to get it onscreen.

Because you used code completion, you do not have to do anything to import the **`Toast`** class. When you accept a code completion suggestion, the necessary classes are imported automatically.

Now, let's see your app in action.

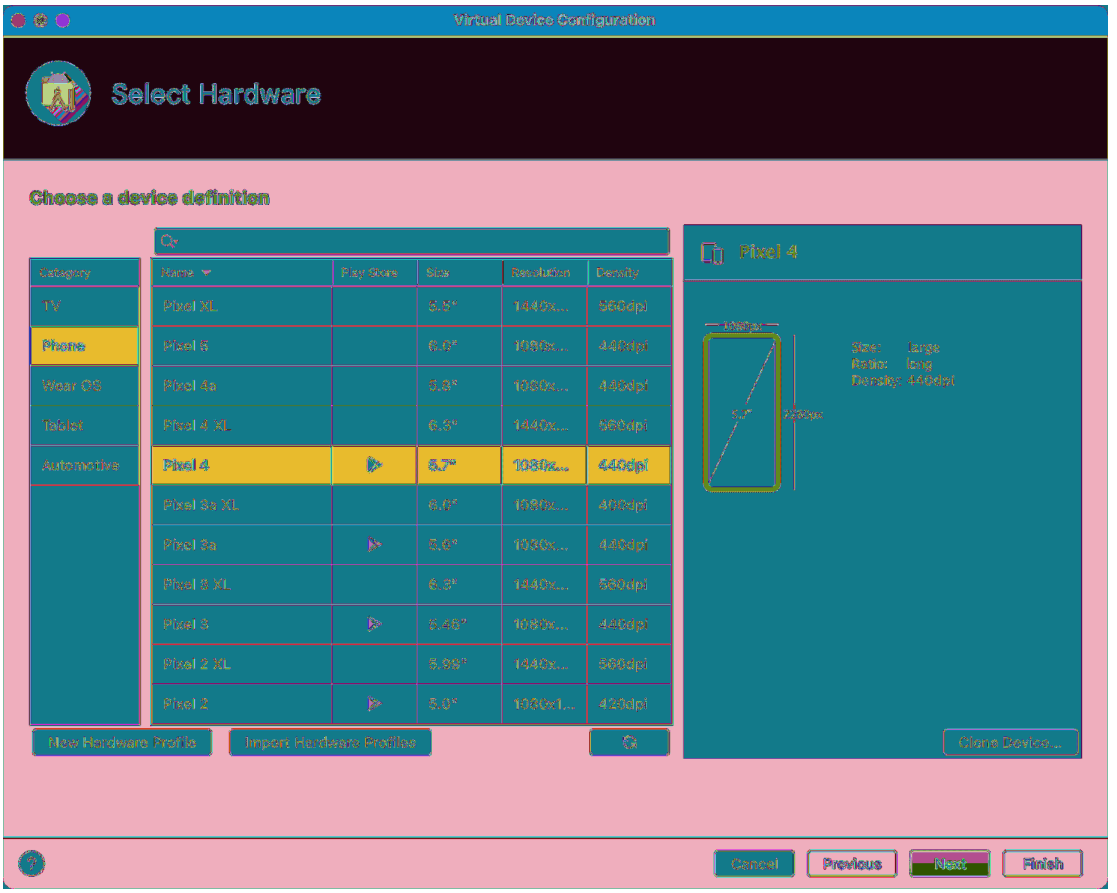
## Running on the Emulator

To run an Android application, you need a device – either a hardware device or a *virtual device*. Virtual devices are powered by the Android emulator, which ships with the developer tools.

To create an Android virtual device (or AVD), choose Tools → AVD Manager. When the AVD Manager appears, click the + Create Virtual Device... button in the middle of the window.

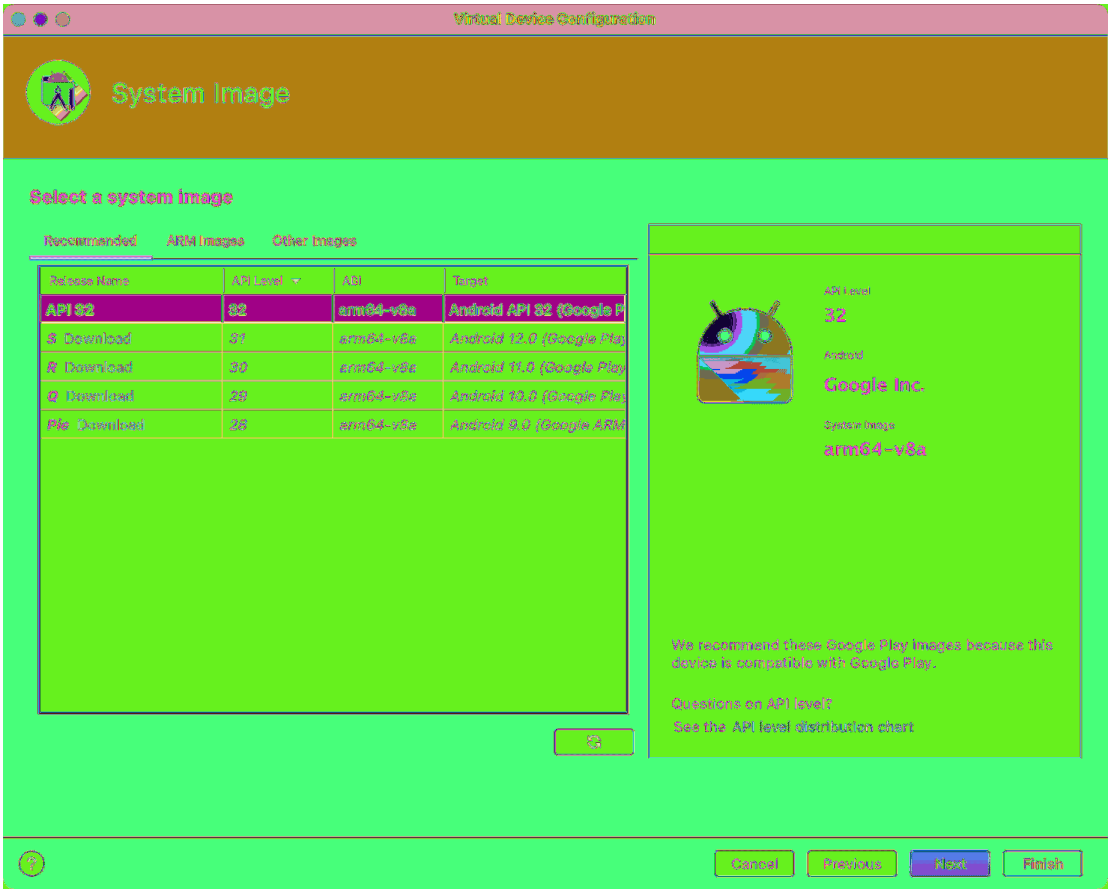
In the dialog that appears, you are offered many options for configuring a virtual device. For your first AVD, choose to emulate a Pixel 4, as shown in Figure 1.15. Click Next.

Figure 1.15 Choosing a virtual device



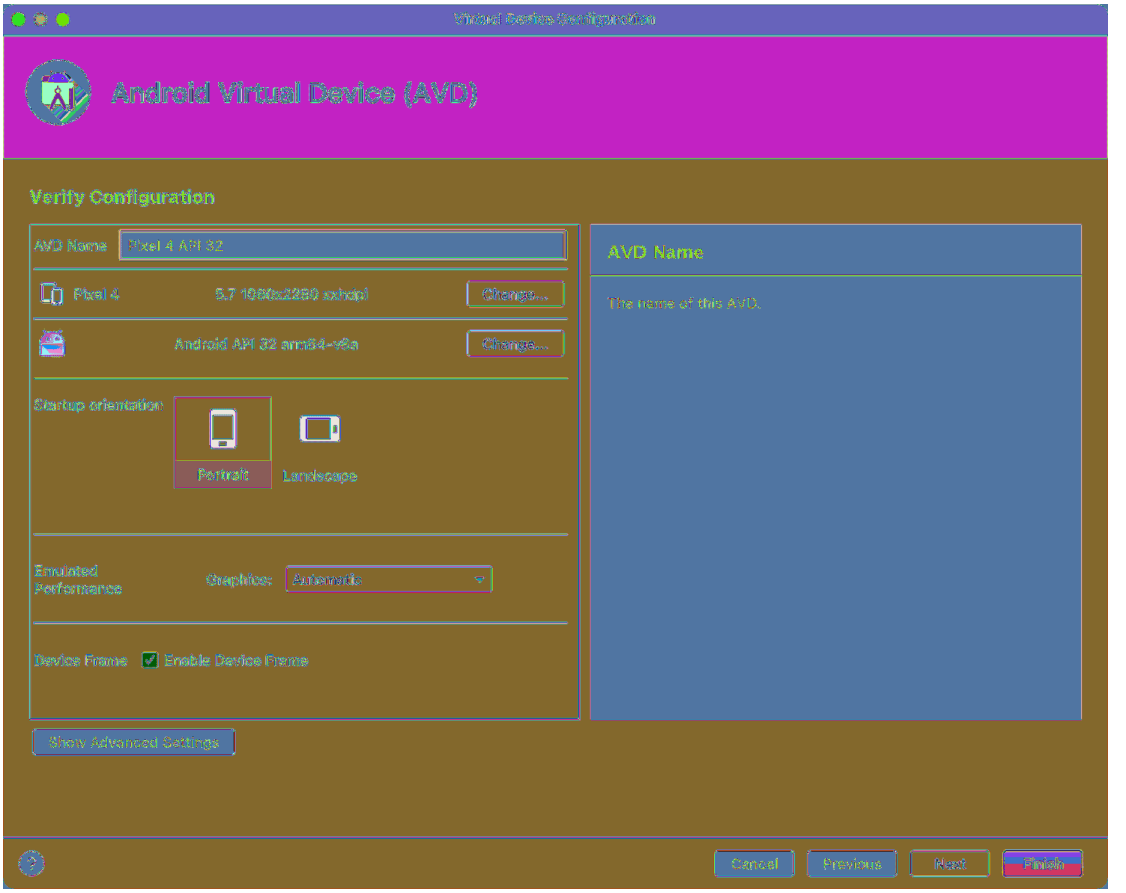
On the next screen, choose a system image for your emulator. For this emulator, select an API 32 emulator from the Recommended tab and select Next (Figure 1.16). (You may need to follow the steps to download the emulator's components before you can click Next.)

Figure 1.16 Choosing a system image



Finally, you can review and tweak properties of the emulator. You can also edit the properties of an existing emulator later. For now, accept the default name, which includes the device type and the API, and click Finish (Figure 1.17).

Figure 1.17 Updating emulator properties

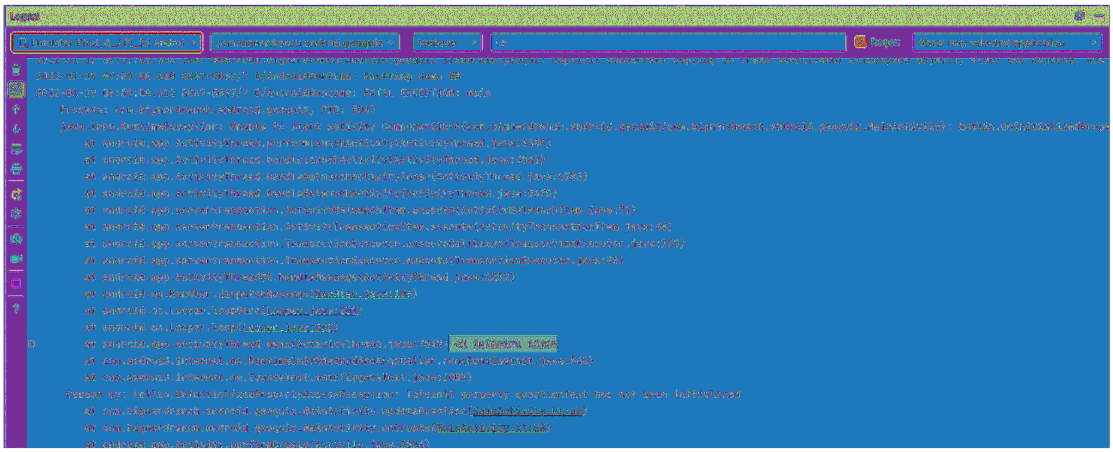



Once you have an AVD, you can run GeoQuiz on it. From the Android Studio toolbar, click the run button (it looks like a green “play” symbol) or press Control-R (Ctrl-R). Android Studio will start your virtual device, install the application package on it, and run the app.

Starting up the emulator can take a while, but eventually your GeoQuiz app will launch on the AVD that you created. Press buttons and admire your toasts.

If GeoQuiz crashes when launching or when you press a button, useful information will appear in the Logcat tool window. (If Logcat did not open automatically when you ran GeoQuiz, you can open it by clicking the Logcat button at the bottom of the Android Studio window.) Type MainActivity into the search box at the top of the Logcat tool window to filter the log messages. Look for exceptions in the log; they will be an eye-catching red color (Figure 1.18).

Figure 1.18 An example **`UninitializedPropertyAccessException`**



Compare your code to the code in the book to try to find the cause of the problem. Then try running again. (You will learn more about using Logcat in Chapter 3 and about debugging in Chapter 5.)

Keep the emulator running – you do not want to wait for it to launch on every run.

You can stop the app by pressing the Back button at the bottom of the emulator. (The Back button is shaped like a left-pointing triangle.) Then rerun the app from Android Studio to test changes.

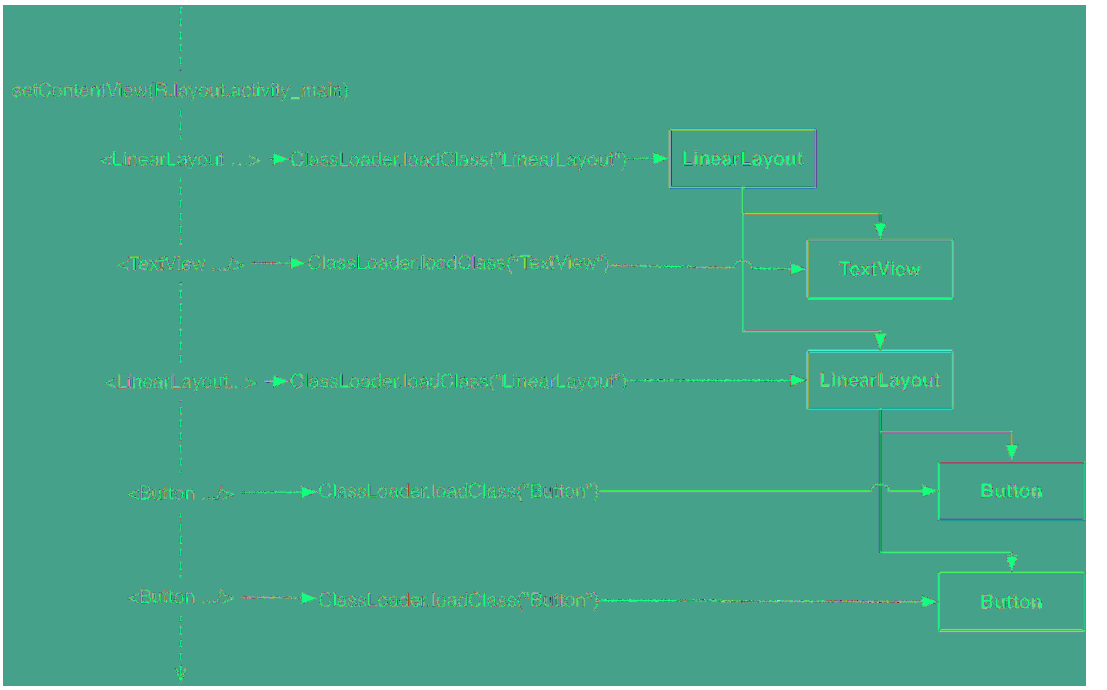
The emulator is useful, but testing on a real device gives more accurate results. In Chapter 2, you will run GeoQuiz on a hardware device. You will also give GeoQuiz more geography questions with which to test the user.

## For the More Curious: The Android Build Process

By now, you probably have some burning questions about how the Android build process works. You have already seen that Android Studio builds your project automatically as you modify it, rather than on command. During the build process, the Android tools take your resources, code, and `AndroidManifest.xml` file (which contains metadata about the application) and turn them into an `.apk` file. This file is then signed with a debug key, which allows it to run on the emulator. (To distribute your `.apk` to the masses, you have to sign it with a release key. There is more information about this process in the Android developer documentation at `developer.android.com/tools/publishing/preparing.html`.)

How do the contents of `activity_main.xml` turn into **`View`** objects in an application? As part of the build process, aapt2 (the Android Asset Packaging Tool) compiles layout file resources into a more compact format. These compiled resources are packaged into the `.apk` file. Then, when **`setContentView(…)`** is called in **`MainActivity`**'s **`onCreate(Bundle?)`** function, **`MainActivity`** uses the **`LayoutInflater`** class to instantiate each of the **`View`** objects as defined in the layout file (Figure 1.19).

Figure 1.19 Inflating `activity_main.xml`




(You can also create your view classes programmatically in the activity instead of defining them in XML. But there are benefits to separating your presentation from the logic of the application. The main one is taking advantage of configuration changes built into the SDK, which you will learn more about in Chapter 3.)

You will learn more details of how the different XML attributes work and how views display themselves on the screen in Chapter 11.

## Android build tools

All the builds you have seen so far have been executed from within Android Studio. This build is integrated into the IDE – it invokes standard Android build tools like `aapt2`, but the build process itself is managed by Android Studio.

You may, for your own reasons, want to perform builds from outside of Android Studio. The easiest way to do this is to use a command-line build tool. The Android build system uses a tool called Gradle.

(You will know if this section applies to you. If it does not, feel free to read along but do not be concerned if you are not sure why you might want to do this or if the commands below do not seem to work. Coverage of the ins and outs of using the command line is beyond the scope of this book.)

To use Gradle from the command line, navigate to your project's directory and run the following command:

```
$ ./gradlew tasks
```

On Windows, your command will look a little different:

```
> gradlew.bat tasks
```

This will show you a list of available tasks you can execute. The one you want is called `installDebug`. Make it so with a command like this:

```
$ ./gradlew installDebug
```

Or, on Windows:

```
> gradlew.bat installDebug
```

This will install your app on whatever device is connected. However, it will not run the app. For that, you will need to pull up the launcher and launch the app by hand.

## Challenges

Challenges are exercises at the end of the chapter for you to do on your own. Some are easy and provide practice doing the same thing you have done in the chapter. Other challenges are harder and require more problem-solving.

We cannot encourage you enough to take on these challenges. Tackling them cements what you have learned, builds confidence in your skills, and bridges the gap between us teaching you Android programming and you being able to do Android programming on your own.

If you get stuck while working on a challenge, take a break and come back to try again fresh. If that does not help, check out the forum for this book at `forums.bignerdranch.com`. In the forum, you can review questions and solutions that other readers have posted as well as ask questions and post solutions of your own.

To protect the integrity of your current project, we recommend you make a copy and work on challenges in the new copy.

In your computer's file explorer, navigate to the root directory of your project. Copy the GeoQuiz folder and paste a new copy next to the original (on macOS, use the Duplicate feature). Rename the new folder something like GeoQuiz Chapter1 Challenge. Back in Android Studio, select File → Open.... Navigate to your new challenge file and select OK, then New Window. The copied project will then appear in a new window ready for work.

## Challenge: Switching Your Toast for a Snackbar

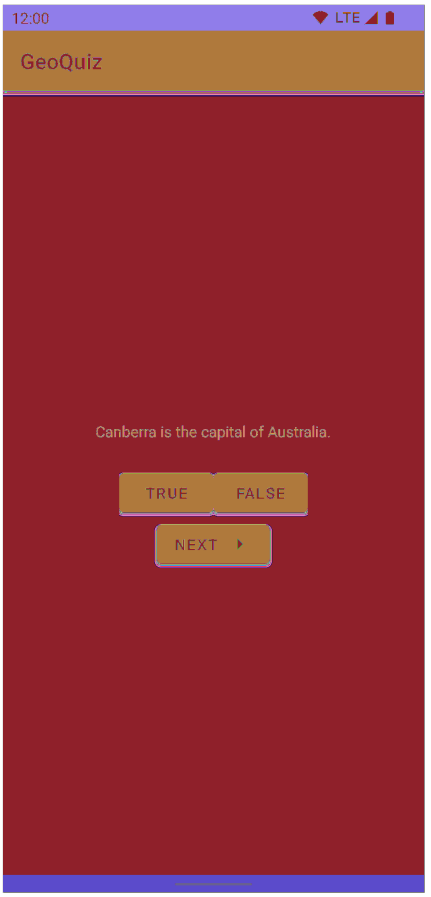
For this challenge, customize the toast by using a **`Snackbar`** instead of a **`Toast`**. While using a **`Toast`** is a convenient way to display UI, we recommend using a **`Snackbar`** in your applications because they are more configurable in both their appearance and behavior. Refer to the developer documentation at `developer.android.com/reference/com/google/android/material/snackbar/Snackbar` for more details. (Hint: Take a look at the **`make`** and **`show`** functions.)

# 2

# Interactive User Interfaces

In this chapter, you are going to upgrade GeoQuiz to present more than one question, as shown in Figure 2.1.

Figure 2.1 Next!




To make this happen, you are going to add a data class named **`Question`** to the GeoQuiz project. An instance of this class will encapsulate a single true/false question.

Then, you will create a collection of **`Question`** objects for **`MainActivity`** to manage.

## Creating a New Class

In the project tool window, right-click the `com.bignerdranch.android.geoquiz` package and select New → Kotlin Class/File. Enter Question for the name. Double-click Class in the list of options (Figure 2.2).

Figure 2.2 Creating the **`Question`** class




Android Studio will create and open a file called `Question.kt`. In this file, add two properties and a constructor.

Listing 2.1 Adding to the **`Question`** class (`Question.kt`)

```
~~class Question {~~
~~}~~
import androidx.annotation.StringRes

data class Question(@StringRes val textResId: Int, val answer: Boolean)
```

A common through-line of many software architecture patterns (such as Model-View-Controller, Model-View-Presenter, Model-View-ViewModel, and so on) is the concept of the *model*. In these architecture patterns, models are classes that contain information that the UI will display.

We recommend that you create these classes using the `data` keyword, as you did here. Doing so clearly indicates that the class is meant to hold model data. Also, the compiler does extra work for data classes that makes your life easier, such as defining useful functions like **`equals()`**, **`hashCode()`**, and a nicely formatted **`toString()`**.

The **`Question`** class holds two pieces of data: a resource ID for the question text and the question answer (true or false).

The `@StringRes` annotation is not required, but we recommend you include it for two reasons. First, the annotation helps the code inspector built into Android Studio (named Lint) verify at compile time that constructor calls provide a valid string resource ID. This prevents runtime crashes where the constructor is used with an invalid resource ID (such as an ID that points to some resource other than a string). Second, the annotation makes your code more readable for other developers.

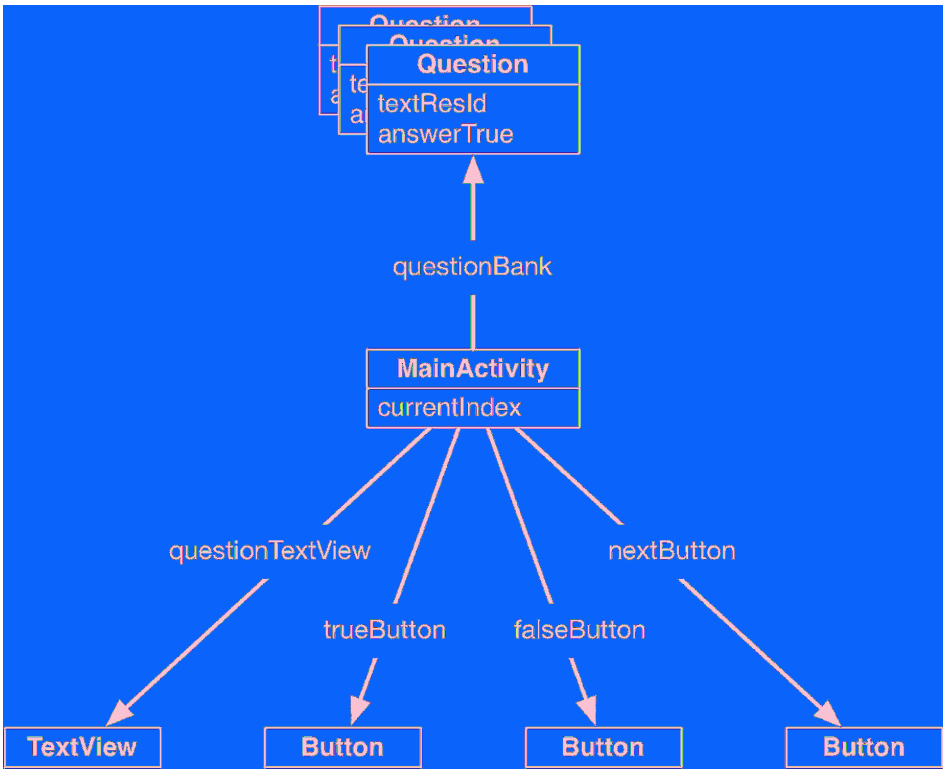
To use the `@StringRes` annotation, you need to import it into your project. You can do that by entering the import statement manually or with the Option-Return (Alt-Enter) keyboard shortcut you learned in the last chapter.

Why is `textResId` an **`Int`** and not a **`String`**? Even though you will eventually display text to the user, the `textResId` variable will hold the resource ID (always an **`Int`**) of the string resource for a question.

Your **`Question`** class is now complete. In a moment, you will modify **`MainActivity`** to work with **`Question`**. First, let's take a look at how the pieces of GeoQuiz will work together.

You are going to have **`MainActivity`** create a list of **`Question`** objects. It will then interact with the **`TextView`** and the three **`Button`**s to display questions and provide feedback. Figure 2.3 diagrams these relationships.

Figure 2.3 Object diagram for GeoQuiz

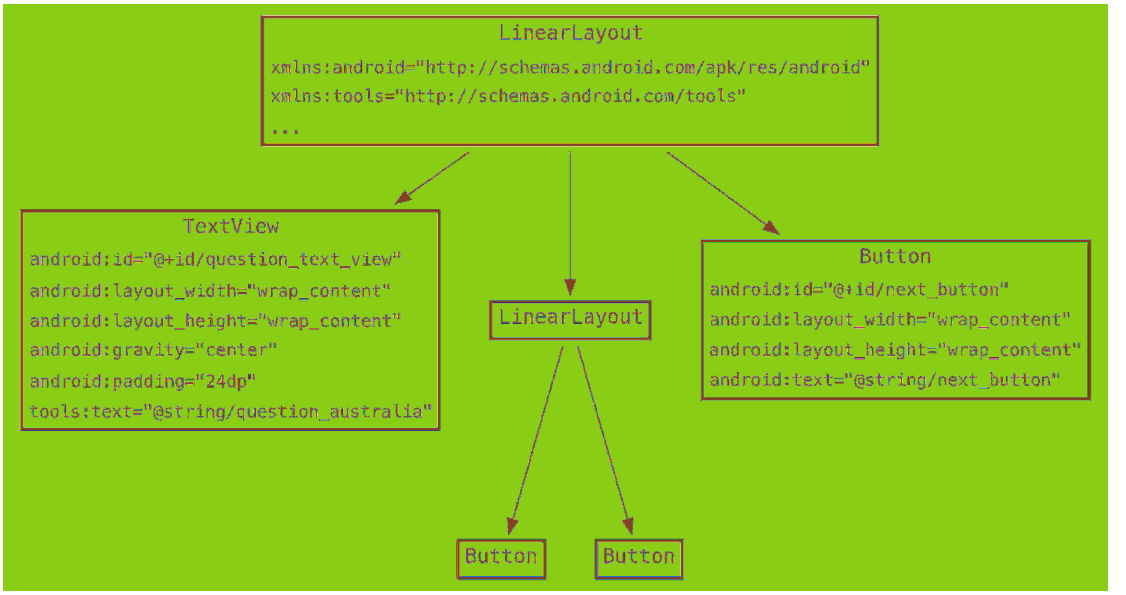


## Updating the Layout

Now that you have defined the `Question` class, you are going to update GeoQuiz's UI to include a NEXT button.

In Android, UI is typically inflated from XML within a layout file. The sole layout in GeoQuiz is defined in `activity_main.xml`. This layout needs to be updated as shown in Figure 2.4. (Note that to save space we are not showing the attributes of unchanged views.)

Figure 2.4 New button!



So the changes you need to make to the layout are:

- Give the **`TextView`** an `android:id` attribute. This view will need a resource ID so that you can set its text in **`MainActivity`**'s code.
- Position the **`TextView`**'s text in the center of the text view by setting `gravity` to `"center"`.
- Replace the `android:text` attribute of the **`TextView`** with the `tools:text` attribute and point it to a string resource representing a question using `@string/`. You will also need to add the `tools` namespace to the root tag of your layout so that Android Studio can make sense of the `tools:text` attribute.

  You no longer want a hardcoded question to be part of the **`TextView`**'s definition. Instead, you will set the question text dynamically as the user clicks through the questions. However, if you only removed the `android:text` line, the layout preview in Android Studio would look like it is missing text. It is helpful to have the layout preview reflect what the end user will see on their device.

  The `tools` namespace allows you to override any attribute on a view for the purpose of displaying it in the Android Studio preview. The `tools` attributes are ignored when rendering the views on a device at runtime. You could also use `android:text` and just overwrite the value at runtime, but using `tools:text` instead makes it clear that the value you provide is for preview purposes only.
- Add the new **`Button`** view as a child of the root **`LinearLayout`**.

Return to `activity_main.xml` and make it happen.

### Listing 2.2 New button ... and changes to the text view (`res/layout/activity_main.xml`)

```
<LinearLayout 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"
        ... >

    <TextView
        android:id="@+id/question_text_view"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:gravity="center"
        android:padding="24dp"
        android:text="@string/question_text"
        tools:text="@string/question_australia" />

    <LinearLayout ... >
        ...
    </LinearLayout>

    <Button
        android:id="@+id/next_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/next_button" />

</LinearLayout>
```

You will see familiar errors alerting you about missing string resources.

Return to `res/values/strings.xml`. Rename `question_text` and add a string for the new button.

### Listing 2.3 Updating strings (`res/values/strings.xml`)

```
<string name="app_name">GeoQuiz</string>
<string name="question_text">Canberra is the capital of Australia.</string>
<string name="question_australia">Canberra is the capital of Australia.</string>
<string name="true_button">True</string>
<string name="false_button">False</string>
<string name="next_button">Next</string>
...
```

While you have `strings.xml` open, go ahead and add the strings for the rest of the geography questions that will be shown to the user.

### Listing 2.4 Adding question strings (`res/values/strings.xml`)

```
<string name="question_australia">Canberra is the capital of Australia.</string>
<string name="question_oceans">The Pacific Ocean is larger than
  the Atlantic Ocean.</string>
<string name="question_mideast">The Suez Canal connects the Red Sea
  and the Indian Ocean.</string>
<string name="question_africa">The source of the Nile River is in Egypt.</string>
<string name="question_americas">The Amazon River is the longest river
  in the Americas.</string>
<string name="question_asia">Lake Baikal is the world\'s oldest and deepest
  freshwater lake.</string>
...
```

Notice that you use the escape sequence `\'` in the last value to get an apostrophe in your string. You can use all the usual escape sequences in your string resources, such as `\n` for a new line.

Return to `activity_main.xml` and preview your layout changes in the graphical layout tool. It should look like Figure 2.1.

That is all for now for GeoQuiz's UI. Time to wire everything up in **`MainActivity`**.

## Wiring Up the User Interface

First, create a list of **`Question`** objects in **`MainActivity`**, along with an index for the list.

### Listing 2.5 Adding a **`Question`** list (`MainActivity.kt`)

```
class MainActivity : AppCompatActivity() {

    private lateinit var trueButton: Button
    private lateinit var falseButton: Button

    private val questionBank = listOf(
            Question(R.string.question_australia, true),
            Question(R.string.question_oceans, true),
            Question(R.string.question_mideast, false),
            Question(R.string.question_africa, false),
            Question(R.string.question_americas, true),
            Question(R.string.question_asia, true))

    private var currentIndex = 0
    ...
}
```

Here you call the **`Question`** constructor several times and create a list of **`Question`** objects.

(In a more complex project, this list would be created and stored elsewhere. In later apps, you will see better options for storing model data. For now, you are keeping it simple and just creating the list within **`MainActivity`**.)

You are going to use `questionBank`, `currentIndex`, and the properties in **`Question`** to get a parade of questions onscreen.

In the previous chapter, there was not much happening in GeoQuiz's **`MainActivity`**. It displayed the layout defined in `activity_main.xml`. Using `Activity.findViewById(id: Int)`, it got references to two buttons. Then, it set listeners on the buttons and wired them to make toasts.

Now that you have multiple questions to retrieve and display, **`MainActivity`** will have to work harder to respond to user input and update the UI. You could continue to use **`Activity.findViewById(…)`** and obtain references to your new views, but that is boring code that you would probably prefer not to write yourself.

Thankfully, there is *View Binding*, a feature of the build process that generates that boilerplate code for you and allows you to safely and easily interact with your UI elements. You will use View Binding to write less code and manage the complexity of even this relatively simple app.

Much like the **`R`** class (which you read about in Chapter 1), View Binding works by generating code during the build process for your app. However, View Binding is not enabled by default, so you must enable it yourself.

In the project tool window, under Gradle Scripts, locate and open the `build.gradle` file labeled (Module: GeoQuiz.app). (This file is actually located within the `app` module, but the Android view collects your project's Gradle files to make them easier to find.)

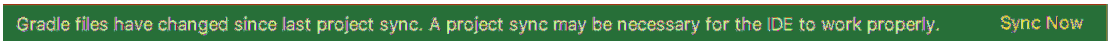
### Listing 2.6 Enabling View Binding (app/build.gradle)

```
plugins {
  id 'com.android.application'
  id 'kotlin-android'
}

android {
  ...
  kotlinOptions {
    jvmTarget = '1.8'
  }
  buildFeatures {
    viewBinding true
  }
}
...
```

After making this change, a banner will appear at the top of the file prompting you to sync the file (Figure 2.5).

### Figure 2.5 Gradle sync prompt




Whenever you make changes in a `.gradle` file, you must sync the changes so the build process for your app is up to date. Click Sync Now in the banner or select File → Sync Project with Gradle Files.

Now that View Binding is enabled, open `MainActivity.kt` and start using this build feature. Make the changes in Listing 2.7, and we will explain them afterward.

### Listing 2.7 Initializing **`ActivityMainBinding`** (MainActivity.kt)

```
  package com.bignerdranch.android.geoquiz

  import android.os.Bundle
  import android.view.View
  ...
  import com.bignerdranch.android.geoquiz.databinding.ActivityMainBinding

  class MainActivity : AppCompatActivity() {

      private lateinit var binding: ActivityMainBinding

      private lateinit var trueButton: Button
      private lateinit var falseButton: Button
      ...
      override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)
        ...
      }
      ...
  }
```

(Like the **R** class, View Binding generates code within your package structure, which is why the import statement includes your package name. If you gave your package a name other than `com.bignerdranch.android.geoquiz`, your import statement will look different.)

View Binding does require a little bit of setup within your **MainActivity**, so let's break down what is happening here.

Much like **Activity.findViewById(…)** allows you to get references to your individual UI elements, **ActivityMainBinding** allows you to get references to each UI element in your `activity_main.xml` layout. (View Binding generates classes based on the layout file's name; so, for example, it would generate an **ActivityCheatBinding** for a layout named `activity_cheat.xml`.)

In Chapter 1, you passed `R.layout.activity_main` into **Activity.setContentView(layoutResID: Int)** to display your UI. That function performed two actions: First, it inflated your `activity_main.xml` layout; then, it put the UI onscreen. Here, when you initialize `binding` you are obtaining a reference to the layout and inflating it in the same line.

You pass `layoutInflater`, a property inherited from the **Activity** class, into the **ActivityMainBinding.inflate()** call. As its name implies, `layoutInflater` is responsible for inflating your XML layouts into UI elements.

When you previously called **Activity.setContentView(layoutResID: Int)**, your **MainActivity** internally used its `layoutInflater` to display your UI. Now, using a different implementation of **setContentView()**, you pass a reference to the root UI element in your layout to display your UI.

Now that View Binding is set up within **MainActivity**, use it to wire up your UI. Start with the TRUE and FALSE buttons:

### Listing 2.8 Using **ActivityMainBinding** (`MainActivity.kt`)

```
class MainActivity : AppCompatActivity() {

    private lateinit var binding: ActivityMainBinding

    private lateinit var trueButton: Button
    private lateinit var falseButton: Button
    ...
    override fun onCreate(savedInstanceState: Bundle?) {
      super.onCreate(savedInstanceState)
      binding = ActivityMainBinding.inflate(layoutInflater)
      setContentView(binding.root)

      trueButton = findViewById(R.id.true_button)
      falseButton = findViewById(R.id.false_button)

      trueButton.setOnClickListener { view: View ->
      binding.trueButton.setOnClickListener { view: View ->
          ...
      }
      falseButton.setOnClickListener { view: View ->
      binding.falseButton.setOnClickListener { view: View ->
          ...
      }
    }
    ...
}
```

For each view with the `android:id` attribute defined in its layout XML, View Binding will generate a property on a corresponding **`ViewBinding`** class. Even better, View Binding automatically declares the type of the property to match the type of the view in your XML. So, for example, `binding.trueButton` is of type **`Button`**, because the view with the ID `true_button` is a `<Button>`.

Unlike a **`findViewById(…)`** call, this has the benefit of keeping your XML layouts and activities in sync if you change the kinds of views in your UI.

Next, using `questionBank` and `currentIndex`, retrieve the resource ID for the question text of the current question. Use the `binding` property to set the text for the question's **`TextView`**.

### Listing 2.9 Wiring up the **`TextView`** (`MainActivity.kt`)

```
override fun onCreate(savedInstanceState: Bundle?) {
    ...
    binding.falseButton.setOnClickListener { view: View ->
        ...
    }

    val questionTextResId = questionBank[currentIndex].textResId
    binding.questionTextView.setText(questionTextResId)
}
```

Save your files and check for any errors. Then run GeoQuiz. You should see the first question in the array appear in the **`TextView`**, as before.

Now, make the NEXT button functional by setting a **`View.OnClickListener`** on it. This listener will increment the index and update the **`TextView`**'s text.

### Listing 2.10 Wiring up the new button (`MainActivity.kt`)

```
override fun onCreate(savedInstanceState: Bundle?) {
    ...
    binding.falseButton.setOnClickListener { view: View ->
        ...
    }

    binding.nextButton.setOnClickListener {
        currentIndex = (currentIndex + 1) % questionBank.size
        val questionTextResId = questionBank[currentIndex].textResId
        binding.questionTextView.setText(questionTextResId)

    }

    val questionTextResId = questionBank[currentIndex].textResId
    binding.questionTextView.setText(questionTextResId)
}
```

You now have the same code in two places that updates the text displayed in `binding.questionTextView`. Take a moment to put this code into a function instead, as shown in Listing 2.11. Then invoke that function in the `nextButton`'s listener and at the end of **`onCreate(Bundle?)`** to initially set the text in the activity's view.

### Listing 2.11 Encapsulating with a function (`MainActivity.kt`)

```
class MainActivity : AppCompatActivity() {
    ...
    override fun onCreate(savedInstanceState: Bundle?) {
        ...
        binding.nextButton.setOnClickListener {
            currentIndex = (currentIndex + 1) % questionBank.size
            val questionTextResId = questionBank[currentIndex].textResId
            binding.questionTextView.setText(questionTextResId)
            updateQuestion()
        }

        val questionTextResId = questionBank[currentIndex].textResId
        binding.questionTextView.setText(questionTextResId)
        updateQuestion()
    }

    private fun updateQuestion() {
        val questionTextResId = questionBank[currentIndex].textResId
        binding.questionTextView.setText(questionTextResId)
    }
}
```

Run GeoQuiz and test your NEXT button.

Now that you have the questions behaving appropriately, it is time to turn to the answers. At the moment, GeoQuiz thinks that the answer to every question is "true." Let's rectify that. You will add a private named function to **`MainActivity`** to encapsulate code rather than writing similar code in two places:

```
    private fun checkAnswer(userAnswer: Boolean)
```

This function will accept a `Boolean` variable that identifies whether the user pressed TRUE or FALSE. Then, it will check the user's answer against the answer in the current **`Question`** object. Finally, after determining whether the user answered correctly, it will make a **`Toast`** that displays the appropriate message to the user.

In `MainActivity.kt`, add the implementation of **`checkAnswer(Boolean)`** shown in Listing 2.12.

### Listing 2.12 Adding **`checkAnswer(Boolean)`** (`MainActivity.kt`)

```
class MainActivity : AppCompatActivity() {
    ...
    private fun updateQuestion() {
        ...
    }

    private fun checkAnswer(userAnswer: Boolean) {
        val correctAnswer = questionBank[currentIndex].answer

        val messageResId = if (userAnswer == correctAnswer) {
            R.string.correct_toast
        } else {
            R.string.incorrect_toast
        }

        Toast.makeText(this, messageResId, Toast.LENGTH_SHORT)
                .show()
    }
}
```

Within the buttons' listeners, call **`checkAnswer(Boolean)`**, as shown in Listing 2.13.

### Listing 2.13 Calling **`checkAnswer(Boolean)`** (`MainActivity.kt`)

```
override fun onCreate(savedInstanceState: Bundle?) {
    ...
    binding.trueButton.setOnClickListener { view: View ->
        Toast.makeText(
            this,
            R.string.correct_toast,
            Toast.LENGTH_SHORT
        )
            .show()
        checkAnswer(true)
    }

    binding.falseButton.setOnClickListener { view: View ->
        Toast.makeText(
            this,
            R.string.correct_toast,
            Toast.LENGTH_SHORT
        )
            .show()
        checkAnswer(false)
    }
    ...
}
```

Run GeoQuiz. Verify that the toasts display the right message based on the answer to the current question and the button you press.

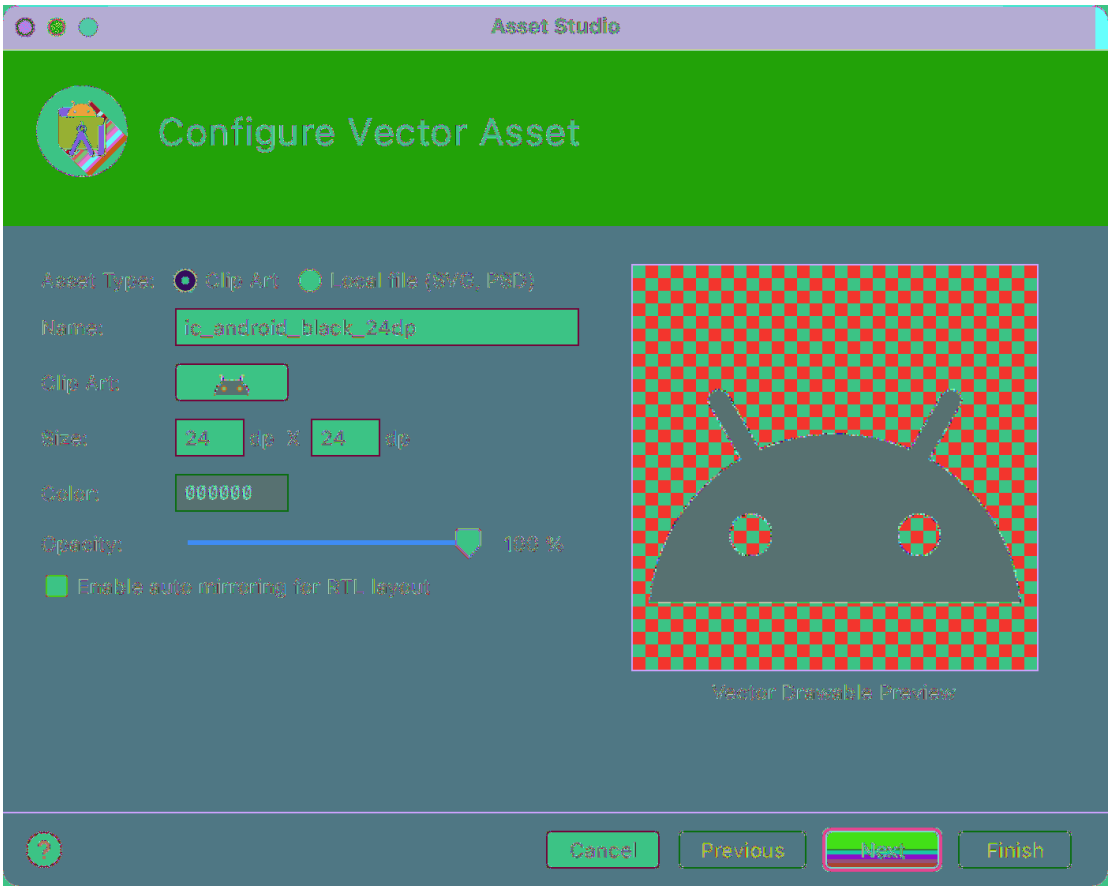
## Adding an Icon

GeoQuiz is up and running, but the UI would be spiffier if the NEXT button also displayed a right-pointing arrow icon.

Since Android 5.0 (Lollipop, API level 21), the Android platform has provided support for vector graphics using the **`VectorDrawable`** class. Whenever possible, we recommend that you use vector drawables to display vector graphics in your apps. Vector drawables are scalable without any loss of visual quality, so they always look crisp and free of image artifacts. And they are more space efficient than traditional bitmap images, resulting in a smaller final application.

Add a vector drawable to the project for the right-pointing arrow icon. First, select File → New → Vector Asset from the menu bar to bring up the Asset Studio (Figure 2.6).

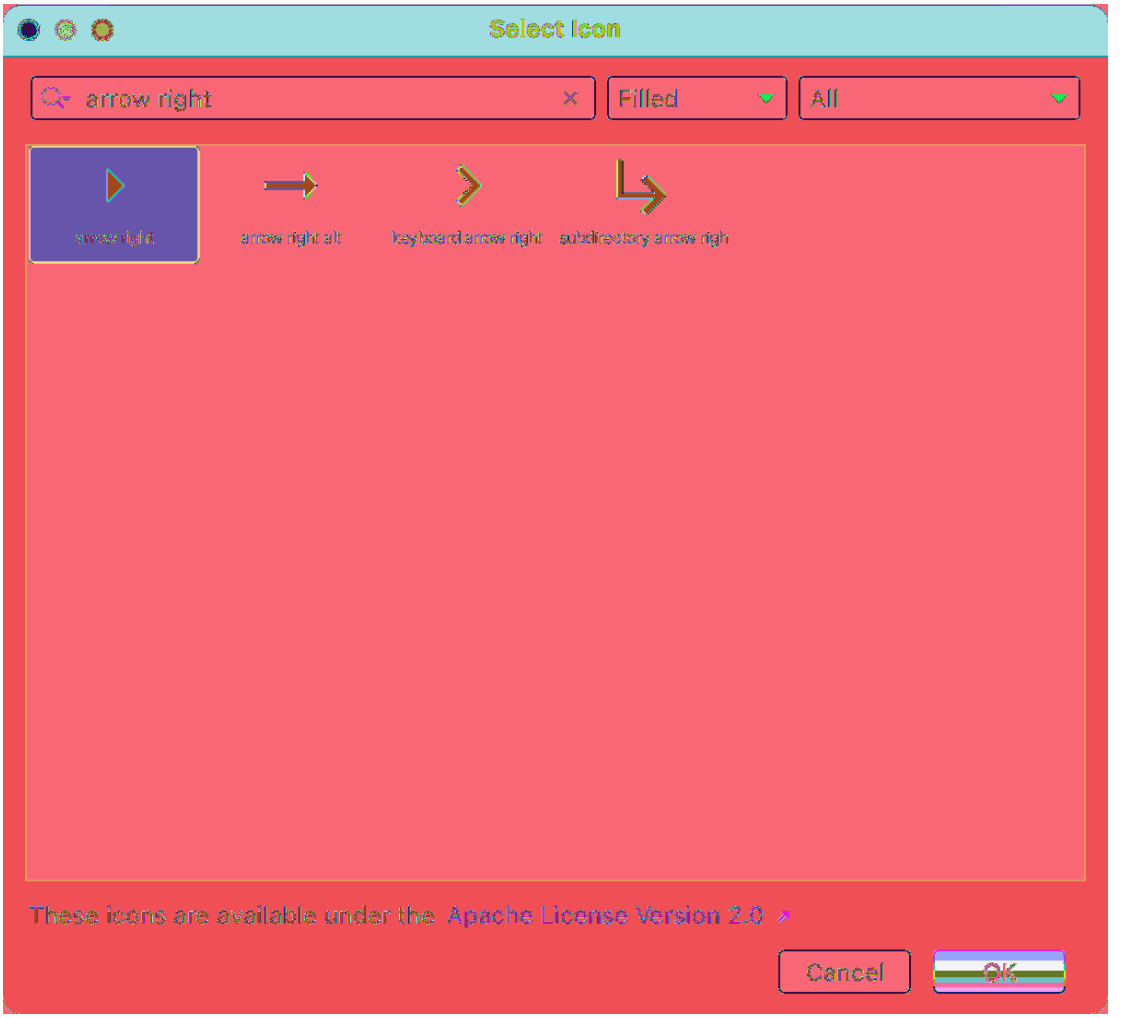
Figure 2.6 The Vector Asset Studio




You can import common file formats for vector graphics (SVG, PSD), but you are going to use an icon from Google's Material icons library. They are free to use and licensed under Apache license 2.0.

For the Asset Type at the top of the Configure Vector Asset dialog, make sure the Clip Art radio button is selected. Then, click the button to the right of the Clip Art: label. The Select Icon window will pop up (Figure 2.7).

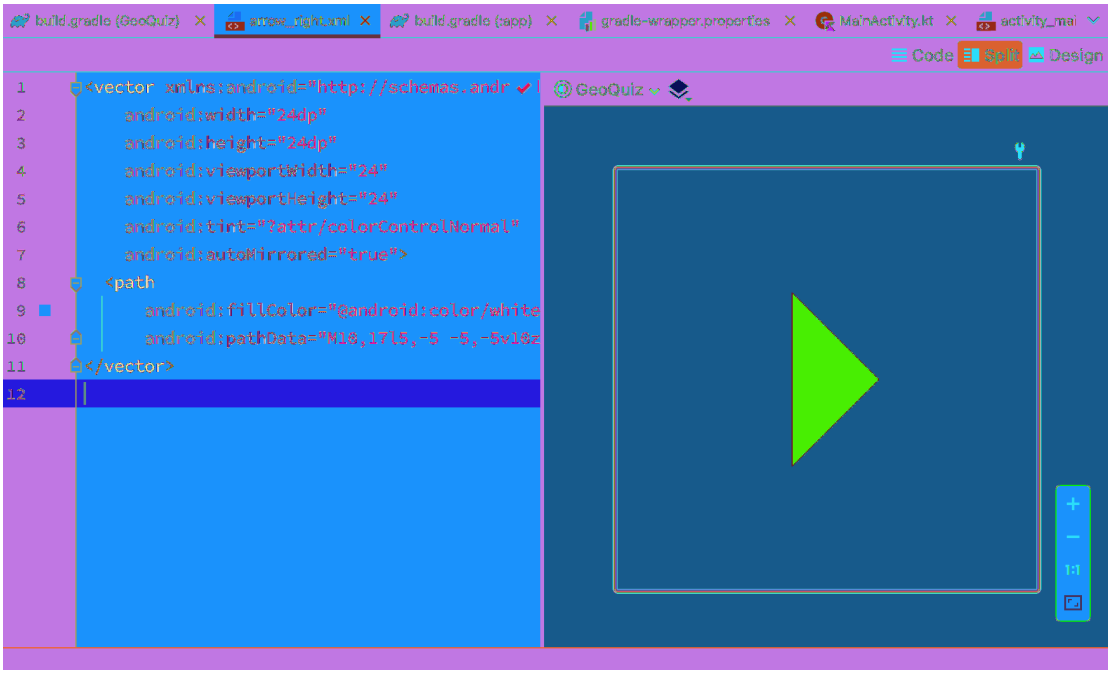
Figure 2.7 The Vector Asset Studio library




Search for “arrow right,” select the icon with that name, and click OK. The name that the Asset Studio generates for you on the configuration dialog is verbose; rename the asset `arrow_right`. Those are all the changes you need to make, so click Next, and then Finish on the following screen.

Now, expand the `app/res/drawable` folder in the project window and open up `arrow_right.xml`. Click the Split tab at the top of the editor to see the XML for your vector drawable on the left and its preview image on the right (Figure 2.8).

Figure 2.8 Your vectorized right arrow




You will learn more about how the Android resource system works starting in Chapter 3. For now, let's put that right arrow to work.

## Referencing resources in XML

You use resource IDs to reference resources in code. But you want to configure the NEXT button to display the arrow in the layout definition. How do you reference a resource from XML?

Answer: with a slightly different syntax. Open `activity_main.xml`. Add the `app` namespace in the root element (you will learn more about this namespace in Chapter 15). Then add two attributes to the third **`Button`** view's definition.

Listing 2.14 Adding an icon to the NEXT button (`res/layout/activity_main.xml`)

```
<LinearLayout
    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:gravity="center"
    android:orientation="vertical">
    ...
    <LinearLayout ... >
        ...
    </LinearLayout>

    <Button
        android:id="@+id/next_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/next_button"
        app:icon="@drawable/arrow_right"
        app:iconGravity="end" />

</LinearLayout>
```

In an XML resource, you refer to another resource by its resource type and name. A reference to a string resource begins with `@string/`. A reference to a drawable resource begins with `@drawable/`. For this situation, you reference your icon with `@drawable/arrow_right`.

You will learn more about naming resources and working in the `res` directory structure starting in Chapter 3.

Run GeoQuiz and admire your button's new appearance. Then test it to make sure it still works as before.

## Screen Pixel Densities

In `activity_main.xml`, you specified attribute values in terms of `dp` units. Now it is time to learn what they are.

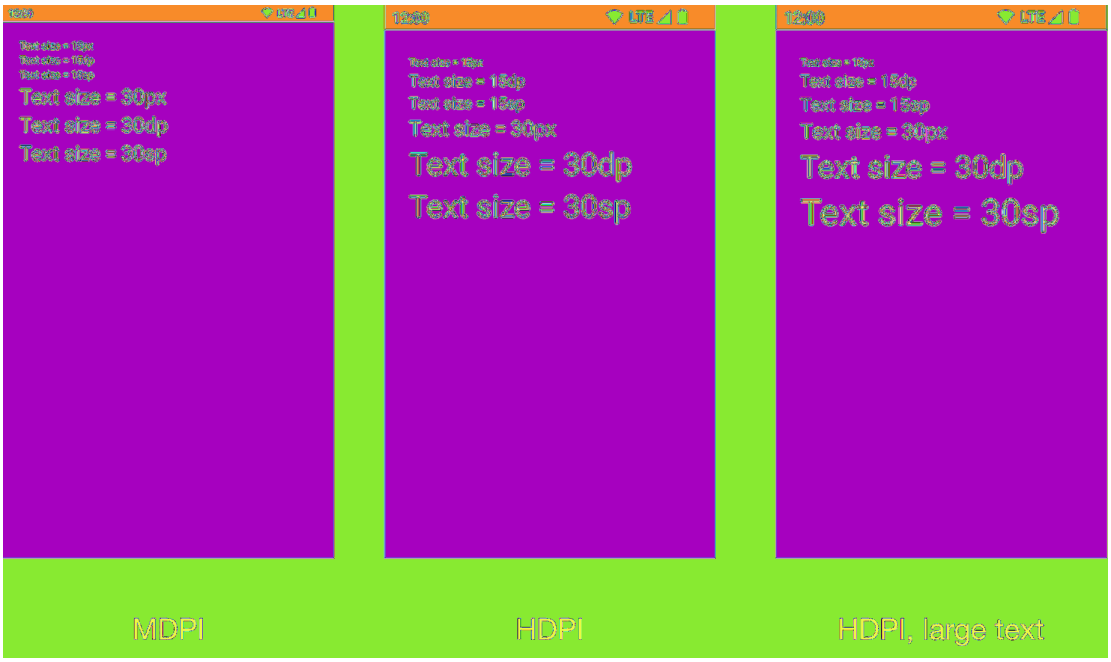
Sometimes you need to specify values for view attributes in terms of specific sizes (usually in pixels, but sometimes points, millimeters, or inches). You see this most commonly with attributes for text size, margins, and padding. Text size is the pixel height of the text on the device's screen. Margins specify the distances between views, and padding specifies the distance between a view's outside edges and its content.

Android devices come in a wide variety of shapes and sizes. Even just among phones, specifications such as screen size and resolution vary widely. Modern phones have pixel densities ranging from less than 300 pixels per inch to more than 800 pixels per inch.

What happens when you want to display the same UI on different density screens? Or when the user configures a larger-than-default text size? It would be a very frustrating user experience if your button were tiny on one device and massive on another.

To provide a consistent experience on all devices, Android provides density-independent dimension units that you can use to get the same size on different screen densities. Android translates these units using the device's defined *density bucket* to pixels at runtime, so there is no tricky math for you to do. These density buckets range from low density (LDPI) to medium density (MDPI) to high density (HDPI) and all the way up to extra-extra-extra-high density (XXXHDPI) (Figure 2.9).

Figure 2.9 Dimension units in action on **`TextView`**




`px` Short for *pixel*. One pixel corresponds to one onscreen pixel, regardless of the display density. Because pixels do not scale appropriately with device display density, their use is not recommended.

`dp` Short for *density-independent pixel* and usually pronounced "dip." You typically use this for margins, padding, or anything else for which you would otherwise specify size with a pixel value. One `dp` is always 1/160 of an inch on a device's screen. You get the same size regardless of screen density: When your display is a higher density, density-independent pixels will fill a larger number of screen pixels.

`sp` Short for *scale-independent pixel*. Scale-independent pixels are density-independent pixels that also take into account the user's font size preference. You will almost always use `sp` to set display text size.

`pt`, `mm`, `in` These are scaled units, like `dp`, that allow you to specify interface sizes in points (1/72 of an inch), millimeters, or inches. However, we do not recommend using them: Not all devices are correctly configured for these units to scale correctly.

In practice and in this book, you will use `dp` and `sp` almost exclusively. Android will translate these values into pixels at runtime.

## Running on a Device

It is fun to interact with your app on an emulator. It is even more fun to interact with your app on physical Android device. In this section, you will set up your system, device, and application to get GeoQuiz running on your hardware device.

First, plug the device into your computer. If you are developing on a Mac, your system should recognize the device right away. On Windows, you may need to install the `adb` (Android Debug Bridge) driver. If Windows cannot find the `adb` driver, then download one from the device manufacturer's website.

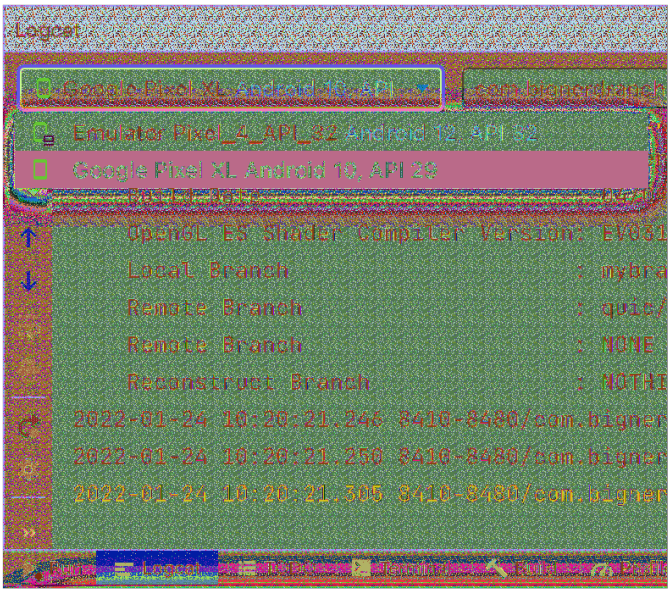
Second, enable USB debugging on your Android device. To do this, you need to access the developer options settings menu, which is not visible by default. To enable developer options, open the Settings app and press the search icon in the top-right corner. Search for "Build number" and select the first search result. The Settings app will navigate you to the Build number row.

Press Build number seven times in quick succession. After several presses, you will see a message telling you how many "steps" (presses of the build number) you are from being a developer. When you see You are now a developer!, you can stop. Then you can return to Settings, search for "Developer options," and enable USB debugging.

The steps to enable USB debugging vary considerably across devices and versions of Android. If you are having problems enabling your device, visit `https://developer.android.com/studio/debug/dev-options#enable` for more help.

Finally, confirm that your device is recognized by finding it in the dropdown to the left of Android Studio's run icon. The text in the dropdown should display the name of your device. If your device is not selected, open the dropdown and select it from the options under the Running devices section.

Figure 2.10 Viewing connected devices



If you are having trouble getting your device recognized, verify that your device is turned on and that the USB debugging option is enabled.

If you are still unable to see your device in the devices view, you can find more help on the Android developers' site. Start at `developer.android.com/tools/device.html`. You can also visit this book's forum at `forums.bignerdranch.com` for more troubleshooting help.

Run GeoQuiz as before. This time, GeoQuiz will launch on your real device.

## Challenge: Add a Listener to the TextView

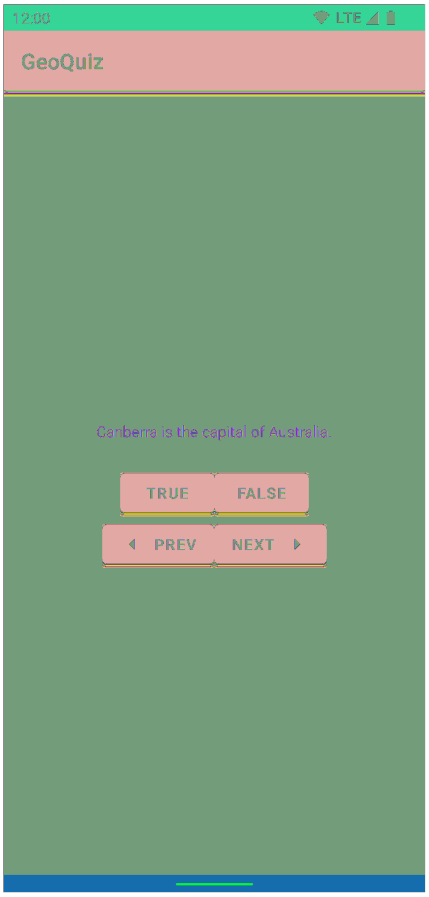
Your NEXT button is nice, but you could also make it so that a user could press the **`TextView`** itself to see the next question.

Hint: You can use the same **`View.OnClickListener`** for the **`TextView`** that you have used with the **`Button`**s, because **`TextView`** also inherits from **`View`**.

## Challenge: Add a Previous Button

Add a button that the user can press to go back one question. The UI should look something like Figure 2.11.

Figure 2.11 Now with a previous button!




This is a great challenge. It requires you to retrace many of the steps in these first two chapters.

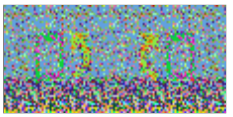
# 3
# The Activity Lifecycle

At this point, you have an app with some functionality. Unfortunately, there is a bug in your app. In this chapter, you will learn about the underlying mechanics that created this bug and how they affect other parts of your application.

## Rotating GeoQuiz

GeoQuiz works great … until you rotate the device. While the app is running, press the NEXT button to show another question. Then rotate the device. If you are running on the emulator, click the rotate left or rotate right button in the floating toolbar to rotate (Figure 3.1).

Figure 3.1 Control the roll



If the emulator does not show GeoQuiz in landscape orientation after you press one of the rotate buttons, turn auto-rotate on. Swipe down from the top of the screen to open Quick Settings. Press the auto-rotate icon (Figure 3.2).

Figure 3.2 Quick Setting for auto-rotate




After you rotate, you will see the first question again. How and why did this happen? The answers to these questions have to do with the activity lifecycle.

You will learn how to fix this problem in Chapter 4. But first, it is important to understand the root of the problem so you can avoid related bugs that might creep up.

## Activity States and Lifecycle Callbacks

Every instance of **`Activity`** has a lifecycle. During this lifecycle, an activity transitions between four states: resumed, started, created, and nonexistent. For each transition, there is an **`Activity`** function that notifies the activity of the change in its state. Figure 3.3 shows the activity lifecycle, states, and functions.

Figure 3.3 Activity state diagram

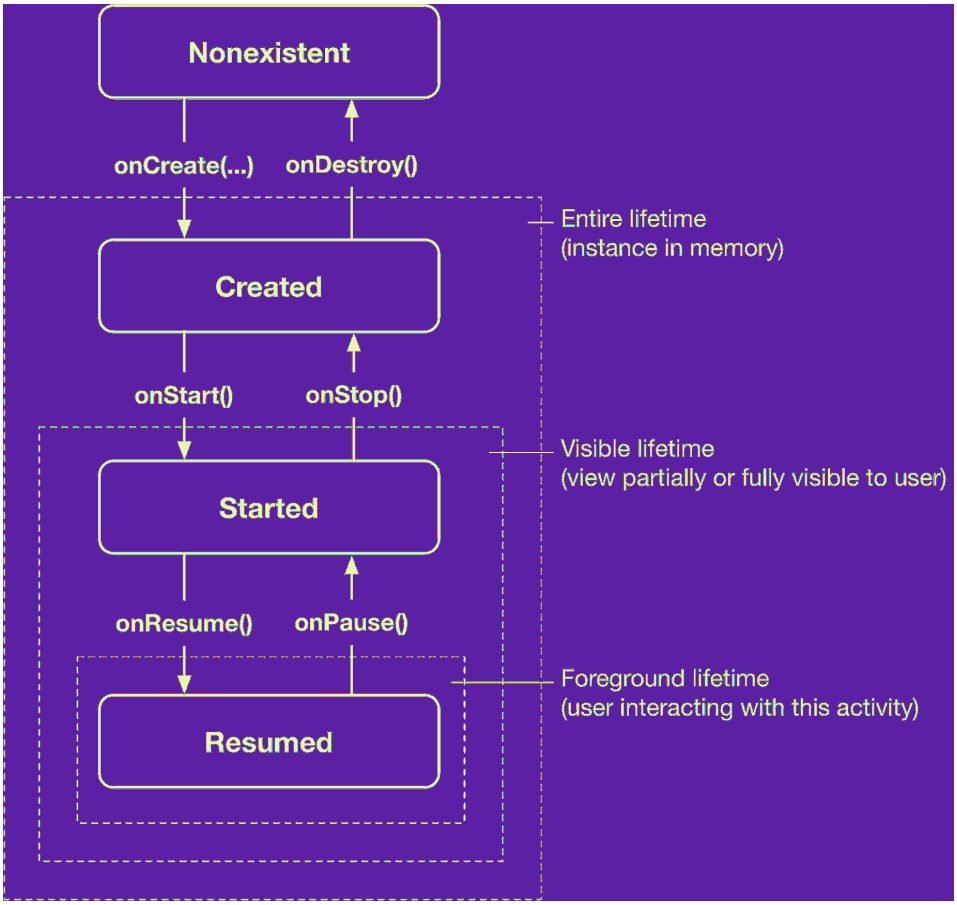


Figure 3.3 indicates for each state whether the activity has an instance in memory, is visible to the user, or is active in the foreground (accepting user input). Table 3.1 summarizes this information.

Table 3.1 Activity states

| State | In memory? | Visible to user? | In foreground? |
|---|---|---|---|
| nonexistent | no | no | no |
| created | yes | no | no |
| started | yes | yes/partially* | no |
| resumed | yes | yes | yes |

(* Depending on the circumstances, a started activity may be fully or partially visible.)

*Nonexistent* represents an activity that has not been launched yet or an activity that was destroyed (by the user completely killing the app, for example). For that reason, this state is sometimes referred to as the "destroyed" state. There is no instance in memory, and there is no associated view for the user to see or interact with.

*Created* represents an activity that has an instance in memory but whose view is not visible on the screen. This state occurs in passing when the activity is first spinning up and reoccurs any time the view is fully out of view (such as when the user launches another full-screen activity to the foreground, navigates to the Home screen, or uses the overview screen to switch tasks).

*Started* represents an activity that has lost focus but whose view is visible or partially visible. An activity would be partially visible, for example, if the user launched a new dialog-themed or transparent activity on top of it. An activity could also be fully visible but not in the foreground if the user is viewing two activities in multi-window mode (also called "split-screen mode").

*Resumed* represents an activity that is in memory, fully visible, and in the foreground. It is usually the state of the activity the user is currently interacting with.

Subclasses of **`Activity`** can take advantage of the functions named in Figure 3.3 to get work done at critical transitions in the activity's lifecycle. These functions are often called *lifecycle callbacks*.

You are already acquainted with one of these lifecycle callback functions – **`onCreate(Bundle?)`**. The OS calls this function after the activity instance is created but before it is put onscreen.

Typically, an activity overrides **`onCreate(Bundle?)`** to prepare the specifics of its UI:

- inflating views and putting them onscreen (in the call to **`setContentView()`**)
- getting references to inflated views (via View Binding or **`findViewById()`**)
- setting listeners on views to handle user interaction
- connecting to external model data

It is important to understand that you never call **`onCreate(Bundle?)`** or any of the other **`Activity`** lifecycle functions yourself. You simply override the callbacks in your activity subclass. Then Android calls the lifecycle callbacks at the appropriate time (in relation to what the user is doing and what is happening across the rest of the system) to notify the activity that its state is changing.

# Logging the Activity Lifecycle

In this section, you are going to override lifecycle functions to eavesdrop on **`MainActivity`**'s lifecycle. Each implementation will simply log a message informing you that the function has been called. This will help you see how **`MainActivity`**'s state changes at runtime in relation to what the user is doing.

## Making log messages

In Android, the **`android.util.Log`** class sends log messages to a shared system-level log. **`Log`** has several functions for logging messages. The one that you will use most often in this book is **`d(…)`**, which stands for "debug." (There are many levels of logging; you can learn more about them in the section called For the More Curious: Log Levels near the end of this chapter.)

This function takes two parameters, both **`String`**s. The first parameter identifies the source of the message, and the second is the contents of the message.

The first string is typically a `TAG` constant with the class name as its value. This makes it easy to determine the source of a particular message.

Open `MainActivity.kt` and add a `TAG` constant:

### Listing 3.1 Adding a `TAG` constant (`MainActivity.kt`)

```
import ...

private const val TAG = "MainActivity"

class MainActivity : AppCompatActivity() {
    ...
}
```

Next, in **`onCreate(Bundle?)`**, call **`Log.d(…)`** to log a message.

### Listing 3.2 Adding a log statement to **`onCreate(Bundle?)`** (`MainActivity.kt`)

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    Log.d(TAG, "onCreate(Bundle?) called")
    binding = ActivityMainBinding.inflate(layoutInflater)
    setContentView(binding.root)
    ...
}
```

Now override five more lifecycle functions in **`MainActivity`** after **`onCreate(Bundle?)`**:

### Listing 3.3 Overriding more lifecycle functions (`MainActivity.kt`)

```
class MainActivity : AppCompatActivity() {
    ...
    override fun onCreate(savedInstanceState: Bundle?) {
        ...
    }

    override fun onStart() {
        super.onStart()
        Log.d(TAG, "onStart() called")
    }

    override fun onResume() {
        super.onResume()
        Log.d(TAG, "onResume() called")
    }

    override fun onPause() {
        super.onPause()
        Log.d(TAG, "onPause() called")
    }

    override fun onStop() {
        super.onStop()
        Log.d(TAG, "onStop() called")
    }

    override fun onDestroy() {
        super.onDestroy()
        Log.d(TAG, "onDestroy() called")
    }

    private fun updateQuestion() {
        ...
    }
    ...
}
```

Notice that you call the superclass implementations before you log your messages. These superclass calls are required. Calling the superclass implementation should be the first line of each callback function override implementation.

You may have been wondering about the `override` keyword. This asks the compiler to ensure that the class actually has the function that you want to override. For example, the compiler would alert you to the following misspelled function name:

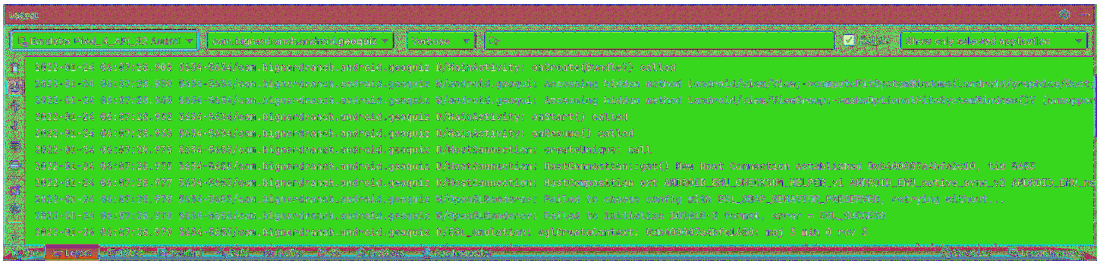
```
    override fun onCreat(savedInstanceState: Bundle?) {
        ...
    }
```

The parent **`AppCompatActivity`** class does not have an **`onCreat(Bundle?)`** function, so the compiler will complain. This way you can fix the typo right away, rather than waiting until you run the app and see strange behavior to discover the error.

## Using Logcat

Rerun GeoQuiz, and messages will start materializing in the Logcat tool window at the bottom of Android Studio, as shown in Figure 3.4. If Logcat does not open automatically when you run GeoQuiz, you can open it by clicking the Logcat tool window bar at the bottom of the Android Studio window.

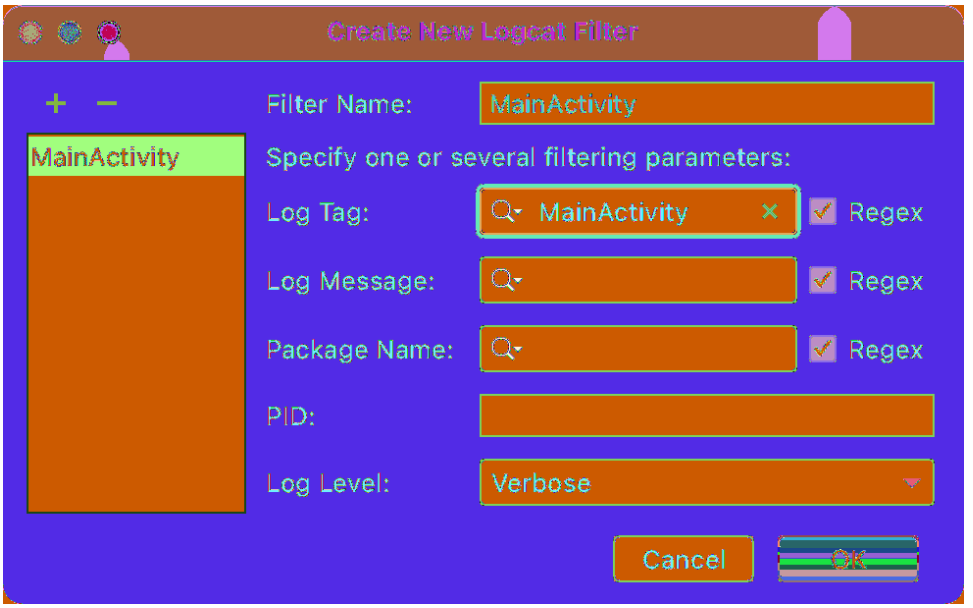
Figure 3.4 Android Studio with Logcat



You will see your own messages along with some system output. To make your messages easier to find, you can filter the output using the value you set for the `TAG` constant. In Logcat, click the dropdown in the top right that reads Show only selected application. This is the filter dropdown, which is currently set to show messages from only your app.

In the filter dropdown, select Edit Filter Configuration to create a new filter. Name the filter MainActivity and enter MainActivity in the Log Tag field (Figure 3.5).

Figure 3.5 Creating a filter in Logcat



Click OK. Now, only messages tagged `MainActivity` will be visible in Logcat (Figure 3.6).

Figure 3.6 Launching GeoQuiz creates, starts, and resumes an activity



Three lifecycle functions were called after GeoQuiz was launched and the initial instance of **`MainActivity`** was created: **`onCreate(Bundle?)`**, **`onStart()`**, and **`onResume()`**. Your **`MainActivity`** instance is now in the resumed state (in memory, visible, and active in the foreground).

## Exploring How the Activity Lifecycle Responds to User Actions

As you continue through this book, you will override the different activity lifecycle functions to do real things for your application. When you do, you will learn more about the uses of each function. For now, have some fun familiarizing yourself with how the lifecycle behaves in common usage scenarios by interacting with your app and checking out the logs in Logcat.

### Temporarily leaving an activity

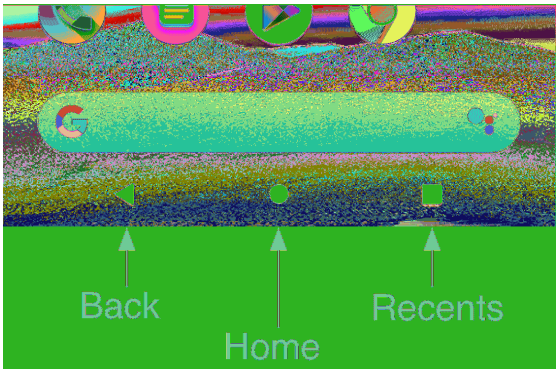
Navigate to your emulator or device's Home screen with the Home gesture – swiping up from the gesture navigation interface at the bottom of the screen (Figure 3.7).

Figure 3.7 Gesture navigation interface



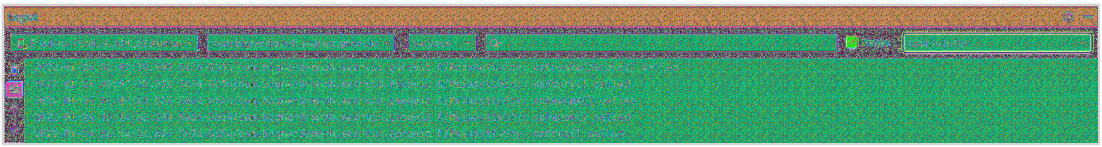
If your device does not use gesture navigation, you may have Back, Home, and Recents buttons at the bottom of the screen (Figure 3.8). (The emulator has these three buttons in the toolbar at the top of its tool window, and it also shows them on the emulated device when appropriate.) In this case, press the Home button to get to the Home screen. If neither of these approaches works on your device, consult the device manufacturer's user guide.

Figure 3.8 Back, Home, and Recents buttons




When the Home screen displays, **`MainActivity`** moves completely out of view. What state is **`MainActivity`** in now? Check Logcat to see. Your activity received calls to **`onPause()`** and **`onStop()`**, but not **`onDestroy()`** (Figure 3.9).

Figure 3.9 Navigating Home stops the activity



By navigating Home, the user is telling Android, "I'm going to go look at something else, but I might come back. I'm not really done with this screen yet." Android pauses and ultimately stops the activity.

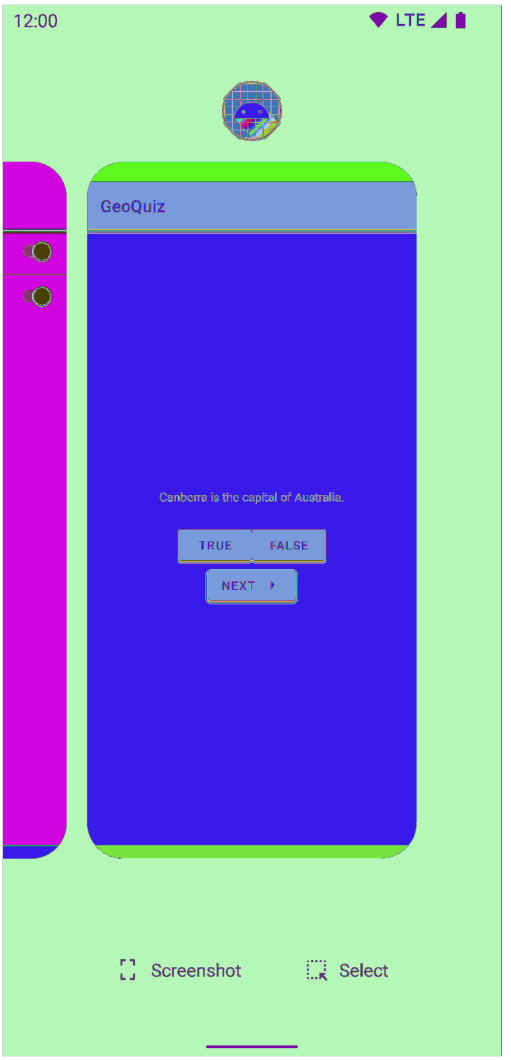
So after you navigate Home from GeoQuiz, your instance of **`MainActivity`** hangs out in the created state (in memory, not visible, and not active in the foreground). Android does this so it can quickly and easily restart **`MainActivity`** where you left off when you come back to GeoQuiz later.

(This is not the whole story about navigating to the Home screen. Applications in the created state can be destroyed at the discretion of the OS. See Chapter 4 for the rest of the story.)

Go back to GeoQuiz by selecting the GeoQuiz task card from the *overview screen*. To do this, use the Recents button or gesture (swiping up from the bottom and holding, then releasing).

Each card in the overview screen represents an app the user has interacted with (Figure 3.10). (By the way, the overview screen is often called the "recents screen" or "task manager" by users. We defer to the developer documentation, which calls it the "overview screen.")

Figure 3.10 Overview screen



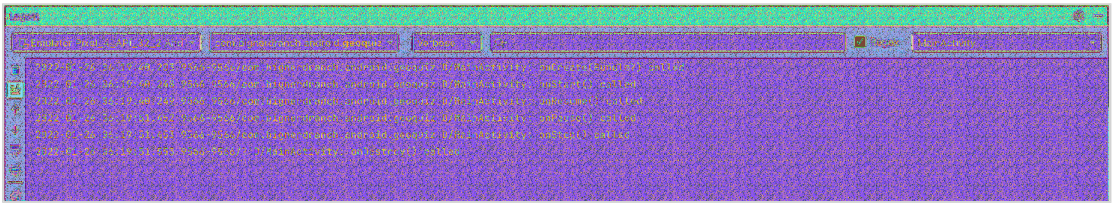

Click the GeoQuiz task card in the overview screen. **`MainActivity`** will fill the screen.

A quick look at Logcat shows that your activity got calls to **`onStart()`** and **`onResume()`**. Note that **`onCreate(…)`** was not called. This is because **`MainActivity`** was in the created state after the user navigated to the Home screen. Because the activity instance was still in memory, it did not need to be created. Instead, the activity only had to be started (moved to the started/visible state) and then resumed (moved to the resumed/foreground state).

## Finishing an activity

Open the overview screen again, then swipe up on the app's card so that it goes offscreen. Check Logcat. Your activity received calls to **`onPause()`**, **`onStop()`**, and **`onDestroy()`** (Figure 3.11). Your **`MainActivity`** instance is now in the nonexistent state (not in memory and thus not visible – and certainly not active in the foreground).

Figure 3.11 Closing the app destroys the activity



When you swiped away GeoQuiz's card, you as the user of the app *finished* the activity. You told Android, "I'm done with this activity, and I won't need it anymore." Android then destroyed your activity's view and removed all traces of the activity from memory. This is Android's way of being frugal with your device's limited resources.

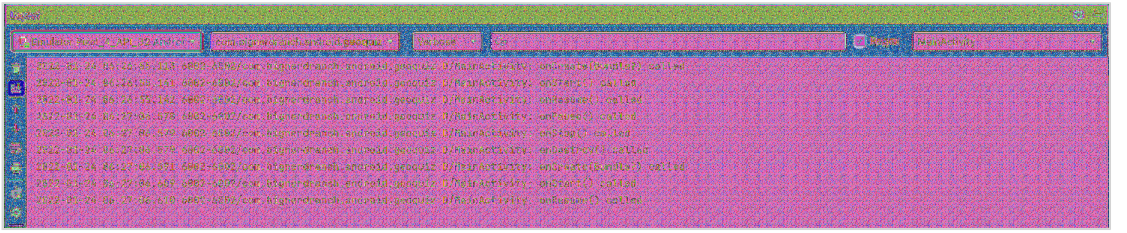
As a developer, you can programmatically finish an activity by calling **`Activity.finish()`**. In Chapter 7, you will learn about another way to finish an activity by navigating Back.

## Rotating an activity

Now it is time to get back to the bug you found at the beginning of this chapter. Run GeoQuiz, press the NEXT button to reveal the second question, and then rotate the device.

After rotating, GeoQuiz will display the first question again. Check Logcat to see what has happened. Your output should look like Figure 3.12.

Figure 3.12 **`MainActivity`** is dead. Long live **`MainActivity`**!



When you rotated the device, the instance of **`MainActivity`** that you were looking at was destroyed and a new one was created. Rotate the device again to witness another round of destruction and rebirth.

This is the source of your GeoQuiz bug. Each time you rotate the device, the current **`MainActivity`** instance is completely destroyed. The value that was stored in `currentIndex` in that instance is wiped from memory. This means that when you rotate, GeoQuiz forgets which question you were looking at. As rotation finishes, Android creates a new instance of **`MainActivity`** from scratch. `currentIndex` is re-initialized to `0`, and the user starts over at the first question.

You will fix this bug in Chapter 4.

# Device Configuration Changes and the Activity Lifecycle

Rotating the device changes the *device configuration*. The device configuration is a set of characteristics that describe the current state of an individual device. The characteristics that make up the configuration include screen orientation, screen density, screen size, keyboard type, dock mode, language, and more.

Applications can provide alternative resources to match device configurations. When a *runtime configuration change* occurs, there may be resources that are a better match for the new configuration. So Android destroys the activity, looks for resources that are the best fit for the new configuration, and then rebuilds a new instance of the activity with those resources.

You can try providing specific resources in the section called For the More Curious: Creating a Landscape Layout below or wait until Chapter 18.

## For the More Curious: Creating a Landscape Layout

One of the core considerations around device configuration is the desire to provide appropriate resources based on the configuration. When the user's language is German, your **`Activity`** should display German text. When the user enables night mode, your color scheme should adjust to be easier on the eyes.

You enable this functionality by providing configuration-specific resources in your app. Another configuration-specific attribute you can provide different resources for is the orientation of the device's screen: You can create a landscape-only layout so that when the user rotates their device to landscape, they see a layout specifically designed for landscape screens. For example, you could define an entirely new layout for **`MainActivity`**, moving the NEXT button to the bottom-right corner of the screen so that it is easier to reach in landscape mode (Figure 3.13):

Figure 3.13 **`MainActivity`** in landscape orientation




Landscape screen orientation is one of many configuration qualifiers available on Android. You will learn more about configuration qualifiers and how they can be used to provide the correct resources in Chapter 18.

## For the More Curious: UI Updates and Multi-Window Mode

Prior to Android 7.0 Nougat, most activities spent very little time in the started state. Instead, activities passed through the started state quickly on their way to either the resumed state or the created state. Because of this, many developers assumed they only needed to update their UI when their activity was in the resumed state. It was common practice to use **`onResume()`** and **`onPause()`** to start or stop any ongoing updates related to the UI (such as animations or data refreshes).

When multi-window mode was introduced in Nougat, it broke the assumption that resumed activities were the only fully visible activities. This, in turn, broke the intended behavior of many apps. Now, started activities can be fully visible for extended periods of time when the user is in multi-window mode. And users will expect those started activities to behave as if they were resumed.

Consider video, for example. Suppose you had a pre-Nougat app that provided simple video playback. You started (or resumed) video playback in **`onResume()`** and paused playback in **`onPause()`**. But then multi-window mode comes along, and your app stops playback when it is started but users are interacting with another app in the second window. Users start complaining, because they want to watch their videos *while* they send a text message in a separate window.

Luckily, the fix is relatively simple: Move your playback resuming and pausing to **`onStart()`** and **`onStop()`**. This goes for any live-updating data, like a photo gallery app that refreshes to show new images as they are pushed to a Flickr stream (as you will see later in this book).

In short, your activities should update the UI during their entire visible lifecycle, from **`onStart()`** to **`onStop()`**.

Unfortunately, not everyone got the memo, and many apps still misbehave in multi-window mode. To fix this, the Android team introduced *multi-resume* for multi-window mode in Android 10. Multi-resume means that the fully visible activity in each of the windows will be in the resumed state when the device is in multi-window mode, regardless of which window the user last touched.

Still, until multi-resume becomes a readily available standard across most devices in the marketplace, use your knowledge of the activity lifecycle to reason about where to place UI update code. You will get a lot of practice doing so throughout this book.

## For the More Curious: Log Levels

When you use the **`android.util.Log`** class to send log messages, you control not only the content of the message but also a *level* that specifies how important the message is. Android supports five log levels, shown in Table 3.2. Each level has a corresponding function in the **`Log`** class. Sending output to the log is as simple as calling the corresponding **`Log`** function.

Table 3.2 Log levels and functions

| Log level | Function | Used for |
|---|---|---|
| ERROR | **`Log.e(…)`** | errors |
| WARNING | **`Log.w(…)`** | warnings |
| INFO | **`Log.i(…)`** | informational messages |
| DEBUG | **`Log.d(…)`** | debug output<br>(may be filtered out) |
| VERBOSE | **`Log.v(…)`** | development only |

In addition, each of the logging functions has two signatures: one that takes a `TAG` string and a message string, and a second that takes those two arguments plus an instance of **`Throwable`**, which makes it easy to log information about a particular exception that your application might throw. Here are some sample log function signatures:

```
// Log a message at DEBUG log level
Log.d(TAG, "Current question index: $currentIndex")

try {
    val question = questionBank[currentIndex]
} catch (ex: ArrayIndexOutOfBoundsException) {
    // Log a message at ERROR log level along with an exception stack trace
    Log.e(TAG, "Index was out of bounds", ex)
}
```

## Challenge: Preventing Repeat Answers

Once a user provides an answer for a particular question, disable the buttons for that question to prevent multiple answers being entered.

## Challenge: Graded Quiz

After the user provides answers for all the quiz questions, display a **`Toast`** with a percentage score for the quiz. Good luck!

# 4

# Persisting UI State

Destroying and re-creating activities on rotation can cause headaches, such as GeoQuiz's bug of reverting to the first question when the device is rotated. To fix this bug, the post-rotation **`MainActivity`** instance needs to retain the old value of `currentIndex`. You need a way to save this data across a runtime configuration change, like rotation.

Luckily, there is a class that survives rotation that you can use to store your state. In this chapter, you will fix GeoQuiz's UI state loss on rotation bug by storing its UI data in a **`ViewModel`**.

A **`ViewModel`** is the perfect complement to an **`Activity`** because of its simple lifecycle and ability to persist data across configuration changes. It is usually scoped to a single screen and is a useful place to put logic involved in formatting the data to display on that screen. Using a **`ViewModel`** aggregates all the data the screen needs in one place, formats the data, and makes it easy to access the end result.

You will also address a bug that is harder to discover, but equally problematic – UI state loss on process death – using a **`SavedStateHandle`**. This class allows you to temporarily store simple data outside the lifecycle of your app's process.

## Including the ViewModel Dependency

Before you can write a **`ViewModel`** class, you need to include two libraries in your project. The **`ViewModel`** class comes from an Android Jetpack library called `androidx.lifecycle`, one of many libraries that you will use throughout this book. (You will learn more about Jetpack later in this chapter.) You will also include the `androidx.activity` library, which adds some functionality to your **`MainActivity`**.

To include libraries in your project, you add them to the list of *dependencies*. Like the configuration for View Binding you did back in Chapter 2, your project's dependencies live in the `app/build.gradle` file. Under Gradle Scripts in the project tool window, open the `build.gradle` file labeled Module: GeoQuiz.app so you can add our new dependencies. You should see something like:

```
    plugins {
      id 'com.android.application'
      id 'kotlin-android'
    }

    android {
        ...
    }

    dependencies {
        implementation 'androidx.core:core-ktx:1.7.0'
        implementation 'androidx.appcompat:appcompat:1.4.1'
        ...
    }
```

The `dependencies` section already includes some libraries that your project requires. You might recognize a few of them, such as Espresso or Kotlin. Gradle also allows you to specify new dependencies. When your app is compiled, Gradle will find, download, and include the dependencies for you. All you have to do is specify an exact string incantation, and Gradle will do the rest.

Add the `androidx.lifecycle:lifecycle-viewmodel-ktx` and `androidx.activity:activity-ktx` dependencies to your `app/build.gradle` file, as shown in Listing 4.1. Their exact placement in the `dependencies` section does not matter, but to keep things tidy it is good to put new dependencies under the last existing `implementation` dependency.

### Listing 4.1 Adding dependencies (`app/build.gradle`)

```
dependencies {
    ...
    implementation 'androidx.constraintlayout:constraintlayout:2.1.3'
    implementation 'androidx.lifecycle:lifecycle-viewmodel-ktx:2.4.1'
    implementation 'androidx.activity:activity-ktx:1.4.0'
    ...
}
```

As it did when you made the change to enable View Binding, Android Studio will prompt you to sync the file. Either click Sync Now in the prompt or select File → Sync Project with Gradle Files, and Gradle will take care of the rest.

## Adding a ViewModel

You are ready to create your **ViewModel** subclass, **QuizViewModel**. In the project tool window, right-click the com.bignerdranch.android.geoquiz package and select New → Kotlin Class/File. Enter QuizViewModel for the name and double-click Class in the list of options below.

In QuizViewModel.kt, add an init block and override **onCleared()**. Log the creation and destruction of the **QuizViewModel** instance, as shown in Listing 4.2.

Listing 4.2 Creating a **ViewModel** class (QuizViewModel.kt)

```
private const val TAG = "QuizViewModel"

class QuizViewModel : ViewModel() {

    init {
        Log.d(TAG, "ViewModel instance created")
    }

    override fun onCleared() {
        super.onCleared()
        Log.d(TAG, "ViewModel instance about to be destroyed")
    }
}
```

The **onCleared()** function is called just before a **ViewModel** is destroyed. This is a useful place to perform any cleanup, such as un-observing a data source. For now, you simply log the fact that the **ViewModel** is about to be destroyed so that you can explore its lifecycle, the same way you explored the lifecycle of **MainActivity** in Chapter 3.

Now, open MainActivity.kt and associate the activity with an instance of **QuizViewModel** by invoking the **viewModels()** property delegate.

Listing 4.3 Accessing the **ViewModel** (MainActivity.kt)

```
class MainActivity : AppCompatActivity() {

    private lateinit var binding: ActivityMainBinding

    private val quizViewModel: QuizViewModel by viewModels()

    override fun onCreate(savedInstanceState: Bundle?) {
        ...
        setContentView(binding.root)

        Log.d(TAG, "Got a QuizViewModel: $quizViewModel")

        binding.trueButton.setOnClickListener { view: View ->
            checkAnswer(true)
        }
        ...
    }
    ...
}
```

The `by` keyword indicates that a property is implemented using a *property delegate*. In Kotlin, a property delegate is, as the name suggests, a way to delegate the functionality of a property to an external unit of code. A very common property delegate in Kotlin is **`lazy`**. The **`lazy`** property delegate allows developers to save resources by waiting to initialize the property only when it is accessed.

The **`viewModels()`** property delegate works the same way: Your **`QuizViewModel`** will not be initialized unless you access it. By referencing it in a logging message, you can initialize it and log the value on the same line.

Under the hood, the **`viewModels()`** property delegate handles many things for you. When the activity queries for a **`QuizViewModel`** for the first time, **`viewModels()`** creates and returns a new **`QuizViewModel`** instance. When the activity queries for the **`QuizViewModel`** after a configuration change, the instance that was first created is returned. When the activity is finished (such as when the user closes the app from the overview screen), the **`ViewModel`**–**`Activity`** pair is removed from memory.

You should not directly instantiate the **`QuizViewModel`** within your **`Activity`**. Instead, rely on the **`viewModels()`** property delegate. It might seem like instantiating the **`ViewModel`** yourself would work just the same, but you would lose the benefit of the same instance being returned after your **`Activity`**'s configuration change.

## ViewModel lifecycle

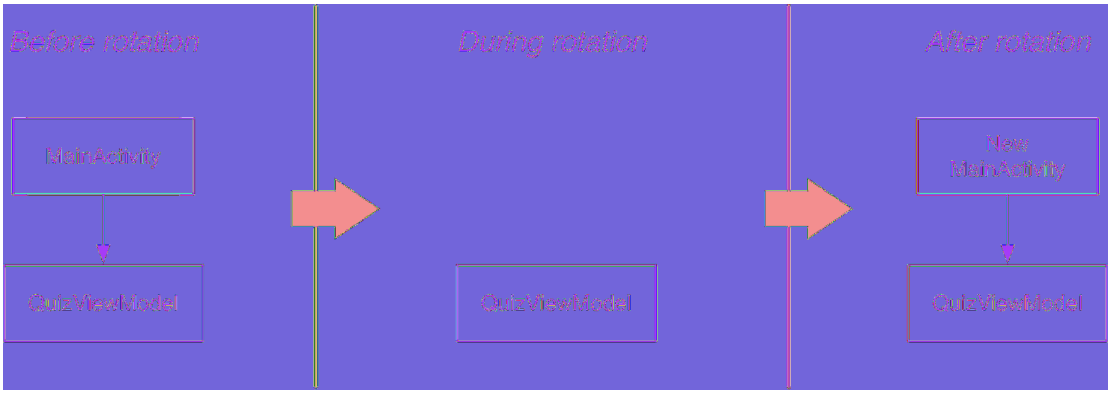
You learned in Chapter 3 that activities transition between four states: resumed, started, created, and nonexistent. You also learned about different ways an activity can be destroyed: either by the user finishing the activity or by the system destroying it as a result of a configuration change.

When the user finishes an activity, they expect their UI state to be reset. When the user rotates an activity, they expect their UI state to be the same after rotation. **`ViewModel`** offers a way to keep an activity's UI state data in memory across configuration changes. Its lifecycle mirrors the user's expectations: It survives configuration changes and is destroyed only when its associated activity is finished.

When you associate a **`ViewModel`** instance with an activity's lifecycle, as you did in Listing 4.3, the **`ViewModel`** is said to be *scoped* to that activity's lifecycle. This means the **`ViewModel`** will remain in memory, regardless of the activity's state, until the activity is finished. Once the activity is finished (such as by the user closing the app from the overview screen), the **`ViewModel`** instance is destroyed.

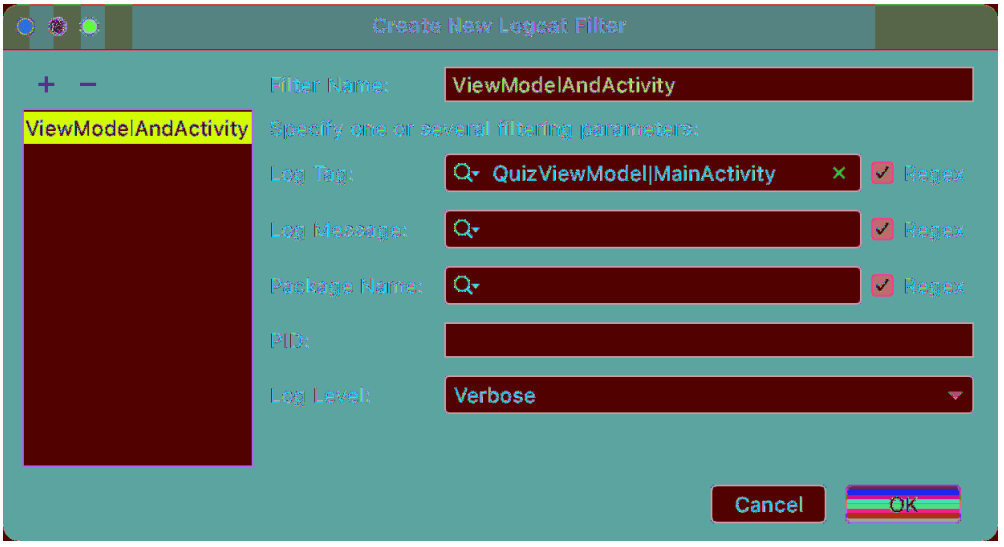
This means that the **`ViewModel`** stays in memory during a configuration change, such as rotation. During the configuration change, the activity instance is destroyed and re-created, but any **`ViewModel`**s scoped to the activity stay in memory. This is depicted in Figure 4.1, using **`MainActivity`** and **`QuizViewModel`**.

Figure 4.1 **`MainActivity`** and **`QuizViewModel`** across rotation



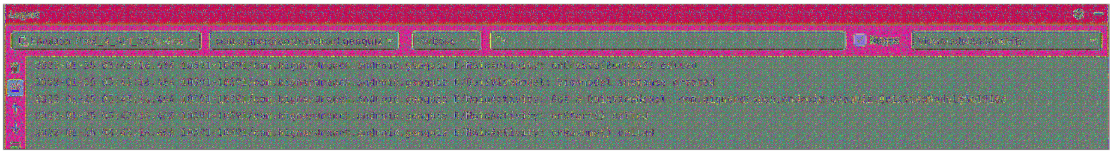

To see this in action, run GeoQuiz. In Logcat, select Edit Filter Configuration in the dropdown to create a new filter. In the Log Tag box, enter QuizViewModel|MainActivity (the two class names with the pipe character "|" between them) to show only logs tagged with either class name. Name the filter ViewModelAndActivity (or another name that makes sense to you) and click OK (Figure 4.2).

Figure 4.2 Filtering **`QuizViewModel`** and **`MainActivity`** logs



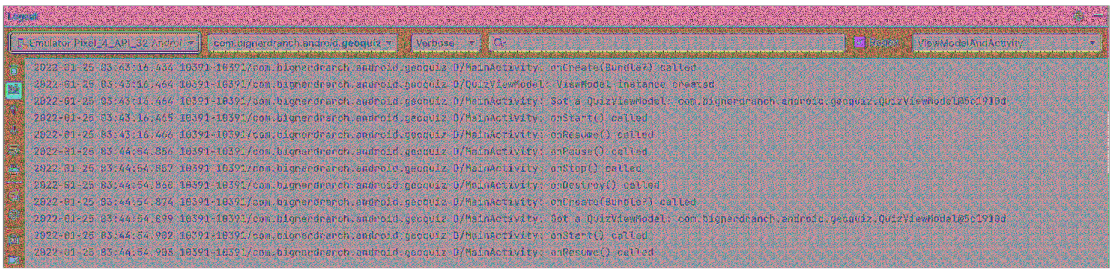
Now look at the logs. When **`MainActivity`** first launches and logs the **`ViewModel`** in **`onCreate(…)`**, a new **`QuizViewModel`** instance is created. This is reflected in the logs (Figure 4.3).

Figure 4.3 **`QuizViewModel`** instance created



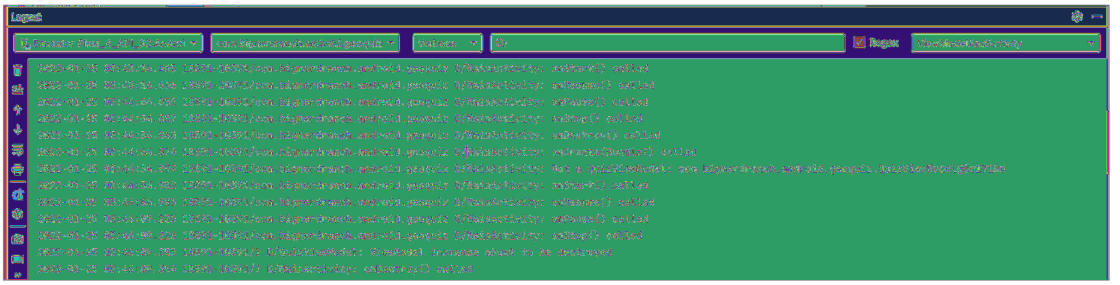
Rotate the device. The logs show the activity is destroyed (Figure 4.4). The **`QuizViewModel`** is not. When the new instance of **`MainActivity`** is created after rotation, it requests a **`QuizViewModel`**. Since the original **`QuizViewModel`** is still in memory, **`viewModels()`** returns that instance rather than creating a new one.

Figure 4.4 **`MainActivity`** is destroyed and re-created; **`QuizViewModel`** persists



Finally, open the overview screen and close the application. **`QuizViewModel.onCleared()`** is called, indicating that the **`QuizViewModel`** instance is about to be destroyed, as the logs show (Figure 4.5). The **`QuizViewModel`** is destroyed, along with the **`MainActivity`** instance.

Figure 4.5 **`MainActivity`** and **`QuizViewModel`** destroyed



The relationship between **MainActivity** and **QuizViewModel** is unidirectional. The activity references the **ViewModel**, but the **ViewModel** does not access the activity. Your **ViewModel** should never hold a reference to an activity or a view, otherwise you will introduce a *memory leak*.

A memory leak occurs when one object holds a strong reference to another object that should be destroyed. The strong reference prevents the garbage collector from clearing the object from memory. Memory leaks due to a configuration change are common bugs. (The details of strong reference and garbage collection are outside the scope of this book. If you are not sure about these concepts, we recommend reading up on them in a Kotlin or Java reference.)

Your **ViewModel** instance stays in memory across rotation, while your original activity instance gets destroyed. If the **ViewModel** held a strong reference to the original activity instance, two problems would occur: First, the original activity instance would not be removed from memory, and thus the activity would be leaked. Second, the **ViewModel** would hold a reference to a stale activity. If the **ViewModel** tried to update the view of the stale activity, it would trigger an **IllegalStateException**.

## Add data to your ViewModel

Now it is finally time to fix GeoQuiz's rotation bug. **QuizViewModel** is not destroyed on rotation the way **MainActivity** is, so you can stash the activity's UI state data in the **QuizViewModel** instance and it, too, will survive rotation.

You are going to cut the question and current index data from your activity and paste them in your **ViewModel**, along with all the logic related to them. Begin by cutting the `currentIndex` and `questionBank` properties from **MainActivity** (Listing 4.4).

### Listing 4.4 Cutting model data from activity (`MainActivity.kt`)

```
class MainActivity : AppCompatActivity() {
    ...
    private val questionBank = listOf(
        Question(R.string.question_australia, true),
        Question(R.string.question_oceans, true),
        Question(R.string.question_mideast, false),
        Question(R.string.question_africa, false),
        Question(R.string.question_americas, true),
        Question(R.string.question_asia, true)
    )

    private var currentIndex = 0
    ...
}
```

Now, paste the `currentIndex` and `questionBank` properties into **`QuizViewModel`**, as shown in Listing 4.5. While you are editing **`QuizViewModel`**, delete the **`init`** and **`onCleared()`** logging, as you will not use them again.

### Listing 4.5 Pasting model data into **`QuizViewModel`** (`QuizViewModel.kt`)

```
class QuizViewModel : ViewModel() {

    init {
        Log.d(TAG, "ViewModel instance created")
    }

    override fun onCleared() {
        super.onCleared()
        Log.d(TAG, "ViewModel instance about to be destroyed")
    }

    private val questionBank = listOf(
        Question(R.string.question_australia, true),
        Question(R.string.question_oceans, true),
        Question(R.string.question_mideast, false),
        Question(R.string.question_africa, false),
        Question(R.string.question_americas, true),
        Question(R.string.question_asia, true)
    )

    private var currentIndex = 0
}
```

Next, add a function to **`QuizViewModel`** to advance to the next question. Also, add computed properties to return the text and answer for the current question.

### Listing 4.6 Adding business logic to **`QuizViewModel`** (`QuizViewModel.kt`)

```
class QuizViewModel : ViewModel() {

    private val questionBank = listOf(
       ...
    )

    private var currentIndex: Int = 0

    val currentQuestionAnswer: Boolean
        get() = questionBank[currentIndex].answer

    val currentQuestionText: Int
        get() = questionBank[currentIndex].textResId

    fun moveToNext() {
        currentIndex = (currentIndex + 1) % questionBank.size
    }
}
```

Earlier, we said that a **`ViewModel`** stores all the data that its associated screen needs, formats it, and makes it easy to access. This allows you to remove presentation logic code, such as the current index, from the activity – which in turn keeps your activity simpler. And keeping activities as simple as possible is a good thing: Any logic you put in your activity might be unintentionally affected by the activity's lifecycle. Also, removing presentation logic means the activity is only responsible for handling what appears on the screen, not the logic behind determining the data to display.

Next, finish cleaning up **`MainActivity`**. You are ready to delete the old computation of `currentIndex`, and you will also make a couple other changes. Since you do not want to directly access **`MainActivity`** from within your **`ViewModel`**, you will leave the **`updateQuestion()`** and **`checkAnswer(Boolean)`** functions in **`MainActivity`** – but you will update them to call through to the new, smarter computed properties in **`QuizViewModel`**.

### Listing 4.7 Updating question through **`QuizViewModel`** (`MainActivity.kt`)

```
class MainActivity : AppCompatActivity() {
    ...
    override fun onCreate(savedInstanceState: Bundle?) {
        ...
        binding.nextButton.setOnClickListener {
            currentIndex = (currentIndex + 1) % questionBank.size
            quizViewModel.moveToNext()
            updateQuestion()
        }
        ...
    }
    ...
    private fun updateQuestion() {
        val questionTextResId = questionBank[currentIndex].textResId
        val questionTextResId = quizViewModel.currentQuestionText
        binding.questionTextView.setText(questionTextResId)
    }

    private fun checkAnswer(userAnswer: Boolean) {
        val correctAnswer = questionBank[currentIndex].answer
        val correctAnswer = quizViewModel.currentQuestionAnswer
        ...
    }
```

Run GeoQuiz, press NEXT, and rotate the device or emulator. No matter how many times you rotate, the newly minted **`MainActivity`** will "remember" what question you were on. Do a happy dance to celebrate solving the UI state loss on rotation bug.

But do not dance too long. There is another, less easily discoverable bug to squash.

## Saving Data Across Process Death

Configuration changes are not the only time the OS can destroy an activity even though the user does not intend it to.

Each app gets its own *process* (more specifically, a Linux process) containing a single thread to execute UI-related work on and a piece of memory to store objects in. An app's process can be destroyed by the OS if the user navigates away for a while and Android needs to reclaim memory. When an app's process is destroyed, all the objects stored in that process's memory are destroyed.

Processes containing resumed or started activities get higher priority than other processes. When the OS needs to free up resources, it will select lower-priority processes first. Practically speaking, a process containing a visible activity will not be reclaimed by the OS. If a foreground process does get reclaimed, that means something is horribly wrong with the device (and your app being killed is probably the least of the user's concerns).

But processes that do not have any activities in the started or resumed state are fair game to be killed. So, for example, if the user navigates to the Home screen and then goes and watches a video or plays a game, your app's process might be killed.

Activities themselves are not individually destroyed in low-memory situations. Instead, Android clears an entire app process from memory, taking any of the app's in-memory activities with it.

When the OS destroys the app's process, any of the app's activities and **`ViewModel`**s stored in memory will be wiped away. And the OS will not be nice about the destruction. There is no guarantee that it will call any of the activity or **`ViewModel`** lifecycle callback functions.

So how can you save UI state data and use it to reconstruct the activity so that the user never even knows the activity was destroyed? One way to do this is to store data in *saved instance state*. Saved instance state is data the OS temporarily stores outside of the activity. You can add values to saved instance state by using a **`SavedStateHandle`**.

Back in the early days of Android, you would be responsible for handling saved instance state like it was a lifecycle callback. Now, you can pass a **`SavedStateHandle`** into your **`ViewModel`** through the constructor. You can use the **`SavedStateHandle`** like a key-value map, storing simple pieces of data like integers and strings. This slides very cleanly into the code you have already written; all you will need to change is to make `currentIndex` a computed property.

Make it happen in `QuizViewModel.kt`.

### Listing 4.8 Storing data in the SavedStateHandle (`QuizViewModel.kt`)

```
private const val TAG = "QuizViewModel"
const val CURRENT_INDEX_KEY = "CURRENT_INDEX_KEY"

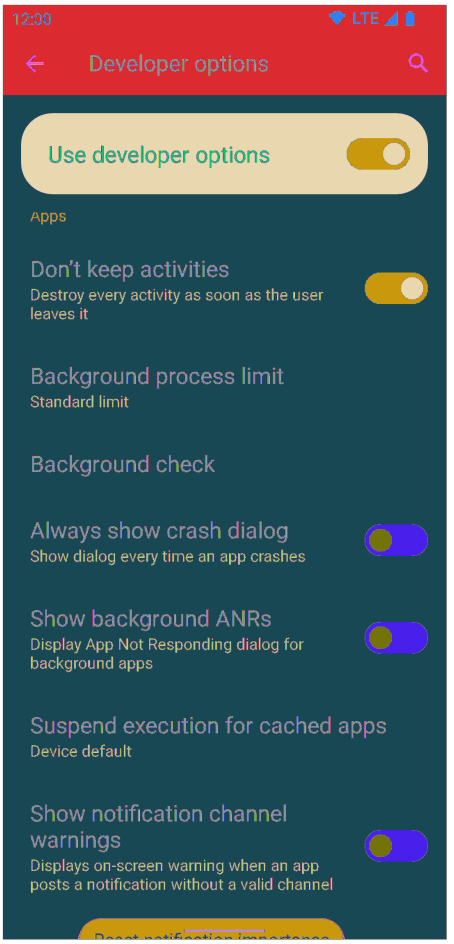
class QuizViewModel(private val savedStateHandle: SavedStateHandle) : ViewModel() {
    ...
    private var currentIndex: Int
        get() = savedStateHandle.get(CURRENT_INDEX_KEY) ?: 0
        set(value) = savedStateHandle.set(CURRENT_INDEX_KEY, value)
    ...
}
```

When you first launch the activity, the value for `currentIndex` in the **`SavedStateHandle`** map is null, so for `currentIndex`'s getter, you provide a default value of 0. This replicates the existing behavior of initializing `currentIndex` to 0.

Rotation is easy to test. And, luckily, so is the low-memory situation. Try it out now to see for yourself.

Connect your hardware device and open up Developer options in its Settings app. (If you do not see Developer options, follow the steps back in Chapter 2 to enable them.) Within Developer options, scroll down to the Apps section and turn on the setting labeled Don't keep activities, as shown in Figure 4.6.

Figure 4.6 Don't keep activities




Now run GeoQuiz on the device, press NEXT to move to another question, and navigate to the Home screen. Returning to the Home screen causes the activity to receive calls to **`onPause()`** and **`onStop()`**, as you know. The logs tell you that the activity has also been destroyed, just as if the Android OS had reclaimed it for its memory.

Restore the app (using the list of apps on the device) to see whether your state was saved as you expected. Pat yourself on the back when GeoQuiz opens to the question you last saw.

Be sure to turn Don't keep activities off when you are done testing, as it will cause a performance decrease.

That is all you have to do to correctly handle process death. You might note that you store `currentIndex` within the **`SavedStateHandle`** and not, say, `questionBank`. **`SavedStateHandle`** has its limitations. The data within **`SavedStateHandle`** is serialized and written to disk, so you should avoid stashing any large or complex objects. You should only use **`SavedStateHandle`** to store the minimal amount of information necessary to re-create the UI state (for example, the current question index).

Neither **`ViewModel`** nor **`SavedStateHandle`** is a solution for long-term storage. If your app needs to store data that should live as long as the app is installed on the device, regardless of your activity's state, use a persistent storage alternative. You will learn about two local persistent storage options in this book: databases, in Chapter 12, and shared preferences, in Chapter 21. In addition to local storage, you could store data on a remote server somewhere. You will learn how to access data from a web server in Chapter 20.

In this chapter, you squashed GeoQuiz's state-loss bugs by correctly accounting for configuration changes and process death. In the next two chapters, you will learn how to use Android Studio's debugging and testing tools to troubleshoot other, more app-specific bugs that might arise and to test your app's functionality. In Chapter 7, you will add a new feature to GeoQuiz: cheating.

## For the More Curious: Jetpack, AndroidX, and Architecture Components

The `androidx.lifecycle` library containing **`ViewModel`** is part of Android Jetpack Components. Android Jetpack Components, called Jetpack for short, is a set of libraries created by Google to make various aspects of Android development easier. You can see a list of the Jetpack libraries at `developer.android.com/jetpack`. You can include any of these libraries in your project by adding the corresponding dependency to your `app/build.gradle` file, as you did in this chapter.

Each Jetpack library is located in a package that starts with `androidx`. For this reason, you will sometimes hear the terms "AndroidX" and "Jetpack" used interchangeably.

Jetpack libraries make up the backbone of most modern Android apps. When you generated GeoQuiz, Android Studio included a few of them by default. As you continue in this book, you will encounter several Jetpack libraries, such as Fragments (Chapter 9), Room (Chapter 12), and WorkManager (Chapter 22).

Some of the Jetpack components are entirely new. Others have been around for a while but were previously lumped into a handful of much larger libraries collectively called the Support Library. If you hear or see anything about the Support Library, know that you should now use the Jetpack (AndroidX) version of that library instead.

## For the More Curious: Avoiding a Half-Baked Solution

Some people try to address the UI state loss on configuration change bug in their app by disabling rotation. If the user cannot rotate the app, they never lose their UI state, right? That is true – but, sadly, this approach leaves your app prone to other bugs. While this smooths over the rough edge of rotation, it leaves other lifecycle bugs that users will surely encounter, but that will not necessarily present themselves during development and testing.

First, there are other configuration changes that can occur at runtime, such as window resizing and night mode changes. And yes, you could also capture and ignore or handle those changes. But this is a bad practice – it disables a feature of the system, which is to automatically select the right resources based on the runtime configuration.

Second, disabling rotation does not solve the process death issue.

If you want to lock your app into portrait or landscape mode *because it makes sense for your app*, you should still program defensively against configuration changes and process death. And you are now equipped to do so with your newfound knowledge of **`ViewModel`** and saved instance state.

In short, dealing with UI state loss by blocking configuration changes is bad form. We are only mentioning it so that you will recognize it as such if you see it out in the wild.

## For the More Curious: Activity and Instance State

**`SavedStateHandle`** is an easy-to-use API that allows you to safely store and retrieve instance state and persist that information even if your process is killed. But it has not always been around. Before **`SavedStateHandle`** was released in 2020, developers were expected to use APIs within **`Activity`**.

To store instance state, developers used the **`Activity.onSaveInstanceState(Bundle)`** function. Similar to **`Activity.onPause()`** and **`Activity.onStop()`**, **`Activity.onSaveInstanceState(Bundle)`** is like a lifecycle callback called during the teardown of an activity. To retrieve instance state, developers used an API you are already familiar with: **`Activity.onCreate(Bundle?)`**. The **`Bundle?`** passed in as a parameter is the saved instance state.

With the adoption of the ViewModel library throughout the Android ecosystem, using these old APIs became awkward and resulted in confusing code. There were multiple places where you had to pass data between your **`ViewModel`** and **`Activity`**, and keeping state consistent between the two was error prone.

With the **`SavedStateHandle`** and **`ViewModel`** classes, you can keep all the instance state business logic within your **`ViewModel`**. That means you can avoid the awkward dance between your **`ViewModel`** and **`Activity`** – and also make your **`Activity`** simpler. So if you have an old codebase and see the old **`Activity`** APIs used to store and retrieve instance state, consider refactoring to use **`SavedStateHandle`** instead.

# 5
# Debugging Android Apps

In this chapter, you will find out what to do when apps get buggy. You will learn how to use Logcat, Android Lint, and the debugger that comes with Android Studio.

To practice debugging, the first step is to break something. In `MainActivity.kt`, comment out the code in **`onCreate(Bundle?)`** where you initialize `binding`.

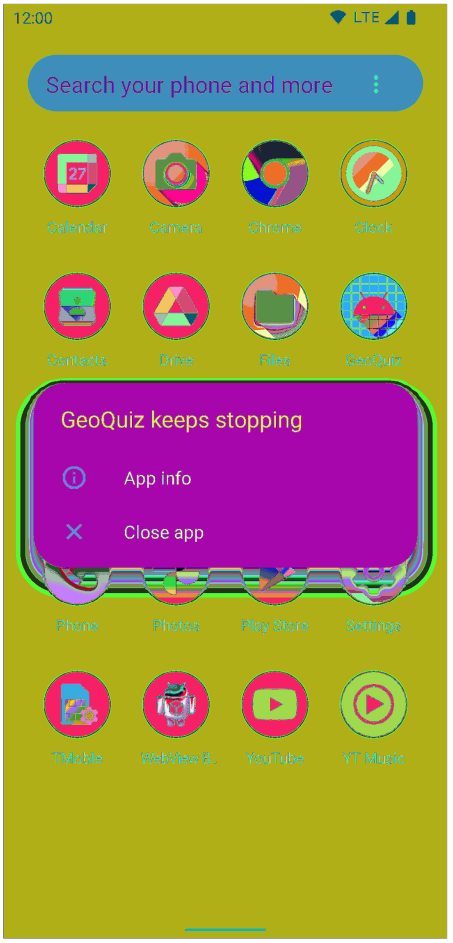
### Listing 5.1 Commenting out a crucial line (`MainActivity.kt`)

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    Log.d(TAG, "onCreate(Bundle?) called")
    // binding = ActivityMainBinding.inflate(layoutInflater)
    ...
}
```

Run GeoQuiz and see what happens. The app will crash and burn almost immediately.

If you watch the screen, you may see the app appear for a brief moment before vanishing without a word. On older versions of Android, you might see a dialog pop up. If you do not, launch the app again by pressing the GeoQuiz icon on the launcher screen. This time, when the app crashes you will see a message like the one shown in Figure 5.1.

Figure 5.1 GeoQuiz is about to E.X.P.L.O.D.E.



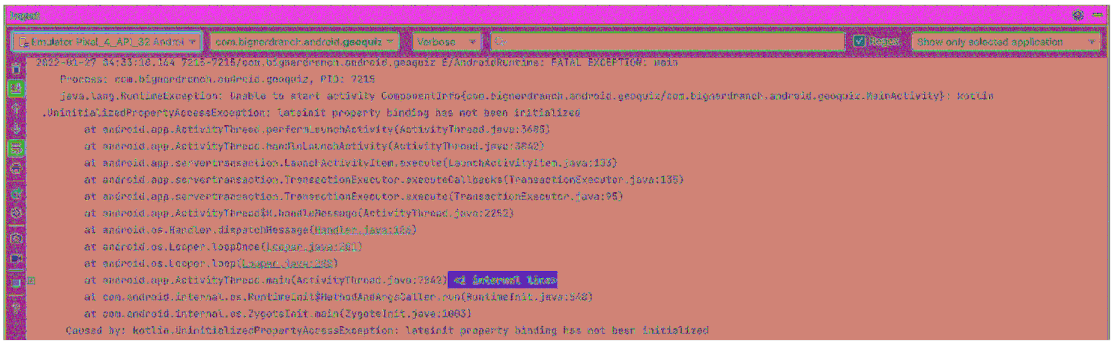

Of course, you know exactly what is wrong with your app. But if you did not, it might help to look at your app from a new perspective.

## Exceptions and Stack Traces

Expand the Logcat tool window so that you can see what has happened. If you scroll up and down in Logcat, you should find an expanse of red, as shown in Figure 5.2. This is a standard `AndroidRuntime` exception report.

If you do not see much in Logcat and cannot find the exception, you may need to select the Show only selected application or No Filters option in the filter dropdown. On the other hand, if you see too much in Logcat, you can adjust the log level from Verbose to Error, which will show only the most severe log messages. You can also search for the text fatal exception, which will bring you straight to the exception that caused the app to crash.

Figure 5.2 Exception and stack trace in Logcat




The report tells you the top-level exception and its stack trace, then the exception that caused that exception and *its* stack trace, and so on until it finds an exception with no cause.

It may seem strange to see a `java.lang` exception in the stack trace, since you are writing Kotlin code. When building for Android, Kotlin code is compiled to the same kind of low-level bytecode Java code is compiled to. During that process, many Kotlin exceptions are mapped to `java.lang` exception classes through type-aliasing. **`kotlin.RuntimeException`** is the superclass of **`kotlin.UninitializedPropertyAccessException`**, and it is aliased to **`java.lang.RuntimeException`** when running on Android.

In most of the code you will write, the last exception in the Logcat report – the one with no cause – is the interesting one. Here, the exception without a cause is a **`kotlin.UninitializedPropertyAccessException`**. The line just below this exception is the first line in its stack trace. This line tells you the class and function where the exception occurred as well as what file and line number the exception occurred on. Click the blue link, and Android Studio will take you to that line in your source code.

The line you are taken to is the first use of the `binding` variable, inside **`onCreate(Bundle?)`**. The name **`UninitializedPropertyAccessException`** gives you a hint to the problem: This variable was not initialized.

Uncomment the line initializing `binding` to fix the bug.

### Listing 5.2 Uncommenting a crucial line (`MainActivity.kt`)

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    Log.d(TAG, "onCreate(Bundle?) called")
    // binding = ActivityMainBinding.inflate(layoutInflater)
    ...
}
```

When you encounter runtime exceptions, remember to look for the last exception in Logcat and the first line in its stack trace that refers to code that you have written. That is where the problem occurred, and it is the best place to start looking for answers.

If a crash occurs while a device is not plugged in, all is not lost. The device will store the latest lines written to the log. The length and expiration of the stored log depends on the device, but you can usually count on retrieving log results within 10 minutes. Just plug in the device and select it in the Devices view. Logcat will fill itself with the stored log.

## Diagnosing misbehaviors

Problems with your apps will not always be crashes. In some cases, they will be misbehaviors. For example, suppose that nothing happened any time you pressed the NEXT button. That would be a noncrashing, misbehaving bug.

In `QuizViewModel.kt`, comment out the code in the **`moveToNext()`** function that increments the current question index.

### Listing 5.3 Forgetting a critical line of code (`QuizViewModel.kt`)

```
fun moveToNext() {
    // currentIndex = (currentIndex + 1) % questionBank.size
}
```

Run GeoQuiz and press the NEXT button. You should see no effect.

This bug is trickier than the last bug. It is not throwing an exception, so fixing the bug is not a simple matter of making the exception go away. On top of that, this misbehavior could be caused in two different ways: The index might not be changed, or code to update the UI might not be called.

You know what caused this bug, because you just introduced it intentionally. But if this type of bug popped up on its own and you had no idea what was causing the problem, you would need to track down the culprit. In the next few sections, you will see two ways to do this: diagnostic logging of a stack trace and using the debugger to set a breakpoint.

## Logging stack traces

In **`QuizViewModel`**, add a log statement to **`moveToNext()`**.

Listing 5.4 **`Exception`** for fun and profit (`QuizViewModel.kt`)

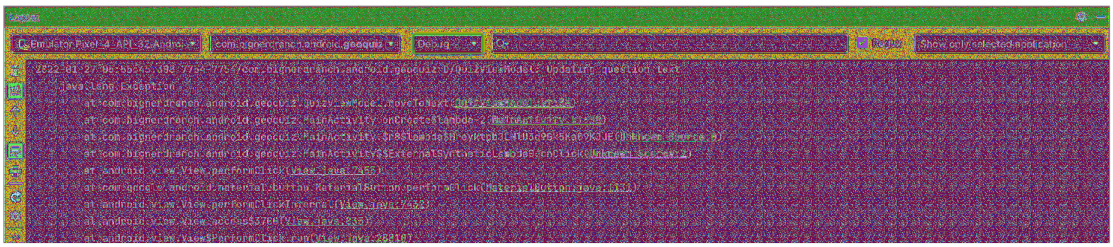
```
fun moveToNext() {
    Log.d(TAG, "Updating question text", Exception())
    // currentIndex = (currentIndex + 1) % questionBank.size
}
```

The **`Log.d(String, String, Throwable)`** version of **`Log.d`** logs the entire stack trace, like the **`UninitializedPropertyAccessException`** you saw earlier. The stack trace will tell you where the call to **`moveToNext()`** was made.

The exception that you pass to **`Log.d(String, String, Throwable)`** does not have to be a thrown exception that you caught. You can create a brand new **`Exception`** and pass it to the function without ever throwing it, and you will get a report of where the exception was created.

Run GeoQuiz, press the NEXT button, and then check the output in Logcat (Figure 5.3).

Figure 5.3 The results



The top line in the stack trace is the line where you logged out the **`Exception`**. On the next line, you can see where **`moveToNext()`** was called from within your **`MainActivity`**. Click the link on this line, and you will be taken to where you commented out the line to increment your question index. But do not get rid of the bug; you are going to use the debugger to find it again in a moment.

Logging out stack traces is a powerful tool, but it is also a verbose one. Leave a bunch of these hanging around, and soon Logcat will be an unmanageable mess. Also, a competitor might steal your ideas by reading your stack traces to understand what your code is doing.

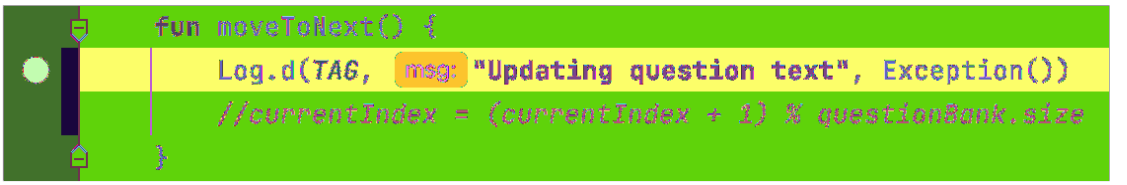
On the other hand, sometimes a stack trace showing what your code does is exactly what you need. If you are seeking help with a problem at `stackoverflow.com` or `forums.bignerdranch.com`, it often helps to include a stack trace. You can copy and paste lines directly from Logcat.

## Setting breakpoints

Now you will use the debugger that comes with Android Studio to track down the same bug. You will set a *breakpoint* in **`moveToNext()`** to see whether it was called. A breakpoint pauses execution before the line executes and allows you to examine line by line what happens next.

In `QuizViewModel.kt`, return to the **`moveToNext()`** function. Next to the first line of this function, click the gray gutter area in the lefthand margin. You should see a red circle in the gutter like the one shown in Figure 5.4. This is a breakpoint.

Figure 5.4 A breakpoint




To engage the debugger and trigger your breakpoint, you need to debug your app instead of running it. To debug your app, click the Debug 'app' button (Figure 5.5). You can also navigate to Run → Debug 'app' in the menu bar. Your device will report that it is waiting for the debugger to attach, and then it will proceed normally.

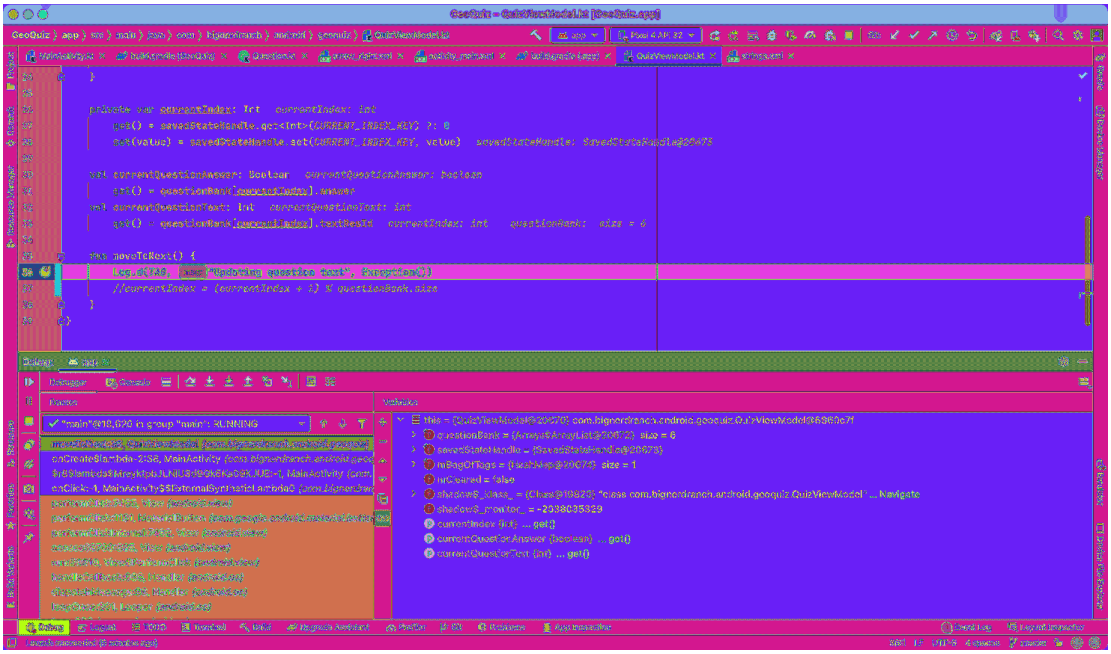
Figure 5.5 Debug app buttons




In some circumstances, you may want to debug a running app without relaunching it. You can attach the debugger to a running application by clicking the Attach Debugger to Android Process button shown in Figure 5.5 or by navigating to Run → Attach to process.... Choose your app's process on the dialog that appears and click OK, and the debugger will attach. Note that breakpoints are only active when the debugger is attached, so any breakpoints that are hit before you attach the debugger will be ignored.

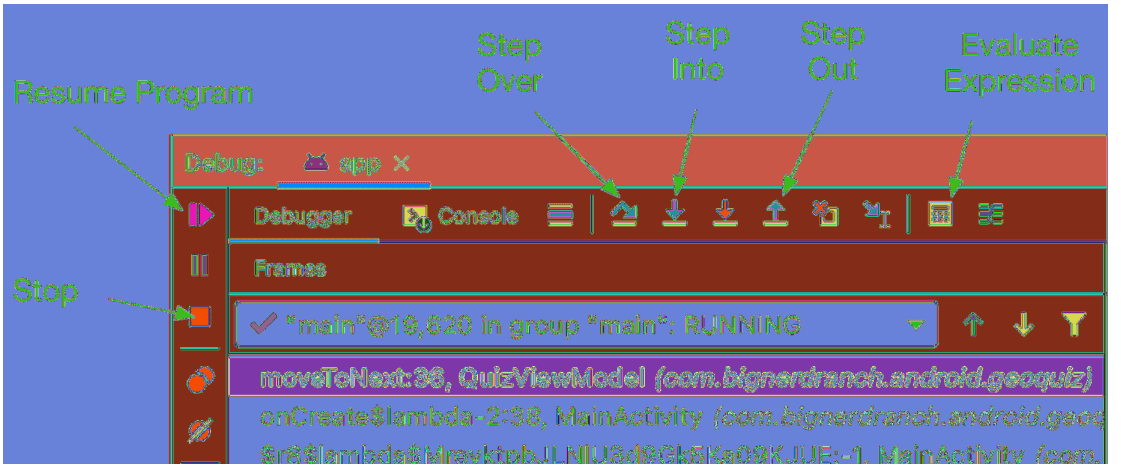
Click the NEXT button. In Figure 5.6, you can see that `QuizViewModel.kt` is now open in the editor and that the line with the breakpoint where execution has paused is highlighted. The debug tool window at the bottom of the screen is now visible. It contains the Frames and Variables views. (If the debug tool window did not open automatically, you can open it by clicking the Debug tool window bar at the bottom of the Android Studio window.)

Figure 5.6 Stop right there!



You can use the arrow buttons at the top of the debug tool window (Figure 5.7) to step through your program. You can use the Evaluate Expression button to execute simple Kotlin statements on demand during debugging, which is a powerful tool.

Figure 5.7 Debug tool window controls

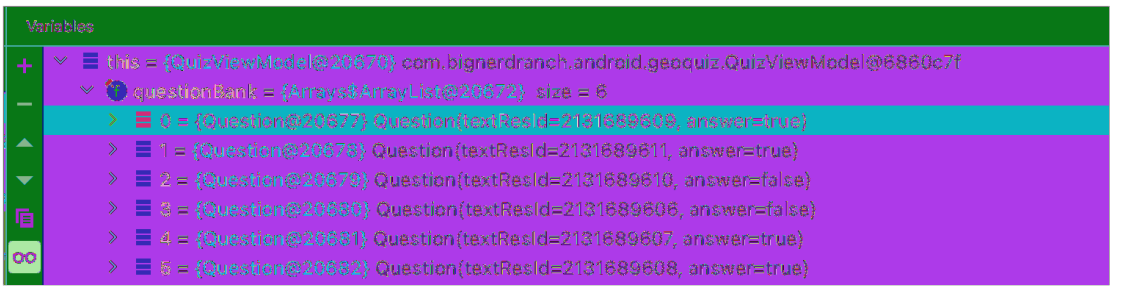



Now that you are stopped at an interesting point of execution, you can take a look around. The Variables view allows you to examine the values of the objects in your program. At the top, you should see the value `this` (the **`QuizViewModel`** instance itself).

Expand the `this` variable to see all the variables declared in **`QuizViewModel`** and in **`QuizViewModel`**'s superclass (**`ViewModel`**). For now, focus on the variables that you created.

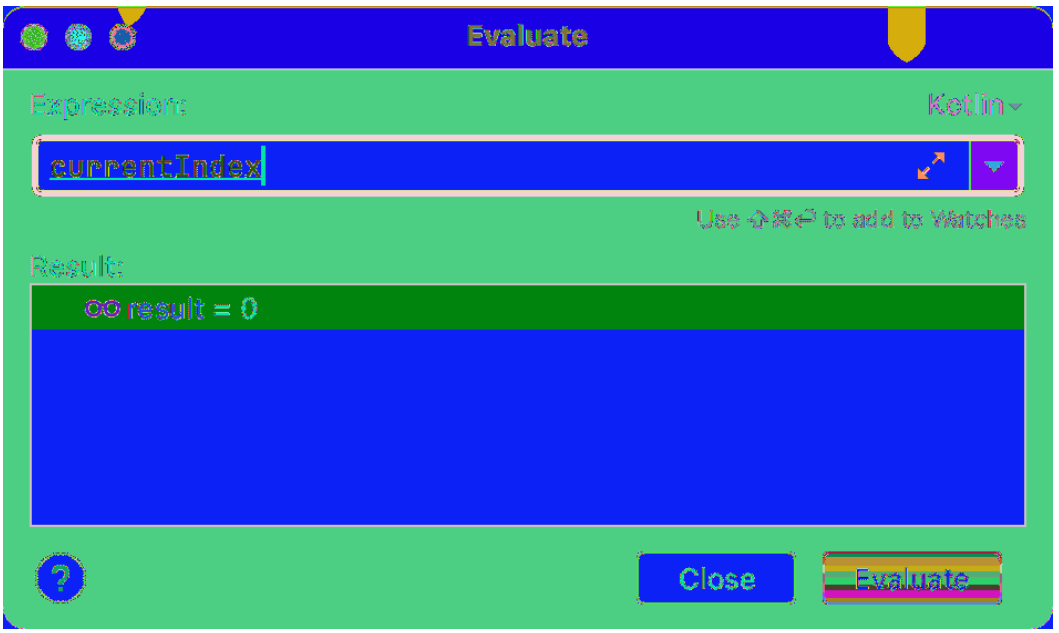
You are only interested in one value – `currentIndex` – but it is not here. That is because `currentIndex` is a computed property. Note that you also do not see `currentQuestionAnswer` or `currentQuestionText`. But `questionBank` is there. Expand it and look at each one of its **`Question`**s (Figure 5.8).

Figure 5.8 Inspecting variable values at runtime



Even though you do not see `currentIndex`, you can still access it. Click the Evaluate Expression button within the debug tool window. In the Expression: text field, enter currentIndex and press the Evaluate button (Figure 5.9).

Figure 5.9 Evaluating the current index




The debugger will evaluate and print the current value of `currentIndex`. You pressed the NEXT button, which should have resulted in `currentIndex` being incremented from `0` to `1`. So you would expect `currentIndex` to have a value of `1`. However, as shown in Figure 5.9, `currentIndex` still has a value of `0`.

Close the Evaluate window. As you already knew, the problematic behavior results from the code within **`QuizViewModel.moveToNext()`** never being called (because you commented it out). You will want to fix this implementation – but before you make any changes to code, you should stop debugging your app. If you edit your code while debugging, the code running with the debugger attached will be out of date compared to what is in the editor tool window, so the debugger can show misleading information compared to the updated code.

You can stop debugging in two ways: You can stop the program, or you can simply disconnect the debugger. To stop the program, click the Stop button shown in Figure 5.7.

Now return your **QuizViewModel** to its former glory. You are done with the log message (and the TAG constant), so delete them to keep your file tidy. Also, remove the breakpoint you set by clicking it in the gutter.

### Listing 5.5 Returning to normalcy (QuizViewModel.kt)

```
const val CURRENT_INDEX_KEY = "CURRENT_INDEX_KEY"
~~private const val TAG = "QuizViewModel"~~

class QuizViewModel(private val savedStateHandle: SavedStateHandle) : ViewModel() {
    ...
    fun moveToNext() {
        ~~Log.d(TAG, "Updating question text", Exception())~~
        ~~//~~ currentIndex = (currentIndex + 1) % questionBank.size
    }
}
```

You have tried two ways of tracking down a misbehaving line of code: stack trace logging and setting a breakpoint in the debugger. Which is better? Each has its uses, and one or the other will probably end up being your favorite.

Logging stack traces has the advantage that you can see stack traces from multiple places in one log. The downside is that to learn something new you have to add new log statements, rebuild, deploy, and navigate through your app to see what happened.

The debugger is more convenient. If you run your app with the debugger attached (or attach the debugger to the application's process after it has started), then you can set a breakpoint while the application is running and poke around to get information about multiple issues.

# Android-Specific Debugging

Most Android debugging is just like Kotlin debugging. However, you will sometimes run into issues with Android-specific parts, such as resources, that the Kotlin compiler knows nothing about. In this section, you will learn about Android Lint and issues with the build system.

## Using Android Lint

Android Lint (or just "Lint") is a *static analyzer* for Android code. A static analyzer is a program that examines your code to find defects without running it. Lint uses its knowledge of the Android frameworks to look deeper into your code and find problems that the compiler cannot. In many cases, Lint's advice is worth taking.

In Chapter 8, you will see Lint warn you about compatibility problems. Lint can also perform type-checking for objects that are defined in XML.

You can manually run Lint to see all the potential issues in your project, including those that are less serious. In fact, let's add a small issue to the project. Suppose you did not like the question text being centered, so you decided to left-align it. Open `activity_main.xml` and make that change.

### Listing 5.6 Shifting the text left (`activity_main.xml`)

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

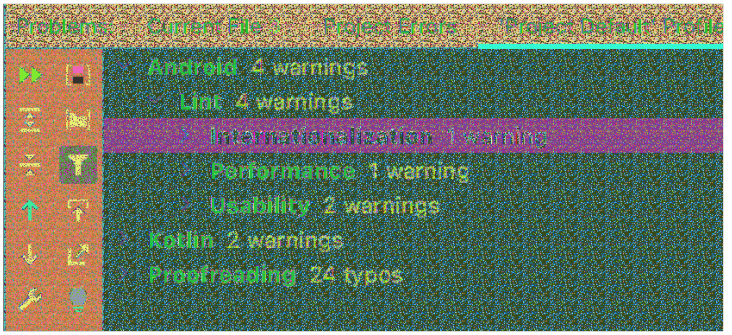
    <TextView
        android:id="@+id/question_text_view"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        ~~android:gravity="center"~~
        android:gravity="left"
        android:padding="24dp"
        tools:text="@string/question_australia"/>

    ...
</LinearLayout>
```

Once you have made that change, select Analyze → Inspect Code... from the menu bar. You will be asked which parts of your project you would like to inspect. Choose Whole project and click OK. Android Studio will run Android Lint as well as a few other static analyzers on your code, such as spelling and Kotlin checks.

When the scan is complete, you will see categories of potential issues in the inspection tool window. Expand the Android and Lint categories to see Lint's information about your project (Figure 5.10).

Figure 5.10 Lint warnings




(Do not be concerned if you see a different number of Lint warnings. The Android toolchain is constantly evolving, and new checks may have been added to Lint, new restrictions may have been added to the Android framework, and newer versions of tools and dependencies may have become available.)

Expand Internationalization and then, under it, expand Bidirectional Text to see more detailed information on this issue in your project. Click Using left/right instead of start/end attributes to learn about this particular warning (Figure 5.11).

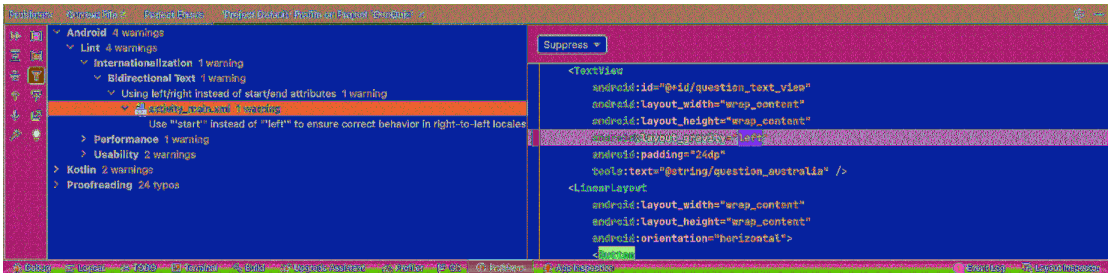
Figure 5.11 Lint warning description



Lint is warning you that using `right` and `left` values for layout attributes could be problematic if your app is used on a device set to a language that reads from right to left instead of left to right. (You will learn about making your app ready for international use in Chapter 18.)

Dig further to see which file and line or lines of code caused the warning. Expand Using left/right instead of start/end attributes. Click the offending file, `activity_main.xml`, to see the snippet of code with the problem (Figure 5.12).

Figure 5.12 Viewing the code that caused the warning




Double-click the warning description that appears under the filename. This will open `activity_main.xml` in the editor tool window and place the cursor on the line causing the warning (which, not coincidentally, is the change you just made).

Undo the change you made and rerun Lint to confirm that the bidirectional text issue you just fixed is no longer listed in the Lint results. For the most part, your app will execute just fine even if you do not address the things Lint warns you about. Often, though, addressing Lint warnings can help prevent problems in the future or make your users' experience better.

We recommend you take all Lint warnings seriously, even if you ultimately decide not to address them. Otherwise you could get used to ignoring Lint and miss a serious problem.

The Lint tool provides detailed information about each issue it finds and provides suggestions for how to address it. We leave it to you as an exercise to review the other issues Lint found in GeoQuiz. You can ignore the issues, fix them as Lint recommends, or use the Suppress button in the problem description pane to suppress the warnings in the future. For the remainder of the GeoQuiz chapters, we will assume you left the remaining Lint issues unaddressed.

## Build issues

Everyone will eventually make a mistake while coding: a forgotten piece of punctuation here, a typo there. It is very common to have a build error happen while you are trying to run your app. Sometimes these build errors will persist or appear seemingly out of nowhere. If this happens to you, here are some things you can try:

*Recheck the validity of the XML in your resource files*

Gradle does a good job of surfacing errors to developers in an actionable way. However, there are times when your app will not compile but Gradle gives you nothing to work with. Often, this is caused by a typo in one of your XML files. Layout XML is not always validated, so typos in these files may not be pointedly brought to your attention. Finding the typo and resaving the file should fix the issue.

*Clean your build*

Select Build → Clean Project. Android Studio will rebuild the project from scratch, which often results in an error-free build. We can all use a deep clean every now and then.

*Sync your project with Gradle*

If you make changes to your `build.gradle` files, you will need to sync those changes to update your project's build settings. Select File → Sync Project with Gradle Files. Android Studio will rebuild the project from scratch with the correct project settings, which can help to resolve issues after changing your Gradle configuration.

*Run Android Lint*

Pay close attention to the warnings from Lint. With this tool, you will often discover unexpected issues.

*Clean your project*

If you have gotten this far down the debugging trail, things are not good. On very rare occasions, clearing out the caches that Android Studio uses could help solve your problem. Select File → Invalidate Caches/Restart.... Android Studio will perform some maintenance on the project and restart itself when it is done.

If you are still having problems with resources (or are having different problems), give the error messages and your layout files a fresh look. It is easy to overlook mistakes in the heat of the moment. Check out any Lint errors and warnings as well. A cool-headed reconsideration of the error messages may turn up a bug or typo.

Finally, if you are stuck or having other issues with Android Studio, check the archives at `stackoverflow.com` or visit the forum for this book at `forums.bignerdranch.com`.

## Challenge: Using Conditional Breakpoints

Breakpoints are a very useful tool in debugging, but sometimes you hit your breakpoints too often during execution and they become a burden rather than benefit. In these instances, you can use a conditional breakpoint to limit the number of times execution pauses. You can access the dialog to set up a conditional breakpoint by right-clicking an existing breakpoint. Try pausing execution within the `updateQuestion()` function within **`MainActivity`** only when the answer to the current question is "true."

## Challenge: Exploring the Layout Inspector

For support debugging layout file issues, the layout inspector can be used to interactively inspect how a layout file is rendered to the screen. To use the layout inspector, make sure GeoQuiz is running in the emulator and select Tools → Layout Inspector from the menu bar. Once the inspector is activated, you can explore the properties of your layout by clicking the elements within the layout inspector view.

## Challenge: Exploring the Profiler

The profiler tool window creates detailed reports for how your application is using an Android device's resources, such as CPU and memory. It is useful when assessing and tuning the performance of your app.

To view the profiler tool window, run your app on a connected Android device or emulator and select View → Tool Windows → Profiler from the menu bar. Once the profiler is open, you can see a timeline with sections for CPU, memory, network, and energy.

Click into a section to see more details about your app's usage of that resource. On the CPU view, make sure to hit the Record button to capture more information about CPU usage. After you have performed any interactions with your app that you would like to record, hit the Stop button to stop the recording.

# 6
# Testing

Up to this point, whenever you have made a change in GeoQuiz, you have been forced to compile and deploy an updated version of your app. After waiting for the updated app to install on your device, you can finally interact with the app and observe the new changes. This is a relatively slow process, even for a small app, and it will only get slower as your app becomes more complex.

*Unit testing* is the practice of writing and using small programs to verify the standalone behavior of a unit of code within your app. It can speed up the cycle of developing new features and then verifying that they work as expected. And as your app gains more functionality, unit testing can give you more confidence that your existing functionality remains intact, preventing regressions in behavior – bugs where features that previously worked stop working.

In this chapter, you will get your feet wet by writing some tests to validate existing functionality within GeoQuiz.

## Two Types of Tests

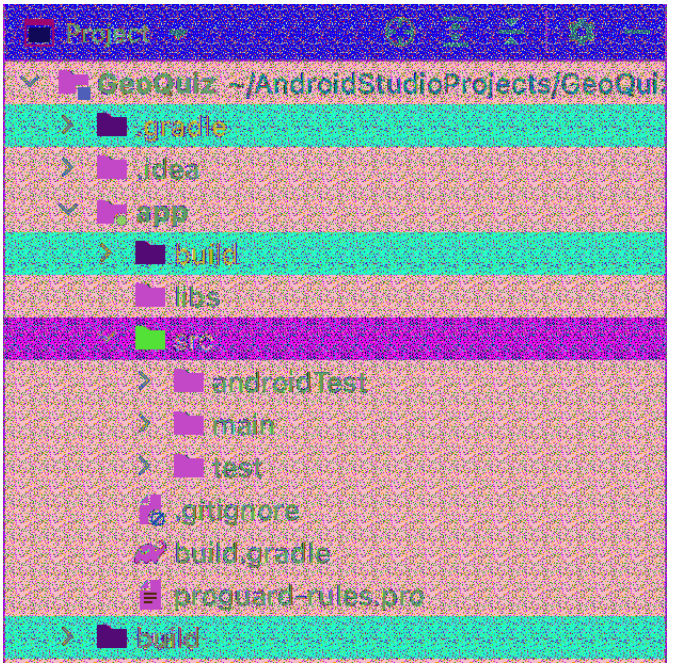
Unit tests on Android fall into two buckets: *JVM tests* and *instrumented tests*. JVM tests execute on your development machine (that is, your laptop or desktop) through a JVM. Instrumented tests execute directly on an Android device. Each type of test has benefits and downsides, so you will find that you use both depending on what you need at a particular point in development.

JVM tests can complete execution in milliseconds, while instrumented tests are orders of magnitude slower and can take seconds to complete. On the flip side, since instrumented tests execute directly on the device that runs your app, you can be confident that the tests accurately represent what will happen when users interact with your app. Additionally, without workarounds, you can only interact with the Android SDK (classes like **`Activity`**, **`TextView`**, and so on) through instrumented tests.

The different test types reside in different *source sets* in the codebase. With Gradle, different groups of code – source sets – are compiled for different situations.

In Android Studio, switch your project tool window to the Project view so you can see GeoQuiz's directory structure. Expand the `GeoQuiz/app/src/` directory. It has three subdirectories: `androidTest`, `main`, and `test [unitTest]`.

Figure 6.1 A look at the `src` subdirectories




These three subdirectories are source sets. So far, you have been writing code in the `main` source set. It holds the code that is compiled and packaged when you install the app on an Android device. JVM tests are located in the `test` source set. Instrumented tests are located in the `androidTest` source set.

Switch back to the Android view. The three source sets are still there, under the `java` directory, but they are labeled differently. The directory with just your package name, `com.bignerdranch.android.geoquiz`, is the `main` source set. The two testing source sets are highlighted green and have their names in parentheses after the package name: `com.bignerdranch.android.geoquiz (androidTest)` for instrumented tests and `com.bignerdranch.android.geoquiz (test)` for JVM tests.

# JVM Tests

Let's look at a JVM test to see how it is structured. When Android Studio generated GeoQuiz's project files, it also generated some unit tests for you. In `com.bignerdranch.android.geoquiz (test)`, find and open `ExampleUnitTest.kt`:

```
class ExampleUnitTest {
    @Test
    fun addition_isCorrect() {
        assertEquals(4, 2 + 2)
    }
}
```

Both JVM and instrumented tests are executed using the *JUnit* testing framework. JUnit is the most popular way of unit testing code in Java and Kotlin and is widely supported on Android.

JUnit tests are encapsulated by classes. Within these classes, individual tests are functions marked by the `@Test` annotation, which you can see in the example. When running your tests, JUnit finds and executes the annotated functions.

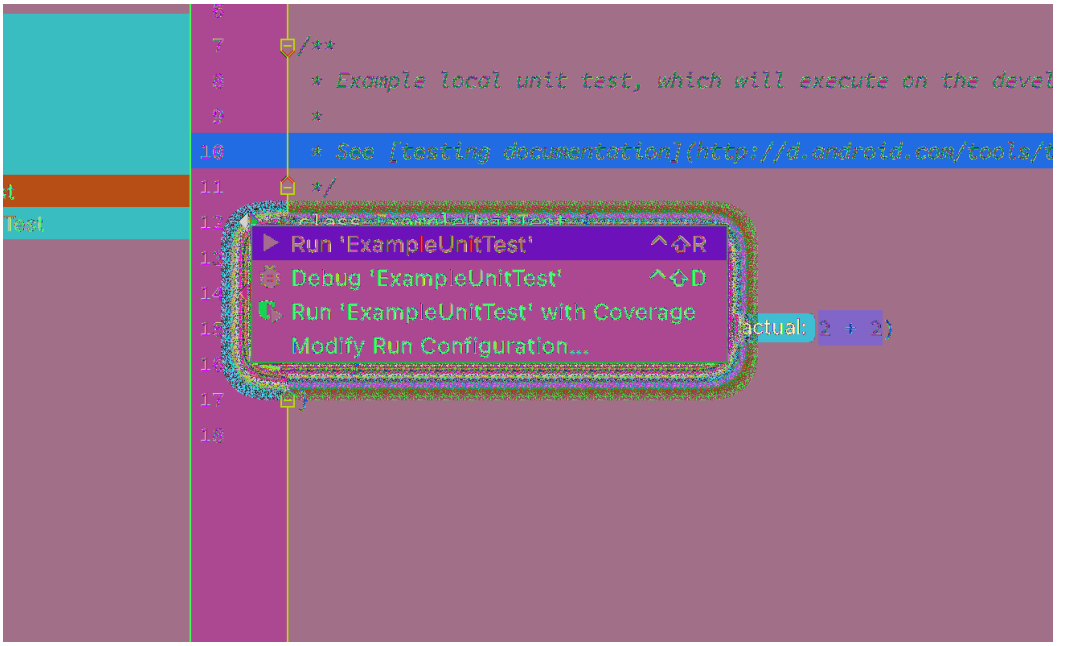
The normal rules for naming functions do not apply to test functions. In fact, names for test functions should be descriptive and verbose. We generally prefer names that describe the behavior we are trying to verify. The name of the example test, **`addition_isCorrect()`**, clearly shows what it is designed to check. It is a simple test that checks an expected value against an operation – here, `4` and `2 + 2`.

To perform this check, **`addition_isCorrect`** uses the **`assertEquals()`** *assertion*. With JUnit, you can assert that two values are equal, as in this example, or that one value is `true`, or any of several other conditions. You can also perform multiple assertions within a single test.

JUnit uses these assertions to determine whether your test passes or fails: If any of the assertions fails, then the entire test fails. Here, **`assertEquals()`** function takes in two parameters: `4` and `2 + 2`. Since those expressions evaluate to the same value, the assertion passes – and, as a result, the test passes.

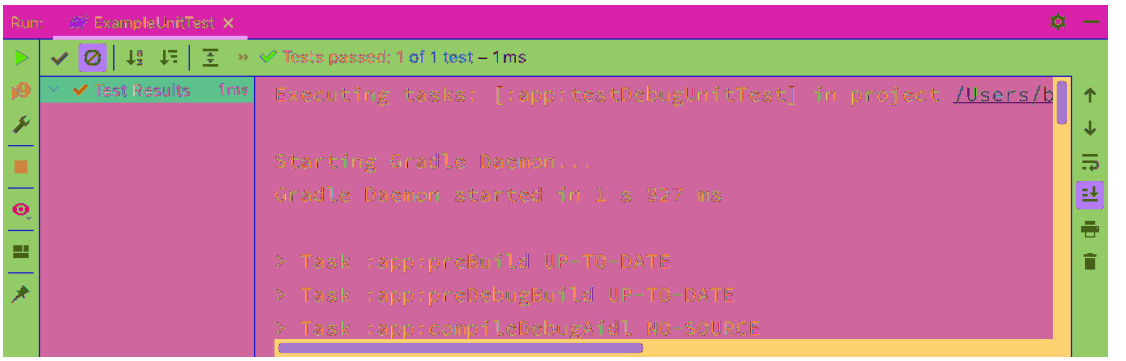
On the left side of the editor tool window, in the gray area known as the gutter, you should see a  icon on the same line as the class definition. Click that icon and then, in the pop-up, click Run 'ExampleUnitTest' (Figure 6.2).

Figure 6.2 Running your unit test




After compiling your code, Android Studio will open the run tool window and execute the unit tests for the class. When the work is complete, you will see that your test passed (Figure 6.3).

Figure 6.3 The results of running your unit test




The `Tests passed: 1` result in the lower-left corner of the Android Studio window indicates the test's successful result. To see what a failing test looks like, try changing the `4` to a `5`. Run the test again. Make sure to revert your changes before moving on.

Unit tests can be quick and easy ways to verify that your code is behaving the way you expect. And JVM tests like **`addition_isCorrect()`** execute *very* quickly: The time required will vary based on the horsepower of your development machine, but it could take as little as a millisecond to complete execution.

Now it is time for you to write your own test. A common pattern is to group and name tests based on the class that they are testing, so, for example, **`MainActivity`** would have its associated tests in a class called **`MainActivityTest`**. The first tests you are going to write will verify the behavior of **`QuizViewModel`**.

You could create a new class file within the `test` source set and set up a **`QuizViewModelTest`** class definition yourself, but Android Studio can help you with this common task. In the project tool window, find and open `QuizViewModel.kt`. Now, in the editor tool window, place your cursor anywhere in the class definition and press Command-Shift-T (Ctrl-Shift-T). In the pop-up, select Create New Test... (Figure 6.4).

Figure 6.4 Creating a new test




This opens the Create Test dialog (Figure 6.5). The defaults for the name (the class name plus “Test”) and the destination package are fine, and you will not be using any of the checkbox options. But there is one field you do need to change: Select JUnit4 from the Testing library dropdown. (JUnit4 is the testing framework Google supports for Android.)

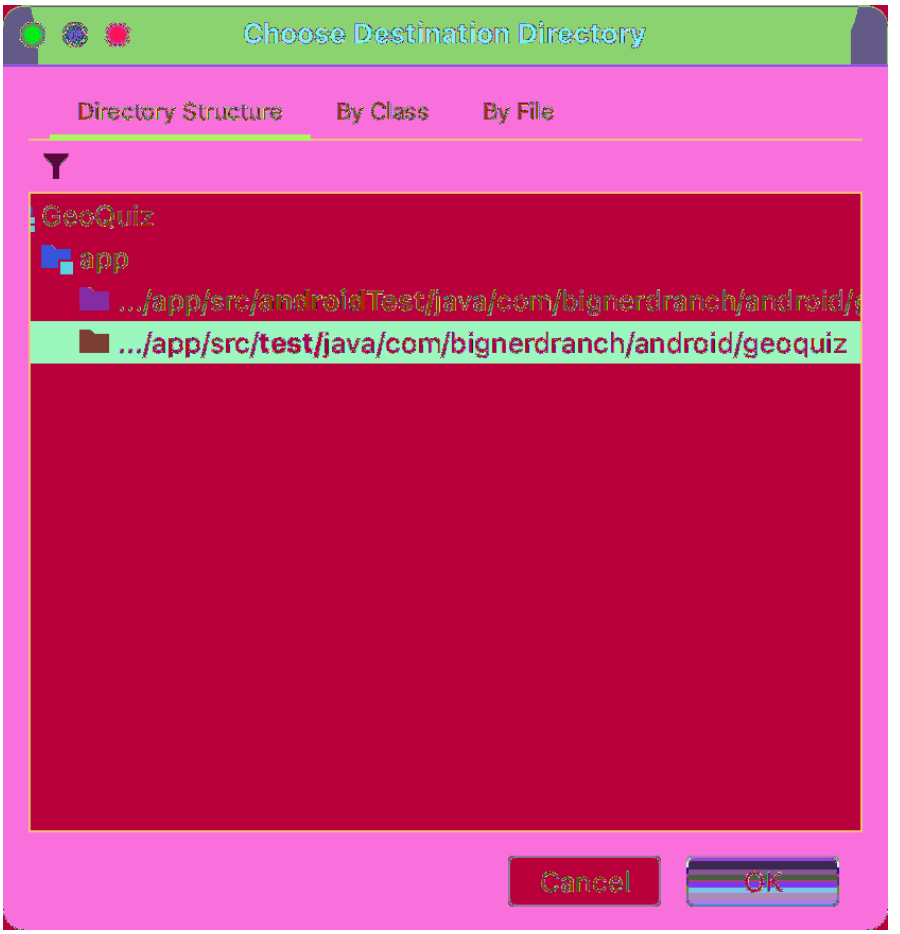
Figure 6.5 Creating your own unit test



(If you see JUnit4 library not found in the module, click the Fix button and wait a moment for the library to be added and synced.)

Click OK. The next dialog asks you to choose a directory for your new test file. Since the code you are testing does not interact with the core Android SDK, you can create a JVM test. Select the folder with `/test/` in its path and click OK (Figure 6.6).

Figure 6.6 The test directory for JVM tests




Your new `QuizViewModelTest.kt` file will open in the editor:

```
package com.bignerdranch.android.geoquiz

import org.junit.Assert.*

class QuizViewModelTest
```

As in **`ExampleUnitTest`**, you are going to write functions with the `@Test` annotation. However, your functions will be slightly more complex than the **`addition_isCorrect()`** example.

Within a unit test, code is normally organized in three distinct phases: first, you *set up* your testing environment, then you *test* a specific unit of code, and finally you *verify* that the unit of code behaves the way you expect. (You will sometimes see this pattern expressed as *given*, *when*, *then*.)

**`addition_isCorrect()`** is a very basic test. It requires no setup, and it tests and verifies behavior on a single line. Your tests will *set up* by directly initializing a **`QuizViewModel`**, *test* by performing some action on your **`QuizViewModel`**, and then *verify* by confirming that the output is what you expect.

The first test you will write will verify that your **`QuizViewModel`** provides the correct question text for the first question just after it is initialized.

Recall that **`QuizViewModel`**'s only constructor parameter is the **`SavedStateHandle`**. You will first need to initialize a saved instance state, which you can do with just an empty constructor, so that you can initialize a **`QuizViewModel`**. Then you can use the same **`assertEquals()`** function you saw in the example to verify that the `currentQuestionText` property on your **`QuizViewModel`** provides the expected value.

Write your test, using the descriptive name **`providesExpectedQuestionText()`**:

### Listing 6.1 Writing your first JVM test (`QuizViewModelTest.kt`)

```
import androidx.lifecycle.SavedStateHandle
import org.junit.Assert.assertEquals
import org.junit.Test

class QuizViewModelTest {
    @Test
    fun providesExpectedQuestionText() {
        val savedStateHandle = SavedStateHandle()
        val quizViewModel = QuizViewModel(savedStateHandle)
        assertEquals(R.string.question_australia, quizViewModel.currentQuestionText)
    }
}
```

Run the test by clicking the  icon next to **`QuizViewModelTest`** and verify that it passes. Note that this test includes set-up and verify phases, but no test behavior. Your next test will include all three phases.

The empty constructor is not the only one available to **`SavedStateHandle`**. You can also pass an initial saved instance state as a map of key-value pairs into the **`SavedStateHandle`** constructor. Take advantage of this functionality to write a test verifying the expected behavior when you are at the end of the question bank and move to the next question: It should wrap around to the first question.

### Listing 6.2 Passing input to your **`QuizViewModel`** (`QuizViewModelTest.kt`)

```
class QuizViewModelTest {
    ...
    @Test
    fun wrapsAroundQuestionBank() {
        val savedStateHandle = SavedStateHandle(mapOf(CURRENT_INDEX_KEY to 5))
        val quizViewModel = QuizViewModel(savedStateHandle)
        assertEquals(R.string.question_asia, quizViewModel.currentQuestionText)
        quizViewModel.moveToNext()
        assertEquals(R.string.question_australia, quizViewModel.currentQuestionText)
    }
}
```

Run both your tests and verify that they pass.

The ability to create instances of a **`ViewModel`** and pass data in as constructor parameters allows you to write useful and reliable unit tests. This is one of the many reasons we recommend keeping business logic in **`ViewModel`**s rather than Android components like **`Activity`**.

## Instrumented Tests with Espresso and ActivityScenario

Let's move on to instrumented tests. Begin by checking out the example that Android Studio created. In `com.bignerdranch.android.geoquiz (androidTest)`, find and open `ExampleInstrumentedTest.kt`:

```
@RunWith(AndroidJUnit4::class)
class ExampleInstrumentedTest {
    @Test
    fun useAppContext() {
        // Context of the app under test.
        val appContext = InstrumentationRegistry.getInstrumentation().targetContext
        assertEquals("com.bignerdranch.android.geoquiz", appContext.packageName)
    }
}
```

Much of this code is similar to the tests you have seen so far: You have a class containing a function annotated with `@Test`, and within that function there is an assertion to verify some behavior. But there are also some differences: First, the class itself has an annotation, `@RunWith(AndroidJUnit4::class)`, which signals to JUnit that this test should be executed on an Android device. And the test function relies on the Android SDK, specifically to verify that the app's package name is the same as the value you set when you created the app.

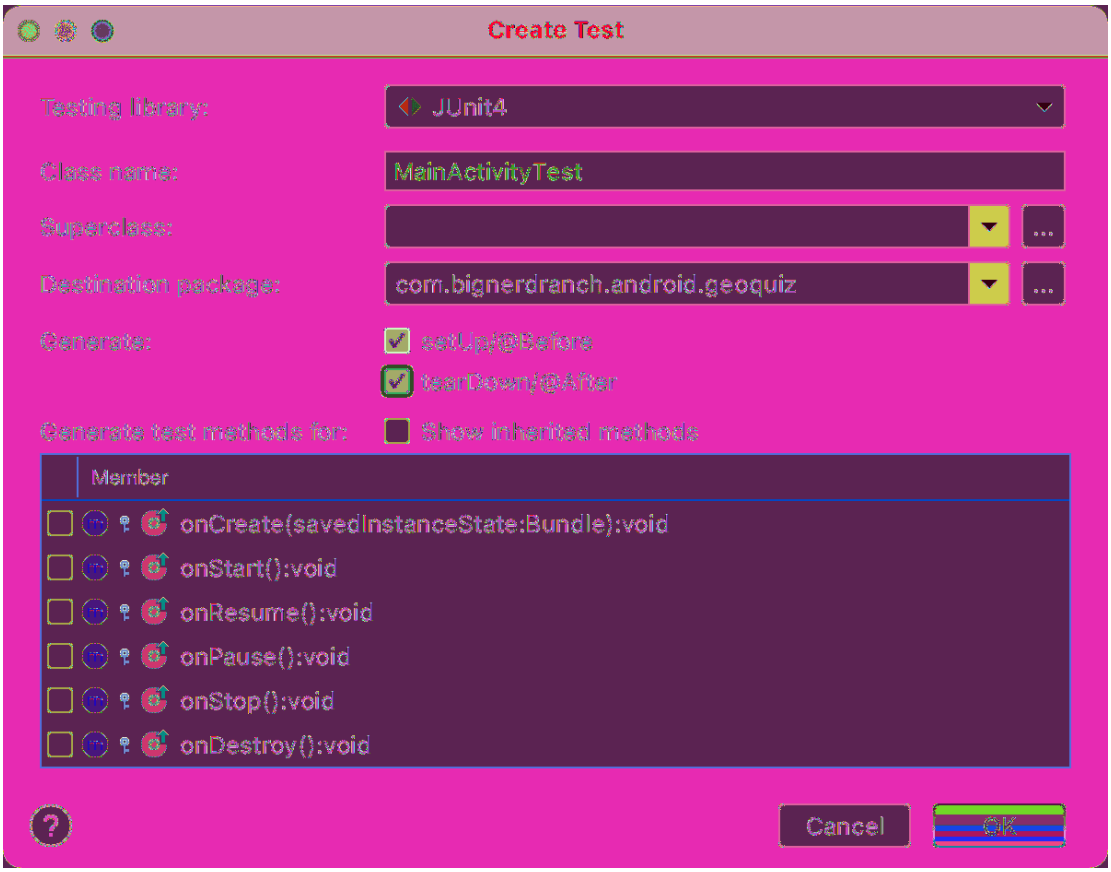
You are about to run **`ExampleInstrumentedTest`**, but you have some housekeeping to take care of first. Since instrumented tests run on an Android device, not your development machine, you need to either connect an Android device, as you did in Chapter 2, or run an emulator. Make sure the device dropdown at the top of the Android Studio window shows the device or emulator you want to use, then click the  icon in the gutter next to **`ExampleInstrumentedTest`** to run the test. Your test will execute, and the successful result should display in the lower-left corner of the Android Studio window.

Now that you have seen how instrumented tests work, you will write a few tests of your own to cover the functionality in **`MainActivity`**. You are going to use an API called **`ActivityScenario`** to set up your testing environment, and you will use the Espresso library to test and verify the behavior within **`MainActivity`**.

Create your test class file using the same Android Studio shortcut you used for **`QuizViewModelTest`**: In the project tool window, find and open `MainActivity.kt`. Place your cursor inside the class definition in the editor and press Command-Shift-T (Ctrl-Shift-T). Select Create New Test... in the pop-up.

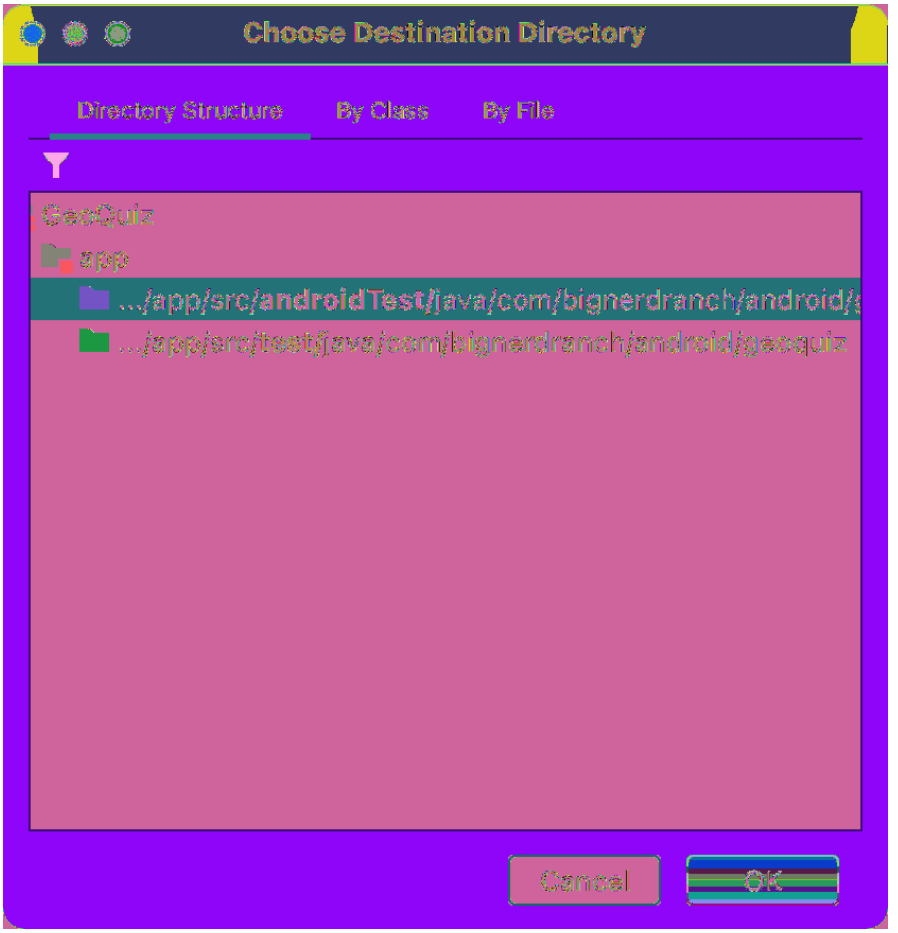
In the Create Test dialog, there are two fields you want to change this time. First, select JUnit4 from the Testing library dropdown. (As before, click the Fix button if you see JUnit4 library not found in the module.) Second, check both setUp/@Before and teardown/@After for the Generate option (Figure 6.7).

Figure 6.7 Creating your own instrumented test



Click OK. The final dialog asks you to choose the destination directory of the test. Since the code you are testing interacts directly with the Android SDK, you need to create an instrumented test. Select the folder containing `/androidTest/` in its path and click OK (Figure 6.8).

Figure 6.8 The androidTest directory selection



Android Studio will generate and open a fresh `MainActivityTest.kt`:

```
package com.bignerdranch.android.geoquiz

import org.junit.After
import org.junit.Before

class MainActivityTest {

    @Before
    fun setUp() {
    }

    @After
    fun tearDown() {
    }
}
```

Tests usually require a consistent environment to run in. The **`setUp()`** function, as the name suggests, allows you to set up that environment. The `@Before` annotation ensures that JUnit executes the **`setUp()`** function before every test. Similarly, the `@After` annotation ensures that **`tearDown()`**, where you can take care of any needed clean-up, executes after each test. (These annotations are also available for JVM tests.)

Since you cannot directly create an instance of **`MainActivity`**, you will use **`ActivityScenario`**. It will handle creating an instance for you and will provide an isolated environment in which you can test the instance. Set up the **`ActivityScenario`** as shown:

### Listing 6.3 Setting up your tests for **`MainActivity`** (`MainActivityTest.kt`)

```
package com.bignerdranch.android.geoquiz

import androidx.test.core.app.ActivityScenario
import androidx.test.core.app.ActivityScenario.launch
import androidx.test.ext.junit.runners.AndroidJUnit4
import org.junit.After
import org.junit.Before
import org.junit.runner.RunWith

@RunWith(AndroidJUnit4::class)
class MainActivityTest {

    private lateinit var scenario: ActivityScenario<MainActivity>

    @Before
    fun setUp() {
        scenario = launch(MainActivity::class.java)
    }

    @After
    fun tearDown() {
        scenario.close()
    }
}
```

(Do not forget the `@RunWith(AndroidJUnit4::class)` annotation.)

Now, **`setUp()`** will provide a fresh **`MainActivity`** before every test. At this point, you have your **`MainActivity`** in the resumed lifecycle state, meaning it is fully visible and capable of user interaction. This is the perfect environment to test its behavior.

The first behavior you are going to verify with a test is that when **`MainActivity`** is launched, the first quiz question should be displayed. Enter the code below; we will walk you through what is happening after you enter it.

### Listing 6.4 Writing your first **`MainActivity`** test (`MainActivityTest.kt`)

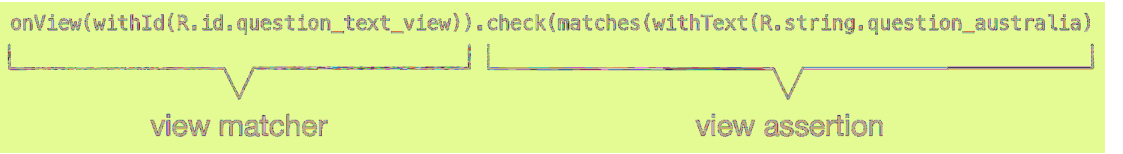
```
...
import androidx.test.core.app.ActivityScenario.launch
import androidx.test.espresso.Espresso.onView
import androidx.test.espresso.assertion.ViewAssertions.matches
import androidx.test.espresso.matcher.ViewMatchers.withId
import androidx.test.espresso.matcher.ViewMatchers.withText
import androidx.test.ext.junit.runners.AndroidJUnit4
...
@RunWith(AndroidJUnit4::class)
class MainActivityTest {
    ...
    @After
    fun tearDown() {
        scenario.close()
    }

    @Test
    fun showsFirstQuestionOnLaunch() {
        onView(withId(R.id.question_text_view))
            .check(matches(withText(R.string.question_australia)))
    }
}
```

The position of your new **`showsFirstQuestionOnLaunch()`** function in the file does not matter, but it is convention for the `@Before` and `@After` functions to come first. Note that you are using the same `@Test` annotation that you used in your JVM tests.

This test shows off your first Espresso test. Espresso's API is built as a fluent interface, meaning it relies heavily on chaining methods together to perform complicated actions. This expression can be broken into two pieces (Figure 6.9).

### Figure 6.9 Breaking down an Espresso assertion




The first half, called the *view matcher*, finds the particular view you are interested in. In this case, that is the **`TextView`** that displays the question text. The second half, called the *view assertion*, verifies the behavior you are interested in. Here, that is displaying the question about Australia.

Fluent interfaces are meant to be easily readable. A simple translation of your test into English would be: "On the view with the ID `R.id.question_text_view`, check that it matches the text from `R.string.question_australia`."

Use the  icon in the gutter next to **`MainActivityTest`** to run the test you just wrote. After compiling and eventually executing, you should see the test pass.

(If your test fails and gives you a cryptic error containing the message "lateinit property scenario has not been initialized," make sure you have disabled the "Don't keep activities" setting on your device that you enabled back in the section called Saving Data Across Process Death in Chapter 4.)

If you kept a close eye on your Android device while the test was running, you might have noticed something: There were brief flashes where your **`MainActivity`** was displayed on the device. **`ActivityScenario`** launched **`MainActivity`**, Espresso did its checks, and then your **`MainActivity`** was closed.

Espresso is not limited to only observing UI. It can also perform actions (such as clicking buttons or inputting text) on that UI. Write a second instrumented test to verify that when the user clicks the NEXT button, they see the second question in the quiz. In addition to checking that your view matches assertions, you will perform a click on your view:

### Listing 6.5 Writing your second **`MainActivity`** test (`MainActivityTest.kt`)

```
...
import androidx.test.espresso.Espresso.onView
import androidx.test.espresso.action.ViewActions.click
import androidx.test.espresso.assertion.ViewAssertions.matches
...
@RunWith(AndroidJUnit4::class)
class MainActivityTest {
    ...
    @Test
    fun showsFirstQuestionOnLaunch() {
        onView(withId(R.id.question_text_view))
            .check(matches(withText(R.string.question_australia)))
    }

    @Test
    fun showsSecondQuestionAfterNextPress() {
        onView(withId(R.id.next_button)).perform(click())
        onView(withId(R.id.question_text_view))
            .check(matches(withText(R.string.question_oceans)))
    }
}
```

Run the tests. You should see that they both pass.

The last test you are going to write will verify that you fixed the UI state loss on rotation bug introduced back in Chapter 3.

**`ActivityScenario`** is a container for your **`MainActivity`**, and it gives you many ways to poke and prod **`MainActivity`**. One of those tools is the ability to tear down and rebuild your **`Activity`** at will. The function you are about to use, **`recreate()`**, will produce the same situation as when you rotate the device. Thankfully, you have already fixed the state loss bug using **`SavedStateHandle`**. Your new test will verify that.

Listing 6.6 Checking that re-creation is handled (`MainActivityTest.kt`)

```
@RunWith(AndroidJUnit4::class)
class MainActivityTest {
    ...
    @Test
    fun showsSecondQuestionAfterNextPress() {
        onView(withId(R.id.next_button)).perform(click())
        onView(withId(R.id.question_text_view))
            .check(matches(withText(R.string.question_oceans)))
    }

    @Test
    fun handlesActivityRecreation() {
        onView(withId(R.id.next_button)).perform(click())
        scenario.recreate()
        onView(withId(R.id.question_text_view))
            .check(matches(withText(R.string.question_oceans)))
    }
}
```

Run all three tests and confirm that they all pass.

Instrumented tests for **`Activity`**s are important – and they are also more complicated to write and slower to run than JVM tests on **`ViewModel`**. Plus, **`ViewModel`** tests allow you to fairly easily control the input and output of a unit of code, which is the foundation for a well-written test. This difference in "testability" is one of the many reasons we recommend keeping business logic in **`ViewModel`**s rather than Android components like **`Activity`**.

You now have the fundamentals of testing on Android down. But there is still so much more to learn. There are entire books written on how to test software.

The best advice we can give here is to keep trying out new techniques during testing. Testing is a skill that must be honed over time. As you gain more experience writing Android code and writing tests to validate behavior, you will write better tests. And with better tests, you will release better apps.

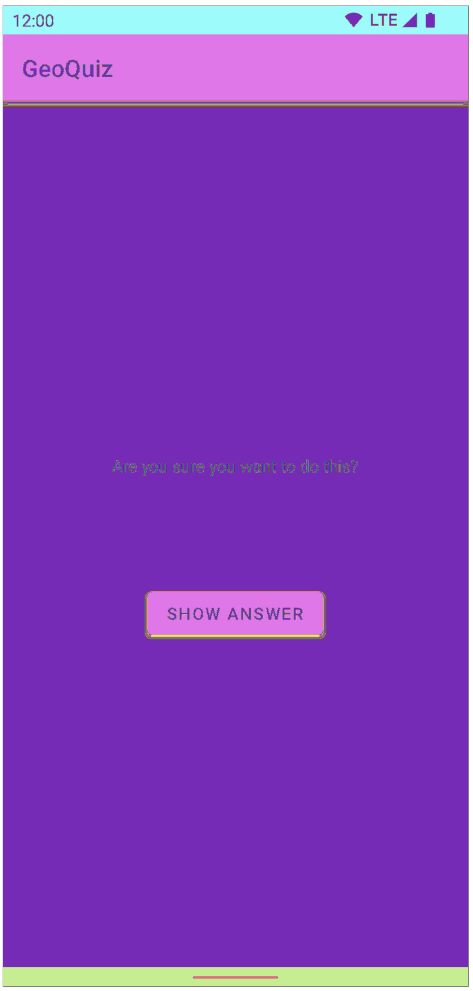
## Challenge: Asserting Yourself

In this chapter, you used **`assertEquals()`** to make assertions on your **`QuizViewModel`**. Checking for equality is not the only assertion you can make in JUnit tests. You can also check whether something is null (using **`assertNull()`**) or is true or false (using **`assertTrue()`** or **`assertFalse()`**, respectively), among others. Try using **`assertTrue()`** or **`assertFalse()`** to verify that `QuizViewModel.currentQuestionAnswer` behaves the way you expect it to.

# 7
# Your Second Activity

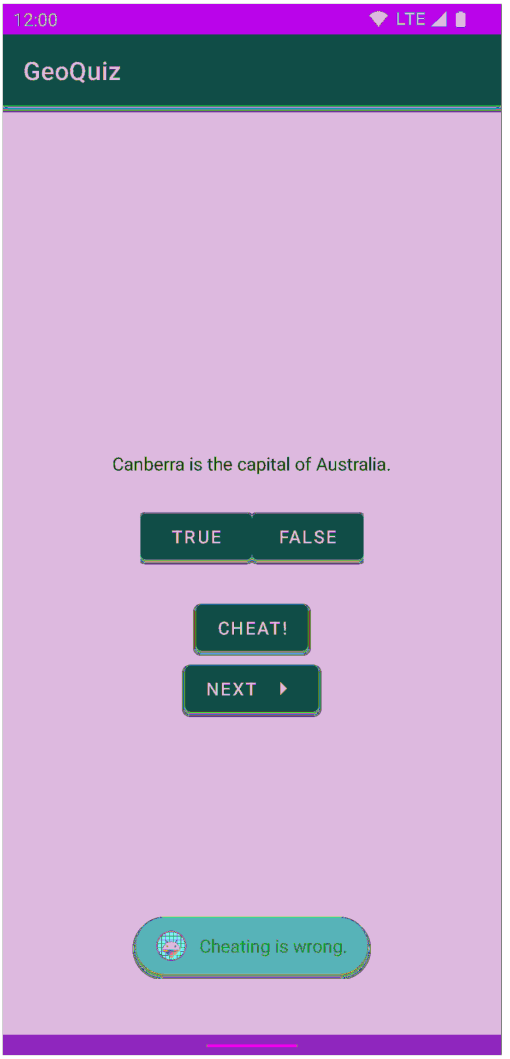
In this chapter, you will add a second activity to GeoQuiz. An activity controls a screen of information, and your new activity will add a second screen that offers users a chance to cheat on the current question by showing the answer. Figure 7.1 shows the new activity.

Figure 7.1 **`CheatActivity`** offers the chance to peek at the answer



If users choose to view the answer and then return to the **`MainActivity`** and answer the question, they will get a new message, shown in Figure 7.2.

Figure 7.2 **`MainActivity`** knows if you've been cheating




Why is this a good Android programming exercise? Because you will learn how to:

- Create a new activity and a new layout for it.
- Start an activity from another activity. *Starting* an activity means asking the OS to create an activity instance and call its **`onCreate(Bundle?)`** function.
- Pass data between the parent (starting) activity and the child (started) activity.

## Setting Up a Second Activity

There is a lot to do in this chapter. Fortunately, some of the grunt work can be done for you by Android Studio's New Android Activity wizard.

Before you invoke the magic, open `res/values/strings.xml` and add all the strings you will need for this chapter.

Listing 7.1 Adding strings (`res/values/strings.xml`)

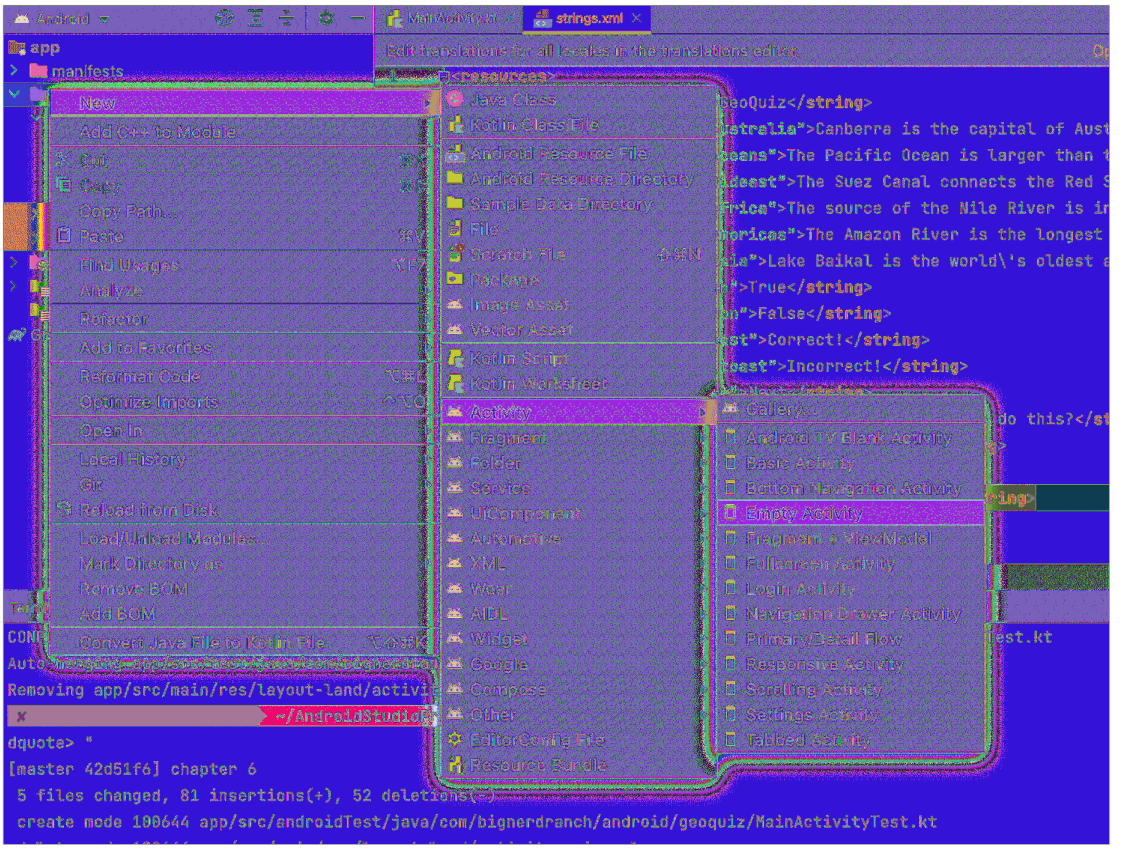
```
<resources>
    ...
    <string name="incorrect_toast">Incorrect!</string>
    <string name="warning_text">Are you sure you want to do this?</string>
    <string name="show_answer_button">Show Answer</string>
    <string name="cheat_button">Cheat!</string>
    <string name="judgment_toast">Cheating is wrong.</string>
</resources>
```

## Creating a new activity

Creating an activity typically involves touching at least three files: the Kotlin class file, an XML layout file, and the application manifest. If you touch those files in the wrong ways, Android can get mad. To ensure that you do it right, you can use Android Studio's New Android Activity wizard.

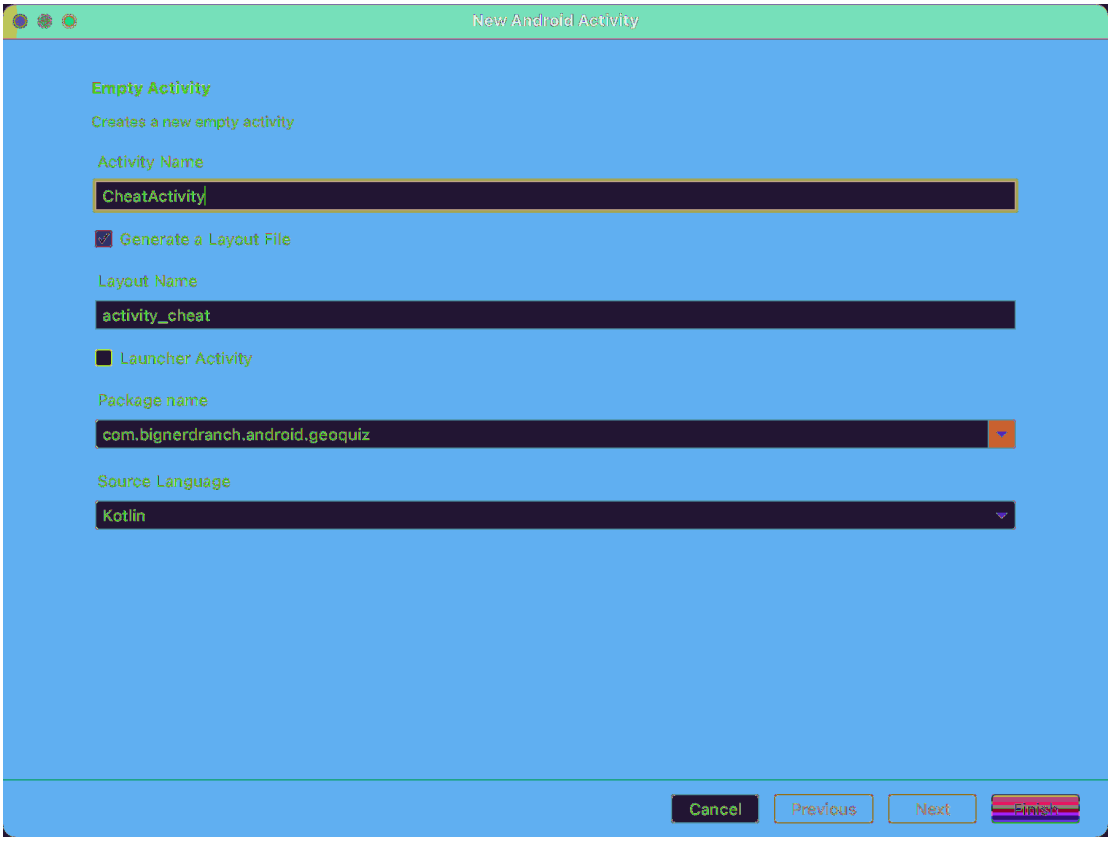
Launch the New Android Activity wizard by right-clicking the `app/java` folder in the project tool window. Choose New → Activity → Empty Activity, as shown in Figure 7.3.

Figure 7.3 The New Activity menu



You should see a dialog like Figure 7.4. Set Activity Name to CheatActivity. This is the name of your **`Activity`** subclass. Layout Name will be automatically set to activity_cheat. This is the base name of the layout file the wizard creates.

Figure 7.4 The New Empty Activity wizard




Check that the package name is `com.bignerdranch.android.geoquiz`. This determines where `CheatActivity.kt` will live on the filesystem. The defaults for the remaining fields are fine. Click the Finish button to make the magic happen.

Now it is time to make the UI look good. The screenshot at the beginning of the chapter shows you what **CheatActivity**'s view should look like. Figure 7.5 shows the view definitions.

Figure 7.5 Diagram of layout for **CheatActivity**




Open `res/layout/activity_cheat.xml` and switch to the Code view.

Try creating the XML for the layout using Figure 7.5 as a guide. Replace the sample layout with a new **LinearLayout** and so on down the tree. You can check your work against Listing 7.2.

Listing 7.2 Filling out the second activity's layout (`res/layout/activity_cheat.xml`)

```
<LinearLayout 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"
    android:gravity="center"
    android:orientation="vertical"
    tools:context="com.bignerdranch.android.geoquiz.CheatActivity">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:padding="24dp"
        android:text="@string/warning_text"/>

    <TextView
        android:id="@+id/answer_text_view"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:padding="24dp"
        tools:text="Answer"/>

    <Button
        android:id="@+id/show_answer_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/show_answer_button"/>

</LinearLayout>
```

Switch to the Design tab of the editor tool window to preview the layout. If it looks like Figure 7.5, you are ready to move on.

## A new activity subclass

`CheatActivity.kt` may have opened automatically in the editor tool window. If it did not, open it from the project tool window.

The **`CheatActivity`** class already includes a basic implementation of **`onCreate(Bundle?)`** that passes the resource ID of the layout defined in `activity_cheat.xml` to **`setContentView(…)`**. To match your first activity, update your new activity to use View Binding. Since your layout for **`CheatActivity`** is named `activity_cheat.xml`, View Binding will generate a class named **`ActivityCheatBinding`**.

Listing 7.3 Using View Binding in **`CheatActivity`** (`CheatActivity.kt`)

```
class CheatActivity : AppCompatActivity() {
    private lateinit var binding: ActivityCheatBinding

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_cheat)
        binding = ActivityCheatBinding.inflate(layoutInflater)
        setContentView(binding.root)
    }
}
```

**`CheatActivity`** will eventually do more in its **`onCreate(Bundle?)`** function. For now, let's take a look at another thing the New Android Activity wizard did for you: declaring **`CheatActivity`** in the application's manifest.

## Declaring activities in the manifest

The *manifest* is an XML file containing metadata that describes your application to the Android OS. The file is always named `AndroidManifest.xml`, and it lives in the `app/manifests` directory of your project.

In the project tool window, find and open `manifests/AndroidManifest.xml`. You can also use Android Studio's Find File dialog by pressing Command-Shift-O (Ctrl-Shift-N) and starting to type the filename. Once it has guessed the right file, press Return to open it.

Every activity in an application must be declared in the manifest so that the OS can access it.

When you used the New Project wizard to create **`MainActivity`**, the wizard declared the activity for you. Likewise, the New Android Activity wizard declared **`CheatActivity`** by adding the XML highlighted below:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.bignerdranch.android.geoquiz">

    <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/Theme.GeoQuiz">
        <activity
            android:name=".CheatActivity"
            android:exported="false" />
        <activity
            android:name=".MainActivity"
            android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

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

</manifest>
```

The `android:name` attribute is required, and the dot at the start of this attribute's value tells the OS that this activity's class is in the package specified in the `package` attribute in the `manifest` element at the top of the file.

You will sometimes see a fully qualified `android:name` attribute, like `android:name="com.bignerdranch.android.geoquiz.CheatActivity"`. The long-form notation is identical to the version above.

There are many interesting things in the manifest, but for now, let's stay focused on getting **`CheatActivity`** up and running. You will learn about the different parts of the manifest as you work through this book.

## Adding a cheat button to MainActivity

The plan is for the user to press a button in **MainActivity** to get an instance of **CheatActivity** onscreen. So you need a new button in `res/layout/activity_main.xml`.

You can see in Figure 7.2 that the new CHEAT! button is positioned above the NEXT button. In the layout, define the new button as a direct child of the root **LinearLayout**, right before the definition of the NEXT button.

### Listing 7.4 Adding a cheat button to the layout (`res/layout/activity_main.xml`)

```
        ...
    </LinearLayout>

    <Button
        android:id="@+id/cheat_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="24dp"
        android:text="@string/cheat_button" />

    <Button
        android:id="@+id/next_button"
        .../>

</LinearLayout>
```

Now, in `MainActivity.kt`, set a **View.OnClickListener** stub for the CHEAT! button.

### Listing 7.5 Wiring up the cheat button (`MainActivity.kt`)

```
class MainActivity : AppCompatActivity() {

    private lateinit var binding: ActivityMainBinding

    private val quizViewModel: QuizViewModel by viewModels()

    override fun onCreate(savedInstanceState: Bundle?) {
        ...
        binding.nextButton.setOnClickListener {
            quizViewModel.moveToNext()
            updateQuestion()
        }

        binding.cheatButton.setOnClickListener {
            // Start CheatActivity
        }

        updateQuestion()
    }
    ...
}
```

Now you can get to the business of starting **CheatActivity**.

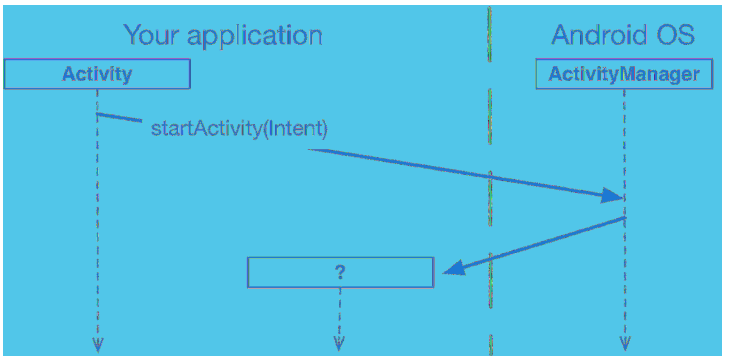
## Starting an Activity

The simplest way one activity can start another is with the **`startActivity(Intent)`** function.

If you are coming from other programming languages and platforms, your first instinct might be to call the constructor on the activity you want to start. Unfortunately, that will not work. Instead you have to call **`startActivity(Intent)`**, and the OS will manage creating your activity for you.

In particular, your call is sent to a part of the OS called the **`ActivityManager`**. The **`ActivityManager`** then creates the **`Activity`** instance and calls its **`onCreate(Bundle?)`** function, as shown in Figure 7.6.

Figure 7.6 Starting an activity




How does the **`ActivityManager`** know which **`Activity`** to start? That information is in the **`Intent`** parameter.

## Communicating with intents

An *intent* is an object that a *component* can use to communicate with the OS. The only components you have seen so far are activities, but there are also services, broadcast receivers, and content providers.

Intents are multipurpose communication tools, and the **`Intent`** class provides different constructors depending on what you are using the intent to do.

In this case, you are using an intent to tell the **`ActivityManager`** which activity to start, so you will use a constructor that allows you to pass in a **`Context`** and a reference to the **`CheatActivity`** class.

Within `cheatButton`'s listener, create an **`Intent`** that includes the **`CheatActivity`** class. Then pass the intent into **`startActivity(Intent)`**.

Listing 7.6 Starting **`CheatActivity`** (`MainActivity.kt`)

```
binding.cheatButton.setOnClickListener {
    // Start CheatActivity
    val intent = Intent(this, CheatActivity::class.java)
    startActivity(intent)
}
```

The **`Class`** argument you pass to the **`Intent`** constructor specifies the activity class that the **`ActivityManager`** should start. The **`Context`** argument tells the **`ActivityManager`** which application package the activity class can be found in.

Before starting the activity, the **ActivityManager** checks the package's manifest for a declaration with the same name as the specified **Class**. If it finds a declaration, it starts the activity, and all is well. If it does not, you get a nasty **ActivityNotFoundException**, which will crash your app. This is why all your activities must be declared in the manifest.

Run GeoQuiz. Press the CHEAT! button, and an instance of your new activity will appear onscreen. Now press the Back button. This will destroy the **CheatActivity** and return you to the **MainActivity**.

### Explicit and implicit intents

When you create an **Intent** with a **Context** and a **Class** object, you are creating an *explicit intent*. You use explicit intents to start specific activities, most often within your own application.

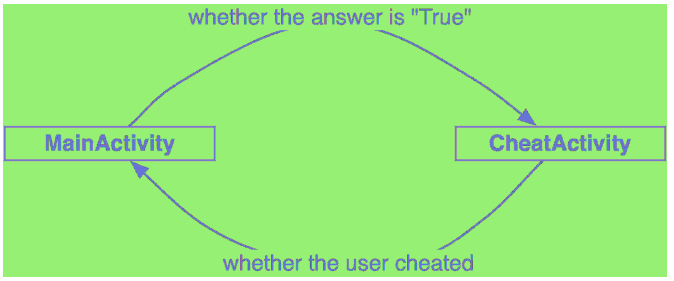
It may seem strange that two activities within your application must communicate via the **ActivityManager**, which is outside your application. However, this pattern makes it easy for an activity in one application to work with an activity in another application.

When an activity in your application wants to start an activity in another application, you create an *implicit intent*. You will use implicit intents in Chapter 16.

## Passing Data Between Activities

Now that you have a **MainActivity** and a **CheatActivity**, you can think about passing data between them. Figure 7.7 shows what data you will pass between the two activities.

Figure 7.7 The conversation between **MainActivity** and **CheatActivity**




The **MainActivity** will inform the **CheatActivity** of the answer to the current question when the **CheatActivity** is started.

When the user presses the Back button to return to the **MainActivity**, the **CheatActivity** will be destroyed. In its last gasp, it will send data to the **MainActivity** about whether the user cheated.

You will start with passing data from **MainActivity** to **CheatActivity**.

## Using intent extras

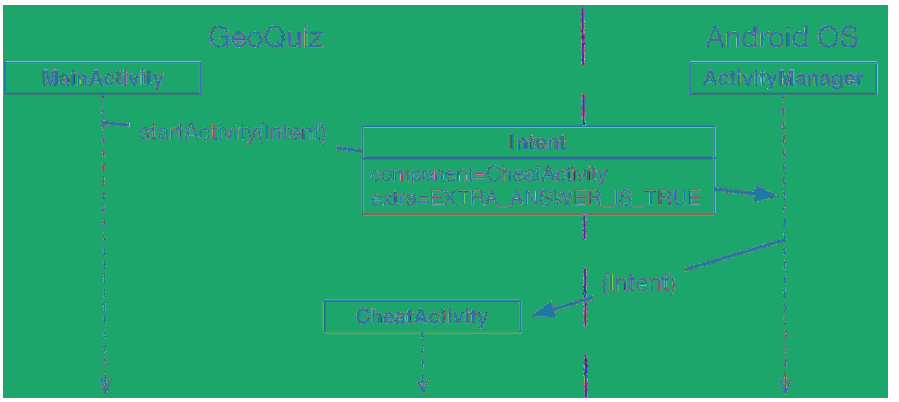
To inform the **CheatActivity** of the answer to the current question, you will pass it the value of:

```
quizViewModel.currentQuestionAnswer
```

You will send this value as an *extra* on the **Intent** that is passed into **startActivity(Intent)**.

Extras are arbitrary data that the calling activity can include with an intent. You can think of them like constructor arguments, even though you cannot use a custom constructor with an activity subclass. (Android creates activity instances and is responsible for their lifecycle.) The OS forwards the intent to the recipient activity, which can then access the extras and retrieve the data, as shown in Figure 7.8.

Figure 7.8 Intent extras: communicating with other activities




An extra is structured as a key-value pair, like the one you used to save out the value of `currentIndex` in **QuizViewModel**.

To add an extra to an intent, you use **Intent.putExtra(…)**. In particular, you will be calling **putExtra(name: String, value: Boolean)**.

**Intent.putExtra(…)** comes in many flavors, but it always has two arguments. The first argument is always a **String** key, and the second argument is the value, whose type will vary. It returns the **Intent** itself, so you can chain multiple calls if you need to.

In `CheatActivity.kt`, add a key for the extra. (We have broken the new line of code to fit on the printed page. You can enter it on one line.)

Listing 7.7 Adding an extra constant (`CheatActivity.kt`)

```
private const val EXTRA_ANSWER_IS_TRUE =
        "com.bignerdranch.android.geoquiz.answer_is_true"

class CheatActivity : AppCompatActivity() {
    ...
}
```

An activity may be started from several different places, so you should define keys for extras on the activities that retrieve and use them. Using your package name as a qualifier for your extra, as shown in Listing 7.7, prevents name collisions with extras from other apps.

Now you could return to **MainActivity** and put the extra on the intent, but there is a better approach. There is no reason for **MainActivity**, or any other code in your app, to know the implementation details of what **CheatActivity** expects as extras on its **Intent**. Instead, you can encapsulate that work into a **newIntent(…)** function.

Create this function in **CheatActivity** now. Place the function inside a companion object.

### Listing 7.8 A **newIntent(…)** function for **CheatActivity** (CheatActivity.kt)

```
class CheatActivity : AppCompatActivity() {

    private lateinit var binding: ActivityCheatBinding

    override fun onCreate(savedInstanceState: Bundle?) {
        ...
    }

    companion object {
        fun newIntent(packageContext: Context, answerIsTrue: Boolean): Intent {
            return Intent(packageContext, CheatActivity::class.java).apply {
                putExtra(EXTRA_ANSWER_IS_TRUE, answerIsTrue)
            }
        }
    }
}
```

This function allows you to create an **Intent** properly configured with the extras **CheatActivity** will need. The `answerIsTrue` argument, a `Boolean`, is put into the intent with a private name using the `EXTRA_ANSWER_IS_TRUE` constant. You will extract this value momentarily.

A companion object allows you to access functions without having an instance of a class, similar to static functions in Java. Using a **newIntent(…)** function inside a companion object like this for your activity subclasses will make it easy for other code to properly configure its launching intents.

Speaking of other code, use this new function in **MainActivity**'s cheat button listener now.

### Listing 7.9 Launching **CheatActivity** with an extra (MainActivity.kt)

```
binding.cheatButton.setOnClickListener {
    // Start CheatActivity
    val intent = Intent(this, CheatActivity::class.java)
    val answerIsTrue = quizViewModel.currentQuestionAnswer
    val intent = CheatActivity.newIntent(this@MainActivity, answerIsTrue)
    startActivity(intent)
}
```

You only need one extra in this case, but you can put multiple extras on an **Intent** if you need to. If you do, add more arguments to your **newIntent(…)** function to stay consistent with the pattern.

To retrieve the value from the extra, you will use **Intent.getBooleanExtra(String, Boolean)**.

The first argument of **getBooleanExtra(…)** is the name of the extra. The second argument is a default answer to use if the key is not found.

In **CheatActivity**, retrieve the value from the extra in **onCreate(Bundle?)** and store it in a member variable.

### Listing 7.10 Using an extra (CheatActivity.kt)

```
class CheatActivity : AppCompatActivity() {

    private lateinit var binding: ActivityCheatBinding

    private var answerIsTrue = false

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityCheatBinding.inflate(layoutInflater)
        setContentView(binding.root)

        answerIsTrue = intent.getBooleanExtra(EXTRA_ANSWER_IS_TRUE, false)
    }
    ...
}
```

Note that **Activity.getIntent()** always returns the **Intent** that started the activity. This is what you sent when calling **startActivity(Intent)**.

Finally, wire up the answer **TextView** and the SHOW ANSWER button to use the retrieved value.

### Listing 7.11 Enabling cheating (CheatActivity.kt)

```
class CheatActivity : AppCompatActivity() {

    private lateinit var binding: ActivityCheatBinding

    private var answerIsTrue = false

    override fun onCreate(savedInstanceState: Bundle?) {
        ...

        answerIsTrue = intent.getBooleanExtra(EXTRA_ANSWER_IS_TRUE, false)

        binding.showAnswerButton.setOnClickListener {
            val answerText = when {
                answerIsTrue -> R.string.true_button
                else -> R.string.false_button
            }
            binding.answerTextView.setText(answerText)
        }
    }
    ...
}
```

This code is pretty straightforward. You set the **TextView**'s text using **TextView.setText(Int)**. **TextView.setText(…)** has many variations, and here you use the one that accepts the resource ID of a string resource.

Run GeoQuiz. Press CHEAT! to get to **CheatActivity**. Then press SHOW ANSWER to reveal the answer to the current question.

## Getting a result back from a child activity

At this point, the user can cheat with impunity. Let's fix that by having the **`CheatActivity`** tell the **`MainActivity`** whether the user chose to view the answer.

When you want to hear back from the child activity, you register your **`MainActivity`** for an **`ActivityResult`** using the Activity Results API.

The Activity Results API is different from other APIs you have interacted with so far within the **`Activity`** class. Instead of overriding a lifecycle method, you will initialize a class property within your **`MainActivity`** using the **`registerForActivityResult()`** function. That function takes in two parameters: The first is a *contract* that defines the input and output of the **`Activity`** you are trying to start. And the second is a lambda in which you parse the output that is returned.

In **`MainActivity`**, initialize a property named `cheatLauncher` using **`registerForActivityResult()`**.

### Listing 7.12 Creating `cheatLauncher` (`MainActivity.kt`)

```
class MainActivity : AppCompatActivity() {

    private lateinit var binding: ActivityMainBinding

    private val quizViewModel: QuizViewModel by viewModels()

    private val cheatLauncher = registerForActivityResult(
        ActivityResultContracts.StartActivityForResult()
    ) { result ->
        // Handle the result
    }

    override fun onCreate(savedInstanceState: Bundle?) {
        ...
    }
    ...
}
```

The contract you are using is **`ActivityResultContracts.StartActivityForResult`**. It is a basic contract that takes in an **`Intent`** as input and provides an **`ActivityResult`** as output. There are many other contracts you can use to accomplish other tasks (such as capturing video or requesting permissions). You can even define your own custom contract. In Chapter 16, you will use a different contract to allow the user to select a contact from their contacts list.

For now, you will do nothing with the result, but you will get back to this in a bit.

To use your `cheatLauncher`, call the **`launch(Intent)`** function, which takes in the **`Intent`** you already created.

Listing 7.13 Launching `cheatLauncher` (`MainActivity.kt`)

```
class MainActivity : AppCompatActivity() {
    ...
    override fun onCreate(savedInstanceState: Bundle?) {
        ...
        binding.cheatButton.setOnClickListener {
            // Start CheatActivity
            val answerIsTrue = quizViewModel.currentQuestionAnswer
            val intent = CheatActivity.newIntent(this@MainActivity, answerIsTrue)
            startActivity(intent)
            cheatLauncher.launch(intent)
        }

        updateQuestion()
    }
    ...
}
```

## Setting a result

There are two functions you can call in the child activity to send data back to the parent:

```
    setResult(resultCode: Int)
    setResult(resultCode: Int, data: Intent)
```

Typically, the *result code* is one of two predefined constants: `Activity.RESULT_OK` or `Activity.RESULT_CANCELED`. (You can use another constant, `RESULT_FIRST_USER`, as an offset when defining your own result codes.)

Setting result codes is useful when the parent needs to take different action depending on how the child activity finished.

For example, if a child activity had an OK button and a Cancel button, the child activity would set a different result code depending on which button was pressed. Then the parent activity would take a different action depending on the result code.

Calling **`setResult(…)`** is not required of the child activity. If you do not need to distinguish between results or receive arbitrary data on an intent, then you can let the OS send a default result code. A result code is always returned to the parent if the child activity was started with **`startActivityForResult(…)`**. If **`setResult(…)`** is not called, then when the user presses the Back button, the parent will receive `Activity.RESULT_CANCELED`.

### Sending back an intent

In this implementation, you are interested in passing some specific data back to **`MainActivity`**. So you are going to create an **`Intent`**, put an extra on it, and then call **`Activity.setResult(Int, Intent)`** to get that data into **`MainActivity`**'s hands.

In **`CheatActivity`**, add a constant for the extra's key and a private function that does this work. Then call this function in the SHOW ANSWER button's listener.

Listing 7.14 Setting a result (`CheatActivity.kt`)

```
const val EXTRA_ANSWER_SHOWN = "com.bignerdranch.android.geoquiz.answer_shown"
private const val EXTRA_ANSWER_IS_TRUE =
        "com.bignerdranch.android.geoquiz.answer_is_true"

class CheatActivity : AppCompatActivity() {
    ...
    override fun onCreate(savedInstanceState: Bundle?) {
        ...
        binding.showAnswerButton.setOnClickListener {
            ...
            binding.answerTextView.setText(answerText)
            setAnswerShownResult(true)
        }
    }

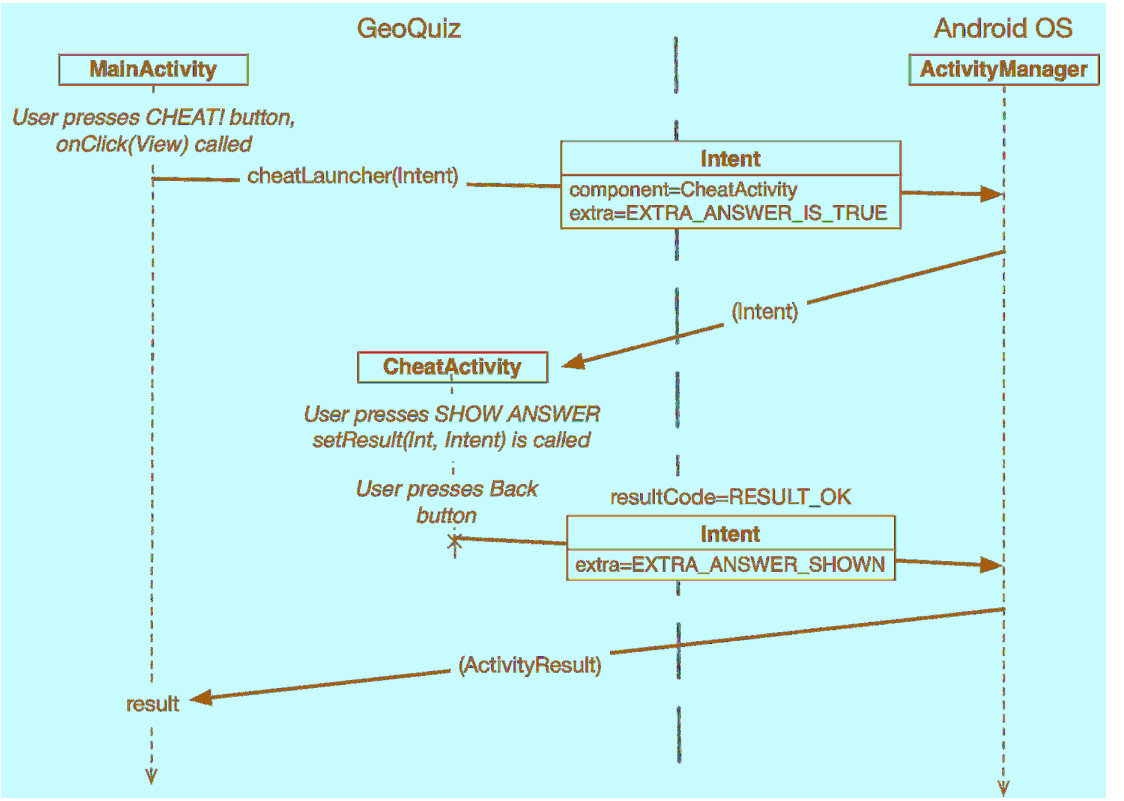
    private fun setAnswerShownResult(isAnswerShown: Boolean) {
        val data = Intent().apply {
            putExtra(EXTRA_ANSWER_SHOWN, isAnswerShown)
        }
        setResult(Activity.RESULT_OK, data)
    }
    ...
}
```

When the user presses the SHOW ANSWER button, the **`CheatActivity`** packages up the result code and the intent in the call to **`setResult(Int, Intent)`**.

Then, when the user presses the Back button to return to the **`MainActivity`**, the **`ActivityManager`** invokes the lambda defined within `cheatLauncher` on the parent activity. The parameters are the original request code from **`MainActivity`** and the result code and intent passed into **`setResult(Int, Intent)`**.

Figure 7.9 shows this sequence of interactions.

Figure 7.9 Sequence diagram for GeoQuiz



The final step is to extract the data returned in the lambda of `cheatLauncher` in **`MainActivity`**.

### Handling a result

In `QuizViewModel.kt`, add a new property to hold the value that **`CheatActivity`** is passing back. The user's cheat status is part of the UI state. Stashing the value in **`QuizViewModel`** and using **`SavedStateHandle`** means the value will persist across configuration changes and process death rather than being destroyed with the activity, as discussed in Chapter 4.

Listing 7.15 Tracking cheating in **`QuizViewModel`** (`QuizViewModel.kt`)

```
...
const val IS_CHEATER_KEY = "IS_CHEATER_KEY"

class QuizViewModel(private val savedStateHandle: SavedStateHandle) : ViewModel() {
    ...
    private val questionBank = listOf(
        ...
    )

    var isCheater: Boolean
        get() = savedStateHandle.get(IS_CHEATER_KEY) ?: false
        set(value) = savedStateHandle.set(IS_CHEATER_KEY, value)
    ...
}
```

Next, in `MainActivity.kt`, add the following lines in the lambda of `cheatLauncher` to pull the value out of the result sent back from **`CheatActivity`**. You do not want to accidentally mark the user as a cheater, so check whether the result code is `Activity.RESULT_OK` first.

Listing 7.16 Pulling out the data in `cheatLauncher` (`MainActivity.kt`)

```
class MainActivity : AppCompatActivity() {

    private lateinit var binding: ActivityMainBinding

    private val quizViewModel: QuizViewModel by viewModels()

    private val cheatLauncher = registerForActivityResult(
        ActivityResultContracts.StartActivityForResult()
    ) { result ->
        // Handle the result
        if (result.resultCode == Activity.RESULT_OK) {
            quizViewModel.isCheater =
                result.data?.getBooleanExtra(EXTRA_ANSWER_SHOWN, false) ?: false
        }
    }

    override fun onCreate(savedInstanceState: Bundle?) {
        ...
    }
    ...
}
```

Finally, modify the **`checkAnswer(Boolean)`** function in **`MainActivity`** to check whether the user cheated and respond appropriately.

### Listing 7.17 Changing toast message based on value of `isCheater` (`MainActivity.kt`)

```
class MainActivity : AppCompatActivity() {
    ...
    private fun checkAnswer(userAnswer: Boolean) {
        val correctAnswer: Boolean = quizViewModel.currentQuestionAnswer

        val messageResId = if (userAnswer == correctAnswer) {
            R.string.correct_toast
        } else {
            R.string.incorrect_toast
        }
        val messageResId = when {
            quizViewModel.isCheater -> R.string.judgment_toast
            userAnswer == correctAnswer -> R.string.correct_toast
            else -> R.string.incorrect_toast
        }
        Toast.makeText(this, messageResId, Toast.LENGTH_SHORT)
                .show()
    }
}
```

Run GeoQuiz. Press CHEAT!, then press SHOW ANSWER on the cheat screen. Once you cheat, press the Back button. Try answering the current question. You should see the judgment toast appear.

What happens if you go to the next question? Still a cheater. If you wish to relax your rules around cheating, try your hand at the challenge outlined in the section called Challenge: Tracking Cheat Status by Question.

At this point, GeoQuiz is feature complete. In the next chapter, you will learn how to include the newest Android features available while still supporting older versions of Android in the same application.

## How Android Sees Your Activities

Let's look at what is going on OS-wise as you move between activities. First, when you click the GeoQuiz app in the launcher, the OS does not start the application; it starts an activity in the application. More specifically, it starts the application's *launcher activity*. For GeoQuiz, **`MainActivity`** is the launcher activity.

When the New Project wizard created the GeoQuiz application and **`MainActivity`**, it made **`MainActivity`** the launcher activity by default. Launcher activity status is specified in the manifest by the `intent-filter` element in **`MainActivity`**'s declaration:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    ... >

    <application
        ... >
        <activity
            android:name=".CheatActivity"
            android:exported="true" />
        <activity
            android:name=".MainActivity"
            android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>

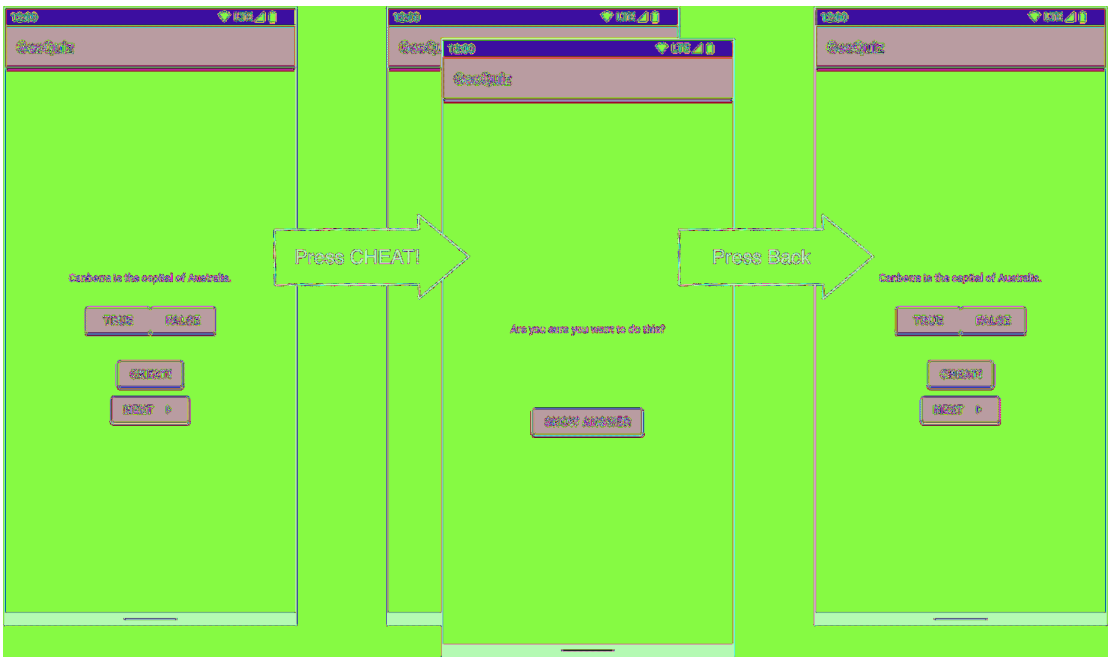
                <category android:name="android.intent.category.LAUNCHER"/>
            </intent-filter>
        </activity>
    </application>

</manifest>
```

After the instance of **`MainActivity`** is onscreen, the user can press the CHEAT! button. When this happens, an instance of **`CheatActivity`** is started – on top of the **`MainActivity`**. These activities exist in a stack (Figure 7.10).

Pressing the Back button in **`CheatActivity`** pops this instance off the stack, and the **`MainActivity`** resumes its position at the top, as shown in Figure 7.10.

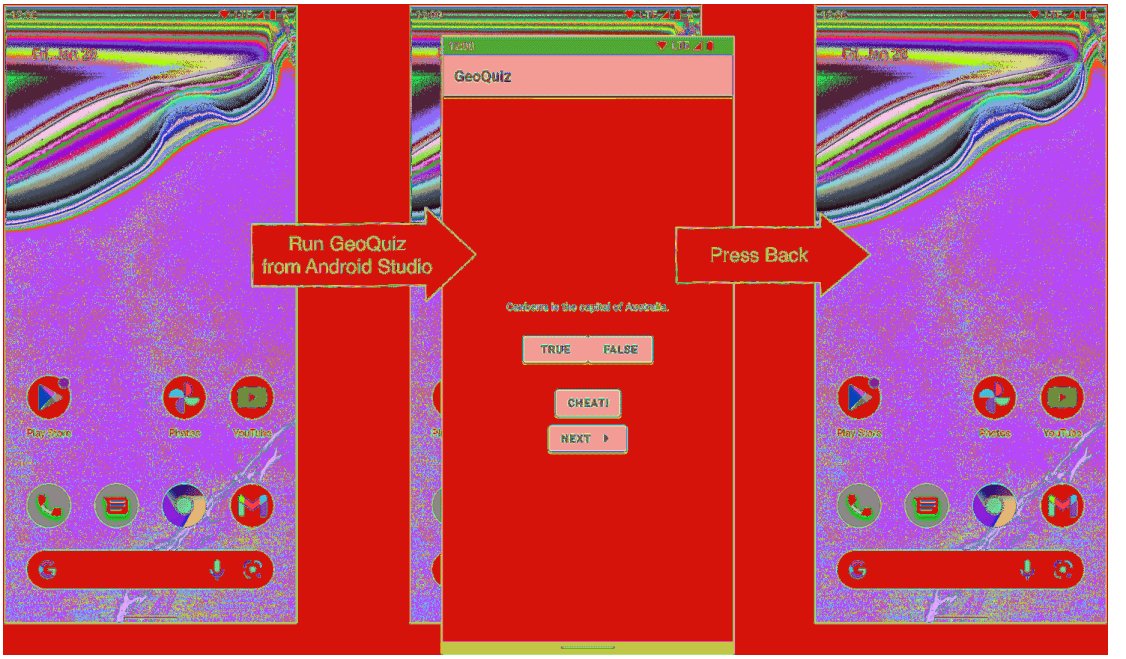
Figure 7.10 GeoQuiz's back stack




A call to **`Activity.finish()`** in **`CheatActivity`** would also pop the **`CheatActivity`** off the stack.

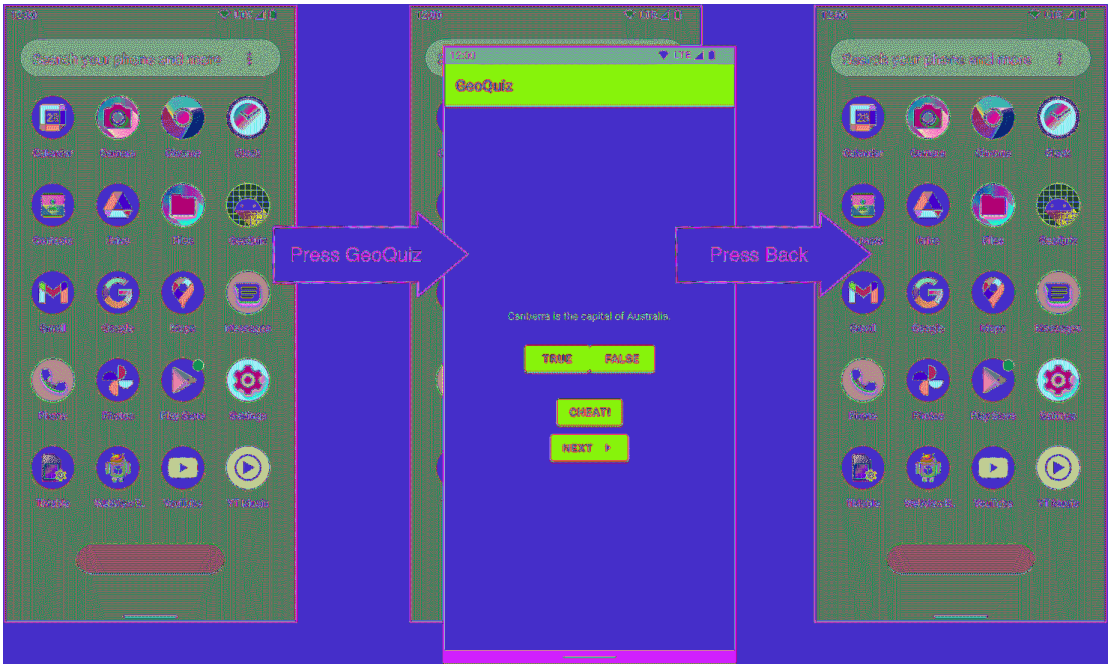
If you run GeoQuiz and press the Back button from the **`MainActivity`**, the **`MainActivity`** will be moved to the background in its created state and you will return to the last screen you were viewing before running GeoQuiz (Figure 7.11). For a little more detail on the difference in behavior when navigating Back from **`MainActivity`** versus what happens in **`CheatActivity`**, check out the section called For the More Curious: The Back Button and the Activity Lifecycle near the end of this chapter.

Figure 7.11 Looking at the Home screen



If you start GeoQuiz from the launcher application, pressing the Back button from **`MainActivity`** will return you to the launcher (Figure 7.12).

Figure 7.12 Running GeoQuiz from the launcher




Pressing the Back button from the launcher will return you to the screen you were looking at before you opened the launcher.

What you are seeing here is that the **`ActivityManager`** maintains a *back stack* and that this back stack is not just for your application's activities. Activities for all applications share the back stack, which is one reason the **`ActivityManager`** is involved in starting your activities and lives with the OS and not your application. The stack represents the use of the OS and device as a whole rather than the use of a single application.

## For the More Curious: startActivityForResult

This chapter uses the Activity Result APIs to start and pass information back from **`CheatActivity`**. These APIs are a relatively recent addition to Android. They were actually built on top of existing APIs: the **`startActivityForResult()`** function and **`onActivityResult()`** callback. We recommend using the Activity Result APIs, as they enable you to consume type-safe results and they encourage developers to write more modular code – but you might see applications that call the older, lower-level APIs directly.

There are many parallels between these two approaches.

The **`startActivityForResult()`** function is analogous to the **`launch(Intent)`** function you used on `cheatLauncher`. The one additional parameter necessary for **`startActivityForResult()`** is a `requestCode` that uniquely identifies your request for a result.

On the result side, the **`onActivityResult()`** callback maps closely to the lambda invoked on `cheatLauncher`. You have access to the data and the result code and also the `requestCode` you passed into **`startActivityForResult()`**. Since you can start many different activities for results, using the legacy APIs means many requests could invoke the **`onActivityResult()`** callback. This is where your `requestCode` comes in handy, as you can choose to perform certain actions only for certain requests.

If you see an application using the legacy **`startActivityForResult()`** and **`onActivityResult()`** methods, consider migrating to the Activity Result APIs. These new APIs are easier to use and make it easier to see how data is moving between activities in your application.

## For the More Curious: The Back Button and the Activity Lifecycle

In this chapter, you added **`CheatActivity`** and learned how the Android OS manages your activities through **`ActivityManager`**. You also saw what happens to each of your activities when you press the Back button while they are in the foreground: When you press Back from **`CheatActivity`**, it is popped off the back stack and removed from system memory. But when you press Back from **`MainActivity`**, it is only moved to the background – it still exists in memory, though it is not visible in its created state. The same happens with **`MainActivity`** when you press the Home button to temporarily leave the activity, as you did in the section called Temporarily leaving an activity in Chapter 3.

You may be wondering why Android treats your two activities differently with the same interaction. The answer is because **`MainActivity`** is declared as the launcher activity in `AndroidManifest.xml`. This declaration not only lets Android know this is the activity to start when users first launch the app but also tells it to treat this entry point specially when popping it off the back stack. Keeping the launcher activity in memory allows users to quickly resume using the app in a "warm state" after navigating back instead of having to completely restart the application.

This is a fairly new behavior on Android, introduced with Android 12 (API 31). In previous Android versions, your launcher activity would mirror the behavior of **`CheatActivity`** and be popped off the back stack and removed from memory when the user navigated back. You will learn more about Android versions and how they introduce new behaviors for the system and applications in Chapter 8. For now, be aware that users will see subtly different behaviors when they press the Back button on your launcher activity depending on which version of Android they are using.

If you would like to see this difference in action, we encourage you play around with different versions of Android (either on an emulator or your physical device) and observe the Logcat statements you added in **`MainActivity`**. Pay attention to when **`onDestroy()`** is called and when it is not.

## Challenge: Closing Loopholes for Cheaters

Cheaters never win. Unless, of course, they persistently circumvent your anticheating measures. Which they probably will. Because they are cheaters.

GeoQuiz has a major loophole: Users can rotate **`CheatActivity`** after they cheat to clear out the cheating result. When they go back to **`MainActivity`**, it is as if they never cheated at all.

Fix this bug by persisting **`CheatActivity`**'s UI state across rotation and process death using the techniques you learned in Chapter 4.

## Challenge: Tracking Cheat Status by Question

Currently, when the user cheats on a single question, they are considered a cheater on all the questions. Update GeoQuiz to track whether the user cheated on a question-by-question basis. When the user cheats on a given question, present them with the judgment toast any time they attempt to answer that question. When a user answers a question they have not cheated on yet, show the correct or incorrect toast accordingly.

# 8

# Android SDK Versions and Compatibility

Now that you have gotten your feet wet with GeoQuiz, let's review some background information about the different versions of Android. The information in this chapter is important to have under your belt as you continue with the book and develop more complex and realistic apps.

## Android SDK Versions

The Android operating system has been around for many years, starting with its first public release in fall 2008. From a developer's perspective, there have been 32 releases of Android – and counting.

Each update is referred to by a variety of names. The most familiar to users is the marketing name. For years, Google named all releases for tasty treats (such as Donut, Jelly Bean, and Pie – the last one to get that kind of name) in alphabetical order. Beginning with Android 10, released in 2019, the marketing names for major releases use an incrementing number.

For each release, there are often additional names, such as a version number or a version code – but the "name" most useful to developers is the API level. The first update was API level 1, and that number has incremented by 1 for each subsequent update. The 32nd and latest release has the marketing name Android 12L, version number 12, version code Sv2, and API level 32. Table 8.1 shows the information for several recent releases.

Table 8.1 Recent Android versions

| Marketing name | Version number | Version code | API level |
|---|---|---|---|
| Android Nougat | 7.0 | N | 24 |
| Android Nougat | 7.1 – 7.1.2 | N_MR1 | 25 |
| Android Oreo | 8.0 | O | 26 |
| Android Oreo | 8.1.0 | O_MR1 | 27 |
| Android Pie | 9 | P | 28 |
| Android 10 | 10 | Q | 29 |
| Android 11 | 11 | R | 30 |
| Android 12 | 12 | S | 31 |
| Android 12L | 12 | Sv2 | 32 |

Because there have been so many updates to Android over the years, it is very common for a device to be running an older version of Android. The percentage of devices using each version changes continuously, but one constant is that only around 5% of users adopt the newest version of Android right away. Google has worked hard to improve the adoption rates (through efforts like Project Mainline and Project Treble), but there are still many devices on older versions.

Why do so many devices still run older versions of Android? This is mostly due to heavy competition among Android device manufacturers and US carriers. Carriers want features and phones that no other network has. Device manufacturers feel this pressure, too – all their phones are based on the same OS, but they want to stand out from the competition. The combination of pressures from the market and the carriers means that there is a bewildering array of devices with proprietary, one-off modifications of Android.

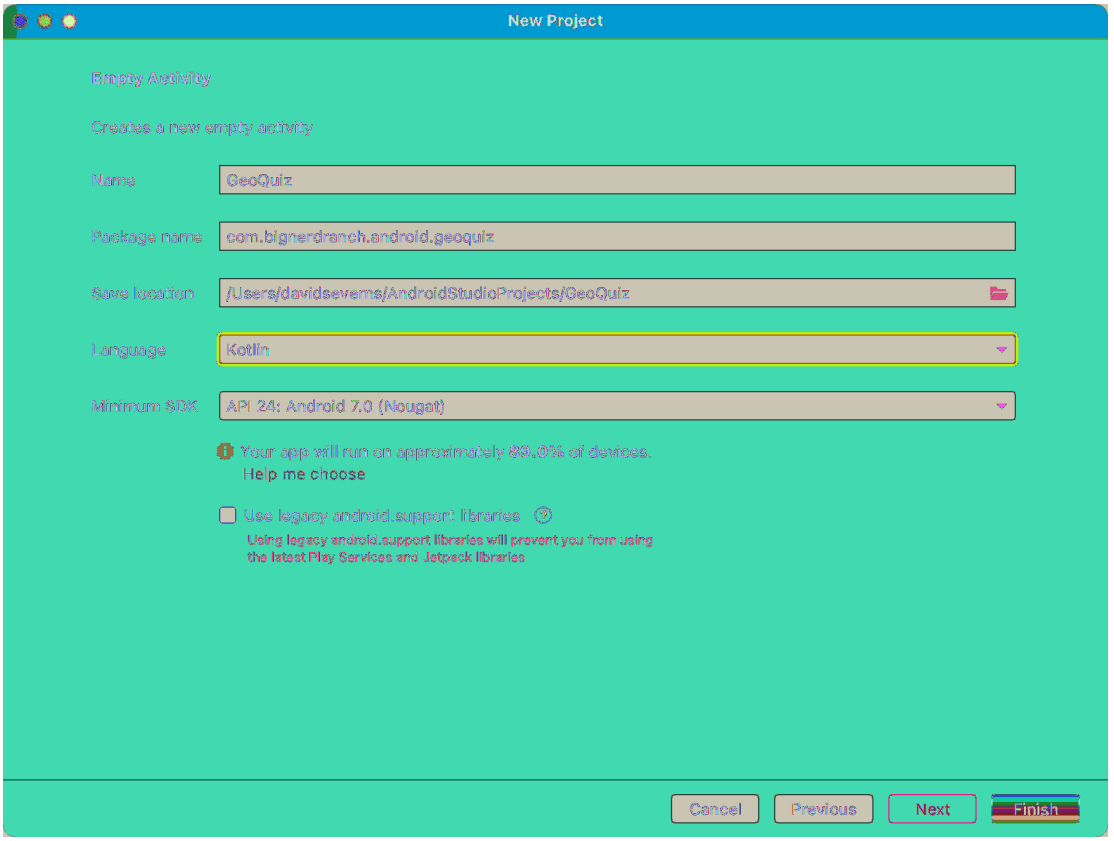
A device with a proprietary version of Android is not able to run a new version of Android released by Google. Instead, it must wait for a compatible proprietary upgrade. That upgrade might not be available until months after Google releases its version, if it is ever available. Manufacturers often choose to spend resources on newer devices rather than keeping older ones up to date.

## A sane minimum

The oldest version of Android that the exercises in this book support is API level 24. There are references to legacy versions of Android, but the focus is on what we consider to be modern versions (API level 24 and up). With the distribution of older releases dropping month by month, the amount of work required to support those older versions eclipses the value supporting them can provide.

When you created the GeoQuiz project, you set a minimum SDK version within the New Project wizard (Figure 8.1). (Note that Android uses the terms “SDK version” and “API level” interchangeably.)

Figure 8.1 Remember me?




In addition to the minimum supported version, you can also set the *target* version and the *compile* version. Let's take a look at the default choices and how to change them.

All these values are set in your build environment, so open the `build.gradle` file labeled Module (GeoQuiz.app). Notice the values for `compileSdk`, `minSdk`, and `targetSdk`:

```
...
compileSdk 32

defaultConfig {
    applicationId "com.bignerdranch.android.geoquiz"
    minSdk 24
    targetSdk 32
    ...
}
...
```

## Minimum SDK version

The `minSdk` value is a hard floor below which the OS should refuse to install the app.

By setting this version to API level 24, you give Android permission to install GeoQuiz on devices running API level 24 or higher. Android will refuse to install GeoQuiz on a device running anything lower than API level 24.

## Target SDK version

The `targetSdk` value tells Android which API level your app is *designed* to run on. Most often this will be the latest Android release.

When would you lower the target SDK? New SDK releases can change how your app appears on a device or even how the OS behaves behind the scenes. If you have already developed an app, you should confirm that it works and looks as expected on new releases. Check the documentation at `developer.android.com/reference/android/os/Build.VERSION_CODES.html` and `developer.android.com/about/versions` to see where problems might arise.

If your app will have issues with a new release of Android, you can modify your app to work with the new behavior and update the target SDK – or you can leave the codebase and target SDK as they were. Not increasing the target SDK means that your app will continue running with the appearance and behavior of the targeted version on which it worked well. This provides compatibility with newer versions of Android, as changes in subsequent releases are ignored until the `targetSdk` is increased.

However, you cannot ignore new Android releases forever by keeping your target SDK low. Google has restrictions on how low an app's target SDK can be and still ship on the Google Play Store. As of this writing, any new apps or app updates must have a target SDK of at least API level 30 – or they will be rejected by the Play Store. This ensures that users can benefit from the performance and security improvements in recent versions of Android. These version requirements will increase over time, as new versions of Android are released, so make sure you keep an eye on the documentation to know when you need to update your target version.

## Compile SDK version

The last SDK setting is the `compileSdk`. While the minimum and target SDK versions are placed in the `AndroidManifest.xml` when you build your app, to advertise those values to the OS, the compile SDK version is private information between you and the compiler.

Android's features are exposed through the classes and functions in the SDK. The compile SDK version specifies which version to use when building your code. When Android Studio is looking to find the classes and functions you refer to in your imports, the compile SDK version determines which SDK version it checks against.

The best choice for a compile SDK version is the latest API level available. Later versions of the compile SDK provide bug fixes, more compilation checks, and new APIs that you can use. There are instances where incrementing the compile SDK version will cause build issues, but they are extremely rare and can often be worked around. Unlike your target SDK version, changing your compile SDK version will not change any runtime behavior of your app.

You can modify the minimum SDK version, target SDK version, and compile SDK version in your `app/build.gradle` file. Remember that you must sync your project with the Gradle changes before they will be reflected.

# Compatibility and Android Programming

The delay in upgrades combined with regular new releases makes compatibility an important issue in Android programming. To reach a broad market, Android developers must create apps that perform well on devices running the most current version of Android plus previous versions – as well as on different device form factors.

## Jetpack libraries

In Chapter 4, you learned about the Jetpack libraries and AndroidX. In addition to offering new features (like **`ViewModel`**), the Jetpack libraries offer backward compatibility for new features on older devices and provide (or attempt to provide) consistent behavior across Android versions. Some libraries, such as AppCompat (which you will learn about in Chapter 11), ensure that your app has a consistent look and feel across all modern versions of Android. Other libraries, such as WorkManager (which you will use in Chapter 22), provide a consistent environment to perform essential tasks within your app.

Many of the AndroidX libraries in Jetpack are modifications of previous support libraries. You should strive to use these libraries any time you can. This makes your life as a developer easier, because you no longer have to worry about different results on different API versions. Your users also benefit, because they will have the same experience no matter what version their device is running.

Unfortunately, the Jetpack libraries are not a compatibility cure-all, because not all the features you will want to use are available in them. The Android team does a good job of adding new APIs to the Jetpack libraries as time goes on, but you will still find cases where a certain API is unavailable. In this case, you will need to use explicit version checks until a Jetpack version of the feature is added.

## Safely adding code from later APIs

The difference between GeoQuiz's minimum SDK version and compile SDK version leaves you with a compatibility gap to manage. For example, what happens if you call code from an API later than the minimum of API level 24? When your app is installed and run on a device running API level 24, it will crash.

This used to be a testing nightmare. However, thanks to improvements in Android Lint, potential problems caused by calling newer code on older devices can be caught at compile time. If you use code from a higher version than your minimum SDK, Android Lint will report build errors.

Right now, all of GeoQuiz's simple code was introduced in API level 24 or earlier. Let's add some code from after API level 24 and see what happens.

Open `MainActivity.kt`. At the bottom of the class, add a function that will blur the cheat button when called.

### Listing 8.1 Blurring the cheat button (`MainActivity.kt`)

```
class MainActivity : AppCompatActivity() {
    ...
    private fun checkAnswer(userAnswer: Boolean) {
        ...
    }

    private fun blurCheatButton() {
        val effect = RenderEffect.createBlurEffect(
            10.0f,
            10.0f,
            Shader.TileMode.CLAMP
        )
        binding.cheatButton.setRenderEffect(effect)
    }
}
```

Notice that a Lint error appears on the lines where you call **`RenderEffect.createBlurEffect(…)`** and **`View.setRenderEffect(…)`**, in the form of a red squiggly under the function name and, when you click on the function, a red light bulb icon. These functions were added to the Android SDK in API level 31, so this code would crash on a device running API level 30 or lower.

Because your compile SDK version is API level 32, the compiler has no problem with this code. Android Lint, on the other hand, knows about your minimum SDK version, so it complains.

The error message reads something like Call requires API level 31 (Current min is 24). You can still run the code with this warning (try it and see), but Lint knows it is not safe.

How do you get rid of this error? One option is to raise the minimum SDK version to 31. However, that means your app can only be run on a select few devices. Plus, raising the minimum SDK version is not really dealing with this compatibility problem as much as ducking it.

The way to appease Android Lint is to add an annotation declaring that the code you just wrote can only run on devices running API level 31.

### Listing 8.2 Making Android Lint happy (`MainActivity.kt`)

```
class MainActivity : AppCompatActivity() {
    ...
    private fun checkAnswer(userAnswer: Boolean) {
        ...
    }

    @RequiresApi(Build.VERSION_CODES.S)
    private fun blurCheatButton() {
        val effect = RenderEffect.createBlurEffect(
            10.0f,
            10.0f,
            Shader.TileMode.CLAMP
        )
        binding.cheatButton.setRenderEffect(effect)
    }
}
```

Now, invoke **`blurCheatButton()`** in **`onCreate(Bundle?)`**:

### Listing 8.3 Blurring the cheat button (`MainActivity.kt`)

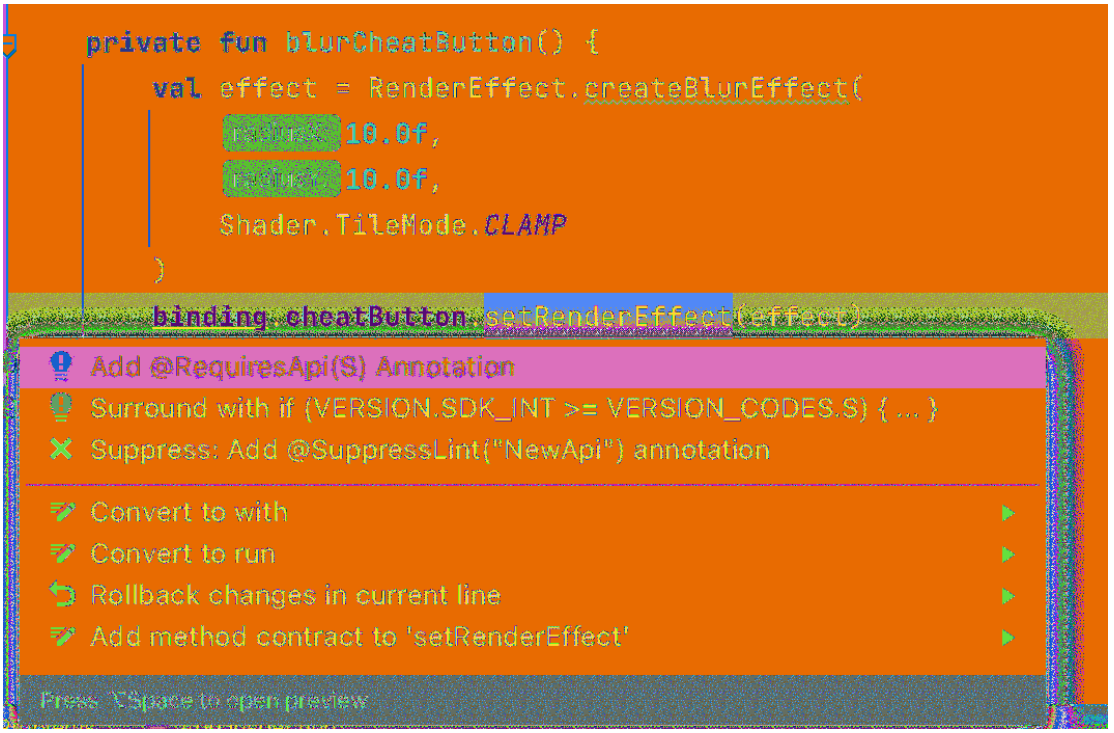
```
class MainActivity : AppCompatActivity() {
    ...
    override fun onCreate(savedInstanceState: Bundle?) {
        ...
        binding.cheatButton.setOnClickListener {
            ...
        }

        updateQuestion()

        blurCheatButton()
    }
    ...
}
```

The same Android Lint error appears (Figure 8.2). (If you do not see it right away, try rebuilding your project with Build → Rebuild Project.)

Figure 8.2 Android Lint suggestions



The `@RequiresApi` annotation by itself does not resolve the compatibility issue – it makes callers responsible for ensuring compatibility. To safely call your new function, you need to wrap the higher API code in a conditional statement that checks the device's version of Android.

### Listing 8.4 Checking the device's Android version first (`MainActivity.kt`)

```
class MainActivity : AppCompatActivity() {
    ...
    override fun onCreate(savedInstanceState: Bundle?) {
        ...
        binding.cheatButton.setOnClickListener {
            ...
        }

        updateQuestion()

        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.S) {
            blurCheatButton()
        }
    }
    ...
}
```

The `Build.VERSION.SDK_INT` constant contains the API level for the version of Android used by the device. You compare that version with the constant that stands for the S (API level 31) release. (Version codes are listed at `developer.android.com/reference/android/os/Build.VERSION_CODES.html`.)

Now, your blurring code will only be called when the app is running on a device with API level 31 or higher. You have made your code safe for API level 24, and Android Lint should now be content.

Run GeoQuiz on a device running API level 31 and check out your new blurred cheat button, then run it on a device running a lower API level to ensure that it works as before.

## Using the Android Developer Documentation

Android Lint errors will tell you what API level your incompatible code is from. But you can also find out which API level particular classes and functions belong to in Android's developer documentation.

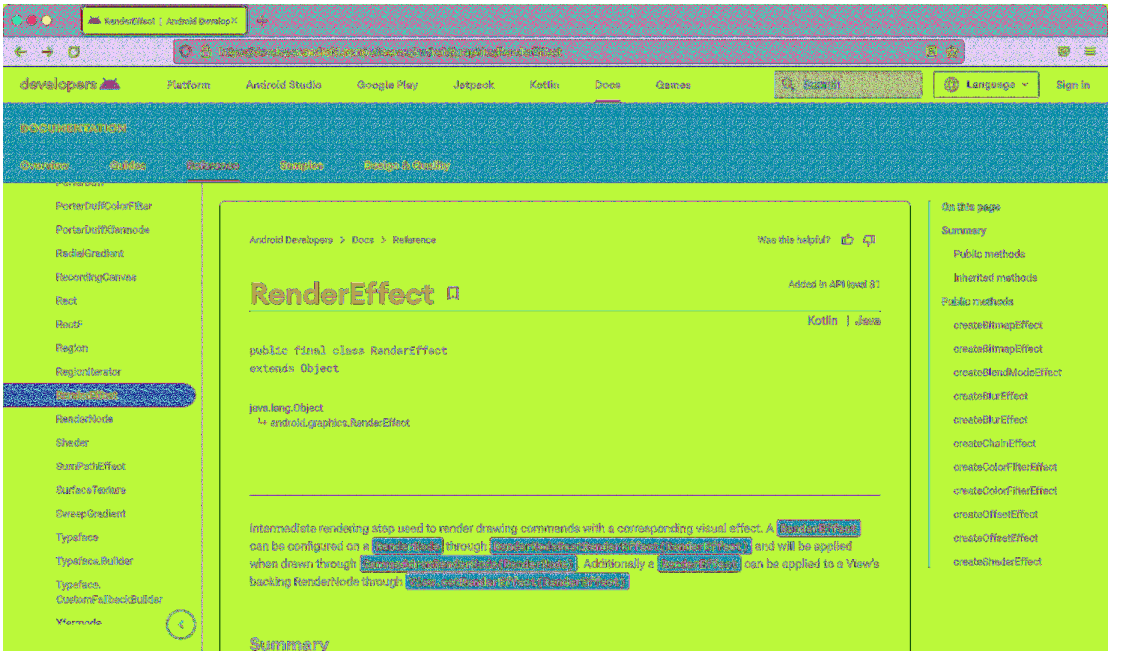
It is a good idea to get comfortable using the developer documentation right away. There is far too much in the Android SDKs to keep in your head. And with new versions appearing regularly, you will often need to find out what is new and how to use it.

The main page of the documentation is `developer.android.com`. It is split into seven parts: Platform, Android Studio, Google Play, Jetpack, Kotlin, Docs, and Games. It is all worth perusing when you get a chance. Each section outlines a different aspect of Android development, from just getting started to deploying your app to the Play Store.

| | |
|---|---|
| Platform | Information on the basic platform, focusing on the supported form factors and the different Android versions. |
| Android Studio | Articles on the IDE to help you learn tools and workflows to make your life as a developer easier. |
| Google Play | Tips and tricks for deploying your apps as well as making your apps more successful with users. |
| Jetpack | Information about the Jetpack libraries and how the Android team is striving to improve the app development experience. Some of the Jetpack libraries are used in this book, but you should explore this section for the full list. |
| Kotlin | Documentation on how to develop Android apps with Kotlin. |
| Docs | The main page for the developer documentation. Here you will find information on individual classes as well as a trove of tutorials and codelabs that you can work through to improve your skills. |
| Games | Documentation for making games that run on Android. |

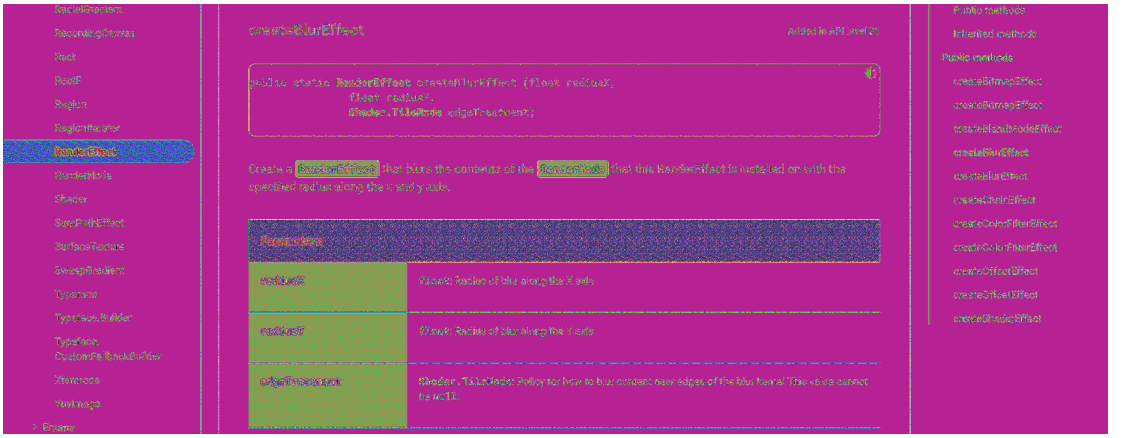
Open the developer documentation website and click the Docs tab. In the search bar at the top right, enter RenderEffect.createBlurEffect to determine what API level the function belongs to. Select the **`RenderEffect`** result (which is likely the first search result), and you will be taken to the class reference page (Figure 8.3). On the right side of this page are links to its different sections.

Figure 8.3 **`RenderEffect`** reference page




Find the **`createBlurEffect(…)`** function in the list on the right and click the function name to see a description. To the right of the function signature, you can see that **`createBlurEffect(…)`** was introduced in API level 31.

Figure 8.4 **`createBlurEffect(…)`** documentation

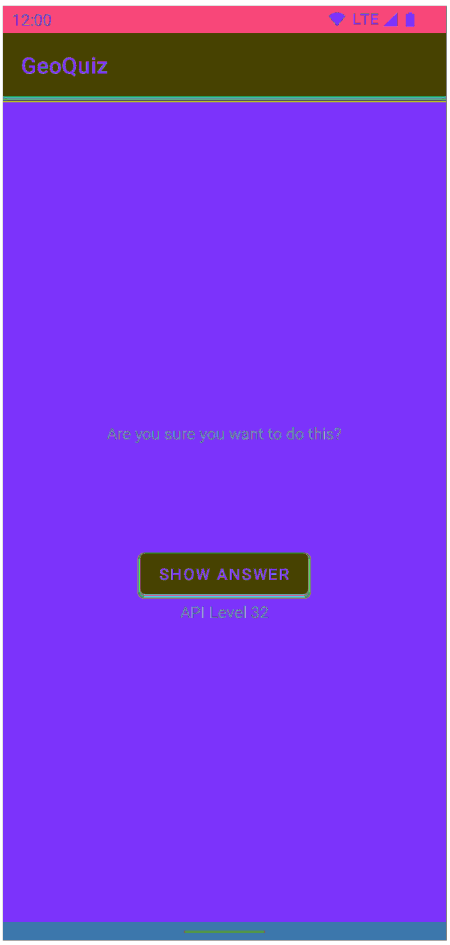


As you continue through this book, visit the developer documentation often. You will certainly need the documentation to tackle the challenge exercises, but you should also explore it whenever you get curious about particular classes, functions, or other topics. Android is constantly updating and improving the documentation, so there is always something new to learn.

## Challenge: Reporting the Device's Android Version

Add a **`TextView`** to the GeoQuiz layout that reports to the user what API level the device is running. Figure 8.5 shows what the final result should look like.

Figure 8.5 Finished challenge




You cannot set this **`TextView`**'s text in the layout, because you will not know the device's Android version until runtime. Find the **`TextView`** function for setting text in the **`TextView`** reference page in Android's documentation. You are looking for a function that accepts a single argument – a string (or a **`CharSequence`**).

Use other XML attributes listed in the **`TextView`** reference to adjust the size or typeface of the text.

## Challenge: Limited Cheats

Allow the user to cheat a maximum of three times. Keep track of the user's cheat occurrences and display the number of remaining cheat tokens below the cheat button. If no tokens remain, disable the cheat button.

# 9

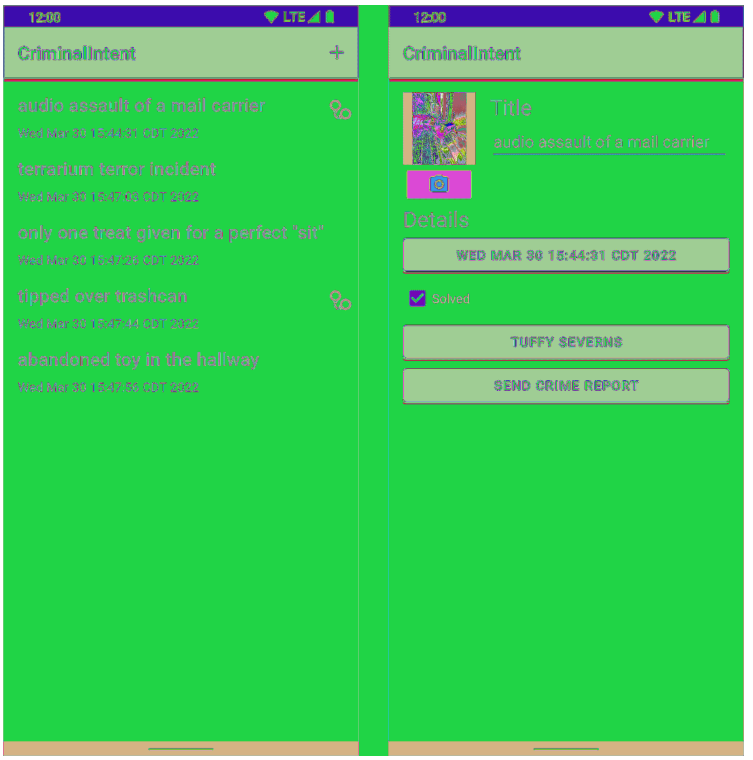
# Fragments

In this chapter, you will start building an application named CriminalIntent. CriminalIntent records the details of “office crimes” – things like leaving dirty dishes in the break room sink or walking away from an empty shared printer after documents have printed.

With CriminalIntent, you can make a record of a crime including a title, a date, and a photo. You can also identify a suspect from your contacts and lodge a complaint via email, Twitter, Facebook, or another app. After documenting and reporting a crime, you can proceed with your work free of resentment and ready to focus on the business at hand.

CriminalIntent is a complex app that will take 11 chapters to complete. It will have a *list-detail interface*: The main screen will display a list of recorded crimes, and users will be able to add new crimes or select an existing crime to view and edit its details (Figure 9.1).

Figure 9.1 CriminalIntent, a list-detail app



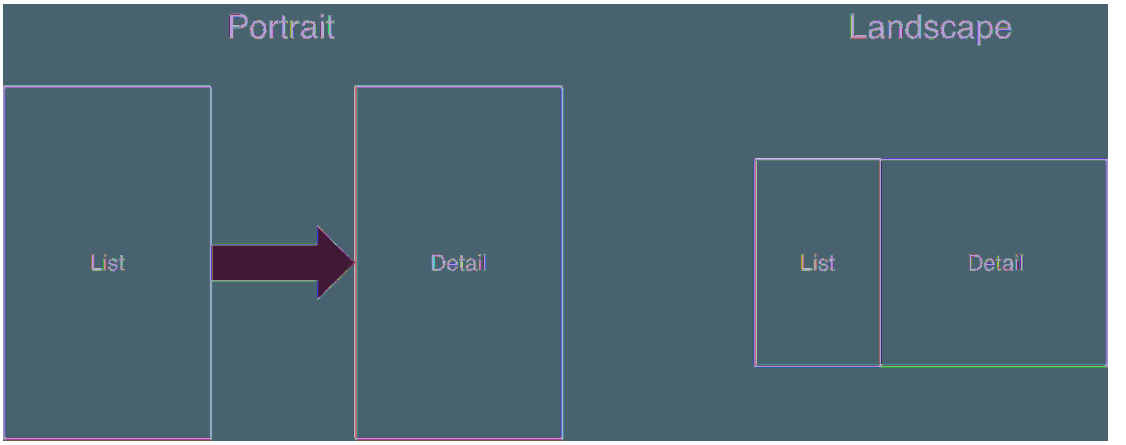
## The Need for UI Flexibility

You might imagine that a list-detail application consists of two activities: one managing the list and the other managing the detail view. Pressing a crime in the list would start an instance of the detail activity. Pressing the Back button would destroy the detail activity and return you to the list, where you could select another crime.

That would work, but what if you wanted more sophisticated presentation and navigation between screens?

Consider the possibility of CriminalIntent running on a large device. Some devices have screens large enough to show the list and detail at the same time – at least in landscape orientation (Figure 9.2).

Figure 9.2 Ideal list-detail interface for varying screen widths




Or imagine a user is viewing a crime and wants to see the next crime in the list. It would be better if they could select a different crime from the list without navigating back to the previous screen first. Going beyond the CriminalIntent app, common UI elements such as navigation drawers and bottom tab bars keep users on one "screen" while child views are swapped in and out.

What these scenarios have in common is UI flexibility: the ability to compose and recompose an activity's view at runtime depending on what the user or the device requires.

Activities were not designed with this level of flexibility in mind. Activities control an entire window of your application, so one activity should be able to render everything your app needs to show onscreen at a time. As a result, activities are tightly coupled to the particular screen being used. So you *could* continue to keep all your UI code within activities, but as your apps and the screens within them become more complicated, this approach will become more confusing and less maintainable.

# Introducing Fragments

You can make your app's UI more flexible by moving UI management from the activity to one or more *fragments*.

Similar to how you have used your activities so far, a **`Fragment`** has a view of its own, often defined in a separate layout file. The fragment's view contains the interesting UI elements that the user wants to see and interact with.

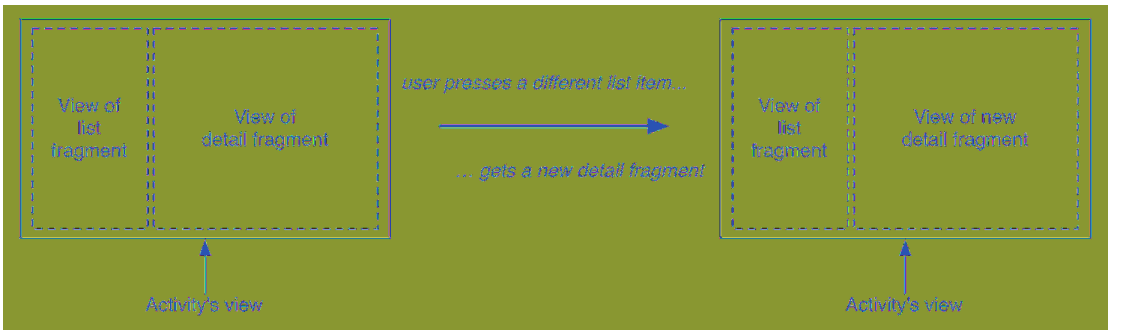
Instead of containing the UI, the activity acts as a container for the fragment. The fragment's view is inserted into the container once it is initialized. In this chapter, the activity will host a single fragment, but an activity can have multiple containers in its view for different fragments.

Fragments are designed to hold reusable chunks of the UI. You can use the fragment (or fragments) associated with the activity to compose and recompose the screen as your app and users require. There is only one **`Activity`** class responsible for displaying your app's content, but it hands control over parts of the screen to its **`Fragment`**s. Because of this, your activity will be much simpler and will not violate any Android laws.

Let's see how this would work in a list-detail application to display the list and detail together. You would compose the activity's view from a list fragment and a detail fragment. The detail view would show the details of the selected list item.

Selecting another item should display a new detail view. This is easy with fragments: Your app would replace the detail fragment with another detail fragment (Figure 9.3). No activities need to die for this major view change to happen.

Figure 9.3 Swapping out a detail fragment

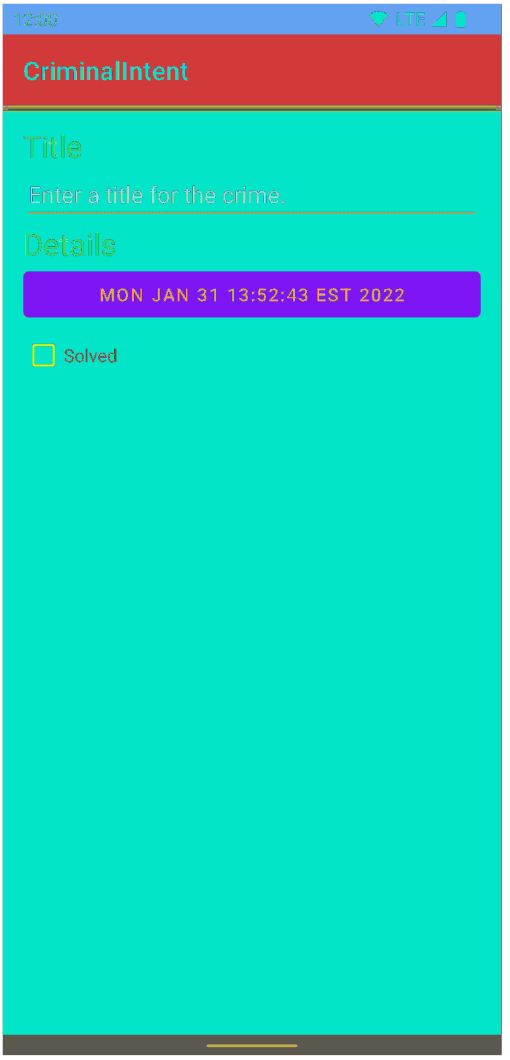



Using fragments separates the UI of your app into building blocks, which is useful for more than just list-detail applications. Working with individual blocks, it is easy to build tab interfaces, tack on animated sidebars, and more. Additionally, some of the new Android Jetpack APIs, such as the navigation controller, work best with fragments. So using fragments sets you up to integrate nicely with Jetpack APIs.

## Starting CriminalIntent

In this chapter, you are going to start on the detail part of CriminalIntent. Figure 9.4 shows you what CriminalIntent will look like at the end of this chapter.

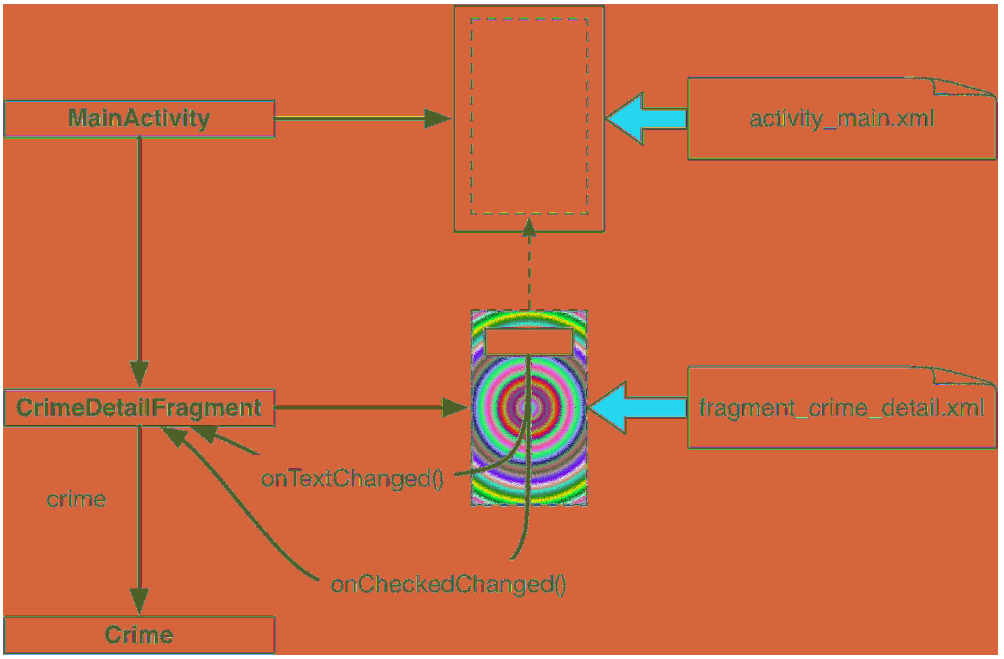
Figure 9.4 CriminalIntent at the end of this chapter




The screen shown in Figure 9.4 will be managed by a fragment named **`CrimeDetailFragment`**. An instance of **`CrimeDetailFragment`** will be *hosted* by an activity named **`MainActivity`**.

For now, think of hosting as the activity providing a spot in its view hierarchy to contain the fragment and its view (Figure 9.5). A fragment is incapable of getting a view onscreen itself. Only when it is inserted in an activity's hierarchy will its view appear.

Figure 9.5 **`MainActivity`** hosting a **`CrimeDetailFragment`**




By the end of the project, CriminalIntent will be a large codebase, but you will begin much like the way you built GeoQuiz. After some build setup, you will define the **`Crime`** class, which will model the data you are displaying. Next, you will create the UI in an XML layout in `fragment_crime_detail.xml`. Once that is complete, you will create a **`CrimeDetailFragment`** to hook up the data to the view.

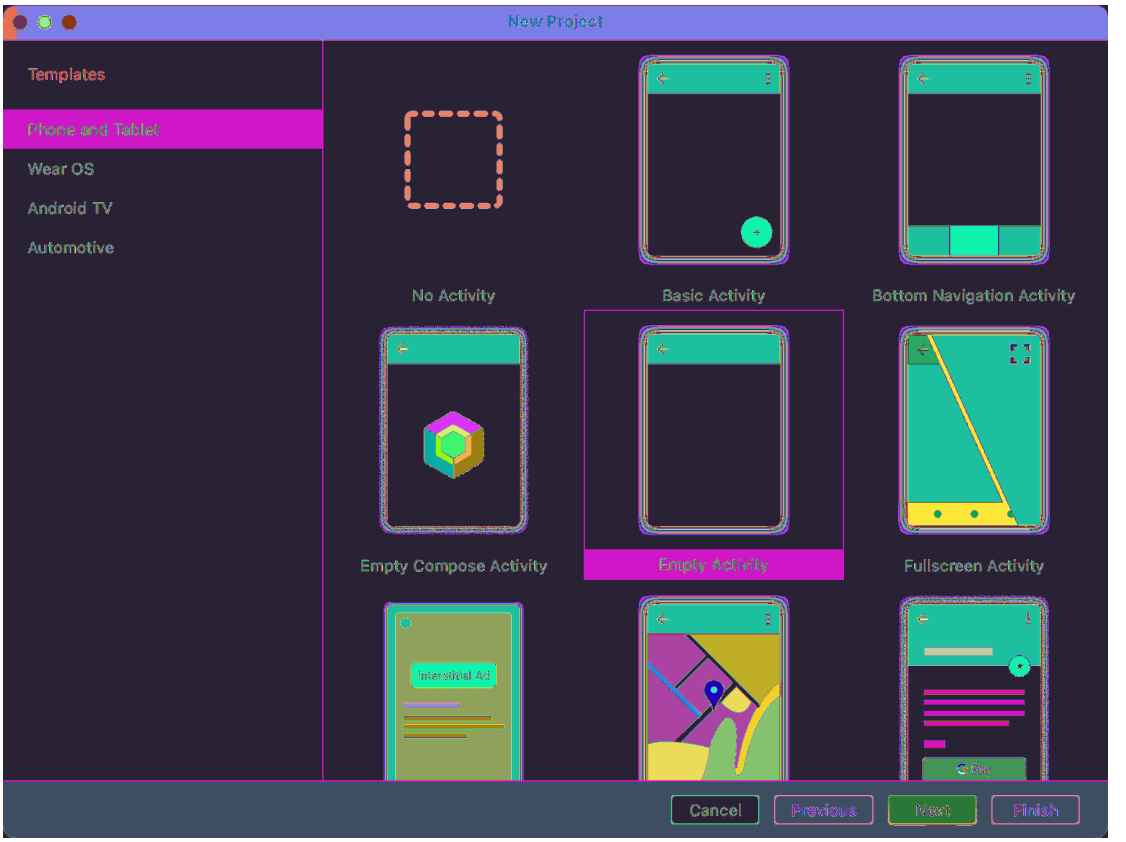
Those steps will feel familiar to the work you did back in GeoQuiz, even if the names are different this time. Since you are now working with fragments, you will also have to take care of one other step: You will add the **`CrimeDetailFragment`** to a container within **`MainActivity`**.

Let's get started.

## Creating a new project

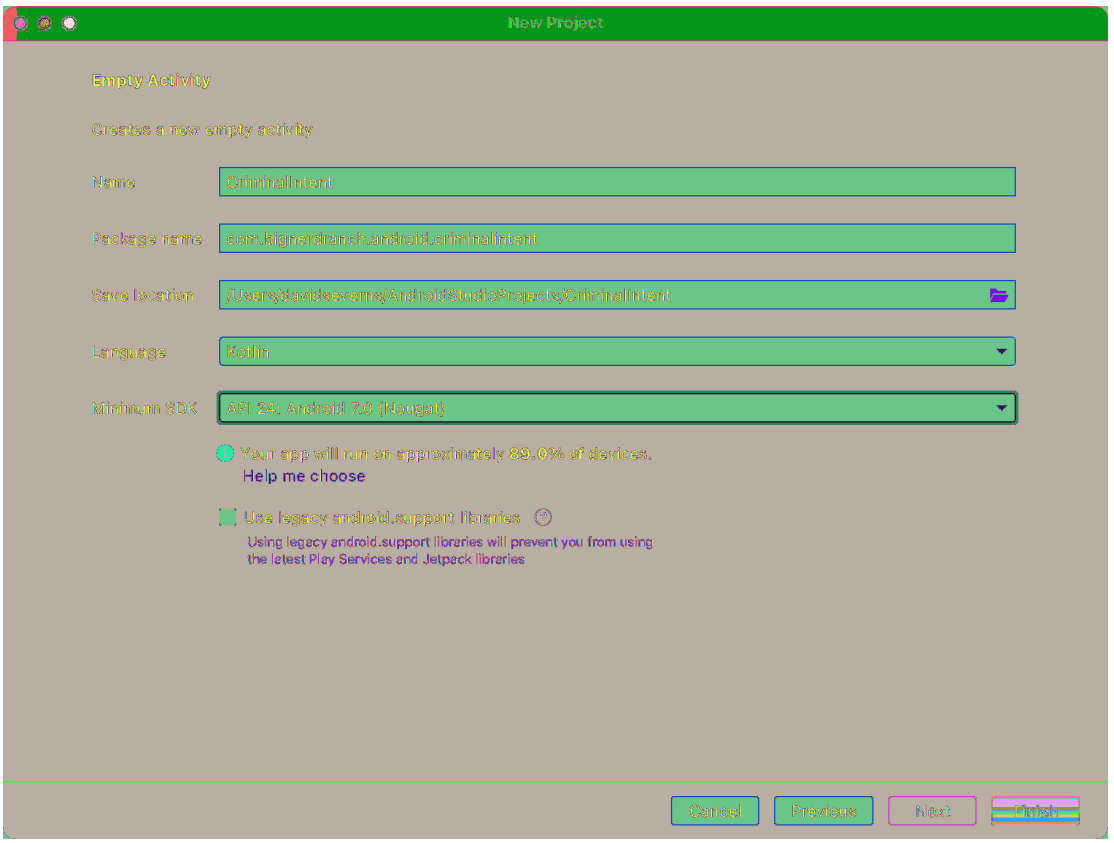
Create a new Android application (File → New → New Project...). Select the Empty Activity template (Figure 9.6). Click Next.

Figure 9.6 Creating the CriminalIntent application



Configure your project as shown in Figure 9.7: Name the application CriminalIntent. Make sure the Package name is com.bignerdranch.android.criminalintent and the Language is Kotlin. Select API 24: Android 7.0 (Nougat) from the Minimum SDK dropdown.

Figure 9.7 Configuring the CriminalIntent project




Click Finish to generate the project.

Before writing code, you need to make two changes to your Gradle build files. Open the `build.gradle` file labeled (Module: CriminalIntent.app). Like the ViewModel library you used in GeoQuiz, the Fragment library must be added as a dependency on your project. Also, enable View Binding, as you did for GeoQuiz. View Binding integrates seamlessly with fragments, and you will be using it in this project as well.

### Listing 9.1 Setting up your project's build (`app/build.gradle`)

```
...
android {
    ...
    kotlinOptions {
        jvmTarget = '1.8'
    }
    buildFeatures {
        viewBinding true
    }
}

dependencies {
    ...
    implementation 'androidx.constraintlayout:constraintlayout:2.1.3'
    implementation 'androidx.fragment:fragment-ktx:1.4.1'
    testImplementation 'junit:junit:4.13.2'
    ...
}
```

Do not forget to click the  Sync Project with Gradle Files button or the Sync Now button after you have made these changes. Now, on to the code.

## Creating a Data Class

In the project tool window, right-click the `com.bignerdranch.android.criminalintent` package and select New → Kotlin Class/File. Name the file Crime and, since this class will be used to store data, make it a Data Class.

For this project, an instance of **`Crime`** will represent a single office crime. To begin with, a **`Crime`** will have four properties:

- an ID to uniquely identify the instance
- a descriptive title, like “Toxic sink dump” or “Someone stole my yogurt!”
- a date
- a Boolean indication of whether the crime has been solved

In `Crime.kt`, add these four properties to **`Crime`**'s constructor.

### Listing 9.2 Adding the **`Crime`** data class (`Crime.kt`)

```
data class Crime(
    val id: UUID,
    val title: String,
    val date: Date,
    val isSolved: Boolean
)
```

When importing **`Date`**, you will be presented with multiple options. Make sure to import **`java.util.Date`**.

That is all you need for the **`Crime`** class for this chapter. Now that the data is set up, let’s move on to the **`CrimeDetailFragment`**.

# Creating a Fragment

The steps to create a fragment are the same as those you followed to create an activity:

- compose a UI by defining views in a layout file
- create the class and set its view to be the layout that you defined
- wire up the views inflated from the layout in code

## Defining CrimeDetailFragment's layout

**`CrimeDetailFragment`**'s view will display the information contained in an instance of **`Crime`**.

First, define the strings that the user will see in `res/values/strings.xml`.

### Listing 9.3 Adding strings (`res/values/strings.xml`)

```
<resources>
    <string name="app_name">CriminalIntent</string>
    <string name="crime_title_hint">Enter a title for the crime.</string>
    <string name="crime_title_label">Title</string>
    <string name="crime_details_label">Details</string>
    <string name="crime_solved_label">Solved</string>
</resources>
```

Next, you will define the UI. The layout for **`CrimeDetailFragment`** will consist of a vertical **`LinearLayout`** that contains two **`TextView`**s, an **`EditText`**, a **`Button`**, and a **`CheckBox`**.

To create a layout file, right-click the res/layout folder in the project tool window and select New → Layout resource file. Name this file fragment_crime_detail.xml and enter LinearLayout as the root element.

Android Studio creates the file and adds the **`LinearLayout`** for you. Add the views that make up the fragment's layout to `res/layout/fragment_crime_detail.xml`.

### Listing 9.4 Layout file for fragment's view (`res/layout/fragment_crime_detail.xml`)

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_margin="16dp">

    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:textAppearance="?attr/textAppearanceHeadline5"
        android:text="@string/crime_title_label" />

    <EditText
        android:id="@+id/crime_title"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="@string/crime_title_hint"
        android:importantForAutofill="no"
        android:inputType="text" />

    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:textAppearance="?attr/textAppearanceHeadline5"
        android:text="@string/crime_details_label" />

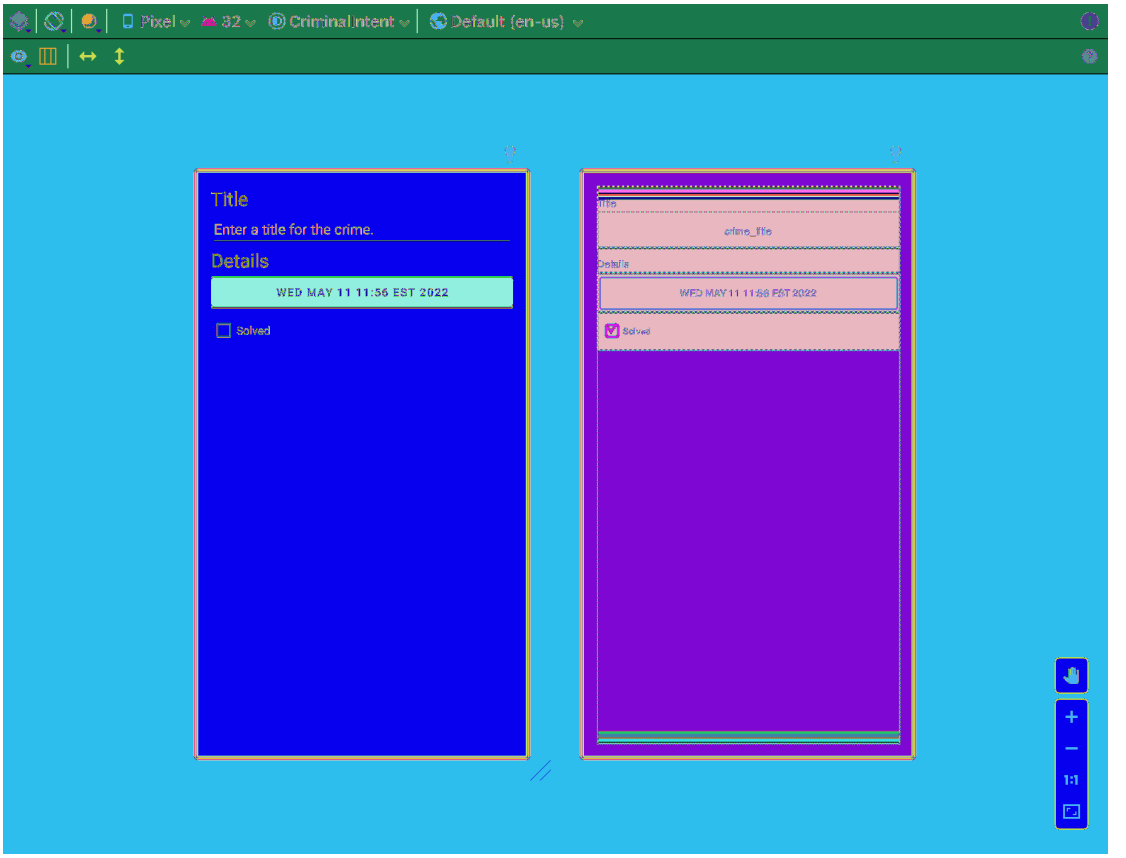
    <Button
        android:id="@+id/crime_date"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        tools:text="Wed May 11 11:56 EST 2022" />

    <CheckBox
        android:id="@+id/crime_solved"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/crime_solved_label" />
</LinearLayout>
```

(The **`TextView`**s' definitions include some new syntax related to view style: `textAppearance="?attr/textAppearanceHeadline5"`. This *theme attribute* applies the Headline 5 typography settings to the text as specified by Google's Material Design library. [It can also be customized in your application theme, if you want.] You will learn more about this syntax in the section called Styles, Themes, and Theme Attributes in Chapter 11.)

Recall that the `tools` namespace allows you to provide information that the preview is able to display. In this case, you are adding text to the date button so that it will not be empty in the preview. Check the Design tab to see a preview of your fragment's view (Figure 9.8).

Figure 9.8 Previewing updated crime fragment layout




## Creating the CrimeDetailFragment class

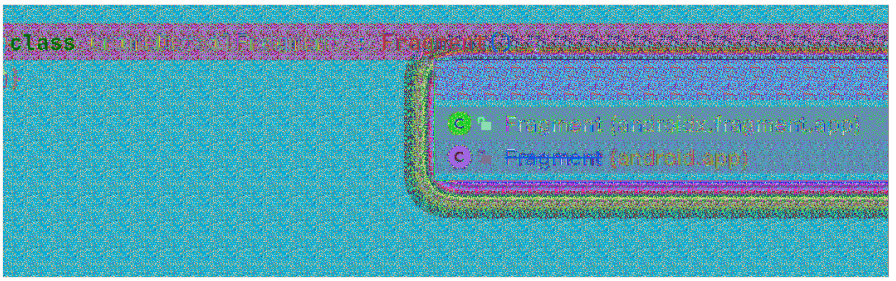
Create a Kotlin file for the **`CrimeDetailFragment`** class. This time, select Class for the file type, and Android Studio will stub out the class definition for you. Turn the class into a fragment by subclassing the **`Fragment`** class.

Listing 9.5 Subclassing **`Fragment`** (`CrimeDetailFragment.kt`)

```
class CrimeDetailFragment : Fragment() {
}
```

As you subclass the **`Fragment`** class, you will notice that Android Studio finds two classes with the **`Fragment`** name. You will see **`android.app.Fragment`** and **`androidx.fragment.app.Fragment`**. The **`android.app.Fragment`** is the version of fragments built into the Android OS. You will use the Jetpack version, so be sure to select **`androidx.fragment.app.Fragment`**, as shown in Figure 9.9. (Recall that the Jetpack libraries are in packages that begin with `androidx`.)

Figure 9.9 Choosing the Jetpack **`Fragment`** class




If you do not see this dialog, try clicking the **`Fragment`** class name. If the dialog still does not appear, you can manually import the correct class: Add the line `import androidx.fragment.app.Fragment` at the top of the file.

If, on the other hand, you have an import for **`android.app.Fragment`**, remove that line of code. Then import the correct **`Fragment`** class with Option-Return (Alt-Enter).

## Different types of fragments

New Android apps should always be built using the Jetpack (`androidx`) version of fragments. If you maintain older apps, you may see two other versions of fragments being used: the framework version and the v4 support library version. These are legacy versions of the **`Fragment`** class, and you should migrate apps that use them to the current Jetpack version.

Fragments were introduced in API level 11, when the first Android tablets created the need for UI flexibility. The framework implementation of fragments was built into devices running API level 11 or higher. Shortly afterward, a **`Fragment`** implementation was added to the v4 support library to enable fragment support on older devices. With each new version of Android, both of these fragment versions were updated with new features and security patches.

But as of Android 9.0 (API 28), the framework version of fragments is deprecated and the earlier support library fragments have been moved to the Jetpack libraries. No further updates will be made to either of those versions, so you should not use them for new projects. All future updates will apply only to the Jetpack version.

Bottom line: Always use the Jetpack fragments in your new projects, and migrate existing projects to ensure they stay current with new features and bug fixes.

### Implementing fragment lifecycle functions

**`CrimeDetailFragment`** is the class that interacts with model and view objects. Its job is to present the details of a specific crime and update those details as the user changes them.

In GeoQuiz, your activities did most of that work in activity lifecycle functions. In CriminalIntent, this work will be done by fragments in fragment lifecycle functions. Many of these functions correspond to the **`Activity`** functions you already know, such as **`onCreate(Bundle?)`**. (You will learn more about the fragment lifecycle in the section called The fragment lifecycle later in this chapter.)

In `CrimeDetailFragment.kt`, add a property for the **`Crime`** instance and an implementation of **`Fragment.onCreate(Bundle?)`**.

Android Studio can provide some assistance when overriding functions. Begin typing **`onCreate(Bundle?)`**; Android Studio will provide a list of suggestions, as shown in Figure 9.10.

Figure 9.10 Overriding the **`onCreate(Bundle?)`** function



Press Return to select the option to override the **`onCreate(Bundle?)`** function, and Android Studio will create the declaration for you, including the call to the superclass implementation. Update your code to create a new **`Crime`**, matching Listing 9.6.

### Listing 9.6 Overriding **`Fragment.onCreate(Bundle?)`** (`CrimeDetailFragment.kt`)

```
class CrimeDetailFragment : Fragment() {

    private lateinit var crime: Crime

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        crime = Crime(
            id = UUID.randomUUID(),
            title = "",
            date = Date(),
            isSolved = false
        )
    }
}
```

Much like activities, fragments are re-created on configuration changes by default, so they are not good places to hold state. In Chapter 13, you will use a **`ViewModel`** to hold this state, but this will work for now.

Kotlin functions default to public when no visibility modifier is included in the definition. So **`Fragment.onCreate(Bundle?)`**, which has no visibility modifier, is public. This differs from the **`Activity.onCreate(Bundle?)`** function, which is protected. **`Fragment.onCreate(Bundle?)`** and other **`Fragment`** lifecycle functions must be public, because they will be called by whichever activity is hosting the fragment.

Also, note what does *not* happen in **`Fragment.onCreate(Bundle?)`**: You do not inflate the fragment's view. You configure the fragment instance in **`Fragment.onCreate(Bundle?)`**, but you create and configure the fragment's view in another fragment lifecycle function: **`onCreateView(LayoutInflater, ViewGroup?, Bundle?)`**.

This function is where you inflate and bind the layout for the fragment's view and return the inflated **`View`** to the hosting activity. The **`LayoutInflater`** and **`ViewGroup`** parameters are necessary to inflate and bind the layout. The **`Bundle`** will contain data that this function can use to re-create the view from a saved state.

In `CrimeDetailFragment.kt`, add an implementation of **`onCreateView(…)`** that inflates and binds `fragment_crime_detail.xml`. You can use the same trick from Figure 9.10 to fill out the function declaration.

### Listing 9.7 Overriding **`onCreateView(…)`** (`CrimeDetailFragment.kt`)

```
class CrimeDetailFragment : Fragment() {

    private lateinit var binding: FragmentCrimeDetailBinding

    private lateinit var crime: Crime

    override fun onCreate(savedInstanceState: Bundle?) {
        ...
    }

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        binding =
            FragmentCrimeDetailBinding.inflate(inflater, container, false)
        return binding.root
    }
}
```

Much like in GeoQuiz, View Binding will generate a binding class that you can use to inflate and bind your layout. This time it is called **`FragmentCrimeDetailBinding`**.

As before, you call the **`inflate(…)`** function to accomplish the task. However, this time you call a slightly different version of the function – one that takes in three parameters instead of one. The first parameter is the same **`LayoutInflater`** you used before. The second parameter is your view's parent, which is usually needed to configure the views properly.

The third parameter tells the layout inflater whether to immediately add the inflated view to the view's parent. You pass in `false` because the fragment's view will be hosted in the activity's container view. The fragment's view does not need to be added to the parent view immediately – the activity will handle adding the view later.

Once you return the root view within the `binding`, you are ready to start wiring up the views.

### Wiring up views in a fragment

You are now going to hook up the **EditText**, **CheckBox**, and **Button** in your fragment. Your first instinct might be to add some code to **onCreateView(…)**, but it is best if you keep **onCreateView(…)** simple and do not do much more there than bind and inflate your view. The **onViewCreated(…)** lifecycle callback is invoked immediately after **onCreateView(…)**, and it is the perfect spot to wire up your views.

Start by adding a listener to the **EditText** in the **onViewCreated(…)** lifecycle callback.

Listing 9.8 Adding a listener to the **EditText** view (CrimeDetailFragment.kt)

```
class CrimeDetailFragment : Fragment() {
    ...
    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        ...
    }

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        binding.apply {
            crimeTitle.doOnTextChanged { text, _, _, _ ->
                crime = crime.copy(title = text.toString())
            }
        }
    }
}
```

Setting listeners in a fragment works exactly the same as in an activity. Here, you add a listener that will be invoked whenever the text in the **EditText** is changed. The lambda is invoked with four parameters, but you only care about the first one, `text`. The `text` is provided as a **CharSequence**, so to set the **Crime**'s title, you call **toString()** on it.

(The **doOnTextChanged()** function is actually a Kotlin extension function on the **EditText** class. Do not forget to import it from the `androidx.core.widget` package.)

When you are not using a parameter, like the remaining lambda parameters here, you name it _. Lambda arguments named _ are ignored, which removes unnecessary variables and can help keep your code tidy.

Next, connect the **`Button`** to display the date of the crime.

### Listing 9.9 Setting **`Button`** text (`CrimeDetailFragment.kt`)

```
class CrimeDetailFragment : Fragment() {
    ...
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        binding.apply {
            crimeTitle.doOnTextChanged { text, _, _, _ ->
                crime = crime.copy(title = text.toString())
            }

            crimeDate.apply {
                text = crime.date.toString()
                isEnabled = false
            }
        }
    }
}
```

Disabling the button ensures that it will not respond to the user pressing it. It also changes its appearance to advertise its disabled state. In Chapter 14, you will enable the button and allow the user to choose the date of the crime.

The last change you need to make within this class is to set a listener on the **`CheckBox`** that will update the `isSolved` property of the **`Crime`**, as shown in Listing 9.10.

### Listing 9.10 Listening for **`CheckBox`** changes (`CrimeDetailFragment.kt`)

```
class CrimeDetailFragment : Fragment() {
    ...
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        binding.apply {
            crimeTitle.doOnTextChanged { text, _, _, _ ->
                crime = crime.copy(title = text.toString())
            }

            crimeDate.apply {
                text = crime.date.toString()
                isEnabled = false
            }

            crimeSolved.setOnCheckedChangeListener { _, isChecked ->
                crime = crime.copy(isSolved = isChecked)
            }
        }
    }
}
```

It would be great if you could run CriminalIntent and play with the code you have written. But you cannot – yet. Remember, fragments cannot put their views onscreen on their own. To realize your efforts, you first have to add a **`CrimeDetailFragment`** to **`MainActivity`**.

# Hosting a Fragment

When fragments were first introduced, developers had to jump through numerous hoops to display them. In 2019, Google introduced the **FragmentContainerView**, which makes it easier to create host containers for a fragment. In this section, you will use a **FragmentContainerView** to host your **CrimeDetailFragment**. Then you will learn about the **FragmentManager** and the fragment lifecycle. Finally, you will tie up one loose end in **CrimeDetailFragment**.

## Defining a FragmentContainerView

**FragmentContainerView** is, as its name suggests, built to contain fragments. Fragments have changed significantly over the years, so **FragmentContainerView** helps provide a consistent environment for fragments to operate in. Much like the views you have used so far, the **FragmentContainerView** has common XML attributes to define its ID and its size.

Locate and open **MainActivity**'s layout in `res/layout/activity_main.xml`. Replace the default layout with a **FragmentContainerView**, as shown in Listing 9.11.

Listing 9.11 Creating the fragment container layout (`res/layout/activity_main.xml`)

```
<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">

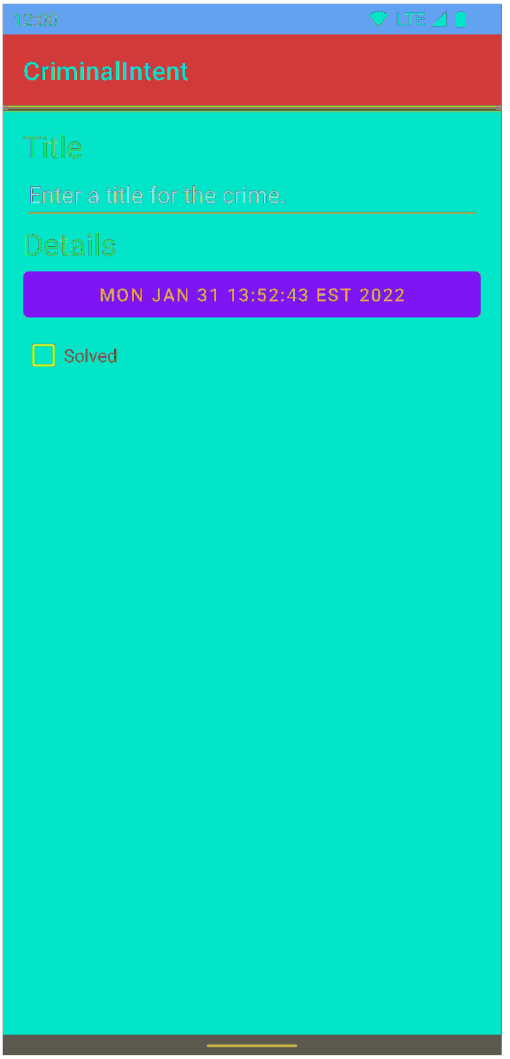
    <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Hello World!"
            app:layout_constraintBottom_toBottomOf="parent"
            app:layout_constraintLeft_toLeftOf="parent"
            app:layout_constraintRight_toRightOf="parent"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

</androidx.constraintlayout.widget.ConstraintLayout>
<androidx.fragment.app.FragmentContainerView
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/fragment_container"
    android:name="com.bignerdranch.android.criminalintent.CrimeDetailFragment"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity" />
```

**FragmentContainerView** has one XML attribute that you have not seen on other views: `android:name`, whose value here is the full package name for **CrimeDetailFragment**. With that, the **FragmentContainerView** will manage creating your **CrimeDetailFragment** and inserting it in the activity's layout.

At last, it is time to run CriminalIntent to check your code. You will see your **`CrimeDetailFragment`** below an app bar that shows CriminalIntent's name (Figure 9.11). (The app bar – the toolbar at the top of your app – is included automatically because of the way you configured your activity. You will learn more about the app bar in Chapter 15.)

Figure 9.11 **`CrimeDetailFragment's`** view hosted by **`MainActivity`**

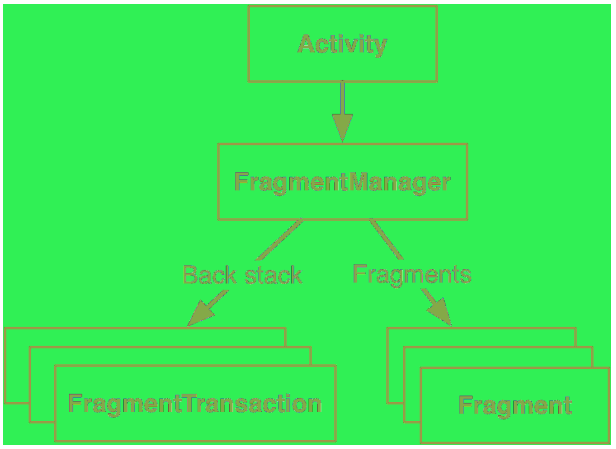



Now that you have seen the results of your work, let's go behind the scenes and discuss how fragments and their lifecycles are managed.

## The FragmentManager

When the **`Fragment`** class was introduced in Honeycomb, the **`Activity`** class was changed to include a piece called the **`FragmentManager`**. The **`FragmentManager`** is responsible for adding the fragments' views to the activity's view hierarchy and driving the fragments' lifecycles. It handles two things: a list of fragments and a back stack of fragment transactions (which you will learn about shortly) (Figure 9.12).

Figure 9.12 The **`FragmentManager`**




Right now, your **`FragmentContainerView`** interacts with the **`FragmentManager`** to display your **`CrimeDetailFragment`**. The **`FragmentContainerView`** uses the **`FragmentManager`** to create and host the fragment you specified in the `android:name` XML attribute.

As an alternative to using the `android:name` XML attribute, you can attach fragments to your activities in code with the **`FragmentManager`**. Also, in addition to the basic functionality provided by your **`FragmentContainerView`**, you can use the **`FragmentManager`** to remove a fragment from view, replace it with another, and even alter the navigation backstack.

To add a fragment to an activity in code, you make explicit calls to the activity's **`FragmentManager`**. You can access the activity's fragment manager using the `supportFragmentManager` property. You use `supportFragmentManager` because you are using the Jetpack library and the **`AppCompatActivity`** class. (The name is prefixed with "support" because the property originated in the v4 support library, but the support library has since been repackaged as an `androidx` library within Jetpack.)

Actions such as adding, removing, or replacing fragments are accomplished using fragment transactions. They allow you to group multiple operations, such as adding multiple fragments to different containers at the same time. They are the heart of how you use fragments to compose and recompose screens at runtime.

The **FragmentManager** maintains a back stack of fragment transactions that you can navigate. If your fragment transaction includes multiple operations, they are reversed when the transaction is removed from the back stack. This provides more control over your UI state when you group your fragment operations into a single transaction.

```
val fragment = CrimeDetailFragment()
supportFragmentManager
    .beginTransaction()
    .add(R.id.fragment_container, fragment)
    .commit()
```

In this example, the **FragmentManager.beginTransaction()** function creates and returns an instance of **FragmentTransaction**. The **FragmentTransaction** class uses a *fluent interface* – functions that configure **FragmentTransaction** return a **FragmentTransaction** instead of `Unit`, which allows you to chain them together. So the code highlighted above says, "Create a new fragment transaction, include one add operation in it, and then commit it."

The **add(…)** function is the meat of the transaction. It has two parameters: a container view ID and the newly created **CrimeDetailFragment**. The container view ID should look familiar. It is the resource ID of the **FragmentContainerView** that you would define in an activity's layout.

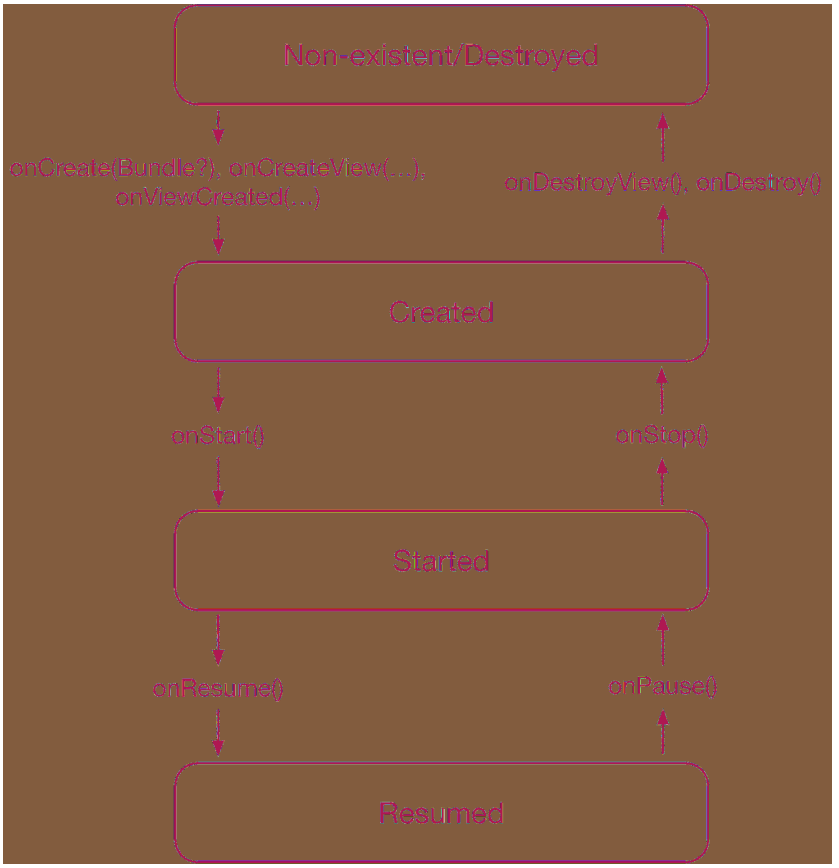
A container view ID serves two purposes:

- It tells the **FragmentManager** where in the activity's view the fragment's view should appear.
- It is used as a unique identifier for a fragment in the **FragmentManager**'s list.

## The fragment lifecycle

As we mentioned, another responsibility of the **`FragmentManager`** is driving the fragment lifecycle, which is shown in Figure 9.13. The fragment lifecycle is similar to the activity lifecycle: It has created, started, and resumed states, and it has functions you can override to get things done at critical points – many of which correspond to activity lifecycle functions.

Figure 9.13 Fragment lifecycle diagram




The correspondence is important. Because a fragment works on behalf of an activity, its state should reflect the activity's state. Thus, it needs corresponding lifecycle functions to handle the activity's work.

One critical difference between the fragment lifecycle and the activity lifecycle is that fragment lifecycle functions are called by the **`FragmentManager`** of the hosting activity, not the OS. The OS knows nothing about the fragments that an activity is using to manage things. Fragments are the activity's internal business. The **`onAttach(Context?)`**, **`onCreate(Bundle?)`**, **`onCreateView(…)`**, and **`onViewCreated(…)`** functions are called when you add the fragment to the **`FragmentManager`**.

The **`onActivityCreated(Bundle?)`** function is called after the hosting activity's **`onCreate(Bundle?)`** function has executed. You are adding the **`CrimeDetailFragment`** in **`MainActivity.onCreate(Bundle?)`**, so this function will be called after the fragment has been added.

What happens if you add a fragment while the activity is already resumed? In that case, the **`FragmentManager`** immediately walks the fragment through whatever steps are necessary to get it caught up to the activity's state. For example, as a fragment is added to an activity that is already resumed, that fragment gets calls to **`onAttach(Context?)`**, **`onCreate(Bundle?)`**, **`onCreateView(…)`**, **`onViewCreated(…)`**, **`onActivityCreated(Bundle?)`**, **`onStart()`**, and then **`onResume()`**.

Once the fragment's state is caught up to the activity's state, the hosting activity's **`FragmentManager`** will call further lifecycle functions around the same time that it receives the corresponding calls from the OS to keep the fragment's state aligned with that of the activity.

## Fragments and memory management

Fragments can be swapped in and out as the user navigates your app. For CriminalIntent, you will make another **`Fragment`** in Chapter 10 that displays a list of crimes. By the time you have completed developing the app, you will be able to navigate from the list fragment to the detail fragment, and your list will disappear from the user's view. Because the user can navigate back to the list screen, the fragment is retained in memory so it is ready to be used when the user presses the Back button.

But what about its view? Because the previous fragment is not being displayed, the system does not need to keep its view in memory. And, in fact, **`Fragment`** has a lifecycle method to destroy its view when it is no longer needed. This method is called **`onDestroyView()`**. When the **`Fragment`** becomes visible again, its **`onCreateView(…)`** method will be called again to re-create the view.

And here we come to the loose end we mentioned earlier: Although you have an **`onCreateView(…)`** callback in CriminalIntent, your view is not currently being freed from memory, because you are holding a reference to it via the `binding` property. The system sees that there is a chance you might access the view later and prevents the system from clearing its memory.

This wastes resources, since the view is being held in memory even though it is not used – and even though the view will be re-created when the **`Fragment`** becomes visible again. With your current implementation, the system cannot free the memory associated with your old view until either the view is re-created by calling **`onCreateView(…)`** again or the entire **`Fragment`** is destroyed.

The good news is that there is a straightforward solution to this problem: Null out any references to views in the **`onDestroyView()`** lifecycle callback. As long as you make sure to clean up any references to your views in **`onDestroyView()`**, you will be safe from the issues associated with this second lifecycle – and you benefit from a performance boost by freeing up unused resources.

### Listing 9.12 Nulling out references to your view (`CrimeDetailFragment.kt`)

```
class CrimeDetailFragment : Fragment() {
    ...
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        ...
    }

    override fun onDestroyView() {
        super.onDestroyView()
        binding = null
    }
}
```

After you make those changes, Android Studio will start complaining. Currently, your `binding` is not nullable. But with a few small changes, you can null out your references and have easy access to your `binding`. Create a nullable backing property, named `_binding`, and change the `binding` property to become a computed property. By using the **`checkNotNull()`** precondition, Kotlin will be able to smart cast the `binding` property to be non-null.

### Listing 9.13 Having the best of both worlds (`CrimeDetailFragment.kt`)

```
class CrimeDetailFragment : Fragment() {

    ~~private lateinit var binding: FragmentCrimeDetailBinding~~
    private var _binding: FragmentCrimeDetailBinding? = null
    private val binding
        get() = checkNotNull(_binding) {
            "Cannot access binding because it is null. Is the view visible?"
        }
    ...
    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        ~~binding =~~
            ~~FragmentCrimeDetailBinding.inflate(inflater, container, false)~~
        _binding =
            FragmentCrimeDetailBinding.inflate(inflater, container, false)
        ...
    }
    ...
    override fun onDestroyView() {
        super.onDestroyView()
        ~~binding = null~~
        _binding = null
    }
}
```

When accessing `binding`, you still have the benefit of a non-nullable property, but now you also have the backing `_binding` property that you can null out in **`onDestroyView()`**.

In this chapter, you used fragments to display an individual screen free of the limitations associated with activities. In the next chapter, you will create another fragment and leverage a **`RecyclerView`** to display your crimes in a list.

## Challenge: Testing with FragmentScenario

Much like the **`ActivityScenario`** class you used back in Chapter 6, Google has a corresponding **`FragmentScenario`** to test fragments in isolation. Built on the same infrastructure as **`ActivityScenario`**, **`FragmentScenario`** behaves in a similar fashion and uses a similar API. Try writing a test for your **`CrimeDetailFragment`** using a **`FragmentScenario`** with Espresso.

For example, you could test and verify that the **`CheckBox`** and **`EditText`** are hooked up to your fragment and update the **`Crime`**. By removing the `private` visibility modifier on the property and using the **`FragmentScenario.onFragment(…)`** function, you can get access to a **`Crime`** and perform the appropriate assertions.

**`FragmentScenario`** exists in a separate library, so do not forget to add the line below to your dependencies in the `build.gradle` file labeled (Module: CriminalIntent.app). Note the usage of `debugImplementation` – the **`FragmentScenario`** class works a little differently than other testing libraries you have used so far. Under the hood, the library inserts an activity into your app and uses it to host your fragment in a container it can control.

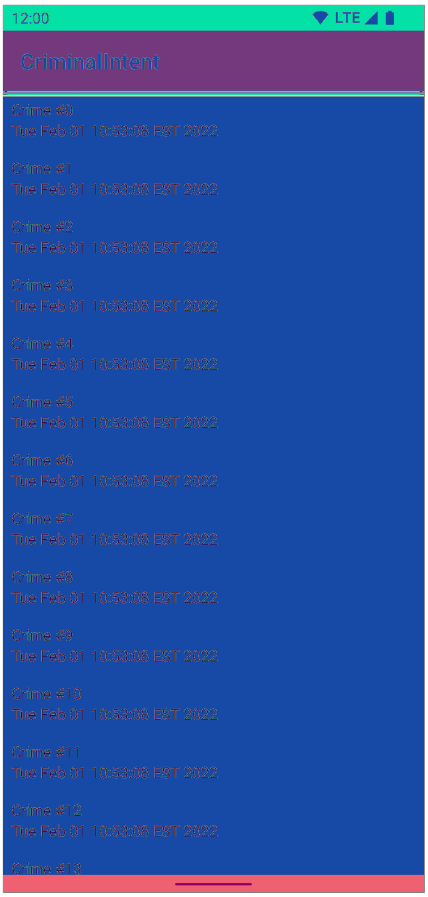
```
dependencies {
    ...
    debugImplementation "androidx.fragment:fragment-testing:1.4.1"
}
```

# 10

# Displaying Lists with RecyclerView

Currently, CriminalIntent can only display information about a single instance of **`Crime`**. In this chapter, you will update CriminalIntent to work with a list of crimes. The list will display each **`Crime`**'s title and date, as shown in Figure 10.1.

Figure 10.1 A list of crimes



Many aspects of the work you will complete in this chapter will feel familiar from the work you did in previous chapters. For example, much like you did in GeoQuiz, you will create a new **`ViewModel`** to encapsulate the data for the new screen. **`CrimeListViewModel`** will store a list of **`Crime`** objects.

Since this data will be displayed on a new screen, you will also create a new fragment, called **`CrimeListFragment`**. Your **`MainActivity`** will host an instance of **`CrimeListFragment`**, which in turn will display the list of crimes on the screen.

The activity's view will still consist of a single **`FragmentContainerView`**. The fragment's view will consist of a **`RecyclerView`**, a class that allows you to efficiently recycle views.

For now, the list and detail parts of CriminalIntent will lead separate lives. In Chapter 13, you will connect them.

# Adding a New Fragment and ViewModel

The first step is to add a **`ViewModel`** to store the **`List`** of **`Crime`** objects you will eventually display on the screen. As you learned in Chapter 4, the **`ViewModel`** class is part of the `lifecycle-viewmodel-ktx` library. So begin by adding the `lifecycle-viewmodel-ktx` dependency to your `app/build.gradle` file (that is, the `build.gradle` file labeled Module: CriminalIntent.app).

### Listing 10.1 Adding `lifecycle-viewmodel-ktx` dependency (`app/build.gradle`)

```
dependencies {
    ...
    implementation 'androidx.constraintlayout:constraintlayout:2.1.3'
    implementation 'androidx.fragment:fragment-ktx:1.4.1'
    implementation 'androidx.lifecycle:lifecycle-viewmodel-ktx:2.4.1'
    ...
}
```

Do not forget to sync your Gradle files after making this change.

Next, create a Kotlin class called **`CrimeListViewModel`**. Update the new **`CrimeListViewModel`** class to extend from **`ViewModel`**. Add a property to store a list of **`Crime`**s. In the `init` block, populate the list with dummy data.

### Listing 10.2 Generating crimes (`CrimeListViewModel.kt`)

```
class CrimeListViewModel : ViewModel() {

    val crimes = mutableListOf<Crime>()

    init {
        for (i in 0 until 100) {
            val crime = Crime(
                id = UUID.randomUUID(),
                title ="Crime #$i",
                date = Date(),
                isSolved = i % 2 == 0
            )

            crimes += crime
        }
    }
}
```

Eventually, the **List** will contain user-created **Crime**s that can be saved and reloaded. For now, you populate the **List** with 100 boring **Crime** objects.

The **CrimeListViewModel** is not a solution for long-term storage of data, but it does encapsulate all the data necessary to populate **CrimeListFragment**'s view. In Chapter 12, you will learn more about long-term data storage when you update CriminalIntent to store the crime list in a database.

Next, create the **CrimeListFragment** class and associate it with **CrimeListViewModel**. Make it a subclass of **androidx.fragment.app.Fragment**.

Listing 10.3 Implementing **CrimeListFragment** (CrimeListFragment.kt)

```
private const val TAG = "CrimeListFragment"

class CrimeListFragment : Fragment() {

    private val crimeListViewModel: CrimeListViewModel by viewModels()

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        Log.d(TAG, "Total crimes: ${crimeListViewModel.crimes.size}")
    }
}
```

For now, **CrimeListFragment** is an empty shell of a fragment. It does not even have a UI to display; it just logs the number of crimes found in **CrimeListViewModel**. You will flesh the fragment out later in this chapter.

## ViewModel lifecycle with fragments

In Chapter 4, you learned about the **ViewModel** lifecycle when used with an activity. This lifecycle is slightly different when the **ViewModel** is used with a fragment. It still only has two states, created or destroyed/nonexistent, but it is now tied to the lifecycle of the fragment instead of the activity.

The **ViewModel** will remain active as long as the fragment's view is onscreen. This means the **ViewModel** will persist across rotation (even though the fragment instance will not) and be accessible to the new fragment instance.

The **ViewModel** will be destroyed when the fragment is destroyed. This can happen when the hosting activity replaces the fragment with a different one. Even though the same activity is on the screen, both the fragment and its associated **ViewModel** will be destroyed, since they are no longer needed.

One special case is when you add the fragment transaction to the back stack. When the activity replaces the current fragment with a different one, if the transaction is added to the back stack, the fragment instance and its **ViewModel** will not be destroyed. This maintains your state: If the user presses the Back button, the fragment transaction is reversed. The original fragment instance is put back on the screen, and all the data in the **ViewModel** is preserved.

Next, update `activity_main.xml` to host an instance of **`CrimeListFragment`** instead of **`CrimeDetailFragment`**.

### Listing 10.4 Adding **`CrimeListFragment`** (`activity_main.xml`)

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.fragment.app.FragmentContainerView
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/fragment_container"
    android:name="com.bignerdranch.android.criminalintent.CrimeDetailFragment"
    android:name="com.bignerdranch.android.criminalintent.CrimeListFragment"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity" />
```

For now, you have hardcoded **`MainActivity`** to always display a **`CrimeListFragment`**. In Chapter 13, you will update **`MainActivity`** to use the Fragment Navigation library to navigate between **`CrimeListFragment`** and **`CrimeDetailFragment`** as the user moves through the app.

Run CriminalIntent, and you will see **`MainActivity`**'s **`FragmentContainerView`** hosting an empty **`CrimeListFragment`**, as shown in Figure 10.2.

### Figure 10.2 Blank **`MainActivity`** screen



Search the Logcat output for CrimeListFragment. You will see a log statement showing the total number of crimes:

```
2022-02-25 15:19:39.950 26140-26140/com.bignerdranch.android.criminalintent
    D/CrimeListFragment: Total crimes: 100
```

## Adding a RecyclerView

You want **`CrimeListFragment`** to display a list of crimes to the user. To do this, you will use a **`RecyclerView`**.

The **`RecyclerView`** class lives in another Jetpack library. So the first step to using a **`RecyclerView`** is to add the RecyclerView library as a dependency.

Listing 10.5 Adding RecyclerView dependency (`app/build.gradle`)

```
dependencies {
    ...
    implementation 'androidx.lifecycle:lifecycle-viewmodel-ktx:2.4.1'
    implementation 'androidx.recyclerview:recyclerview:1.2.1'
    ...
}
```

Again, sync your Gradle files before moving on.

Your **`RecyclerView`** will live in **`CrimeListFragment`**'s layout file. To create the layout file, right-click the res/layout directory in the project tool window and choose New → Layout resource file. Name the new file fragment_crime_list. For the Root element, specify androidx.recyclerview.widget.RecyclerView (Figure 10.3).

Figure 10.3 Adding **`CrimeListFragment`**'s layout file



In the new `layout/fragment_crime_list.xml` file, add an ID attribute to the **`RecyclerView`**. Collapse the close tag into the opening tag, since you will not add any children to the **`RecyclerView`**.

### Listing 10.6 Adding **`RecyclerView`** to a layout file (`layout/fragment_crime_list.xml`)

```
<androidx.recyclerview.widget.RecyclerView
        xmlns:android="http://schemas.android.com/apk/res/android"
        android:id="@+id/crime_recycler_view"
        android:layout_width="match_parent"
        ~~android:layout_height="match_parent">~~
        android:layout_height="match_parent"/>

~~</androidx.recyclerview.widget.RecyclerView>~~
```

Now that **`CrimeListFragment`**'s view is set up, hook up the view to the fragment in `CrimeListFragment.kt`. Inflate and bind your layout – and do not forget to null out your `binding` in **`onDestroyView()`**.

### Listing 10.7 Hooking up the view for **`CrimeListFragment`** (`CrimeListFragment.kt`)

```
class CrimeListFragment : Fragment() {

    private var _binding: FragmentCrimeListBinding? = null
    private val binding
        get() = checkNotNull(_binding) {
            "Cannot access binding because it is null. Is the view visible?"
        }

    private val crimeListViewModel: CrimeListViewModel by viewModels()

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        Log.d(TAG, "Total crimes: ${crimeListViewModel.crimes.size}")
    }

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        _binding = FragmentCrimeListBinding.inflate(inflater, container, false)
        return binding.root
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }
    ...
}
```

## Implementing a LayoutManager

**RecyclerView** is a view with a narrow focus, and on its own it does not do much. All it does is "recycle," or reuse, views to display a list of data. It delegates all the other responsibilities of displaying that list of data to other components: **LayoutManager**, **ViewHolder**, and **Adapter**. We will walk you through these pieces one at a time.

The **RecyclerView** delegates the responsibility for positioning items on the screen to the **LayoutManager**. The **LayoutManager** positions each item and also defines how scrolling works. So if **RecyclerView** wants to display items but the **LayoutManager** is not there, it will give up and display nothing.

There are a few built-in **LayoutManager**s to choose from, and you can find more as third-party libraries. Set a **LinearLayoutManager** as the **LayoutManager** for your **RecyclerView**. It will position the items in the list vertically, one after the other, like a **LinearLayout**.

Listing 10.8 Setting up the **LayoutManager** (CrimeListFragment.kt)

```
class CrimeListFragment : Fragment() {
    ...
    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        _binding = FragmentCrimeListBinding.inflate(inflater, container, false)

        binding.crimeRecyclerView.layoutManager = LinearLayoutManager(context)

        return binding.root
    }
    ...
}
```

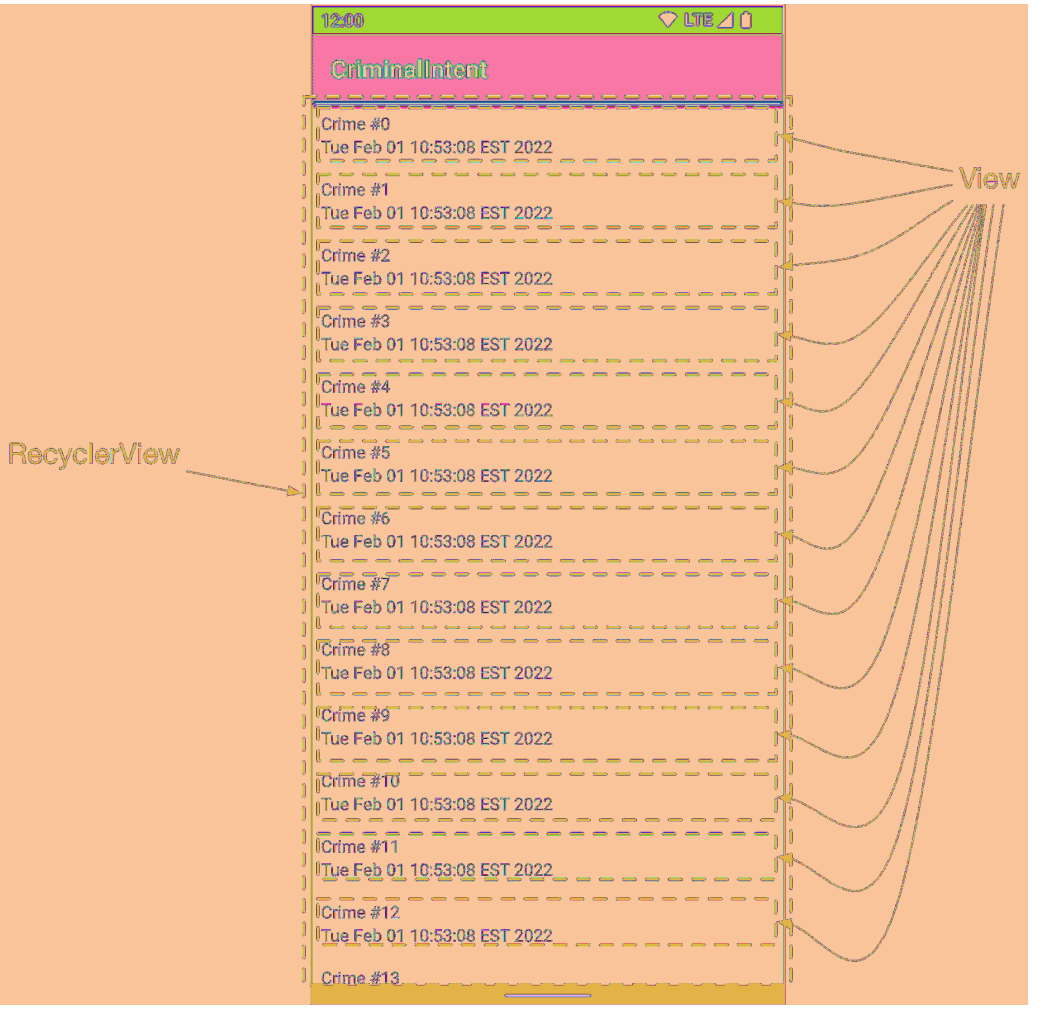
Run the app again. You will still see a blank screen, but now you are looking at an empty **RecyclerView**.

## Creating an Item View Layout

**`RecyclerView`** is a subclass of **`ViewGroup`**. It displays a list of child **`View`** objects, called *item views*. Each item view represents a single object from the list of data backing the recycler view (in your case, a single crime from the crime list). Depending on the complexity of what you need to display, these child **`View`**s could be complex or very simple.

For your first implementation, each item in the list will display the title and date of a **`Crime`**, as shown in Figure 10.4.

Figure 10.4 A **`RecyclerView`** with child **`View`**s




Each item displayed on the **`RecyclerView`** will have its own view hierarchy, exactly the way **`CrimeDetailFragment`** has a view hierarchy for the entire screen. Specifically, the **`View`** object on each row will be a **`LinearLayout`** containing two **`TextView`**s.

You create a new layout for a list item view the same way you do for the view of an activity or a fragment. Create a new layout resource file called list_item_crime and set the root element to LinearLayout.

Update your layout file to add padding to the **LinearLayout** and to add the two **TextView**s, as shown in Listing 10.9.

### Listing 10.9 Updating the list item layout file (layout/list_item_crime.xml)

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
              android:orientation="vertical"
              android:layout_width="match_parent"
              android:layout_height="match_parent">
              android:layout_height="wrap_content"
              android:padding="8dp">

    <TextView
            android:id="@+id/crime_title"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:text="Crime Title"/>

    <TextView
            android:id="@+id/crime_date"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:text="Crime Date"/>

</LinearLayout>
```

Take a look at the design preview, and you will see that you have laid out exactly one row of the completed product. In a moment, you will see how **RecyclerView** will create those rows for you.

## Implementing a ViewHolder

The **RecyclerView** expects an item view to be wrapped in an instance of **ViewHolder**. A **ViewHolder** stores a reference to an item's view. But, as usual, you are not going to interact directly with the **View**. You are going to use View Binding.

Create a new file named `CrimeListAdapter.kt`. In it, define a view holder by adding a **CrimeHolder** class that extends from **RecyclerView.ViewHolder**.

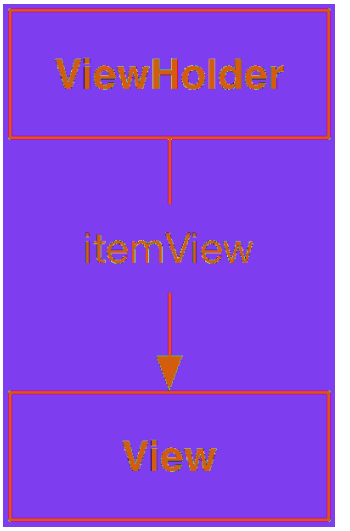
Listing 10.10 The beginnings of a **ViewHolder** (`CrimeListAdapter.kt`)

```
class CrimeHolder(
    val binding: ListItemCrimeBinding
) : RecyclerView.ViewHolder(binding.root) {

}
```

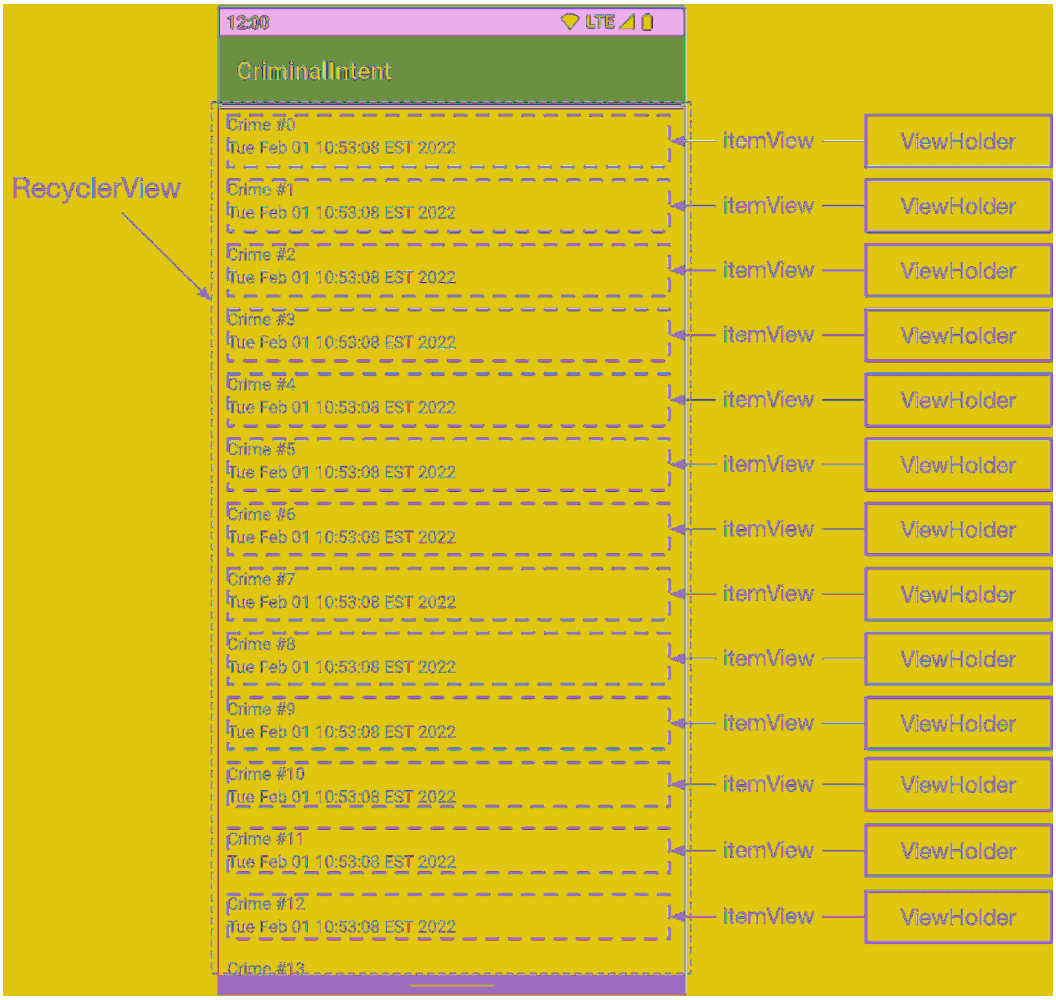
In **CrimeHolder**'s constructor, you take in the binding to hold on to. Immediately, you pass its root view as the argument to the **RecyclerView.ViewHolder** constructor. The base **ViewHolder** class will then hold on to the view in a property named `itemView` (Figure 10.5).

Figure 10.5 The **ViewHolder** and its `itemView`



A **`RecyclerView`** never creates **`View`**s by themselves. It always creates **`ViewHolder`**s, which bring their `itemViews` along for the ride (Figure 10.6).

Figure 10.6 The **`ViewHolder`** visualized




When the **`View`** for each item is simple, **`ViewHolder`** has few responsibilities. For more complicated **`View`**s, the **`ViewHolder`** makes wiring up the different parts of a `binding` to a **`Crime`** simpler and more efficient. (For example, you do not need to search through the item view hierarchy to get a handle for the title text view every time you need to set the title.)

The updated view holder now holds a reference to the `binding` so you can easily change the value displayed. Note that the **`CrimeHolder`** assumes that the `binding` you pass to its constructor has the type **`ListItemCrimeBinding`**. You may be wondering, "What creates crime holder instances, and where do I get the **`ListItemCrimeBinding`**?" You will learn the answer to these questions in just a moment.

## Implementing an Adapter to Populate the RecyclerView

Figure 10.6 is somewhat simplified. **`RecyclerView`** does not create **`ViewHolder`**s itself. Instead, it asks an *adapter*. An adapter is a controller object that sits between the **`RecyclerView`** and the data set that the **`RecyclerView`** should display.

The adapter is responsible for:

- creating the necessary **`ViewHolder`**s when asked
- binding data to **`ViewHolder`**s from the model layer when asked

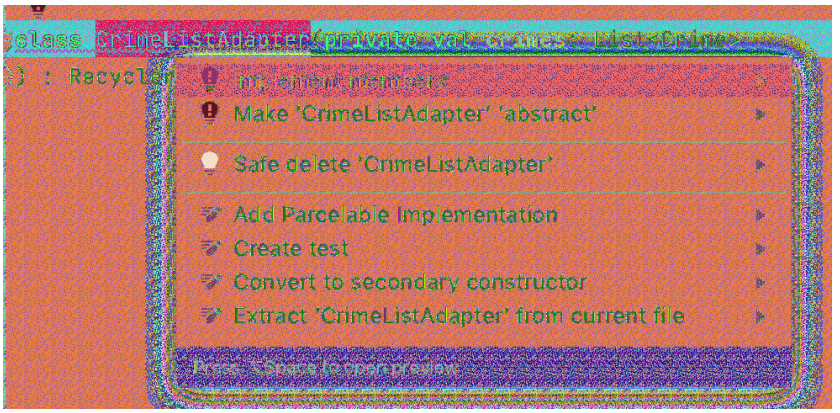
The recycler view is responsible for:

- asking the adapter to create a new **`ViewHolder`**
- asking the adapter to bind a **`ViewHolder`** to the item from the backing data at a given position

Time to create your adapter. Add a new class named **`CrimeListAdapter`** in `CrimeListAdapter.kt`. Add a primary constructor that expects a list of crimes as input and stores the passed-in crime list in a property, as shown in Listing 10.11.

In your new **`CrimeListAdapter`**, you are also going to override three functions: **`onCreateViewHolder(…)`**, **`onBindViewHolder(…)`**, and **`getItemCount()`**. To save you typing (and typos), Android Studio can generate these overrides for you. Once you have added the constructor, put your cursor on **`CrimeListAdapter`** and press Option-Return (Alt-Enter) (Figure 10.7). Select Implement members from the pop-up. In the Implement members dialog, select all three function names and click OK. Then you only need to fill in the bodies as shown.

Figure 10.7 Extending the **`RecyclerView.Adapter`** class



Listing 10.11 Creating **`CrimeListAdapter`** (`CrimeListAdapter.kt`)

```
class CrimeHolder(
    val binding: ListItemCrimeBinding
) : RecyclerView.ViewHolder(binding.root) {

}

class CrimeListAdapter(
    private val crimes: List<Crime>
) : RecyclerView.Adapter<CrimeHolder>() {

    override fun onCreateViewHolder(
        parent: ViewGroup,
        viewType: Int
    ) : CrimeHolder {
        val inflater = LayoutInflater.from(parent.context)
        val binding = ListItemCrimeBinding.inflate(inflater, parent, false)
        return CrimeHolder(binding)
    }

    override fun onBindViewHolder(holder: CrimeHolder, position: Int) {
        val crime = crimes[position]
        holder.apply {
            binding.crimeTitle.text = crime.title
            binding.crimeDate.text = crime.date.toString()
        }
    }

    override fun getItemCount() = crimes.size
}
```

**`Adapter.onCreateViewHolder(…)`** is responsible for creating a `binding` to display, wrapping the view in a view holder, and returning the result. In this case, you inflate and bind a **`ListItemCrimeBinding`** and pass the resulting `binding` to a new instance of **`CrimeHolder`**.

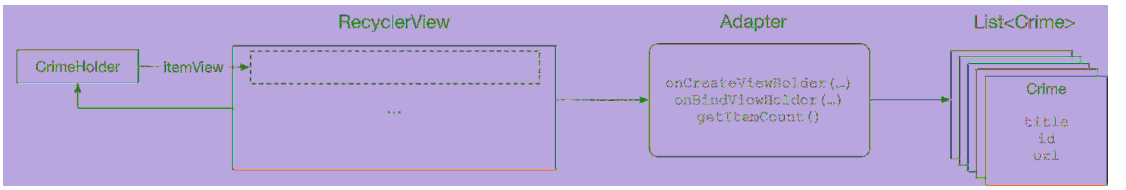
(For now, you can ignore **`onCreateViewHolder(…)`**'s parameters. You only need these values if you are doing something fancy, like displaying different types of views within the same recycler view. See the section called Challenge: RecyclerView View Types at the end of this chapter for more information.)

**`Adapter.onBindViewHolder(…)`** is responsible for populating a given `holder` with the crime from a given `position`. In this case, you get the crime from the crime list at the requested position. You then use the title and date from that crime to set the text in the corresponding text views.

When the recycler view needs to know how many items are in the data set backing it (such as when the recycler view first spins up), it will ask its adapter by calling **`Adapter.getItemCount()`**. Here, **`getItemCount()`** returns the number of items in the list of crimes to answer the recycler view's request.

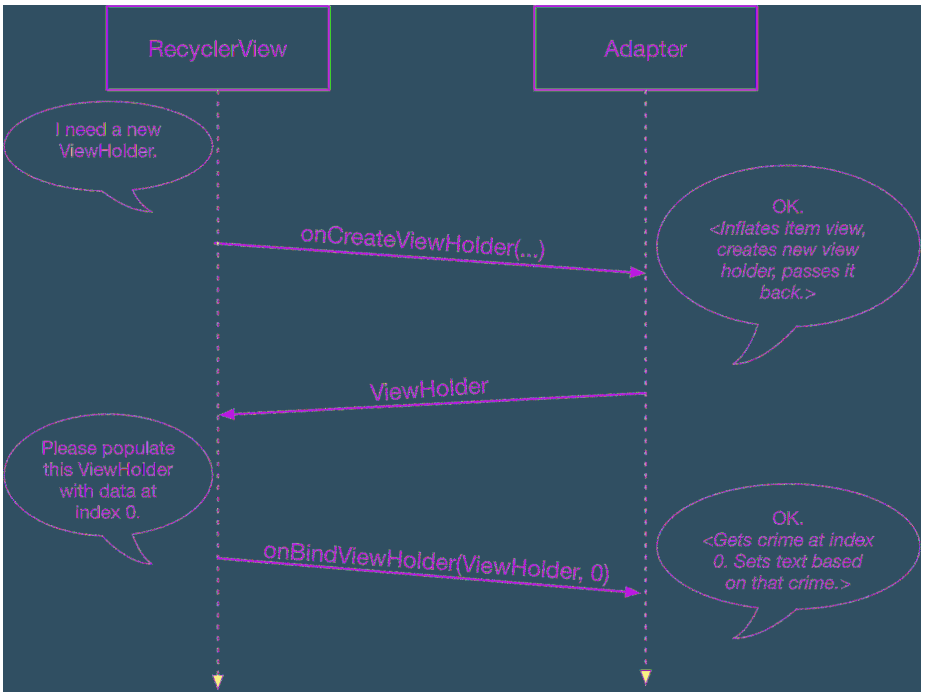
The **RecyclerView** itself does not know anything about the **Crime** object or the list of **Crime** objects to be displayed. Instead, the **CrimeListAdapter** knows all of a **Crime**'s intimate and personal details. The adapter also knows about the list of crimes that backs the recycler view (Figure 10.8).

Figure 10.8 **Adapter** sits between recycler view and data set




When the **RecyclerView** needs a view object to display, it will have a conversation with its adapter. Figure 10.9 shows an example of a conversation that a **RecyclerView** might initiate.

Figure 10.9 A scintillating **RecyclerView**–**Adapter** conversation




The **RecyclerView** calls the adapter's **onCreateViewHolder(ViewGroup, Int)** function to create a new **ViewHolder**, along with its juicy payload: a **View** to display. The **ViewHolder** that the adapter creates and hands back to the **RecyclerView** (along with its `binding`) has not yet been populated with data from a specific item in the data set.

Next, the **RecyclerView** calls **onBindViewHolder(ViewHolder, Int)**, passing a **ViewHolder** into this function along with the position. The adapter looks up the model data for that position and binds it to the **ViewHolder**'s **View**. To bind it, the adapter fills in the **View** to reflect the data in the model object.

After this process is complete, **RecyclerView** places the list item on the screen.

## Setting the RecyclerView's adapter

Now that you have an **`Adapter`**, all you need to do is instantiate an instance with your crime data and connect it to your **`RecyclerView`**.

Listing 10.12 Setting an **`Adapter`** (`CrimeListFragment.kt`)

```
class CrimeListFragment : Fragment() {
    ...
    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        _binding = FragmentCrimeListBinding.inflate(inflater, container, false)

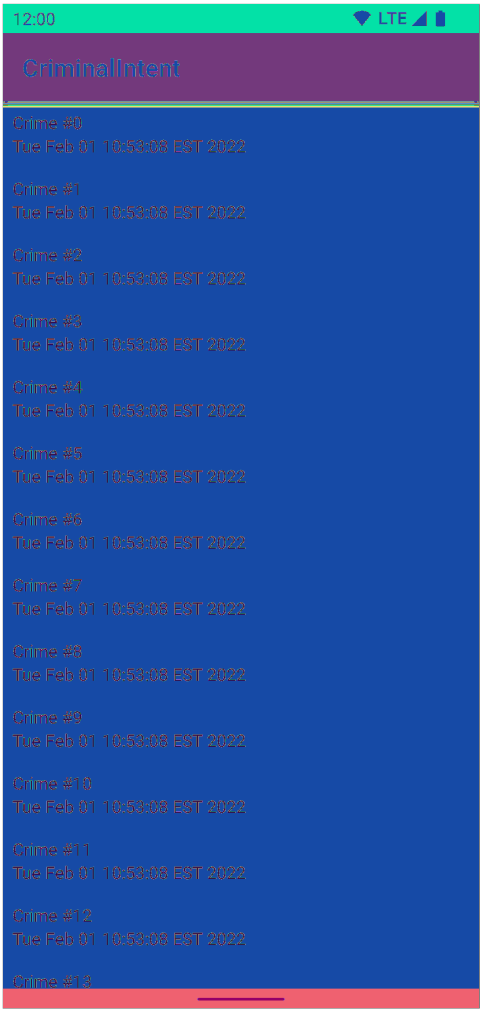
        binding.crimeRecyclerView.layoutManager = LinearLayoutManager(context)

        val crimes = crimeListViewModel.crimes
        val adapter = CrimeListAdapter(crimes)
        binding.crimeRecyclerView.adapter = adapter

        return binding.root
    }
    ...
}
```

Run CriminalIntent and scroll through your **`RecyclerView`**, which should now look like Figure 10.10.

Figure 10.10 **`RecyclerView`** populated with **`Crime`**s




Swipe or drag down and you will see even more views scroll across your screen. Every visible **`CrimeHolder`** should display a distinct **`Crime`**. (If your rows are much taller than these, or if you only see one row on the screen, double-check that the `layout_height` on your row's **`LinearLayout`** is set to `wrap_content`.)

When you fling the view up, the scrolling animation should feel as smooth as warm butter. This effect is a direct result of keeping **`onBindViewHolder(…)`** small and efficient, doing only the minimum amount of work necessary. Take heed: Always be efficient in your **`onBindViewHolder(…)`**. Otherwise, your scroll animation could feel as chunky as cold Parmesan.

## Recycling Views

Figure 10.10 shows 13 (and a half) rows of **`View`**s. You can swipe to scroll through 100 **`View`**s to see all of your **`Crime`**s. Does that mean that you have 100 **`View`** objects in memory? Thanks to your **`RecyclerView`**, no.

Creating a **`View`** for every item in the list all at once could easily become unworkable. As you might imagine, a list can have far more than 100 items, and your list items can be much more involved than your simple implementation here. Also, a **`Crime`** only needs a **`View`** when it is onscreen, so there is no need to have 100 **`View`**s ready and waiting. It would make far more sense to create view objects only as you need them.

**`RecyclerView`** does just that. Instead of creating 100 **`View`**s, it creates just enough to fill the screen. When a view is scrolled off the screen, **`RecyclerView`** reuses it rather than throwing it away. In short, it lives up to its name: It recycles views over and over.

Because of this, **`onCreateViewHolder(…)`** will happen a lot less often than **`onBindViewHolder(…)`**: Once enough **`ViewHolder`**s have been created, **`RecyclerView`** stops calling **`onCreateViewHolder(…)`**. Instead, it saves time and memory by recycling old **`ViewHolder`**s and passing those into **`onBindViewHolder(…)`**.

## Cleaning Up Binding List Items

Right now, the **Adapter** binds crime data directly to a crime holder's text views in **Adapter.onBindViewHolder(…)**. This works fine, but it is better to more cleanly separate concerns between the view holder and the adapter. The adapter should know as little as possible about the inner workings and details of the view holder.

Instead, place all the code that will do the real work of binding inside your **CrimeHolder**. Start by adding a **bind(Crime)** function to **CrimeHolder**. In this new function, cache the crime being bound into a property and set the text values on `crimeTitle` and `crimeDate`. While you are at it, make the existing `binding` property private.

### Listing 10.13 Writing a **bind(Crime)** function (`CrimeListAdapter.kt`)

```
class CrimeHolder(
    private val binding: ListItemCrimeBinding
) : RecyclerView.ViewHolder(binding.root) {
    fun bind(crime: Crime) {
        binding.crimeTitle.text = crime.title
        binding.crimeDate.text = crime.date.toString()
    }
}
...
```

When given a **Crime** to bind, **CrimeHolder** will now update the title **TextView** and date **TextView** to reflect the state of the **Crime**.

Next, call your newly minted **bind(Crime)** function each time the **RecyclerView** requests that a given **CrimeHolder** be bound to a particular crime.

### Listing 10.14 Calling **bind(Crime)** (`CrimeListAdapter.kt`)

```
...
class CrimeListAdapter(
    private val crimes: List<Crime>
) : RecyclerView.Adapter<CrimeHolder>() {

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): CrimeHolder {
        ...
    }

    override fun onBindViewHolder(holder: CrimeHolder, position: Int) {
        val crime = crimes[position]
        holder.apply {
            binding.crimeTitle.text = crime.title
            binding.crimeDate.text = crime.date.toString()
        }
        holder.bind(crime)
    }

    override fun getItemCount() = crimes.size
}
```

Run CriminalIntent one more time. The result should look the same as it did in Figure 10.10.

## Responding to Presses

As icing on the **RecyclerView** cake, CriminalIntent should also respond to a press on these list items. In Chapter 13, you will launch the detail view for a **Crime** when the user presses that **Crime** in the list. For now, show a **Toast** when the user takes action on a **Crime**.

As you may have noticed, **RecyclerView**, while powerful and capable, has precious few real responsibilities. (May it be an example to us all.) The same goes here: Handling touch events is mostly up to you. If you need them, **RecyclerView** can forward along raw touch events. But most of the time this is not necessary.

Instead, you can handle them like you normally do: by setting an **OnClickListener**. Because you want the entire row to be clickable, set the **OnClickListener** on the `root` property of the `binding`.

Listing 10.15 Detecting presses in **CrimeHolder** (`CrimeListAdapter.kt`)

```
class CrimeHolder(
    private val binding: ListItemCrimeBinding
) : RecyclerView.ViewHolder(binding.root) {
    fun bind(crime: Crime) {
        binding.crimeTitle.text = crime.title
        binding.crimeDate.text = crime.date.toString()

        binding.root.setOnClickListener {
            Toast.makeText(
                binding.root.context,
                "${crime.title} clicked!",
                Toast.LENGTH_SHORT
            ).show()
        }
    }
}
...
```

Run CriminalIntent and press an item in the list. You should see a **Toast** indicating that the item was pressed.

## Lists and Grids: Past, Present, and Future

The core Android OS includes **ListView**, **GridView**, and **Adapter** classes. Until the release of Android 5.0, these were the preferred ways to create lists or grids of items.

The API for these components is very similar to **RecyclerView**'s. The **ListView** or **GridView** class is responsible for scrolling a collection of items, but it does not know much about each of those items. The **Adapter** is responsible for creating each of the **View**s in the list. However, **ListView** and **GridView** do not enforce that you use the **ViewHolder** pattern (though you can – and should – use it).

These old implementations are replaced by the **RecyclerView** implementation because of the complexity required to alter the behavior of a **ListView** or **GridView**.

Creating a horizontally scrolling **ListView**, for example, is not included in the **ListView** API and requires a lot of work. Creating custom layout and scrolling behavior with a **RecyclerView** is still a lot of work, but **RecyclerView** was built to be extended, so it is not quite so bad. And, as you will see in Chapter 20, **RecyclerView** can be used with a **GridLayoutManager** to arrange items in a grid.

Another key feature of **RecyclerView** is the animation of items in the list. Animating the addition or removal of items in a **ListView** or **GridView** is a complex and error-prone task. **RecyclerView** makes this much easier; it includes a few built-in animations and allows for easy customization of these animations.

For example, if you found out that the crime at position 0 moved to position 5, you could animate that change like so:

```
recyclerView.adapter.notifyItemMoved(0, 5)
```

**RecyclerView** is powerful and extensible, but it is also complex and requires a lot of setup for even simple UIs. With Jetpack Compose, which you will start learning about in Chapter 26, you have access to the **LazyColumn** and **LazyRow** composables. These composables have all the customizability and performance of **RecyclerView**, but they can be created with a fraction of the code.

## For the More Curious: A Smarter Adapter with ListAdapter

As the backing data changes, **`RecyclerView`** provides all the tools needed to perform animations to reflect those changes. As in the example above, you could call APIs like **`RecyclerView.Adapter.notifyItemMoved(…)`** or **`RecyclerView.Adapter.notifyItemInserted(…)`** to tell the **`RecyclerView`** to animate those changes in. However, you do not usually have visibility on specific changes to your data, so you cannot easily call those functions on the individual changes in the list.

Instead, it is much more common to be presented with a copy of the list of data, with the changes embedded in it. Unless you manually calculate all the changes between the old list of data and the new list of data, the best you can do is reassign the backing list of data of your **`RecyclerView.Adapter`** and force it to re-render every element in the list. That calculation of changes is difficult, so developers often rely on **`RecyclerView.Adapter.notifyDataSetChanged(…)`**, which will redraw and rebind all the items in the **`RecyclerView`**'s layout.

A key aspect of **`RecyclerView`**'s design is that it tries to be efficient, avoiding unnecessary work. The benefit of APIs such as **`RecyclerView.Adapter.notifyItemMoved(…)`** and **`RecyclerView.Adapter.notifyItemInserted(…)`** is that they will only update or alter the relevant views to perform those animations. All the other views in the list will remain untouched. In comparison, **`RecyclerView.Adapter.notifyDataSetChanged(…)`** is a blunt instrument that often does a lot of unnecessary work.

When you are presented with an entirely new list of (very similar) data, it would be convenient to have a tool that calculates the differences between the new and old lists and then calls the appropriate **`RecyclerView.Adapter.notifyItem…()`** functions to animate those changes in. That is where **`ListAdapter`** comes in.

**`ListAdapter`** extends the regular **`RecyclerView.Adapter`**, bringing with it extra goodies to help you efficiently display and update lists of data. By using **`ListAdapter`** instead of **`RecyclerView.Adapter`**, you can have the library calculate the individual insert/move/remove/update operations to efficiently update the views in your **`RecyclerView`**.

This calculation does not happen magically. **`ListAdapter`** uses a class called **`DiffUtil`**, which is included in the RecyclerView library. **`DiffUtil`** can detect which items have changed between your original list and your updated list, but it needs a bit of help. The key component that makes this process work is an instance of a class you define that extends **`DiffUtil.ItemCallback`**.

The **`DiffUtil.ItemCallback`** class has two functions that you must implement (**`areContentsTheSame(…)`** and **`areItemsTheSame(…)`**), which **`ListAdapter`** uses internally to determine the differences between the lists and then call the appropriate APIs.

With that set up, whenever you have a new list of data to display in your **`RecyclerView`**, all you have to do is call **`ListAdapter.submitList(…)`**, passing in the new list of data, and your **`RecyclerView`** will beautifully and efficiently animate the new data onscreen.

## Challenge: RecyclerView View Types

For this advanced challenge, create two types of rows in your **`RecyclerView`**: a normal row and a row for more serious crimes. To implement this, you will need to work with the *view type* feature available in **`RecyclerView.Adapter`**.

Add a new property, `requiresPolice`, to the **`Crime`** object and use it to determine which view to load on the **`CrimeListAdapter`** by implementing the **`getItemViewType(Int)`** function (`developer.android.com/reference/androidx/recyclerview/widget/RecyclerView.Adapter#getItemViewType(int)`).

In the **`onCreateViewHolder(ViewGroup, Int)`** function, you will also need to add logic that returns a different **`ViewHolder`** based on the new `viewType` value returned by **`getItemViewType(Int)`**. Use the original layout for crimes that do not require police intervention and a new layout with a streamlined interface containing a button that says “contact police” for crimes that do.

# 11

# Creating User Interfaces with Layouts and Views

In this chapter, you will learn more about layouts and views while adding some style to your list items in the **`RecyclerView`**. You will also learn how to use the layout editor to arrange views within a **`ConstraintLayout`**.

Figure 11.1 shows what **`CrimeListFragment`**'s view will look like once you build up your masterpiece.

Figure 11.1 CriminalIntent, now with beautiful pictures




In previous chapters, you used nested layout hierarchies to arrange your views. For example, the `layout/activity_main.xml` file you created for GeoQuiz in Chapter 1 nested a **`LinearLayout`** within another **`LinearLayout`**. This nesting is hard for you and other developers to read and edit. Worse, nesting can degrade your app's performance. Nested layouts can take a long time for the Android OS to measure and lay out, meaning your users might experience a delay before they see your views onscreen.

Flat, or non-nested, layouts take less time for the OS to measure and lay out. And this is one of the areas where **`ConstraintLayout`** really shines. You can create beautifully intricate layouts without using nesting.

Before you start learning about **`ConstraintLayout`**, you need to take care of one task: You need a copy of that fancy handcuff image from Figure 11.1 in your project. Open the solutions file (`www.bignerdranch.com/android-5e-solutions`) and find the solution for this chapter in `11. Creating User Interfaces with Layouts and Views/Solution/CriminalIntent`. From the `app/src/main/res/drawable` directory, copy the `ic_solved.xml` file into the `drawable/` folder in your project.

## Introducing ConstraintLayout

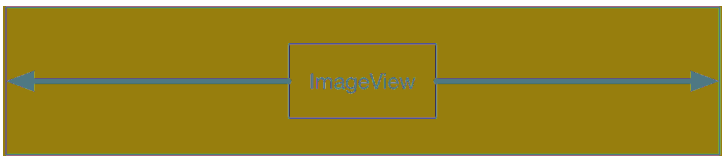
With **`ConstraintLayout`**, instead of using nested layouts you add a series of *constraints* to your layout. A constraint is like a rubber band: It pulls two things toward each other. So, for example, if you have an **`ImageView`**, you can attach a constraint from its right edge to the right edge of its parent (the **`ConstraintLayout`** itself), as shown in Figure 11.2. The constraint will hold the **`ImageView`** to the right.

Figure 11.2 **`ImageView`** with a constraint on the right edge




You can create a constraint from all four edges of your **`ImageView`** (left, top, right, and bottom). If you have opposing constraints, they will equal out, and your **`ImageView`** will be in the center of the two constraints (Figure 11.3).

Figure 11.3 **`ImageView`** with opposing constraints




So that is the big picture: To place views where you want them to go in a **`ConstraintLayout`**, you give them constraints.

What about sizing views? For that, you have three options: Let the view decide (your old friend `wrap_content`), decide for yourself, or let your view expand to fit your constraints.

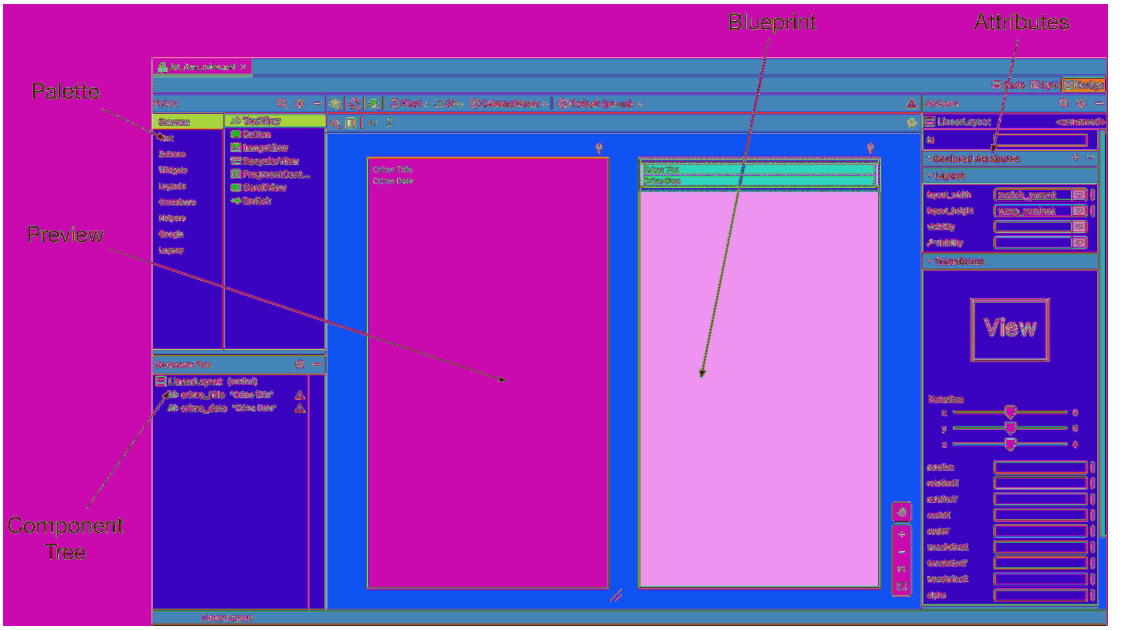
With those tools, you can achieve a great many layouts with a single **`ConstraintLayout`**, no nesting required. As you go through this chapter, you will see how to use constraints with your `list_item_crime.xml` layout file.

## Introducing the Layout Editor

So far, you have created layouts by typing XML. In this section, you will use Android Studio's layout editor.

Open `layout/list_item_crime.xml` and select the Design tab near the top-right corner of the editor tool window to open the design view (Figure 11.4).

Figure 11.4 Views in the layout editor



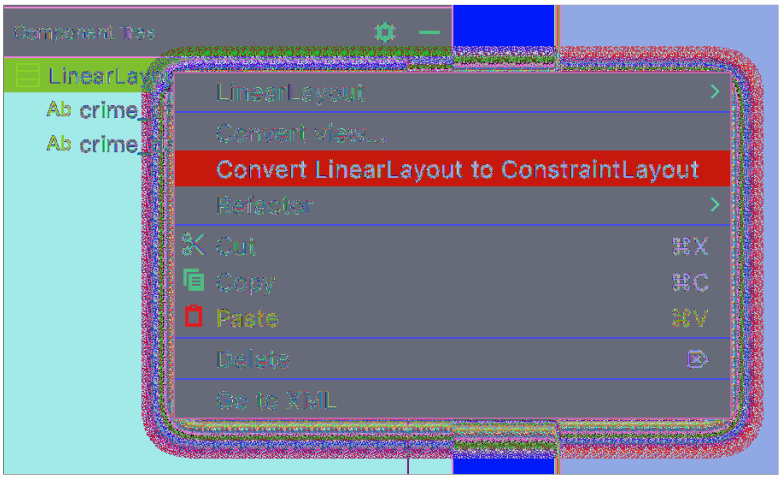

In the middle of the layout editor is the preview you have seen before. To the right of the preview is the blueprint, which, as you saw in Chapter 1, is like the preview but shows the outline of each of your views. This can be useful when you need to see how big each view is, not just what it is displaying.

In the top-left area of the screen is the palette, which contains all the views you could wish for, organized by category. The component tree is in the bottom left. The tree shows how the views are organized in the layout. If you do not see the palette or component tree, click the tabs on the left side of the preview to open the tool windows.

On the right side of the screen is the attributes tool window. Here, you can view and edit the attributes of the view selected in the component tree.

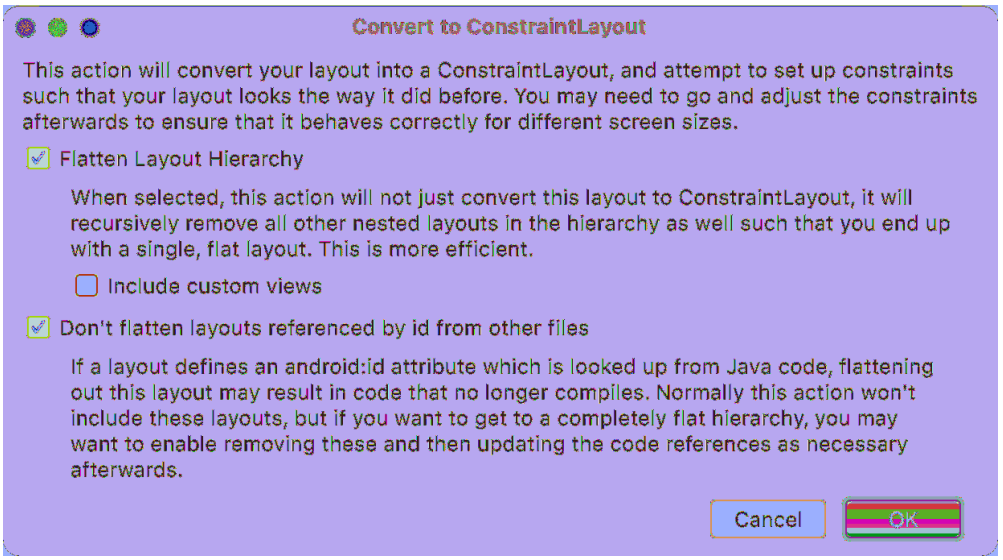
The first thing you need to do is convert `list_item_crime.xml` to use a **`ConstraintLayout`**. Right-click the root **`LinearLayout`** in the component tree and select Convert LinearLayout to ConstraintLayout (Figure 11.5).

Figure 11.5 Converting the root view to a **`ConstraintLayout`**



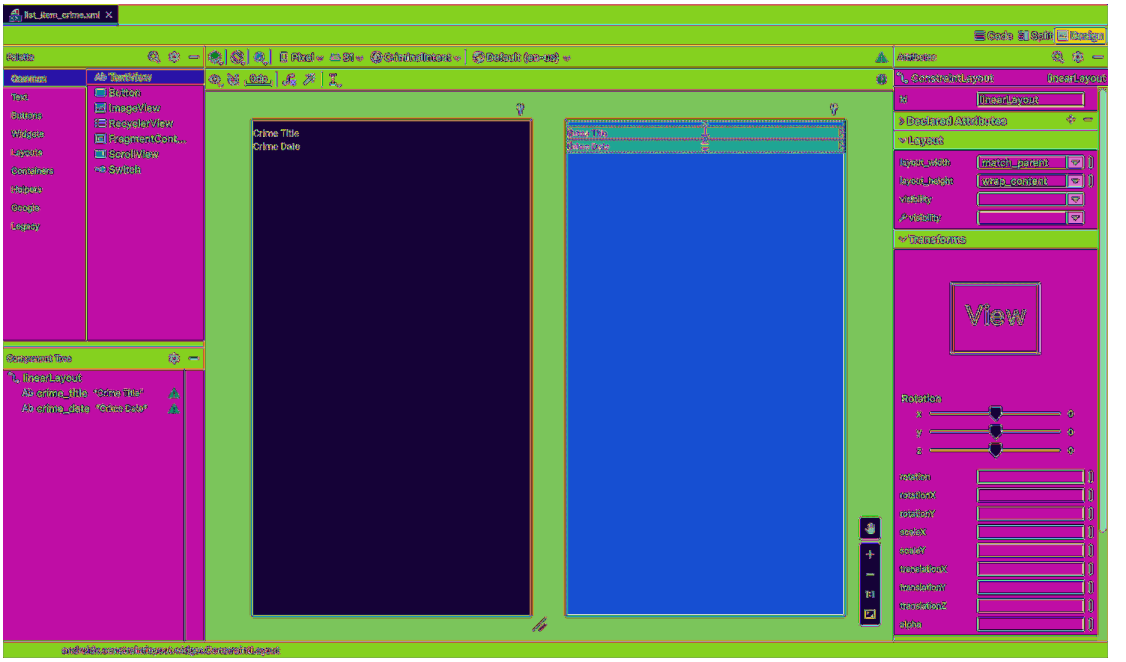

Android Studio will ask you in a pop-up how aggressive you would like this conversion process to be (Figure 11.6). Since `list_item_crime.xml` is a simple layout file, there is not much that Android Studio can optimize. Leave the default values checked and select OK.

Figure 11.6 Converting with the default configuration



Be patient. The conversion process might take a little while. But when it is complete, you will have a fine, new **`ConstraintLayout`** to work with (Figure 11.7).

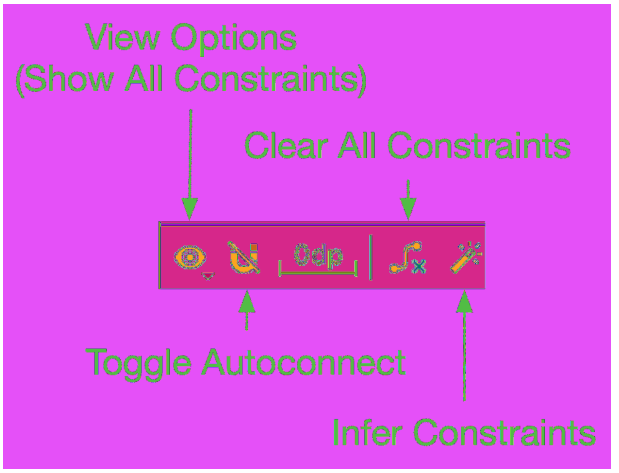
Figure 11.7 Life as a **`ConstraintLayout`**




(Wondering why the component tree says you have a linearLayout? We will get to that in a moment.)

The toolbar above the preview has a few editing controls (Figure 11.8). You may need to click into the preview to see all the controls.

Figure 11.8 Constraint controls




| | |
|---|---|
| View Options | View Options → Show All Constraints reveals the constraints that are set up in the preview and blueprint views. You will find this option helpful at times and unhelpful at others. If you have many constraints, this setting will trigger an overwhelming amount of information.<br><br>The view options menu includes other useful options, such as Show System UI. Selecting Show System UI displays the app bar as well as some system UI (such as the status bar) the user sees at runtime. You will learn more about the app bar in Chapter 15. |
| Toggle Autoconnect | When autoconnect is enabled, constraints will be automatically configured as you drag views into the preview. Android Studio will guess the constraints that you want a view to have and make those connections automatically. |
| Clear All Constraints | This button removes all existing constraints in the layout file. You will use this shortly. |
| Infer Constraints | This option is similar to autoconnect in that Android Studio will automatically create constraints for you, but it is only triggered when you select this button. Autoconnect is active any time you add a view to your layout file. |

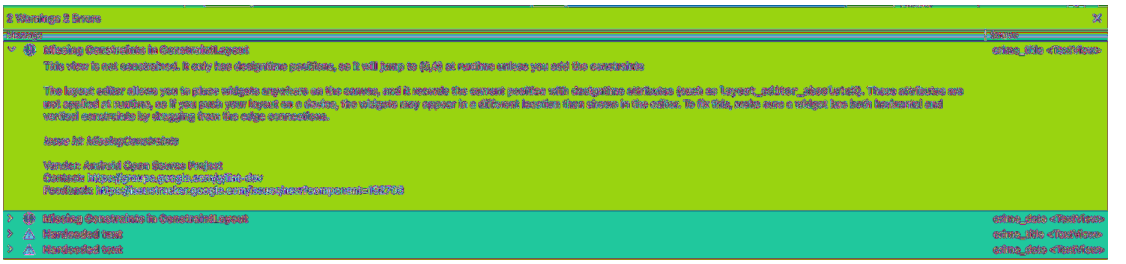
## Using ConstraintLayout

When you converted `list_item_crime.xml` to use **`ConstraintLayout`**, Android Studio automatically added the constraints it thinks will replicate the behavior of your old layout. However, to learn how constraints work you are going to start from scratch.

Select the top-level view in the component tree, labeled linearLayout. Why does it say linearLayout, when you converted your view to a **`ConstraintLayout`**? That is the ID the **`ConstraintLayout`** converter supplied. linearLayout is, in fact, your **`ConstraintLayout`**. You can check the XML version of your layout if you want to confirm this.

With linearLayout selected in the component tree, click the Clear All Constraints button (shown in Figure 11.8) and confirm your intentions in the pop-up. You will immediately see red warning flags, including one at the top right of the screen. Click it to see what that is all about (Figure 11.9).

Figure 11.9 **`ConstraintLayout`** warnings



When views do not have enough constraints, **`ConstraintLayout`** cannot know exactly where to put them. Your **`TextView`**s have no constraints at all, so they each have a warning that says they will not appear in the right place at runtime.

In the preview, the views look the same as they did when you were using a **`LinearLayout`**. But – as the errors indicate – the positioning you see in the preview is not what you would see if you ran the app. The preview allows you to position your views anywhere on the canvas to make it easier to add your constraints, but these positions are only valid in the preview, not at runtime.

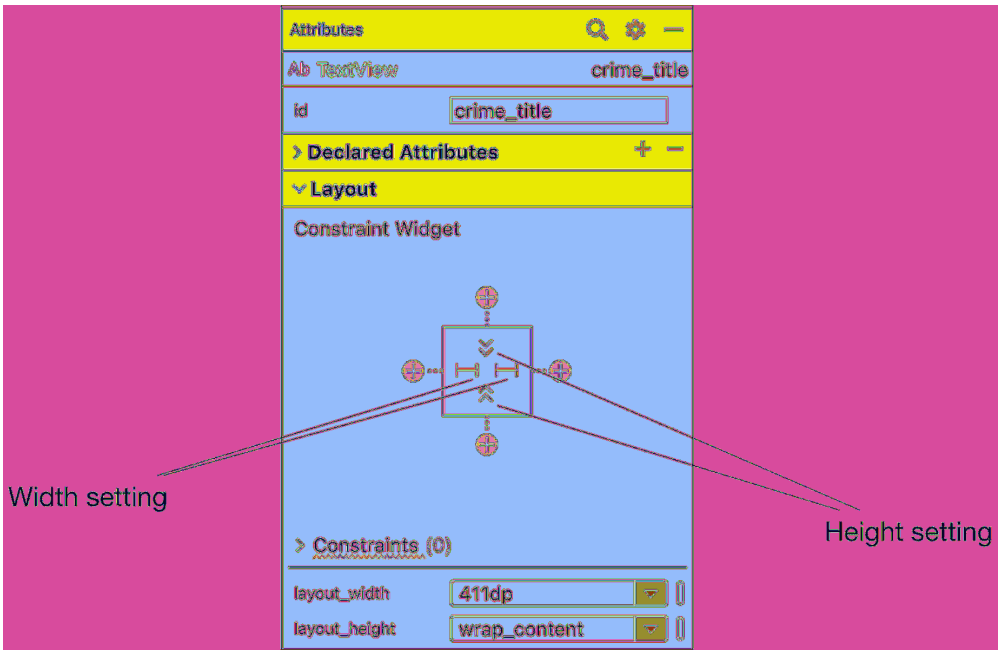
As you go through the chapter, you will add constraints to fix those warnings. In your own work, keep an eye on that warning indicator to avoid unexpected behavior at runtime.

## Making room

You need to make some room to work in the layout editor. Your two **`TextView`**s are taking up the entire area, which will make it hard to wire up anything else. Time to shrink those two views.

Select crime_title in the component tree and look at the attributes window on the right (Figure 11.10). If this window is not open for you, click the Attributes tab on the right to open it.

Figure 11.10 Title **`TextView`**'s attributes




The vertical and horizontal sizes of your **`TextView`** are governed by the height setting and width setting, respectively. There are three setting types for height and width (shown in Figure 11.11 and summarized in Table 11.1), each of which corresponds to a value for `layout_height` or `layout_width`.

Figure 11.11 Three view size settings

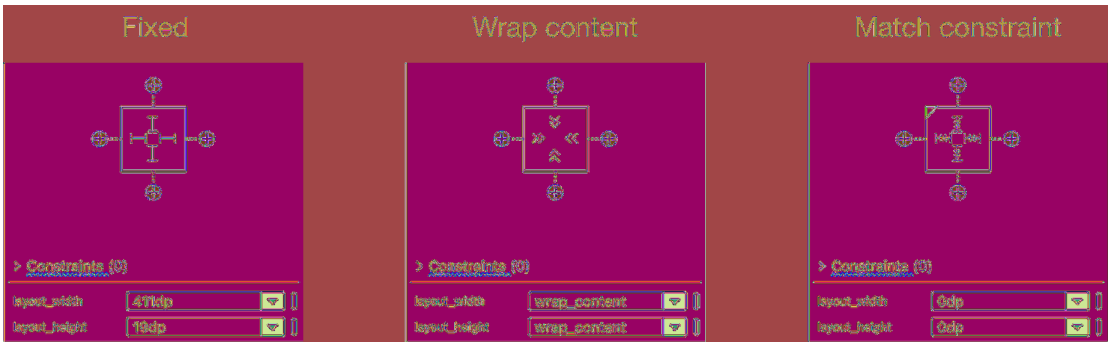
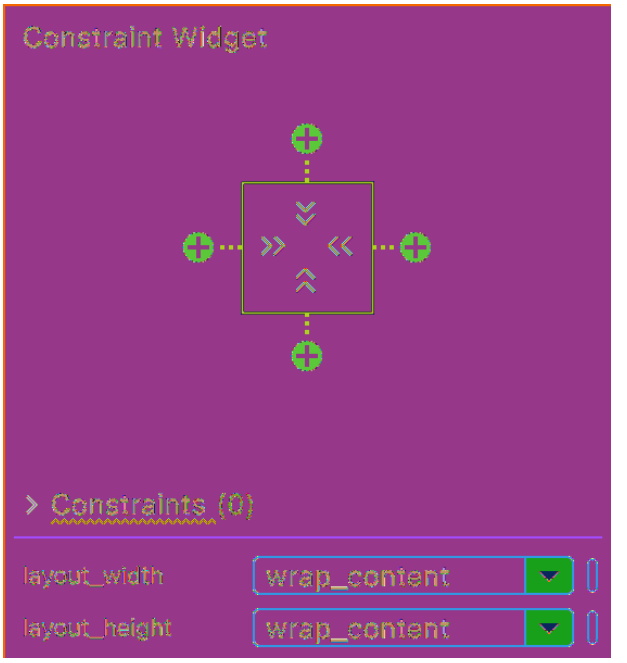


Table 11.1 View size setting types

| Setting type | Setting value | Usage |
|---|---|---|
| fixed | `Xdp` | Specifies an explicit size (that will not change) for the view. The size is specified in `dp` units and should be a positive number. (If you need a refresher on `dp` units, see the section called Screen Pixel Densities in Chapter 2.) |
| wrap content | `wrap_content` | Assigns the view its "desired" size. For a **`TextView`**, this means that the size will be just big enough to show its contents. |
| match constraint | `0dp` | Allows the view to stretch to meet the specified constraints. |

Both the title and date **`TextView`**s are set to a large fixed width, which is why they are taking up the whole screen. Adjust the width and height of both views. With crime_title still selected in the component tree, click the width setting until it cycles around to the wrap content setting (or use the drop-down menu to choose the setting). If necessary, adjust the height setting until the height is also set to wrap content (Figure 11.12).

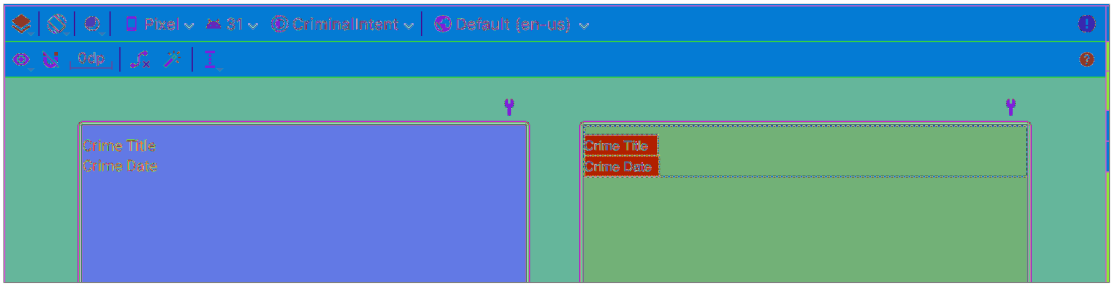
Figure 11.12 Adjusting the title width and height




Repeat the process with the crime_date view to set its width and height.

Now the two views are the correct size (Figure 11.13).

Figure 11.13 Correctly sized **`TextView`**s

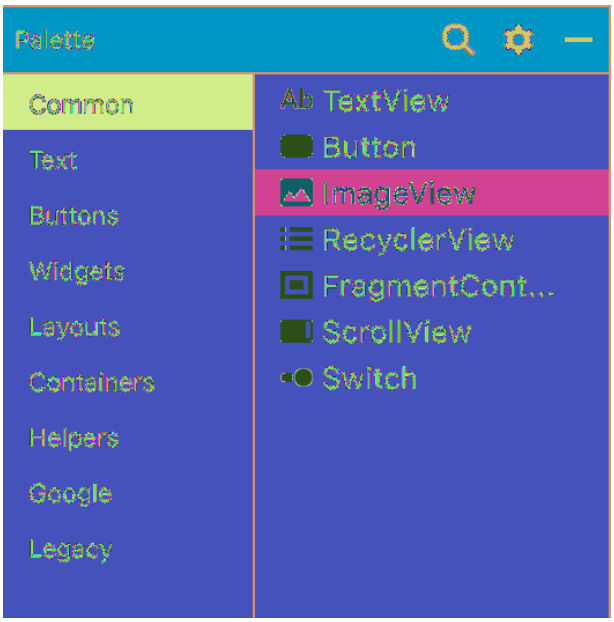



This change has not fixed the errors in your layout, because your views still have no constraints. You will add constraints to correctly position your **`TextView`**s and get rid of the errors later. First, you will add the third view you need in your layout.

## Adding views

With your other views out of the way, you can add the handcuffs image to your layout. Add an **`ImageView`** to your layout file. In the palette, find ImageView in the Common category (Figure 11.14). Drag it into your component tree as a child of the ConstraintLayout, just underneath crime_date.

Figure 11.14 Finding the **`ImageView`**



In the dialog that pops up, scroll to the CriminalIntent.app.main section and choose ic_solved as the resource for the **`ImageView`** (Figure 11.15). This image will be used to indicate which crimes have been solved. Click OK.

Figure 11.15 Choosing the **`ImageView`**'s resource

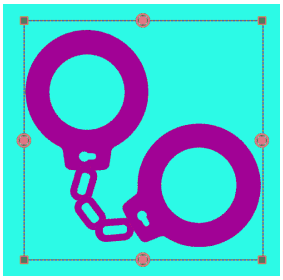



(The "ic" is short for "icon," by the way. Just as fragment layout files begin with the "fragment_" prefix and activity layout files begin with the "activity_" prefix, it is a convention to prefix your icons with the "ic_" prefix. That allows you to easily organize your various drawables.)

The **`ImageView`** is now a part of your layout, but it has no constraints. So while the layout editor gives it a position, that position does not really mean anything.

Click your **ImageView** in the preview. (You may want to zoom the preview in to get a better look. The zoom controls are in the toolbar in the lower right of the canvas.) You will see circles on each side of the **ImageView** (Figure 11.16). Each of these circles represents a *constraint handle*.

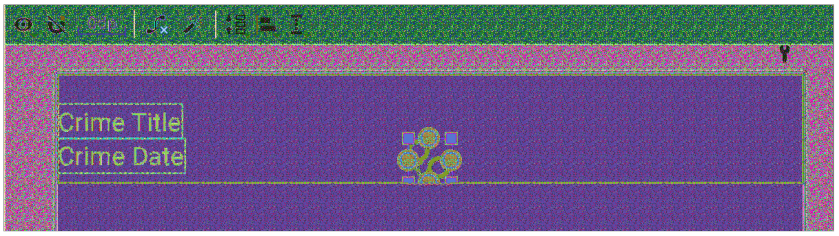
Figure 11.16 **ImageView**'s constraint handles



You want the **ImageView** to be anchored on the right side of the view and centered vertically. To accomplish this, you need to create constraints from the top, right, and bottom edges of the **ImageView**.

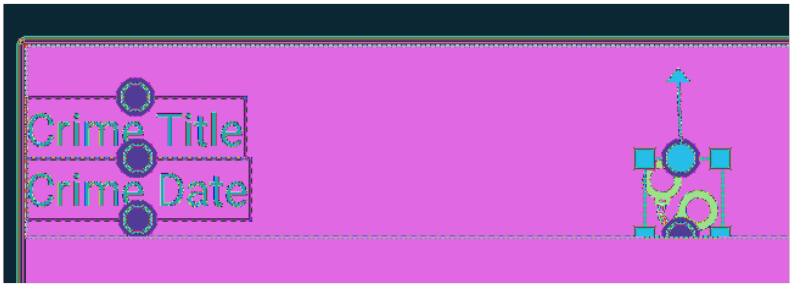
Before adding constraints, drag the **ImageView** to the right and down to move it away from the **TextView**s (Figure 11.17). Do not worry about where you place the **ImageView**. This placement will be ignored once you get your constraints in place.

Figure 11.17 Moving a view temporarily



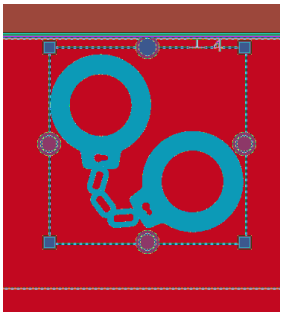
Time to add some constraints. First, you are going to set a constraint between the top of the **`ImageView`** and the top of the **`ConstraintLayout`**. In the preview, drag the top constraint handle from the **`ImageView`** to the top of the **`ConstraintLayout`**. The handle will display an arrow and turn solid blue (Figure 11.18).

Figure 11.18 Part of the way through creating a top constraint




Watch for the constraint handle to turn blue, then release the mouse to create the constraint (Figure 11.19).

Figure 11.19 Creating a top constraint

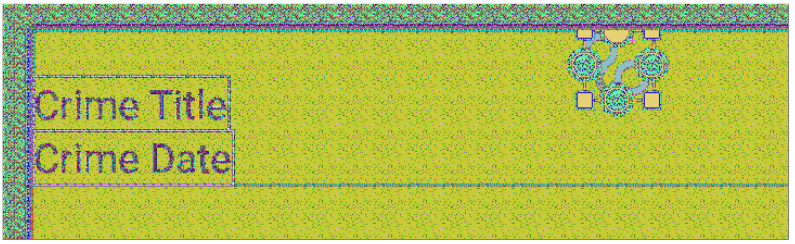


Be careful to avoid dragging one of the square handles in the corner of the image view – this will resize it instead. Also, make sure you do not inadvertently attach the constraint to one of your **`TextView`**s. If you do, click the constraint handle to delete the bad constraint, then try again.

When you let go and set the constraint, the view will snap into position to account for the presence of the new constraint. This is how you move views around in a **`ConstraintLayout`** – by setting and removing constraints.

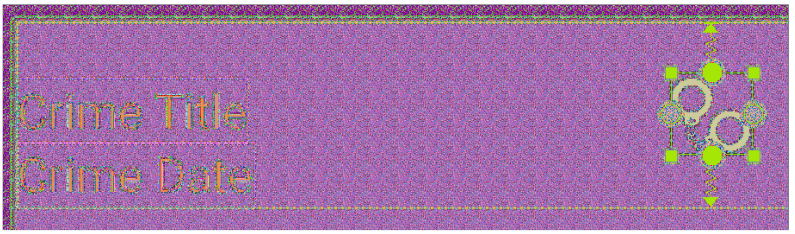
Verify that your **`ImageView`** has a top constraint connected to the top of the **`ConstraintLayout`** by hovering over the top constraint handle with your mouse. You should see an animated box around the constraint layout, with the top edge blue to show where the constraint handle is connected (Figure 11.20).

Figure 11.20 **`ImageView`** with a top constraint




Do the same for the bottom constraint handle, dragging it from the **`ImageView`** to the bottom of the root view (Figure 11.21), again taking care to avoid attaching it to the **`TextView`**s.

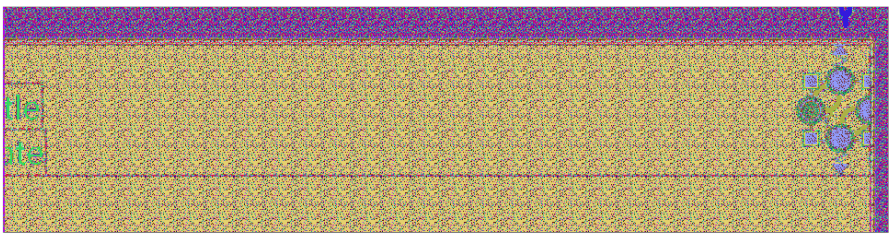
Figure 11.21 **`ImageView`** with top and bottom constraints




(The squiggly lines you see in the preview represent constraints that are stretching.)

Finally, drag the right constraint handle from the **`ImageView`** to the right side of the root view. That should set all the constraints for the image view. Your constraints should look like Figure 11.22.

Figure 11.22 **`ImageView`**'s three constraints



## ConstraintLayout's inner workings

Any edits that you make to constraints with the layout editor are reflected in the XML behind the scenes. You can still edit the raw **ConstraintLayout** XML, but the layout editor will often be easier for adding the initial constraints. **ConstraintLayout** is more verbose than other **ViewGroup**s, so adding the initial constraints manually can be a lot of work. On the other hand, working directly with the XML can be more useful when you need to make smaller changes to the layout.

Switch to the code view to see what happened to the XML when you created the three constraints on your **ImageView**:

```
<androidx.constraintlayout.widget.ConstraintLayout
        ... >
    ...
    <ImageView
        android:id="@+id/imageView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:srcCompat="@drawable/ic_solved" />

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

(You will still see errors related to the two **TextView**s. Do not worry – you will fix them shortly.)

All the views are direct children of the single **ConstraintLayout** – there are no nested layouts. If you had created the same layout using **LinearLayout**, you would have had to nest one inside another. As we said earlier, reducing nesting also reduces the time needed to render the layout, and that results in a quicker, more seamless user experience.

Take a closer look at the top constraint on the **ImageView**:

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

This attribute begins with `layout_`. All attributes that begin with `layout_` are known as *layout parameters*. Unlike other attributes, layout parameters are directions to that view's *parent*, not the view itself. They tell the parent layout how to arrange the child element within itself. You have seen a few layout parameters so far, like `layout_width` and `layout_height`.

The name of the constraint is `constraintTop`. This means that this is the top constraint on your **ImageView**.

Finally, the attribute ends with `toTopOf="parent"`. This means that this constraint is connected to the top edge of the parent. The parent here is the **ConstraintLayout**.

Whew, what a mouthful. Time to leave the raw XML behind and return to the layout editor.

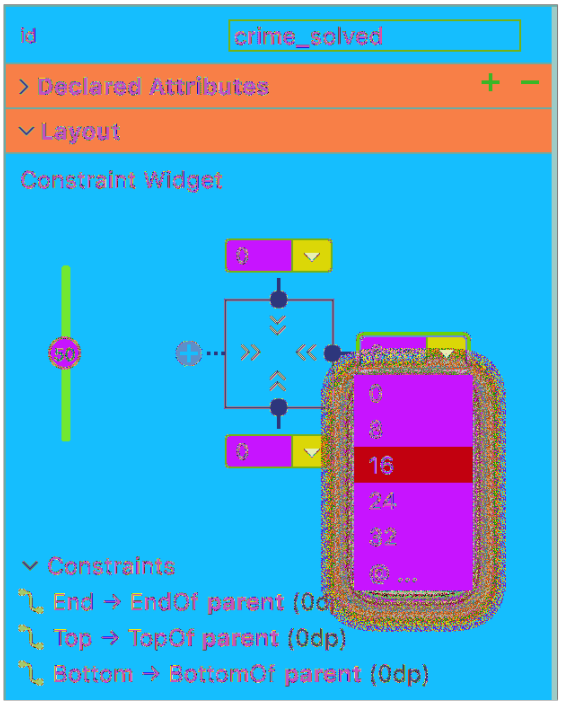
(The layout editor's tools are useful, especially with **ConstraintLayout**. But not everyone is a fan. You do not have to choose sides – you can switch back and forth between the layout editor and directly editing XML. After this chapter, use whatever approach you prefer to create the layouts in this book – XML, layout editor, or some of each.)

## Editing properties

Your **`ImageView`** is almost positioned correctly. Since the parent view is larger than the image and the image is centered vertically, it looks good on the vertical axis. However, the image is flush against the right side of the parent view. This looks a little off.

With the image still selected in the preview, check out the attributes window to the right. Because you added constraints to the top, bottom, and right of the **`ImageView`**, drop-down menus appear to allow you to select the margin for each constraint (Figure 11.23). You do not need to add margins to the top or bottom, but select `16dp` for the right margin.

Figure 11.23 Adding a margin to the end of the **`ImageView`**

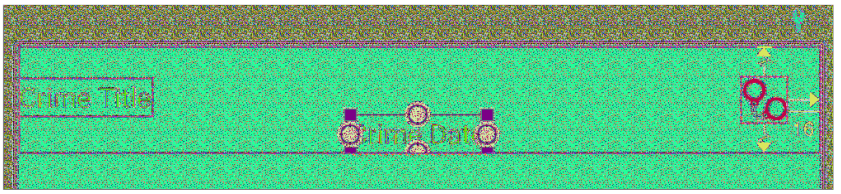



Notice that Android Studio offers you margin values in increments of 8dp. These values follow Android's material design guidelines. You can find all the Android design guidelines at `developer.android.com/design/index.html`. Your Android apps should follow these guidelines as closely as possible.

With that taken care of, let's move on to the text. Start with the position and size of the title **`TextView`**.

First, select Crime Date in the preview and drag it out of the way (Figure 11.24). Remember that any changes you make to the position in the preview will not be represented when the app is running. At runtime, only constraints remain.

Figure 11.24 Get out of here, date



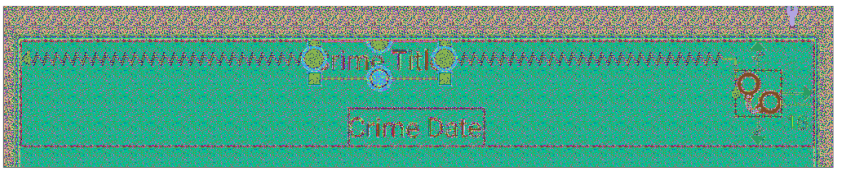

Now, select crime_title in the component tree. This will also highlight Crime Title in the preview.

You want Crime Title to be at the top left of your layout, positioned to the left of your **`ImageView`**. That requires three constraints:

- from the left side of your view to the left side of the parent
- from the top of your view to the top of the parent
- from the right of your view to the left side of the **`ImageView`**

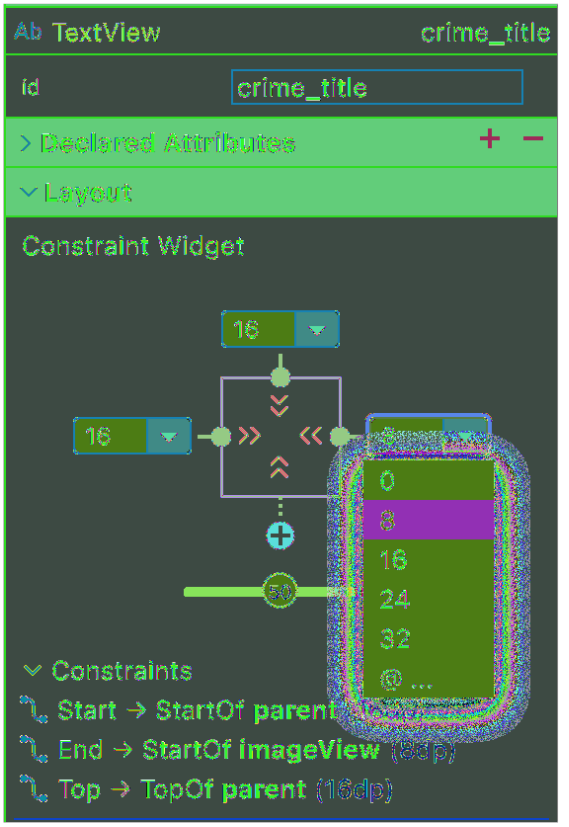
Modify your layout so that all these constraints are in place (Figure 11.25). If a constraint does not work as you expected, press Command-Z (Ctrl-Z) to undo and try again.

Figure 11.25 **`TextView`** constraints



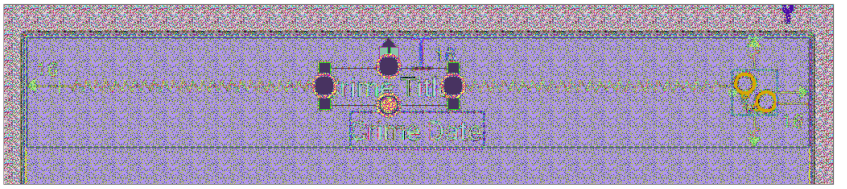
Now you are going to add margins to the constraints on your **`TextView`**. With Crime Title still selected in the preview, check out the attributes window to the right. Because you added constraints to the top, left, and right of the **`TextView`**, drop-down menus appear to allow you to select the margin for each constraint (Figure 11.26). Select 16dp for the left and top margins and 8dp for the right margin.

Figure 11.26 Adding margins to the **`TextView`**



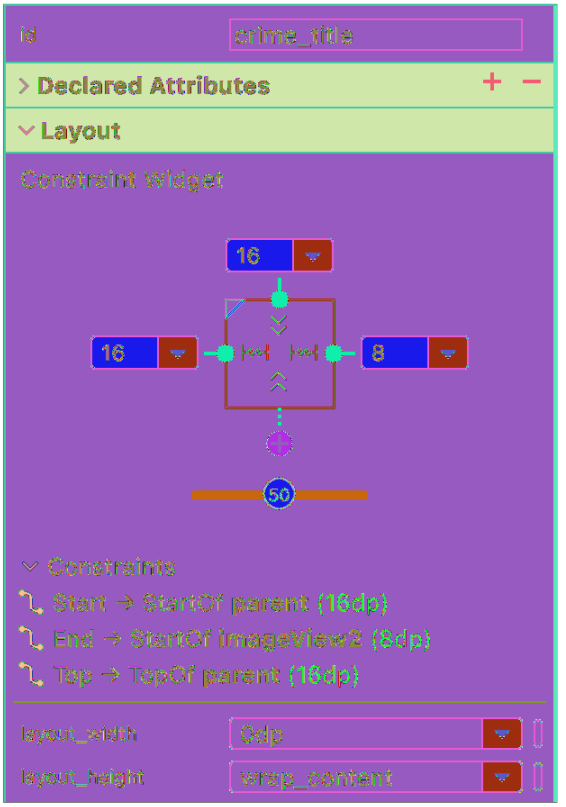

Verify that your constraints look like Figure 11.27.

Figure 11.27 Title **`TextView`**'s constraints



Now that the constraints are set up, you can restore the width of the title **`TextView`** to its full glory. Adjust its horizontal view setting to 0 dp (match constraint) to allow the **`TextView`** to fill all the space available within its constraints. Make the vertical view setting `wrap_content`, if it is not already, so that the **`TextView`** will be just tall enough to show the title of the crime. Verify that your settings match those shown in Figure 11.28.

Figure 11.28 `crime_title` view settings

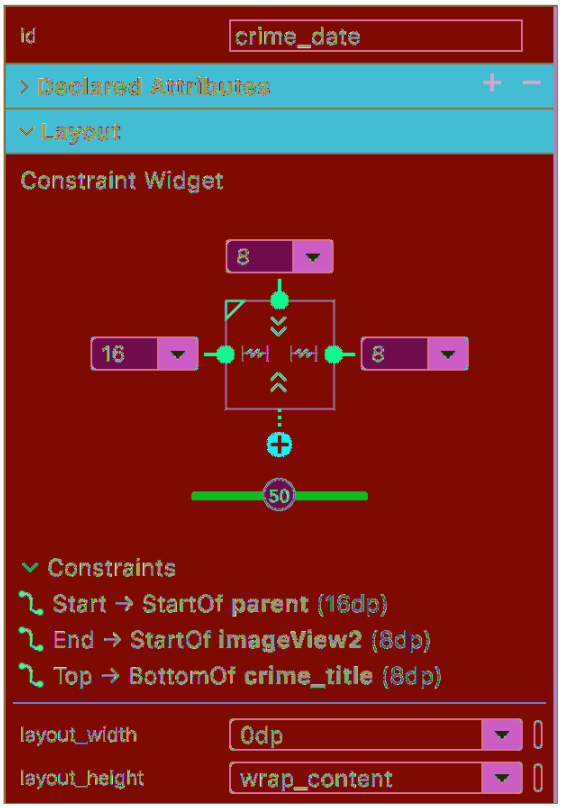



Now, add constraints to the date **`TextView`**. Select crime_date in the component tree. Repeat the steps from the title **`TextView`** to add three constraints:

- from the left side of your view to the left side of the parent, with a `16dp` margin
- from the top of your view to the bottom of the crime title, with an `8dp` margin
- from the right of your view to the left side of the **`ImageView`**, with an `8dp` margin

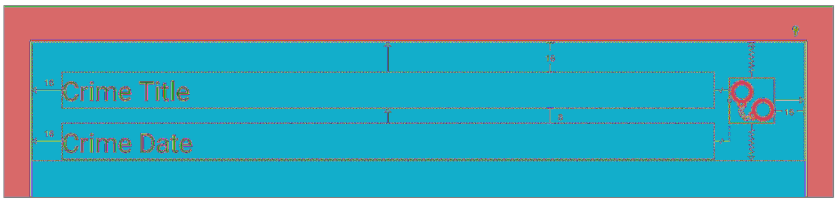
After adding the constraints, adjust the properties of the **`TextView`**. You want the width of your date **`TextView`** to be match_constraint and the height to be wrap_content, just like the title **`TextView`**. Verify that your settings match those shown in Figure 11.29.

Figure 11.29 `crime_date` view settings




Your layout in the preview should look similar to Figure 11.1, at the beginning of the chapter. Up close, your preview should match Figure 11.30.

Figure 11.30 Final constraints up close



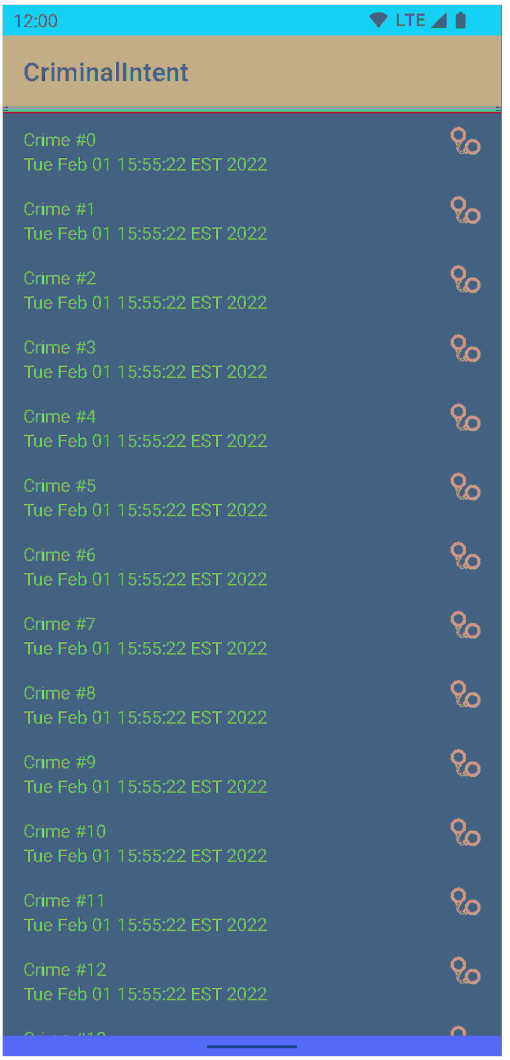

Switch to the code view in the editor tool window to review the XML resulting from the changes you made in the layout editor. Red underlines no longer appear under the **`TextView`** tags. This is because the **`TextView`** views are now adequately constrained, so the **`ConstraintLayout`** that contains them can figure out where to properly position the views at runtime.

Two yellow warning indicators remain related to the **`TextView`**s, and if you explore them you will see that the warnings have to do with their hardcoded strings. These warnings would be important for a production application, but for CriminalIntent you can disregard them. (If you prefer, feel free to follow the advice to extract the hardcoded text into string resources. This will resolve the warnings.)

Additionally, one warning remains on the **`ImageView`**, indicating that it does not have a content description. For now, you can disregard this warning as well. You will address this issue when you learn about accessibility in Chapter 19. In the meantime, your app will function, although the image will not be accessible to users utilizing a screen reader.

Run CriminalIntent and verify that you see all three components lined up nicely in each row of your **`RecyclerView`** (Figure 11.31).

Figure 11.31 Now with three views per row



## Making list items dynamic

Now that the layout includes the correct constraints, update the **`ImageView`** so that the handcuffs are only shown on crimes that have been solved.

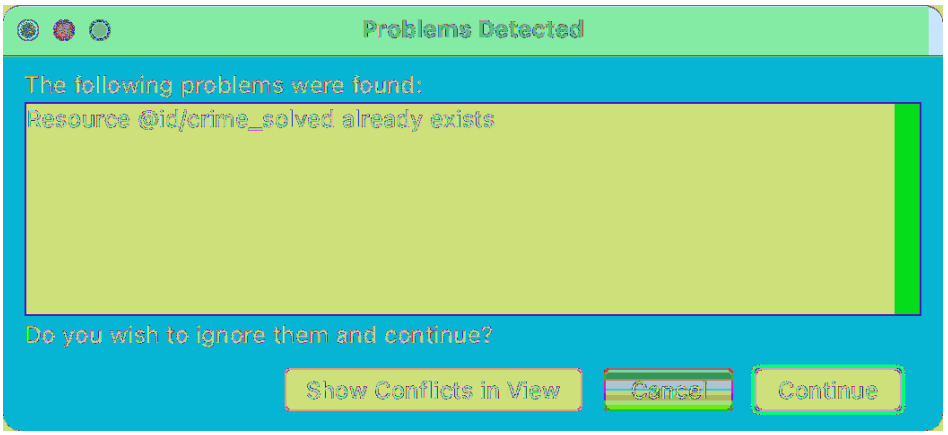
First, update the ID of your **`ImageView`**. When you added the **`ImageView`** to your **`ConstraintLayout`**, it was given a default name. That name is not very descriptive. In the design view, select your **`ImageView`** and, in the attributes window, update the ID attribute to crime_solved (Figure 11.32).

Figure 11.32 Updating the image ID




You will be asked whether Android Studio should update all usages of the ID; select Refactor. Next, Android Studio will warn you that you are already using the ID `crime_solved` (Figure 11.33).

Figure 11.33 Reusing an ID




The `crime_solved` ID is used in both the `list_item_crime.xml` and `fragment_crime_detail.xml` layouts. You might think that reusing an ID would be a problem, but in this case it is not. Layout IDs only need to be unique in the same layout. Since your IDs are defined in different layout files, there is no problem using the same ID in both. Click Continue to ignore this warning.

With a proper ID in place, you are ready to update your code. Open `CrimeListAdapter.kt`. In **`CrimeHolder`**, add an **`ImageView`** instance variable and toggle its visibility based on the solved status of the crime.

### Listing 11.1 Updating handcuff visibility (`CrimeListAdapter.kt`)

```
class CrimeHolder(
    private val binding: ListItemCrimeBinding
) : RecyclerView.ViewHolder(binding.root) {
    ...
    fun bind(crime: Crime) {
        ...
        binding.root.setOnClickListener {
            Toast.makeText(
                binding.root.context,
                "${crime.title} clicked!",
                Toast.LENGTH_SHORT
            ).show()
        }

        binding.crimeSolved.visibility = if (crime.isSolved) {
            View.VISIBLE
        } else {
            View.GONE
        }
    }
    ...
}
```

Run CriminalIntent and verify that the handcuffs now appear on every other row. (Check **`CrimeListViewModel`** if you do not recall why this would be the case.)

## Styles, Themes, and Theme Attributes

Now you are going to add a few more tweaks to the design in `list_item_crime.xml` and, in the process, answer some lingering questions you might have about views and attributes.

In previous chapters, you used XML attributes to define various aspects of your views, such as the text within a **`TextView`** or the padding on a **`LinearLayout`**. You can also use XML attributes to set the size or color of text within a **`TextView`**.

Navigate back to the design view of `list_item_crime.xml`. Select crime_title and, in the attributes window, expand the textAppearance section under Common Attributes.

You could style your text by setting the individual attributes in this section, such as textSize and textColor – but that is not a sustainable approach for large applications. If you wanted to apply the same styling to other text in other layouts, you would have to copy those settings to the appropriate places. As your app gets more complex and your text's appearance becomes more stylized, duplicating settings all over your codebase quickly becomes unmaintainable.

A step in the right direction would be to define a custom style and reference it when setting your text's appearance. This might look like:

```
<style name="FancyListItemText">
    <item name="android:textSize">20sp</item>
    <item name="android:textColor">@color/black</item>
</style>
```

You can define a custom style much like you define string resources in XML. These styles reside in a `themes.xml` file in the `/res/values/` directory. In your layout, you reference your style using the `@style/` prefix and whatever name you gave the style. Then, wherever you want to use that custom style, you only have to set a single attribute: `android:textAppearance="@style/FancyListItemText"`.

However, since you are already using the Material Design library to theme your app, it would be better to lean on that library to provide appropriate styling for your text. When you created CriminalIntent, the wizard set up a *theme* for the app that is referenced on the `application` tag in the manifest. By default, that theme extends a theme provided by the Material Design library.

```
<style name="Theme.CriminalIntent"
    parent="Theme.MaterialComponents.DayNight.DarkActionBar">
    <!-- Primary brand color. -->
    <item name="colorPrimary">@color/purple_500</item>
    ...
</style>
```

A theme is a collection of styles, and it defines *theme attributes* to reference those styles. Structurally, a theme is itself a style resource whose attributes point to other style resources.

The theme attribute you will use here is a reference to a custom style defined by the Material Design library. The Material Design library uses theme attributes heavily to provide access to different aspects of the design system, like color, shape, and typographic style. The AppCompat library and even the Android platform also offer theme attributes to allow you to hook into their provided themes.

Unlike other resource types, which are referenced with the `@` character followed by the resource type (like `@string/` or `@drawable/`), theme attributes use the `?attr/` prefix. You saw this syntax in Chapter 9 when you styled the **`TextView`** labels for **`CrimeDetailFragment`**. Now, you want to use the Headline 6 typographic style for your title text, so enter `?attr/textAppearanceHeadline6` for the `textAppearance` attribute (Figure 11.34).

Figure 11.34 Updating the title color and size



Run CriminalIntent and be amazed at how much better everything looks with a fresh coat of paint (Figure 11.35).

Figure 11.35 Fresh paint



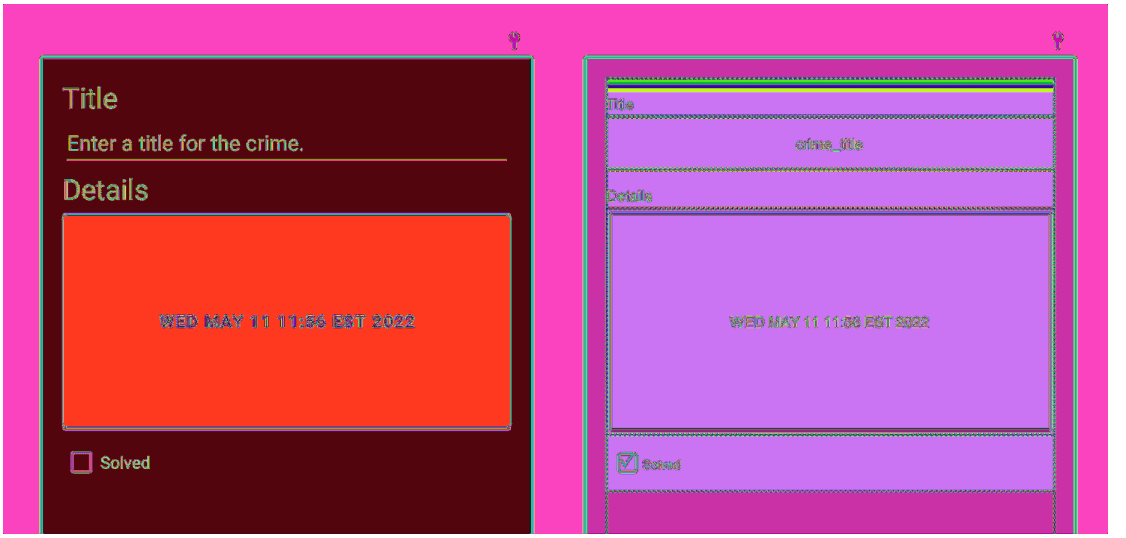
## For the More Curious: Margins vs Padding

In both GeoQuiz and CriminalIntent, you have given views margin and padding attributes. Beginning developers sometimes get confused about these two. Now that you understand what a layout parameter is, the difference is easier to explain.

Margin attributes are layout parameters. They determine the distance between views. Because a view can only know about itself, margins must be the responsibility of the view's parent.

Padding, on the other hand, is not a layout parameter. The `android:padding` attribute tells a view how much bigger than its contents it should draw itself. For example, say you wanted the date button to be spectacularly large without changing its text size (Figure 11.36).

Figure 11.36 I like big buttons and I cannot lie...




You could add the following attribute to the **`Button`**:

```
<Button
    android:id="@+id/crime_date"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:padding="80dp"
    tools:text="Wed May 11 11:56 EST 2022"/>
```

Alas, you should probably remove this attribute before continuing.

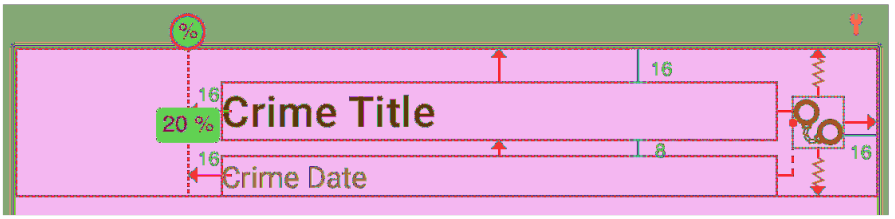
## For the More Curious: Advanced Features in ConstraintLayout

**ConstraintLayout** has additional capabilities to help arrange its child views. In this chapter you positioned views by constraining them to the parent as well as to other, sibling views. **ConstraintLayout** also includes helper views, such as **Guideline**s, that simplify arranging views on the screen.

Guidelines do not display on the app screen; they are just a tool to help you position views. There are both horizontal and vertical guidelines, and they can be placed at a specific location on the screen using `dp` values or by setting them to be a percentage of the screen. Other views can be constrained to the guideline to ensure that they appear at the same location, even if the screen size is different.

Figure 11.37 shows an example of a vertical **Guideline**. It is positioned at 20% of the width of the parent. Both the crime title and date have a left constraint to the **Guideline** instead of to the parent.

Figure 11.37 Using a **Guideline**




Another tool offered by **ConstraintLayout** is **MotionLayout**. **MotionLayout** is an extension of **ConstraintLayout** that simplifies adding animations to your views. To use **MotionLayout**, you create a MotionScene file that describes how the animations should be performed and which views map to each other in the starting and ending layouts. You can also set **Keyframe**s that provide intermediary views in the animation. **MotionLayout** can animate from the starting view through the various keyframes you provide, then ensure that the view animates to the ending layout appropriately.

## Challenge: Formatting the Date

The **Date** object is more of a timestamp than a conventional date. A timestamp is what you see when you call **toString()** on a **Date**, so that is what you have in each of your **RecyclerView** rows. While timestamps make for good documentation, it might be nicer if the rows just displayed the date as humans think of it – like "May 11, 2022." You can do this with an instance of the **android.text.format.DateFormat** class. The place to start is the reference page for this class in the Android documentation.

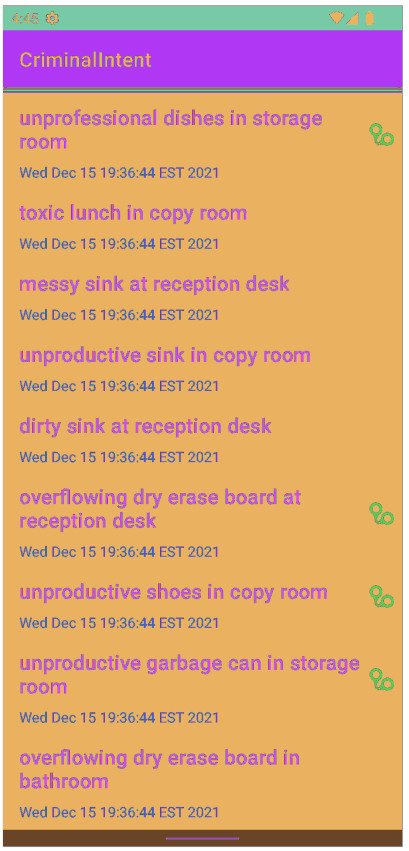
You can use functions in the **DateFormat** class to get a common format. Or you can prepare your own format string. For a more advanced challenge, create a format string that will display the day of the week as well – like "Wednesday, May 11, 2022."

# 12
# Coroutines and Databases

Almost every application needs a place to save data for the long term. In this chapter you will implement a database for CriminalIntent and seed it with dummy data. However, reading and writing to a database is a slow process (in the scale at which computers operate), so first you will learn how to perform operations asynchronously – allowing multiple tasks to run at the same time.

Once you understand the basics of working with asynchronous code on Android and you have implemented the database, you will update the app to pull crime data from the database and display it in the crime list (Figure 12.1).

Figure 12.1 Displaying crimes from the database



## An Introduction to Asynchronous Code on Android

Many programming languages rely on the concept of a *thread* for work that runs in the background – or, as it is often called, *asynchronously*. Threads are responsible for managing execution of your program. A thread has a sequence of instructions that it executes, performing them in the order they are declared in.

An individual thread can only do so much work in a set period of time, so to keep the system responsive to the user while also performing complicated tasks, developers distribute work across many threads. On an individual device, the system can have multiple threads, and each of those threads can execute their instructions simultaneously.

The primary thread, which manages the work the user interacts with directly, is called the *main thread* or *UI thread*. Up until now, all the code that you have written has executed on the main thread. In fact, on Android all the code that directly interacts with the UI *must* be executed on the main thread.

On the other hand, Android forbids code that makes network requests or interacts with a database on the main thread. These kinds of operations can take a long time to execute, so they can *block* the thread. When a thread is blocked, it is unable to respond to user input, making your application appear frozen. Thankfully, you have access to many threads to perform various types of work. For the database work in this chapter, you will offload that execution to a *background thread*.

By using a background thread, you will be able to execute long-running work while the main thread continues without pause. Once you have successfully queried the database for the list of crimes on the background thread, you will pass that list back over to the main thread, where you can update your **`RecyclerView`**.

Unfortunately, threads are a fairly low-level API, making them difficult to work with. There is an implementation of threads on the Java platform, and you can create threads directly on Android, but it is very easy to make mistakes when doing so – mistakes that can lead to the application wasting resources or crashing unexpectedly.

This is where *coroutines* come in. Coroutines are Kotlin's first-party solution for defining work that will run asynchronously and are fully supported on Android. They are based on the idea of functions being able to *suspend*, meaning that a function can be paused until a long-running operation completes. When the code running in a coroutine is suspended, the thread that the coroutine was executing on is free to work on other things, like drawing your UI, responding to touch events, or making more expensive calculations.

Coroutines provide a high-level and safer set of tools to help you build asynchronous code. Under the hood, Kotlin's coroutines use threads to perform work in parallel, but you often do not have to worry about this detail. Coroutines make it easy to start work on the main thread, hop over to a background thread to perform asynchronous work, and then return the result back to the main thread.

To keep this book at a reasonable length, we cannot explain Kotlin coroutines in full. If coroutines are entirely new to you, JetBrains has excellent documentation on how to use them (`kotlinlang.org/docs/coroutines-overview.html`). Also, there is a little book written by some very cool folks titled "Kotlin Programming: The Big Nerd Ranch Guide." That book does an excellent job explaining the basics of coroutines and how to use them, as well as other Kotlin topics. We highly recommend that book.

In this chapter and throughout this book, we will primarily focus on how to use coroutines in the context of an Android app.

## Using coroutines

Your work will begin with the familiar step of adding dependencies to your `build.gradle` file. You need to add the core Coroutines library, a library to hook up the main thread in Android to your coroutines, and a library to enable you to safely consume data coming from a coroutine inside your **`Fragment`**. Open the `build.gradle` file labeled (Module: CriminalIntent.app) and add those three dependencies:

Listing 12.1 Adding coroutines to your project's build (`app/build.gradle`)

```
...
dependencies {
    ...
    implementation 'androidx.recyclerview:recyclerview:1.2.1'
    implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-core:1.6.0'
    implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-android:1.6.0'
    implementation 'androidx.lifecycle:lifecycle-runtime-ktx:2.4.1'
    testImplementation 'junit:junit:4.13.2'
    ...
}
```

Do not forget to click the  Sync Project with Gradle Files button or the Sync Now button after you have made these changes.

Before you get into using coroutines with a database, let's take a quick tour of coroutines using the existing hardcoded data. To run your code in a coroutine, you use a *coroutine builder*. A coroutine builder is a function that creates a new coroutine. Most coroutine builders also start executing the code within the coroutine immediately after creating it.

Several builders are defined for you in the Coroutines library. The most commonly used coroutine builder is **`launch`**, a function that is defined as an extension to a class called **`CoroutineScope`**.

Every coroutine builder launches its coroutines inside a *coroutine scope*. A coroutine's scope has control over how the coroutine's code executes. This includes setting up the coroutine, canceling the coroutine, and choosing which thread will be used to run the code.

On Android, this idea of scopes maps neatly onto the various lifecycles you have encountered so far. The **`Activity`**, **`Fragment`**, and **`ViewModel`** classes have unique lifecycles and coroutine scopes to match. For **`ViewModel`**s, you have access to the `viewModelScope` class property. This `viewModelScope` is available from the time your **`ViewModel`** is initialized, and it cancels any coroutine work still running when the **`ViewModel`** is cleared out from memory.

Open `CrimeListViewModel.kt` and launch a coroutine using the `viewModelScope` property, wrapping the initialization of your list of crimes inside the new coroutine.

### Listing 12.2 Launching your first coroutine (`CrimeListViewModel.kt`)

```
class CrimeListViewModel : ViewModel() {

    val crimes = mutableListOf<Crime>()

    init {
        viewModelScope.launch {
            for (i in 0 until 100) {
                val crime = Crime(
                    id = UUID.randomUUID(),
                    title ="Crime #$i",
                    date = Date(),
                    isSolved = i % 2 == 0
                )

                crimes += crime
            }
        }
    }
}
```

On its own, that code does not do much. But now that you have launched a coroutine, you can invoke suspending functions within it. A suspending function is a function that can be paused until a long-running operation can be completed. This may sound similar to long-running functions that block a thread; the big difference is that coroutines are able to be much more resource friendly.

Behind the scenes, Kotlin saves and restores the function state between suspending function calls. This allows the original function call to be temporarily freed from memory until it is ready to be resumed. Because of these optimizations, coroutines are considerably more resource efficient than native threads.

One of the most basic suspending functions is **`delay(timeMillis: Long)`**. As the name suggests, this function delays a coroutine – without blocking a thread – for a specified number of milliseconds. Add a call to this function, as well as some logging calls, to your initialization block.

### Listing 12.3 Delaying work (`CrimeListViewModel.kt`)

```
private const val TAG = "CrimeListViewModel"

class CrimeListViewModel : ViewModel() {

    val crimes = mutableListOf<Crime>()

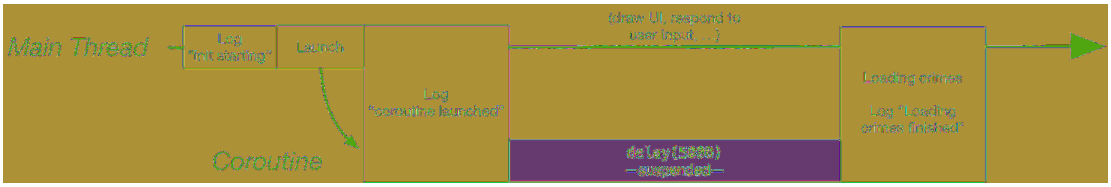
    init {
        Log.d(TAG, "init starting")
        viewModelScope.launch {
            Log.d(TAG, "coroutine launched")
            delay(5000)
            for (i in 0 until 100) {
                val crime = Crime(
                    id = UUID.randomUUID(),
                    title ="Crime #$i",
                    date = Date(),
                    isSolved = i % 2 == 0
                )

                crimes += crime
            }
            Log.d(TAG, "Loading crimes finished")
        }
    }
}
```

Open Logcat and search for CrimeListViewModel, then run your app. You should see the two initialization messages print out and then, five seconds later, “Loading crimes finished” should print out.

Because **`delay`** is running inside a coroutine, during the five seconds that the function is counting milliseconds your UI is still capable of drawing any new updates and can instantly respond to user input (Figure 12.2). (If this were not the case, your users would see a system dialog saying “CriminalIntent isn’t responding” with the options to kill the app or wait for it to respond.)

### Figure 12.2 A timeline of the coroutine work done in **`CrimeListViewModel`**




Coroutines allow you to perform asynchronous code in a resource- and performance-friendly way. If you were using a thread directly, you would need to do more setup to accomplish the same result correctly and without wasting system resources.

(If you took a peek at your Android device when running this new code, you might have noticed that the crimes no longer display in your **`RecyclerView`**. You might already know why that is happening, and you will learn how to solve that problem shortly.)

You can also define your own suspending functions. Suspending functions can take in parameters, use visibility modifiers, and return values, just like regular functions. All you need to do to convert one of your regular functions to a suspending function is add the `suspend` modifier to the function definition.

Marking a function as a suspending function does limit the number of places where you can invoke it, because you need a coroutine scope to invoke a suspending function. But when you make a function a suspending function, you can then call other suspending functions within it. To see this, move your crime loading code into its own suspending function. Within the new **`loadCrimes()`** suspending function, you can call the **`delay()`** suspending function.

### Listing 12.4 Defining your own suspending function (`CrimeListViewModel.kt`)

```
private const val TAG = "CrimeListViewModel"

class CrimeListViewModel : ViewModel() {
    ...
    init {
        Log.d(TAG, "init starting")
        viewModelScope.launch {
            Log.d(TAG, "coroutine launched")
            delay(5000)
            for (i in 0 until 100) {
                val crime = Crime(
                    id = UUID.randomUUID(),
                    title = "Crime #$i",
                    date = Date(),
                    isSolved = i % 2 == 0
                )

                crimes += crime
            }
            crimes += loadCrimes()
            Log.d(TAG, "Loading crimes finished")
        }
    }

    suspend fun loadCrimes(): List<Crime> {
        val result = mutableListOf<Crime>()
        delay(5000)
        for (i in 0 until 100) {
            val crime = Crime(
                id = UUID.randomUUID(),
                title = "Crime #$i",
                date = Date(),
                isSolved = i % 2 == 0
            )

            result += crime
        }
        return result
    }
}
```

Run your app again to confirm that the behavior is the same.

## Consuming data from coroutines

Right now, you access the `crimes` property from your **`CrimeListViewModel`** in the **`onCreateView(…)`** callback within your **`CrimeListFragment`**. This callback is invoked right after the fragment is created. But with the changes you have made in this chapter, you do not add the list of crimes to the `crimes` property until five seconds have passed. That is why your **`RecyclerView`** is no longer showing the list of crimes.

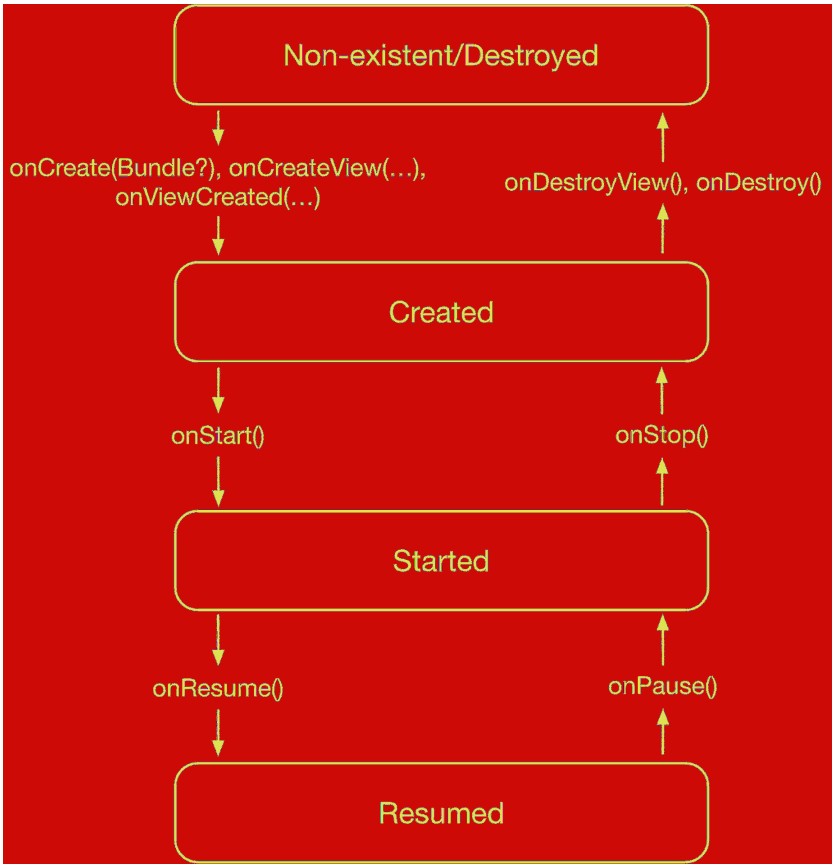
This is a textbook example of a race condition, a common problem in multithreaded code where the timing of independent events affects the output of the code. In this case, it is extremely unlikely that the `crimes` property can be properly loaded with data before the **`onCreateView(…)`** function is invoked.

Instead of trying to access your asynchronously loaded data in an error-prone way, you should instead reach for a more reliable approach. As we mentioned earlier, the **`Fragment`** and **`Activity`** classes have properties to access coroutine scopes for their respective lifecycles. Both classes have a `lifecycleScope` property, but you should prefer the `viewLifecycleScope` when using coroutines with **`Fragment`**s.

(The reasons for using `viewLifecycleScope` in a **`Fragment`** go back to the same subtle detail about memory management with **`Fragment`**s we discussed in the section called Fragments and memory management in Chapter 9. It is wasteful and potentially dangerous to execute coroutine code when your **`Fragment`** does not have a view.)

Figure 12.3 shows the fragment lifecycle.

Figure 12.3 Fragment lifecycle diagram




`viewLifecycleScope` is active for as long as the view is in memory (in other words, after **`onViewCreated()`** but before **`onDestroyView()`**). After the view is destroyed, the coroutine scope – and all work within it – is canceled. But you should only update the UI while the **`Fragment`** is in the started lifecycle state or higher. It does not make sense to update the UI when it is not visible.

To do this, you will ensure that you only load crimes when the view is running. Instead of relying on the coroutine scope to only run during the appropriate time, you will have to manage that work yourself. The way you manage coroutine work is via the **`Job`** class. When you launch a new coroutine, a **`Job`** instance is returned to you, and you can use it to cancel the work at the appropriate time.

Launch your work in the **`onStart()`** callback and then cancel it in the **`onStop()`** callback.

### Listing 12.5 Calling coroutines from your **`CrimeListFragment`** (`CrimeListFragment.kt`)

```
class CrimeListFragment : Fragment() {
    ...
    private val crimeListViewModel: CrimeListViewModel by viewModels()

    private var job: Job? = null
    ...
    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        ...
    }

    override fun onStart() {
        super.onStart()

        job = viewLifecycleOwner.lifecycleScope.launch {
            val crimes = crimeListViewModel.loadCrimes()
            binding.crimeRecyclerView.adapter = CrimeListAdapter(crimes)
        }
    }

    override fun onStop() {
        super.onStop()
        job?.cancel()
    }
    ...
}
```

This approach will work, but it is annoying to keep a reference to a **`Job`** and to make sure you cancel the work when the fragment goes in the background. That is where the **`repeatOnLifecycle(…)`** function comes into play.

With the **`repeatOnLifecycle(…)`** function, you can execute coroutine code while your fragment is in a specified lifecycle state. For example, you only want this coroutine code to execute while your fragment is in the started or resumed state. Also, **`repeatOnLifecycle`** is itself a suspending function. You will launch it in your view lifecycle scope, which will cause your work to be canceled permanently when your view is destroyed.

You are not required to call the **`repeatOnLifecycle(…)`** function in the **`onStart()`** callback. Normally, you use the **`onViewCreated(…)`** callback to hook up listeners to views and to set the data within those views. This is the perfect spot to handle your coroutine code, too. Change your implementation to use **`repeatOnLifecycle(…)`**:

### Listing 12.6 Using **`repeatOnLifecycle(…)`** (`CrimeListFragment.kt`)

```
class CrimeListFragment : Fragment() {
    ...
    private var job: Job? = null
    ...
    override fun onStart() {
        super.onStart()

        job = viewLifecycleOwner.lifecycleScope.launch {
            val crimes = crimeListViewModel.loadCrimes()
            binding.crimeRecyclerView.adapter = CrimeListAdapter(crimes)
        }
    }

    override fun onStop() {
        super.onStop()
        job?.cancel()
    }

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        viewLifecycleOwner.lifecycleScope.launch {
            viewLifecycleOwner.repeatOnLifecycle(Lifecycle.State.STARTED) {
                val crimes = crimeListViewModel.loadCrimes()
                binding.crimeRecyclerView.adapter =
                    CrimeListAdapter(crimes)
            }
        }
    }
    ...
}
```

Your code will behave exactly as it did with **`onStart()`** and **`onStop()`**, but now there are fewer lifecycle methods to override and you will not have to worry about forgetting to cancel your **`Job`**. **`repeatOnLifecycle(…)`** handles all that for you.

**`repeatOnLifecycle(…)`** will begin executing your coroutine code when your fragment enters the started state and will continue running in the resumed state. But if your app is backgrounded and your fragment is no longer visible, **`repeatOnLifecycle(…)`** will cancel the work once the fragment falls from the started state to the created state. If your lifecycle re-enters the started state without fully being destroyed, your coroutine will be restarted from the beginning, *repeating* its work. (This explains the function's name.)

Before running your app, clean up some unneeded code. Remove your **`onCreate(…)`** implementation that logs the number of crimes; you do not need it anymore. Also, delete the code that tries to initialize your **`CrimeListAdapter`** with missing data.

### Listing 12.7 Cleaning up (`CrimeListFragment.kt`)

```
~~private const val TAG = "CrimeListFragment"~~

class CrimeListFragment : Fragment() {
    ...
    ~~override fun onCreate(savedInstanceState: Bundle?) {~~
        ~~super.onCreate(savedInstanceState)~~
        ~~Log.d(TAG, "Total crimes: ${crimeListViewModel.crimes.size}")~~
    ~~}~~

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        _binding = FragmentCrimeListBinding.inflate(inflater, container, false)

        binding.crimeRecyclerView.layoutManager = LinearLayoutManager(context)

        ~~val crimes = crimeListViewModel.crimes~~
        ~~val adapter = CrimeListAdapter(crimes)~~
        ~~binding.crimeRecyclerView.adapter = adapter~~

        return binding.root
    }
    ...
}
```

Run CriminalIntent. When the fragment is created, there is a five second delay, and then the crimes load and display in your **`RecyclerView`**. That is good! But now try rotating your device.

Another five second delay. Unfortunately, your list of crimes is being recalculated every time your fragment is re-created. In GeoQuiz, you used a **`ViewModel`** to store state across configuration changes. It is also an excellent place to perform expensive calculations that would be painful to do every time your fragment is created. You will learn how to do this by the end of this chapter.

First, let's implement your database.

# Creating a Database

On Android, there are many ways to create and access a database. In this book, we will use the Room library from Google. Room is a Jetpack architecture component library that simplifies database setup and access. It allows you to define your database structure and queries using annotated Kotlin classes.

## Room architecture component library

Room is composed of an API, annotations, and a compiler. The API contains classes you extend to define your database and build an instance of it. You use the annotations to indicate things like which classes need to be stored in the database, which class represents your database, and which class specifies the accessor functions to your database tables. The compiler processes the annotated classes and generates the implementation of your database.

To use Room, you first need to add the dependencies it requires. Add the `room-runtime`, `room-ktx`, and `room-compiler` dependencies to your `app/build.gradle` file.

### Listing 12.8 Adding dependencies (`app/build.gradle`)

```
plugins {
    id 'com.android.application'
    id 'org.jetbrains.kotlin.android'
    id 'org.jetbrains.kotlin.kapt'
}

android {
    ...
}
...
dependencies {
    ...
    implementation 'androidx.lifecycle:lifecycle-runtime-ktx:2.4.1'
    implementation 'androidx.room:room-runtime:2.4.2'
    implementation 'androidx.room:room-ktx:2.4.2'
    kapt 'androidx.room:room-compiler:2.4.2'
    ...
}
```

Near the top of the file, you added a new *plugin*. Plugins are a way to add functionality and features to the configuration of a project.

`kapt` stands for "Kotlin annotation processing tool." `kapt` enables your project to generate code for you when compiling your app. You have already used two tools that generate code: the **`R`** class and View Binding. Those two tools are bundled in the Android Gradle plugin; when using other libraries to generate code, you will often rely on `kapt` to handle the code generation for you. `kapt` can generate code during the build process, and it makes that generated code accessible throughout the rest of your project.

The first dependency you added, `room-runtime`, is for the Room API, containing all the classes and annotations you will need to define your database. The second dependency, `room-ktx`, adds Kotlin-specific functionality and support for coroutines. And the third dependency, `room-compiler`, is for the Room compiler, which will generate your database implementation based on the annotations you specify. The compiler uses the `kapt` keyword, instead of `implementation`, so that the generated classes from the compiler are visible to Android Studio, thanks to the `kotlin-kapt` plugin.

There is one last change you need to make in your build setup. Much like how you declared which version of a particular library you want in your build (such as 2.4.0 for `room-runtime`), you need to declare which version of the `kapt` plugin you want to use. This is defined at the project level, in the `build.gradle` file labeled (Project: CriminalIntent):

### Listing 12.9 Defining plugin versions (`build.gradle`)

```
// Top-level build file where you can add configuration options common to all
    sub-projects/modules.
plugins {
    id 'com.android.application' version '7.1.2' apply false
    id 'com.android.library' version '7.1.2' apply false
    id 'org.jetbrains.kotlin.android' version '1.6.10' apply false
    id 'org.jetbrains.kotlin.kapt' version '1.6.10' apply false
}

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

If your `org.jetbrains.kotlin.android` version is older than 1.6.10, update it to match the `kapt` plugin.

Do not forget to sync your Gradle files. With your dependencies in place, you can move on to preparing your model layer for storage in the database.

There are three steps to creating a database with Room:

- annotating your model class to make it a database entity
- creating the class that will represent the database itself
- creating a type converter so that your database can handle your model data

Room makes each of these steps straightforward, as you are about to see.

## Defining entities

Room structures the database tables for your application based on the *entities* you define. Entities are model classes you create and annotate with `@Entity`. Room will create a database table for any class with that annotation that is associated with a database.

Since you want to store crime objects in your database, update **`Crime`** to be a Room entity. Open `Crime.kt` and add two annotations:

### Listing 12.10 Making **`Crime`** an entity (`Crime.kt`)

```
@Entity
data class Crime(
    @PrimaryKey val id: UUID,
    val title: String,
    val date: Date,
    val isSolved: Boolean
)
```

The first annotation, `@Entity`, is applied at the class level. This entity annotation indicates that the class defines the structure of a table, or set of tables, in the database. In this case, each row in the table will represent an individual **`Crime`**. Each property defined on the class will be a column in the table, with the name of the property as the name of the column. The table that stores your crimes will have four columns: `id`, `title`, `date`, and `isSolved`.

The other annotation you added is `@PrimaryKey`, which you added to the `id` property. The *primary key* in a database is a column that holds data that is unique for each entry, or row, so that it can be used to look up individual entries. The `id` property is unique for every **`Crime`**, so by adding `@PrimaryKey` to this property you will be able to query a single crime from the database using its `id`.

Now that your **`Crime`** class is annotated, you can move on to creating your database class.

## Creating a database class

Entity classes define the structure of database tables. A single entity class could be used across multiple databases, should your app have more than one database. That case is not common, but it is possible. For this reason, an entity class is not used by Room to create a table unless you explicitly associate it with a database, which you will do shortly.

First, create a new package called `database` for your database-specific code. In the project tool window, right-click the `com.bignerdranch.android.criminalintent` folder and choose New → Package. Name your new package database.

Now, create a new class called **`CrimeDatabase`** in the `database` package and define the class as shown below.

Listing 12.11 Initial **`CrimeDatabase`** class (`database/CrimeDatabase.kt`)

```
@Database(entities = [ Crime::class ], version=1)
abstract class CrimeDatabase : RoomDatabase() {
}
```

The `@Database` annotation tells Room that this class represents a database in your app. The annotation itself requires two parameters. The first parameter is a list of entity classes, which tells Room which entity classes to use when creating and managing tables for this database. In this case, you only pass the **`Crime`** class, since it is the only entity in the app.

The second parameter is the version of the database. When you first create a database, the version should be 1. As you develop your app in the future, you may add new entities and new properties to existing entities. When this happens, you will need to modify your entities list and increment your database version to tell Room something has changed. (You will do this in Chapter 16.)

The database class itself is empty at this point. **`CrimeDatabase`** extends from **`RoomDatabase`** and is marked as `abstract`, so you cannot make an instance of it directly. You will learn how to use Room to get a database instance you can use later in this chapter.

## Creating a type converter

Room uses SQLite under the hood. SQLite is an open-source relational database, like MySQL or PostgreSQL. (SQL, short for Structured Query Language, is a standard language used for interacting with databases. People pronounce “SQL” as either “sequel” or as an initialism, “S-Q-L.”) Unlike other databases, SQLite stores its data in simple files you can read and write using the SQLite library. Android includes this SQLite library in its standard library, along with some additional helper classes.

Room makes using SQLite even easier and cleaner, serving as an object-relational mapping (or ORM) layer between your Kotlin objects and database implementation. For the most part, you do not need to know or care about SQLite when using Room, but if you want to learn more you can visit `www.sqlite.org`, which has complete SQLite documentation.

Room is able to store primitive types, enum classes, and the **`UUID`** type with ease in the underlying SQLite database tables, but other types will cause issues. Your **`Crime`** class contains a property of the type **`Date`**, which Room does not know how to store by default. You need to give the database a hand so it knows how to store that type and how to pull it out of the database table correctly.

To tell Room how to convert your data type, you specify a *type converter*. A type converter tells Room how to convert a specific type to the format it needs to store in the database. You will need two functions, which you will annotate with `@TypeConverter`. One will tell Room how to convert the type to store it in the database, and the other will tell Room how to convert from the database representation back to the original type.

Create a class called **`CrimeTypeConverters`** in the `database` package and add two functions to convert the **`Date`** type.

### Listing 12.12 Adding **`TypeConverter`** functions (`database/CrimeTypeConverters.kt`)

```
class CrimeTypeConverters {
    @TypeConverter
    fun fromDate(date: Date): Long {
        return date.time
    }

    @TypeConverter
    fun toDate(millisSinceEpoch: Long): Date {
        return Date(millisSinceEpoch)
    }
}
```

Make sure you import the `java.util.Date` version of the **`Date`** class.

Declaring the converter functions does not enable your database to use them. You must explicitly add the converters to your database class.

### Listing 12.13 Enabling **`TypeConverter`**s (`database/CrimeDatabase.kt`)

```
@Database(entities = [ Crime::class ], version=1)
@TypeConverters(CrimeTypeConverters::class)
abstract class CrimeDatabase : RoomDatabase() {
}
```

By adding the `@TypeConverters` annotation and passing in your **`CrimeTypeConverters`** class, you tell your database to use the functions in that class when converting your types.

With that, your database and table definitions are complete.

## Defining a Data Access Object

A database table does not do much good if you cannot edit or access its contents. The first step to interacting with your database tables is to create a *data access object*, or DAO (pronounced either to rhyme with "cow" or as an initialism). A DAO is an interface that contains functions for each database operation you want to perform. In this chapter, CriminalIntent's DAO needs two query functions: one to return a list of all crimes in the database and another to return a single crime matching a given **`UUID`**.

Add a file named `CrimeDao.kt` to the `database` package. In it, define an empty interface named **`CrimeDao`** annotated with Room's `@Dao` annotation.

### Listing 12.14 Creating an empty DAO (`database/CrimeDao.kt`)

```
@Dao
interface CrimeDao {
}
```

The `@Dao` annotation lets Room know that **`CrimeDao`** is one of your data access objects. When you hook **`CrimeDao`** up to your database class, Room will generate implementations of the functions you add to this interface.

Speaking of adding functions, now is the time. Add two query functions to **`CrimeDao`**.

### Listing 12.15 Adding database query functions (`database/CrimeDao.kt`)

```
@Dao
interface CrimeDao {
    @Query("SELECT * FROM crime")
    suspend fun getCrimes(): List<Crime>

    @Query("SELECT * FROM crime WHERE id=(:id)")
    suspend fun getCrime(id: UUID): Crime
}
```

The `@Query` annotation indicates that **`getCrimes()`** and **`getCrime(UUID)`** are meant to pull information out of the database, rather than inserting, updating, or deleting items from the database. The return type of each query function in the DAO interface reflects the type of result the query will return.

The `@Query` annotation expects a string containing a SQL command as input. In most cases you only need to know minimal SQL to use Room, but if you are interested in learning more check out the SQL Syntax section at `www.sqlite.org`.

`SELECT * FROM crime` asks Room to pull all columns for all rows in the `crime` database table. `SELECT * FROM crime WHERE id=(:id)` asks Room to pull all columns from only the row whose `id` matches the ID value provided.

You might have noticed that you included the `suspend` modifier on these functions. Because you already added the `room-ktx` library to your project as a dependency, Room can implement these functions as suspending functions. Now, you can asynchronously call these functions within a coroutine.

With that, the **`CrimeDao`** is complete, at least for now. In Chapter 13 you will add a function to update an existing crime. In Chapter 15 you will add a function to insert a new crime and – if you choose to complete a challenge – another to delete a crime.

Next, you need to register your DAO class with your database class. Since the **`CrimeDao`** is an interface, Room will handle generating the concrete version of the class for you. But for that to work, you need to tell your database class to generate an instance of the DAO.

To hook up your DAO, open `CrimeDatabase.kt` and add an abstract function that has **`CrimeDao`** as the return type.

### Listing 12.16 Registering the DAO in the database (`database/CrimeDatabase.kt`)

```
@Database(entities = [Crime::class], version = 1)
@TypeConverters(CrimeTypeConverters::class)
abstract class CrimeDatabase : RoomDatabase() {
    abstract fun crimeDao(): CrimeDao
}
```

Now, when the database is created, Room will generate a concrete implementation of the DAO that you can access. Once you have a reference to the DAO, you can call any of the functions defined on it to interact with your database.

## Accessing the Database Using the Repository Pattern

To access your database, you will use the *repository pattern* recommended by Google in its Guide to App Architecture (`developer.android.com/jetpack/guide`).

A *repository* class encapsulates the logic for accessing data from a single source or a set of sources. It determines how to fetch and store a particular set of data, whether locally in a database or from a remote server. Your UI code will request all the data from the repository, because the UI does not care how the data is actually stored or fetched. Those are implementation details of the repository itself.

Because CriminalIntent is a simpler app, the repository will only handle fetching data from the database.

Create a class called **`CrimeRepository`** in the `com.bignerdranch.android.criminalintent` package and define a companion object in the class.

Listing 12.17 Implementing a repository (`CrimeRepository.kt`)

```
class CrimeRepository private constructor(context: Context) {

     companion object {
        private var INSTANCE: CrimeRepository? = null

         fun initialize(context: Context) {
             if (INSTANCE == null) {
                 INSTANCE = CrimeRepository(context)
             }
        }

         fun get(): CrimeRepository {
            return INSTANCE ?:
            throw IllegalStateException("CrimeRepository must be initialized")
        }
    }
}
```

**`CrimeRepository`** is a *singleton*. This means there will only ever be one instance of it in your app process.

A singleton exists as long as the application stays in memory, so storing any properties on the singleton will keep them available throughout any lifecycle changes in your activities and fragments. Be careful with singleton classes, as they are destroyed when Android removes your application from memory. The **`CrimeRepository`** singleton is not a solution for long-term storage of data. Instead, it gives the app an owner for the crime data and provides a way to easily pass that data between components.

To make **`CrimeRepository`** a singleton, you add two functions to its companion object. One initializes a new instance of the repository, and the other accesses the repository. You also mark the constructor as private to ensure no components can go rogue and create their own instance.

The getter function is not very nice if you have not called **`initialize()`** before it. It will throw an **`IllegalStateException`**, so you need to make sure that you initialize your repository when your application is starting.

To do work as soon as your application is ready, you can create an **`Application`** subclass. This allows you to access lifecycle information about the application itself. Create a class called **`CriminalIntentApplication`** that extends **`Application`**, and override **`Application.onCreate()`** to set up the repository initialization.

### Listing 12.18 Creating an application subclass (`CriminalIntentApplication.kt`)

```
class CriminalIntentApplication : Application() {
    override fun onCreate() {
        super.onCreate()
        CrimeRepository.initialize(this)
    }
}
```

Similar to **`Activity.onCreate(…)`**, **`Application.onCreate()`** is called by the system when your application is first loaded into memory. What makes it different is the fact that your **`CriminalIntentApplication`** is not re-created on configuration changes. It is created when the app launches and destroyed when your app process is destroyed. That makes it a good place to do any kind of one-time initialization operations. The only lifecycle function you will override in CriminalIntent is **`onCreate()`**.

In a moment, you are going to pass the application instance to your repository as a **`Context`** object. This object is valid as long as your application process is in memory, so it is safe to hold a reference to it in the repository class.

But in order for your application class to be used by the system, you need to register it in your manifest. Open `manifests/AndroidManifest.xml` and specify the `android:name` property to set up your application.

### Listing 12.19 Hooking up the application subclass (`manifests/AndroidManifest.xml`)

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
          package="com.bignerdranch.android.criminalintent">

    <application
            android:name=".CriminalIntentApplication"
            android:allowBackup="true"
            ... >
        ...
    </application>

</manifest>
```

With the application class registered in the manifest, the OS will create an instance of **`CriminalIntentApplication`** when launching your app. The OS will then call **`onCreate()`** on the **`CriminalIntentApplication`** instance. Your **`CrimeRepository`** will be initialized, and you can access it from your other components.

Next, add a private property on your **`CrimeRepository`** to store a reference to your database.

### Listing 12.20 Setting up repository properties (`CrimeRepository.kt`)

```
private const val DATABASE_NAME = "crime-database"

class CrimeRepository private constructor(context: Context) {

    private val database: CrimeDatabase = Room
        .databaseBuilder(
            context.applicationContext,
            CrimeDatabase::class.java,
            DATABASE_NAME
        )
        .build()

    companion object {
        ...
    }
}
```

**`Room.databaseBuilder()`** creates a concrete implementation of your abstract **`CrimeDatabase`** using three parameters. It first needs a **`Context`** object, since the database is accessing the filesystem. You pass in the application context because, as discussed above, the singleton will most likely live longer than any of your activity classes.

The second parameter is the database class that you want Room to create. The third is the name of the database file you want Room to create for you. You are using a private string constant defined in the same file, since no other components need to access it.

Next, fill out your **CrimeRepository** so your other components can perform any operations they need to on your database. Add a function to your repository for each function in your DAO.

Listing 12.21 Adding repository functions (CrimeRepository.kt)

```
class CrimeRepository private constructor(context: Context) {

    private val database: CrimeDatabase = Room
        .databaseBuilder(
            context.applicationContext,
            CrimeDatabase::class.java,
            DATABASE_NAME
        )
        .build()

    suspend fun getCrimes(): List<Crime> = database.crimeDao().getCrimes()

    suspend fun getCrime(id: UUID): Crime = database.crimeDao().getCrime(id)

    companion object {
        ...
    }
}
```

Since Room provides the query implementations in the DAO, you call through to those implementations from your repository. This helps keep your repository code short and easy to understand.

This may seem like a lot of work for little gain, since the repository is just calling through to functions on your **CrimeDao**. But fear not; you will be adding functionality soon to encapsulate additional work the repository needs to handle.

## Importing Prepopulated Data

With your repository in place, there is one last step before you can test your query functions. Currently, your database is empty, because you have not added any crimes to it. To speed things up, you will package a prepopulated database with your app that Room can import when your app is launched for the first time. The database file has been provided for you in the solutions file for this chapter (www.bignerdranch.com/android-5e-solutions).

You could programmatically generate and insert dummy data into the database, like the 100 dummy crimes you have been using. However, you have not yet implemented a DAO function to insert new database entries (you will do so in Chapter 15). Packaging a preexisting database file with your application allows you to easily seed the database without altering your app's code unnecessarily.

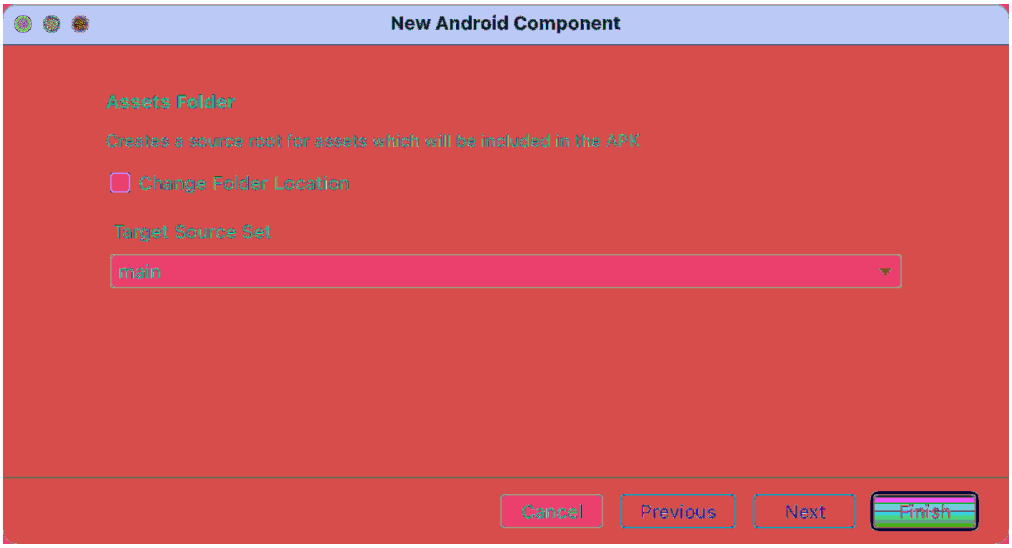
First, you need to add the database file to your project so your code can use it at runtime. Rather than use the resources system for this job, you will use raw *assets*. You can think of assets as stripped-down resources: They are packaged into your APK like resources, but without any of the configuration system tooling that goes on top of resources.

In some ways, that is good. Because there is no configuration system, you can name assets whatever you want and organize them with your own folder structure. But there are some drawbacks. Without a configuration system, you cannot automatically respond to changes in pixel density, language, or orientation, nor can you automatically use the assets in layout files or other resources.

Usually, resources are the better deal. However, in cases where you only access files programmatically, assets can come out ahead. Most games use assets for graphics and sound, for example.

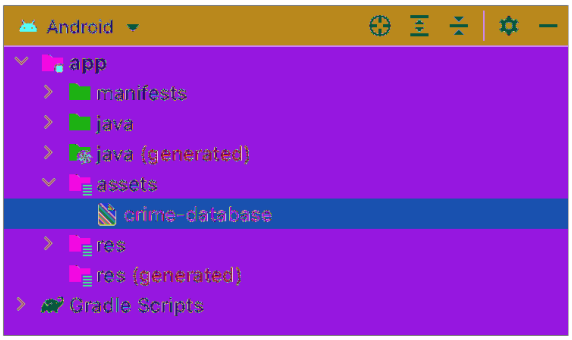
Create an assets folder inside your project by right-clicking your `app` module and selecting New → Folder → Assets Folder. In the dialog that pops up, leave the Change Folder Location checkbox unchecked and leave the Target Source Set set to main (Figure 12.4). Click Finish.

Figure 12.4 Creating the assets folder




Everything in the `assets/` folder will be deployed with your app. Copy or move the database file from the downloaded solutions into the `assets/` folder (Figure 12.5).

Figure 12.5 Imported assets




(Make sure the file is named exactly `crime-database`. There is no file extension.)

Once the database file is properly situated, configuring Room to use it for prepopulation is a snap. In `CrimeRepository.kt`, call **`createFromAsset(databaseFilePath)`** where you initialize your `database` property. Since the file has the same name as the `DATABASE_NAME` constant value, you can reuse it here.

Listing 12.22 Prepopulating your database (`CrimeRepository.kt`)

```
private const val DATABASE_NAME = "crime-database"

class CrimeRepository private constructor(context: Context) {

    private val database: CrimeDatabase = Room
        .databaseBuilder(
            context.applicationContext,
            CrimeDatabase::class.java,
            DATABASE_NAME
        )
        .createFromAsset(DATABASE_NAME)
        .build()
    ...
}
```

## Querying the Database

Now that **`CrimeRepository`** is set up to pull data from a populated database, update **`CrimeListViewModel`** to access the database when loading crimes instead of using the dummy data.

Listing 12.23 Accessing your database (`CrimeListViewModel.kt`)

```
class CrimeListViewModel : ViewModel() {
  private val crimeRepository = CrimeRepository.get()

  val crimes = mutableListOf<Crime>()
  ...
  suspend fun loadCrimes(): List<Crime> {
      val result = mutableListOf<Crime>()
      delay(5000)
      for (i in 0 until 100) {
          val crime = Crime(
              id = UUID.randomUUID(),
              title = "Crime #$i",
              date = Date(),
              isSolved = i % 2 == 0
          )

          result += crime
      }
      return result
      return crimeRepository.getCrimes()
  }
}
```

Run the app and see the prepopulated crimes display onscreen.

## Keeping the Changes Flowing

At this point, your database has been fully set up and connected to your UI. However, your current code is only suited to query the database once. Eventually, you will be able to add and update individual crimes – but at the moment, if other parts of your app tried to update the database, **`CrimeListFragment`** would be oblivious to the changes and would happily present stale data.

While you could add code to manually reconcile updates from specific parts of your app, it would be better to "observe" the database so that **`CrimeListFragment`** automatically receives all updates to the database, regardless of where they come from.

Which brings us back to coroutines, along with two new classes: **`Flow`** and **`StateFlow`**.

Built into the Coroutines library, a *flow* represents an asynchronous stream of data. Throughout their lifetime, flows emit a sequence of values over an indefinite period of time that get sent to a collector. The collector will observe the flow and will be notified every time a new value is emitted in the flow.

Flows are a great tool for observing changes to a database. In a moment, you will create a flow that contains all the **`Crime`** objects in your database. If a crime is added, removed, or updated, the flow will automatically emit the updated set of crimes to its collectors, keeping them in sync with the database. This all ties in nicely with the end goal of this chapter: to have **`CrimeListFragment`** display the freshest data from your database.

Refactoring your code to use a **`Flow`** will touch a handful of files in your project:

- your **`CrimeDao`**, to make it emit a flow of crimes
- your **`CrimeRepository`**, to pass that flow of crimes along
- your **`CrimeListViewModel`**, to get rid of **`loadCrimes()`** and instead present that flow of crimes in an efficient way to its consumers
- your **`CrimeListFragment`**, to collect the crimes from the flow and update its UI

You will start making your changes at the database level and work up the layers until you get to your **`CrimeListFragment`**.

Refactoring the database to provide you with a **`Flow`** of crimes is relatively straightforward. Room has built-in support to query a database and receive the results in a **`Flow`**.

Since you are not making any changes to the structure of the database, you do not need to make any changes in the **`Crime`** and **`CrimeDatabase`** classes. In **`CrimeDao`**, update **`getCrimes()`** to return a **`Flow<List<Crime>>`** instead of a **`List<Crime>`**. Also, you do not need a coroutine scope to handle a reference to a **`Flow`**, so remove the `suspend` modifier. (You will need a coroutine scope when trying to read from the stream of values within the **`Flow`**, but you will handle that in just a second.)

### Listing 12.24 Creating a **`Flow`** from your database (`CrimeDao.kt`)

```
@Dao
interface CrimeDao {
    @Query("SELECT * FROM crime")
    suspend fun getCrimes(): List<Crime> Flow<List<Crime>>

    @Query("SELECT * FROM crime WHERE id=(:id)")
    suspend fun getCrime(id: UUID): Crime
}
```

Make sure you import the `kotlinx.coroutines.flow` version of **`Flow`**.

Since you access the **`CrimeDatabase`** through the **`CrimeRepository`**, make the same changes there:

### Listing 12.25 Refactoring a level higher (`CrimeRepository.kt`)

```
class CrimeRepository private constructor(context: Context) {
    ...
    suspend fun getCrimes(): List<Crime> Flow<List<Crime>>
            = database.crimeDao().getCrimes()
    ...
}
```

Next, clean up your **`CrimeListViewModel`**. You will no longer be using the **`loadCrimes()`** function, and you can get rid of the logging statements. Also, update the `crimes` property to pass the **`Flow`** along.

### Listing 12.26 Clearing the slate (`CrimeListViewModel.kt`)

```
class CrimeListViewModel : ViewModel() {
    private val crimeRepository = CrimeRepository.get()

    val crimes = mutableListOf<Crime>() crimeRepository.getCrimes()

    init {
        Log.d(TAG, "init starting")
        viewModelScope.launch {
            Log.d(TAG, "coroutine launched")
            crimes += loadCrimes()
            Log.d(TAG, "Loading crimes finished")
        }
    }

    suspend fun loadCrimes(): List<Crime> {
        return crimeRepository.getCrimes()
    }
}
```

You have reached the layer where you display UI. To access the values within the **`Flow`**, you must observe it using the **`collect {}`** function.

**`collect {}`** is a suspending function, so you need to call it within a coroutine scope. Thankfully, you already set up a coroutine scope within **`CrimeListFragment`**'s **`onViewCreated()`** callback.

In that callback, replace your call to **`loadCrimes()`** (whose definition you just deleted) with a **`collect {}`** function call on the `crimes` property from **`CrimeListViewModel`**. The lambda you pass into the **`collect {}`** function will be invoked every time there is a new value in the **`Flow`**, so that is the perfect place to set the adapter on your **`RecyclerView`**.

### Listing 12.27 Collecting your **`StateFlow`** from **`CrimeListFragment`** (`CrimeListFragment.kt`)

```
class CrimeListFragment : Fragment() {
    ...
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        viewLifecycleOwner.lifecycleScope.launch {
            viewLifecycleOwner.repeatOnLifecycle(Lifecycle.State.STARTED) {
                val crimes = crimeListViewModel.loadCrimes()
                crimeListViewModel.crimes.collect { crimes ->
                    binding.crimeRecyclerView.adapter =
                        CrimeListAdapter(crimes)
                }
            }
        }
    }
    ...
}
```

(Make sure you import `kotlinx.coroutines.flow.collect`.)

Compile and run the app. Once again, you should see the prepopulated crimes from the database. However, you are not done. If you rotate your device and initiate a configuration change, you might notice a brief moment when the screen is blank, waiting to load the crimes from the database. You are still performing a new database query on each configuration change.

Earlier in this chapter, we talked about how you can use a **`ViewModel`** to perform expensive calculations and cache results across configuration changes. Every time you collect values from the `crimes` property within your **`CrimeListFragment`**, you are creating a new **`Flow`** and performing a new database query for that **`Flow`**. That is an inefficient use of resources, and if your database query takes a long time to execute, your users will see a blank screen while the data is being loaded.

It would be better to maintain a single stream of data from your database and cache the results so they can quickly be displayed to the user. And that is where **`StateFlow`** comes in.

**`StateFlow`** is a specialized version of **`Flow`** that is designed specifically to share application state. **`StateFlow`** always has a value that observers can collect from its stream. It starts with an initial value and caches the latest value that was emitted into the stream. It is the perfect companion to the **`ViewModel`** class, because a **`StateFlow`** will always have a value to provide to fragments and activities as they get re-created.

The first step in setting up a **`StateFlow`** is to create an instance of a **`MutableStateFlow`**. Analogous to **`List`** and **`MutableList`**, **`StateFlow`** is a read-only **`Flow`** while **`MutableStateFlow`** allows you to update the value within the stream. When creating a **`MutableStateFlow`**, you must provide an initial value, so in this situation you will provide an empty list. This is the value that collectors will receive before any other values are put in the stream.

Using a `viewModelScope` in the `init` block of your **`CrimeListViewModel`**, you can collect values from your **`CrimeRepository`**. Once you have your value from the database **`Flow`**, you can set the value on your **`MutableStateFlow`**.

To keep your code maintainable and the stream of data flowing in one direction from the database all the way to the UI, you need to be careful about how you provide your data to consumers. If you provide your data in the form of a **`MutableStateFlow`**, then you are giving the fragments and activities that collect from it the ability to put values directly into the stream. Normally, you want to protect access to the stream, so it is a common practice to keep your **`MutableStateFlow`** private to the class and only expose it to collectors as a read-only **`StateFlow`**.

Add the following code to implement your **`StateFlow`** in `CrimeListViewModel.kt`:

### Listing 12.28 Efficiently caching the database results (`CrimeListViewModel.kt`)

```
class CrimeListViewModel : ViewModel() {
    private val crimeRepository = CrimeRepository.get()

    val crimes = crimeRepository.getCrimes()
    private val _crimes: MutableStateFlow<List<Crime>> = MutableStateFlow(emptyList())
    val crimes: StateFlow<List<Crime>>
        get() = _crimes.asStateFlow()

    init {
        viewModelScope.launch {
            crimeRepository.getCrimes().collect {
                _crimes.value = it
            }
        }
    }
}
```

Run the app again to make sure everything works as expected. Now, you efficiently access the data within your database, and your UI will always display the latest data. In the next chapter, you will connect the crime list and crime detail screens and populate the crime detail screen with data for the crime you click in the database.

## Challenge: Addressing the Schema Warning

If you look through the logs in the build window, you will find a warning about your app not providing a schema export directory:

```
warning: Schema export directory is not provided to the annotation processor
so we cannot export the schema. You can either provide `room.schemaLocation`
annotation processor argument OR set exportSchema to false.
```

(If the build window does not open automatically when you run the app, you can open it with the Build tab at the bottom of the Android Studio window.)

A database *schema* represents the structure of the database, including what tables are in the database, what columns are in those tables, and any constraints on and relationships between those tables. Room supports exporting your database schema into a file so you can store it in a source control. Exporting your schema is often useful so that you have a versioned history of your database.

The warning you see means that you are not providing a file location where Room can save your database schema. You can either provide a schema location to the `@Database` annotation, or you can disable the export to remove the warning. For this challenge, resolve the schema warning by choosing one of these options.

To provide a location for the export, you provide a path for the annotation processor's `room.schemaLocation` property. To do this, add the following `javaCompileOptions{}` block to your `app/build.gradle` file:

```
...
android {
    ...
    defaultConfig {
        ...
        testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"

        javaCompileOptions {
            annotationProcessorOptions {
                arguments += [
                    "room.schemaLocation": "$projectDir/schemas".toString(),
                ]
            }
        }
    }
}
...
```

To disable the export, set `exportSchema` to `false`:

```
@Database(entities = [Crime::class], version=1, exportSchema = false)
@TypeConverters(CrimeTypeConverters::class)
abstract class CrimeDatabase : RoomDatabase() {
    abstract fun crimeDao(): CrimeDao
}
```

## For the More Curious: Singletons

The singleton pattern, as used in the **`CrimeRepository`**, is very common in Android. But singletons get a bad rap because they can be misused in a way that makes an app hard to maintain.

Singletons are often used in Android because they outlive a single fragment or activity. A singleton will still exist across rotation and as you move between activities and fragments in your application.

Singletons also make a convenient owner for your model objects. Imagine a more complex CriminalIntent application with many activities and fragments modifying crimes. When one component modifies a crime, how would you make sure that updated crime was sent over to the other components?

If the **`CrimeRepository`** is the owner of crimes and all modifications to crimes pass through it, propagating changes is much easier. As you transition between components, you can pass the crime ID as an identifier for a particular crime and have each component pull the full crime object from the **`CrimeRepository`** using that ID.

However, singletons do have a few downsides. For example, while they allow for an easy place to stash data with a longer lifetime than other components, singletons do have a lifetime. Singletons will be destroyed, along with all their instance variables, as Android reclaims memory at some point after you switch out of an application. Singletons are not a long-term storage solution. (Writing the files to disk or sending them to a web server is.)

Singletons can also make your code hard to unit test. For example, there is not a great way to replace the **`CrimeRepository`** instance with a mock version of itself. In practice, Android developers usually solve this problem using a tool called a *dependency injector*. This tool allows for objects to be shared as singletons while still making it possible to replace them when needed. To learn more about dependency injection, read the section called For the More Curious: Managing Dependencies in Chapter 20.

And, as we said, singletons have the potential to be misused. It might be tempting to use singletons for everything, because they are convenient – you can get to them wherever you are, and you can store in them whatever information you need to get at later. But when you do that, you are avoiding answering important questions: Where is this data used? Where is this function important?

A singleton does not answer those questions. So whoever comes after you will open up your singleton and find something that looks like somebody's disorganized junk drawer. Batteries, zip ties, old photographs? What is all this here for? Make sure that anything in your singleton is truly global and has a strong reason for being there.

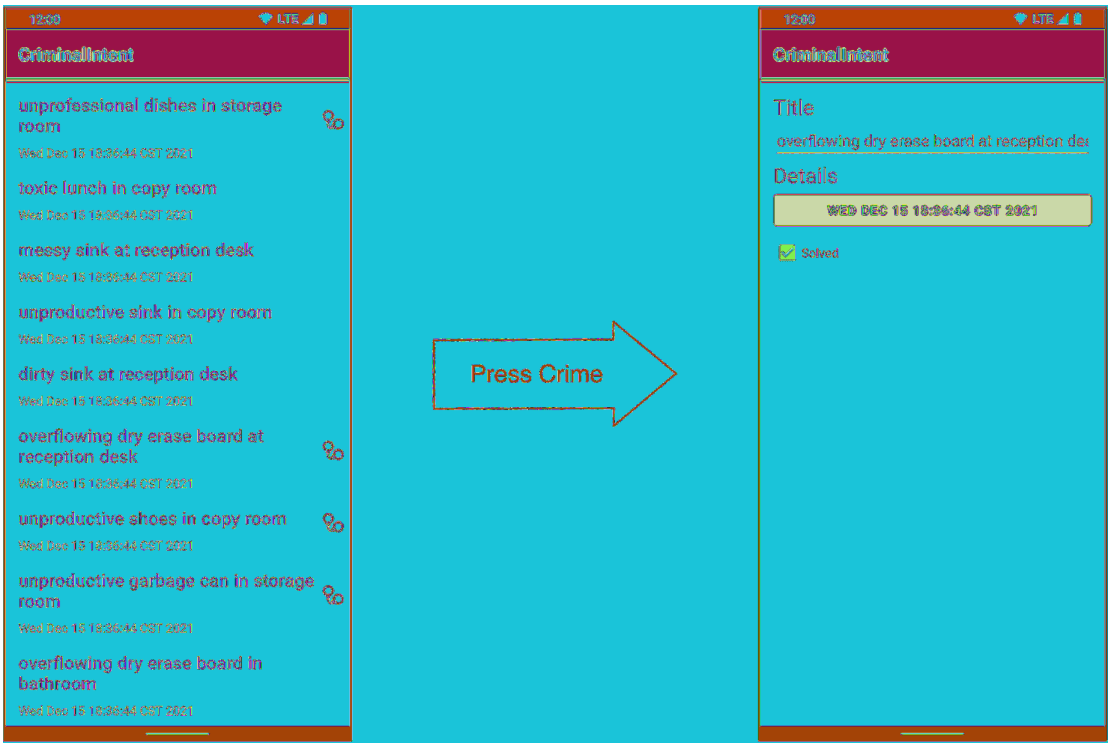
On balance, singletons are a key component of a well-architected Android app – when used correctly.

# 13
# Fragment Navigation

In this chapter, you will get the list and the detail parts of CriminalIntent working together. Using the Navigation Jetpack library, you will define the screens your users will be able to see and how your users can move between them.

When a user presses an item in the list of crimes, the Navigation library will swap out **`CrimeListFragment`** with a new instance of **`CrimeDetailFragment`** displaying the details for the crime that was pressed (Figure 13.1).

Figure 13.1 Swapping **`CrimeListFragment`** for **`CrimeDetailFragment`**




To get this working, you will learn how to implement navigation using the Jetpack Navigation library. You will also learn how to pass data to a fragment instance using the Safe Args Gradle plugin. Finally, you will learn how to use the unidirectional data flow architecture to manage and mutate state in response to UI changes.

## Performing Navigation

Few apps are composed of a single screen. As apps add new features, developers create new screens to house those features. And managing how users navigate through an individual app is a difficult task. There might be many paths a user could take as they navigate deeper into your app.

The Navigation component in the Jetpack libraries helps you define screens and paths between them and then gives you the tools to perform that navigation. At its core, the library relies on a *navigation graph*, which defines a group of screen destinations as well as the paths between destinations. The navigation graph is contained in an XML file, and Android Studio provides a handy graphical tool for editing it.

At the end of this chapter, you will have a navigation graph including your two screens, **`CrimeListFragment`** and **`CrimeDetailFragment`**, as well as a single path defining the navigation from the list screen to the detail screen. In Chapter 15, you will add a destination for a dialog – a modal pop-up – and that will complete your navigation graph for CriminalIntent.

Historically, navigation in Android apps was done using **`FragmentTransaction`**s, which we briefly mentioned back in Chapter 9. That API, while powerful and expressive, is difficult to use and prone to errors. Under the hood, the Navigation library still uses those APIs to perform navigation, but it provides that functionality in a safer and easier-to-use form.

In GeoQuiz, you had one activity (**`MainActivity`**) start another activity (**`CheatActivity`**). In CriminalIntent, you are instead going to use a *single activity architecture*. An app that uses single activity architecture has one activity and multiple fragments. Each fragment acts as its own screen. The activity is solely used as a container for whatever fragment is currently being displayed.

By keeping everything in the same activity, you ensure that your app is in control of everything being rendered onscreen. If you use multiple activities, the system will take control of navigation and animations – sometimes adding behaviors that you do not want, without options to effectively customize those behaviors. When you use the single activity architecture, you maintain more control and flexibility over how your app behaves.

The Navigation library is agnostic to the type of destination used in your app – activities or fragments. Google provides first-party support for fragments, but you can define other destinations if you want to.

## Implementing the Navigation component library

As usual, the first thing you need to do to use the Navigation library is to include it in your Gradle dependencies. You will need to include two separate modules: one that handles the core functionality (`navigation-ui-ktx`) and one that enables support for fragments (`navigation-fragment-ktx`). Open the `app/build.gradle` file (the one labeled (Module: CriminalIntent.app)) add the dependencies:

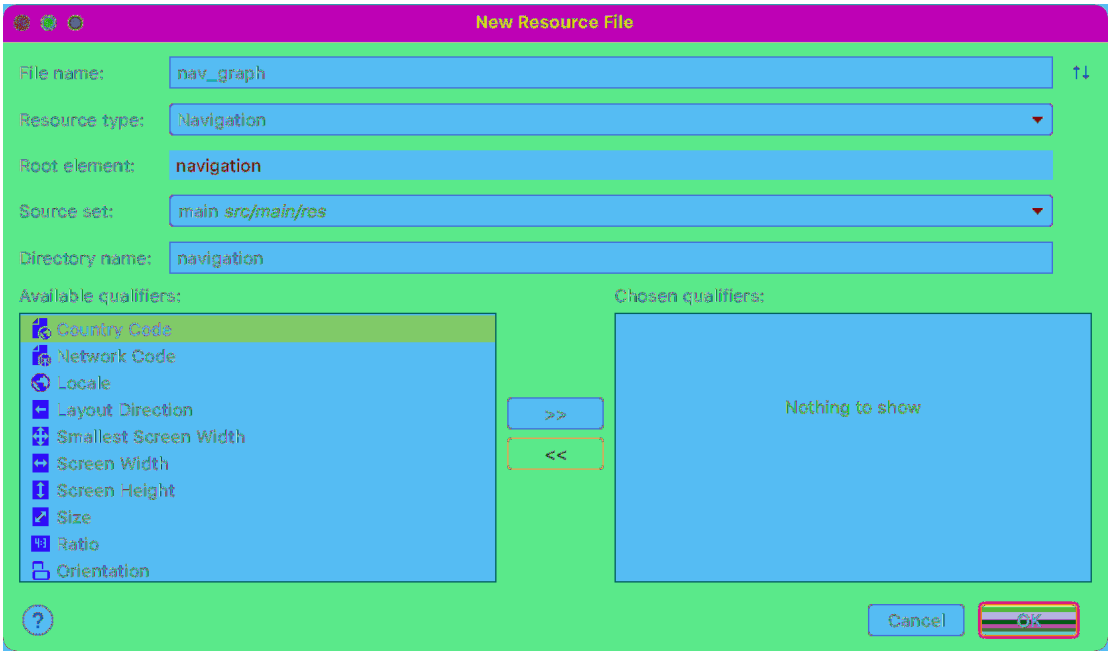
### Listing 13.1 Adding the Navigation dependencies (`app/build.gradle`)

```
dependencies {
    ...
    kapt 'androidx.room:room-compiler:2.4.2'
    implementation "androidx.navigation:navigation-fragment-ktx:2.4.1"
    implementation "androidx.navigation:navigation-ui-ktx:2.4.1"
    testImplementation 'junit:junit:4.13.2'
}
```

Do not forget to click the Sync Project with Gradle Files button or the Sync Now button after you have made these changes.

Once Gradle finishes syncing, you will create the file that will house your navigation graph. In the project tool window, right-click the res directory and choose New → Android Resource file. Name the file nav_graph, set the Resource type to Navigation, and click OK (Figure 13.2).

### Figure 13.2 Creating your navigation graph



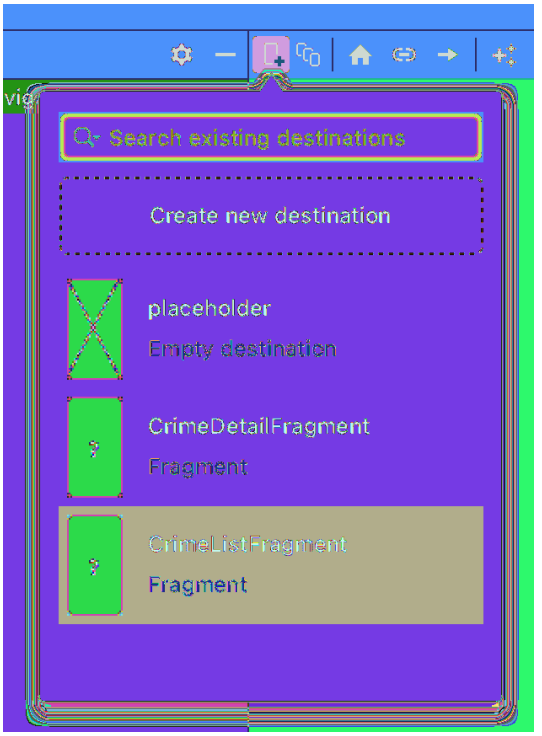
Android Studio opens your new `nav_graph.xml` in the editor. Like the layout editor, this graphical editor has tabs in the top-right corner for the three view options: Code, Split, and Design. Make sure the design view is selected (Figure 13.3).

Figure 13.3 An empty navigation graph




Currently, this navigation graph is empty – it has no screens that it can present to your user or navigate between. To make your navigation graph more useful, you need to add a destination to the graph, which will define a screen that can be presented to your users. As the text in the middle of the editor indicates, click the Add Destination icon located in the top-left corner of the editor. In the pop-up, select **`CrimeListFragment`** from the list of possible destinations (Figure 13.4).

Figure 13.4 Adding your first destination




You have just added a destination to your navigation graph. And because it is the first destination you added, it will also be the first page that your user sees – once you connect this navigation graph to your application, which you will do in a moment.

In the navigation graph, the destination is labeled crimeListFragment, but the screen it depicts just says Preview Unavailable. It would be better if it showed a small preview of the screen, so other developers looking at your navigation graph could quickly understand how users move through your app. You can make this happen with a couple of changes in the XML.

Switch over to the code view by clicking the Code tab in the top right and add a `tools:layout` attribute to your **`CrimeListFragment`** destination:

Listing 13.2 Enabling previews (`nav_graph.xml`)

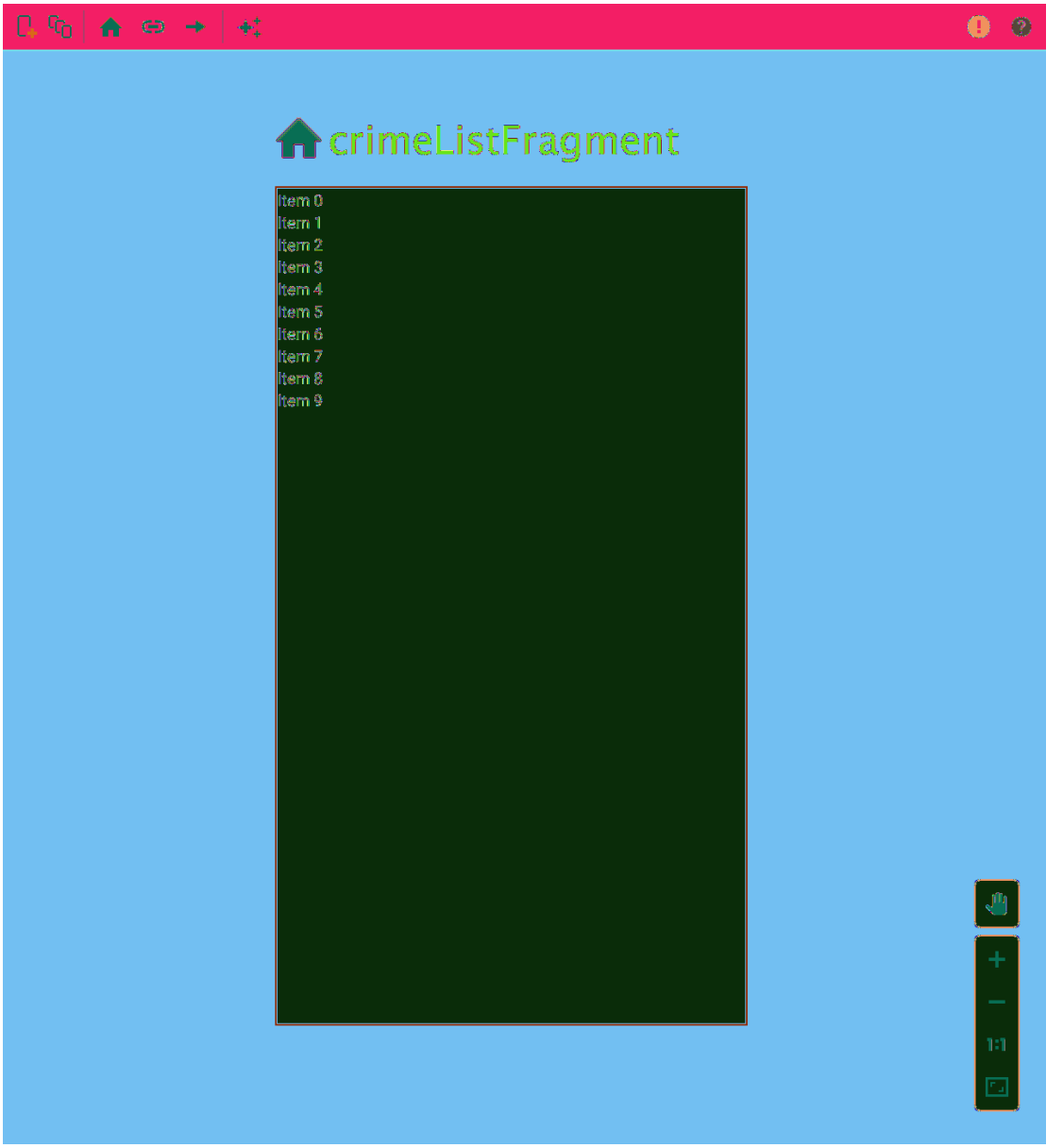
```
<?xml version="1.0" encoding="utf-8"?>
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/nav_graph"
    app:startDestination="@id/crimeListFragment">

    <fragment
        android:id="@+id/crimeListFragment"
        android:name="com.bignerdranch.android.criminalintent.CrimeListFragment"
        android:label="CrimeListFragment"
        tools:layout="@layout/fragment_crime_list" />
</navigation>
```

In Chapter 2, you used the `tools` namespace to improve the layout preview for GeoQuiz's **`MainActivity`**. You use it here to improve the appearance of your navigation graph in the design view. By using the `tools:layout` attribute on a navigation destination, you can provide a preview of how that destination will look to users by referencing the XML layout.

Take a peek at the design view to see your improved destination preview (Figure 13.5).

Figure 13.5 A prettier preview



Switch back to the code view.

Notice that in the root `<navigation>` element, you have an ID for the entire navigation graph and a starting destination. You will use the ID to reference this navigation graph in your layout.

The starting destination defines the screen that will first appear when you start your activity. In an inner element, **`CrimeListFragment`** is also defined as a possible destination. The ID of the starting destination matches **`CrimeListFragment`**'s ID, so **`CrimeListFragment`** will appear when **`MainActivity`** starts.

The navigation graph on its own defines the screens your app can navigate between. To make use of these definitions, you must also connect the graph to your UI. You can accomplish this by defining a container in your **`Activity`** to host your navigation graph. The container is responsible for swapping out fragments as the user navigates between different screens.

The easiest way to define a container is using a **`NavHostFragment`** inside a **`FragmentContainerView`**. You are already using a **`FragmentContainerView`** in the `activity_main.xml` layout to host your **`CrimeListFragment`**. Add the following code to inflate a **`NavHostFragment`** in your **`MainActivity`** and set it up to load up your navigation graph.

### Listing 13.3 Hooking up the **`NavHostFragment`** (`activity_main.xml`)

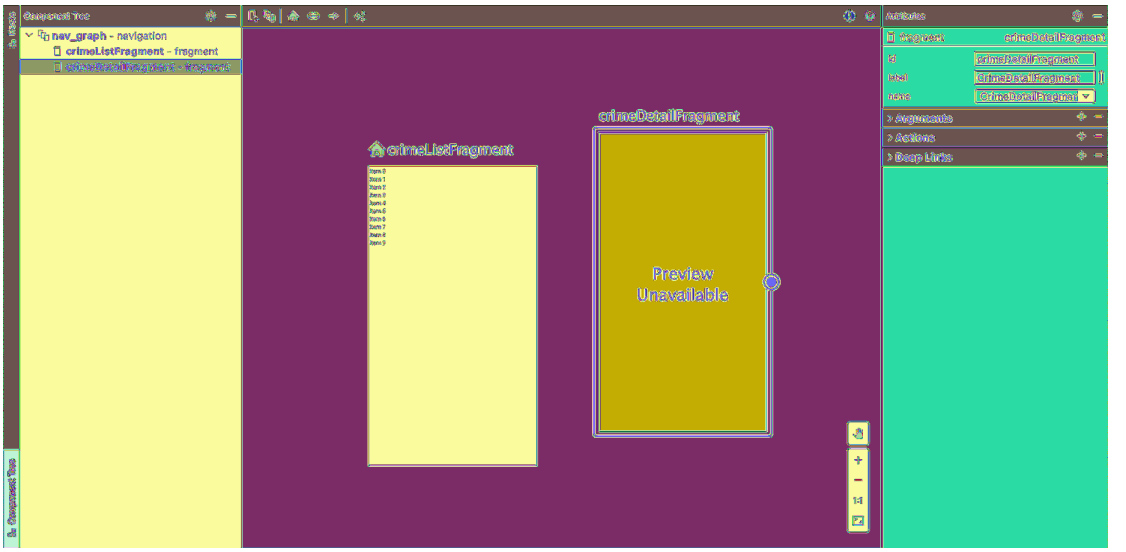
```
<androidx.fragment.app.FragmentContainerView
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/fragment_container"
    ~~android:name="com.bignerdranch.android.criminalintent.CrimeListFragment"~~
    android:name="androidx.navigation.fragment.NavHostFragment"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:defaultNavHost="true"
    app:navGraph="@navigation/nav_graph"
    tools:context=".MainActivity" />
```

With that, you have done all the work needed to display your starting destination onscreen. Run your app and confirm that **`CrimeListFragment`** appears onscreen, as before.

## Navigating to the detail screen

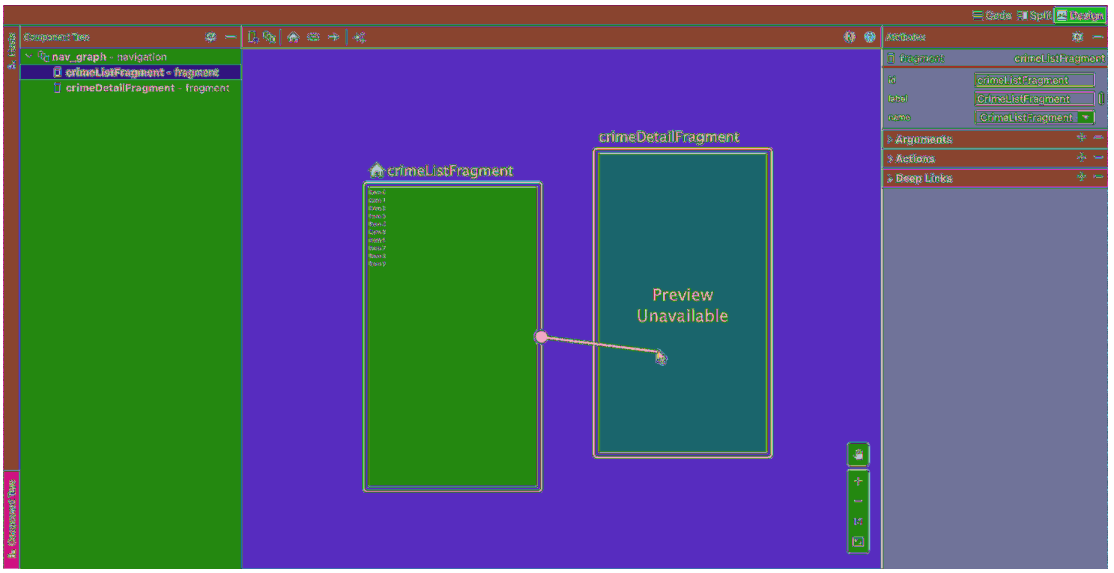
With your navigation foundation set up, you can now add a second destination to your navigation graph. Return to `nav_graph.xml`'s design view. Click the Add Destination button again, this time selecting **`CrimeDetailFragment`**. You will see **`CrimeDetailFragment`** added to the canvas as a second destination (Figure 13.6).

Figure 13.6 Two destinations



You will set up the preview for your new destination in a moment. To enable navigation between the two screens, you need to define an action that specifies the screen you start from and the screen you end at when navigating. With the crimeListFragment destination selected, click and drag from the circle on its right edge to the crimeDetailFragment destination. The action you are defining is shown first as a line (Figure 13.7), then, when you release the mouse, as an arrow between the two destinations.

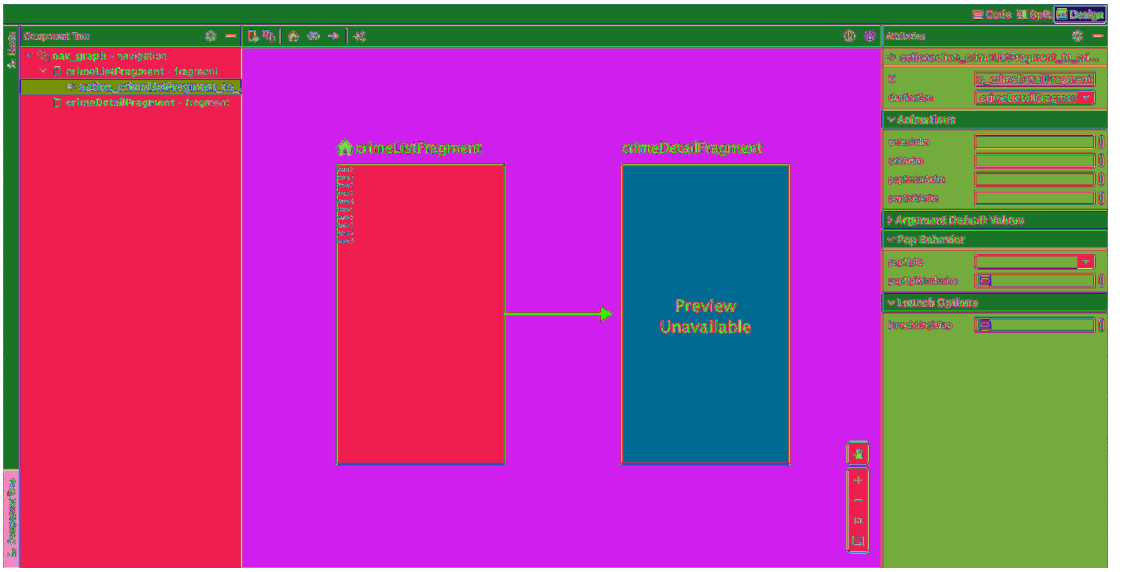
Figure 13.7 Connecting the two destinations




In large projects, as developers add more and more destinations in a navigation graph, it can be difficult to organize them on the design preview's canvas so that the path between destinations is clear and visible. The positioning of screens in the design preview is entirely up to you and does not affect how navigation in your application behaves.

You can manually rearrange your destinations into an orderly presentation, but that can be time consuming and tedious. Thankfully, Android Studio provides functionality to rearrange the items on the canvas and put each destination into a reasonable spot. Click the Auto Arrange button (to the right of the Add Destination button). With that, Android Studio will arrange your two destinations neatly in the preview (Figure 13.8).

Figure 13.8 Your two destinations, looking good




Switch over to the code view. You will see your new **`CrimeDetailFragment`** destination added as an element. You will also see an action defined in the **`CrimeListFragment`** destination (shown shaded in Listing 13.4). Since the action is within the **`CrimeListFragment`** destination, it starts at **`CrimeListFragment`**. Within the action, you can see that it defines the ID associated with the **`CrimeDetailFragment`** destination as its destination.

Add the `tools:layout` attribute to your new destination to enable its preview.

Listing 13.4 Enabling the detail view's preview (`nav_graph.xml`)

```
<?xml version="1.0" encoding="utf-8"?>
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/nav_graph"
    app:startDestination="@id/crimeListFragment">

    <fragment
        android:id="@+id/crimeListFragment"
        android:name="com.bignerdranch.android.criminalintent.CrimeListFragment"
        android:label="CrimeListFragment"
        tools:layout="@layout/fragment_crime_list" >
        <action
            android:id="@+id/action_crimeListFragment_to_crimeDetailFragment"
            app:destination="@id/crimeDetailFragment" />
    </fragment>
    <fragment
        android:id="@+id/crimeDetailFragment"
        android:name="com.bignerdranch.android.criminalintent.CrimeDetailFragment"
        android:label="CrimeDetailFragment"
        tools:layout="@layout/fragment_crime_detail" />
</navigation>
```

When performing a navigation action in your Kotlin code, you reference the ID for the action. The ID that was automatically provided for your action, `action_crimeListFragment_to_crimeDetailFragment`, is accurate, but it is also a little verbose. Rename it `show_crime_detail`, which is just as clear but much more concise.

### Listing 13.5 Renaming the action (`nav_graph.xml`)

```
<?xml version="1.0" encoding="utf-8"?>
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/nav_graph"
    app:startDestination="@id/crimeListFragment">

    <fragment
        android:id="@+id/crimeListFragment"
        android:name="com.bignerdranch.android.criminalintent.CrimeListFragment"
        android:label="CrimeListFragment"
        tools:layout="@layout/fragment_crime_list" >
        <action
            ~~android:id="@+id/action_crimeListFragment_to_crimeDetailFragment"~~
            android:id="@+id/show_crime_detail"
            app:destination="@id/crimeDetailFragment" />
    </fragment>
    <fragment
        android:id="@+id/crimeDetailFragment"
        android:name="com.bignerdranch.android.criminalintent.CrimeDetailFragment"
        android:label="CrimeDetailFragment"
        tools:layout="@layout/fragment_crime_detail" />
</navigation>
```

Before you can perform the navigation from your Kotlin code, you need to do some refactoring. The goal, remember, is that when the user presses an item in the list of crimes, they will navigate to the detail screen for that crime. You may recall that you set an **`OnClickListener`** on the root view for the **`CrimeHolder`** back in Chapter 10. But that **`OnClickListener`** does not do much. Right now, it just prints the title of the crime that was pressed in a toast message.

You could swap out the toast printing code for some code that navigates the user to the detail screen. However, that would tightly couple **`CrimeHolder`** and **`CrimeListAdapter`** to being used in **`CrimeListFragment`**. That is not a good approach for building a maintainable codebase.

A better approach would be to pass a lambda expression into the **`CrimeHolder`** and **`CrimeListAdapter`** classes to allow whatever class creates instances of those classes to configure what happens when the user presses a list item. That is the approach you will take here.

First, pass a lambda expression named `onCrimeClicked` into the **`bind`** function in **`CrimeHolder`**. This will be the lambda that is invoked when the user presses the root view for that particular **`CrimeHolder`**.

### Listing 13.6 Passing in a lambda expression (`CrimeListAdapter.kt`)

```
class CrimeHolder(
    private val binding: ListItemCrimeBinding
) : RecyclerView.ViewHolder(binding.root) {
    fun bind(crime: Crime, onCrimeClicked: () -> Unit) {
        binding.crimeTitle.text = crime.title
        binding.crimeDate.text = crime.date.toString()

        binding.root.setOnClickListener {
            Toast.makeText(
                binding.root.context,
                "${crime.title} clicked!",
                Toast.LENGTH_SHORT
            ).show()
            onCrimeClicked()
        }

        binding.crimeSolved.visibility = if (crime.isSolved) {
            View.VISIBLE
        } else {
            View.GONE
        }
    }
}
...
```

Now, in **`CrimeListAdapter`**, include that same lambda as a constructor parameter and pass it along to the **`bind`** function on the **`CrimeHolder`** class.

### Listing 13.7 Hooking up the adapter (`CrimeListAdapter.kt`)

```
class CrimeListAdapter(
    private val crimes: List<Crime>,
    private val onCrimeClicked: () -> Unit
) : RecyclerView.Adapter<CrimeHolder>() {
    override fun onCreateViewHolder(
        parent: ViewGroup,
        viewType: Int
    ): CrimeHolder {
        val inflater = LayoutInflater.from(parent.context)
        val binding = ListItemCrimeBinding.inflate(inflater, parent, false)
        return CrimeHolder(binding)
    }

    override fun onBindViewHolder(holder: CrimeHolder, position: Int) {
        val crime = crimes[position]
        holder.bind(crime, onCrimeClicked)
    }

    override fun getItemCount() = crimes.size
}
```

With the new and improved **`CrimeListAdapter`**, you can finally perform the navigation from your **`CrimeListFragment`**. Using the Navigation library, you perform navigations through a class called **`NavController`**. With this class, you can navigate to new screens, implement the Back button to return to previous screens, and much more.

You do not have to create your own instance of this class. With the **`NavHostFragment`** specified in your `activity_main.xml` layout file, you already have access to an instance. All you need to do is find it.

The way you do *that* is through the **`findNavController`** extension function. This function will search the view hierarchy and fragment for the **`NavController`** and return it to you. Because navigation is a crucial part of your application, **`findNavController`** is available for several components in the Android framework, including both activities and fragments.

Once you get the **`NavController`**, you call the **`navigate`** function on it, passing in a resource ID for either a destination or a navigation action. Here, you are going to use the `R.id.show_crime_detail` resource ID that you just defined for the action of navigating from the list to the detail view. Do this where you bind the **`CrimeListAdapter`**, using the trailing lambda syntax for the **`CrimeListAdapter`** constructor.

### Listing 13.8 Performing the navigation (`CrimeListFragment.kt`)

```
class CrimeListFragment : Fragment() {
    ...
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        viewLifecycleOwner.lifecycleScope.launch {
            viewLifecycleOwner.repeatOnLifecycle(Lifecycle.State.STARTED) {
                crimeListViewModel.crimes.collect { crimes ->
                    binding.crimeRecyclerView.adapter =
                        CrimeListAdapter(crimes) {
                            findNavController().navigate(
                                R.id.show_crime_detail
                            )
                        }
                }
            }
        }
    }
    ...
}
```

Be sure to import the fragment version of **`findNavController()`**.

Run your app and press or click a crime. Presto! It will navigate you to an empty detail screen. Press the Back button, and you will return to the list screen.

## Passing data to a fragment

So far, so good. But you do not want pressing a crime to take you to a blank crime page. You want a **`CrimeDetailFragment`** populated with the selected crime's details. The process of passing data to your fragment is much like the process you used back in Chapter 7 to pass data to an activity.

In your current setup, the Navigation library and the framework are responsible for instantiating your fragments, just as your activities are instantiated by the Android OS. The classic approach to passing values to a **`Fragment`** involves using a **`Bundle`** to store key-value pairs for your arguments – much like you saw with the **`Intent`** system. However, this approach falls victim to the same limitations activities pose: You rely on convention and boilerplate code that can cause your app to crash if done incorrectly. And it is all too easy to make a mistake with that boilerplate code or forget to use the conventional function or constant value.

Good news: As a part of the Navigation library, Google has developed a Gradle plugin to help you safely pass data between navigation destinations. Similar to how View Binding and the Room library generate code for their own purposes, the Safe Args plugin generates code for you to package up data when performing navigation and unpackage data once at the destination.

The plugin is included in at the project level, so open the `build.gradle` file labeled (Project: CriminalIntent) and include Safe Args in the list of plugins for the project:

### Listing 13.9 Including the Safe Args plugin in your project (`build.gradle`)

```
plugins {
    id 'com.android.application' version '7.1.2' apply false
    id 'com.android.library' version '7.1.2' apply false
    id 'org.jetbrains.kotlin.android' version '1.6.10' apply false
    id 'org.jetbrains.kotlin.kapt' version '1.6.10' apply false
    id 'androidx.navigation.safeargs.kotlin' version '2.4.1' apply false
}
...
```

Once that is done, open the `app/build.gradle` file and enable the plugin for your application. The Safe Args plugin does not require any additional libraries be added as dependencies.

### Listing 13.10 Adding the Safe Args plugin to your app (`app/build.gradle`)

```
plugins {
    id 'com.android.application'
    id 'org.jetbrains.kotlin.android'
    id 'org.jetbrains.kotlin.kapt'
    id 'androidx.navigation.safeargs'
}
...
```

As always, sync your Gradle files before moving on.

Currently, you are performing navigation by referencing the resource ID of an action that you defined in `nav_graph.xml`. The Safe Args plugin works by generating classes based on the contents of your navigation graph. When navigating, you will use these generated classes instead of using resource IDs.

Direction classes contain all the information needed to perform navigation, including the ID of the action. For fragment destinations, Safe Args names the classes it generates with the fragment's name plus "Directions." So for **`CrimeListFragment`**, the Safe Args plugin generates a class named **`CrimeListFragmentDirections`**.

The Safe Args plugin also generates functions within its destination classes for each possible action within the destination. Since you only have one action for **`CrimeListFragment`**'s destination, the Safe Args plugin only generates one function for you to use. The function name is based on the resource ID you declared for that action, so your usage of `R.id.show_crime_detail` will become a function call to **`CrimeListFragmentDirections.showCrimeDetail()`**.

Make use of the Safe Args plugin in **`CrimeListFragment`** by swapping in the generated **`CrimeListFragmentDirections`** class.

### Listing 13.11 Asking for directions (`CrimeListFragment.kt`)

```
class CrimeListFragment : Fragment() {
    ...
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        viewLifecycleOwner.lifecycleScope.launch {
            viewLifecycleOwner.repeatOnLifecycle(Lifecycle.State.STARTED) {
                crimeListViewModel.crimes.collect { crimes ->
                    binding.crimeRecyclerView.adapter =
                        CrimeListAdapter(crimes) {
                            findNavController().navigate(
                                R.id.show_crime_detail
                                CrimeListFragmentDirections.showCrimeDetail()
                            )
                        }
                }
            }
        }
    }
    ...
}
```

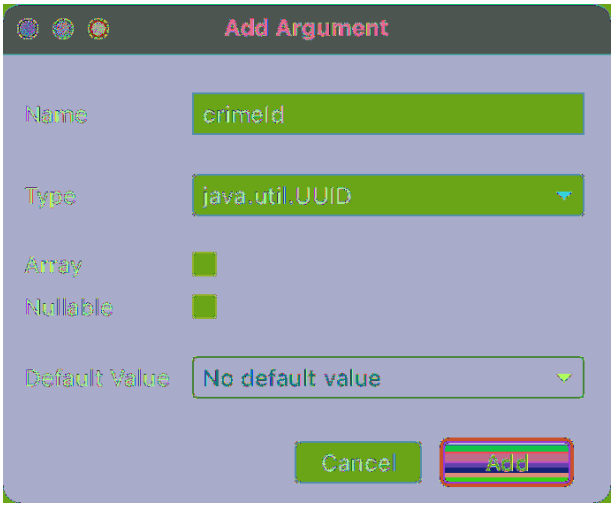
Run the app and select one of the crimes in the list. You will still see it navigate to a blank **`CrimeDetailFragment`**. It does not show the details for the crime you selected because you have not specified which crime it should display.

To wire this up, you need to pass an argument specifying the crime to display. Back in `nav_graph.xml`, view your navigation graph in the design view. Click the crimeDetailFragment. In the attributes window to the right of the editor, click the plus icon next to the Arguments section header. A window will pop up.

Although you could pass in the entire **`Crime`** to your **`CrimeDetailFragment`**, this would add more complexity to your app. Instead, you can pass the ID of the crime and have the **`CrimeDetailFragment`** query the crime information from your database.

In the Add Argument dialog, name your argument crimeId. The **`UUID`** class implements the **`Serializable`** interface, so for the Type select Custom Serializable..., then search for and select UUID in the window that pops up. Leave the rest of the options alone and click Add to add the argument to your navigation graph (Figure 13.9).

Figure 13.9 Adding an argument to **`CrimeDetailFragment`**




Switch over to the code view and note the addition to the **`CrimeDetailFragment`** destination entry:

```
<?xml version="1.0" encoding="utf-8"?>
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/nav_graph"
    app:startDestination="@id/crimeListFragment">

    <fragment
        android:id="@+id/crimeListFragment"
        android:name="com.bignerdranch.android.criminalintent.CrimeListFragment"
        android:label="CrimeListFragment"
        tools:layout="@layout/fragment_crime_list" >
        <action
            android:id="@+id/show_crime_detail"
            app:destination="@id/crimeDetailFragment" />
    </fragment>
    <fragment
        android:id="@+id/crimeDetailFragment"
        android:name="com.bignerdranch.android.criminalintent.CrimeDetailFragment"
        android:label="CrimeDetailFragment"
        tools:layout="@layout/fragment_crime_detail" >
        <argument
            android:name="crimeId"
            app:argType="java.util.UUID" />
    </fragment>
</navigation>
```

Go back to `CrimeListFragment.kt`. You will see that you have an error where you are trying to perform navigation. The **`showCrimeDetails()`** function now expects a **`UUID`** as a parameter. Before you can fix this error, you need to access the ID of the crime that was pressed.

A crime's ID is known when setting the **`View.OnClickListener`** for the **`CrimeHolder`** root view. What you need to do is pass that ID back up to the **`CrimeListFragment`**.

You are already passing a lambda expression into the **`bind`** function for **`CrimeHolder`**. Since you also have access to the crime within that function, update the `onCrimeClicked` parameter to accept a **`UUID`** argument. When invoking that **`onCrimeClicked`** lambda expression, pass in the ID for the crime. Finally, update the lambda expression passed into **`CrimeListAdapter`**.

### Listing 13.12 Passing the ID back from the adapter (`CrimeListAdapter`)

```
class CrimeHolder(
    private val binding: ListItemCrimeBinding
) : RecyclerView.ViewHolder(binding.root) {
    fun bind(crime: Crime, onCrimeClicked: (crimeId: UUID) -> Unit) {
        binding.crimeTitle.text = crime.title
        binding.crimeDate.text = crime.date.toString()

        binding.root.setOnClickListener {
            onCrimeClicked(crime.id)
        }

        binding.crimeSolved.visibility = if (crime.isSolved) {
            View.VISIBLE
        } else {
            View.GONE
        }
    }
}

class CrimeListAdapter(
    private val crimes: List<Crime>,
    private val onCrimeClicked: (crimeId: UUID) -> Unit
) : RecyclerView.Adapter<CrimeHolder>() {
    ...
}
```

Now you have access to the crime's ID back in **`CrimeListFragment`**. Using that, along with the **`CrimeListFragmentDirections`** class generated by the Safe Args plugin, pass the crime's ID along while performing navigation.

### Listing 13.13 Performing the navigation (`CrimeListFragment.kt`)

```
class CrimeListFragment : Fragment() {
    ...
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        viewLifecycleOwner.lifecycleScope.launch {
            viewLifecycleOwner.repeatOnLifecycle(Lifecycle.State.STARTED) {
                crimeListViewModel.crimes.collect { crimes ->
                    binding.crimeRecyclerView.adapter =
                        CrimeListAdapter(crimes) { crimeId ->
                            findNavController().navigate(
                                CrimeListFragmentDirections.showCrimeDetail(crimeId)
                            )
                        }
                }
            }
        }
    }
    ...
}
```

The Safe Args plugin not only generates code to perform type-safe navigation but also allows you to safely access navigation arguments once the user is at their destination. By using the `navArgs` property delegate, you can access the navigation arguments for a particular destination in a type-safe manner. The Safe Args plugin generates classes that hold all the arguments for a destination, naming them with the name of the destination plus "Args." So the navigation arguments for the **`CrimeDetailFragment`** class are accessed using the **`CrimeDetailFragmentArgs`** class.

In **CrimeDetailFragment**, create a class property called `args` using the `navArgs` property delegate. In a little bit, you will use the crime ID to implement the behavior you want in **CrimeDetailFragment**, but you have some additional work to do before you can wrap that up. For now, just log the crime ID to confirm that it is being passed along correctly.

### Listing 13.14 Accessing the arguments in **CrimeDetailFragment** (`CrimeDetailFragment.kt`)

```
private const val TAG = "CrimeDetailFragment"

class CrimeDetailFragment : Fragment() {
    ...
    private lateinit var crime: Crime

    private val args: CrimeDetailFragmentArgs by navArgs()

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        crime = Crime(
            id = UUID.randomUUID(),
            title = "",
            date = Date(),
            isSolved = false
        )

        Log.d(TAG, "The crime ID is: ${args.crimeId}")
    }
    ...
}
```

Run the app, select a crime from the list, and look at Logcat. You will see the crime's ID logged by **CrimeDetailFragment**:

```
    D/CrimeDetailFragment: The crime ID is: 4f916c0c-faa1-486b-b9a9-0d55922fd2e1
```

With that, you have done all the setup you need for navigation. You are moving between **CrimeListFragment** and **CrimeDetailFragment**, passing along the relevant ID for the crime you want to display. Now, you need to get that crime from the database and let your users modify it.

## Unidirectional Data Flow

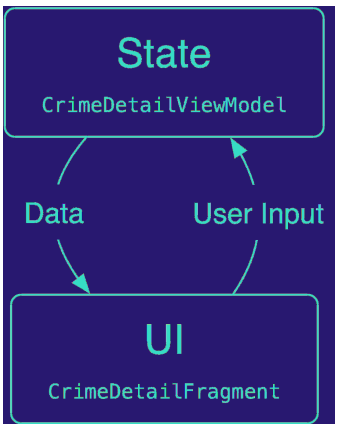
Applications must respond to input from multiple sources: data being loaded from the back end as well as inputs from the user. If you do not have a plan in place to combine these sources, you could code yourself into a mess of complex logic that is difficult to maintain.

*Unidirectional data flow* is an architecture pattern that has risen to prominence and that plays nicely with the reactive patterns you have been using with **`Flow`** and **`StateFlow`**. Unidirectional data flow tries to simplify application architecture by encapsulating these two forces – data from the back end and input from the user – and clarifying their responsibilities.

Data comes from a variety of sources, such as the network, a database, or a local file. It is often generated as part of a transformation of application state, such as the user's authentication state or the contents of their shopping cart. These sources of state send the data down to the UI, where the UI can render it.

Once data is displayed as UI, the user can interact with it through various forms of input. The user can check boxes, press buttons, enter text – and all that input is sent back up to those sources of state, mutating them in response to the user's actions. These two streams travel in opposite directions, forming a circular stream of information (Figure 13.10).

Figure 13.10 Unidirectional data flow



You are going to implement the business logic in **`CrimeDetailFragment`** using the unidirectional data flow pattern. The source of state for **`CrimeDetailFragment`** will be a **`ViewModel`**. It will hold a reference to a **`StateFlow`**, which will hold the latest version of the particular crime the user is viewing. The **`CrimeDetailFragment`** will observe that **`StateFlow`**, updating its UI whenever the crime updates.

As the user edits the details of the current crime, the **`CrimeDetailFragment`** will send that user input up to its **`ViewModel`**. After updating the crime's data, the **`ViewModel`** will send the updated crime back to the **`CrimeDetailFragment`**. Looping and looping, the state and the UI will always remain in sync.

Before adding new code to implement this pattern, clear the decks by deleting some code you no longer need. Delete the plain, boring `crime` class property in **`CrimeDetailFragment`**, as well as any lines of code that reference it. Also, delete the **`onCreate`** code.

### Listing 13.15 Deleting references to the old crime (`CrimeDetailFragment.kt`)

```
~~private const val TAG = "CrimeDetailFragment"~~

class CrimeDetailFragment : Fragment() {
    ...
    ~~private lateinit var crime: Crime~~

    private val args: CrimeDetailFragmentArgs by navArgs()

    ~~override fun onCreate(savedInstanceState: Bundle?) {~~
        ~~super.onCreate(savedInstanceState)~~

        ~~crime = Crime(~~
            ~~id = UUID.randomUUID(),~~
            ~~title = "",~~
            ~~date = Date(),~~
            ~~isSolved = false~~
        ~~)~~

        ~~Log.d(TAG, "The crime ID is: ${args.crimeId}")~~
    ~~}~~
    ...

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        binding.apply {
            crimeTitle.doOnTextChanged { text, _, _, _ ->
                ~~crime = crime.copy(title = text.toString())~~
            }

            crimeDate.apply {
                ~~text = crime.date.toString()~~
                isEnabled = false
            }

            crimeSolved.setOnCheckedChangeListener { _, isChecked ->
                ~~crime = crime.copy(isSolved = isChecked)~~
            }
        }
    }
    ...
}
```

As you have seen before, fragments are not well suited for handling state, because they are re-created during configuration changes. Create a **`CrimeDetailViewModel`**, extending the **`ViewModel`** class. Expose the state of the detail screen as a **`StateFlow`** holding a **`Crime`**. As you saw in Chapter 12, the **`StateFlow`** class does a good job of providing consumers with the freshest data. As you update the **`StateFlow`**, those changes will be pushed out to the **`CrimeDetailFragment`**.

### Listing 13.16 Bare skeleton for **`CrimeDetailViewModel`** (`CrimeDetailViewModel.kt`)

```
class CrimeDetailViewModel : ViewModel() {
    private val crimeRepository = CrimeRepository.get()

    private val _crime: MutableStateFlow<Crime?> = MutableStateFlow(null)
    val crime: StateFlow<Crime?> = _crime.asStateFlow()
}
```

Recall from Chapter 12 that you want to expose your data as a **`StateFlow`** and not a **`MutableStateFlow`**. This will help reinforce your unidirectional data flow: The source of data cannot be directly mutated by its consumers. And, as you will see, this also allows you to expose functions in a more deliberate fashion that gives consumers ways to send up user input.

Keeping the properties within the **`Crime`** as read-only `val`s instead of read/write `var`s also helps reinforce unidirectional data flow. While it does not truly make the **`Crime`** class "immutable," it does push consumers to create copies of data instead of directly mutating an instance. All of this works together to keep the flow of data streaming in one direction.

Your **`CrimeDetailViewModel`** will need to know the ID of the crime to load when it is created. There are a few ways to go about getting it this ID, but the most effective is to declare the ID as a constructor parameter so that your **`CrimeDetailViewModel`** can start loading the data as soon as it is created.

Previously, you have not used the constructor when creating an instance of your various **`ViewModel`**s. Instead, you have used the **`viewModels`** property delegate to obtain an instance, so that you get the same instance across configuration changes. By default, when using the **`viewModels`** property delegate, your **`ViewModel`** can only have a constructor with either no arguments or with a single **`SavedStateHandle`** argument.

But there is a way to add additional arguments to a **`ViewModel`**: creating a class that implements the **`ViewModelProvider.Factory`** interface. This interface allows you to control how a **`ViewModel`** is created and provided to fragments and activities. The **`ViewModelProvider.Factory`** interface is an example of the *factory software design pattern*: as a real-life car factory knows how to make cars, **`ViewModelProvider.Factory`** knows how to make **`ViewModel`** instances.

For CriminalIntent, you will create a **`CrimeDetailViewModelFactory`**, and it will know how to create **`CrimeDetailViewModel`** instances. Unlike the **`ViewModel`** subclasses you have seen so far, classes that implement the **`ViewModelProvider.Factory`** interface can take in constructor parameters.

In `CrimeDetailViewModel.kt`, create the **`CrimeDetailViewModelFactory`** class. Then pass in the crime's ID through its constructor and use it to load the crime from the database into the `crime` **`StateFlow`** class property.

### Listing 13.17 Building a factory for **`CrimeDetailViewModel`** (`CrimeDetailViewModel.kt`)

```
class CrimeDetailViewModel(crimeId: UUID) : ViewModel() {
    private val crimeRepository = CrimeRepository.get()

    private val _crime: MutableStateFlow<Crime?> = MutableStateFlow(null)
    val crime: StateFlow<Crime?> = _crime.asStateFlow()

    init {
        viewModelScope.launch {
            _crime.value = crimeRepository.getCrime(crimeId)
        }
    }
}

class CrimeDetailViewModelFactory(
    private val crimeId: UUID
) : ViewModelProvider.Factory {
    override fun <T : ViewModel> create(modelClass: Class<T>): T {
        return CrimeDetailViewModel(crimeId) as T
    }
}
```

Here, you create an instance of your **`CrimeDetailViewModelFactory`** by invoking its constructor, passing in the crime's ID in as a constructor parameter. Once you have the crime's ID as a class property for **`CrimeDetailViewModelFactory`**, you use it when creating instances of **`CrimeDetailViewModel`**. That is how you will be able to pass in the crime ID to **`CrimeDetailViewModel`** through its constructor.

The last part of this work uses the new **`CrimeDetailViewModelFactory`** class to access the **`CrimeDetailViewModel`** in **`CrimeDetailFragment`**. Under the hood, the **`viewModels`** property delegate is a function. This function has two parameters, each of them a lambda with a default value.

Override the default value for the last parameter and have **`viewModels`** return an instance of your new **`CrimeDetailViewModelFactory`**.

### Listing 13.18 Accessing your **`CrimeDetailViewModel`** (`CrimeDetailFragment.kt`)

```
class CrimeDetailFragment : Fragment() {
    ...
    private val args: CrimeDetailFragmentArgs by navArgs()

    private val crimeDetailViewModel: CrimeDetailViewModel by viewModels {
        CrimeDetailViewModelFactory(args.crimeId)
    }
    ...
}
```

With that, you have everything set up to start displaying crime information. As you did back in Chapter 12 with **`CrimeListFragment`**, you will use **`repeatOnLifecycle`** to collect from the crime's **`StateFlow`**. To make things a little more readable, update your UI in a private function called **`updateUi`**.

Most of the **`updateUi`** function will look similar to the code you had before. The one piece that is a little different is where you set the text on the **`EditText`**. There, you need to check whether the existing value and the new value being passed in are different. If they are different, then you update the **`EditText`**. If they are the same, you do nothing. This will prevent an infinite loop when you start listening to changes on the **`EditText`**.

### Listing 13.19 Updating your UI (`CrimeDetailFragment.kt`)

```
class CrimeDetailFragment : Fragment() {
    ...
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        binding.apply {
            ...
        }

        viewLifecycleOwner.lifecycleScope.launch {
            viewLifecycleOwner.lifecycle.repeatOnLifecycle(Lifecycle.State.STARTED) {
                crimeDetailViewModel.crime.collect { crime ->
                    crime?.let { updateUi(it) }
                }
            }
        }
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }

    private fun updateUi(crime: Crime) {
        binding.apply {
            if (crimeTitle.text.toString() != crime.title) {
                crimeTitle.setText(crime.title)
            }
            crimeDate.text = crime.date.toString()
            crimeSolved.isChecked = crime.isSolved
        }
    }
}
```

Run the app and select a crime. You will see the crime's details displayed onscreen.

Now that you have the UI displaying the crime's data, you need a way to send user input back up to the **`CrimeDetailViewModel`**. You could create individual functions to update each property on the crime (for example, a **`setTitle`** to update the crime's title and a **`setIsSolved`** to update the solved status), but that would be tedious.

Instead, write one function that takes in a lambda expression as a parameter. In the lambda expression, have the **`CrimeDetailViewModel`** provide the latest crime available and the **`CrimeDetailFragment`** update it in a safe manner. This will allow you to safely expose the crime as a **`StateFlow`** (instead of a **`MutableStateFlow`**) while still being able to easily update the crime as the user inputs data.

### Listing 13.20 Updating your crime (`CrimeDetailViewModel.kt`)

```
class CrimeDetailViewModel(crimeId: UUID) : ViewModel() {
    ...
    init {
        viewModelScope.launch {
            _crime.value = crimeRepository.getCrime(crimeId)
        }
    }

    fun updateCrime(onUpdate: (Crime) -> Crime) {
        _crime.update { oldCrime ->
            oldCrime?.let { onUpdate(it) }
        }
    }
}
```

Finally, hook the UI up to your new function. This will complete the loop of your unidirectional data flow.

### Listing 13.21 Responding to user input (`CrimeDetailFragment.kt`)

```
class CrimeDetailFragment : Fragment() {
    ...
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        binding.apply {
            crimeTitle.doOnTextChanged { text, _, _, _ ->
                crimeDetailViewModel.updateCrime { oldCrime ->
                    oldCrime.copy(title = text.toString())
                }
            }

            crimeDate.apply {
                isEnabled = false
            }

            crimeSolved.setOnCheckedChangeListener { _, isChecked ->
                crimeDetailViewModel.updateCrime { oldCrime ->
                    oldCrime.copy(isSolved = isChecked)
                }
            }
        }
        ...
    }
    ...
}
```

Run the app one more time. Select a crime and edit the title or toggle the checkbox. You will see your UI update. Now, your UI and your **`CrimeDetailViewModel`** will always stay in sync.

## Updating the Database

You can now modify a crime's details – but when you navigate away from the detail screen, all your changes are wiped away. Your last task in this chapter will be to save those changes to the database.

To accomplish this task, you will start at the database layer and work your way up. To begin, open `CrimeDao.kt`. Previously, you used the `@Query` annotation to write functions that query the database. Other annotations allow you to create functions that add, delete, or update records in the database.

Since the crime you are altering already exists in the database, write a function to update the entry. Annotate your function with `@Update` and use the `suspend` modifier so that you can call it from a coroutine scope.

### Listing 13.22 Adding a way to update the database (`CrimeDao.kt`)

```
@Dao
interface CrimeDao {
    @Query("SELECT * FROM crime")
    fun getCrimes(): Flow<List<Crime>>

    @Query("SELECT * FROM crime WHERE id=(:id)")
    suspend fun getCrime(id: UUID): Crime

    @Update
    suspend fun updateCrime(crime: Crime)
}
```

Expose that new function through your **`CrimeRepository`**.

### Listing 13.23 Updating **`CrimeRepository`** (`CrimeRepository.kt`)

```
class CrimeRepository private constructor(context: Context) {
    ...
    suspend fun getCrime(id: UUID): Crime = database.crimeDao().getCrime(id)

    suspend fun updateCrime(crime: Crime) {
        database.crimeDao().updateCrime(crime)
    }

    companion object {
        ...
    }
}
```

As you learned in Chapter 4, the **`ViewModel`** class has a very simple lifecycle. Unlike fragments and activities, which have many states, the **`ViewModel`** class only has two: It is either alive or dead. During the destruction of an instance, such as when you navigate away from a fragment, the **`onCleared()`** function is invoked on the **`ViewModel`**. This is the perfect place to save your changes to the crime.

In **onCleared()**, use the viewModelScope class property to launch a coroutine. Within that coroutine, access the latest value from the crime **StateFlow** and save it to the database.

### Listing 13.24 Updating the database when **CrimeDetailViewModel** is cleared (CrimeDetailViewModel.kt)

```
class CrimeDetailViewModel(crimeId: UUID) : ViewModel() {
    ...
    fun updateCrime(onUpdate: (Crime) -> Crime) {
        _crime.update { oldCrime ->
            oldCrime?.let { onUpdate(it) }
        }
    }

    override fun onCleared() {
        super.onCleared()

        viewModelScope.launch {
            crime.value?.let { crimeRepository.updateCrime(it) }
        }
    }
}
```

Run the app, select a crime, and edit it. Then back out of the detail screen. You would expect the changes to be reflected on the list screen – but, unfortunately, that is not happening.

As we mentioned in Chapter 12, coroutine scopes are tied to the lifecycles of the components they are associated to. For **ViewModel**, the viewModelScope property is alive and active as long as its associated **ViewModel** is. Once the **ViewModel** is destroyed, all the work running within the viewModelScope scope is canceled.

To save your changes, you will need a coroutine scope that outlives your **CrimeDetailFragment** and **CrimeDetailViewModel**.

One such scope is **GlobalScope**. As its name suggests, **GlobalScope** is a coroutine scope that is available globally and operates throughout the entire application lifecycle.

The work that is launched within **GlobalScope** is never canceled. However, work running inside **GlobalScope** cannot keep your application running. If your application is in the process of being stopped, the work within the **GlobalScope** will be unceremoniously stopped as well.

With many of the safety features of coroutines removed, **GlobalScope** can be dangerous if it is not used correctly. If work hangs within **GlobalScope**, it can needlessly consume resources.

However, for your purposes here, **GlobalScope** is a useful tool. **GlobalScope** lives longer than a viewModelScope, so you can use it to update your database in the background once the user moves away from **CrimeDetailFragment**.

Pass in **`GlobalScope`** as the default parameter for a new `coroutineScope` constructor property on the **`CrimeRepository`** class. You will have easy access to it while also having the flexibility to provide a new coroutine scope to **`CrimeRepository`** if functionality needs to change in the future. Use the new `coroutineScope` property to save the updated crime to the database. Also, since **`CrimeRepository`** handles managing the work of interacting with coroutine scopes, you no longer need the **`updateCrime`** function in **`CrimeRepository`** to be a suspending function, so remove the `suspend` modifier.

### Listing 13.25 Using **`GlobalScope`** (`CrimeRepository.kt`)

```
class CrimeRepository private constructor(
    context: Context,
    private val coroutineScope: CoroutineScope = GlobalScope
) {
    ...
    suspend fun getCrime(id: UUID): Crime = database.crimeDao().getCrime(id)

    suspend fun updateCrime(crime: Crime) {
        coroutineScope.launch {
            database.crimeDao().updateCrime(crime)
        }
    }
    ...
}
```

Finally, call your updated function from outside a coroutine scope in **`CrimeDetailViewModel`**.

### Listing 13.26 Making the final touches (`CrimeDetailViewModel.kt`)

```
class CrimeDetailViewModel(crimeId: UUID) : ViewModel() {
    ...
    override fun onCleared() {
        super.onCleared()

        viewModelScope.launch {
            crime.value?.let { crimeRepository.updateCrime(it) }
        }
    }
}
```

Run your app. You can now navigate between screens, update a crime's details, and see those details saved when you return to the list view. You now have a real, functional app. Congratulations! Over the next six chapters, you will refine CriminalIntent, building on this solid foundation.

## For the More Curious: A Better List Preview

In this chapter, you used the `tools` namespace to enable previews for the destinations in your navigation graph. But the preview for **`CrimeListFragment`** might not look the way you would expect. Each list item is a single, generic line of text, not the wonderful layout that you have built.

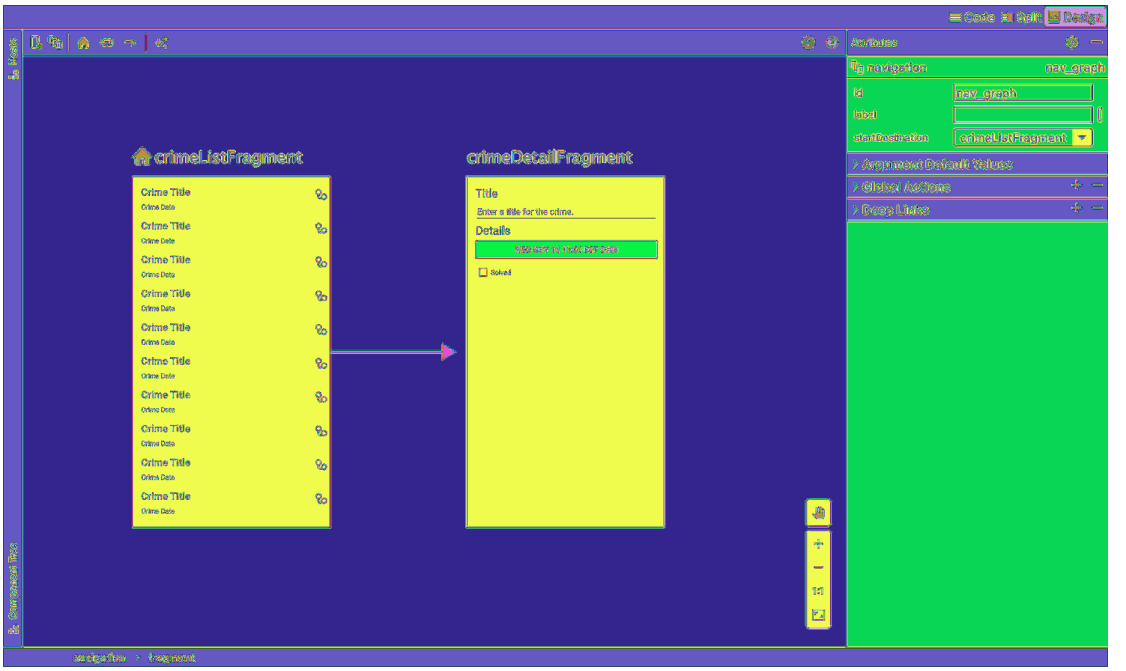
Thankfully, Android Studio also gives you tooling to preview the layout of a list item in a **`RecyclerView`**. Open up `fragment_crime_list.xml` and use the `tools:listitem` attribute to help the preview use the right layout for each list item.

### Listing 13.27 Providing previews for your list items (`fragment_crime_list.xml`)

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.recyclerview.widget.RecyclerView
        xmlns:android="http://schemas.android.com/apk/res/android"
        xmlns:tools="http://schemas.android.com/tools"
        android:id="@+id/crime_recycler_view"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        tools:listitem="@layout/list_item_crime" />
```

With that, your preview in `nav_graph.xml` is looking a lot better (Figure 13.11).

### Figure 13.11 Your navigation graph with previews



## Challenge: No Untitled Crimes

It is impossible to solve a crime if you do not know what crime occurred. But if you open the detail view for a crime, erase its title, and navigate back to the list view, you will save a crime with no name.

For this challenge, while in **`CrimeDetailFragment`**, prevent the user from navigating back to the list if the selected crime's title is blank. Using an **`OnBackPressedCallback`**, you can override the default Back button behavior. If the title is blank, give the user a hint that they should provide a description of the crime.

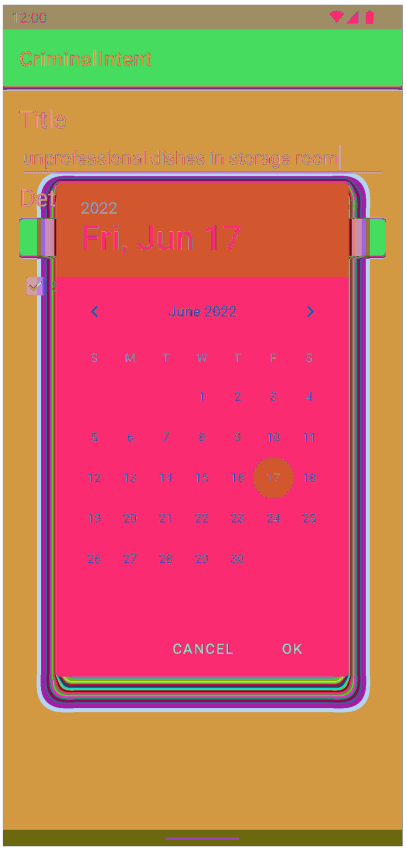
Some developer documentation that will be helpful for this challenge is at `developer.android.com/guide/navigation/navigation-custom-back`.

When using an **`OnBackPressedCallback`**, no navigation will happen unless you say so. So if there *is* a title, do not forget to use the **`NavController`** to pop off the **`CrimeDetailFragment`** from the navigation back stack.

# 14
# Dialogs and DialogFragment

Dialogs demand attention and input from the user. They are useful for presenting a choice or important information. In this chapter, you will add a dialog in which users can change the date of a crime. Pressing the date button in **`CrimeDetailFragment`** will present this dialog (Figure 14.1).

Figure 14.1 A dialog for picking the date of a crime




The dialog in Figure 14.1 is an instance of **`DatePickerDialog`**, a subclass of **`AlertDialog`**. **`DatePickerDialog`** displays a date selection prompt to the user and provides a listener interface you can implement to capture the selection. For creating more custom dialogs, **`AlertDialog`** is the all-purpose **`Dialog`** subclass that you will use most often.

## Creating a DialogFragment

When displaying a **DatePickerDialog**, it is a good idea to wrap it in an instance of **DialogFragment**, a subclass of **Fragment**. It is possible to display a **DatePickerDialog** without a **DialogFragment**, but it is not recommended. Having the **DatePickerDialog** managed by the **FragmentManager** gives you more options for presenting the dialog.

In addition, a bare **DatePickerDialog** will vanish if the device is rotated. If the **DatePickerDialog** is wrapped in a fragment, then the dialog will be re-created and put back onscreen after rotation.

For CriminalIntent, you are going to create a **DialogFragment** subclass named **DatePickerFragment**. Within **DatePickerFragment**, you will create and configure an instance of **DatePickerDialog**. Then you will add the new fragment to your navigation graph and navigate to it using the Navigation library.

Your first tasks are:

- creating the **DatePickerFragment** class
- building a **DatePickerFragment**
- getting the dialog onscreen using the Navigation library

Later in this chapter, you will pass the necessary data between **CrimeDetailFragment** and **DatePickerFragment**.

Create a new class named **DatePickerFragment** and make its superclass **DialogFragment**. Be sure to choose the Jetpack version of **DialogFragment**, which is **androidx.fragment.app.DialogFragment**.

Instead of overriding the **onCreateView** lifecycle function to display your UI, as you usually do, override the **onCreateDialog** lifecycle function to build a **DatePickerDialog** initialized with the current date. We will explain why after you make these changes.

### Listing 14.1 Creating a **DialogFragment** class (DatePickerFragment.kt)

```
class DatePickerFragment : DialogFragment() {

    override fun onCreateDialog(savedInstanceState: Bundle?): Dialog {
        val calendar = Calendar.getInstance()
        val initialYear = calendar.get(Calendar.YEAR)
        val initialMonth = calendar.get(Calendar.MONTH)
        val initialDay = calendar.get(Calendar.DAY_OF_MONTH)

        return DatePickerDialog(
            requireContext(),
            null,
            initialYear,
            initialMonth,
            initialDay
        )
    }
}
```

**DialogFragment**'s responsibility is to manage the dialog you want to display. The dialog itself does all the rendering to present itself onscreen. Because of this, your **DialogFragment** will not have a view of its own, like your other fragments have. If you need to customize the appearance or content of the dialog, you will do that by picking the dialog most appropriate for what you want to display – and modifying it if customizations are needed.

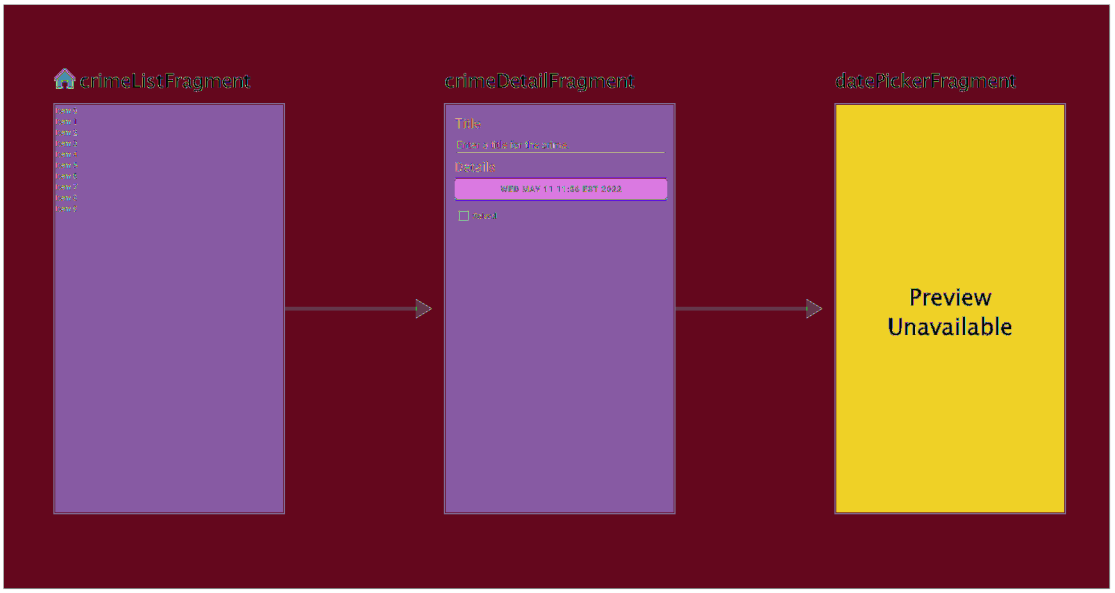
The **DatePickerDialog** constructor takes in several parameters. The first is a context object, which is required to access the necessary resources for the view. The second parameter is for the date listener, which you will add later in this chapter. The last three parameters are the year, month, and day that the date picker should be initialized to. Until you know the date of the crime, you can just initialize it to the current date.

## Showing a DialogFragment

You can integrate a **DialogFragment** into a navigation graph like any other fragment. Open up the `nav_graph.xml` file and switch to the design view. Click the Add destination button and select **DatePickerFragment** from the list of destinations. Since you will be navigating to this destination from **CrimeDetailFragment**, add a navigation action connecting the two destinations together.

Your updated navigation graph will look like Figure 14.2.

Figure 14.2 An updated navigation graph



When generating the navigation action for you, the tools gave it a verbose ID. In the code view, rename it `select_date`.

### Listing 14.2 Renaming a navigation action (`nav_graph.xml`)

```
<?xml version="1.0" encoding="utf-8"?>
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/nav_graph"
    app:startDestination="@id/crimeListFragment">

    <fragment
        android:id="@+id/crimeListFragment"
        android:name="com.bignerdranch.android.criminalintent.CrimeListFragment"
        android:label="CrimeListFragment"
        tools:layout="@layout/fragment_crime_list" >
        <action
            android:id="@+id/show_crime_detail"
            app:destination="@id/crimeDetailFragment" />
    </fragment>
    <fragment
        android:id="@+id/crimeDetailFragment"
        android:name="com.bignerdranch.android.criminalintent.CrimeDetailFragment"
        android:label="CrimeDetailFragment"
        tools:layout="@layout/fragment_crime_detail" >
        <argument
            android:name="crimeId"
            app:argType="java.util.UUID" />
        <action
            ~~android:id="@+id/action_crimeDetailFragment_to_datePickerFragment"~~
            android:id="@+id/select_date"
            app:destination="@id/datePickerFragment" />
    </fragment>
    <dialog
        android:id="@+id/datePickerFragment"
        android:name="com.bignerdranch.android.criminalintent.DatePickerFragment"
        android:label="DatePickerFragment" />
</navigation>
```

With that set up, you can go back to the Kotlin code. Open up **`CrimeDetailFragment`** and, in **`onViewCreated(…)`**, remove the code that disables the date button.

### Listing 14.3 Enabling your button (`CrimeDetailFragment.kt`)

```
class CrimeDetailFragment : Fragment() {
    ...
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        binding.apply {
            crimeTitle.doOnTextChanged { text, _, _, _ ->
                crimeDetailViewModel.updateCrime { oldCrime ->
                    oldCrime.copy(title = text.toString())
                }
            }

            crimeDate.apply {
                isEnabled = false
            }

            crimeSolved.setOnCheckedChangeListener { _, isChecked ->
                crimeDetailViewModel.updateCrime { oldCrime ->
                    oldCrime.copy(isSolved = isChecked)
                }
            }
        }
        ...
    }
    ...
}
```

Next, in the **`updateUi`** function, set a **`View.OnClickListener`** that navigates to your **`DatePickerFragment`** when the date button is pressed. This might seem like a strange place to set a **`View.OnClickListener`**, but the **`updateUi`** function is the only place where you have access to the latest crime, and you will soon need that access.

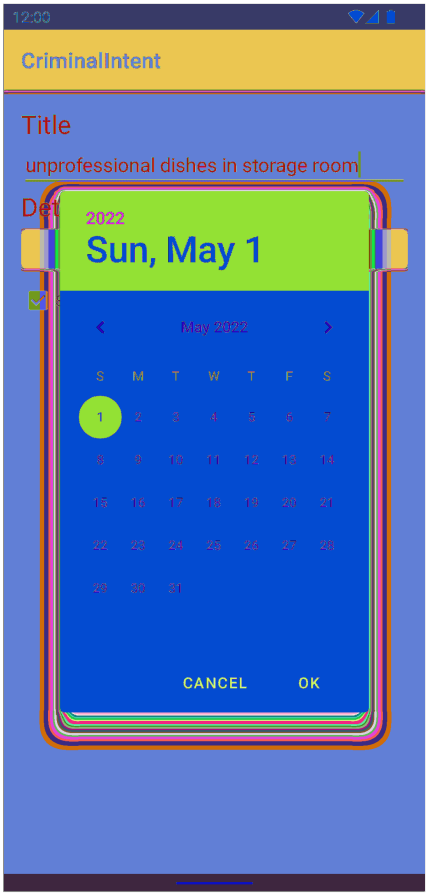
### Listing 14.4 Showing your **`DialogFragment`** (`CrimeDetailFragment.kt`)

```
class CrimeDetailFragment : Fragment() {
    ...
    private fun updateUi(crime: Crime) {
        binding.apply {
            if (crimeTitle.text.toString() != crime.title) {
                crimeTitle.setText(crime.title)
            }
            crimeDate.text = crime.date.toString()
            crimeDate.setOnClickListener {
                findNavController().navigate(
                    CrimeDetailFragmentDirections.selectDate()
                )
            }

            crimeSolved.isChecked = crime.isSolved
        }
    }
}
```

Run CriminalIntent, select a crime, and press the date button in the detail view to see the dialog (Figure 14.3).

Figure 14.3 A configured dialog




Your dialog is onscreen and looks good. In the next section, you will wire it up to present the **`Crime`**'s date and allow the user to change it.

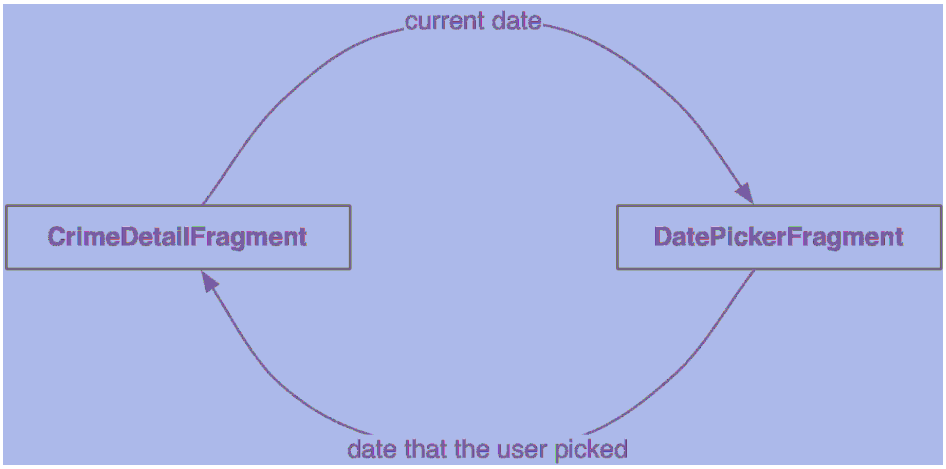
## Passing Data Between Two Fragments

In Chapter 13, you passed a crime's ID from **`CrimeListFragment`** to **`CrimeDetailFragment`**. Passing data to **`DialogFragment`** destinations works just the same.

What is different this time is that you are also passing a result back to **`CrimeDetailFragment`**: When the user selects a new date, **`CrimeDetailFragment`** needs the date to update its UI. Instead of using the Navigation library to handle that communication, this time you will rely on the Fragment Results API. Its usage is slightly different than the Activity Results API you used back in Chapter 7, but the concepts are the same, so you should feel right at home.

The conversation between your fragments will look like Figure 14.4. When the **`DatePickerFragment`** is started, the current date will be passed to it as an argument with the help of the Navigation library. Once the user chooses a date to set on the crime, it will be passed back to **`CrimeDetailFragment`** using the Fragment Results API. If the user does not choose a date and cancels their action, no result will be sent back.

Figure 14.4 Conversation between **`CrimeDetailFragment`** and **`DatePickerFragment`**



## Passing data to DatePickerFragment

Back in `nav_graph.xml`, view your navigation graph in the design view. Click the datePickerFragment. You want to add an argument to this destination, so click the + icon to the right of the Arguments section header in the attributes window. The Add Argument pop-up will appear.

In the pop-up, name your argument crimeDate. The **`Date`** class implements the **`Serializable`** interface, so select Custom Serializable... in the Type dropdown and then search for and select Date in the window that pops up (Figure 14.5). Leave the rest of the options alone and click Add to add the argument to your navigation graph.

Figure 14.5 Adding an argument to **`DatePickerFragment`**




Recall from Chapter 13 that the Safe Args plug-in bases the classes and functions it generates on your navigation graph. Since you have changed the required arguments for the **`DatePickerFragment`** destination, all the generated navigation actions to that class will be updated as well.

This means you now need to pass in a date when performing navigation with the **`CrimeDetailFragmentDirections.selectDate(date)`** function. Back in **`CrimeDetailFragment`**, update the code that performs the dialog navigation.

### Listing 14.5 Passing along the date (`CrimeDetailFragment.kt`)

```
class CrimeDetailFragment : Fragment() {
    ...
    private fun updateUi(crime: Crime) {
        binding.apply {
            if (crimeTitle.text.toString() != crime.title) {
                crimeTitle.setText(crime.title)
            }
            crimeDate.text = crime.date.toString()
            crimeDate.setOnClickListener {
                findNavController().navigate(
                    CrimeDetailFragmentDirections.selectDate(crime.date)
                )
            }

            crimeSolved.isChecked = crime.isSolved
        }
    }
}
```

**`DatePickerFragment`** needs to initialize the **`DatePickerDialog`** using the information held in the **`Date`**. However, initializing the **`DatePickerDialog`** requires **`Int`**s for the month, day, and year. **`Date`** is more of a timestamp and cannot provide **`Int`**s like this directly.

To get the **`Int`**s you need, you provide the **`Date`** to **`DatePickerFragment`**'s **`Calendar`** object. Then you can retrieve the required information from the **`Calendar`**.

In **`DatePickerFragment`**'s **`onCreateDialog(Bundle?)`**, get the **`Date`** from the navigation arguments and use it and the **`Calendar`** to initialize the **`DatePickerDialog`**.

### Listing 14.6 Extracting the date and initializing **`DatePickerDialog`** (`DatePickerFragment.kt`)

```
class DatePickerFragment : DialogFragment() {

    private val args: DatePickerFragmentArgs by navArgs()

    override fun onCreateDialog(savedInstanceState: Bundle?): Dialog {
        val calendar = Calendar.getInstance()
        calendar.time = args.crimeDate
        val initialYear = calendar.get(Calendar.YEAR)
        val initialMonth = calendar.get(Calendar.MONTH)
        val initialDate = calendar.get(Calendar.DAY_OF_MONTH)

        return DatePickerDialog(
            requireContext(),
            null,
            initialYear,
            initialMonth,
            initialDate
        )
    }
    ...
}
```

Now **`CrimeDetailFragment`** is successfully telling **`DatePickerFragment`** what date to show. Run your app and confirm this by selecting a crime and pressing the date button.

## Returning data to CrimeDetailFragment

To have **CrimeDetailFragment** receive the date back from **DatePickerFragment**, you need a way to keep track of the relationship between the two fragments.

With activities, you use the Activity Result APIs, and the **ActivityManager** keeps track of the parent-child activity relationship. When the child activity dies, the **ActivityManager** knows which activity should receive the result.

### Setting a fragment result

You can create a similar connection by making **CrimeDetailFragment** listen to results from the **DatePickerFragment**. This connection is automatically re-established after both **CrimeDetailFragment** and **DatePickerFragment** are destroyed and re-created by the OS. To create this relationship, you call the following **Fragment** function:

```
    setFragmentResultListener(
        requestKey: String,
        listener: ((requestKey: String, bundle: Bundle) -> Unit)
    )
```

This function uses a `requestKey` that will be shared between the two fragments and a lambda expression that will be invoked when **CrimeDetailFragment** is in the started lifecycle state with a result to consume. Under the hood, the **FragmentManager** keeps track of the listener.

In **DatePickerFragment**, define the `requestKey` in a companion object. That way, the constant will be easily accessible to both fragments.

Listing 14.7 Defining a constant (`DatePickerFragment.kt`)

```
class DatePickerFragment : DialogFragment() {

    private val args: DatePickerFragmentArgs by navArgs()

    override fun onCreateDialog(savedInstanceState: Bundle?): Dialog {
        ...
    }

    companion object {
        const val REQUEST_KEY_DATE = "REQUEST_KEY_DATE"
    }
}
```

Back in **CrimeDetailFragment**, use the new constant to call the **setFragmentResultListener** function in the **onViewCreated()** lifecycle function. Leave the lambda expression empty for now – you will get back to it shortly.

### Listing 14.8 Setting a listener (`CrimeDetailFragment.kt`)

```
class CrimeDetailFragment : Fragment() {
    ...
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        ...
        viewLifecycleOwner.lifecycleScope.launch {
            viewLifecycleOwner.lifecycle.repeatOnLifecycle(Lifecycle.State.STARTED) {
                crimeDetailViewModel.crime.collect { crime ->
                    crime?.let { updateUi(it) }
                }
            }
        }

        setFragmentResultListener(
            DatePickerFragment.REQUEST_KEY_DATE
        ) { requestKey, bundle ->
           // TODO
        }
    }
```

Back in **`DatePickerFragment`**, you need to set the fragment result once the user selects a new date. Add a listener to the **`DatePickerDialog`** that sends the date back to **`CrimeDetailFragment`**.

### Listing 14.9 Sending back the date (`DatePickerFragment.kt`)

```
class DatePickerFragment : DialogFragment() {

    private val args: DatePickerFragmentArgs by navArgs()

    override fun onCreateDialog(savedInstanceState: Bundle?): Dialog {
        val dateListener = DatePickerDialog.OnDateSetListener {
                _: DatePicker, year: Int, month: Int, day: Int ->

            val resultDate = GregorianCalendar(year, month, day).time

            setFragmentResult(REQUEST_KEY_DATE,
                              bundleOf(BUNDLE_KEY_DATE to resultDate))
        }

        val calendar = Calendar.getInstance()
        ...

        return DatePickerDialog(
            requireContext(),
            null,
            dateListener,
            initialYear,
            initialMonth,
            initialDate
        )
    }

    companion object {
        const val REQUEST_KEY_DATE = "REQUEST_KEY_DATE"
        const val BUNDLE_KEY_DATE = "BUNDLE_KEY_DATE"
    }
}
```

The **`OnDateSetListener`** is used to receive the date the user selects. The first parameter is for the **`DatePicker`** the result is coming from. Remember that when you are not using a parameter, you name it _ so it will be ignored.

The selected date is provided in year, month, and day format, but you need a **`Date`** to send back to **`CrimeDetailFragment`**. You pass these values to the **`GregorianCalendar`** and access the `time` property to get a **`Date`** object.

Once you have the date, it needs to be sent back to **`CrimeDetailFragment`**. To pass data between fragments, you need to package your results inside a key-value **`Bundle`**.

With the work in **`DatePickerFragment`** complete, the last thing you need to do is access the date from the **`Bundle`** passed back to **`CrimeDetailFragment`** and use it to update the crime in **`CrimeDetailViewModel`**. This lambda expression is only invoked when the user tries to save their changes, so you can be confident that the new data is in the **`Bundle`**. If the user dismisses the dialog or cancels their request to update the date, the lambda expression will not be invoked.

### Listing 14.10 Handling a result (`CrimeDetailFragment.kt`)

```
class CrimeDetailFragment : Fragment() {
    ...
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        ...
        viewLifecycleOwner.lifecycleScope.launch {
            viewLifecycleOwner.lifecycle.repeatOnLifecycle(Lifecycle.State.STARTED) {
                crimeDetailViewModel.crime.collect { crime ->
                    crime?.let { updateUi(it) }
                }
            }
        }

        setFragmentResultListener(
            DatePickerFragment.REQUEST_KEY_DATE
        ) { ~~requestKey~~ _, bundle ->
            ~~// TODO~~
            val newDate =
                bundle.getSerializable(DatePickerFragment.BUNDLE_KEY_DATE) as Date
            crimeDetailViewModel.updateCrime { it.copy(date = newDate) }
        }
    }
```

Now the circle is complete. The dates must flow. Those who control the dates control time itself. Run CriminalIntent to ensure that you can, in fact, control the dates. Change the date of a **`Crime`** and confirm that the new date appears in **`CrimeDetailFragment`**'s view. Then return to the list of crimes and check the **`Crime`**'s date to ensure that the database was updated.

In the next chapter, you will allow CriminalIntent's users to create a new crime (and get rid of the boring default crimes). And later in this book, in Chapter 28, you will see how dialogs are created in Jetpack Compose.

## Challenge: More Dialogs

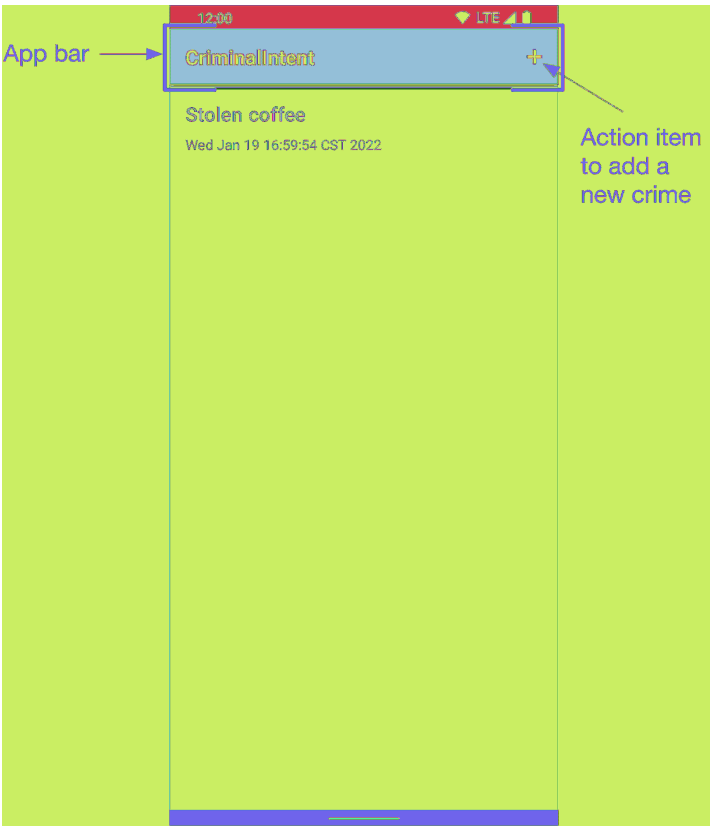
Write another dialog fragment named **`TimePickerFragment`** that allows the user to select the time of day the crime occurred using a **`TimePicker`**. Add another button to **`CrimeFragment`** to display the **`TimePickerFragment`**.

# 15
# The App Bar

A key component of any well-designed Android app is the *app bar*. The app bar presents actions that the user can take, provides an additional mechanism for navigation, and also offers design consistency and branding.

In this chapter, you will add a menu option to the app bar that lets users add a new crime (Figure 15.1).

Figure 15.1 CriminalIntent's app bar

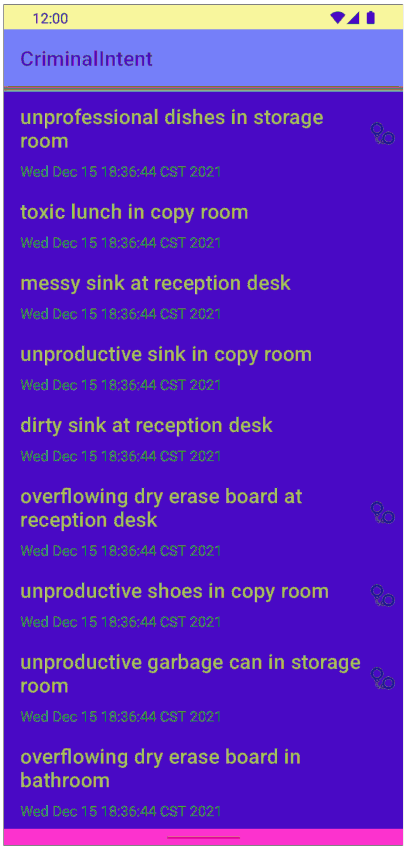



The app bar is often referred to as either the *action bar* or *toolbar*, depending on who you ask. You will learn more about these overlapping terms in the section called For the More Curious: App Bar vs Action Bar vs Toolbar near the end of this chapter.

## The Default App Bar

CriminalIntent already has a simple app bar in place (Figure 15.2).

Figure 15.2 The app bar




This is because Android Studio sets up all new projects with activities that extend from **`AppCompatActivity`** to include an app bar by default. It does this by:

- adding the Jetpack AppCompat and Material Components library dependencies
- applying one of the themes that includes an app bar

Open your `app/build.gradle` file (the one labeled (Module: CriminalIntent.app)) to see the AppCompat and Material Components dependencies:

```
dependencies {
    ...
    implementation 'androidx.appcompat:appcompat:1.4.1'
    implementation 'com.google.android.material:material:1.5.0'
    ...
```

"AppCompat" is short for "application compatibility." The Jetpack AppCompat foundation library contains classes and resources that are core to providing a consistent-looking UI across different versions of Android.

Android Studio automatically defines your app's theme when it creates your project. By default, this theme extends from `Theme.MaterialComponents.DayNight.DarkActionBar`. This theme is a part of the Material Components library, which brings the latest Material Design features (Google's design language) to your app. The Material Components library builds on AppCompat to make its design touches work across different versions.

Your app's theme, which specifies default styling for the entire app, is set in `res/values/themes.xml`:

```
<resources xmlns:tools="http://schemas.android.com/tools">
<!-- Base application theme. -->
    <style name="Theme.CriminalIntent"
        parent="Theme.MaterialComponents.DayNight.DarkActionBar">
        <!-- Primary brand color. -->
        <item name="colorPrimary">@color/purple_500</item>
        ...
    </style>
</resources>
```

The theme for your application is specified at the application level and optionally per activity in your manifest. Open `manifests/AndroidManifest.xml` and look at the `<application>` tag. Check out the `android:theme` attribute. You should see something similar to this:

```
<manifest ... >
    <application
        ...
        android:theme="@style/Theme.CriminalIntent" >
        ...
    </application>
</manifest>
```

OK, enough background. It is time to add an action to the app bar.

## Menus

The top-right area of the app bar is reserved for the app bar's menu. The menu consists of *action items* (sometimes referred to as *menu items*), which can perform an action on the current screen or on the app as a whole. You will add an action item to allow the user to create a new crime.

Your new action item will need a string resource for its label. Open `res/values/strings.xml` and add a string label describing your new action.

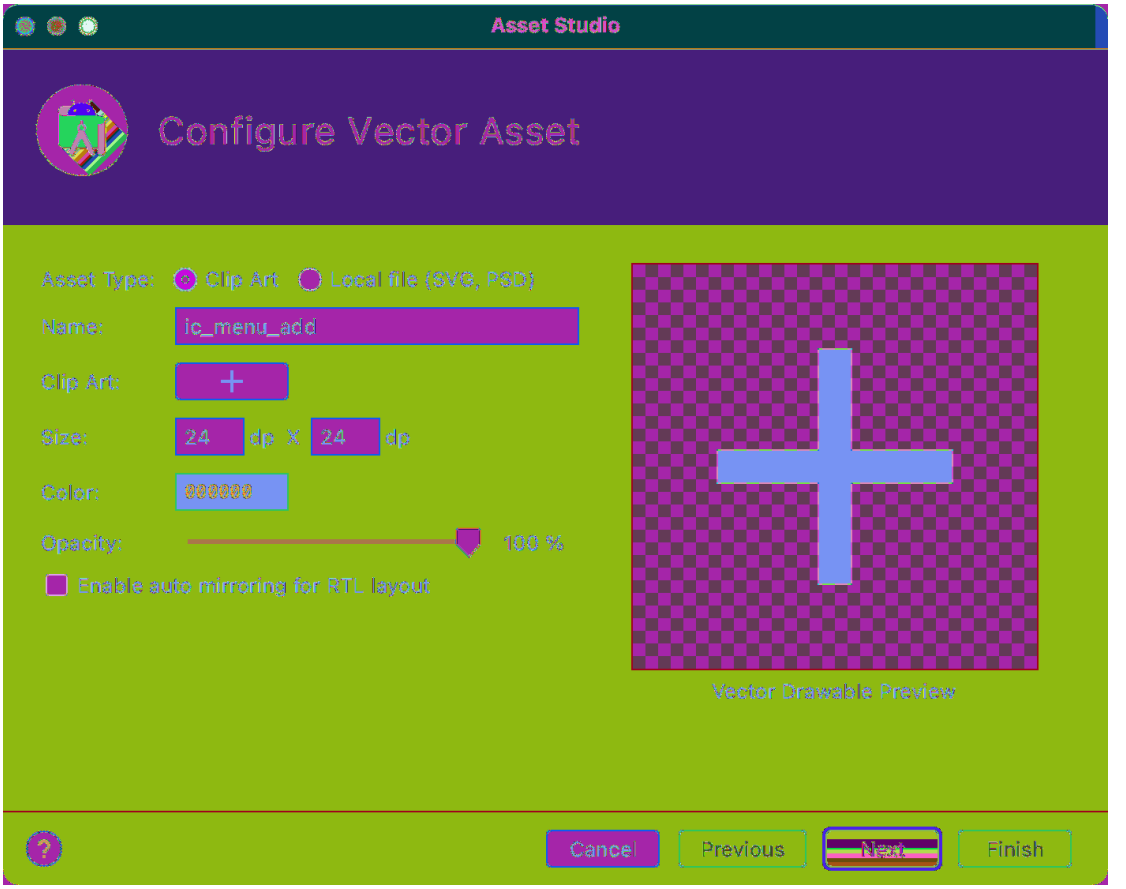
### Listing 15.1 Adding a string for menu (`res/values/strings.xml`)

```
<resources>
    ...
    <string name="crime_solved_label">Solved</string>
    <string name="new_crime">New Crime</string>
</resources>
```

The action item also needs an icon. Just like when you added a right arrow icon back in Chapter 2, you will use the Vector Asset Studio to add a vector drawable to your project. Select File → New → Vector Asset from the menu bar to bring up the Asset Studio. Click the button to the right of the Clip Art: label.

Within the Select Icon window, search for add and select the plus-shaped icon. Back on the Configure Vector Asset window, rename the asset `ic_menu_add`. The system will automatically tint the icon to the correct color, so you can leave its color set to the default black (Figure 15.3). With that done, you can click Next and then click Finish on the following screen to add the icon to your project.

Figure 15.3 Your new menu icon

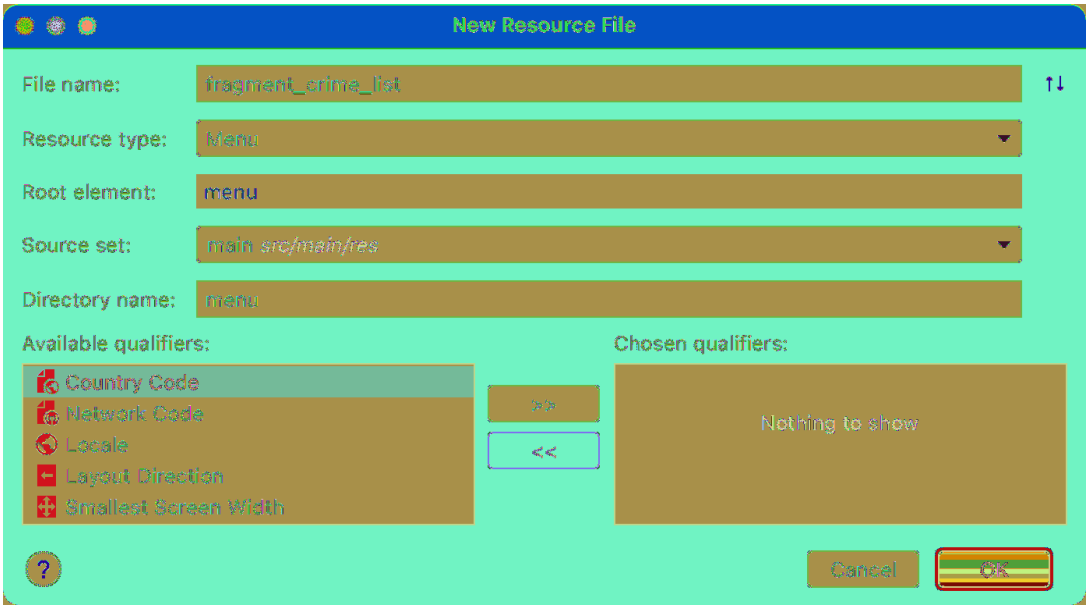


## Defining a menu in XML

Menus are a type of resource, similar to layouts. You create an XML description of a menu and place the file in the `res/menu` directory of your project. Android generates a resource ID for the menu file that you then use to inflate the menu in code.

In the project tool window, right-click the res directory and select New → Android resource file. Name the menu resource fragment_crime_list, change the Resource type to Menu, and click OK (Figure 15.4).

Figure 15.4 Creating a menu file




Here, you use the same naming convention for menu files as you do for layout files. Android Studio will generate `res/menu/fragment_crime_list.xml`, which has the same name as your **`CrimeListFragment`**'s layout file but lives in the `menu` folder. In the new file, switch to the code view and add an `item` element, as shown in Listing 15.2.

Listing 15.2 Creating a menu resource for **`CrimeListFragment`** (`res/menu/fragment_crime_list.xml`)

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
      xmlns:app="http://schemas.android.com/apk/res-auto">
    <item
        android:id="@+id/new_crime"
        android:icon="@drawable/ic_menu_add"
        android:title="@string/new_crime"
        app:showAsAction="ifRoom|withText"/>
</menu>
```

The `showAsAction` attribute refers to whether the item will appear in the app bar itself or in the *overflow menu*. You have piped together two values, `ifRoom` and `withText`, so the item's icon and text will appear in the app bar if there is room. If there is room for the icon but not the text, then only the icon will be visible. If there is no room for either, then the item will be relegated to the overflow menu.

If you have items in the overflow menu, those items will be represented by the three dots on the far-right side of the app bar, as shown in Figure 15.5.

Figure 15.5 Overflow menu in the app bar




Other options for `showAsAction` include `always` and `never`. Using `always` is not recommended; it is better to use `ifRoom` and let the OS decide. Using `never` is a good choice for less-common actions. In general, you should only put action items that users will access frequently in the app bar to avoid cluttering the screen.

### The app namespace

Notice that `fragment_crime_list.xml` uses the `xmlns` tag to define a namespace, `app`, which is separate from the usual `android` namespace declaration. This `app` namespace is then used to specify the `showAsAction` attribute.

You have used this namespace a few times already, such as for the Navigation library and **`ConstraintLayout`**. Libraries can use this namespace to declare custom attributes specific to their function. The `app:navGraph` attribute is an attribute that the Navigation library knows how to handle. The `app:layout_constraintEnd_toStartOf` attribute is specific to the **`ConstraintLayout`** library.

There is an `android:showAsAction` attribute built into the OS, but the AppCompat library defines a custom `app:showAsAction` to provide a consistent experience for all versions of Android. That is what you are using here.

## Creating the menu

In code, menus are managed by callbacks from the **`Activity`** class. When the menu is needed, Android calls the **`Activity`** function **`onCreateOptionsMenu(Menu)`**.

However, your design calls for code to be implemented in a fragment, not an activity. **`Fragment`** comes with its own set of menu callbacks, which you will implement in **`CrimeListFragment`**. The functions for creating the menu and responding to the selection of an action item are:

```
onCreateOptionsMenu(menu: Menu, inflater: MenuInflater)
onOptionsItemSelected(item: MenuItem): Boolean
```

In `CrimeListFragment.kt`, override **`onCreateOptionsMenu(Menu, MenuInflater)`** to inflate the menu defined in `fragment_crime_list.xml`.

### Listing 15.3 Inflating a menu resource (`CrimeListFragment.kt`)

```
class CrimeListFragment : Fragment() {
    ...
    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }

    override fun onCreateOptionsMenu(menu: Menu, inflater: MenuInflater) {
        super.onCreateOptionsMenu(menu, inflater)
        inflater.inflate(R.menu.fragment_crime_list, menu)
    }
    ...
}
```

Within this function, you call **`MenuInflater.inflate(Int, Menu)`** and pass in the resource ID of your menu file. This populates the **`Menu`** instance with the items defined in your file.

Notice that you call through to the superclass implementation of **`onCreateOptionsMenu(…)`**. This is not required, but we recommend calling through as a matter of convention. That way, any menu functionality defined by the superclass will still work. However, it is only a convention – the base **`Fragment`** implementation of this function does nothing.

By default, your overridden **`onCreateOptionsMenu(…)`** will not be invoked when your fragment is created. You must explicitly tell the system that your fragment should receive a call to **`onCreateOptionsMenu(…)`**. You do this by calling the following **`Fragment`** function:

```
setHasOptionsMenu(hasMenu: Boolean)
```

Override **`CrimeListFragment.onCreate(Bundle?)`** and let the system know that **`CrimeListFragment`** needs to receive menu callbacks.

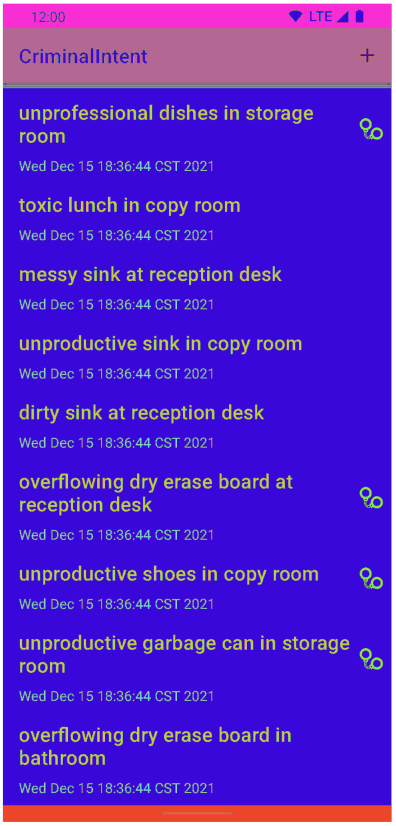
### Listing 15.4 Receiving menu callbacks (`CrimeListFragment.kt`)

```
class CrimeListFragment : Fragment() {
    ...
    private val crimeListViewModel: CrimeListViewModel by viewModels()

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setHasOptionsMenu(true)
    }
    ...
}
```

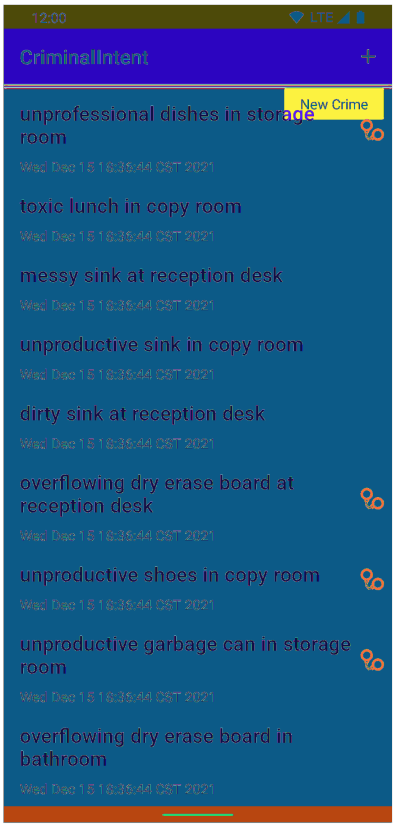
You can run CriminalIntent now to see your menu (Figure 15.6).

### Figure 15.6 Icon for the New Crime action item in the app bar



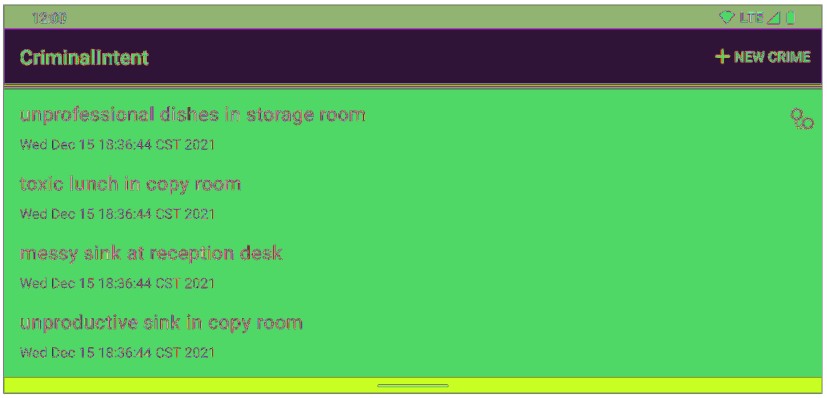
Where is the action item's text? Most phones only have enough room for the icon in portrait orientation. You can long-press an icon in the app bar to reveal its title (Figure 15.7).

Figure 15.7 Long-pressing an icon in the app bar shows the title




In landscape orientation, there is room in the app bar for the icon and the text (Figure 15.8).

Figure 15.8 Icon and text in the landscape app bar



## Responding to menu selections

To respond to the user pressing the new crime action item, you need a way for **`CrimeListFragment`** to add a new crime to the database. As before, you will build this new functionality from the database level up to the UI.

Starting all the way down at the **`CrimeDao`** interface, add a function to insert a crime into the database. Similar to the `@Query` and `@Update` annotations, the `@Insert` annotation will tell Room to generate code to enable you to insert a crime into the database.

### Listing 15.5 Adding a new crime to the database (`CrimeDao.kt`)

```
@Dao
interface CrimeDao {
    ...
    @Update
    suspend fun updateCrime(crime: Crime)

    @Insert
    suspend fun addCrime(crime: Crime)
}
```

Next, expose that function through the **`CrimeRepository`** class.

### Listing 15.6 Passing it through another layer (`CrimeRepository.kt`)

```
class CrimeRepository private constructor(
    context: Context,
    private val coroutineScope: CoroutineScope = GlobalScope
) {
    ...
    fun updateCrime(crime: Crime) {
        coroutineScope.launch {
            database.crimeDao().updateCrime(crime)
        }
    }

    suspend fun addCrime(crime: Crime) {
        database.crimeDao().addCrime(crime)
    }
    ...
}
```

Now, add a function to **`CrimeListViewModel`** to wrap a call to the repository's **`addCrime(Crime)`** function. Unlike other functions you have created within various **`ViewModel`** implementations, here you *do* want to expose this function as a suspending function. In your **`CrimeListFragment`**, you will want to navigate to **`CrimeDetailFragment`** after completing the insert into the database, and handling the asynchronous work within **`CrimeListFragment`** will be the simplest approach.

### Listing 15.7 Adding a new crime (`CrimeListViewModel.kt`)

```
class CrimeListViewModel : ViewModel() {
  ...
  init {
      ...
  }

  suspend fun addCrime(crime: Crime) {
      crimeRepository.addCrime(crime)
  }
}
```

When the user presses an action item, your fragment receives a callback to the function **`onOptionsItemSelected(MenuItem)`**. This function receives an instance of **`MenuItem`** that describes the user's selection.

Although your menu only contains one action item, menus often have more than one. You can determine which action item has been selected by checking the ID of the **`MenuItem`** and then respond appropriately. This ID corresponds to the ID you assigned to the **`MenuItem`** in your menu file.

In `CrimeListFragment.kt`, implement **`onOptionsItemSelected(MenuItem)`** to respond to **`MenuItem`** selection by creating a new **`Crime`**, saving it to the database, and then navigating to **`CrimeDetailFragment`**.

### Listing 15.8 Responding to menu selection (`CrimeListFragment.kt`)

```
class CrimeListFragment : Fragment() {
    ...
    override fun onCreateOptionsMenu(menu: Menu, inflater: MenuInflater) {
        super.onCreateOptionsMenu(menu, inflater)
        inflater.inflate(R.menu.fragment_crime_list, menu)
    }

    override fun onOptionsItemSelected(item: MenuItem): Boolean {
        return when (item.itemId) {
            R.id.new_crime -> {
                showNewCrime()
                true
            }
            else -> super.onOptionsItemSelected(item)
        }
    }

    private fun showNewCrime() {
        viewLifecycleOwner.lifecycleScope.launch {
            val newCrime = Crime(
                id = UUID.randomUUID(),
                title = "",
                date = Date(),
                isSolved = false
            )
            crimeListViewModel.addCrime(newCrime)
            findNavController().navigate(
                CrimeListFragmentDirections.showCrimeDetail(newCrime.id)
            )
        }
    }
}
```

This function returns a **`Boolean`** value. Once you have handled the **`MenuItem`**, you should return `true` to indicate that no further processing is necessary. If you return `false`, menu processing will continue by calling the hosting activity's **`onOptionsItemSelected(MenuItem)`** function (or, if the activity hosts other fragments, the **`onOptionsItemSelected`** function will get called on those fragments). The default case calls the superclass implementation if the item ID is not in your implementation.

In this brave new world where you can add crimes yourself, the seed database data you packaged within the app is no longer necessary. In **`CrimeRepository`**, remove the line including the prepackaged database from the `assets/` folder.

### Listing 15.9 Excluding the prepopulated database (`CrimeRepository.kt`)

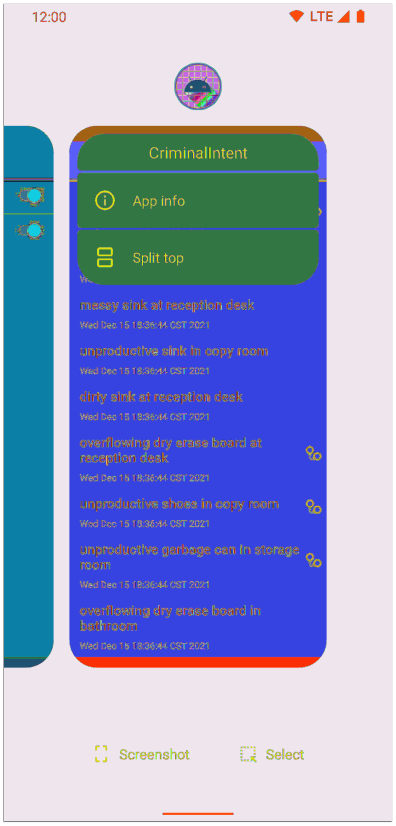
```
class CrimeRepository private constructor(
    context: Context,
    private val coroutineScope: CoroutineScope = GlobalScope
) {

    private val database: CrimeDatabase = Room
        .databaseBuilder(
            context.applicationContext,
            CrimeDatabase::class.java,
            DATABASE_NAME
        )
        .createFromAsset(DATABASE_NAME)
        .build()
    ...
}
```

Also, delete the `crime-database` file from your `assets` folder. It served you well. (If you use Android Studio's safe delete option, which is a good idea, it will warn you that there is a remaining usage of `crime-database`. The usage it has found is actually the database name, which you can verify for yourself before choosing Delete Anyway.)

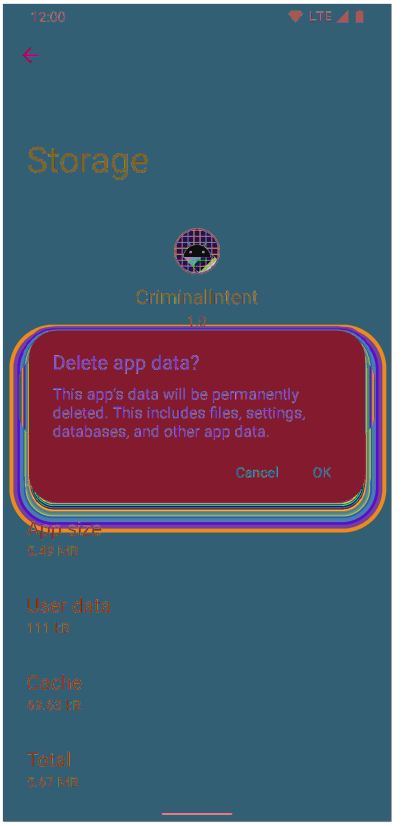
To get rid of the database loaded with the app on your device or emulator, you will need to clear the app's cache. Run the app and, while it is running, go to the overview screen on the device or emulator. Long-press the launcher icon for CriminalIntent. Press the App info option in the dropdown that appears (Figure 15.9).

Figure 15.9 Opening the app info for CriminalIntent



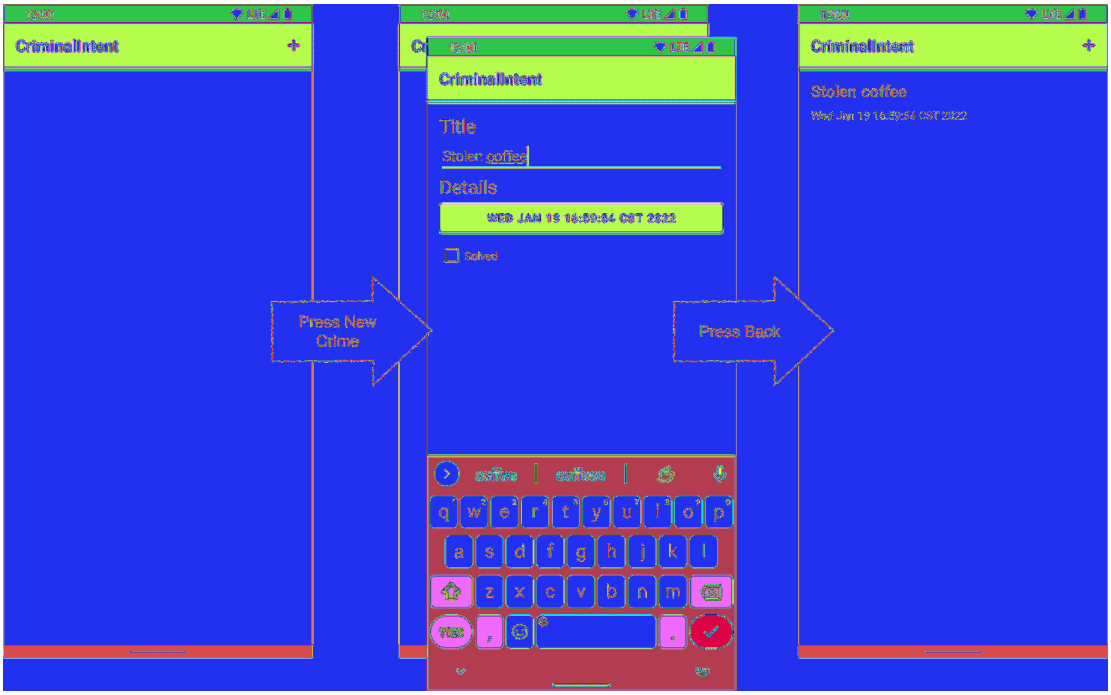
On the App info screen, select Storage. Next, select Clear storage and confirm your action on the dialog that appears (Figure 15.10).

Figure 15.10 Deleting CriminalIntent's data



Now, compile and run CriminalIntent. You should see an empty list to start with. Try out your new menu item to add a new crime. You should see the new crime appear in the crime list (Figure 15.11).

Figure 15.11 New crime flow




The empty list that you see before you add any crimes might be disconcerting. If you tackle the challenge in the section called Challenge: An Empty View for the RecyclerView at the end of this chapter, you will present a helpful clue when the list is empty.

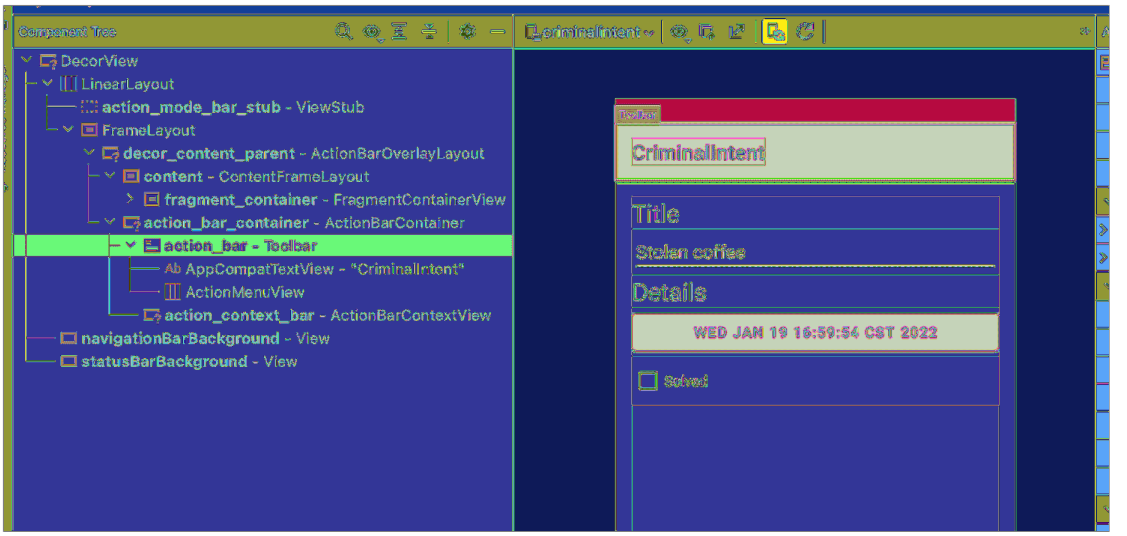
## For the More Curious: App Bar vs Action Bar vs Toolbar

You will often hear people refer to the app bar as the "toolbar" or the "action bar." And the official Android documentation uses these terms interchangeably. But are the app bar, action bar, and toolbar really the same thing? In short, no. The terms are related, but they are not exactly equivalent.

The UI design element itself is called an app bar. Prior to Android 5.0 (Lollipop, API level 21), the app bar was implemented using the **`ActionBar`** class. The terms action bar and app bar came to be treated as one and the same. Then, starting with Android 5.0, the **`Toolbar`** class was introduced as the preferred method for implementing the app bar. And, to keep things interesting, in Jetpack Compose, the newest way to create Android UIs, the app bar is implemented with a composable element called **`TopAppBar`**. (You will see this in Chapter 29.)

As of this writing, AppCompat uses the Jetpack **`Toolbar`** view to implement the app bar (Figure 15.12).

Figure 15.12 Layout inspector view of app bar




The **`ActionBar`** and **`Toolbar`** are very similar components. In fact, the toolbar builds on top of the action bar. It has a tweaked UI and is more flexible in the ways that you can use it.

The action bar has many constraints. It will always appear at the top of the screen. There can only be one action bar. The size of the action bar is fixed and should not be changed. The toolbar does not have these constraints.

In this chapter, you used a toolbar that was provided by one of the AppCompat themes. Alternatively, you can manually include a toolbar as a normal view in your activity or fragment's layout file. You can place this toolbar anywhere you like, and you can even include multiple toolbars on the screen at the same time.

This flexibility allows for interesting designs; for example, imagine if each fragment that you use maintained its own toolbar. When you host multiple fragments on the screen at the same time, each of them could bring along its own toolbar instead of sharing a single toolbar at the top of the screen.

If you are using the single activity architecture recommended by Google – as you are for CriminalIntent – you should strongly consider taking the approach of having each fragment provide its own app bar. Having your fragments messing with a shared app bar maintained by a single activity is a recipe for disaster. Taking this approach will keep all the functionality for an individual fragment encapsulated within that fragment. It will also enable you to improve and refactor individual fragments without worrying about breaking functionality for other fragments.

Equipped with this bit of history about the app bar-related APIs, you are now armed to more easily navigate the official developer documentation about this topic. And perhaps you can even spread the love and help clarify the overlap in these terms to future Android developers, since it is very confusing without the historical perspective.

## For the More Curious: Accessing the AppCompat App Bar

As you saw in this chapter, you can change the contents of the app bar by adding menu items. You can also change other attributes of the app bar at runtime, such as the title it displays.

To access the AppCompat app bar, you reference your **`AppCompatActivity`**'s `supportFragmentManager` property. From **`CrimeFragment`**, it would look something like this:

```
val appCompatActivity = activity as AppCompatActivity
val appBar = appCompatActivity.supportActionBar as Toolbar
```

The activity that is hosting the fragment is cast to an **`AppCompatActivity`**. Recall that because CriminalIntent uses the AppCompat library, you made your **`MainActivity`** a subclass of **`AppCompatActivity`**, which allows you to access the app bar.

Casting `supportActionBar` to a **`Toolbar`** allows you to call any **`Toolbar`** functions. (Remember, AppCompat uses a **`Toolbar`** to implement the app bar. But it used to use an **`ActionBar`**, as you just read, hence the somewhat-confusing name of the property to access the app bar.)

Once you have a reference to the app bar, you can apply changes like so:

```
appBar.setTitle(R.string.some_cool_title)
```

See the **`Toolbar`** API reference page for a list of other functions you can apply to alter the app bar (assuming your app bar is a **`Toolbar`**) at `developer.android.com/reference/androidx/appcompat/widget/Toolbar`.

Note that if you need to alter the contents of the app bar's menu while the activity is still displayed, you can trigger the **`onCreateOptionsMenu(Menu, MenuInflater)`** callback by calling the **`invalidateOptionsMenu()`** function. You can change the contents of the menu programmatically in the **`onCreateOptionsMenu`** callback, and those changes will appear once the callback is complete.

## Challenge: An Empty View for the RecyclerView

Currently, when CriminalIntent launches it displays an empty **`RecyclerView`** – a big white void. You should give users something to interact with when there are no items in the list.

For this challenge, display a message like There are no crimes and add a button to the view that will trigger the creation of a new crime.

Use the `visibility` property that exists on any **`View`** class to show and hide this new placeholder view when appropriate.

## Challenge: Deleting Crimes

Right now, you have the ability to add and update crimes in your database. But once the crime has been solved and justice has been served, it is often best to forgive and forget.

For this challenge, add the ability to delete a selected crime in **`CrimeDetailFragment`**. You can add an icon to **`CrimeDetailFragment`**'s app bar and hook it up the same way you did in this chapter.

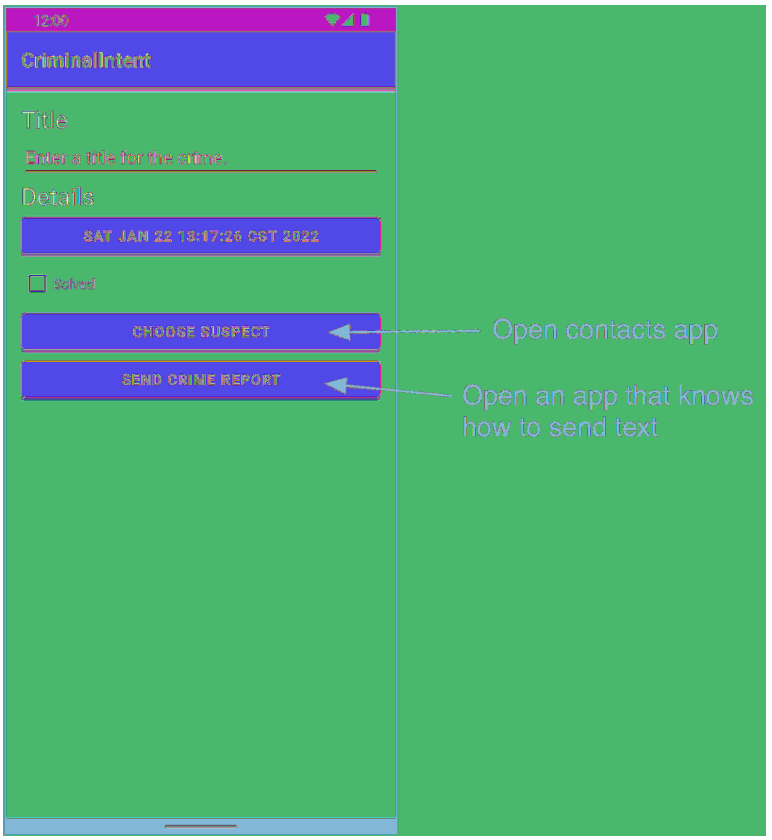
In addition to the `@Query`, `@Insert`, and `@Update` annotations you have used in your **`CrimeDao`**, there is one more annotation that might come in handy: `@Delete`.

# 16
# Implicit Intents

In Android, you can start an activity in another application on the device using an *intent*. In an *explicit intent*, you specify the class of the activity to start, and the OS will start it. In an *implicit intent*, you describe the job that you need done, and the OS will start an activity in an appropriate application for you.

In CriminalIntent, you will use implicit intents to enable picking a suspect for a **`Crime`** from the user's list of contacts and sending a text-based report of a crime. The user will choose a suspect from whatever contacts app is installed on the device and will be offered a choice of apps to send the crime report (Figure 16.1).

Figure 16.1 Opening contacts and text-sending apps



Using implicit intents to harness other applications is far easier than writing your own implementations for common tasks. Users also appreciate being able to use apps they already know and like in conjunction with your app.

Before you can create these implicit intents, there is some setup to do in CriminalIntent:

- add CHOOSE SUSPECT and SEND CRIME REPORT buttons to **`CrimeDetailFragment`**'s layouts
- add a `suspect` property to the **`Crime`** class that will hold the name of a suspect
- create a crime report using a set of format strings

## Adding Buttons

Update **`CrimeDetailFragment`**'s layout to include new buttons for accusation and tattling: namely, a suspect button and a report button. First, add the strings that these buttons will display.

### Listing 16.1 Adding button strings (`res/values/strings.xml`)

```
<resources>
    ...
    <string name="new_crime">New Crime</string>
    <string name="crime_suspect_text">Choose Suspect</string>
    <string name="crime_report_text">Send Crime Report</string>
</resources>
```

In `res/layout/fragment_crime_detail.xml`, add two buttons, as shown in Listing 16.2.

### Listing 16.2 Adding CHOOSE SUSPECT and SEND CRIME REPORT buttons (`res/layout/fragment_crime_detail.xml`)

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
              ... >
    ...
    <CheckBox
            android:id="@+id/crime_solved"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:text="@string/crime_solved_label"/>

    <Button
            android:id="@+id/crime_suspect"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:text="@string/crime_suspect_text"/>

    <Button
            android:id="@+id/crime_report"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:text="@string/crime_report_text"/>
</LinearLayout>
```

Preview the updated layout or run CriminalIntent to confirm that your new buttons are in place.

## Adding a Suspect Property

Next, open `Crime.kt` and give **`Crime`** a property to hold the name of a suspect. Provide a default value – an empty string – so that you do not have to update places in your codebase where you create a **`Crime`**.

Listing 16.3 Adding a suspect property (`Crime.kt`)

```
@Entity
data class Crime(
    @PrimaryKey val id: UUID,
    val title: String,
    val date: Date,
    val isSolved: Boolean,
    val suspect: String = ""
)
```

Since you updated your **`Crime`** class, and Room uses that class to create database tables for you, you need to make some changes to your database as well. Specifically, you need to increment the version of your **`CrimeDatabase`** class and tell Room how to migrate your database between the versions.

Room uses a versioning system to manage how data is structured within a database. Databases are intended as long-term storage for data, but as your app grows and adds new features, your model classes – and, by extension, your database – might change. You might need to add a new property to one of your entities, as you just did, which would cause a new column to be added to your database. Or you might need to change the type of one of your entity's properties, or perhaps remove a property altogether.

In all these cases, Room needs to know how to manage the change so that the structure of your database stays in sync with your database entity classes.

Room tracks the version of your database with the `version` property inside the `@Database` annotation on your **`CrimeDatabase`** class. When you first created the **`CrimeDatabase`** class, you set that value to `1`. As you make changes to the structure of your database, such as adding the `suspect` property on the **`Crime`** class, you increment that value. Since your initial database version is set to 1, you need to bump it up to 2 now.

When your app launches and Room builds the database, it will first check the version of the existing database on the device. If this version does not match the one you define in the `@Database` annotation, Room will begin the process to migrate that database to the latest version.

Room offers you two ways to handle migrations. The easy way is to call the **`fallbackToDestructiveMigration()`** function when building your **`CrimeDatabase`** instance. But, as the name hints, when this function is invoked Room will delete all the data within the database and re-create a new version. This means that all the data will be lost, leading to very unhappy users.

The better way to handle migrations is to define **`Migration`** classes. The **`Migration`** class constructor takes in two parameters. The first is the database version you are migrating from, and the second is the version you are migrating to. In this case, you will provide the version numbers `1` and `2`.

The only function you need to implement in your **`Migration`** object is **`migrate(SupportSQLiteDatabase)`**. You use the database parameter to execute any SQL commands necessary to upgrade your tables. (Room uses SQLite under the hood, as you read about in Chapter 12.) The `ALTER TABLE` and `ADD COLUMN` commands will add the new suspect column to the crime table.

Open `CrimeDatabase.kt`, increment the version, and add a migration. Between versions 1 and 2 of CriminalIntent's database, a new property was added to **`Crime`**: a **`String`** property named `suspect`. The corresponding migration will include a single instruction to add a `suspect` column to the table that stores your crimes.

### Listing 16.4 Adding database migration (`database/CrimeDatabase.kt`)

```
@Database(entities = [Crime::class], version = 1 version = 2)
@TypeConverters(CrimeTypeConverters::class)
abstract class CrimeDatabase : RoomDatabase() {
    abstract fun crimeDao(): CrimeDao
}

val migration_1_2 = object : Migration(1, 2) {
    override fun migrate(database: SupportSQLiteDatabase) {
        database.execSQL(
            "ALTER TABLE Crime ADD COLUMN suspect TEXT NOT NULL DEFAULT ''"
        )
    }
}
```

After you create your **`Migration`**, you need to provide it to your database when it is created. Open `CrimeRepository.kt` and provide the migration to Room when creating your **`CrimeDatabase`** instance. Call **`addMigrations(…)`** before calling the **`build()`** function. **`addMigrations()`** takes in a variable number of **`Migration`** objects, so you can pass all your migrations in when you declare them.

### Listing 16.5 Providing migration to Room (`CrimeRepository.kt`)

```
class CrimeRepository private constructor(
    context: Context,
    private val coroutineScope: CoroutineScope = GlobalScope
) {

    private val database: CrimeDatabase = Room
        .databaseBuilder(
            context.applicationContext,
            CrimeDatabase::class.java,
            DATABASE_NAME
        )
        .addMigrations(migration_1_2)
        .build()
    ...
}
```

Once your migration is in place, run CriminalIntent to make sure everything builds correctly. The app behavior should be the same as before you applied the migration, and you should see the crime you added in Chapter 15. You will make use of the newly added column shortly.

## Using a Format String

The last preliminary step is to create a template crime report that can be configured with the specific crime's details. Because you will not know a crime's details until runtime, you must use a format string with placeholders that can be replaced at runtime. Here is the format string you will use:

```
%1$s! The crime was discovered on %2$s. %3$s, and %4$s
```

`%1$s`, `%2$s`, etc. are placeholders that expect string arguments. In code, you will call **`getString(…)`** and pass in the format string and four other strings in the order in which they should replace the placeholders. The result will be a report along the lines of, "Stolen yogurt! The crime was discovered on Wed., May 11. The case is not solved, and there is no suspect."

First, in `strings.xml`, add the strings shown in Listing 16.6.

Listing 16.6 Adding string resources (`res/values/strings.xml`)

```
<resources>
    ...
    <string name="crime_suspect_text">Choose Suspect</string>
    <string name="crime_report_text">Send Crime Report</string>
    <string name="crime_report">%1$s!
      The crime was discovered on %2$s. %3$s, and %4$s
    </string>
    <string name="crime_report_solved">The case is solved</string>
    <string name="crime_report_unsolved">The case is not solved</string>
    <string name="crime_report_no_suspect">there is no suspect.</string>
    <string name="crime_report_suspect">the suspect is %s.</string>
    <string name="crime_report_subject">CriminalIntent Crime Report</string>
    <string name="send_report">Send crime report via</string>
</resources>
```

In `CrimeDetailFragment.kt`, add a function that creates four strings and then pieces them together and returns a complete report.

Listing 16.7 Adding a **`getCrimeReport(crime: Crime)`** function (`CrimeDetailFragment.kt`)

```
private const val DATE_FORMAT = "EEE, MMM, dd"

class CrimeDetailFragment : Fragment() {
    ...
    private fun updateUi(crime: Crime) {
        ...
    }

    private fun getCrimeReport(crime: Crime): String {
        val solvedString = if (crime.isSolved) {
            getString(R.string.crime_report_solved)
        } else {
            getString(R.string.crime_report_unsolved)
        }

        val dateString = DateFormat.format(DATE_FORMAT, crime.date).toString()
        val suspectText = if (crime.suspect.isBlank()) {
            getString(R.string.crime_report_no_suspect)
        } else {
            getString(R.string.crime_report_suspect, crime.suspect)
        }

        return getString(
            R.string.crime_report,
            crime.title, dateString, solvedString, suspectText
        )
    }
}
```

(There are multiple **`DateFormat`** classes. Make sure you import **`android.text.format.DateFormat`**.)

Now the preliminaries are complete, and you can turn to implicit intents.

# Using Implicit Intents

An **`Intent`** is an object that describes to the OS something that you want it to do. With the *explicit* intents that you have created thus far, you explicitly name the activity that you want the OS to start, like:

```
val intent = Intent(this, CheatActivity::class.java)
startActivity(intent)
```

With an *implicit* intent, you describe to the OS the job that you want done. The OS then starts the activity that has advertised itself as capable of doing that job. If the OS finds more than one capable activity, then the user is offered a choice.

## Parts of an implicit intent

Here are the critical parts of an intent that you can use to define the job you want done:

the *action* that you are trying to perform

> Actions are typically constants from the **`Intent`** class. For example, if you want to view a URL, you can use `Intent.ACTION_VIEW` for your action. To send something, you use `Intent.ACTION_SEND`.

the location of any *data*

> The data can be something outside the device, like the URL of a web page, but it can also be a URI to a file or a *content URI* pointing to a record in a **`ContentProvider`**.

the *type* of data that the action is for

> This is a MIME type, like `text/html` or `audio/mpeg3`. If an intent includes a location for data, then the type can usually be inferred from that data.

optional *categories*

> If the action is used to describe *what* to do, the category usually describes *where*, *when*, or *how* you are trying to use an activity. Android uses the category `android.intent.category.LAUNCHER` to indicate that an activity should be displayed in the top-level app launcher. The `android.intent.category.INFO` category, on the other hand, indicates an activity that shows information about a package to the user but should not show up in the launcher.

So, for example, a simple implicit intent for viewing a website would include an action of `Intent.ACTION_VIEW` and a data **`Uri`** that is the URL of a website.

Based on this information, the OS will launch the appropriate activity of an appropriate application. (Or, if it finds more than one candidate, present the user with a choice.)

An activity advertises itself as an appropriate activity for `ACTION_VIEW` via an *intent filter* in the manifest. If you wanted to write a browser app, for instance, you would include the following intent filter in the declaration of the activity that should respond to `ACTION_VIEW`:

```
<activity
    android:name=".BrowserActivity"
    android:label="@string/app_name"
    android:exported="true" >
    <intent-filter>
        <action android:name="android.intent.action.VIEW" />
        <category android:name="android.intent.category.DEFAULT" />
        <data android:scheme="http" android:host="www.bignerdranch.com" />
    </intent-filter>
</activity>
```

To respond to implicit intents, an activity must have the `android:exported` attribute set to `true` and, in an intent filter, the `DEFAULT` category explicitly included. The `action` element in the intent filter tells the OS that the activity is capable of performing the job, and the `DEFAULT` category tells the OS that this activity should be considered for the job when the OS is asking for volunteers. This `DEFAULT` category is implicitly added to every implicit intent.

Implicit intents can also include extras, just like explicit intents. But any extras on an implicit intent are not used by the OS to find an appropriate activity. The action and data parts of an intent can also be used in conjunction with an explicit intent. That would be the equivalent of telling a particular activity to do something specific.

## Sending a crime report

Let's see how this works by creating an implicit intent to send a crime report in CriminalIntent. The job you want done is sending plain text; the crime report is a string. So the implicit intent's action will be `ACTION_SEND`. It will not point to any data or have any categories, but it will specify a type of `text/plain`.

In **`CrimeDetailFragment`**'s **`updateUi()`** method, set a listener on your new crime report button. Within the listener's implementation, create an implicit intent and pass it into **`startActivity(Intent)`**.

### Listing 16.8 Sending a crime report (`CrimeDetailFragment.kt`)

```
class CrimeDetailFragment : Fragment() {
    ...
    private fun updateUi(crime: Crime) {
      binding.apply {
          ...
          crimeSolved.isChecked = crime.isSolved

          crimeReport.setOnClickListener {
              val reportIntent = Intent(Intent.ACTION_SEND).apply {
                  type = "text/plain"
                  putExtra(Intent.EXTRA_TEXT, getCrimeReport(crime))
                  putExtra(
                      Intent.EXTRA_SUBJECT,
                      getString(R.string.crime_report_subject)
                  )
              }

              startActivity(reportIntent)
          }
      }
  }
    ...
}
```

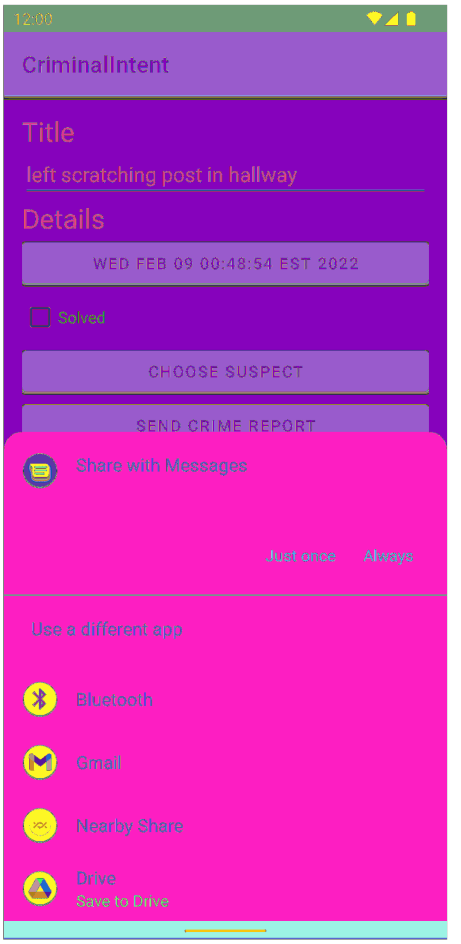
Here you use the **`Intent`** constructor that accepts a string that is a constant defining the action. There are other constructors that you can use depending on what kind of implicit intent you need to create. You can find them all on the **`Intent`** reference page in the documentation. There is no constructor that accepts a type, so you set it explicitly.

You include the text of the report and the string for the subject of the report as extras. Note that these extras use constants defined in the **`Intent`** class. Any activity responding to this intent will know these constants and what to do with the associated values.

Starting an activity from a fragment works nearly the same as starting an activity from another activity. You call **`Fragment`**'s **`startActivity(Intent)`** function, which calls the corresponding **`Activity`** function behind the scenes.

Run CriminalIntent and press the SEND CRIME REPORT button. Because this intent will likely match many activities on the device, you will probably see a list of activities presented in a chooser (Figure 16.2). You may need to scroll down in the list to see all of the activities.

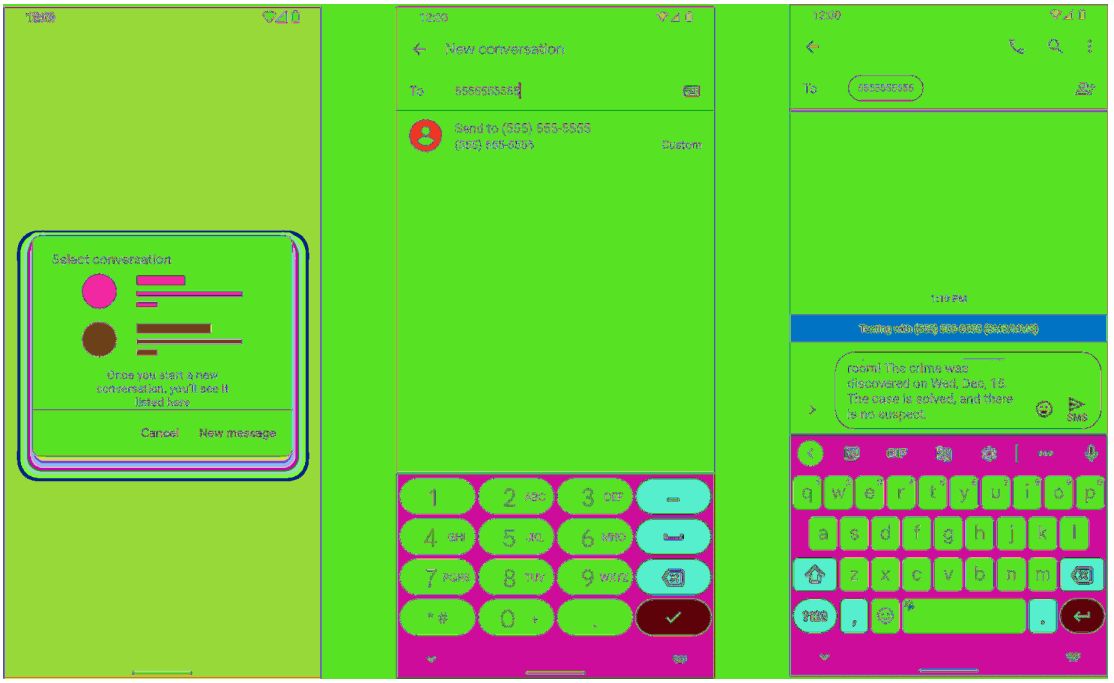
Figure 16.2 Activities volunteering to send your crime report




If you are offered a choice, make a selection. You will see your crime report loaded into the app that you chose. All you have to do is address and send it.

Apps like Gmail and Google Drive require you to log in with a Google account. It is simpler to choose the Messages app, which does not require you to log in. Press New message in the Select conversation dialog window, type any phone number in the To field, and press the Send to *phone number* label that appears (Figure 16.3). You will see the crime report in the body of the message.

Figure 16.3 Sending a crime report with the Messages app




If, on the other hand, you do not see a chooser, that means one of two things. Either you have already set a default app for an identical implicit intent, or your device has only a single activity that can respond to this intent.

Often, it is best to go with the user's default app for an action. But in this situation, that is not ideal. It is very common for people to use different messaging apps for different groups of people. The user might use WhatsApp with their family, Slack with their coworkers, and Discord with their friends. Here, it would be best to present the user with all of their options for sending a message so they can choose which app to use each time.

With a little extra configuration, you can create a chooser to be shown every time an implicit intent is used to start an activity. After you create your implicit intent as before, you call the **`Intent.createChooser(Intent, String)`** function and pass in the implicit intent and a string for the chooser's title.

Then you pass the intent returned from **`createChooser(…)`** into **`startActivity(…)`**.

In `CrimeDetailFragment.kt`, create a chooser to display the activities that respond to your implicit intent.

### Listing 16.9 Using a chooser (`CrimeDetailFragment.kt`)

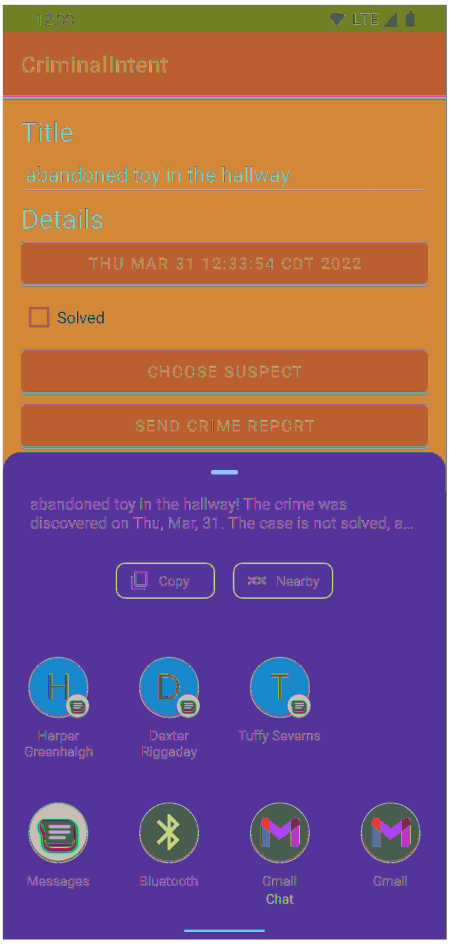
```
class CrimeDetailFragment : Fragment() {
    ...
    private fun updateUi(crime: Crime) {
        binding.apply {
            ...
            crimeReport.setOnClickListener {
                val reportIntent = Intent(Intent.ACTION_SEND).apply {
                    type = "text/plain"
                    putExtra(Intent.EXTRA_TEXT, getCrimeReport(crime))
                    putExtra(
                        Intent.EXTRA_SUBJECT,
                        getString(R.string.crime_report_subject)
                    )
                }

                ~~startActivity(reportIntent)~~
                val chooserIntent = Intent.createChooser(
                    reportIntent,
                    getString(R.string.send_report)
                )
                startActivity(chooserIntent)
            }
        }
    }
    ...
}
```

Run CriminalIntent again and press the SEND CRIME REPORT button. As long as you have more than one activity that can handle your intent, you will be offered a list to choose from (Figure 16.4).

This chooser has changed many times over the various versions of Android. On older versions of Android, you might see the title you passed in when creating the `chooserIntent` on the chooser. On newer versions of Android, you might be presented with the people in your contacts for various apps to select.

Figure 16.4 Sending text with a chooser



## Asking Android for a contact

Now you are going to create another implicit intent that enables users to choose a suspect from their contacts. You could set up the **`Intent`** by hand, but it is easier to use the Activity Results APIs that you used in GeoQuiz.

In Chapter 7, you learned about classes that define a contract between you and the **`Activity`** you are starting. This contract defines the input you provide to start the **`Activity`** and the output you expect to receive as a result. There, you used the contract **`ActivityResultContracts.StartActivityForResult()`** – a basic contract that takes in an **`Intent`** and provides an **`ActivityResult`** as output.

You could use **`ActivityResultContracts.StartActivityForResult()`** again. But instead, you will use the more specific **`ActivityResultContracts.PickContact()`** class. It is a better option here because, as its name indicates, it is specifically designed for this use case.

**`ActivityResultContracts.PickContact()`** will send the user to an activity where they can select a contact. Once the user selects a contact, you will receive a **`Uri`** back as the result. You will see how to read the contact data from this **`Uri`** later in this chapter.

You expect a result back from the started activity, so you will use **`registerForActivityResult(…)`** again. In `CrimeDetailFragment.kt`, add the following:

### Listing 16.10 Registering for a result (`CrimeDetailFragment.kt`)

```
class CrimeDetailFragment : Fragment() {
    ...
    private val crimeDetailViewModel: CrimeDetailViewModel by viewModels {
        CrimeDetailViewModelFactory(args.crimeId)
    }

    private val selectSuspect = registerForActivityResult(
        ActivityResultContracts.PickContact()
    ) { uri: Uri? ->
        // Handle the result
    }
    ...
}
```

In **`onViewCreated()`**, set a click listener on the `crimeSuspect` button. Inside the listener, call the **`launch()`** function on your `selectSuspect` property. Unlike the work you did in Chapter 7, selecting a contact requires no input, so pass `null` into the **`launch()`** function.

### Listing 16.11 Sending an implicit intent (`CrimeDetailFragment.kt`)

```
class CrimeDetailFragment : Fragment() {
    ...
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        binding.apply {
            ...
            crimeSolved.setOnCheckedChangeListener { _, isChecked ->
                crimeDetailViewModel.updateCrime { oldCrime ->
                    oldCrime.copy(isSolved = isChecked)
                }
            }

            crimeSuspect.setOnClickListener {
                selectSuspect.launch(null)
            }
        }
    }
    ...
}
```

Next, modify **`updateUi(crime: Crime)`** to set the text on the CHOOSE SUSPECT button if the crime has a suspect. Use the **`String.ifEmpty()`** extension function to provide default text if there is no current suspect.

### Listing 16.12 Setting CHOOSE SUSPECT button text (`CrimeDetailFragment.kt`)

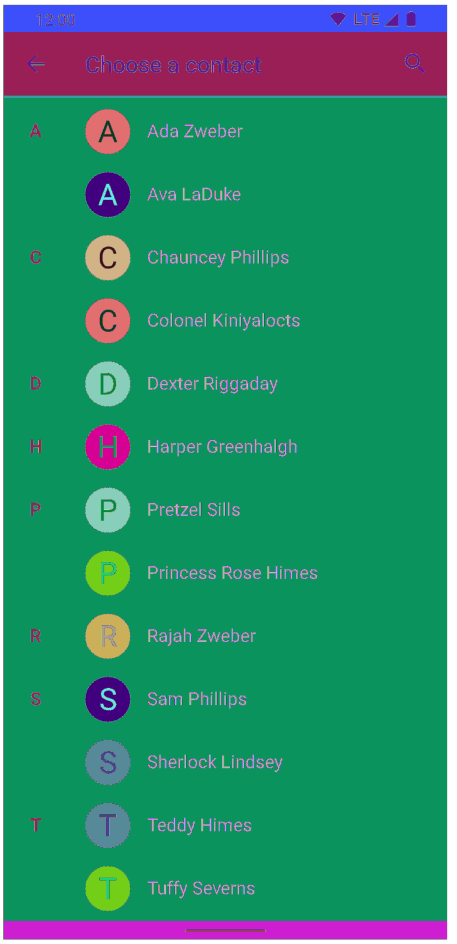
```
class CrimeDetailFragment : Fragment() {
    ...
    private fun updateUi(crime: Crime) {
        binding.apply {
            ...
            crimeReport.setOnClickListener {
                ...
            }

            crimeSuspect.text = crime.suspect.ifEmpty {
                getString(R.string.crime_suspect_text)
            }
        }
    }
    ...
}
```

Run CriminalIntent on a device that has a contacts app – use the emulator if your Android device does not have one. If you are using the emulator, add a few contacts using its Contacts app before you run CriminalIntent. Then run your app.

Press the CHOOSE SUSPECT button. You should see a list of contacts (Figure 16.5).

Figure 16.5 A list of possible suspects




If you have a different contacts app installed, your screen will look different. Again, this is one of the benefits of implicit intents. You do not have to know the name of the contacts application to use it from your app. Users can install whatever app they like best, and the OS will find and launch it.

### Getting data from the contacts list

Now you need to get a result back from the contacts application. Contacts information is shared by many applications, so Android provides an in-depth API for working with contacts information through a **`ContentProvider`**. Instances of this class wrap databases and make the data available to other applications. You can access a **`ContentProvider`** through a **`ContentResolver`**.

(The contacts database is a large topic in itself. We will not cover it here. If you would like to know more, read the Content Provider API guide at `developer.android.com/guide/topics/providers/content-provider-basics`.)

Because you started the activity with the **`ActivityResultContracts.PickContact()`** class, you might receive a data **`Uri`** as output. (We say "might" here because if the user cancels and does not select a suspect, your output will be `null`.) The **`Uri`** is not your suspect's name or any data about them; rather, it points at a resource you can query for that information.

In `CrimeDetailFragment.kt`, add a function to retrieve the contact's name from the contacts application. This is a lot of new code; we will explain it step by step after you enter it.

Listing 16.13 Pulling the contact's name out (`CrimeDetailFragment.kt`)

```
class CrimeDetailFragment : Fragment(), DatePickerFragment.Callbacks {
    ...
    private fun getCrimeReport(crime: Crime): String {
        ...
    }

    private fun parseContactSelection(contactUri: Uri) {
        val queryFields = arrayOf(ContactsContract.Contacts.DISPLAY_NAME)

        val queryCursor = requireActivity().contentResolver
            .query(contactUri, queryFields, null, null, null)

        queryCursor?.use { cursor ->
            if (cursor.moveToFirst()) {
                val suspect = cursor.getString(0)
                crimeDetailViewModel.updateCrime { oldCrime ->
                    oldCrime.copy(suspect = suspect)
                }
            }
        }
    }
    ...
}
```

In Listing 16.13, you create a query that asks for all the display names of the contacts in the returned data. Then you query the contacts database and get a **`Cursor`** object to work with. The **`Cursor`** points to a database table containing a single row and a single column. The row represents the contact the user selected, and the specified column has the contact's name.

The **`Cursor.moveToFirst()`** function accomplishes two things for you: It moves the cursor to the first row, and it returns a Boolean you use to determine whether there is data to read from. To extract the suspect's name, you call **`Cursor.getString(Int)`**, passing in `0`, to pull the contents of the first column in that first row as a string. Finally, you update the crime within your **`CrimeDetailViewModel`**.

Now, the suspect information is stored in the **`CrimeDetailViewModel`**, and your UI will update as it observes the **`StateFlow`**'s changes.

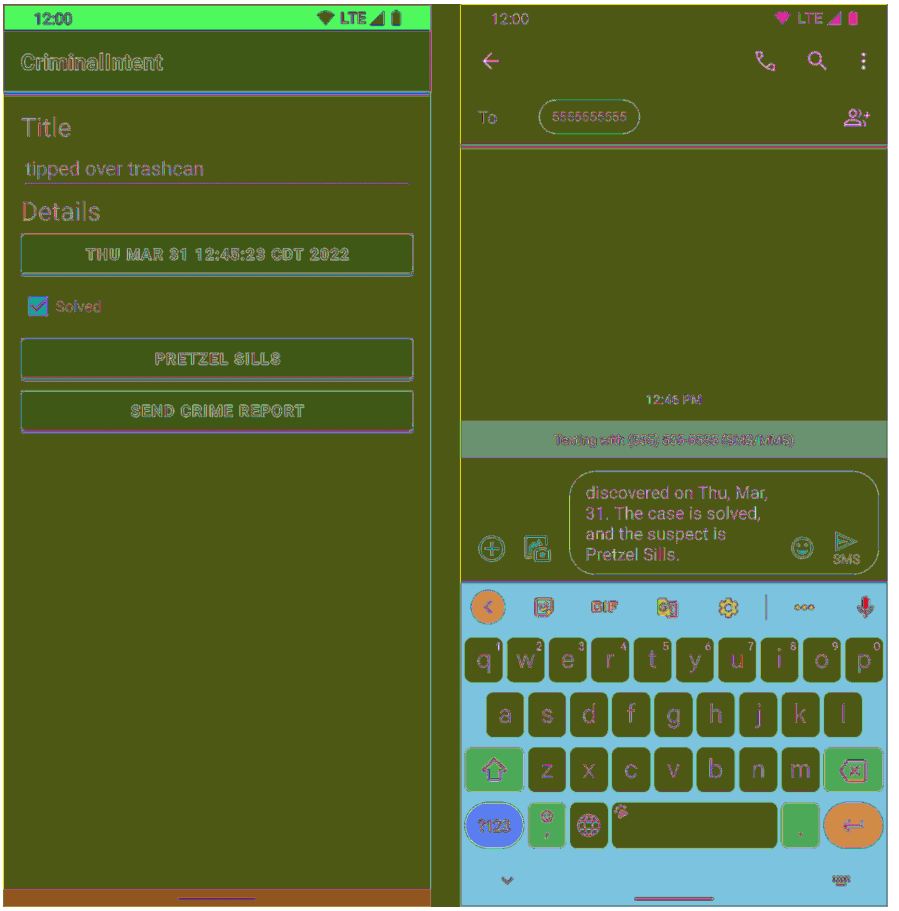
But there is one more step: You need to call **`parseContactSelection(Uri)`** when you get a result back. Invoke it when calling **`registerForActivityResult(…)`**.

### Listing 16.14 Invoking your function (`CrimeDetailFragment.kt`)

```
class CrimeDetailFragment : Fragment() {
    ...
    private val selectSuspect = registerForActivityResult(
        ActivityResultContracts.PickContact()
    ) { uri: Uri? ->
        // Handle the result
        uri?.let { parseContactSelection(it) }
    }
    ...
}
```

Run your app, select a crime, and pick a suspect. The name of the suspect you chose should appear on the CHOOSE SUSPECT button. Then send a crime report. The suspect's name should appear in the crime report (Figure 16.6).

### Figure 16.6 Suspect name on button and in crime report



### Contacts permissions

You might be wondering how you are getting permission to read from the contacts database. The contacts app is extending its permissions to you.

The contacts app has full permissions to the contacts database. When the contacts app returns a data URI as the result, it also adds the flag `Intent.FLAG_GRANT_READ_URI_PERMISSION`. This flag signals to Android that CriminalIntent should be allowed to use this data one time. This works well, because you do not really need access to the entire contacts database. You only need access to one contact inside that database.

## Checking for responding activities

The first implicit intent you created in this chapter will always be responded to in some way – there may be no way to send a report, but the chooser will still display properly. However, that is not the case for the second example: Some devices or users may not have a contacts app. This is a problem, because if the OS cannot find a matching activity, then the app will crash.

To determine whether your user has an appropriate contacts application, you will need to query the system to determine which activities will respond to your implicit intent. If one or more activities are returned, then your user is all set to pick a contact. If no activities come back, then the user does not have an appropriate contact picker and the functionality should be disabled in CriminalIntent.

### Disclosing queries

To successfully make that query, you must first disclose that you are going to make it. This provides extra security for users, because apps have to declare what types of external requests they make to the system. In the past, scummy apps would abuse the ability to query for apps in order to "fingerprint," or uniquely identify, the device. Those apps could then use that fingerprint to track that device across apps.

To prevent this invasion of privacy, you provide this disclosure within your `AndroidManifest.xml`. Open the file and make the following updates:

Listing 16.15 Adding external queries to manifest (`AndroidManifest.xml`)

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

    <application ...>
        ...
    </application>
    <queries>
        <intent>
            <action android:name="android.intent.action.PICK" />
            <data android:mimeType="vnd.android.cursor.dir/contact" />
        </intent>
    </queries>
</manifest>
```

The `queries` block at the end of the manifest includes all the external intents the app is going to look up. Because CriminalIntent wants to check for a contacts app, the relevant intent information is provided so the system is aware. If you do not provide this disclosure, then on newer versions of Android the system will always tell you that no activities can handle your request.

## Querying the PackageManager

Now that you have provided your disclosure, you can determine whether the OS can handle your request through the **`PackageManager`** class. **`PackageManager`** knows about all the components installed on an Android device, including all its activities. By calling **`resolveActivity(Intent, Int)`**, you can ask it to find an activity that matches the **`Intent`** you gave it. The `MATCH_DEFAULT_ONLY` flag restricts this search to activities with the `CATEGORY_DEFAULT` flag, just like **`startActivity(Intent)`** does.

If this search is successful, it will return an instance of **`ResolveInfo`** telling you all about which activity it found. On the other hand, if the search returns `null`, the game is up – no app can handle your **`Intent`**. You can use this knowledge to enable or disable features, such as selecting a suspect from the list of contacts, based on whether the system can handle the request.

Add the **`canResolveIntent()`** function to the bottom of **`CrimeDetailFragment`**. It will take in an **`Intent`** and return a Boolean indicating whether that **`Intent`** can be resolved.

### Listing 16.16 Resolving **`Intent`**s (`CrimeDetailFragment.kt`)

```
class CrimeDetailFragment : Fragment() {
    ...
    private fun parseContactSelection(contactUri: Uri) {
        ...
    }

    private fun canResolveIntent(intent: Intent): Boolean {
        val packageManager: PackageManager = requireActivity().packageManager
        val resolvedActivity: ResolveInfo? =
            packageManager.resolveActivity(
                intent,
                PackageManager.MATCH_DEFAULT_ONLY
            )
        return resolvedActivity != null
    }
}
```

Under the hood, the Activity Results API uses **`Intent`**s to perform its actions. You can create an instance of those **`Intent`**s by invoking **`createIntent()`** on the launcher's `contract` property. Use your new **`canResolveIntent()`** function with the **`Intent`** backing the `selectSuspect` property to enable or disable the suspect button in **`onViewCreated(…)`**. That way, the device will not crash if the user tries to select a suspect when the device does not have a contacts app.

### Listing 16.17 Guarding against no contacts app (`CrimeDetailFragment.kt`)

```
class CrimeDetailFragment : Fragment() {
    ...
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        binding.apply {
            ...
            crimeSuspect.setOnClickListener {
                selectSuspect.launch(null)
            }

            val selectSuspectIntent = selectSuspect.contract.createIntent(
                requireContext(),
                null
            )
            crimeSuspect.isEnabled = canResolveIntent(selectSuspectIntent)
        }
    }
    ...
}
```

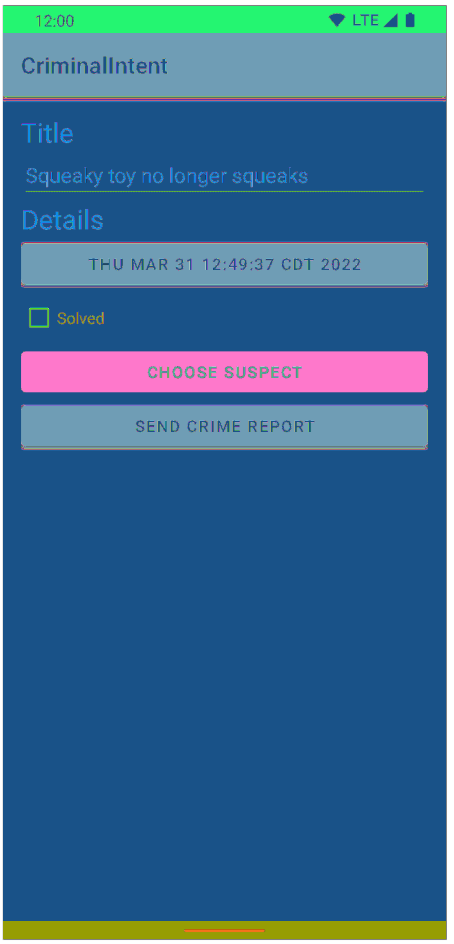
If you would like to verify that your filter works, but you do not have a device without a contacts application, temporarily add an additional category to the intent trying to be resolved. This category does nothing, but it will prevent any contacts applications from matching your intent.

### Listing 16.18 Adding dummy code to verify filter (`CrimeDetailFragment.kt`)

```
class CrimeDetailFragment : Fragment() {
    ...
    private fun canResolveIntent(intent: Intent): Boolean {
        intent.addCategory(Intent.CATEGORY_HOME)
        val packageManager: PackageManager = requireActivity().packageManager
        val resolvedActivity: ResolveInfo? =
            packageManager.resolveActivity(
                intent,
                PackageManager.MATCH_DEFAULT_ONLY
            )
        return resolvedActivity != null
    }
}
```

Run CriminalIntent again, and you should see the CHOOSE SUSPECT button disabled (Figure 16.7).

Figure 16.7 Disabled CHOOSE SUSPECT button




Delete the dummy code once you are done verifying this behavior.

Listing 16.19 Deleting dummy code (`CrimeDetailFragment.kt`)

```
class CrimeDetailFragment : Fragment() {
    ...
    private fun canResolveIntent(intent: Intent): Boolean {
        ~~intent.addCategory(Intent.CATEGORY_HOME)~~
        val packageManager: PackageManager = requireActivity().packageManager
        val resolvedActivity: ResolveInfo? =
            packageManager.resolveActivity(
                intent,
                PackageManager.MATCH_DEFAULT_ONLY
            )
        return resolvedActivity != null
    }
}
```

## Challenge: Another Implicit Intent

Instead of sending a crime report, an angry user may prefer a phone confrontation with the suspect. Add a new button that calls the named suspect.

You will need the phone number from the contacts database. This will require you to query another table in the **`ContactsContract`** database called **`CommonDataKinds.Phone`**. Check out the documentation for **`ContactsContract`** and **`ContactsContract.CommonDataKinds.Phone`** for more information on how to query for this information.

A couple of tips: To query for additional data, you can use the `android.permission.READ_CONTACTS` permission. This is a *runtime permission*, so you need to explicitly ask the user's permission to access their contacts. If you would like to know more, read the Request App Permissions guide at `developer.android.com/training/permissions/requesting`.

With that permission in hand, you can read the `ContactsContract.Contacts._ID` to get a contact ID on your original query. You can then use that ID to query the **`CommonDataKinds.Phone`** table.

Once you have the phone number, you can create an implicit intent with a telephone URI:

```
Uri number = Uri.parse("tel:5551234");
```

The action can be `Intent.ACTION_DIAL` or `Intent.ACTION_CALL`. What is the difference? `ACTION_CALL` pulls up the phone app and immediately calls the number sent in the intent; `ACTION_DIAL` just enters the number and waits for the user to initiate the call.

We recommend using `ACTION_DIAL`. It is the kinder, gentler option. `ACTION_CALL` may be restricted and will definitely require a permission. Your user may also appreciate the chance to cool down before starting the call.

# 17

# Taking Pictures with Intents

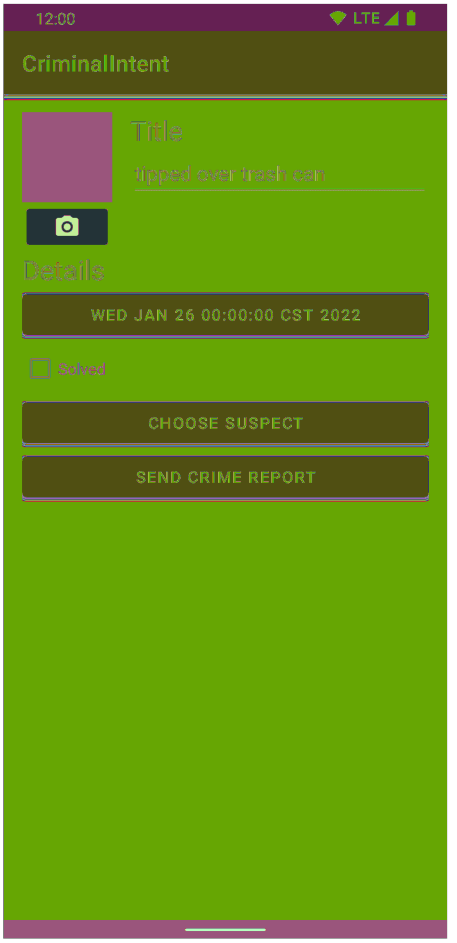
Now that you know how to work with implicit intents, you can document crimes in even more detail. With a picture of the crime, you can share the gory details with everyone. Taking a picture will involve a couple of new tools used in combination with a tool you recently got to know: the implicit intent.

An implicit intent can be used to start up the user's favorite camera application and receive a new picture from it. But where do you put the picture the camera takes? And once the picture comes in, how do you display it? In this chapter, you will answer both of those questions.

## A Place for Your Photo

The first step for this chapter is to build out a place for your photo to live on the crime detail screen. You will need two new **`View`** objects: an **`ImageView`** to display the photo and a **`Button`** to press to take a new photo (Figure 17.1).

Figure 17.1 New UI

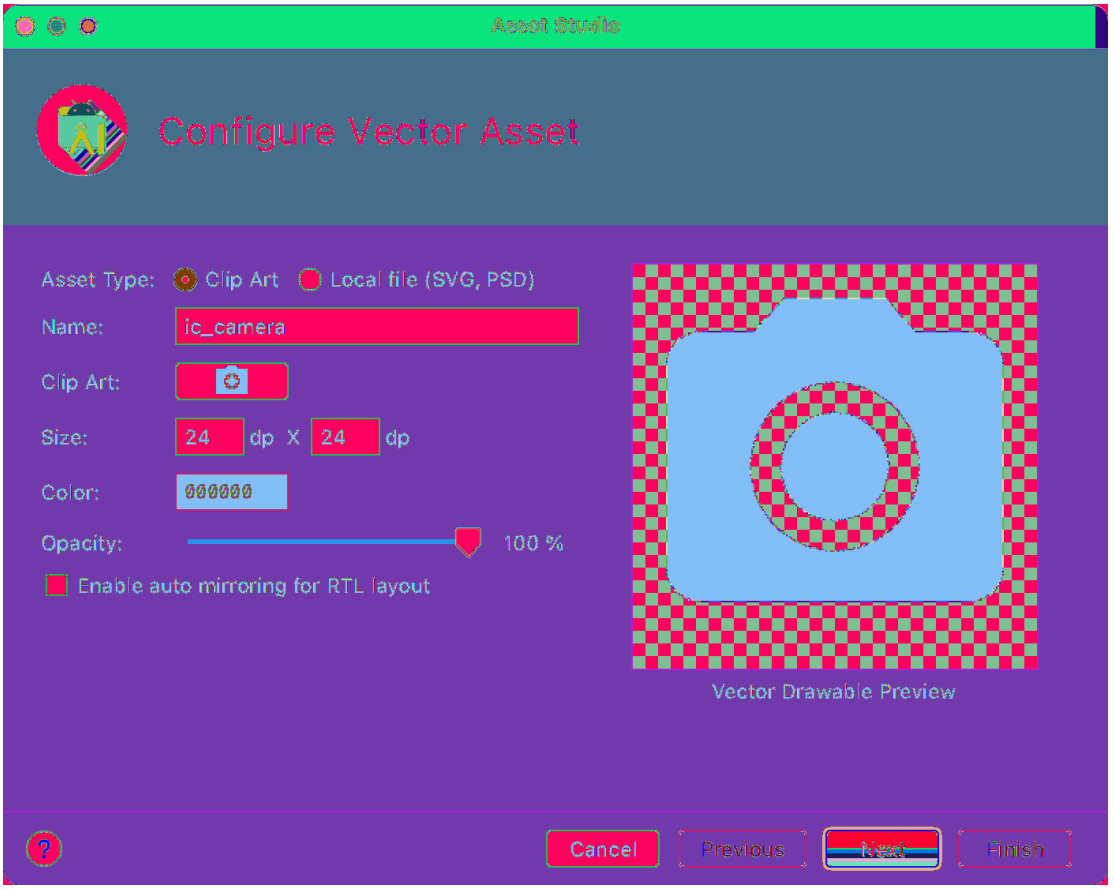



Dedicating an entire row to a thumbnail and a button would make your app look clunky and unprofessional. You do not want that, so you will arrange things nicely. You will put the picture of the crime and the button to take the photo alongside the title.

Instead of using text to label your new button, this time you will label it with an icon. As with previous icons you have used for buttons and the app bar, you will use a vector asset. Select File → New → Vector Asset from Android Studio's menu bar to bring up the Asset Studio. Click the + button to the right of the Clip Art: label.

Within the Select Icon window, search for photo camera and select the first icon. Rename the asset to `ic_camera` (Figure 17.2). With that done, click Next, and then Finish on the following screen to add the icon to your project.

Figure 17.2 Your camera icon



With the icon now in the project, add new views to `res/layout/fragment_crime_detail.xml` to build out this new area. Start with the lefthand side, adding an **`ImageView`** for the picture and an **`ImageButton`** to take a picture.

### Listing 17.1 Adding an image and camera button to the layout (`res/layout/fragment_crime_detail.xml`)

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
              ... >
    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal">

        <LinearLayout
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:orientation="vertical"
            android:layout_marginEnd="16dp">

            <ImageView
                android:id="@+id/crime_photo"
                android:layout_width="80dp"
                android:layout_height="80dp"
                android:scaleType="centerInside"
                android:cropToPadding="true"
                android:background="@color/black"/>

            <ImageButton
                android:id="@+id/crime_camera"
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:src="@drawable/ic_camera"/>
        </LinearLayout>
    </LinearLayout>

    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:textAppearance="?attr/textAppearanceHeadline5"
        android:text="@string/crime_title_label" />
    ...
</LinearLayout>
```

Now set up the righthand side, moving your title **`TextView`** and **`EditText`** into a new **`LinearLayout`** child to the **`LinearLayout`** you just built.

Listing 17.2 Updating the title layout (`res/layout/fragment_crime.xml`)

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
              ... >
    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal">

        <LinearLayout
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:orientation="vertical"
            android:layout_marginEnd="16dp">
                ...
        </LinearLayout>
    </LinearLayout>

        <LinearLayout
            android:orientation="vertical"
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_weight="1">

            <TextView
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:textAppearance="?attr/textAppearanceHeadline5"
                android:text="@string/crime_title_label" />

            <EditText
                android:id="@+id/crime_title"
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:importantForAutofill="no"
                android:hint="@string/crime_title_hint"
                android:inputType="text" />
        </LinearLayout>
    </LinearLayout>
    ...
</LinearLayout>
```

Run CriminalIntent and press a crime to see its details. You should see your new UI looking just like Figure 17.1.

And with that, you are done with the UI for the time being. (You will wire those buttons up in a minute or two.)

## File Storage

Your photo needs more than a place on the screen. Full-size pictures are too large to stick inside a SQLite database, much less an **`Intent`**. They will need a place to live on your device's filesystem.

Luckily, you have a place to stash these files: your private storage. Each application on an Android device has a directory in the device's *sandbox*. Keeping files in the sandbox protects them from being accessed by other applications or even the prying eyes of users (unless the device has been rooted, in which case the user can get to whatever they like).

Your crime database is actually a file within this private sandbox. The Room library knows how to find and access this file to provide you with a working database that persists across app launches. With functions like **`Context.getFileStreamPath(String)`** and **`Context.getFilesDir()`**, you can do the same thing with regular files, too (which will live in a subfolder adjacent to the `databases` subfolder your database lives in).

These are the basic file and directory functions in the **`Context`** class:

**`getFilesDir()`**: **`File`**

> returns a handle to the directory for private application files

**`openFileInput(name: String)`**: **`FileInputStream`**

> opens an existing file in the files directory for input

**`openFileOutput(name: String, mode: Int)`**: **`FileOutputStream`**

> opens a file in the files directory for output, possibly creating it

**`getDir(name: String, mode: Int)`**: **`File`**

> gets (and possibly creates) a subdirectory within the files directory

**`fileList(…)`**: **`Array<String>`**

> gets a list of filenames in the main files directory, such as for use with **`openFileInput(String)`**

**`getCacheDir()`**: **`File`**

> returns a handle to a directory you can use specifically for storing cache files; you should take care to keep this directory tidy and use as little space as possible

There is a catch. Because these files are private, *only your own application* can read or write to them. As long as no other app needs to access those files, these functions are sufficient.

However, they are not sufficient if another application needs to write to your files. This is the case for CriminalIntent, because the external camera app will need to save the picture it takes as a file in your app.

In those cases, the functions above do not go far enough: While there is a `Context.MODE_WORLD_READABLE` flag you can pass into **`openFileOutput(…)`**, it is deprecated and not completely reliable in its effects on newer devices. Once upon a time you could also transfer files using publicly accessible external storage, but this has been locked down in recent versions of Android for security reasons.

If you need to share files with or receive files from other apps, you need to expose those files through a **`ContentProvider`**. A **`ContentProvider`** allows you to expose content URIs to other apps. They can then download from or write to those content URIs. Either way, you are in control and always have the option to deny those reads or writes if you so choose.

## Using FileProvider

When all you need to do is receive a file from another application, implementing an entire **`ContentProvider`** is overkill. Fortunately, Google provides a convenience class called **`FileProvider`**. **`FileProvider`** extends the **`ContentProvider`** class and is designed to easily and securely share files between apps. Instead of implementing all the methods required for the **`ContentProvider`** class, you can just configure a **`FileProvider`** and have it do the rest of the work.

The first step is to declare **`FileProvider`** as a **`ContentProvider`** hooked up to a specific *authority*. Do this by adding a content provider declaration to your Android manifest.

### Listing 17.3 Adding a **`FileProvider`** declaration (`manifests/AndroidManifest.xml`)

```
<activity android:name=".MainActivity">
    ...
</activity>
<provider
    android:name="androidx.core.content.FileProvider"
    android:authorities="com.bignerdranch.android.criminalintent.fileprovider"
    android:exported="false"
    android:grantUriPermissions="true">
</provider>
...
```

The authority is a location – a place that files will be saved to. The string you choose for `android:authorities` must be unique across the entire OS. To help ensure this, the convention is to prepend the authority string with your package name. (We show the package name `com.bignerdranch.android.criminalintent` above. If your app's package name is different, use your package name instead.)

Classes that extend the **`ContentProvider`** class are often used to share content between apps. But you need to be careful about what you share. Your users place a lot of trust in you with their information. You do not want to inadvertently expose their data to the entire world.

The **`FileProvider`** class helps you carefully and intentionally share data with other apps. It requires a certain configuration so that you only expose content when you intend to. By using the `exported="false"` attribute, you keep it from being publicly visible to random applications querying the system.

When you do want to expose some content to the larger system, the `grantUriPermissions` attribute gives you the ability to temporarily grant other apps permission to write to URIs on this authority when you send them out in an intent.

Now that you have told Android where your **FileProvider** is, you also need to tell your **FileProvider** which files it is exposing. This bit of configuration is done with an XML resource file. Right-click your app/res folder in the project tool window and select New → Android resource file. Enter files for the name, and for Resource type select XML. Click OK, and Android Studio will add and open the new resource file.

In the code view of your new `res/xml/files.xml`, replace the boilerplate code with details about the file path (Listing 17.4).

### Listing 17.4 Filling out the paths description (`res/xml/files.xml`)

```
<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android">

</PreferenceScreen>
<paths>
    <files-path name="crime_photos" path="."/>
</paths>
```

This XML file says, "Map the root path of my private storage as `crime_photos`." You will not use the `crime_photos` name – **FileProvider** uses that internally.

Now, hook up `files.xml` to your **FileProvider** by adding a `meta-data` tag in your `AndroidManifest.xml`.

### Listing 17.5 Hooking up the paths description (`manifests/AndroidManifest.xml`)

```
<provider
    android:name="androidx.core.content.FileProvider"
    android:authorities="com.bignerdranch.android.criminalintent.fileprovider"
    android:exported="false"
    android:grantUriPermissions="true">
    <meta-data
        android:name="android.support.FILE_PROVIDER_PATHS"
        android:resource="@xml/files"/>
</provider>
```

## Designating a picture location

Now you have a place to store photos on the device. Next, you need to add a place to store the photo's filename within a **`Crime`**. Start by adding a new property to **`Crime`** to store the photo's filename.

### Listing 17.6 Adding the filename property (`Crime.kt`)

```
@Entity
data class Crime(
    @PrimaryKey val id: UUID,
    val title: String,
    val date: Date,
    val isSolved: Boolean,
    val suspect: String = "",
    val photoFileName: String? = null
)
```

Next, since you have added a new property to your **`Crime`** class, create a migration for this new property in the database and increment the version.

### Listing 17.7 Migrating the database (`CrimeDatabase.kt`)

```
@Database(entities = [Crime::class], version = 2 version = 3)
@TypeConverters(CrimeTypeConverters::class)
abstract class CrimeDatabase : RoomDatabase() {
    abstract fun crimeDao(): CrimeDao
}

val migration_1_2 = object : Migration(1, 2) {
    ...
}

val migration_2_3 = object : Migration(2, 3) {
    override fun migrate(database: SupportSQLiteDatabase) {
        database.execSQL(
            "ALTER TABLE Crime ADD COLUMN photoFileName TEXT"
        )
    }
}
```

Finally, include that migration when creating the database in **`CrimeRepository`**.

### Listing 17.8 Including the migration (`CrimeRepository.kt`)

```
class CrimeRepository private constructor(
    context: Context,
    private val coroutineScope: CoroutineScope = GlobalScope
) {

    private val database: CrimeDatabase = Room
        .databaseBuilder(
            context.applicationContext,
            CrimeDatabase::class.java,
            DATABASE_NAME
        )
        .addMigrations(migration_1_2, migration_2_3)
        .build()
}
```

## Using a Camera Intent

The basic process to take a photo is relatively straightforward: You launch an external camera app, the user takes a photo, and then you update the crime with the path to the new file. To implement this, you are once again going to rely on the Activity Results API.

This time you are going to use the **`ActivityResultContracts.TakePicture()`** contract. It takes in a **`Uri`**, which will be generated by the **`FileProvider`** class using a **`File`** you will create. Once the user is finished taking the photo, the contract does not return that same **`Uri`**. Instead, it returns a **`Boolean`** telling you whether an image was saved to the file.

Create a class property on **`CrimeDetailFragment`** named `takePhoto` and initialize it using the Activity Results API. Leave the lambda expression that is invoked once there is a result empty for now.

### Listing 17.9 Setting up your activity result (`CrimeDetailFragment.kt`)

```
class CrimeDetailFragment : Fragment() {
    ...
    private val selectSuspect = registerForActivityResult(
        ActivityResultContracts.PickContact()
    ) { uri: Uri? ->
        uri?.let { parseContactSelection(it) }
    }

    private val takePhoto = registerForActivityResult(
        ActivityResultContracts.TakePicture()
    ) { didTakePhoto: Boolean ->
        // Handle the result
    }
}
```

To invoke the `takePhoto` launcher, you need to create a shareable **`Uri`**. Creating this variable takes a few steps. First, you create a string that holds the filename where the photo will be stored. Because you do not want to accidentally overwrite an existing file, the string will include a timestamp representing when the photo was taken.

With that filename, you create a **`File`** that is stored within the app's internal storage. Finally, you call the **`FileProvider.getUriForFile(…)`** function, and that will translate your local file path into a **`Uri`** the camera app can see. The function takes in your activity, provider authority, and photo file to create the URI that points to the file. The authority string you pass to **`FileProvider.getUriForFile(…)`** must match the authority string you defined in the manifest in Listing 17.3.

Create these variables and launch the `takePhoto` property with the new **`Uri`** inside the click listener for the **`ImageButton`**.

### Listing 17.10 Launching the camera app (`CrimeDetailFragment.kt`)

```
class CrimeDetailFragment : Fragment() {
    ...
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        binding.apply {
            ...
            crimeSuspect.isEnabled = canResolveIntent(selectSuspectIntent)

            crimeCamera.setOnClickListener {
                val photoName = "IMG_${Date()}.JPG"
                val photoFile = File(requireContext().applicationContext.filesDir,
                                     photoName)
                val photoUri = FileProvider.getUriForFile(
                    requireContext(),
                    "com.bignerdranch.android.criminalintent.fileprovider",
                    photoFile
                )

                takePhoto.launch(photoUri)
            }
        }
    }
    ...
}
```

Run the app and try to take a photo. You should be able to launch a camera app from your crime detail screen. (The emulator has a photo app, so you can try it even if you are not connected to a device.) This is progress! But you are not yet updating the crime or displaying the photo. Let's update the crime first.

In the lambda expression of the `takePhoto` property, update your crime for a successful picture capture. You only want to update the crime when the photo is taken, so you will use the **`Boolean`** that is passed into your lambda expression to check this.

The `photoName` string you defined when launching the camera app is the value you want to update your crime with. Since you need access to it after taking the photo, make it a class property instead of just a variable. Also, make it nullable, so that when the property is set you can be confident that the user took a photo.

### Listing 17.11 Handling the result (`CrimeDetailFragment.kt`)

```
class CrimeDetailFragment : Fragment() {
    ...
    private val takePhoto = registerForActivityResult(
        ActivityResultContracts.TakePicture()
    ) { didTakePhoto ->
        // Handle the result
        if (didTakePhoto && photoName != null) {
            crimeDetailViewModel.updateCrime { oldCrime ->
                oldCrime.copy(photoFileName = photoName)
            }
        }
    }

    private var photoName: String? = null

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        ...
    }

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        binding.apply {
            ...
            crimeCamera.setOnClickListener {
                val photoName = "IMG_${Date()}.JPG"
                val photoFile = File(requireContext().applicationContext.filesDir,
                                     photoName)
                ...
            }
        }
    }
}
```

There is one small piece of housekeeping you need to take care of before you can display that photo. Just as you cannot be sure a device has a contacts app, you cannot guarantee a device has a camera app. So, similar to what you did in Chapter 16, you need to disable the camera button if your implicit intent cannot be resolved.

This time, you will generate an **`Intent`** based on the contract for the `takePhoto` property. Since you are not going to launch an activity with this **`Intent`**, you can pass an empty **`Uri`** as the input. Reuse the **`canResolveIntent()`** function to disable the camera button if there is no activity on the system that can take a picture for you.

### Listing 17.12 Disabling the camera button (`CrimeDetailFragment.kt`)

```
class CrimeDetailFragment : Fragment() {
    ...
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        binding.apply {
            ...
            crimeCamera.setOnClickListener {
                ...
            }

            val captureImageIntent = takePhoto.contract.createIntent(
                requireContext(),
                Uri.parse("")
            )
            crimeCamera.isEnabled = canResolveIntent(captureImageIntent)
        }
    }
}
```

You also need to add a query intent to the manifest to allow CriminalIntent to query for camera applications.

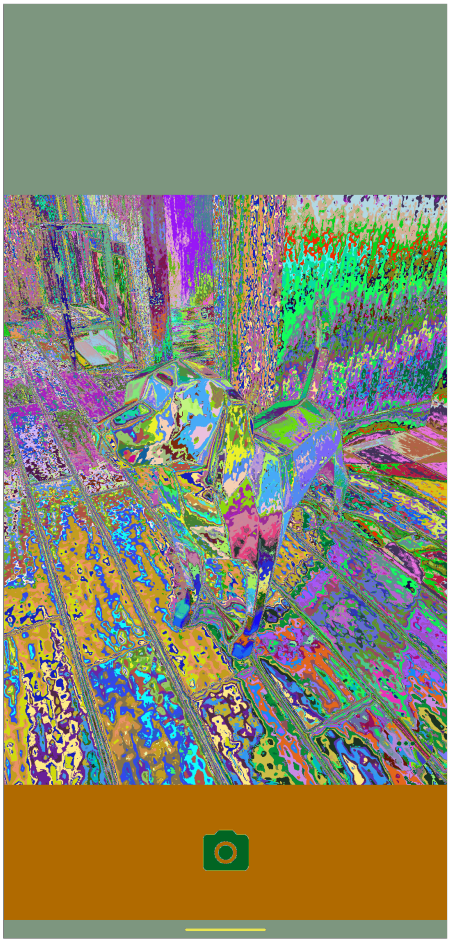
### Listing 17.13 Adding another query declaration (`AndroidManifest.xml`)

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.bignerdranch.android.criminalintent">

    <application ...>
        ...
    </application>
    <queries>
        <intent>
            <action android:name="android.intent.action.PICK" />
            <data android:mimeType="vnd.android.cursor.dir/contact" />
        </intent>
        <intent>
            <action android:name="android.media.action.IMAGE_CAPTURE" />
        </intent>
    </queries>
</manifest>
```

Run CriminalIntent and press the camera button to run your camera app (Figure 17.3). You can now take a picture, but you still have some work to do to display it.

Figure 17.3 [Insert your camera app here]



## Scaling and Displaying Bitmaps

You are successfully taking pictures, and your image will be saved to a file on the filesystem for you to use.

Your next step is to take this file, load it, and show it to the user. To do this, you need to load it into a reasonably sized **`Bitmap`** object. To get a **`Bitmap`** from a file, all you need to do is use the **`BitmapFactory`** class:

```
val bitmap = BitmapFactory.decodeFile(photoFile.getPath())
```

There has to be a catch, right? Otherwise we would have put that in bold, you would have typed it in, and you would be done.

Here is the catch: When we say "reasonably sized," we mean it. A **`Bitmap`** is a simple object that stores literal pixel data. That means that even if the original file were compressed, there would be no compression in the **`Bitmap`**. So a 16-megapixel, 24-bit camera image – which might only be a 5 MB JPG – would blow up to 48 MB loaded into a **`Bitmap`** object (!).

You can get around this, but it does mean that you will need to scale the bitmap down by hand. You will first scan the file to see how big it is, next figure out how much you need to scale it by to fit it in a given area, and finally reread the file to create a scaled-down **`Bitmap`** object.

Create a new file called `PictureUtils.kt` and add a file-level function to it called **`getScaledBitmap(String, Int, Int)`**:

### Listing 17.14 Creating **`getScaledBitmap(…)`** (`PictureUtils.kt`)

```
fun getScaledBitmap(path: String, destWidth: Int, destHeight: Int): Bitmap {
    // Read in the dimensions of the image on disk
    val options = BitmapFactory.Options()
    options.inJustDecodeBounds = true
    BitmapFactory.decodeFile(path, options)

    val srcWidth = options.outWidth.toFloat()
    val srcHeight = options.outHeight.toFloat()

    // Figure out how much to scale down by
    val sampleSize = if (srcHeight <= destHeight && srcWidth <= destWidth) {
        1
    } else {
        val heightScale = srcHeight / destHeight
        val widthScale = srcWidth / destWidth

        minOf(heightScale, widthScale).roundToInt()
    }

    // Read in and create final bitmap
    return BitmapFactory.decodeFile(path, BitmapFactory.Options().apply {
        inSampleSize = sampleSize
    })
}
```

The key parameter in this code is `sampleSize`. This determines how big the "sample" should be for each pixel – a sample size of `1` has one final horizontal pixel for each horizontal pixel in the original file, and a sample size of `2` has one horizontal pixel for every two horizontal pixels in the original file. So when `sampleSize` is `2`, the pixel count in the image is one-quarter of the pixel count in the original.

But when your fragment initially starts up, you will not know how big the **`PhotoView`** is. Until a layout pass happens, views do not have dimensions onscreen. This used to be a difficult problem to solve, but now, with the **`doOnLayout()`** extension function, you can easily wait for a **`View`** to be measured and laid out so you can do delicate UI work using exact layout measurements.

Add a function named **updatePhoto()** to **CrimeDetailFragment** that uses **doOnLayout()** to display your image at a reasonable resolution.

### Listing 17.15 Updating crimePhoto (CrimeDetailFragment.kt)

```
class CrimeDetailFragment : Fragment() {
    ...
    private fun canResolveIntent(intent: Intent): Boolean {
      ...
    }

    private fun updatePhoto(photoFileName: String?) {
      if (binding.crimePhoto.tag != photoFileName) {
          val photoFile = photoFileName?.let {
              File(requireContext().applicationContext.filesDir, it)
          }

          if (photoFile?.exists() == true) {
              binding.crimePhoto.doOnLayout { measuredView ->
                  val scaledBitmap = getScaledBitmap(
                      photoFile.path,
                      measuredView.width,
                      measuredView.height
                  )
                  binding.crimePhoto.setImageBitmap(scaledBitmap)
                  binding.crimePhoto.tag = photoFileName
              }
          } else {
              binding.crimePhoto.setImageBitmap(null)
              binding.crimePhoto.tag = null
          }
      }
  }
}
```

This **updatePhoto()** function will be invoked every time you get a new emission from the `crime` **StateFlow** from your **CrimeDetailViewModel**. But you do not want to read the photo from disk every time the user adds a character into the crime's title! That would be wildly inefficient and would probably lead to a stuttering UI.

To update the **ImageView** only when necessary, you set the `tag` property on the view. The `tag` property allows you to store simple information on a particular view. Here, you set the filename of the photo. If the `tag` property and the crime's photo filename match, then you know the **ImageView** is already displaying the correct photo.

Call the new **updatePhoto()** function when you have access to the latest value from the `crime` **StateFlow**. That way your **ImageView** will always display the latest photo from the crime scene.

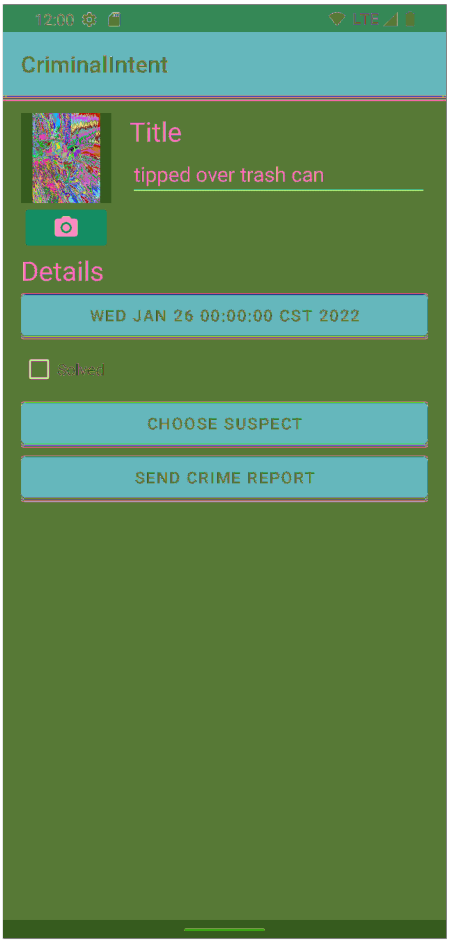
Listing 17.16 Updating the **`Crime`** with the latest photo (`CrimeDetailFragment.kt`)

```
class CrimeDetailFragment : Fragment() {
    ...
    private fun updateUi(crime: Crime) {
        binding.apply {
            ...
            crimeSuspect.text = crime.suspect.ifEmpty {
                getString(R.string.crime_suspect_text)
            }

            updatePhoto(crime.photoFileName)
        }
    }
    ...
}
```

Run CriminalIntent again. Open a crime's detail screen and use the camera button to take a photo. You should see your image displayed in the thumbnail view (Figure 17.4).

Figure 17.4 Thumbnail proudly appearing on the crime detail screen



## Declaring Features

Your camera implementation works great now. One more task remains: Tell potential users about it. When your app uses a feature like the camera – or near-field communication, or any other feature that may vary from device to device – it is strongly recommended that you tell Android about it. This allows other apps (like the Play Store) to refuse to install your app if it uses a feature the device does not support.

To declare that you use the camera, add a `<uses-feature>` tag to your manifest.

### Listing 17.17 Adding a `<uses-feature>` tag (`manifests/AndroidManifest.xml`)

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.bignerdranch.android.criminalintent" >

    <uses-feature android:name="android.hardware.camera"
                  android:required="false"/>
    ...
</manifest>
```

You include the optional attribute `android:required` here. Why? By default, declaring that you use a feature means that your app will not work correctly without that feature. This is not the case for CriminalIntent. You call **`resolveActivity(…)`** to check for a working camera app, then gracefully disable the camera button if you do not find one.

Passing in `android:required="false"` handles this situation correctly. You tell Android that your app can work fine without the camera, but that some parts will be disabled as a result.

## Challenge: Detail Display

While you can certainly see the image you display here, you cannot see it very well.

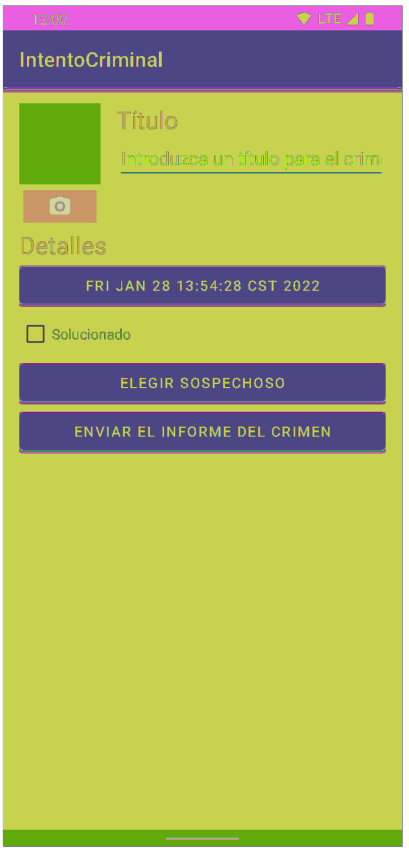
For this challenge, create a new **`DialogFragment`** that displays a zoomed-in version of your crime scene photo. When you press the thumbnail, it should pull up the zoomed-in **`DialogFragment`**.

# 18
# Localization

Knowing that CriminalIntent is going to be a wildly popular app, you have decided to make it accessible to a larger audience. Your first step is to *localize* all the user-facing text so your app can be read in Spanish or English.

Localization is the process of providing the appropriate resources for your app based on the user's language setting. In this chapter, you will provide a Spanish version of `res/values/strings.xml`. When a device's language is set to Spanish, Android will automatically find and use the Spanish strings at runtime (Figure 18.1).

Figure 18.1 IntentoCriminal



## Localizing Resources

Language settings are part of the device's configuration, like the screen orientation configuration you have encountered. Android provides qualifiers for different languages just as it does for screen orientation, screen size, and other configuration factors. This makes localization straightforward: You create resource subdirectories with the desired language configuration qualifier and put the alternative resources in them. The Android resource system does the rest.

In your CriminalIntent project, create a new values resource file: In the project tool window, right-click `res/values/` and select New → Values resource file. Enter strings for the File name. Leave the Source set option set to main and make sure Directory name is set to values.

Next, select Locale in the Available qualifiers list and click the >> button to move Locale to the Chosen qualifiers section. Select es: Spanish in the Language list. Any Region will be automatically selected in the Specific Region Only list – which is just what you want, so leave that selection be.

The resulting New Resource File window should look similar to Figure 18.2.

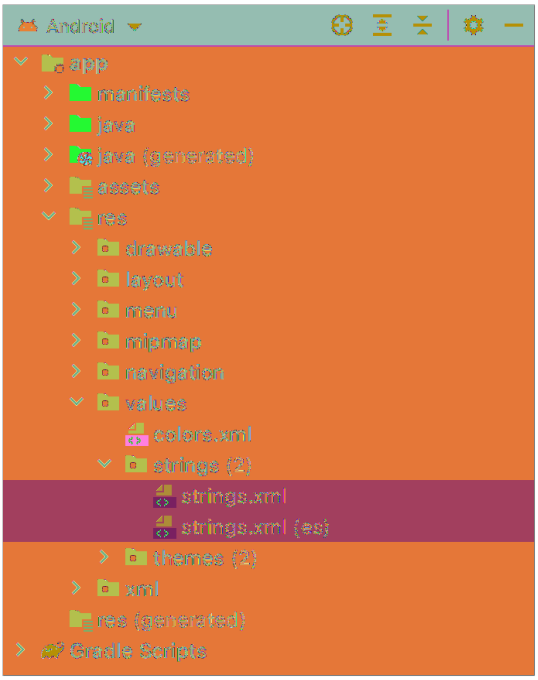
Figure 18.2 Adding a qualified strings resource file




Android Studio automatically changes the Directory name field to values-es. The language configuration qualifiers are taken from ISO 639-1 codes, and each consists of two characters. For Spanish, the qualifier is `–es`.

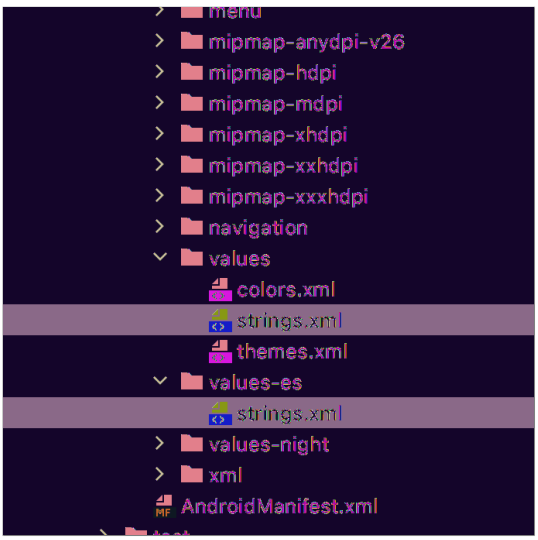
Click OK. The new `strings.xml` file will be listed under `res/values`, with `(es)` after its name. The strings files are grouped in the project tool window's Android view (Figure 18.3).

Figure 18.3 Viewing new `strings.xml` in Android view




However, if you explore the directory structure, you will see that your project now contains an additional values directory: `res/values-es`. The newly generated `strings.xml` is in this new directory (Figure 18.4).

Figure 18.4 Viewing new `strings.xml` in Project view



Now it is time to make the magic happen. Add Spanish versions of all your strings to `res/values-es/strings.xml`. (If you do not want to type these strings in, copy the contents from the solutions file at `www.bignerdranch.com/android-5e-solutions`.)

### Listing 18.1 Adding Spanish alternatives for string resources (`res/values-es/strings.xml`)

```
<resources>
    <string name="app_name">IntentoCriminal</string>
    <string name="crime_title_hint">Introduzca un título para el crimen.</string>
    <string name="crime_title_label">Título</string>
    <string name="crime_details_label">Detalles</string>
    <string name="crime_solved_label">Solucionado</string>
    <string name="new_crime">Crimen Nuevo</string>
    <string name="crime_suspect_text">Elegir Sospechoso</string>
    <string name="crime_report_text">Enviar el Informe del Crimen</string>
    <string name="crime_report">%1$s!
        El crimen fue descubierto el %2$s. %3$s, y %4$s
    </string>
    <string name="crime_report_solved">El caso está resuelto</string>
    <string name="crime_report_unsolved">El caso no está resuelto</string>
    <string name="crime_report_no_suspect">no hay sospechoso.</string>
    <string name="crime_report_suspect">el/la sospechoso/a es %s.</string>
    <string name="crime_report_subject">IntentoCriminal Informe del Crimen</string>
    <string name="send_report">Enviar el informe del crimen a través de</string>
</resources>
```

That is all you have to do to provide localized string resources for your app. To confirm, change your device's settings to Spanish by opening Settings and finding the language settings. Depending on your version of Android, these settings will be labeled Language and input, Language and Keyboard, or something similar. On the Pixel 4 emulator, the Languages & input settings are within the System section.

When you get to a list of language options, choose a setting for Español. The region (España or Estados Unidos) will not matter, because the qualification `-es` matches both. (On newer versions of Android, users can select multiple languages and assign a priority order. If you are on a newer device, make sure Español appears first in your language settings list.)

Now run CriminalIntent and bask in the glory of your newly localized app. When you are done basking, return your device's language setting to English. Look for Ajustes or Configuración (Settings) in the launcher and find the setting that includes Idioma (Language).

## Default resources

The configuration qualifier for English is `-en`. In a fit of localization, you might think to rename your existing `values` directory to `values-en`. This is not a good idea, but pretend for a moment you did just that: Your hypothetical update means your app now has an English `strings.xml` in `values-en` and a Spanish `strings.xml` in `values-es`.

As you might expect, the app would run just fine on devices with the language set to Spanish or English. But what happens if the user's device language is set to Italian? Bad things. Very bad things. If the app is allowed to run, Android will not find string resources that match the current configuration. This will cause your app to crash with a **`Resources.NotFoundException`**.

Android Studio takes steps to save you from this fate. The Android Asset Packaging Tool (AAPT) does many checks while packaging up your resources. If AAPT finds that you are using resources that are not included in the default resource files, it will throw an error at compile time:

```
Android resource linking failed

warn: removing resource
com.bignerdranch.android.criminalintent:string/crime_title_label
without required default value.

AAPT: error: resource string/crime_title_label
(aka com.bignerdranch.android.criminalintent:string/crime_title_label)
not found.

error: failed linking file resources.
```

The moral of the story is this: Provide a *default resource* for each of your resources. Resources in unqualified resource directories are your default resources. Default resources will be used if no match for the current device configuration is found. Your app will misbehave if Android looks for a resource and cannot find either one that matches the device configuration or a default.

## Checking string coverage using the Translations Editor

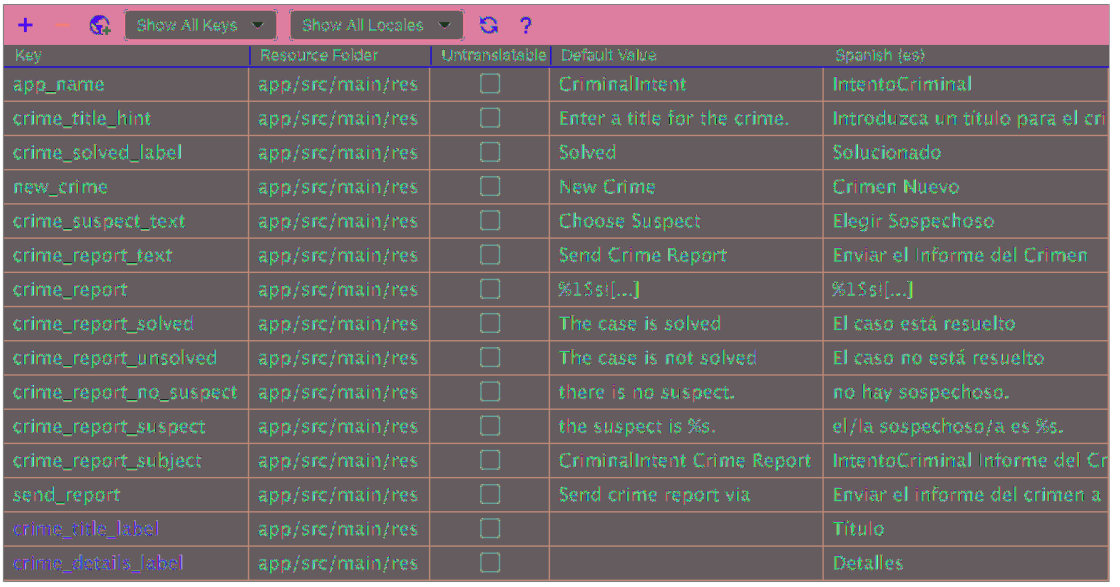
As the number of languages you support grows, making sure you provide a version of each string for each language becomes more difficult. Luckily, Android Studio provides a handy Translations Editor to see all your translations in one place. Before starting, create some "missing" strings by opening your default `strings.xml` and commenting out `crime_title_label` and `crime_details_label` (Listing 18.2).

### Listing 18.2 Commenting out strings (`res/values/strings.xml`)

```
<resources>
    <string name="app_name">CriminalIntent</string>
    <string name="crime_title_hint">Enter a title for the crime.</string>
    <!--<string name="crime_title_label">Title</string>-->
    <!--<string name="crime_details_label">Details</string>-->
    <string name="crime_solved_label">Solved</string>
    ...
</resources>
```

To launch the Translations Editor, right-click one of the `strings.xml` files in the project tool window and select Open Translations Editor. The Translations Editor displays all the app's strings and the translation status for each of the languages for which your app provides any qualified string values. Since `crime_title_label` and `crime_details_label` are commented out, you will see those field names in red (Figure 18.5).

### Figure 18.5 Using the Translations Editor to check your string coverage




This provides an easy way to identify resources that are missing from any locale configuration and add them to the related strings file.

Although you can add strings right in the Translations Editor, in your case you only need to uncomment `crime_title_label` and `crime_details_label`. Do that before moving on.

## Targeting a region

You can qualify a resource directory with a language-plus-region qualifier that targets resources even more specifically. For instance, the qualifier for Spanish spoken in Spain is `-es-rES`, where the `r` denotes a region qualifier and `ES` is the ISO 3166-1-alpha-2 code for Spain. The qualifier for Spanish spoken in Mexico is `-es-rMX`. (Configuration qualifiers are not case sensitive, but it is good to follow Android's convention here: Use a lowercase language code and an uppercase region code prefixed with a lowercase `r`.)

Note that a language-region qualifier, such as `-es-rES`, may look like two distinct configuration qualifiers that have been combined, but it is just one. The region is not a valid qualifier on its own.

A resource qualified with both a locale and region has two opportunities for matching a user's locale. An exact match occurs when both the language and region qualifiers match the user's locale. If no exact match is found, the system will strip off the region qualifier and look for an exact match for the language only.

Which brings us to an important point: Always provide strings in as general a context as possible, using language-only qualified directories as much as possible and region-qualified directories only when necessary. For example, if the differences between European and North or South American Spanish are an issue, it is better to store most of the Spanish strings in a language-only qualified `values-es` directory and provide region-qualified strings only for words and phrases that are different in the different regional dialects.

(For more about supporting languages and regions, check out `developer.android.com/guide/topics/resources/multilingual-support`.)

In fact, this advice goes for all types of alternative resources in the `values` directories: Provide shared resources in more general directories and only include those resources that need to be tailored in more specifically qualified directories.

## Configuration Qualifiers

In the section called For the More Curious: Creating a Landscape Layout in Chapter 3, you saw the configuration qualifier `layout-land`, for landscape screen orientation. The device configurations for which Android provides configuration qualifiers to target resources are:

1. mobile country code (MCC), optionally followed by mobile network code (MNC)
2. language code, optionally followed by region code
3. layout direction
4. smallest width
5. available width
6. available height
7. screen size
8. screen aspect
9. round screen
10. wide color gamut
11. high dynamic range
12. screen orientation
13. UI mode
14. night mode
15. screen density (dpi)
16. touchscreen type
17. keyboard availability
18. primary text input method
19. navigation key availability
20. primary non-touch navigation method
21. API level

You can find descriptions of these characteristics and examples of specific configuration qualifiers at `developer.android.com/guide/topics/resources/providing-resources.html#AlternativeResources`.

Not all qualifiers are supported by earlier versions of Android. Luckily, the system implicitly adds a platform version qualifier to qualifiers that were introduced after Android 1.0. For example, if you use the `highhdr` qualifier, Android will automatically include the `v26` qualifier, because high dynamic range screen qualifiers were added in API level 26. This means you do not have to worry about problems on older devices when you introduce resources qualified for newer devices.

## Prioritizing alternative resources

Given the many types of configuration qualifiers for targeting resources, there may be times when the device configuration will match more than one alternative resource. When this happens, qualifiers are given precedence in the order shown in the list above.

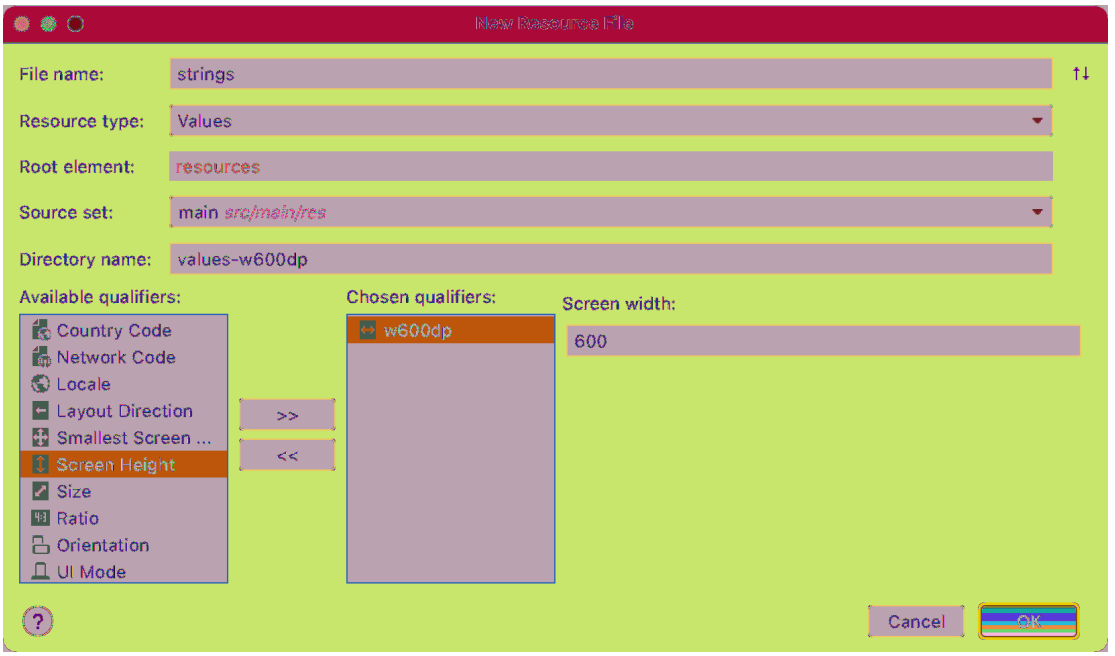
To see this prioritizing in action, add another alternative resource to CriminalIntent – a longer English version of the `crime_title_hint` string resource – to be displayed when the current configuration's width is at least 600dp. The `crime_title_hint` resource is displayed in the crime title text box before the user enters any text. When CriminalIntent is running on a screen that is at least 600dp (such as on a tablet, or perhaps in landscape mode on a smaller device), this change will display a more descriptive, engaging hint for the title field.

Create a new values resource file called `strings`. Follow the steps from the section called Localizing Resources earlier in this chapter to create the resource file, but select Screen Width in the Available qualifiers list and click the >> button to move Screen Width to the Chosen qualifiers section. In the Screen width box that appears, enter 600.

The directory name will automatically be set to values-w600dp; `-w600dp` will match any device with a current screen width of 600dp or more, meaning a device could match when in landscape mode but not in portrait mode. (To learn more about screen size qualifiers, read the section called For the More Curious: More on Determining Device Size near the end of this chapter.)

Your dialog should look like Figure 18.6.

Figure 18.6 Adding strings for a wider screen



Now, add a longer value for `crime_title_hint` to `res/values-w600dp/strings.xml`.

### Listing 18.3 Creating an alternative string resource for a wider screen (`res/values-w600dp/strings.xml`)

```
<resources>
    <string name="crime_title_hint">
        Enter a meaningful, memorable title for the crime.
    </string>
</resources>
```

The only string resource you want to be different on wider screens is `crime_title_hint`, so that is the only string you specify in `values-w600dp`. As we said earlier, you should provide alternatives for only those resources that will be different based on some configuration qualification. You do not need to duplicate strings when they are the same. More than that, you *should* not: Those duplicated strings would only end up being a maintenance hassle down the road.

Now you have three versions of `crime_title_hint`: a default version in `res/values/strings.xml`, a Spanish alternative in `res/values-es/strings.xml`, and a wide-screen alternative in `res/values-w600dp/strings.xml`.

With your device's language set to Spanish, run CriminalIntent, press the + button to open a blank crime detail screen, and rotate to landscape (Figure 18.7). The Spanish language alternative has precedence, so you see the string from `res/values-es/strings.xml` instead of `res/values-w600dp/strings.xml`.

### Figure 18.7 Android prioritizes language over available screen width




Change your settings back to English and check the app again to confirm that the alternative wide-screen string appears as expected.

## Multiple qualifiers

You may have noticed that the New Resource File dialog has many available qualifiers. You can put more than one qualifier on a resource directory. When using multiple qualifiers on directories, you must put them in the order of their precedence. Thus, `values-es-w600dp` is a valid directory name, but `values-w600dp-es` is not. (When you use the New Resource File dialog, it correctly configures the directory name for you.)

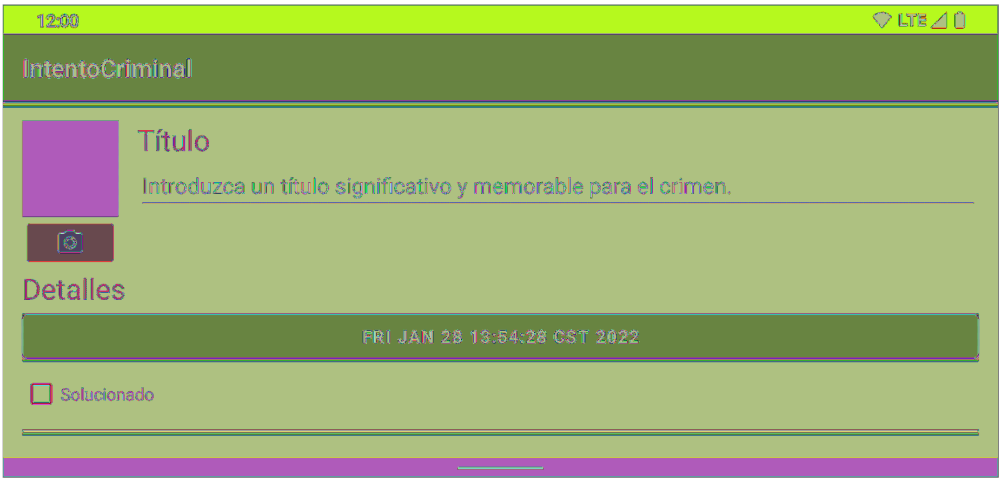
Create a directory for a wide-screen Spanish string by selecting both Locale and Screen Width in the New Resource File dialog. It should be named `values-es-w600dp` and have a file named `strings.xml`. Add a string resource for `crime_title_hint` to `values-es-w600dp/strings.xml`:

### Listing 18.4 Creating a wide-screen Spanish string resource (`res/values-es-w600dp/strings.xml`)

```
<resources>
    <string name="crime_title_hint">
        Introduzca un título significativo y memorable para el crimen.
    </string>
</resources>
```

Now, with your language set to Spanish, run CriminalIntent to confirm that your new alternative resource appears on cue (Figure 18.8).

### Figure 18.8 Spanish wide-screen string resource



## Finding the best-matching resources

Let's walk through how Android determined which version of `crime_title_hint` to display in this run. First, consider the four alternatives for the string resource named `crime_title_hint` and an example landscape device configuration for a Pixel 4 set to Spanish language and with an available screen width greater than 600dp:

**Device configuration**

- Language: `es` (Spanish)
- Available height: `393dp`
- Available width: `830dp`
- (etc.)

**App values for `crime_title_hint`**

- `values`
- `values-es`
- `values-es-w600dp`
- `values-w600dp`

The first step that Android takes to find the best resource is to rule out any resource directory that is incompatible with the current configuration.

None of the four choices is incompatible with the current configuration. (If you rotated the device to portrait, the available width would become 393dp, and the resource directories `values-w600dp/` and `values-es-w600dp/` would be incompatible and thus ruled out.)

After the incompatible resource directories have been ruled out, Android starts working through the precedence list shown in the section called Configuration Qualifiers earlier in this chapter, starting with the highest priority qualifier: MCC. If there is a resource directory with an MCC qualifier, then all resource directories that *do not* have an MCC qualifier are ruled out.

If there is still more than one matching directory, then Android considers the next-highest precedence qualifier and continues until only one directory remains.

In our example, no directories contain an MCC qualifier, so no directories are ruled out, and Android moves down the list to the language qualifier. Two directories (`values-es` and `values-es-w600dp`) contain the matching language qualifier `-es`. The `values` and `values-w600dp` directories do not contain a language qualifier and thus are ruled out.

(However, as you read earlier in this chapter, the unqualified `values` directory serves as the default resource, or fallback. So while it is ruled out for now due to lack of a language qualifier, `values` could still end up being the best match if the other values directories have a mismatch in one or more of the lower-order qualifiers.)

**Device configuration**

- Language: `es` (Spanish)
- Available height: `393dp`
- Available width: `830dp`
- (etc.)

**App values for `crime_title_hint`**

- ~~`values`~~ (not language specific)
- `values-es`
- `values-es-w600dp`
- ~~`values-w600dp`~~ (not language specific)

Because there are multiple values still in the running, Android keeps stepping down the qualifier list. When it reaches available width, it finds one directory with an available width qualifier and one without. It rules out `values-es`, leaving only `values-es-w600dp`:

**Device configuration**

- Language: `es` (Spanish)
- Available height: `393dp`
- Available width: `830dp`
- (etc.)

**App values for `crime_title_hint`**

- ~~`values`~~ (not language or width specific)
- ~~`values-es`~~ (not width specific)
- **`values-es-w600dp`** (best match)
- ~~`values-w600dp`~~ (not language specific)

Thus, Android uses the resource in `values-es-w600dp`.

## Testing Alternative Resources

It is important to test your app on different device configurations to see how your layouts and other resources look on those configurations. You can test on both real and virtual devices. You can also use the layout editor.

The layout editor has many options for previewing how a layout will appear in different configurations. You can preview the layout on different screen sizes, device types, API levels, languages, and more.

To see these options, open `res/layout/fragment_crime_detail.xml` in the layout editor. Then try some of the settings in the toolbar shown in Figure 18.9.

Figure 18.9 Using the layout editor to preview various device configurations




The layout editor lets you try different device orientations and device locales based on the configurations you have provided. To see your default resources in action, set a device or emulator to a language that you have not localized any resources for. Run your app and put it through its paces. Visit all the views and rotate them.

Before continuing to the next chapter, you will probably want to set your device's language back to English.

Congratulations! Now your CriminalIntent app can be enjoyed fully in both Spanish and English. Crimes will be logged. Los casos se resolverán. And all in the comfort of your user's native language (so long as that is either Spanish or English). And adding support for more languages is simply a matter of including additional qualified strings files.

## For the More Curious: More on Determining Device Size

Android provides three qualifiers that allow you to test for the dimensions of the device. Table 18.1 shows these new qualifiers.

Table 18.1 Discrete screen dimension qualifiers

| Qualifier format | Description |
|---|---|
| `wXXXdp` | available width: width greater than or equal to XXX dp |
| `hXXXdp` | available height: height greater than or equal to XXX dp |
| `swXXXdp` | smallest width: width or height (whichever is smaller) greater than or equal to XXX dp |

Let's say that you wanted to specify a layout that would only be used if the display were at least 300dp wide. In that case, you could use an available width qualifier and put your layout file in `res/layout-w300dp` (the "w" is for "width"). You can do the same thing for height by using an "h" (for "height").

However, the height and width may swap depending on the orientation of the device. To detect a particular size of screen, you can use `sw`, which stands for *smallest width*. This specifies the smallest dimension of your screen. Depending on the device's orientation, this can be either width or height. If the screen is 1024x800, then `sw` is 800. If the screen is 800x1024, `sw` is still 800.

## Challenge: Localizing Dates

You may have noticed that, regardless of the device's locale, the dates displayed in CriminalIntent are always formatted in the default US style, with the month before the day. Take your localization a step further by formatting the dates according to the locale configuration. It is easier than you might think.

Check out the developer documentation on the **`DateFormat`** class, which is provided as part of the Android framework. **`DateFormat`** provides a date-time formatter that will take into consideration the current locale. You can control the output further by using configuration constants built into **`DateFormat`**.

# 19
# Accessibility

In this chapter, you will finish your work on CriminalIntent by making it more *accessible*. An accessible app is usable by anyone, regardless of any impairments in vision, mobility, or hearing. These impairments may be permanent, but they could also be temporary or situational: Dilated eyes after an eye exam might make focusing difficult. Greasy hands while cooking may mean you do not want to touch the screen. And if you are at a loud concert, the music drowns out any sounds made by your device. The more accessible an app is, the more pleasant it is to use for everyone.

Making an app fully accessible is a tall order. But that is no excuse not to try. In this chapter you will take some steps toward making CriminalIntent more usable for people with a visual impairment. This is a good place to begin learning about accessibility issues and accessible app design.

For this chapter, we recommend that you work through the exercise with a physical device instead of the emulator. It is possible to work through this chapter with an emulator, but some of the user input required is awkward and difficult to execute on an emulator. It is much easier on a physical device. If you do not have access to a physical device, please read through the section called For the More Curious: Using TalkBack with an Emulator in this chapter before starting the exercise.

The changes you make in this chapter will not alter the appearance of the app. Instead, the changes will make your app easier to explore with *TalkBack*.

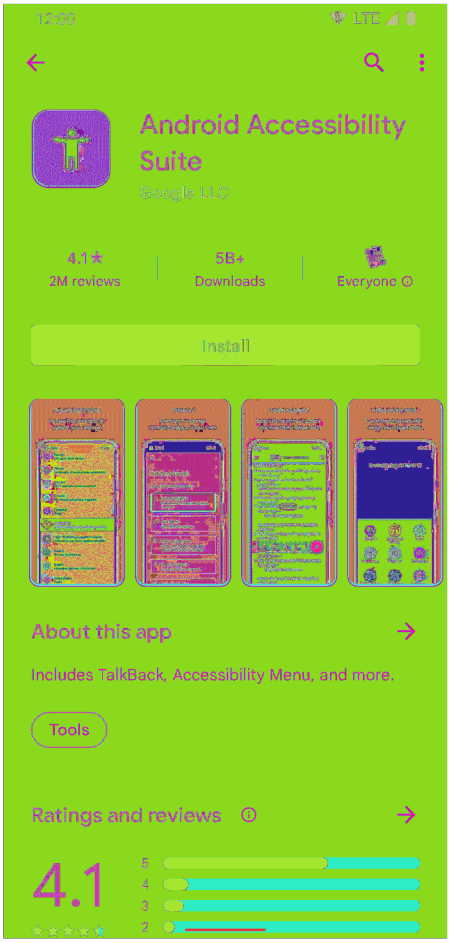
## TalkBack

TalkBack is an Android screen reader made by Google. It speaks out the contents of a screen based on what the user is doing.

TalkBack works because it is an *accessibility service*, which is a special component that can read information from the screen (no matter which app you are using). Anyone can write their own accessibility service, but TalkBack is the most popular.

To use TalkBack, install the Android Accessibility Suite through the Google Play Store on your device (Figure 19.1).

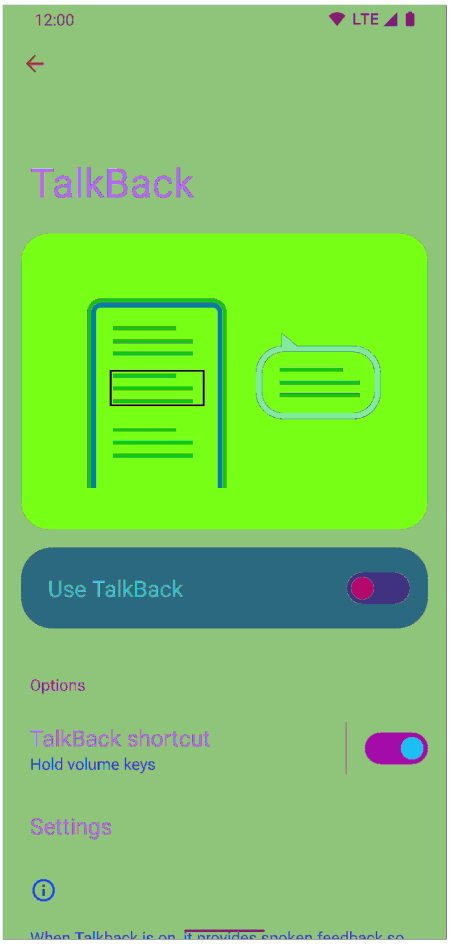
Figure 19.1 Android Accessibility Suite



Next, make sure the device's sound output is not muted – but you may want to grab headphones, because once TalkBack is enabled the device will do a lot of "talking."

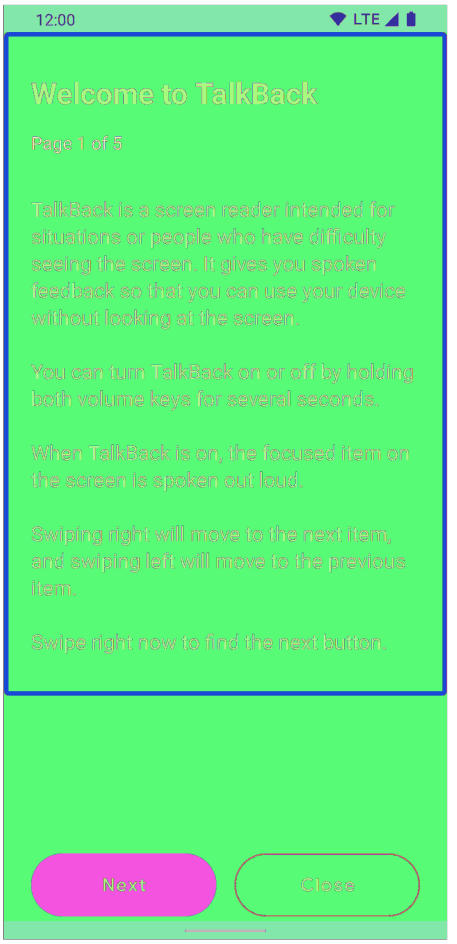
To enable TalkBack, launch Settings and press Accessibility. Press TalkBack under the Screen readers heading. Then press the Use TalkBack (or Use service) switch to turn TalkBack on (Figure 19.2).

Figure 19.2 TalkBack settings screen



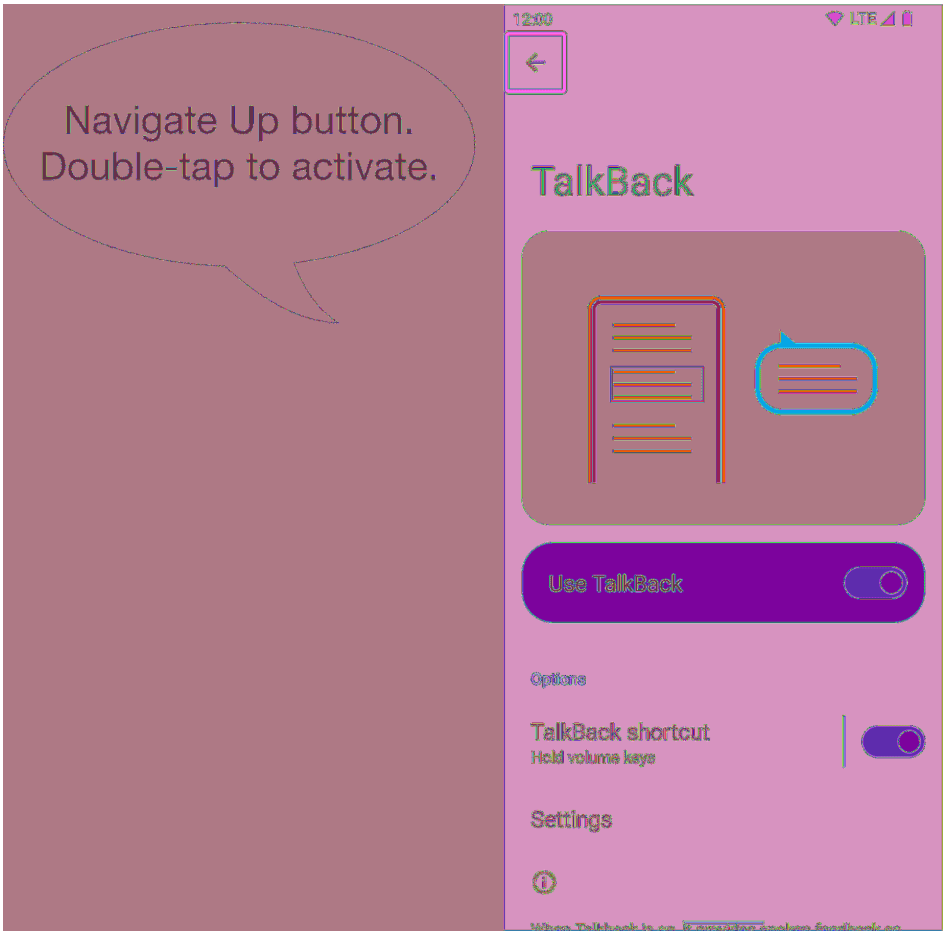
If this is your first time using TalkBack on the device, you will be presented with a tutorial. Go through the tutorial to learn the basic ways to navigate the system. Once the tutorial is done, TalkBack may request additional permissions. Press Allow.

Figure 19.3 Walking through the TalkBack tutorial



You will notice something different right away. A green outline appears around the Up button (Figure 19.4) and the device speaks: “Navigate Up button. Double-tap to activate.”

Figure 19.4 TalkBack enabled




(Although “press” is the usual terminology for Android devices, TalkBack uses “tap.” Also, TalkBack uses double-taps, which are not commonly used in Android.)

The green outline indicates which UI element has *accessibility focus*. Only one UI element can have accessibility focus at a time. When a UI element receives focus, TalkBack will provide information about that element.

When TalkBack is enabled, a single press (or “tap”) gives an element accessibility focus. Double-tapping anywhere on the screen activates the element that has focus. So double-tapping anywhere when the Up button has focus navigates up, double-tapping when a checkbox has focus toggles its check state, and so on. (Also, if your device locks, you can unlock it by pressing the lock icon and then double-tapping anywhere on the screen.)

## Explore by Touch

By turning TalkBack on, you have also enabled TalkBack's Explore by Touch mode. This means the device will speak information about an item immediately after it is pressed. (This assumes that the item pressed specifies information TalkBack can read, which you will learn more about shortly.)

Leave the Up button selected with accessibility focus. Double-tap anywhere on the screen. The device returns you to the Accessibility menu, and TalkBack announces information about what is showing and what has accessibility focus: "Accessibility. Navigate Up button. Double-tap to activate."

Android framework views, such as **`Toolbar`**, **`RecyclerView`**, and **`Button`**, have basic TalkBack support built in. You should use framework views as much as possible so you can leverage the accessibility work that has already been done for them. It is possible to properly respond to accessibility events for custom views, but that is beyond the scope of this book.

(In Chapter 26 through Chapter 29, you will learn about a new way to build layouts on Android called Jetpack Compose. Compose's built-in UI elements also support TalkBack and behave very similarly to what you will see in this chapter.)

## Linear navigation by swiping

Imagine what it would be like to explore an app by touch for the first time. You would not know where things are located. What if the only way to learn what was on the screen was to press all around until you landed on an element that TalkBack could read? You might end up pressing the same thing multiple times – worse, you might miss elements altogether.

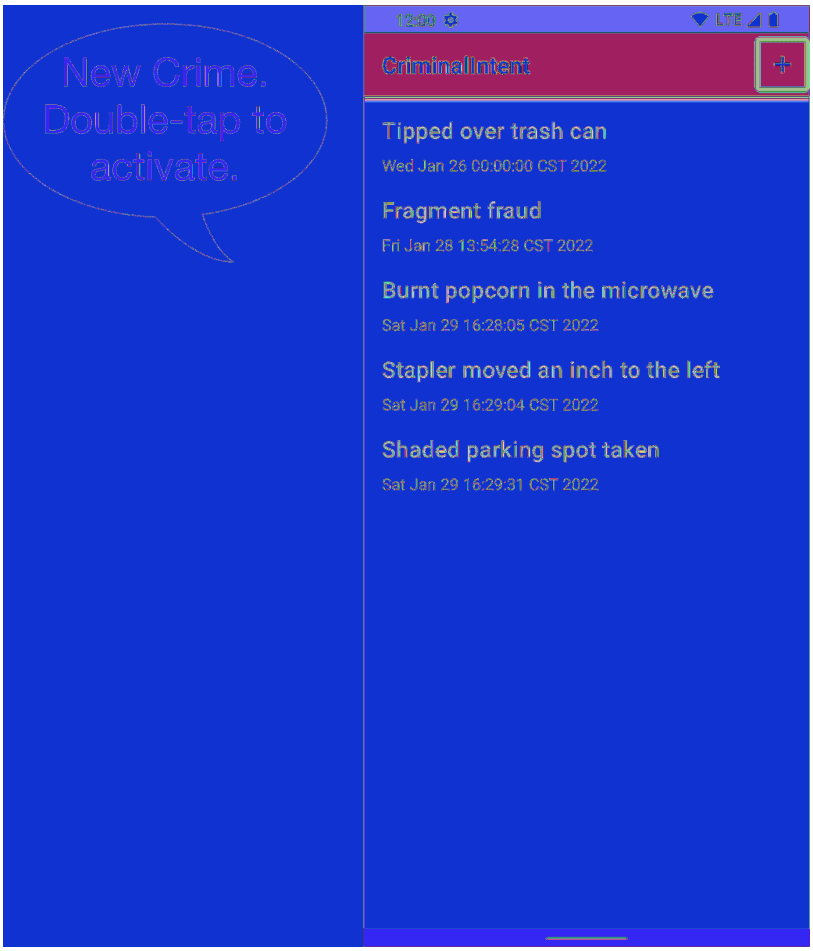
Luckily, there is a way to explore the UI linearly, and in fact this is the more common way to use TalkBack: Swiping right moves accessibility focus to the next item on the screen. Swiping left moves accessibility focus to the previous item on the screen. This allows the user to walk through each item on the screen in a linear fashion, rather than trial-and-error poking around in hopes of landing on something meaningful.

Navigate around the Accessibility settings page using the swiping gestures. (If you are using the emulator, you can mimic this behavior with the keyboard's arrow keys.)

Navigating between apps and screens with the swipe system navigation behaves differently when TalkBack is enabled. When TalkBack is not enabled, you navigate to the Home screen by swiping with one finger from the bottom of the device. With TalkBack enabled, you navigate using the same gesture, but with two fingers. Likewise for Back navigation, you need a two finger swipe from the left or right edge of the device when TalkBack is enabled. Without TalkBack, you use a single finger. You also use two fingers to scroll when TalkBack is enabled.

Now, try out TalkBack in CriminalIntent. Compile and launch the app. When you open it, accessibility focus will be given to the + action item in the app bar by default. (If it is not, press the + to give it accessibility focus.) The device reads out, “CriminalIntent. New Crime. Double-tap to activate” (Figure 19.5).

Figure 19.5 New Crime action item selected




For framework views, such as menu items and buttons, TalkBack will read the visible text content displayed on the view by default. But the New Crime menu item is just an icon and does not have any visible text. In this case, TalkBack looks for other information in the view. You specified a `title` in your menu XML, and that is what TalkBack reads to the user. TalkBack will also provide details about actions the user can take on the view and sometimes information about what kind of view it is.

Now swipe left. Accessibility focus moves to the CriminalIntent title in the app bar. TalkBack announces, “CriminalIntent” (Figure 19.6).

Figure 19.6 App bar title selected




Swipe right, and TalkBack reads information about the + (New Crime) menu button again. Swipe right a second time; accessibility focus moves to the first crime in the list. Swipe left, and focus moves back to the + menu button. Android does its best to move accessibility focus in an order that makes sense.

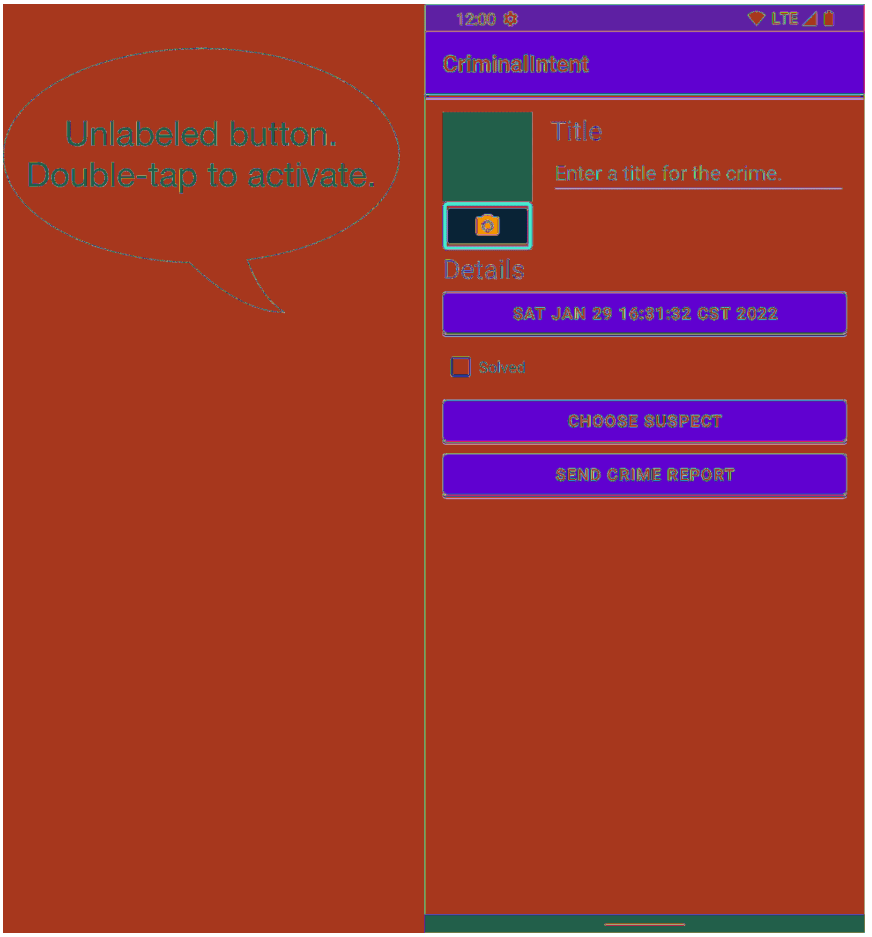
# Making Non-Text Elements Readable by TalkBack

With the New Crime button selected, double-tap anywhere on the screen to launch the crime details screen.

## Adding content descriptions

On the crime details screen, press the image capture button to give it accessibility focus (Figure 19.7). TalkBack announces, “Unlabeled button. Double-tap to activate.” (You may get slightly different results depending on the version of Android you are using.)

Figure 19.7 Image capture button selected




The camera button does not display any text, so TalkBack describes the button as well as it can. While this is TalkBack’s best effort, the information is not very helpful to a user with a vision impairment.

This problem is very easy to fix. You can specify details for TalkBack to read by adding a *content description* to the **`ImageButton`**. A content description is a piece of text that describes the view and is read by TalkBack. (While you are at it, you are going to add a content description for the **`ImageView`** that displays the selected picture, too.)

You can set a view's content description in the XML layout file by setting a value for the attribute `android:contentDescription`. That is what you are going to do next. You can also set it in your UI setup code, using ***`someView`*`.contentDescription = `*`someString`***, which you will do later in this chapter.

The text you set should be meaningful without being overly wordy. Remember, TalkBack users will be listening to the audio, which can be slow. They can speed up the pace of TalkBack's speech output, but even so you want to avoid adding extraneous information and wasting users' time. For example, if you are setting the description for a framework view, avoid including information about what kind of view it is (like "a button"), because TalkBack already knows and includes that information.

First, some housekeeping. Add the content description strings to the unqualified `res/values/strings.xml`.

### Listing 19.1 Adding content description strings (`res/values/strings.xml`)

```
<resources>
    ...
    <string name="crime_details_label">Details</string>
    <string name="crime_solved_label">Solved</string>
    <string name="crime_photo_button_description">Take photo of crime scene</string>
    <string name="crime_photo_no_image_description">
        Crime scene photo (not set)
    </string>
    <string name="crime_photo_image_description">Crime scene photo (set)</string>
    ...
</resources>
```

Android Studio will underline the newly added strings in red, warning you that you have not defined the Spanish version of these new strings. To fix this, add the content description strings to `res/values-es/strings.xml`.

### Listing 19.2 Adding Spanish content description strings (`res/values-es/strings.xml`)

```
<resources>
    ...
    <string name="crime_details_label">Detalles</string>
    <string name="crime_solved_label">Solucionado</string>
    <string name="crime_photo_button_description">
        Tomar foto de la escena del crimen
    </string>
    <string name="crime_photo_no_image_description">
        Foto de la escena del crimen (no establecida)
    </string>
    <string name="crime_photo_image_description">
        Foto de la escena del crimen (establecida)
    </string>
    ...
</resources>
```

Next, open `res/layout/fragment_crime_detail.xml` and set the content description for the **`ImageButton`**.

### Listing 19.3 Setting the content description for **`ImageButton`** (`res/layout/fragment_crime_detail.xml`)

```
...
<ImageButton
    android:id="@+id/crime_camera"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:src="@drawable/ic_camera"
    android:contentDescription="@string/crime_photo_button_description"/>
...
```

Run CriminalIntent again and press the camera button. TalkBack helpfully announces, “Take photo of crime scene button. Double-tap to activate.” This spoken information is much more helpful than “unlabeled button.”

Next, press the crime scene image (which at the moment is just the black placeholder). You might expect the accessibility focus to move to the **`ImageView`**, but the green border does not appear. TalkBack remains silent rather than announcing information about the **`ImageView`**. What gives?

## Making a view focusable

The problem is that the **`ImageView`** is not registered to receive focus. Some views, such as **`Button`**s, are focusable by default. Other views, such **`ImageView`**s, are not. You can make a view focusable by setting its `android:focusable` attribute to `true` or by adding a click listener. You can also make a view focusable by adding an `android:contentDescription`.

Make the crime photo's **`ImageView`** focusable by giving it a content description.

Listing 19.4 Making the photo **`ImageView`** focusable with a content description (`res/layout/fragment_crime_detail.xml`)

```
...
<ImageView
  android:id="@+id/crime_photo"
  ...
  android:background="@color/black"
  android:contentDescription="@string/crime_photo_no_image_description" />
...
```

Run CriminalIntent again and press the crime photo. The **`ImageView`** now accepts focus, and TalkBack announces, "Crime scene photo (not set)" (Figure 19.8).

Figure 19.8 Focusable **`ImageView`**



## Creating a Comparable Experience

You should specify a content description for any UI view that provides information to the user but does not use text to do it (such as an image). If there is a view that does not provide any value other than decoration, you should explicitly tell TalkBack to ignore it by setting the `android:importantForAccessibility` attribute to `no`.

You might think, “If a user cannot see, why do they need to know whether there is an image?” But you should not make assumptions about your users. More importantly, you should make sure a user with a visual impairment gets the same amount of information and functionality as a user without one. The overall experience and flow may be different, but all users should be able to get the same functionality from the app.

Good accessibility design is not about reading out every single thing on the screen. Instead, it focuses on comparable experiences. Which pieces of information and context are important?

Right now, the user experience related to the crime photo is limited. TalkBack will always announce that the image is not set, even if an image is indeed set. To see this for yourself, press the camera button and then double-tap anywhere on the screen to activate it. The camera app launches, and TalkBack announces, “Camera.” Capture a photo by pressing the shutter button and then double-tapping anywhere on the screen.

Accept the photo. (The steps will be different depending on which camera app you are using, but remember that you will need to press to select a button and then double-tap anywhere to activate it.) The crime details screen will appear with the updated photo. Press the photo to give it accessibility focus. TalkBack announces, “Crime scene photo (not set).”

To provide more relevant information to TalkBack users, dynamically set the content description of the **`ImageView`** in **`updatePhoto()`**.

### Listing 19.5 Dynamically setting the content description (`CrimeDetailFragment.kt`)

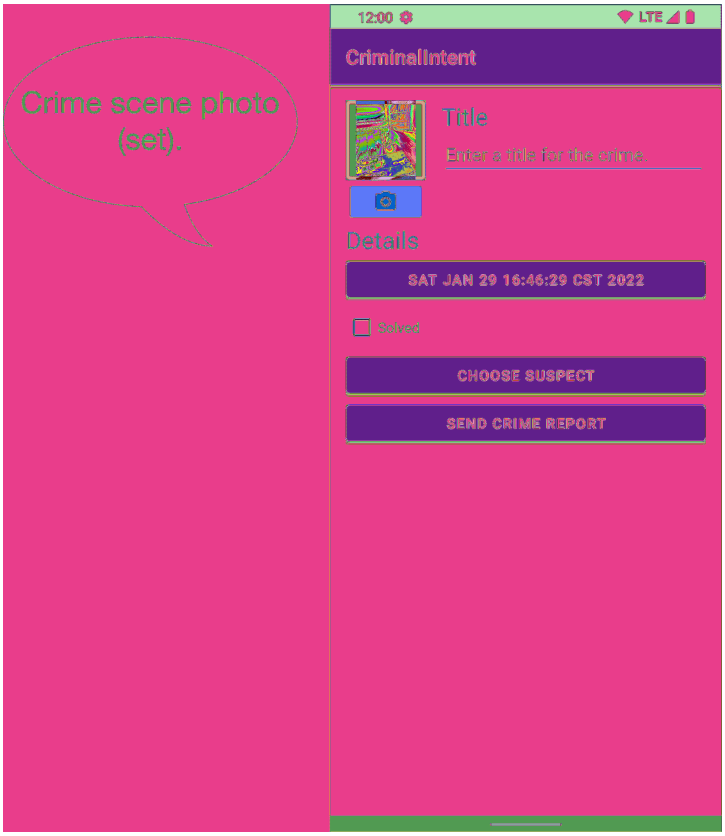
```
class CrimeDetailFragment : Fragment() {
    ...
    private fun updatePhoto(photoFileName: String?) {
      if (binding.crimePhoto.tag != photoFileName) {
          val photoFile = photoFileName?.let {
              File(requireContext().applicationContext.filesDir, it)
          }

          if (photoFile?.exists() == true) {
              binding.crimePhoto.doOnLayout { measuredView ->
                  val scaledBitmap = getScaledBitmap(
                      photoFile.path,
                      measuredView.width,
                      measuredView.height
                  )
                  binding.crimePhoto.setImageBitmap(scaledBitmap)
                  binding.crimePhoto.tag = photoFileName
                  binding.crimePhoto.contentDescription =
                      getString(R.string.crime_photo_image_description)
              }
          } else {
              binding.crimePhoto.setImageBitmap(null)
              binding.crimePhoto.tag = null
              binding.crimePhoto.contentDescription =
                  getString(R.string.crime_photo_no_image_description)
          }
      }
  }
}
```

Now, whenever the photo view is updated, **`updatePhoto()`** will update the content description. If the photo does not exist, it will set the content description to indicate that there is no photo. Otherwise, it will set the content description to indicate that a photo is present.

Run CriminalIntent. View the crime detail screen for the crime you just added a photo to. Press the photo of the crime scene (Figure 19.9). TalkBack proudly announces, “Crime scene photo (set).”

Figure 19.9 Focusable **`ImageView`** with a dynamic description




Congratulations on making your app more accessible. One of the most common reasons developers cite for not making their apps more accessible is lack of awareness about the topic. You are now aware and can see how easy it is to make your apps more usable to TalkBack users. And, as a bonus, improving your app’s TalkBack support means it will also be more likely to support other accessibility services, such as BrailleBack.

Designing and implementing an accessible app may seem overwhelming. People make entire careers out of being accessibility engineers. But rather than forgoing accessibility altogether because you fear you will not do it right, start with the basics: Make sure every meaningful piece of content is reachable and readable by TalkBack. Make sure TalkBack users get enough context to understand what is going on in your app – without having to listen to extraneous information that wastes their time. And, most importantly, listen to your users and learn from them.

With that, you have reached the end of your time with CriminalIntent. In 11 chapters, you have created a complex application that uses fragments, talks to other apps, takes pictures, stores data, and even speaks Spanish. Why not celebrate with a piece of cake?

Just be sure to clean up after yourself. You never know who might be watching.

## For the More Curious: Using TalkBack with an Emulator

If you do not have access to a physical Android device, it is possible to work through this chapter on an emulator. TalkBack is not intended to work in conjunction with keyboard and mouse input from your computer, but the emulator does provide some workarounds to help you get by.

First, you will need an emulator image that has the Play Store installed, so that you can download the Android Accessibility Suite. Not all emulators include the Play Store in the device image, so look for the Play Store icon when creating your emulator. (Back in Chapter 1, you created a Pixel 4 emulator, which does have the Play Store.) You will need to log in to a Google account to use the Play Store.

To scroll on an emulator, hold down the Control button on the keyboard, click one of the two semitransparent circles that appear, and drag up or down with your mouse or trackpad (Figure 19.10). This gesture can also be used on the emulator to perform a “pinch” gesture.

Figure 19.10 Scrolling on the emulator



Navigating between apps and screens with the swipe system navigation is different when TalkBack is enabled. Many actions, such as navigating to the Home screen or switching between apps, are accomplished with a single finger swipe when TalkBack is not enabled. With TalkBack, you perform the same gesture, but with two fingers.

However, performing these two-finger gestures is easy on a real device, but not on an emulator. If you are using the emulator, use the three system buttons in the emulator's control toolbar (Figure 19.11) or on the emulated device to navigate between apps and screens.

Figure 19.11 Using three button navigation on the emulator



## For the More Curious: Using Accessibility Scanner

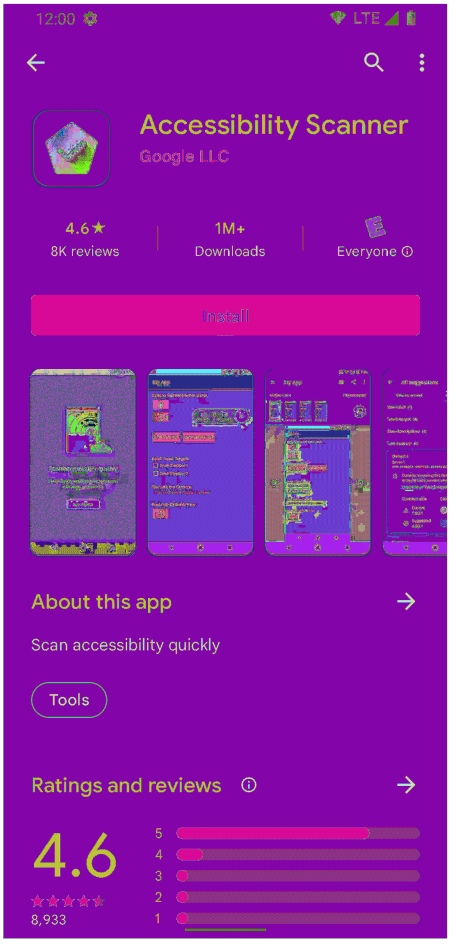
In this chapter you focused on making your app more accessible using TalkBack. But this is not the whole story. Accommodating people with visual impairments is just one subset of accessibility.

Testing your application for accessibility ideally involves user tests by people who actually use accessibility services on a regular basis. If this is not possible, you should still do your best to make your app accessible.

Google's Accessibility Scanner analyzes apps and evaluates how accessible they are. It provides suggestions based on its findings. Try it out on CriminalIntent.

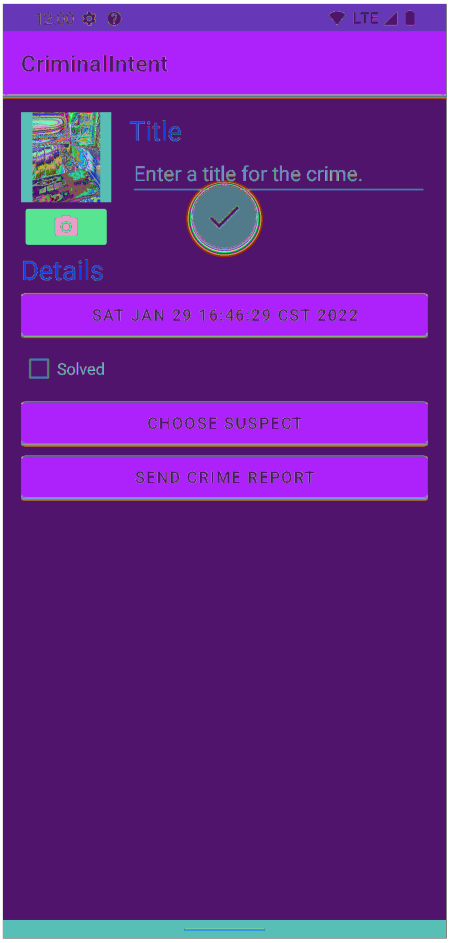
Begin by installing the Accessibility Scanner app on your device (Figure 19.12).

Figure 19.12 Installing Accessibility Scanner



Run Accessibility Scanner, and it will walk you through several setup steps. When you have it running and you see a large blue check mark icon hovering over your screen, the real fun can begin. Launch CriminalIntent from the app launcher or overview screen, leaving the check mark alone. Once CriminalIntent appears, make sure it is displaying a crime details screen (Figure 19.13).

Figure 19.13 Launching CriminalIntent for analysis



Press the check mark. If Accessibility Scanner asks for more permissions, grant them. With the check mark button expanded, you will see two options: Record and Snapshot. Select Snapshot and Accessibility Scanner will go to work.

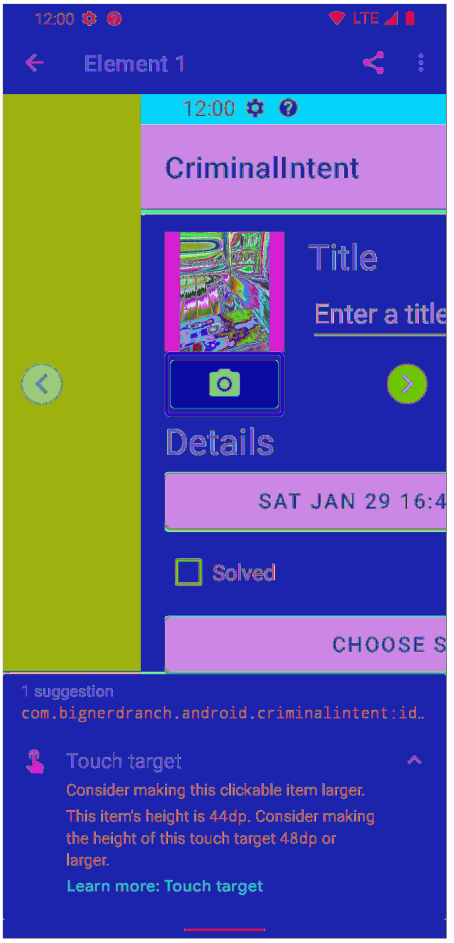
You will see a progress spinner while the analysis happens. Once the analysis is complete, the app bar at the top of the screen will indicate how many suggestions Accessibility Scanner has for you, and some UI elements will have orange outlines around them (Figure 19.14).

Figure 19.14 Accessibility Scanner results summary



The **`ImageButton`** and **`EditText`** have outlines around them. This indicates that the scanner found potential accessibility problems with those views. Press the **`ImageButton`** to view accessibility suggestions for that view. Press the down arrow in the bottom sheet to drill into the details (Figure 19.15).

Figure 19.15 Accessibility Scanner **`ImageButton`** recommendations

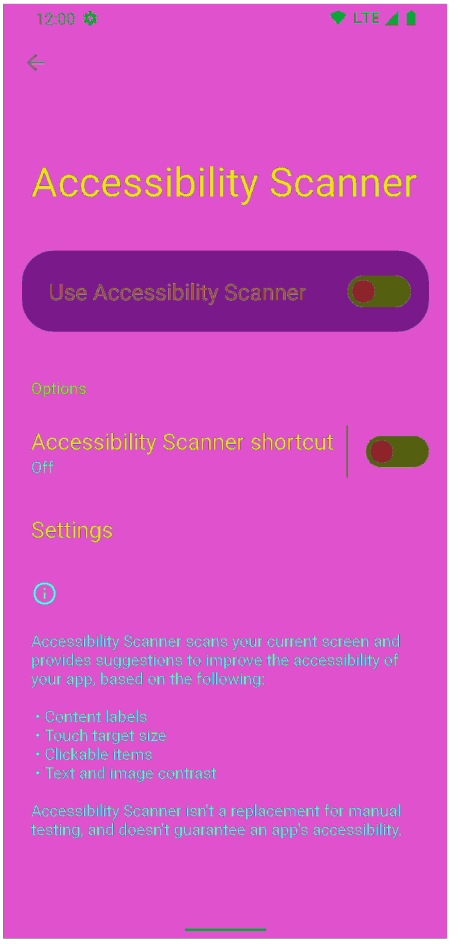



Accessibility Scanner suggests you increase the size of the **`ImageButton`**. The recommended minimum size for all touch targets is 48dp. The **`ImageButton`**'s height is smaller, which you can easily fix by specifying an `android:minHeight` attribute for the view.

You can learn more about Accessibility Scanner's recommendation by pressing Learn more: Touch target.

To turn Accessibility Scanner off, go back to Settings. Press Accessibility, then Accessibility Scanner. Press the toggle to turn the scanner off (Figure 19.16).

Figure 19.16 Turning Accessibility Scanner off



## Challenge: Improving the List

On the crime list screen, TalkBack reads the title and date of each item. However, it does not indicate whether the crime is solved. Fix this problem by giving the handcuff icon a content description.

Note that the readout is a bit lengthy, given the date format, and that the solved status is read at the very end – or not at all, if the crime is not solved. To take this challenge one step further, instead of having TalkBack read off the contents of the two **`TextView`**s and the content description of the icon (if the icon is present), add a dynamic content description to each row in the recycler view. In the description, summarize the data the user sees in the row.

## Challenge: Providing Enough Context for Data Entry

The date button and CHOOSE SUSPECT button both suffer from a similar problem. Users, whether using TalkBack or not, are not explicitly told what the button with the date on it is for. Similarly, once users select a contact as the suspect, they are no longer told or shown what the button represents. Users can probably infer the meaning of the buttons and the text on those buttons, but should they have to?

This is one of the nuances of UI design. It is up to you (or your design team) to figure out what makes the most sense for your application – to balance simplicity of the UI with ease of use.

For this challenge, update the implementation of the details screen so that users do not lose context about what the data they have chosen means. This could be as simple as adding labels for each field. To do this, you could add a **`TextView`** label for each button. Then you would tell TalkBack that the **`TextView`** is a label for the **`Button`** using the `android:labelFor` attribute.

```
<TextView
    android:id="@+id/crime_date_label"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="Date"
    <shd>android:labelFor="@+id/crime_date"/></shd>
<Button
    android:id="@+id/crime_date"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    tools:text="Wed May 11 11:56 EST 2022"/>
```

The `android:labelFor` attribute tells TalkBack that the **`TextView`** serves as a label to the view specified by the ID value. `labelFor` is defined on the **`View`** class, so you can associate any view as the label for any other view. Note that you must use the `@+id` syntax here because you are referring to an ID that has not been defined at that point in the file. You could now remove the + from the `android:id="@+id/crime_date"` line in the **`TextView`**'s definition, but it is not necessary to do so.

## Challenge: Announcing Events

By adding dynamic content descriptions to the crime scene photo **`ImageView`**, you improved the crime scene photo experience. But the onus is on the TalkBack user to press the **`ImageView`** to check its status. A sighted user has the benefit of seeing the image change (or not) when returning from the camera app.

You can provide a similar experience via TalkBack by announcing what happened as a result of the camera app closing. Read up on the **`View.announceForAccessibility(…)`** function in the documentation and use it in CriminalIntent at the appropriate time.

You might consider making the announcement when you get a result back from taking a photo. If you do, there will be some timing issues related to the activity lifecycle. You can get around these by delaying the announcement. Posting a **`Runnable`** allows you to execute some code after a small amount of time. It might look something like this:

```
someView.postDelayed(Runnable {
    // code for making announcements here
}, SOME_DURATION_IN_MILLIS)
```

Or you could avoid using a **`Runnable`** by instead using some other mechanism for knowing when to announce the change. For example, you might consider making the announcement in **`onResume()`** instead – though you would then need to keep track of whether the user has just returned from the camera app.

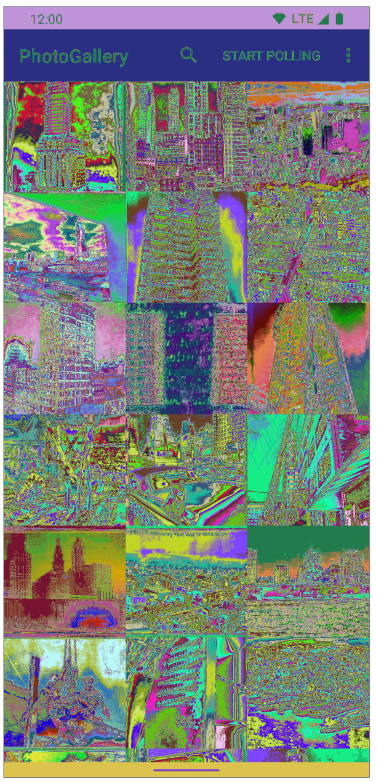
# 20

# Making Network Requests and Displaying Images

The apps that dominate the brains of users are networked apps. Those people fiddling with their phones instead of talking to each other at dinner? They are maniacally checking their newsfeeds, responding to text messages, or playing networked games.

To get started with networking in Android, you are going to create a new app called PhotoGallery. PhotoGallery is a client for the photo-sharing site Flickr. It will fetch and display the most interesting public photos of the day according to Flickr. Figure 20.1 gives you an idea of what the app will look like.

Figure 20.1 Complete PhotoGallery

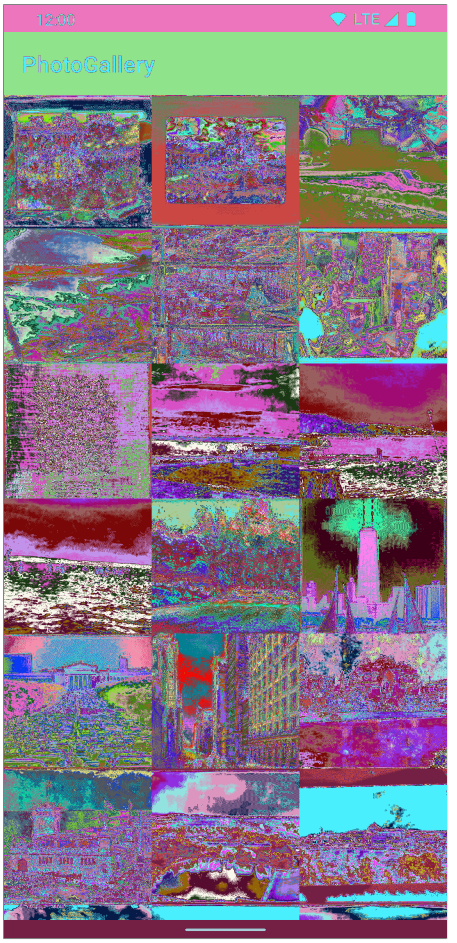


(We have added a filter to our PhotoGallery implementation to show only photos listed on Flickr as having no known copyright restrictions. Visit `flickr.com/commons/usage/` to learn more about unrestricted images. All other photos on Flickr are the property of the person who posted them and are subject to usage restrictions depending on the license specified by the owner. To read more about permissions for using images that you retrieve from Flickr, visit `flickr.com/creativecommons/`.)

You will spend several chapters with PhotoGallery. In this chapter, you will learn how to use the Retrofit library to make web requests to REST APIs and the Moshi library to deserialize the response to these requests from JSON into Kotlin objects. Almost all day-to-day programming of web services these days is based on the HTTP networking protocol. Retrofit provides a type-safe way to access HTTP and HTTP/2 web services easily from Android apps.

In addition, you will be downloading and displaying photos using URLs generated from the data in the network response. Instead of doing this part by hand, you will rely on the Coil library, developed by Instacart to perform all the tricky work of efficiently displaying those images from within a **`RecyclerView`**. By the end of the chapter, you will be fetching, parsing, and displaying photos from Flickr (Figure 20.2).

Figure 20.2 PhotoGallery at the end of the chapter



## Creating PhotoGallery

Create a new Android application (File → New → New Project...). Select the Empty Activity template and click Next.

Configure your project as shown in Figure 20.3: Name the application PhotoGallery. Make sure the Package name is com.bignerdranch.android.photogallery and the Language is Kotlin. Select API 24: Android 7.0 (Nougat) from the Minimum SDK dropdown.

Figure 20.3 Configuring the PhotoGallery project




Click Finish to generate the project.

Many of the initial steps to set up this project will look familiar from your earlier projects. Once again, you will use libraries and tools like the **Fragment** class, the **ViewModel** class, the **RecyclerView** component, and View Binding. Start by opening the app/build.gradle file (the one labeled (Module: PhotoGallery.app)) to add the dependencies you will need and to enable View Binding.

### Listing 20.1 Setting up your project's build (app/build.gradle)

```
...
android {
    ...
    kotlinOptions {
        jvmTarget = '1.8'
    }
    buildFeatures {
        viewBinding true
    }
}

dependencies {
    ...
    implementation 'androidx.constraintlayout:constraintlayout:2.1.3'
    implementation 'androidx.fragment:fragment-ktx:1.4.1'
    implementation 'androidx.recyclerview:recyclerview:1.2.1'
    implementation 'androidx.lifecycle:lifecycle-viewmodel-ktx:2.4.1'
    implementation 'androidx.lifecycle:lifecycle-runtime-ktx:2.4.1'
    testImplementation 'junit:junit:4.13.2'
    ...
}
```

Do not forget to sync your files after you have made these changes. Now, on to the code.

Create the fragment that will display when the app launches. PhotoGallery will display its results in a **RecyclerView**, using the built-in **GridLayoutManager** to arrange the items in a grid. The Kotlin class will be named **PhotoGalleryFragment**, so first right-click the res/layout folder in the project tool window and select New → Layout resource file. Name this file fragment_photo_gallery.xml and enter androidx.recyclerview.widget.RecyclerView as the root element.

Click OK. The generated file is mostly correct. Just add an ID so that you can reference the **RecyclerView** within your Kotlin code.

### Listing 20.2 Adding an ID (res/layout/fragment_photo_gallery.xml)

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.recyclerview.widget.RecyclerView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/photo_grid"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

</androidx.recyclerview.widget.RecyclerView>
```

Next, create a Kotlin file for the **PhotoGalleryFragment** class. In the project tool window, right-click the com.bignerdranch.android.photogallery package and select New → Kotlin Class/File.

Subclass the **Fragment** class and inflate and bind your layout using View Binding. While you are at it, set the recycler view's layoutManager to a new instance of **GridLayoutManager**. Hardcode the number of columns to 3.

Listing 20.3 Setting up the fragment (`PhotoGalleryActivity.kt`)

```
class PhotoGalleryFragment : Fragment() {
    private var _binding: FragmentPhotoGalleryBinding? = null
    private val binding
        get() = checkNotNull(_binding) {
            "Cannot access binding because it is null. Is the view visible?"
        }

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View {
        _binding =
            FragmentPhotoGalleryBinding.inflate(inflater, container, false)
        binding.photoGrid.layoutManager = GridLayoutManager(context, 3)
        return binding.root
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }
}
```

With the skeleton of your **`PhotoGalleryFragment`** class complete, include it in the **`MainActivity`** using a **`FragmentContainerView`**.

Listing 20.4 Adding a fragment container (`res/layout/activity_main.xml`)

```
<?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">

    <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>
<androidx.fragment.app.FragmentContainerView
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/fragment_container"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:name="com.bignerdranch.android.photogallery.PhotoGalleryFragment"
    tools:context=".MainActivity" />
```

Run PhotoGallery to make sure everything is wired up correctly before moving on. If all is well, you will have a very nice blank screen.

## Networking Basics with Retrofit

Although it is not developed by Google, Retrofit is the de facto official way to communicate with an HTTP API on Android. Retrofit is an open-source library created and maintained by Square (`square.github.io/retrofit`). It is highly configurable and extendable, allowing you to easily and safely communicate with a remote web server. It is organized into components that serve a specific purpose, and you can swap out individual components as you need.

Retrofit is meant to define the contracts for many different types of network requests. Similar to using the Room database library, you write an interface with annotated instance methods, and Retrofit creates the implementation. Under the hood, Retrofit's implementation uses OkHttp, another library by Square, to handle making an HTTP request and parsing the HTTP response.

Head back to your `app/build.gradle` file and add the Retrofit and OkHttp dependencies. Retrofit integrates seamlessly with Kotlin coroutines, so add those dependencies as well. Sync your Gradle file after you make these changes.

### Listing 20.5 Adding the Retrofit dependency (`app/build.gradle`)

```
dependencies {
    ...
    implementation 'androidx.lifecycle:lifecycle-runtime-ktx:2.4.1'
    implementation 'com.squareup.retrofit2:retrofit:2.9.0'
    implementation 'com.squareup.okhttp3:okhttp:4.9.3'
    implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-core:1.6.0'
    implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-android:1.6.0'
    ...
}
```

Before implementing the Flickr REST API, you need to configure Retrofit to fetch and log the contents of a web page URL – specifically, Flickr's home page.

Using Retrofit involves a bunch of moving parts. Starting simple will allow you to see the foundations. Later, you will build on this basic implementation to create your Flickr requests and *deserialize* the responses – meaning convert the linear, serialized data into non-serial pieces of data. That non-serial data will be your model objects.

## Defining an API interface

It is time to define the API calls you want your app to make. First, create a new package for your API-specific code. In the project tool window, right-click the com.bignerdranch.android.photogallery package and choose New → Package. Name your new package api.

Next, add a Retrofit API interface to your new package. A Retrofit API interface is a standard Kotlin interface that uses Retrofit annotations to define API calls. Right-click the api package in the project tool window. Choose New → Kotlin Class/File and name the file FlickrApi. In the new file, define an interface named **`FlickrApi`** and add a single function representing a GET request.

### Listing 20.6 Adding a Retrofit API interface (`api/FlickrApi.kt`)

```
interface FlickrApi {
    @GET("/")
    suspend fun fetchContents(): String
}
```

Since network requests are inherently asynchronous operations, Retrofit naturally supports Kotlin coroutines. If you mark a function with the `suspend` modifier, Retrofit will be able to perform networking requests within a coroutine scope and suspend while waiting for a server response. It has support for many other asynchronous libraries, but in this book we focus on using coroutines for network requests.

Each function in the interface maps to a specific HTTP request and must be annotated with an *HTTP request method annotation*. This annotation tells Retrofit the HTTP request type (also known as an "HTTP verb") that the function in your API interface maps to. The most common request types are `@GET`, `@POST`, `@PUT`, `@DELETE`, and `@HEAD`. (For an exhaustive list of available types, see the API docs at `square.github.io/retrofit/2.x/retrofit`.)

The `@GET("/")` annotation in the code above configures the HTTP request used by **`fetchContents()`** to perform a GET request. The `"/"` is the *relative path* – a path string representing the relative URL from the base URL of your API endpoint. Most HTTP request method annotations include a relative path. In this case, the relative path of `"/"` means the request will be sent to the base URL, which you will provide shortly.

The type you use as the return type specifies the data type you would like Retrofit to deserialize the HTTP response into. Every API request you define in your Retrofit API should include a return type. A general response type called **`OkHttp.ResponseBody`** is provided with Retrofit, which you can use to get the raw response from the server. Specifying **`String`** tells Retrofit that you want the response parsed into a **`String`** object instead.

## Building the Retrofit object and creating an API instance

The Retrofit instance is responsible for implementing and creating instances of your API interface. To make web requests based on the API interface you defined, you need Retrofit to implement and instantiate your **`FlickrApi`** interface.

First, build and configure a Retrofit instance. Open `PhotoGalleryFragment.kt`. In **`onViewCreated(…)`**, build a Retrofit object and use it to create a concrete implementation of your **`FlickrApi`** interface.

### Listing 20.7 Using the Retrofit object to create an instance of the API (`PhotoGalleryFragment.kt`)

```
class PhotoGalleryFragment : Fragment() {
    ...
    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View {
        ...
    }

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        val retrofit: Retrofit = Retrofit.Builder()
            .baseUrl("https://www.flickr.com/")
            .build()

        val flickrApi: FlickrApi = retrofit.create<FlickrApi>()
    }
    ...
}
```

**`Retrofit.Builder()`** is a fluent interface that makes it easy to configure and build your Retrofit instance. You provide a base URL for your endpoint using the **`baseUrl(…)`** function. Here, you provide the Flickr home page: `"https://www.flickr.com/"`. Make sure to include the appropriate protocol with the URL (here, `https://`). Also, always include a trailing / to ensure Retrofit correctly appends the relative paths you provide in your API interface onto the base URL.

Calling **`build()`** returns a Retrofit instance, configured based on the settings you specified using the builder object. Once you have a Retrofit object, you use it to create an instance of your API interface.

Unlike the Room library, Retrofit does not generate any code at compile time – instead, it does all the work at runtime. When you call **`retrofit.create()`**, Retrofit uses the information in the API interface you specify, along with the information you specified when building the Retrofit instance, to create and instantiate an anonymous class that implements the interface on the fly.

### Adding a String converter

Retrofit is not actually handling the nitty-gritty aspects of performing network requests for you. Under the hood, it uses the OkHttp library as its HTTP client (`square.github.io/okhttp`). When getting a response back from the server, by default, Retrofit deserializes web responses into **`okhttp3.ResponseBody`** objects. But for logging the contents of a web page, it is much easier to work with a plain ol' **`String`**.

To get Retrofit to deserialize the response into strings instead, you will specify a *converter* when building your Retrofit object.

A converter knows how to decode a **`ResponseBody`** object into some other object type. You could create a custom converter, but you do not have to. Lucky for you, Square created an open-source converter, called the *scalars converter*, that can convert the response into a string. You will use it to deserialize Flickr responses into string objects.

To use the scalars converter, first add the dependency to your `app/build.gradle` file.

### Listing 20.8 Adding the scalars converter dependency (`app/build.gradle`)

```
dependencies {
    ...
    implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-android:1.6.0'
    implementation 'com.squareup.retrofit2:converter-scalars:2.9.0'
    ...
}
```

Once your Gradle files sync, create an instance of the scalars converter factory and add it to your Retrofit object.

### Listing 20.9 Adding the converter to the Retrofit object (`PhotoGalleryFragment.kt`)

```
class PhotoGalleryFragment : Fragment() {
    ...
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        val retrofit: Retrofit = Retrofit.Builder()
            .baseUrl("https://www.flickr.com/")
            .addConverterFactory(ScalarsConverterFactory.create())
            .build()

        val flickrApi: FlickrApi = retrofit.create<FlickrApi>()
    }
    ...
}
```

**`Retrofit.Builder`**'s **`addConverterFactory(…)`** function expects an instance of **`Converter.Factory`**. A converter factory knows how to create and return instances of a particular converter. **`ScalarsConverterFactory.create()`** returns an instance of the scalars converter factory (**`retrofit2.converter.scalars.ScalarsConverterFactory`**), which in turn will provide instances of a scalars converter when Retrofit needs it.

More specifically, since you specified **`String`** as the return type for **`FlickrApi.fetchContents()`**, the scalars converter factory will provide an instance of the string converter (**`retrofit2.converter.scalars.StringResponseBodyConverter`**). Your Retrofit object will use the string converter to convert the **`ResponseBody`** object into a **`String`** before returning the result.

Square provides other handy open-source converters for Retrofit. Later in this chapter, you will use the Moshi converter. You can see the other available converters, and information about creating your own custom converter, at `square.github.io/retrofit`.

## Executing a web request

Up to this point, you have been writing code to configure your network request. You are very close to the moment you have been waiting for: executing a web request and logging the result. Your next step is to launch a coroutine using the `viewLifecycleOwner.lifecycleScope` property and then call your **`fetchContents()`** function. Do that and log out the result.

Listing 20.10 Making a network request (`PhotoGalleryFragment.kt`)

```
private const val TAG = "PhotoGalleryFragment"

class PhotoGalleryFragment : Fragment() {
    ...
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        val retrofit: Retrofit = Retrofit.Builder()
            .baseUrl("https://www.flickr.com/")
            .addConverterFactory(ScalarsConverterFactory.create())
            .build()

        val flickrApi: FlickrApi = retrofit.create<FlickrApi>()

        viewLifecycleOwner.lifecycleScope.launch {
            val response = flickrApi.fetchContents()
            Log.d(TAG, "Response received: $response")
        }
    }
    ...
}
```

Retrofit makes it easy to respect the two most important Android threading rules:

1. Execute long-running operations only on a background thread, never on the main thread.
2. Update the UI only from the main thread, never from a background thread.

When you call **`fetchContents()`**, Retrofit automatically executes the request *on a background thread*. Retrofit manages the background thread for you, so you do not have to worry about it. When it receives a response, thanks to coroutines, it will pass the result back on the thread where it was first invoked, which in this case is the UI thread.

## Asking permission to network

The last thing that is required to get networking up and running is to ask permission. Just as users would not want you secretly taking their pictures, they also would not want you secretly downloading ASCII pictures of cats.

To ask permission to network, add the following permission to your `manifests/AndroidManifest.xml` file.

### Listing 20.11 Adding networking permission to the manifest (`manifests/AndroidManifest.xml`)

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
          package="com.bignerdranch.android.photogallery" >

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

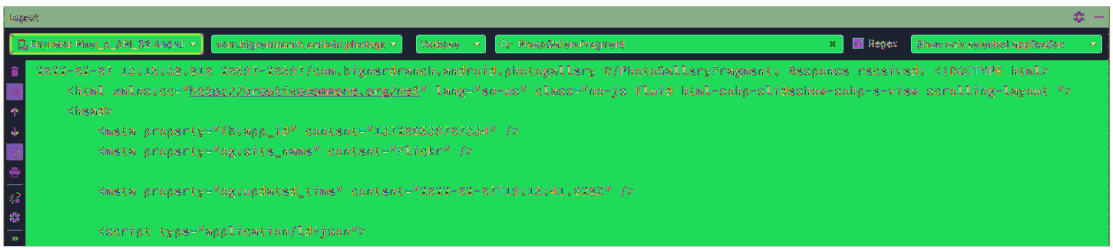
    <application>
        ...
    </application>

</manifest>
```

Android treats the `INTERNET` permission as "normal," since so many apps require it. As a result, all you need to do to use this permission is declare it in your manifest. More dangerous permissions (like the one allowing you to know the device's location) also require a runtime request.

Run your code, and you should see the amazing Flickr home page HTML in Logcat, as shown in Figure 20.4. (Use your `TAG` constant to search or filter Logcat for PhotoGalleryFragment, as shown.)

### Figure 20.4 `Flickr.com` HTML in Logcat



## Moving toward the repository pattern

Right now, your networking code is embedded in your fragment. It would be better if the Retrofit configuration code and API direct access were in a separate class.

Create a new Kotlin file named `PhotoRepository.kt`. Add a property to stash a **`FlickrApi`** instance. Cut the Retrofit configuration code and API interface instantiation code from **`PhotoGalleryFragment`** and paste it into an `init` block in the new class. (These are the two lines that start with `val retrofit: Retrofit = ...` and `val flickrApi = ...` in Listing 20.12.)

Split the `flickrApi` declaration and assignment onto two lines to declare `flickrApi` as a private property on **`PhotoRepository`**. This will allow you to access it elsewhere in the class (outside the `init` block) – but not outside the class.

When you are done, **`PhotoRepository`** should match Listing 20.12.

### Listing 20.12 Creating **`PhotoRepository`** (`PhotoRepository.kt`)

```
class PhotoRepository {
  private val flickrApi: FlickrApi

  init {
      val retrofit: Retrofit = Retrofit.Builder()
          .baseUrl("https://www.flickr.com/")
          .addConverterFactory(ScalarsConverterFactory.create())
          .build()
      flickrApi = retrofit.create()
  }
}
```

If you have not already, cut the redundant Retrofit configuration code from **`PhotoGalleryFragment`** (Listing 20.13). This will cause an error, which you will fix once you finish fleshing out **`PhotoRepository`**.

### Listing 20.13 Cutting Retrofit setup from the fragment (`PhotoGalleryFragment.kt`)

```
class PhotoGalleryFragment : Fragment() {
    ...
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        val retrofit: Retrofit = Retrofit.Builder()
            .baseUrl("https://www.flickr.com/")
            .addConverterFactory(ScalarsConverterFactory.create())
            .build()

        val flickrApi: FlickrApi = retrofit.create<FlickrApi>()

        viewLifecycleOwner.lifecycleScope.launch {
            val response = flickrApi.fetchContents()
            Log.d(TAG, "Response received: $response")
        }
    }
    ...
}
```

Next, add a function named **`fetchContents()`** to **`PhotoRepository`** to wrap the Retrofit API function you defined for fetching the Flickr home page.

### Listing 20.14 Adding **`fetchContents()`** to **`PhotoRepository`** (`PhotoRepository.kt`)

```
class PhotoRepository {

    private val flickrApi: FlickrApi

    init {
        ...
    }

    suspend fun fetchContents() = flickrApi.fetchContents()
}
```

**`PhotoRepository`** will wrap most of the networking code in PhotoGallery (right now it is small and simple, but it will grow over the next several chapters). Now other components in your app, such as **`PhotoGalleryFragment`** (or some **`ViewModel`**, activity, or other component), can create an instance of **`PhotoRepository`** and request photo data without having to know about Retrofit or the source the data is coming from.

Update **`PhotoGalleryFragment`** to use **`PhotoRepository`** to see this magic in action (Listing 20.15).

### Listing 20.15 Using **`PhotoRepository`** in **`PhotoGalleryFragment`** (`PhotoGalleryFragment.kt`)

```
class PhotoGalleryFragment : Fragment() {
    ...
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        viewLifecycleOwner.lifecycleScope.launch {
            val response = ~~flickrApi~~ PhotoRepository().fetchContents()
            Log.d(TAG, "Response received: $response")
        }
    }
    ...
}
```

This refactor moves your app closer to following the repository pattern you learned about in Chapter 12. **`PhotoRepository`** serves as a very basic repository, encapsulating the logic for accessing data from a single source. It determines how to fetch and store a particular set of data – currently HTML, but later photos. Your UI code will request all the data from the repository, because the UI does not care how the data is actually stored or fetched. Those are implementation details of the repository itself.

Right now, all of your app's data comes directly from the Flickr web server. However, in the future you might decide to cache that data in a local database. In that case, the repository would manage getting the data from the right place. Other components in your app can use the repository to get data without having to know where the data is coming from.

Take a moment to run your app and verify that it still works correctly. You should see the contents of the Flickr home page print to Logcat again, as shown in Figure 20.4.

## Fetching JSON from Flickr

JSON stands for JavaScript Object Notation. It is a popular data format, particularly for web services. You can get more information about JSON as a format at `json.org`.

Flickr offers a fine JSON API. All the details you need are available in the documentation at `flickr.com/services/api`. Pull it up in your favorite web browser and find the list of Request Formats. You will be using the simplest – REST. The REST API endpoint is `api.flickr.com/services/rest`, and you can invoke the methods Flickr provides on this endpoint.

Back on the main page of the API documentation, find the list of API Methods. Scroll down to the interestingness section and click flickr.interestingness.getList. The documentation will report that this method "returns the list of interesting photos for the most recent day or a user-specified date." That is exactly what you want for PhotoGallery.

The only required parameter for the **`getList`** method is an API key. To get an API key, return to `flickr.com/services/api` and follow the link for API Keys. You will need a Flickr ID to log in. Once you are logged in, request a new, noncommercial API key. This usually only takes a moment.

Your API key will look something like `4f721bgafa75bf6d2cb9be54f937bb71`. (That is a fake key we made up as an example – you will need to obtain your own Flickr API key.) You do not need the "Secret," which is only used when an app will access user-specific information or images.

With your shiny new key, you can make a request to the Flickr web service. Your GET request URL will look something like this:

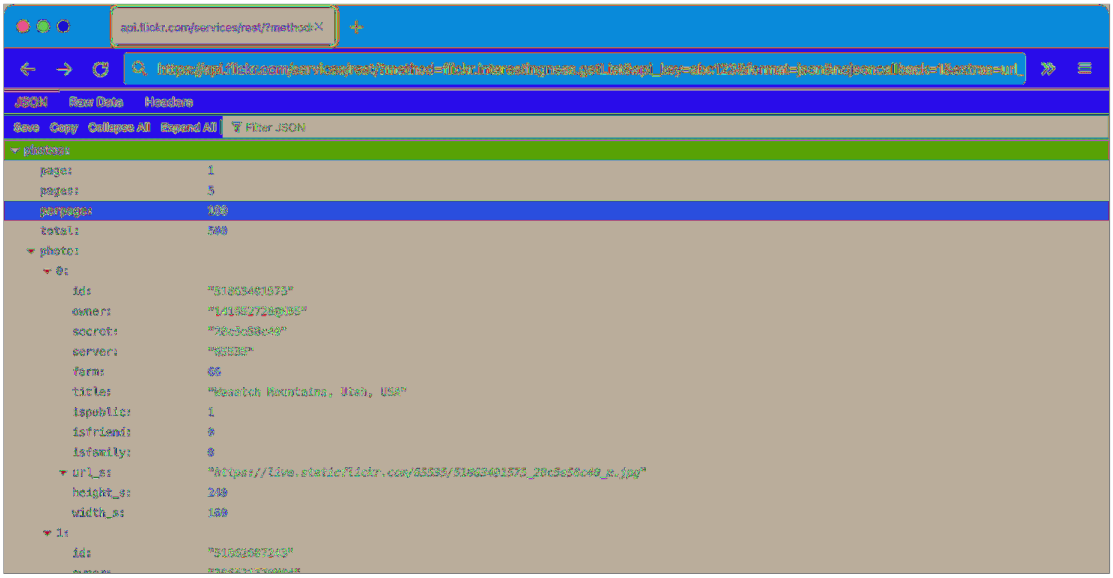
```
https://api.flickr.com/services/rest/?method=flickr.interestingness.getList
  &api_key=yourApiKeyHere&format=json&nojsoncallback=1&extras=url_s
```

The Flickr response is in XML format by default. To get a valid JSON response, you need to specify values for both the `format` and `nojsoncallback` parameters. Setting `nojsoncallback` to `1` tells Flickr to exclude the enclosing method name and parentheses from the response it sends back. This lets your Kotlin code more easily parse the response.

Specifying the parameter called `extras` with the value `url_s` tells Flickr to include the URL for the small version of the picture if it is available.

Copy the example URL into your browser, replacing *yourApiKeyHere* with your actual API key. This will allow you to see an example of what the response data will look like, as shown in Figure 20.5. (Your results may be formatted differently, depending on your browser. But however it is laid out, you should see text like “photos,” “page,” “pages,” and so on.)

Figure 20.5 Sample JSON output



Time to update your existing networking code to request data for recent interesting photos from the Flickr REST API instead of requesting the contents of Flickr's home page. First, add a function to your **`FlickrApi`** API interface. Again, replace *`yourApiKeyHere`* with your API key. For now, hardcode the URL query parameters in the relative path string. (Later, you will abstract these query parameters out and add them in programmatically.)

### Listing 20.16 Defining the "fetch recent interesting photos" request (`api/FlickrApi.kt`)

```
private const val API_KEY = "yourApiKeyHere"

interface FlickrApi {
    @GET("/")
    suspend fun fetchContents() : String
    @GET(
        "services/rest/?method=flickr.interestingness.getList" +
            "&api_key=$API_KEY" +
            "&format=json" +
            "&nojsoncallback=1" +
            "&extras=url_s"
    )
    suspend fun fetchPhotos(): String
}
```

Notice that you added values for the `method`, `api_key`, `format`, `nojsoncallback`, and `extras` parameters.

Next, update the Retrofit instance configuration code in **`PhotoRepository`**. Change the base URL from Flickr's home page to the base API endpoint. Rename the **`fetchContents()`** function to **`fetchPhotos()`** and call through to the new **`fetchPhotos()`** function on the API interface.

### Listing 20.17 Updating the base URL (`PhotoRepository.kt`)

```
class PhotoRepository {
    private val flickrApi: FlickrApi

    init {
        val retrofit: Retrofit = Retrofit.Builder()
            .baseUrl("https://www.flickr.com/")
            .addConverterFactory(ScalarsConverterFactory.create())
            .build()
        flickrApi = retrofit.create()
    }

    suspend fun fetchContents() = flickrApi.fetchContents()
    suspend fun fetchPhotos() = flickrApi.fetchPhotos()
}
```

The base URL you set is `api.flickr.com/`, but the endpoints you want to hit are at `api.flickr.com/services/rest`. This is because you specified the `services` and `rest` parts of the path in your `@GET` annotation in **`FlickrApi`**. The path and other information you included in the `@GET` annotation will be appended onto the URL by Retrofit before it issues the web request.

Finally, update **PhotoGalleryFragment** to execute the web request so that it fetches recent interesting photos instead of the contents of Flickr's home page. Replace the **fetchContents()** call with a call to the new **fetchPhotos()** function. For now, serialize the response into a string, as you did previously.

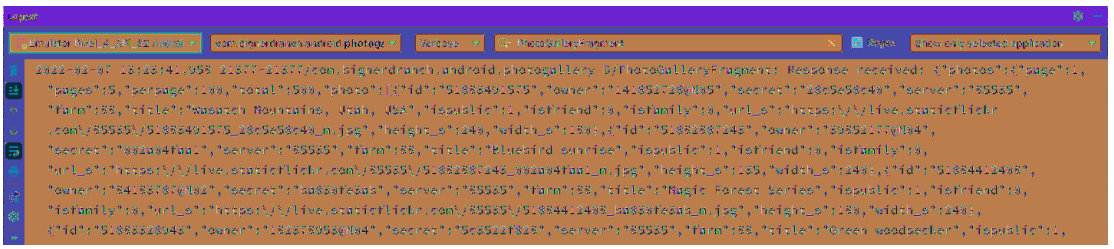
### Listing 20.18 Executing the "fetch recent interesting photos" request (PhotoGalleryFragment.kt)

```
class PhotoGalleryFragment : Fragment() {
    ...
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        viewLifecycleOwner.lifecycleScope.launch {
            val response = PhotoRepository().fetchContent()
            val response = PhotoRepository().fetchPhotos()
            Log.d(TAG, "Response received: $response")
        }
    }
    ...
}
```

Making these few tweaks to your existing code renders your app ready to fetch and log Flickr data. Run PhotoGallery, and you should see rich, fertile Flickr JSON in Logcat, like Figure 20.6. (It will help to search for PhotoGalleryFragment in the Logcat search box.)

### Figure 20.6 Flickr JSON in Logcat




(Logcat can be finicky. Do not panic if you do not get results like ours. Sometimes the connection to the emulator is not quite right and the log messages do not get printed out. Usually it clears up over time, but sometimes you have to rerun your application or even restart your emulator.)

As of this writing, the Android Studio Logcat window does not automatically wrap the output the way Figure 20.6 shows. Scroll to the right to see more of the extremely long JSON response string. Or wrap the Logcat contents by clicking the button on Logcat's left side, shown in Figure 20.6.

Now that you have such fine JSON data from Flickr, what should you do with it? You will do what you do with all data – put it in one or more model objects. The model class you are going to create for PhotoGallery is called **`GalleryItem`**. A gallery item holds meta information for a single photo, including the title, the ID, and the URL to download the image from.

Create the **`GalleryItem`** data class within the `api` subpackage and add the following code:

Listing 20.19 Creating a model object class (`GalleryItem.kt`)

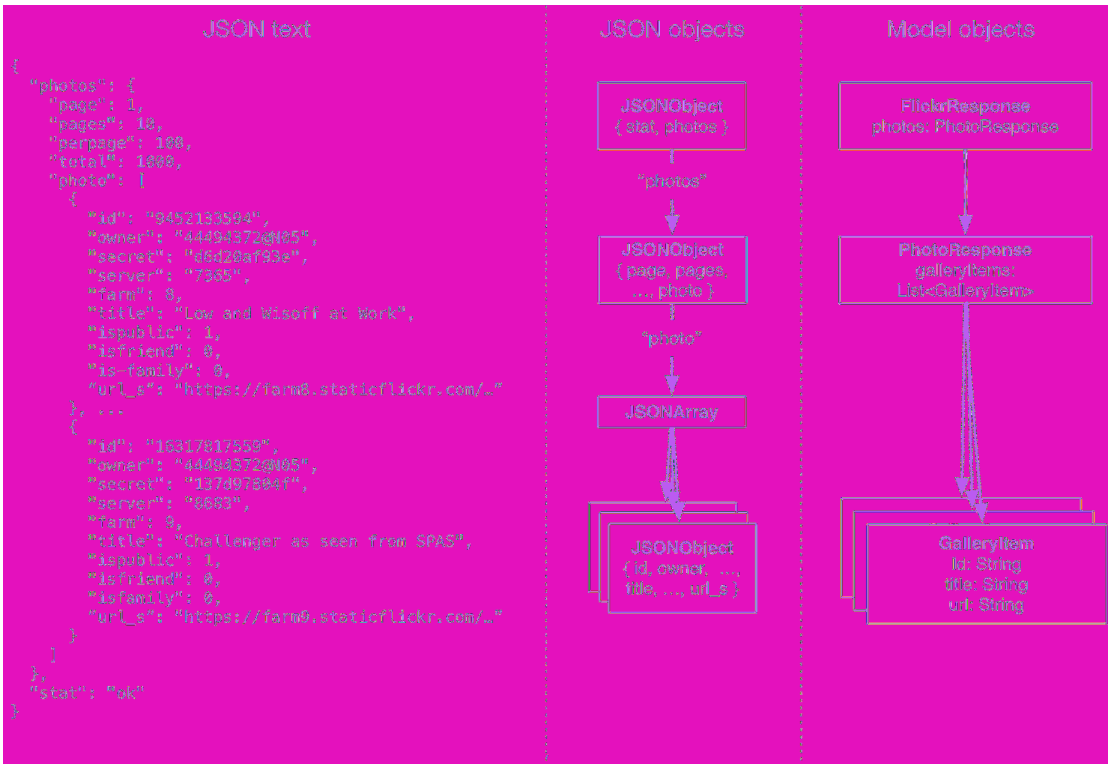
```
data class GalleryItem(
    val title: String,
    val id: String,
    val url: String,
)
```

Now that you have defined a model object, it is time to create and populate instances of that object with data from the JSON output you got from Flickr.

## Deserializing JSON text into model objects

The JSON response displayed in your browser and Logcat window is hard to read. If you *pretty print* the response (format it with white space), it looks something like the text on the left in Figure 20.7.

Figure 20.7 JSON text, JSON hierarchy, and corresponding model objects



A JSON object is a set of name-value pairs enclosed between curly braces, `{ }`. A JSON array is a comma-separated list of JSON objects enclosed in square brackets, `[ ]`. And JSON objects can be nested within each other, resulting in a hierarchy like the one in the middle column of Figure 20.7. (The right side of Figure 20.7 shows the **`GalleryItem`** and the other model objects you will create shortly to represent this data.)

Android includes the standard `org.json` package, which has classes that provide access to creating and parsing JSON text (such as **`JSONObject`** and **`JSONArray`**). However, lots of smart people have created libraries to simplify the process of converting JSON text to Kotlin objects and back again.

One such library is Moshi (`github.com/square/moshi`). Another library from Square, Moshi maps JSON data to Kotlin objects for you automatically. This means you do not need to write any parsing code. Instead, you define Kotlin classes that map to the JSON hierarchy of objects and let Moshi do the rest.

Using Moshi is similar to using the Room database library's `@Entity` data classes. Moshi uses code generation to map JSON to Kotlin classes for you. You will annotate your relevant code, and Moshi will generate code that adapts JSON strings into instances of Kotlin classes. Moshi also has the functionality to parse strings into Kotlin classes dynamically at runtime, but the code generation approach is more performant and easier to set up.

To configure Moshi to do all those things for you, first enable the same `kapt` plugin you used with Room. It is defined at the project level, so add the following line to the `build.gradle` file labeled (Project: PhotoGallery):

### Listing 20.20 Enabling kapt (`build.gradle`)

```
plugins {
    id 'com.android.application' version '7.1.2' apply false
    id 'com.android.library' version '7.1.2' apply false
    id 'org.jetbrains.kotlin.android' version '1.6.10' apply false
    id 'org.jetbrains.kotlin.kapt' version '1.6.10' apply false
}
...
```

Once you have enabled the plugin, apply it to your app's build process in `app/build.gradle`. Include the core library as well as the library that performs the code generation in your dependencies. Finally, Square created a Moshi converter for Retrofit that makes it easy to plug Moshi into your Retrofit implementation. Add the Retrofit Moshi converter library dependencies as well. As always, be sure to sync the file when you are done.

### Listing 20.21 Adding Moshi dependencies (`app/build.gradle`)

```
plugins {
    id 'com.android.application'
    id 'org.jetbrains.kotlin.android'
    id 'org.jetbrains.kotlin.kapt'
}

android {
    ...
}

dependencies {
    ...
    implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-android:1.6.0'
    implementation 'com.squareup.retrofit2:converter-scalars:2.9.0'
    implementation 'com.squareup.moshi:moshi:1.13.0'
    kapt 'com.squareup.moshi:moshi-kotlin-codegen:1.13.0'
    implementation 'com.squareup.retrofit2:converter-moshi:2.9.0'
    ...
}
```

With your dependencies in place, create model objects that map to the JSON data in the Flickr response. You already have **`GalleryItem`**, which maps almost directly to an individual object in the `"photo"` JSON array. By default, Moshi maps JSON object names to property names. If your property names match the JSON object names, you can leave them as is.

But your property names do not need to match the JSON object names. Take your `GalleryItem.url` property, versus the `"url_s"` field in the JSON data. `GalleryItem.url` is more meaningful in the context of your codebase, so it is better to keep it. In this case, you can add a `@Json` annotation to the property to tell Moshi which JSON field the property maps to.

To generate the code to adapt the JSON string into a **`GalleryItem`**, you need to annotate the class with the `@JsonClass` annotation. This will tell Moshi to perform its code generation work during compilation. Update **`GalleryItem`** with these annotations now.

### Listing 20.22 Integrating Moshi (`GalleryItem.kt`)

```
@JsonClass(generateAdapter = true)
data class GalleryItem(
    val title: String,
    val id: String,
    @Json(name = "url_s") val url: String,
)
```

Now, create a **PhotoResponse** class to map to the "photos" object in the JSON data. Place the new class in the api package as well.

Include a property called galleryItems to store a list of gallery items and annotate it with @Json(name = "photo"). Moshi will automatically create a list and populate it with gallery item objects based on the JSON array named "photo".

### Listing 20.23 Adding **PhotoResponse** (PhotoResponse.kt)

```
@JsonClass(generateAdapter = true)
data class PhotoResponse(
    @Json(name = "photo") val galleryItems: List<GalleryItem>
)
```

Right now, the only data you care about in this particular object is the array of photo data in the "photo" JSON object. Later in this chapter, you will want to capture the paging data if you choose to do the challenge in the section called Challenge: Paging.

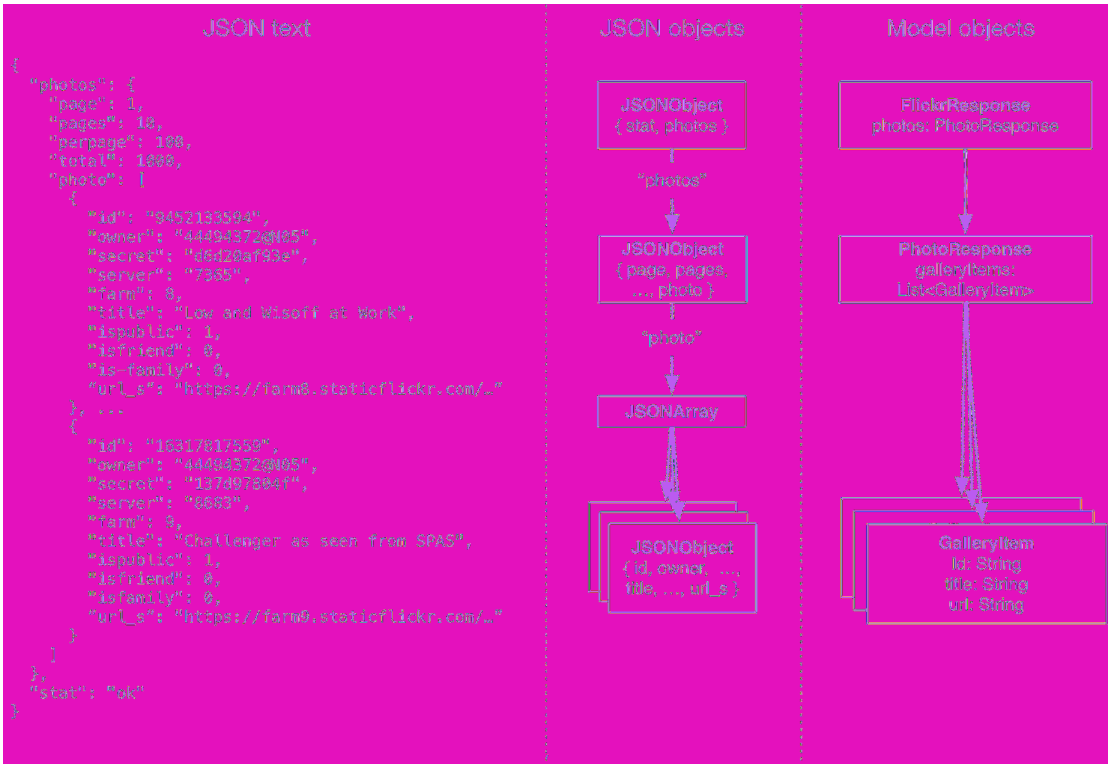
Finally, add a class named **FlickrResponse** to the api package. This class will map to the outermost object in the JSON data (the one at the top of the JSON object hierarchy, denoted by the outermost { }). Add a property to map to the "photos" field.

### Listing 20.24 Adding **FlickrResponse** (FlickrResponse.kt)

```
@JsonClass(generateAdapter = true)
data class FlickrResponse(
    val photos: PhotoResponse
)
```

Take another look at the diagram comparing the JSON text to model objects (copied below in Figure 20.8) and notice how the objects you created map to the JSON data.

Figure 20.8 PhotoGallery data and model objects




Now it is time to make the magic happen: to configure Retrofit to use Moshi to deserialize your data into the model objects you just defined. First, update the return type specified in the Retrofit API interface to **`FlickrResponse`** – the model object you defined to map to the outermost JSON object. This indicates to Moshi that it should use the **`FlickrResponse`** to deserialize the JSON response data.

Listing 20.25 Updating **`fetchPhoto()`**'s return type (`FlickrApi.kt`)

```
interface FlickrApi {
    @GET(...)
    fun fetchPhotos(): ~~String~~FlickrResponse
}
```

Next, update **`PhotoRepository`**. Swap out the scalars converter factory for a Moshi converter factory and update **`fetchPhotos()`** to return the list of gallery items.

### Listing 20.26 Updating **`PhotoRepository`** for Moshi (`PhotoRepository.kt`)

```
class PhotoRepository {
    private val flickrApi: FlickrApi

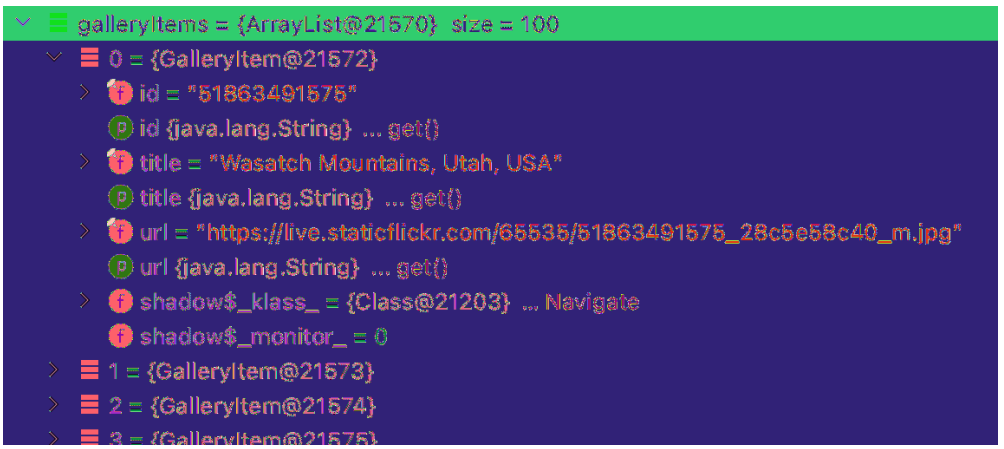
    init {
        val retrofit: Retrofit = Retrofit.Builder()
            .baseUrl("https://api.flickr.com/")
            .addConverterFactory(ScalarsConverterFactory.create())
            .addConverterFactory(MoshiConverterFactory.create())
            .build()
        flickrApi = retrofit.create()
    }

    suspend fun fetchPhotos() = flickrApi.fetchPhotos()
    suspend fun fetchPhotos(): List<GalleryItem> =
        flickrApi.fetchPhotos().photos.galleryItems
}
```

Now that you are no longer using the scalars converter factory, you do not need the `retrofit2.converter.scalars` imports in `PhotoRepository.kt` and `PhotoGalleryFragment.kt`. They might disappear on their own, but if not you should delete them, because they may cause errors.

You do not need to make any changes to the **`PhotoGalleryFragment`** class, since you are only logging out the result. Run PhotoGallery to test your JSON parsing code. You should see the logging output of the gallery item list printed to Logcat. If you want explore the results further, set a breakpoint on the logging line in the lambda and use the debugger to drill down through `galleryItems` (Figure 20.9).

### Figure 20.9 Exploring the Flickr response




If you run into any issues, make sure that your web request is properly formatted. In some cases (such as when the API key is invalid), the Flickr API will return an error response and Moshi will fail to initialize your models. In the next section, you will handle situations where things do not go according to plan.

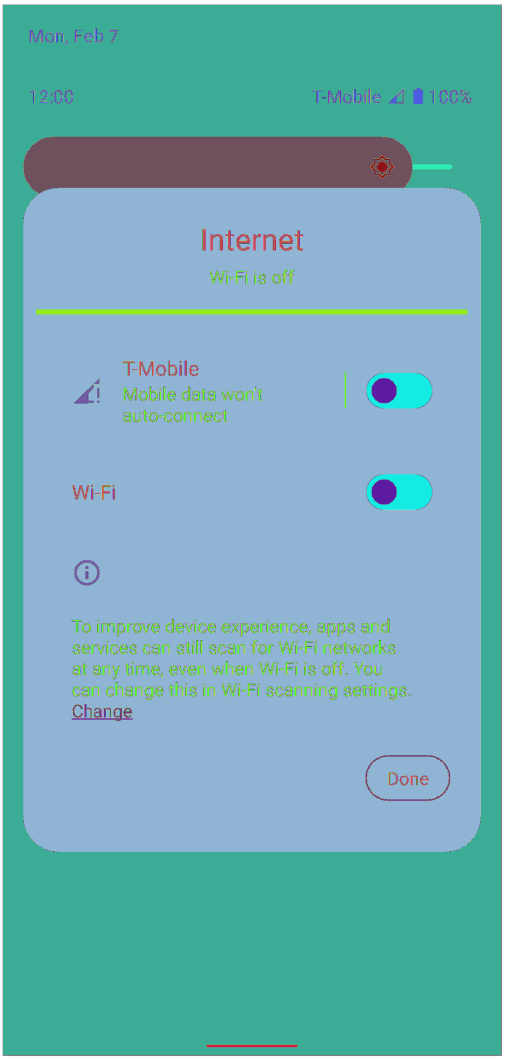
## Handling errors

There are hundreds of ways in which a network request can go wrong. The device might not be connected to the internet. The server might be down and fail to respond to the request. There might be an issue with the contents of your request or the server's response.

In those cases, you will not have an easy-to-use **`FlickrResponse`** to mess with in your code. You will need to handle errors related to these situations yourself.

To model a common network issue, turn off internet access on your device or emulator: Swipe down from the top of the screen to open Quick Settings. Press the Internet icon, and toggle off the mobile data and WiFi options (Figure 20.10). Press Done.

Figure 20.10 Turning off the internet




(The steps to disable internet access might vary, depending on the version of Android. For example, you might instead need to look for separate WiFi and mobile data settings to disable.)

Next, navigate to the overview screen and kill PhotoGallery, if it is running. Finally, try relaunching PhotoGallery. You will see it display briefly – and then crash when it makes a network request that cannot be successfully completed.

There are many ways to handle network request errors through Retrofit, depending on the source of the error and the experience you want to provide to users. But at a minimum, your app should avoid crashing when there is an error with networking.

To prevent a crash in PhotoGallery, you will use Kotlin's `try/catch` syntax. In **`PhotoGalleryFragment`**, wrap your network request in a `try/catch` block and log out the error if an exception is thrown.

### Listing 20.27 Handling network errors (`PhotoGalleryFragment.kt`)

```
class PhotoGalleryFragment : Fragment() {
    ...
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        viewLifecycleOwner.lifecycleScope.launch {
          try {
              val response = PhotoRepository().fetchPhotos()
              Log.d(TAG, "Response received: $response")
          } catch (ex: Exception) {
              Log.e(TAG, "Failed to fetch gallery items", ex)
          }
        }
    }
    ...
}
```

Compile and run your app again. If you look at Logcat, you will see an error logged, but the app will keep running.

Re-enable internet access on your device or emulator before moving on.

## Networking Across Configuration Changes

Now that you have your app deserializing JSON into model objects, take a closer look at how your implementation behaves across a configuration change. Run your app, make sure auto-rotate is turned on for your device or emulator, and then rotate the device quickly five or so times in a row. Inspect the Logcat output, filtering by PhotoGalleryFragment and turning soft wraps off.

```
15:49:07.304 D/PhotoGalleryFragment: Response received: [GalleryItem(...
15:49:16.794 D/PhotoGalleryFragment: Response received: [GalleryItem(...
15:49:20.098 D/PhotoGalleryFragment: Response received: [GalleryItem(...
15:49:23.565 D/PhotoGalleryFragment: Response received: [GalleryItem(...
15:49:27.043 D/PhotoGalleryFragment: Response received: [GalleryItem(...
15:49:30.099 D/PhotoGalleryFragment: Response received: [GalleryItem(...
...
```

What is going on here? A new network request is made every time you rotate the device. This is because you kick off the request in **`onViewCreated(…)`**. Since the fragment is destroyed and re-created every time you rotate, a new request is issued to (unnecessarily) re-download the data.

This is problematic because you are doing duplicate work – you should instead issue a download request when the fragment is first created. That same request (and the resulting data) should persist across rotation to ensure a speedy user experience (and to avoid unnecessarily using up the user's data if they are not on WiFi).

Instead of launching a new web request every time a configuration change occurs, you need to fetch the photo data once, when the fragment is initially created and displayed onscreen. Then you can allow the web request to continue to execute when a configuration change occurs by caching the results in memory for the perceived life of the fragment, across any and all configuration changes (such as rotation). Finally, you can use these cached results when available rather than making a new request.

**`ViewModel`** is the right tool to help you with this job.

You already added the **ViewModel** dependency to the project, so go ahead and create a **ViewModel** class named **PhotoGalleryViewModel**. This **ViewModel** will look very similar to the **ViewModel**s in CriminalIntent. Use a **StateFlow** to expose a list of gallery items to the fragment. Kick off a web request to fetch photo data when the **ViewModel** is first initialized, and stash the resulting data in the property you created. Use a `try/catch` block to handle any errors.

When you are done, your code should match Listing 20.28.

### Listing 20.28 Shiny new **ViewModel** (PhotoGalleryViewModel.kt)

```
private const val TAG = "PhotoGalleryViewModel"

class PhotoGalleryViewModel : ViewModel() {
    private val photoRepository = PhotoRepository()

    private val _galleryItems: MutableStateFlow<List<GalleryItem>> =
        MutableStateFlow(emptyList())
    val galleryItems: StateFlow<List<GalleryItem>>
        get() = _galleryItems.asStateFlow()

    init {
        viewModelScope.launch {
            try {
                val items = photoRepository.fetchPhotos()
                Log.d(TAG, "Items received: $items")
                _galleryItems.value = items
            } catch (ex: Exception) {
                Log.e(TAG, "Failed to fetch gallery items", ex)
            }
        }
    }
}
```

Recall that the first time a **ViewModel** is requested for a given lifecycle owner, a new instance of the **ViewModel** is created. Successive requests for the **ViewModel** return the same instance that was originally created.

You call **PhotoRepository().fetchPhotos()** in **PhotoGalleryViewModel**'s `init{}` block. This kicks off the request for photo data when the **ViewModel** is first created. Since the **ViewModel** is only created once in the lifecycle owner's lifetime (when queried from the **ViewModelProvider** class for the first time), the request will only be made once (when the user launches **PhotoGalleryFragment**).

When the user rotates the device or some other configuration change occurs, the **ViewModel** will remain in memory, and the re-created version of the fragment will be able to access the results of the original request through the **ViewModel**.

Thanks to coroutines, when the `viewModelScope` is canceled, your network request will also be canceled. But in a production app, you might cache the results in a database or some other local storage, so it would make sense to let the fetch continue to completion.

Update **PhotoGalleryFragment** to get access to the **PhotoGalleryViewModel**. Remove the existing code that interacts with **PhotoRepository**, since **PhotoGalleryViewModel** handles that now.

Also, update **PhotoGalleryFragment** to observe **PhotoGalleryViewModel**'s **StateFlow** once the fragment's view is created. For now, log a statement indicating the data was received. Eventually you will use these results to update your recycler view contents.

### Listing 20.29 Getting a **ViewModel** instance from the provider (PhotoGalleryFragment.kt)

```
class PhotoGalleryFragment : Fragment() {
    private var _binding: FragmentPhotoGalleryBinding? = null
    private val binding
        get() = checkNotNull(_binding) {
            "Cannot access binding because it is null. Is the view visible?"
        }

    private val photoGalleryViewModel: PhotoGalleryViewModel by viewModels()
    ...
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        viewLifecycleOwner.lifecycleScope.launch {
            try {
                val response = PhotoRepository().fetchPhotos()
                Log.d(TAG, "Response received: $response")
            } catch (ex: Exception) {
                Log.e(TAG, "Failed to fetch gallery items", ex)
            }
            viewLifecycleOwner.repeatOnLifecycle(Lifecycle.State.STARTED) {
                photoGalleryViewModel.galleryItems.collect { items ->
                  Log.d(TAG, "Response received: $items")
                }
            }
        }
    }
    ...
}
```

Eventually you will update UI-related things (such as the recycler view adapter) in response to data changes. Starting the observation in **onViewCreated(…)** ensures that the UI views and other related objects will be ready. It also ensures that you properly handle the situation where the fragment becomes detached and its view is destroyed. In this scenario, the view will be re-created when the fragment is reattached, and the subscription will be added to the new view once it is created.

Run your app. Filter Logcat by PhotoGalleryViewModel. Rotate the emulator multiple times. You should only see `PhotoGalleryViewModel: Items received` printed to the Logcat window one time, no matter how many times you rotate.

# Displaying Results in RecyclerView

For your last task in this chapter, you will switch to the view layer and get **PhotoGalleryFragment**'s **RecyclerView** to display some images.

Start by creating a layout for an individual list item. In the project tool window, right-click the res/layout directory and choose New → Layout Resource File. Name the file list_item_gallery, set the root element to ImageView, and click OK.

Within the layout, update and add a few XML attributes. The **RecyclerView** will provide the list item with an appropriate width, so keep the `android:layout_width` set to `match_parent`. But limit the height to `120dp`. That will allow the user to see multiple rows of images within your **RecyclerView** onscreen at once.

Flickr does not standardize photo sizing, so you cannot be fully sure of the dimensions of the image. There are two attributes you can use to provide a good experience regardless of any particular image's dimensions.

First, set `android:scaleType` to `centerCrop`. This will make the image spread its contents all the way to the edges of the **ImageView**, while maintaining its aspect ratio. That means that both of the image's dimensions will be equal to or larger than the dimensions of the **ImageView**. Any part of the image that goes beyond the dimensions of the **ImageView** will be cropped off.

Also, set the `android:layout_gravity` attribute to `center`. This will center the image both vertically and horizontally within the dimensions of the **ImageView**.

### Listing 20.30 Updating the list item layout file (`layout/list_item_gallery.xml`)

```
<?xml version="1.0" encoding="utf-8"?>
<ImageView xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/item_image_view"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_height="120dp"
    android:layout_gravity="center"
    android:scaleType="centerCrop" />

</ImageView>
```

With the layout defined, you can start on the Kotlin code. As you may recall from Chapter 10, you will need to create two Kotlin classes: one that will extend **RecyclerView.ViewHolder** and another that will extend **RecyclerView.Adapter**.

First, you will create **PhotoViewHolder**, which extends **RecyclerView.ViewHolder**. It will be responsible for holding onto an instance of the view for the layout you just created and binding a **GalleryItem** to that view. Next, you will create **PhotoListAdapter**, which extends **RecyclerView.Adapter**. It will manage the communication between the **RecyclerView** and the backing data, providing **PhotoViewHolder** instances to the **RecyclerView** and binding those instances with a **GalleryItem** at the correct position.

Let's get started. Create a new file named `PhotoListAdapter.kt`. Define a **`PhotoViewHolder`** class in your new file. It should take in a **`ListItemGalleryBinding`** as a constructor parameter and have a **`bind(galleryItem: GalleryItem)`** function to update itself with the data from a **`GalleryItem`**. You will fill out the body of this function shortly.

### Listing 20.31 Adding a `ViewHolder` implementation (`PhotoListAdapter.kt`)

```
class PhotoViewHolder(
    private val binding: ListItemGalleryBinding
) : RecyclerView.ViewHolder(binding.root) {
    fun bind(galleryItem: GalleryItem) {
        // TODO
    }
}
```

Next, add a **`RecyclerView.Adapter`** to provide **`PhotoViewHolder`**s as needed, based on a list of **`GalleryItem`**s.

### Listing 20.32 Adding a `RecyclerView.Adapter` implementation (`PhotoListAdapter.kt`)

```
class PhotoViewHolder(
    private val binding: ListItemGalleryBinding
) : RecyclerView.ViewHolder(binding.root) {
    ...
}

class PhotoListAdapter(
    private val galleryItems: List<GalleryItem>
) : RecyclerView.Adapter<PhotoViewHolder>() {
    override fun onCreateViewHolder(
        parent: ViewGroup,
        viewType: Int
    ): PhotoViewHolder {
        val inflater = LayoutInflater.from(parent.context)
        val binding = ListItemGalleryBinding.inflate(inflater, parent, false)
        return PhotoViewHolder(binding)
    }

    override fun onBindViewHolder(holder: PhotoViewHolder, position: Int) {
        val item = galleryItems[position]
        holder.bind(item)
    }

    override fun getItemCount() = galleryItems.size
}
```

Now that you have the appropriate nuts and bolts in place for **`RecyclerView`**, attach an adapter with updated gallery item data when the **`StateFlow`** emits a new value.

Listing 20.33 Adding an adapter to the recycler view when data is available or changed (`PhotoGalleryFragment.kt`)

```
class PhotoGalleryFragment : Fragment() {
    ...
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        viewLifecycleOwner.lifecycleScope.launch {
          viewLifecycleOwner.repeatOnLifecycle(Lifecycle.State.STARTED) {
              photoGalleryViewModel.galleryItems.collect { items ->
                  Log.d(TAG, "Response received: $items")
                  binding.photoGrid.adapter = PhotoListAdapter(items)
              }
          }
      }
    }
    ...
}
```

## Displaying images

All the infrastructure is in place to display the images. Unfortunately, you cannot reuse your code from CriminalIntent to display the images this time. For starters, the images are not saved to the device – they will be coming from the internet.

Also, it is more important than ever to efficiently perform this work. You are no longer displaying a single image, like you did in CriminalIntent. This time, you are displaying more than 20. And if the user scrolls down the grid, you will need to load dozens more. This will lead to a significant amount of computing power and memory usage for a seemingly straightforward task. So you need to accomplish as much of that work as possible off the UI thread, because the UI needs to be responsive even while loading all the images.

Efficient image loading is a hard problem. You need to worry about network connections, juggling images across threads, caching images, resizing images to fit their containers, canceling requests when images are no longer needed, and much more. Still, if you wanted to, you could manually write the image loading code you need. In fact, in previous editions of this book, we dedicated an entire chapter to accomplishing this task.

But you should not write this code yourself if you do not have to. And – thanks to modern Android tooling – you do not.

Today, just as there are many libraries for parsing JSON or performing network requests, there are many libraries to help you download and display images on Android. Commonly used ones include Picasso (which is also from Square) and Glide.

In PhotoGallery, you will use Coil, originally developed at Instacart. Coil leverages all the convenient features of the modern Kotlin language and integrates seamlessly with coroutines to manage performing work in the background.

To include Coil as a dependency, add it to `app/build.gradle`.

### Listing 20.34 Adding Coil (`app/build.gradle`)

```
...
dependencies {
    ...
    implementation 'com.squareup.retrofit2:converter-moshi:2.9.0'
    implementation 'io.coil-kt:coil:2.0.0-rc02'
    testImplementation 'junit:junit:4.13.2'
    ...
}
```

Do not forget to sync your Gradle files.

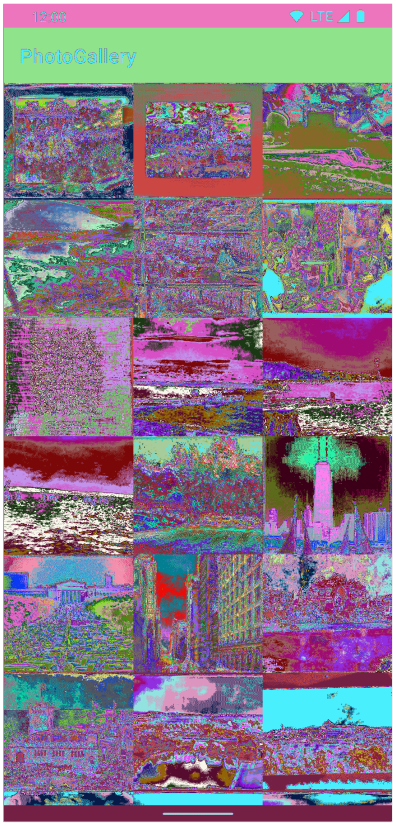
Coil is a highly customizable library, but the basic usage is simple: When binding data to the **`PhotoViewHolder`**, call the **`load()`** extension function that Coil provides for the **`ImageView`**. Pass in the `url` property from the **`GalleryItem`**, and Coil will handle the rest.

### Listing 20.35 Loading the image (`PhotoListAdapter.kt`)

```
class PhotoViewHolder(
    private val binding: ListItemGalleryBinding
) : RecyclerView.ViewHolder(binding.root) {
    fun bind(galleryItem: GalleryItem) {
        // TODO
        binding.itemImageView.load(galleryItem.url)
    }
}
...
```

That is it! Run PhotoGallery and admire the interesting photos it displays (Figure 20.11):

Figure 20.11 Interestingness




Coil also has some fun features that go beyond the basics. You can automatically crop images into a circle. You can cross-fade them as they come in from the network. You can also display a placeholder image while the real image is being downloaded from the internet.

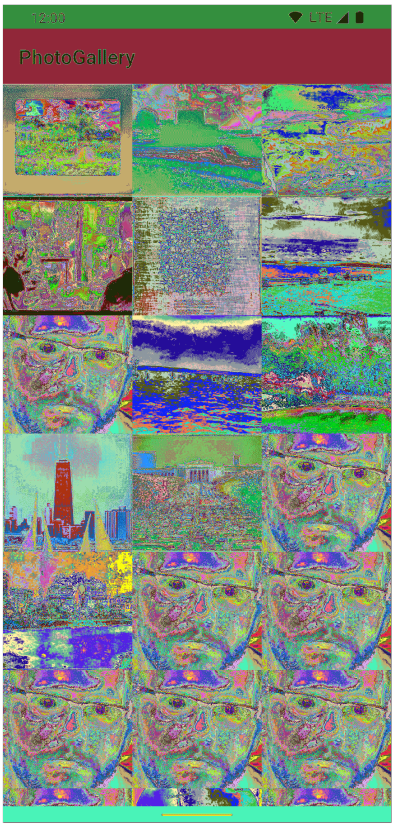
All these customizations can be configured within a lambda expression that is an optional parameter on the **`load()`** function you just used. To see how this works, add a placeholder image so your users do not have to look at a blank screen while they wait for photos to load. Find `bill_up_close.png` in the solutions file (`www.bignerdranch.com/android-5e-solutions`) and put it in `res/drawable`. Use it as a placeholder while images download from Flickr.

### Listing 20.36 Loading the image (`PhotoListAdapter.kt`)

```
class PhotoViewHolder(
    private val binding: ListItemGalleryBinding
) : RecyclerView.ViewHolder(binding.root) {
    fun bind(galleryItem: GalleryItem) {
        binding.itemImageView.load(galleryItem.url) {
          placeholder(R.drawable.bill_up_close)
      }
    }
}
...
```

Run your app again and watch for the placeholder image to fill the recycler view and then disappear as the images from Flickr arrive (Figure 20.12).

### Figure 20.12 A Billsplosion



## For the More Curious: Managing Dependencies

**PhotoRepository** provides a layer of abstraction over the source of Flickr photo metadata. Other components (such as **PhotoGalleryFragment**) use this abstraction to fetch Flickr data without worrying about where the data is coming from.

**PhotoRepository** itself does not know how to download JSON data from Flickr. Instead, **PhotoRepository** relies on **FlickrApi** to know the endpoint URL, to connect to that endpoint, and to perform the actual work of downloading the JSON data. **PhotoRepository** is said to have a *dependency* on **FlickrApi**.

You are initializing **FlickrApi** inside the **PhotoRepository** `init` block:

```
class PhotoRepository {
    ...
    init {
        val retrofit: Retrofit = Retrofit.Builder()
            .baseUrl("https://api.flickr.com/")
            .addConverterFactory(MoshiConverterFactory.create())
            .build()
        flickrApi = retrofit.create()
    }
    ...
}
```

This works well for a simple application, but there are a few potential issues to consider.

First, it is difficult to unit test **PhotoRepository**. Recall from Chapter 6 that the goal of a unit test is to verify the behavior of a class and its interactions with other classes. To properly unit test **PhotoRepository**, you need to isolate it from the real **FlickrApi**. But this is a difficult – if not impossible – task, because **FlickrApi** is initialized inside the **PhotoRepository** `init` block.

Hence, there is no way to provide a mock instance of **FlickrApi** to **PhotoRepository** for testing purposes. This is problematic, because any test you run against **fetchPhotos()** will result in a network request. The success of your tests would be dependent on network state and the availability of the Flickr back-end API at the time of running the test.

Another issue is that **FlickrApi** is tedious to instantiate. You must build and configure an instance of **Retrofit** before you can build an instance of **FlickrApi**. This implementation requires you to duplicate five lines of Retrofit configuration code anywhere you want to create a **FlickrApi** instance.

Finally, creating a new instance of **FlickrApi** everywhere you want to use it results in unnecessary object creation. Object creation is expensive relative to the scarce resources available on a mobile device. Whenever practical, you should share instances of a class across your app and avoid needless object allocation. **FlickrApi** is a perfect candidate for sharing, since there is no variable instance state.

*Dependency injection* (or DI) is a design pattern that addresses these issues by centralizing the logic for creating dependencies, such as **`FlickrApi`**, and supplying the dependencies to the classes that need them. By applying DI to PhotoGallery, you could easily pass an instance of **`FlickrApi`** into **`PhotoRepository`** each time a new instance of **`PhotoRepository`** was constructed. Using DI would allow you to:

- encapsulate the initialization logic of **`FlickrApi`** into a common place outside of **`PhotoRepository`**
- use a singleton instance of **`FlickrApi`** throughout the app
- substitute a mock version of **`FlickrApi`** when unit testing

Applying the DI pattern to **`PhotoRepository`** might look something like this:

```
class PhotoRepository(private val flickrApi: FlickrApi) {
    suspend fun fetchPhotos(): List<GalleryItem> =
        flickrApi.fetchPhotos().photos.galleryItems
}
```

Note that DI does not enforce the singleton pattern for all dependencies. **`PhotoRepository`** is passed an instance of **`FlickrApi`** on construction. This mechanism for constructing **`PhotoRepository`** gives you the flexibility to provide a new instance or a shared instance of **`FlickrApi`** based on your use case.

DI is a broad topic with many facets that extend well beyond Android. This section just scratches the surface. There are entire books dedicated to the concept of DI and many libraries to make DI easier to implement. If you want to use DI in your app, you should consider using one of these libraries. It will help guide you through the process of DI and reduce the amount of code you need to write to implement the pattern.

At the time of this writing, Dagger 2 and its companion Hilt are the official Google-recommended libraries for implementing DI on Android. You can find detailed documentation, code samples, and tutorials about DI on Android at `dagger.dev/hilt`.

## Challenge: Paging

By default, `getList` returns one page of 100 results. There is an additional parameter you can use called `page` that will let you return page two, page three, and so on.

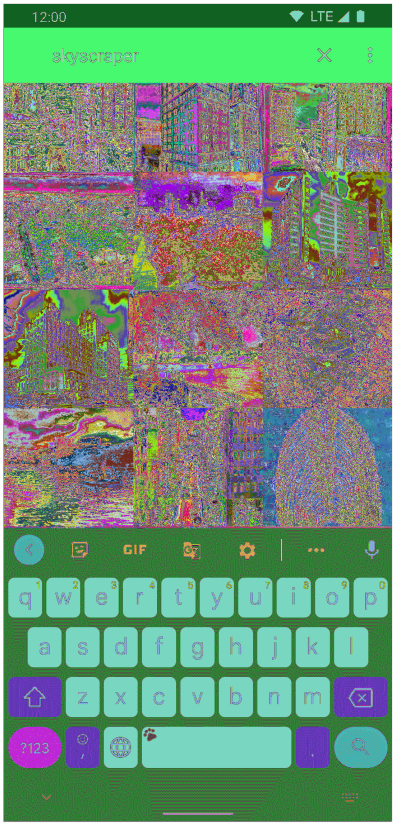
For this challenge, research the Jetpack Paging Library (`developer.android.com/topic/libraries/architecture/paging`) and use it to implement paging for PhotoGallery. This library provides a framework for loading your app's data when it is needed. While you could implement this functionality manually, the paging library will be less work and less prone to error.

# 21
# SearchView and DataStore

Your next task with PhotoGallery is to search photos on Flickr. You will learn how to integrate search into your app using **`SearchView`**. **`SearchView`** is an *action view* class – a view that can be embedded right in your app bar. You will also learn how to easily store data to the device's filesystem using the AndroidX DataStore library.

By the end of this chapter, the user will be able to press the **`SearchView`**, type in a query, and submit it. Submitting the query will send the query string to Flickr's search API and populate the **`RecyclerView`** with the search results (Figure 21.1). The query string will also be persisted to the filesystem. This means the user's last query will be accessible across restarts of the app and even the device.

Figure 21.1 App preview



## Searching Flickr

Let's begin with the Flickr side of things. To search Flickr, you call the **`flickr.photos.search`** method. Here is what a GET request to search for the text "cat" looks like:

```
https://api.flickr.com/services/rest/?method=flickr.photos.search
&api_key=xxx&format=json&nojsoncallback=1&extras=url_s&safe_search=1&text=cat
```

The method is set to **`flickr.photos.search`**. The `text` parameter is set to whatever string you are searching for ("cat," in this case). Setting `safesearch` to `1` filters potentially offensive results from the search data sent back.

Some of the parameter-value pairs, such as `format=json`, are constant across both the **`flickr.photos.search`** and **`flickr.interestingness.getList`** request URLs. You are going to abstract these shared parameter-value pairs out into an *interceptor*.

An interceptor does what you might expect, based on the name – it intercepts a request or response and allows you to manipulate the contents or take some action before the request or response completes. The **`Interceptor`** interface is a part of the OkHttp library, which – as you might remember from Chapter 20 – is the library actually responsible for performing the network requests for Retrofit.

Create a new **`Interceptor`** class named **`PhotoInterceptor`** in your `api` folder. Override **`intercept(chain)`** to access a request, add the shared parameter-value pairs to it, and overwrite the original URL with the newly built URL. (Do not neglect to include your API key, which you created in Chapter 20, in place of *`yourApiKeyHere`*. You can copy it from `api/FlickrApi.kt`.)

### Listing 21.1 Adding an interceptor to insert URL constants (`api/PhotoInterceptor.kt`)

```
private const val API_KEY = "yourApiKeyHere"

class PhotoInterceptor : Interceptor {
    override fun intercept(chain: Interceptor.Chain): Response {
        val originalRequest: Request = chain.request()

        val newUrl: HttpUrl = originalRequest.url.newBuilder()
                .addQueryParameter("api_key", API_KEY)
                .addQueryParameter("format", "json")
                .addQueryParameter("nojsoncallback", "1")
                .addQueryParameter("extras", "url_s")
                .addQueryParameter("safesearch", "1")
                .build()

        val newRequest: Request = originalRequest.newBuilder()
                .url(newUrl)
                .build()

        return chain.proceed(newRequest)
    }
}
```

Android Studio presents you with multiple options when importing **`Request`** and **`Response`**. Select the `okhttp3` options for both.

Here, you call **`chain.request()`** to access the original request. The `originalRequest.url` property contains the original URL from the request, and you use **`HttpUrl.Builder`** to add the query parameters to it. **`HttpUrl.Builder`** creates a new **`Request`** based on the original request and overwrites the original URL with the new one.

Finally, you call **`chain.proceed(newRequest)`** to produce a **`Response`**. If you did not call **`chain.proceed(…)`**, the network request would not happen.

Now, open `PhotoRepository.kt` and add the interceptor to your **`Retrofit`** configuration. Create an **`OkHttpClient`** instance and add **`PhotoInterceptor`** as an interceptor. Then set the newly configured client on your **`Retrofit`** instance. This will replace the default client that was being used; Retrofit will now use the client you provided and apply **`PhotoInterceptor.intercept(…)`** to any requests that are made.

### Listing 21.2 Adding an interceptor to your Retrofit configuration (`PhotoRepository.kt`)

```
class PhotoRepository {
    private val flickrApi: FlickrApi

    init {
        val okHttpClient = OkHttpClient.Builder()
            .addInterceptor(PhotoInterceptor())
            .build()

        val retrofit: Retrofit = Retrofit.Builder()
            .baseUrl("https://api.flickr.com/")
            .addConverterFactory(MoshiConverterFactory.create())
            .client(okHttpClient)
            .build()
        flickrApi = retrofit.create()
    }
    ...
}
```

You no longer need the **`flickr.interestingness.getList`** URL specified in **`FlickrApi`**. Clean that up and, instead, add a **`searchPhotos()`** function to define a search request for your Retrofit API configuration.

### Listing 21.3 Adding a search function to **`FlickrApi`** (`api/FlickrApi.kt`)

```
~~private const val API_KEY = "yourApiKeyHere"~~

interface FlickrApi {
  ~~@GET(~~
      ~~"services/rest/?method=flickr.interestingness.getList" +~~
          ~~"&api_key=$API_KEY" +~~
          ~~"&format=json" +~~
          ~~"&nojsoncallback=1" +~~
          ~~"&extras=url_s"~~
  ~~)~~
  @GET("services/rest/?method=flickr.interestingness.getList")
  suspend fun fetchPhotos(): FlickrResponse

  @GET("services/rest?method=flickr.photos.search")
  suspend fun searchPhotos(@Query("text") query: String): FlickrResponse
}
```

The `@Query` annotation allows you to dynamically append a query parameter appended to the URL. Here you append a query parameter named `text`. The value assigned to `text` depends on the argument passed into **`searchPhotos(String)`**. For example, calling **`searchPhotos("robot")`** would add `text=robot` to the URL.

Add a search function to **`PhotoRepository`** to wrap the newly added **`FlickrApi.searchPhotos(String)`** function.

### Listing 21.4 Adding a search function to **`PhotoRepository`** (`PhotoRepository.kt`)

```
class PhotoRepository {
    ...
    suspend fun fetchPhotos(): List<GalleryItem> =
        flickrApi.fetchPhotos().photos.galleryItems

    suspend fun searchPhotos(query: String): List<GalleryItem> =
        flickrApi.searchPhotos(query).photos.galleryItems
}
```

Finally, update **`PhotoGalleryViewModel`** to kick off a Flickr search. For now, hardcode the search term to be "planets." Hardcoding the query allows you to test your new search code even though you have not yet provided a way to enter a query through the UI. While you are there, delete the debugging log statement. You do not need it anymore.

### Listing 21.5 Kicking off a search request (`PhotoGalleryViewModel.kt`)

```
class PhotoGalleryViewModel : ViewModel() {
    ...
    init {
        viewModelScope.launch {
            try {
                val items = photoRepository.~~fetchPhotos()~~ searchPhotos("planets")
                ~~Log.d(TAG, "Items received: $items")~~
                _galleryItems.value = items
            } catch (ex: Exception) {
                Log.e(TAG, "Failed to fetch gallery items", ex)
            }
        }
    }
}
```

While the search request URL differs from the one you used to request interesting photos, the format of the JSON data returned remains the same. This is good news, because it means you can use the same Moshi configuration and model mapping code you already wrote.

Run PhotoGallery to ensure your search query works correctly. Hopefully, you will see a cool photo or two of Earth. (If you do not get results obviously related to planets, it does not mean your query is not working. Try a different search term – such as "bicycle" or "llama" – and run your app again to confirm that you are indeed seeing search results.)

## Using SearchView

Now that **PhotoRepository** supports searching, it is time to add a way for the user to enter a query and initiate a search. Do this by adding a **SearchView**.

As we said at the beginning of the chapter, **SearchView** is an action view, meaning your entire search interface can live in your application's app bar.

Create a new menu XML file for **PhotoGalleryFragment** called `res/menu/fragment_photo_gallery.xml`. This file will specify the items that should appear in the app bar. (See Chapter 15 for detailed steps on adding the menu XML file if you need a reminder.)

### Listing 21.6 Adding a menu XML file (`res/menu/fragment_photo_gallery.xml`)

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
      xmlns:app="http://schemas.android.com/apk/res-auto">

    <item android:id="@+id/menu_item_search"
          android:title="@string/search"
          app:actionViewClass="androidx.appcompat.widget.SearchView"
          app:showAsAction="ifRoom" />

    <item android:id="@+id/menu_item_clear"
          android:title="@string/clear_search"
          app:showAsAction="never" />
</menu>
```

You will see a couple of errors in the new XML complaining that you have not yet defined the strings you are referencing for the `android:title` attributes. Ignore those for now. You will fix them in a bit.

The first `item` entry in Listing 21.6 tells the app bar to display a **SearchView** by specifying the value `androidx.appcompat.widget.SearchView` for the `app:actionViewClass` attribute. (Notice the usage of the `app` namespace for the `showAsAction` and `actionViewClass` attributes. Refer back to Chapter 15 if you are not sure why this is used.)

The second item in Listing 21.6 will add a "Clear Search" option. This option will always display in the overflow menu, because you set `app:showAsAction` to `never`. Later, you will configure this item so that, when pressed, the user's stored query will be erased from the disk.

Now it is time to address the errors in your menu XML. Open `res/values/strings.xml` and add the missing strings.

### Listing 21.7 Adding search strings (`res/values/strings.xml`)

```
<resources>
    ...
    <string name="search">Search</string>
    <string name="clear_search">Clear Search</string>

</resources>
```

Finally, open `PhotoGalleryFragment.kt`. Add a call to **`setHasOptionsMenu(true)`** in **`onCreate(…)`** to register the fragment to receive menu callbacks. Override **`onCreateOptionsMenu(…)`** and inflate the menu XML file you created. If you forget to call **`setHasOptionsMenu(true)`** in **`onCreate(…)`**, then **`onCreateOptionsMenu(…)`** will never be called and your menu will not appear onscreen. Doing all this will add the items listed in your menu XML to the app bar.

### Listing 21.8 Overriding **`onCreateOptionsMenu(…)`** (`PhotoGalleryFragment.kt`)

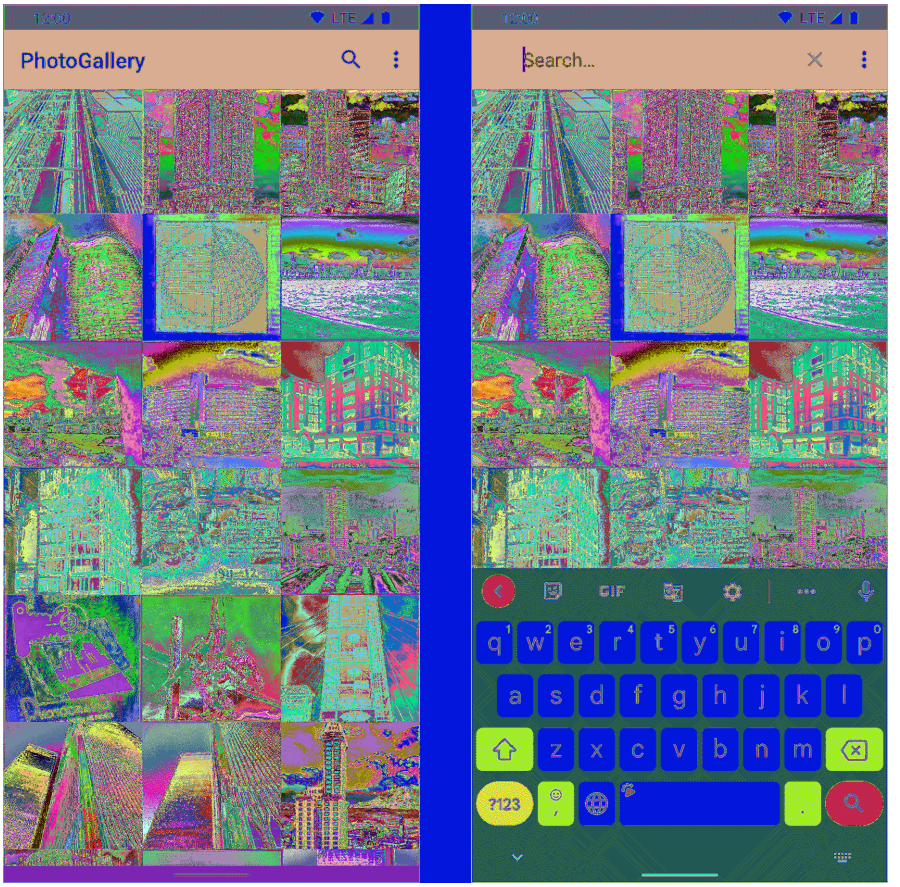
```
class PhotoGalleryFragment : Fragment() {
    ...
    private val photoGalleryViewModel: PhotoGalleryViewModel by viewModels()

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setHasOptionsMenu(true)
    }
    ...
    override fun onDestroyView() {
        ...
    }

    override fun onCreateOptionsMenu(menu: Menu, inflater: MenuInflater) {
        super.onCreateOptionsMenu(menu, inflater)
        inflater.inflate(R.menu.fragment_photo_gallery, menu)
    }
}
```

Fire up PhotoGallery and see what the **`SearchView`** looks like. Pressing the search icon expands the view to display a text box where you can enter a query (Figure 21.2).

Figure 21.2 **`SearchView`** collapsed and expanded




When the **`SearchView`** is expanded, an x icon appears on the right. Pressing the x one time clears out what you typed. Pressing the x again collapses the **`SearchView`** back to a single search icon.

If you try submitting a query, it will not do anything yet. Not to worry. You will make your **`SearchView`** more useful in just a moment.

## Responding to SearchView user interactions

When the user submits a query, your app should execute a search against the Flickr web service and refresh the images the user sees with the search results. First, update **`PhotoGalleryViewModel`** to fire off a network request and update the search results when the query changes. It does not make sense to search for an empty string, so fall back to fetching the most interesting photos when the query is empty. Since the code to make a network request is now appearing in two locations, extract it into its own private function.

### Listing 21.9 Searching in **`PhotoGalleryViewModel`** (`PhotoGalleryViewModel.kt`)

```
class PhotoGalleryViewModel : ViewModel() {
    ...
    init {
        viewModelScope.launch {
            try {
                val items = photoRepository.searchPhotos("planets")
                val items = fetchGalleryItems("planets")

                _galleryItems.value = items
            } catch (ex: Exception) {
                Log.e(TAG, "Failed to fetch gallery items", ex)
            }
        }
    }

    fun setQuery(query: String) {
        viewModelScope.launch { _galleryItems.value = fetchGalleryItems(query) }
    }

    private suspend fun fetchGalleryItems(query: String): List<GalleryItem> {
        return if (query.isNotEmpty()) {
            photoRepository.searchPhotos(query)
        } else {
            photoRepository.fetchPhotos()
        }
    }
}
```

Next, update **PhotoGalleryFragment** to use **PhotoGalleryViewModel.setQuery()** whenever the user submits a new query through the **SearchView**. Fortunately, the **SearchView.OnQueryTextListener** interface provides a way to receive a callback when a query is submitted.

Update **onCreateOptionsMenu(…)** to add a **SearchView.OnQueryTextListener** to your **SearchView**.

### Listing 21.10 Logging **SearchView.OnQueryTextListener** events (PhotoGalleryFragment.kt)

```
class PhotoGalleryFragment : Fragment() {
    ...
    override fun onCreateOptionsMenu(menu: Menu, inflater: MenuInflater) {
        super.onCreateOptionsMenu(menu, inflater)
        inflater.inflate(R.menu.fragment_photo_gallery, menu)

        val searchItem: MenuItem = menu.findItem(R.id.menu_item_search)
        val searchView = searchItem.actionView as? SearchView

        searchView?.setOnQueryTextListener(object : SearchView.OnQueryTextListener {
            override fun onQueryTextSubmit(query: String?): Boolean {
                Log.d(TAG, "QueryTextSubmit: $query")
                photoGalleryViewModel.setQuery(query ?: "")
                return true
            }

            override fun onQueryTextChange(newText: String?): Boolean {
                Log.d(TAG, "QueryTextChange: $newText")
                return false
            }
        })
    }
}
```

When importing **SearchView**, select the androidx.appcompat.widget.SearchView option from the choices presented.

In **onCreateOptionsMenu(…)**, you pull the **MenuItem** representing the search box from the menu and store it in searchItem. Then you pull the **SearchView** object from searchItem using the actionView property.

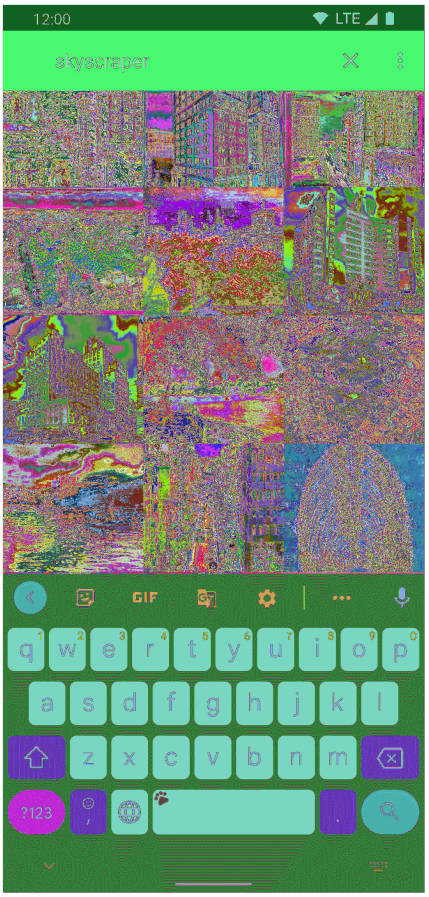
Once you have a reference to the **SearchView**, you are able to set a **SearchView.OnQueryTextListener** using **setOnQueryTextListener(…)**. You must override two functions in the **SearchView.OnQueryTextListener** implementation: **onQueryTextSubmit(String)** and **onQueryTextChange(String)**.

The **onQueryTextChange(String)** callback is executed any time text in the **SearchView** text box changes. This means that it is called every time a single character changes. You will not do anything inside this callback for this app except log the input string and return false. Returning false indicates to the system that your callback override did not handle the text change. This cues the system to perform **SearchView**'s default action (which is to show relevant suggestions, if available).

The **onQueryTextSubmit(String)** callback is executed when the user submits a query. The query the user submitted is passed as input. Returning true signifies to the system that the search request has been handled. This callback is where you call into **PhotoGalleryViewModel** to trigger the photo download for your search query.

Run your app and submit a query. You should see log statements reflecting the execution of your **`SearchView.OnQueryTextListener`** callback functions. You should also see the images displayed change based on the search term you enter (Figure 21.3).

Figure 21.3 Working **`SearchView`**




Note that if you use the hardware keyboard to submit your search query on an emulator (versus the emulator's onscreen keyboard), you may see the search executed two times, one after the other. This is because there is a small bug in **`SearchView`**. You can ignore this behavior; it is a side effect of using the emulator and will not affect your app when it runs on a real Android device.

## Simple Persistence with DataStore

In your app, there will only be one active query at a time. **`PhotoGalleryViewModel`** persists the query for the user's perceived life of the fragment. But the query should also be persisted between restarts of the app (even if the user turns off the device).

You will achieve this using the DataStore AndroidX library. DataStore is a library that helps you interact with *shared preferences*, files stored on the filesystem that your app can use to store key-value pairs. Any time the user submits a query, you will write the search term to shared preferences, overwriting whatever query was there before. When the application first launches, you will pull the stored query from shared preferences and use it to execute a Flickr search.

Shared preferences supports the same basic types that you have seen elsewhere throughout the book: **`String`**, **`Int`**, **`Boolean`**, etc. It is built into the Android OS and has been available since the first release. Prior to DataStore, it was common to access shared preferences directly, but DataStore provides stronger guarantees around data consistency and improved APIs to access and store your data asynchronously and on background threads. Naturally, it supports coroutines to perform this asynchronous work.

In addition to interacting with shared preferences, DataStore also supports interacting with *protocol buffers* for storing complex objects. Protocol buffers are fast and space efficient, but require some additional setup and add some complexity to the code, so you will stick with shared preferences here.

Open `app/build.gradle` and add the DataStore dependency:

Listing 21.11 Adding your dependency (`app/build.gradle`)

```
...
dependencies {
    ...
    implementation 'io.coil-kt:coil:2.0.0-rc02'
    implementation 'androidx.datastore:datastore-preferences:1.0.0'
    ...
}
```

Do not forget to sync your Gradle file after you have made these changes.

In CriminalIntent, you used **`CrimeRepository`** to wrap your usage of the Room library. In PhotoGallery, you are using **`PhotoRepository`** to wrap your usage of Retrofit. Similarly, you will create a **`PreferencesRepository`** to wrap your usage of the DataStore library.

The DataStore library is built with Kotlin in mind, so it uses some more advanced features of Kotlin to perform some neat tricks. Some of the setup you will do in **`PreferencesRepository`** may seem strange compared to what you have seen so far. But do not worry, once you have completed the setup, using **`PreferencesRepository`** will be straightforward. Similar to what you have seen with other libraries, you will interact with the DataStore library through a class property – this time an instance of the **`DataStore<Preferences>`** interface.

You will use a key any time you read or write the query value. DataStore's method of defining keys is a little unique. Rather than simple string keys, with DataStore you create keys using a function in the library based on the type of value being stored.

The function name is prefixed with the type, so if you want to create a key to store a string, you call the **`stringPreferencesKey()`** function. If you want to store an integer, you call the **`intPreferencesKey()`** function.

These functions still require a unique string as a parameter. You do not want to create multiple instances of these keys, so you will keep a reference in the companion object for **`PreferencesRepository`**.

Once you have the key defined, you can use it to access the stored query. DataStore exposes its data through a coroutine **`Flow`**. You want to expose the stored query as a **`Flow<String>`**, so that callers can easily access the latest stored query. You will map over the `data` property on your **`DataStore<Preferences>`** property, extracting the value for the key. To prevent multiple emissions of the same value on the **`Flow`**, use the **`distinctUntilChanged()`** function.

Make it all happen in a new `PreferencesRepository.kt`.

### Listing 21.12 Accessing a stored query (`PreferencesRepository.kt`)

```
class PreferencesRepository(
    private val dataStore: DataStore<Preferences>
) {
    val storedQuery: Flow<String> = dataStore.data.map {
        it[SEARCH_QUERY_KEY] ?: ""
    }.distinctUntilChanged()

    companion object {
        private val SEARCH_QUERY_KEY = stringPreferencesKey("search_query")
    }
}
```

The DataStore library also uses coroutines to write values to the filesystem. By using the **`edit()`** function on the `data` property, you have access to a lambda expression to make changes to the data. All the changes in the lambda expression will be treated as a single transaction, so you can edit many values in one write operation to disk. Using your key for the stored query, you can update it as the user submits new queries.

### Listing 21.13 Setting a stored query (`PreferencesRepository.kt`)

```
class PreferencesRepository(
    private val dataStore: DataStore<Preferences>
) {
    val storedQuery: Flow<String> = dataStore.data.map {
        it[SEARCH_QUERY_KEY] ?: ""
    }.distinctUntilChanged()

    suspend fun setStoredQuery(query: String) {
        dataStore.edit {
            it[SEARCH_QUERY_KEY] = query
        }
    }

    companion object {
        private val SEARCH_QUERY_KEY = stringPreferencesKey("search_query")
    }
}
```

Your app only ever needs one instance of **`PreferencesRepository`**, which can be shared across all other components. As you did with **`CrimeRepository`**, create a singleton instance of **`PreferencesRepository`** within the `companion object`, passing in an instance of the **`DataStore<Preferences>`** class.

Let the **`PreferenceDataStoreFactory`** class create the **`DataStore<Preferences>`** instance for you. The only piece you need to provide is the **`File`** where your data will be saved. Do this by calling the **`preferencesDataStoreFile()`** extension function on your **`Context`**, passing in the name of that **`File`**.

Also, since you do not want other classes creating instances of your **`PreferencesRepository`**, mark its constructor as private.

### Listing 21.14 Creating a singleton (`PreferencesRepository.kt`)

```
class PreferencesRepository private constructor(
    private val dataStore: DataStore<Preferences>
) {
    ...
    companion object {
        private val SEARCH_QUERY_KEY = stringPreferencesKey("search_query")
        private var INSTANCE: PreferencesRepository? = null

        fun initialize(context: Context) {
            if (INSTANCE == null) {
                val dataStore = PreferenceDataStoreFactory.create {
                    context.preferencesDataStoreFile("settings")
                }

                INSTANCE = PreferencesRepository(dataStore)
            }
        }

        fun get(): PreferencesRepository {
            return INSTANCE ?: throw IllegalStateException(
                "PreferencesRepository must be initialized"
            )
        }
    }
}
```

With the **`PreferencesRepository`** class set up, your next steps will repeat some of what you did with **`CrimeRepository`** in CriminalIntent: subclassing the **`Application`** class and referencing the new subclass in the manifest.

Create a new class called **PhotoGalleryApplication** and have it extend the **Application** class. Initialize the **PreferencesRepository** in the **onCreate()** method.

### Listing 21.15 Creating **PhotoGalleryApplication** (PhotoGalleryApplication.kt)

```
class PhotoGalleryApplication : Application() {
    override fun onCreate() {
        super.onCreate()
        PreferencesRepository.initialize(this)
    }
}
```

Now, register **PhotoGalleryApplication** in AndroidManifest.xml.

### Listing 21.16 Registering **PhotoGalleryApplication** (AndroidManifest.xml)

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.bignerdranch.android.photogallery">

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

    <application
        android:name=".PhotoGalleryApplication"
        android:allowBackup="true"
        ...>
        ...
    </application>

</manifest>
```

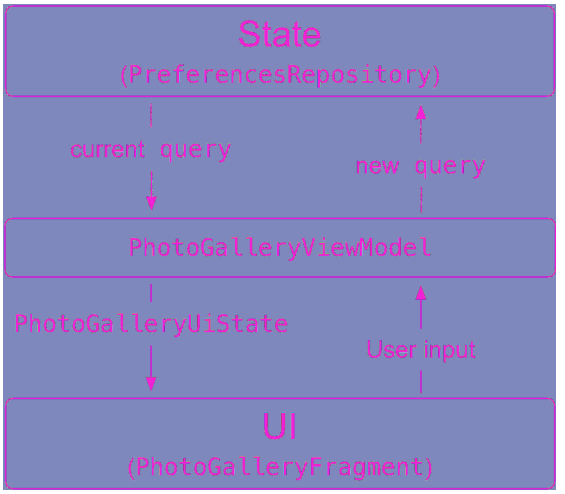
**PreferencesRepository** is your entire persistence engine for PhotoGallery. Now that you have a way to easily store and access the user's most recent query, update **PhotoGalleryViewModel** to read and write the query from disk as necessary.

As in CriminalIntent, you will use the unidirectional data flow pattern to simplify the business logic. Here in PhotoGallery, your source of state will be the data stored on the filesystem.

In CriminalIntent, the source of the state was slightly different. Yes, CriminalIntent did load a crime from the database into **CrimeDetailViewModel**. But once that crime was loaded in, **CrimeDetailViewModel** was free to mutate it as the user interacted with the UI. **CrimeDetailViewModel** was the source of state.

**PhotoGalleryViewModel** will not alter the search query. When taking in user input, it will simply pass the query along to the **PreferencesRepository**. When sending the stream of data down to **PhotoGalleryFragment**, **PhotoGalleryViewModel** will perform a network request using the latest search query provided by **PreferencesRepository** and provide the gallery items through that stream. The stream of data will still be flowing in one direction – you will just change the shape of that data (Figure 21.4).

Figure 21.4 The flow of data in PhotoGallery



Update **PhotoGalleryViewModel** to use the `storedQuery`.

### Listing 21.17 Persisting the query (`PhotoGalleryViewModel.kt`)

```
class PhotoGalleryViewModel : ViewModel() {
    private val photoRepository = PhotoRepository()
    private val preferencesRepository = PreferencesRepository.get()
    ...
    init {
        viewModelScope.launch {
            preferencesRepository.storedQuery.collectLatest { storedQuery ->
                try {
                    val items = fetchGalleryItems(~~"planets"~~ storedQuery)

                    _galleryItems.value = items
                } catch (ex: Exception) {
                    Log.e(TAG, "Failed to fetch gallery items", ex)
                }
            }
        }
    }

    fun setQuery(query: String) {
        ~~viewModelScope.launch { _galleryItems.value = fetchGalleryItems(query) }~~
        viewModelScope.launch { preferencesRepository.setStoredQuery(query) }
    }
    ...
}
```

Since the user can submit many queries in the time that it takes to perform a single network request, you will use **`collectLatest()`** instead of **`collect()`**. If your lambda expression is in the middle of processing the last emission from a **`Flow`** and a new emission arrives, the current work will be canceled and your lambda expression will restart, executing on the new emission.

This suits your use case well. You do not want to continue processing a network request for an outdated search query if the user has submitted a new one.

Next, clear the stored query (set it to "") when the user selects the Clear Search item from the overflow menu.

### Listing 21.18 Clearing a stored query (`PhotoGalleryFragment.kt`)

```
class PhotoGalleryFragment : Fragment() {
    ...
    override fun onCreateOptionsMenu(menu: Menu, inflater: MenuInflater) {
        ...
    }

    override fun onOptionsItemSelected(item: MenuItem): Boolean {
        return when (item.itemId) {
            R.id.menu_item_clear -> {
                photoGalleryViewModel.setQuery("")
                true
            }
            else -> super.onOptionsItemSelected(item)
        }
    }
}
```

Search should now work like a charm. Run PhotoGallery and try searching for something fun like "unicycle." See what results you get. Then fully exit the app using the Back button. Heck, even reboot your phone. When you relaunch your app, you should see the results for the same search term.

## Defining UI State

For a little bit of polish, pre-populate the search text box with the saved query when the user presses the search icon to expand the search view.

Unfortunately, there is not a great way for you to access your stored query at the same time that you initialize your search view. Since DataStore exposes its data asynchronously, there is no good way to access its value during the creation of a fragment. In Chapter 12, you read from your on-disk databases asynchronously to avoid blocking the main thread. Because DataStore is backed by on-disk storage, these same performance concerns apply – and mean that you need coroutines once more.

You have already seen how to send asynchronous values to your UI by leveraging **`StateFlow`** and **`ViewModel`**. If you wanted to, you could create a **`StateFlow<String>`** property in **`PhotoGalleryViewModel`** to track the search query. This would work, but you would then have two flows to collect from and juggle in your **`PhotoGalleryFragment`**. This might be OK if you only had those two flows, but as your application grows, having more and more flows becomes difficult to maintain.

Instead, you can combine the list of photos and the search query into a single value that gets sent to **`PhotoGalleryFragment`**. You can do this by defining a new data class to track your UI state. UI state objects contain all the data required to show a section or the entirety of a screen in your app.

The two pieces of data that describe what is being displayed in **`PhotoGalleryFragment`** are the list of gallery items and the value in the search text box. Create a new `data class` named **`PhotoGalleryUiState`** at the bottom of `PhotoGalleryViewModel.kt`. It will hold those two pieces of data.

Listing 21.19 Creating **`PhotoGalleryUiState`** (`PhotoGalleryViewModel.kt`)

```
class PhotoGalleryViewModel : ViewModel() {
    ...
}

data class PhotoGalleryUiState(
    val images: List<GalleryItem> = listOf(),
    val query: String = "",
)
```

Next, update the **PhotoGalleryViewModel** to expose a **StateFlow<PhotoGalleryUiState>** instead of a **StateFlow<List<GalleryItem>>**.

Listing 21.20 Exposing the search term from **PhotoGalleryViewModel** (PhotoGalleryViewModel.kt)

```
class PhotoGalleryViewModel : ViewModel() {
    ...
    private val _galleryItems: MutableStateFlow<List<GalleryItem>> =
        MutableStateFlow(listOf())
    val galleryItems: StateFlow<List<GalleryItem>>
        get() = _galleryItems.asStateFlow()
    private val _uiState: MutableStateFlow<PhotoGalleryUiState> =
        MutableStateFlow(PhotoGalleryUiState())
    val uiState: StateFlow<PhotoGalleryUiState>
        get() = _uiState.asStateFlow()

    init {
        viewModelScope.launch {
            preferencesRepository.storedQuery.collectLatest { storedQuery ->
                try {
                    val items = fetchGalleryItems(storedQuery)

                    _galleryItems.value = items
                    _uiState.update { oldState ->
                        oldState.copy(
                            images = items,
                            query = storedQuery
                        )
                    }
                } catch (ex: Exception) {
                    Log.e(TAG, "Failed to fetch gallery items", ex)
                }
            }
        }
    }
    ...
}
...
```

To update the **SearchView** with the saved query, you will have to maintain a reference to it. Add a class property to **PhotoGalleryFragment** for that reference. Similar to your _binding property, you do not want to hold a reference to this new property longer than needed. So – just as you de-referenced the _binding property in the **onDestroyView()** function – you will de-reference this new property in **onDestroyOptionsMenu()**.

### Listing 21.21 Holding a reference to your **SearchView** (PhotoGalleryFragment.kt)

```
class PhotoGalleryFragment : Fragment() {
    ...
    private var searchView: SearchView? = null

    private val photoGalleryViewModel: PhotoGalleryViewModel by viewModels()
    ...
    override fun onCreateOptionsMenu(menu: Menu, inflater: MenuInflater) {
        ...
        val searchItem: MenuItem = menu.findItem(R.id.menu_item_search)
        val searchView = searchItem.actionView as? SearchView
        pollingMenuItem = menu.findItem(R.id.menu_item_toggle_polling)
        ...
    }

    override fun onOptionsItemSelected(item: MenuItem): Boolean {
        ...
    }

    override fun onDestroyOptionsMenu() {
        super.onDestroyOptionsMenu()
        searchView = null
    }
}
```

Now, update **`PhotoGalleryFragment`** to use the new **`StateFlow<PhotoGalleryUiState>`**. Set your **`RecyclerView.Adapter`** like before and call the **`setQuery()`** function on the search view to populate it with your latest query.

Listing 21.22 Pre-populating **`SearchView`** (`PhotoGalleryFragment.kt`)

```
class PhotoGalleryFragment : Fragment() {
    ...
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        viewLifecycleOwner.lifecycleScope.launch {
            viewLifecycleOwner.repeatOnLifecycle(Lifecycle.State.STARTED) {
                photoGalleryViewModel.galleryItems.collect { items ->
                    binding.photoGrid.adapter = PhotoListAdapter(items)
                }
                photoGalleryViewModel.uiState.collect { state ->
                    binding.photoGrid.adapter = PhotoListAdapter(state.images)
                    searchView?.setQuery(state.query, false)
                }
            }
        }
    }
    ...
}
```

Run your app and play around with submitting a few searches. Revel at the polish your last bit of code added. Of course, there is always more polish you could add…

## Challenge: Polishing Your App Some More

With your search feature now working, you might notice little defects or shortcomings of your implementation. Modern, high-quality apps do many subtle things to improve the user experience when searching for content. See if you can implement a few of these changes:

- As soon as a query is submitted, hide the soft keyboard.
- While the network request is executing, display a loading indicator (indeterminate progress bar).
- Your current search implementation has a slight problem: If you start typing a new query while a search is ongoing, it will be reset when the search finishes. Let the user start typing in a new query (or disable text input on the **`SearchView`** entirely) while a query is executing.
- Many apps show search suggestions or previous searches to help users enter queries faster. Keep track of queries that have been submitted previously, and show them onscreen when the user is typing into the **`SearchView`**. (You will need a second **`RecyclerView`**) to show search suggestions which will appear in place of the gallery when the search field is active.)

Some of these tasks can be accomplished on their own. Others might require you to make changes to the state of your UI and how you represent that data.

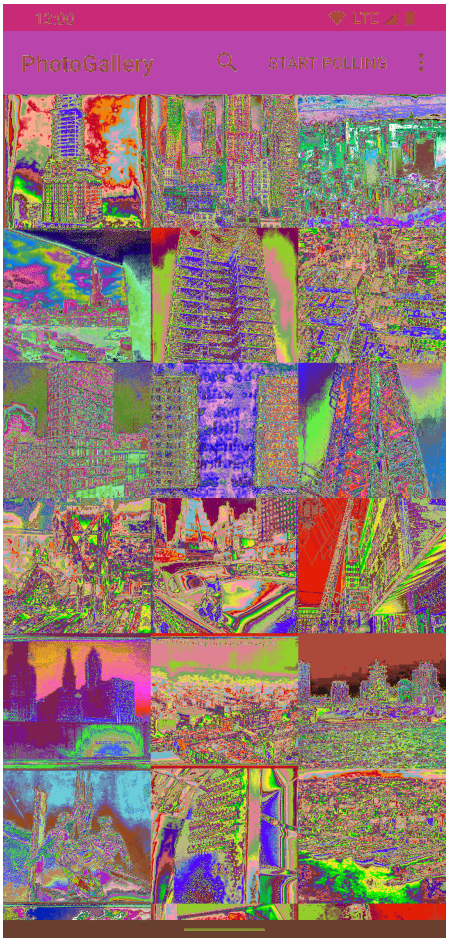
# 22
# WorkManager

PhotoGallery can now download interesting images from Flickr, find images based on a user's search query, and remember the query when the user leaves the app. In this chapter, you will add functionality to poll Flickr and determine whether there are new photos the user has not seen yet.

This work will happen in the background, meaning it will execute even if the user is not actively using your app. If there are new photos, the app will display a notification prompting the user to return to the app and see the new content.

Tools from the Jetpack WorkManager architecture component library will handle the periodic work of checking Flickr for new photos. You will create a **`Worker`** class to perform the actual work, then schedule it to execute on an interval. When new photos are found, you will post a **`Notification`** to the user with the **`NotificationManager`** (Figure 22.1).

Figure 22.1 The end result



## Creating a Worker

The **`Worker`** class is where you will put the logic you want to perform in the background. Once your worker is in place, you will create a **`WorkRequest`** that tells the system when you would like your work to execute.

Before you can add your worker, you first need to add the appropriate dependency in `app/build.gradle`.

### Listing 22.1 Adding the WorkManager dependency (`app/build.gradle`)

```
dependencies {
    ...
    implementation "androidx.datastore:datastore-preferences:1.0.0"
    implementation 'androidx.work:work-runtime-ktx:2.7.1'
    ...
}
```

Do not forget to sync your files after you add the dependency.

With your new library in place, set up your **`Worker`**. Like several of the libraries you have used so far, the WorkManager library integrates with coroutines. Create a new class called **`PollWorker`** that extends the **`CoroutineWorker`** base class. Your **`PollWorker`** will need two parameters, a **`Context`** and a **`WorkerParameters`** object. Both of these will be passed to the superclass constructor. For now, override the **`doWork()`** function and log a message to the console.

### Listing 22.2 Creating the worker (`PollWorker.kt`)

```
private const val TAG = "PollWorker"

class PollWorker(
    private val context: Context,
    workerParameters: WorkerParameters
) : CoroutineWorker(context, workerParameters) {
    override suspend fun doWork(): Result {
        Log.i(TAG, "Work request triggered")
        return Result.success()
    }
}
```

The **`doWork()`** function is called from a background thread, so you can do any long-running tasks you need there. The return values for the function indicate the status of your operation. In this case, you return success, since the function just prints a log to the console.

**`doWork()`** can return a failure result if the work cannot be completed. In that case, the work request would not run again. It can also return a retry result if a temporary error was encountered and you want the work to run again in the future.

The **`PollWorker`** only knows how to *execute* the background work. You need another component to *schedule* the work.

## Scheduling Work

To schedule a **`Worker`** to execute, you need a **`WorkRequest`**. The **`WorkRequest`** class itself is abstract, so you need to use one of its subclasses depending on the type of work you need to execute. If you have something that only needs to execute once, use a **`OneTimeWorkRequest`**. If your work is something that must execute periodically, use a **`PeriodicWorkRequest`**.

For now, you are going to use the **`OneTimeWorkRequest`**. This will let you learn more about creating and controlling the requests and verify that your **`PollWorker`** is functioning correctly. Later you will update your app to use a **`PeriodicWorkRequest`**.

Open `PhotoGalleryFragment.kt`, create a work request, and schedule it for execution.

### Listing 22.3 Scheduling a **`WorkRequest`** (`PhotoGalleryFragment.kt`)

```
class PhotoGalleryFragment : Fragment() {
    ...
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setHasOptionsMenu(true)

        val workRequest = OneTimeWorkRequest
            .Builder(PollWorker::class.java)
            .build()
        WorkManager.getInstance(requireContext())
            .enqueue(workRequest)
    }
    ...
}
```

The **`OneTimeWorkRequest`** class uses a builder to construct an instance. You provide the **`Worker`** class to the builder that the work request will fire. Once your work request is ready, you schedule it with the **`WorkManager`** class. You call the **`getInstance(Context)`** function to access the **`WorkManager`**, then call **`enqueue(…)`** with the work request as a parameter. This will schedule your work request to execute based on the request type and any constraints you add to the request.

Run your app and search for PollWorker in Logcat. You should see your log statement soon after your app starts up:

```
    19:58:39.415 I/PollWorker: Work request triggered
    19:58:39.420 I/WM-WorkerWrapper: Worker result SUCCESS for Work [ id=896...
```

In many cases, the work you want to execute in the background is tied to the network. Maybe you are polling for new information the user has not seen yet, or you are pushing updates from the local database to save them on a remote server. While this work is important, you should make sure you are not needlessly using costly data. The best time for these requests is when the device is connected to an unmetered network.

You can use the **`Constraints`** class to add this information to your work requests. With this class, you can require that certain conditions be met before your work can execute. Requiring a certain network type is one case. You can also require conditions like sufficient battery charge or that the device is charging.

Edit your **`OneTimeWorkRequest`** in **`PhotoGalleryFragment`** to add constraints to the request.

### Listing 22.4 Adding work constraints (`PhotoGalleryFragment.kt`)

```
class PhotoGalleryFragment : Fragment() {
    ...
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setHasOptionsMenu(true)

        val constraints = Constraints.Builder()
            .setRequiredNetworkType(NetworkType.UNMETERED)
            .build()
        val workRequest = OneTimeWorkRequest
            .Builder(PollWorker::class.java)
            .setConstraints(constraints)
            .build()
        WorkManager.getInstance(requireContext())
            .enqueue(workRequest)
    }
    ...
}
```

Be sure to choose the import for **`androidx.work.Constraints`**.

Similar to the work request, the **`Constraints`** object uses a builder to configure a new instance. In this case, you specify that the device must be on an unmetered network for the work request to execute.

To test this functionality, you will need to simulate different network types on your emulator. By default, an emulator connects to a simulated WiFi network. Since WiFi is an unmetered network, if you run your app now, with the constraints in place, you should see the log message from your **`PollWorker`**.

To verify that the work request does not execute when the device is on a metered network, you will need to modify the network settings for your emulator. Quit PhotoGallery and pull down on the notification shade to expose the device's Quick Settings. Click the Internet icon. Within the internet quick settings, toggle off the WiFi option (Figure 22.2). This will force the emulator to use its (simulated) cellular network, which is metered.

Figure 22.2 Turning off WiFi




With the WiFi disabled, rerun PhotoGallery from Android Studio and verify that the log from **`PollWorker`** does not appear. Before moving on, return to the Quick Settings and re-enable the WiFi network.

## Checking for New Photos

Now that your worker is executing, you can add the logic to check for new photos. There are a couple pieces needed for this functionality. You will first need a way to save the ID of the most recent photo the user has seen, then you will need to update your worker class to pull the new photos and compare the stored ID with the newest one from the server.

The first change you will make is to update **`PreferencesRepository`** to store and retrieve the latest photo ID from shared preferences.

Listing 22.5 Saving the latest photo ID (`PreferencesRepository.kt`)

```
class PreferencesRepository private constructor(
    private val dataStore: DataStore<Preferences>
) {
    ...
    suspend fun setStoredQuery(query: String) {
        dataStore.edit {
            it[SEARCH_QUERY_KEY] = query
        }
    }

    val lastResultId: Flow<String> = dataStore.data.map {
        it[PREF_LAST_RESULT_ID] ?: ""
    }.distinctUntilChanged()

    suspend fun setLastResultId(lastResultId: String) {
        dataStore.edit {
            it[PREF_LAST_RESULT_ID] = lastResultId
        }
    }

    companion object {
        private val SEARCH_QUERY_KEY = stringPreferencesKey("search_query")
        private val PREF_LAST_RESULT_ID = stringPreferencesKey("lastResultId")
        private var INSTANCE: PreferencesRepository? = null
        ...
    }
}
```

With your preferences set up, you can start the work in **`PollWorker`**. You will need access to both **`PreferencesRepository`** and **`PhotoRepository`** to perform your work. You can get a single value out of each of the **`Flow`** properties on **`PreferencesRepository`** by calling the **`first()`** function on them. If the user has not searched for anything yet, you do not have a search term to look for new content. In that case, you can finish your work early.

### Listing 22.6 Starting your work (`PollWorker.kt`)

```
class PollWorker(
    private val context: Context,
    workerParameters: WorkerParameters
) : CoroutineWorker(context, workerParameters) {
    override suspend fun doWork(): Result {
        Log.i(TAG, "Work request triggered")
        val preferencesRepository = PreferencesRepository.get()
        val photoRepository = PhotoRepository()

        val query = preferencesRepository.storedQuery.first()
        val lastId = preferencesRepository.lastResultId.first()

        if (query.isEmpty()) {
            Log.i(TAG, "No saved query, finishing early.")
            return Result.success()
        }

        return Result.success()
    }
}
```

When your user does have a stored query, you want to try to make a request to get the gallery items for that query. If the network request fails for any reason, have the **`PollWorker`** return **`Result.failure()`**. It is OK to fail sometimes. There are many reasons the network request could fail, and in most situations, there is nothing you can do to fix it.

If the network request does succeed, then you want to check whether the most recent photo ID matches the one you have saved. If they do not match, then you will show the user a notification. Whether or not the photo IDs match, you will return **`Result.success()`**.

### Listing 22.7 Getting the work done (`PollWorker.kt`)

```
class PollWorker(
    private val context: Context,
    workerParameters: WorkerParameters
) : CoroutineWorker(context, workerParameters) {
    override suspend fun doWork(): Result {
        val preferencesRepository = PreferencesRepository.get()
        val photoRepository = PhotoRepository()

        val query = preferencesRepository.storedQuery.first()
        val lastId = preferencesRepository.lastResultId.first()

        if (query.isEmpty()) {
            Log.i(TAG, "No saved query, finishing early.")
            return Result.success()
        }

        return Result.success()
        return try {
            val items = photoRepository.searchPhotos(query)

            if (items.isNotEmpty()) {
                val newResultId = items.first().id
                if (newResultId == lastId) {
                    Log.i(TAG, "Still have the same result: $newResultId")
                } else {
                    Log.i(TAG, "Got a new result: $newResultId")
                    preferencesRepository.setLastResultId(newResultId)
                }
            }

            Result.success()
        } catch (ex: Exception) {
            Log.e(TAG, "Background update failed", ex)
            Result.failure()
        }
    }
}
```

Run your app on a device or emulator. The first time you run it, there will not be a last result ID saved in **`QueryPreferences`**, so you should see the log statement indicating that **`PollWorker`** found a new result. If you quickly run the app again, you should see that your worker finds the same ID.

```
20:08:05.930 I/PollWorker: Got a new result: 51873395252
20:08:05.987 I/WM-WorkerWrapper: Worker result SUCCESS for Work [ id=988...
20:08:35.189 I/PollWorker: Still have the same result: 51873395252
20:08:35.192 I/WM-WorkerWrapper: Worker result SUCCESS for Work [ id=88b...
```

## Notifying the User

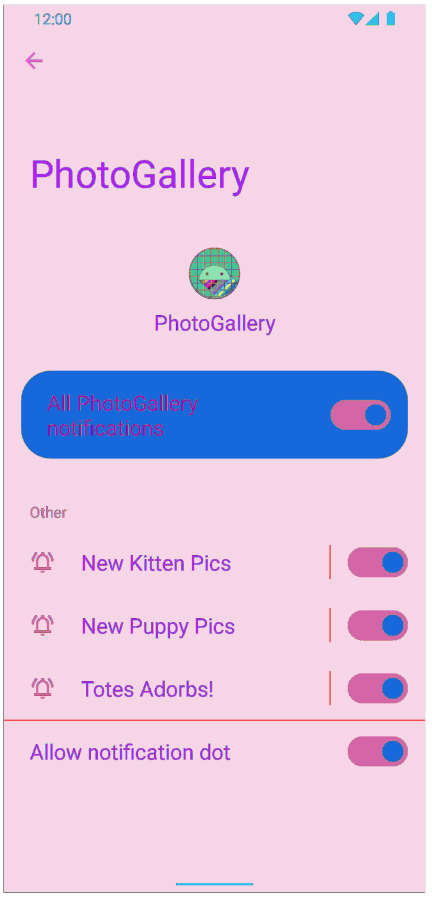
Your worker is now running and checking for new photos in the background, but the user does not know anything about it. When PhotoGallery finds new photos the user has not seen yet, it should prompt the user to open the app and see the new content.

When your app needs to communicate something to the user, the proper tool is almost always a *notification*. Notifications are items that appear in the notifications drawer, which the user can access by dragging down from the top of the screen.

Before you can create notifications on Android devices running Android Oreo (API level 26) and higher, you must create a **`Channel`**. A **`Channel`** categorizes notifications and gives the user fine-grained control over notification preferences. Rather than only having the option to turn off notifications for your entire app, the user can choose to turn off certain categories of notifications within your app. The user can also customize muting, vibration, and other notification settings channel by channel.

For example, suppose you wanted PhotoGallery to send three categories of notifications when new cute animal pictures were fetched: New Kitten Pics, New Puppy Pics, and Totes Adorbs! (for all adorable animal pictures, regardless of species). You would create three channels, one for each of the notification categories, and the user could configure them independently (Figure 22.3).

Figure 22.3 Fine-grained notification configuration for channels



Your application must create at least one channel to support Android Oreo and higher. There is no documented upper limit on the number of channels an app can create. But be reasonable – keep the number small and meaningful for the user. Remember that the goal is to allow the user to configure notifications in your app. Adding too many channels would ultimately confuse the user and make for a poor user experience.

Update **PhotoGalleryApplication** to create and add a channel if the device is running Android Oreo or higher.

### Listing 22.8 Creating a notification channel (PhotoGalleryApplication.kt)

```
const val NOTIFICATION_CHANNEL_ID = "flickr_poll"

class PhotoGalleryApplication : Application() {
    override fun onCreate() {
        super.onCreate()
        PreferencesRepository.initialize(this)

        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
            val name = getString(R.string.notification_channel_name)
            val importance = NotificationManager.IMPORTANCE_DEFAULT
            val channel =
                NotificationChannel(NOTIFICATION_CHANNEL_ID, name, importance)
            val notificationManager: NotificationManager =
                getSystemService(NotificationManager::class.java)
            notificationManager.createNotificationChannel(channel)
        }
    }
}
```

The channel name is a user-facing string, displayed in the notification settings screen for your app (shown in Figure 22.3). Add a string resource to res/values/strings.xml to store the channel name. While you are there, go ahead and add the other strings needed for your notification.

### Listing 22.9 Adding strings (res/values/strings.xml)

```
<resources>
    <string name="clear_search">Clear Search</string>
    <string name="notification_channel_name">Background updates</string>
    <string name="new_pictures_title">New PhotoGallery Pictures</string>
    <string name="new_pictures_text">You have new pictures in PhotoGallery.</string>
</resources>
```

To post a notification, you create a **`Notification`** object. **`Notification`**s are created with a builder object, much like the **`AlertDialog`** that you used in Chapter 14. At a minimum, your **`Notification`** should have:

- an *icon* to show in the status bar
- a *view* to show in the notification drawer to represent the notification itself
- a ***`PendingIntent`*** to fire when the user presses the notification in the drawer
- a ***`NotificationChannel`*** to apply styling and provide user control over the notification

You will also add ticker text to the notification. This text does not display when the notification shows, but it is sent to the accessibility services to support screen readers.

Once you have created a **`Notification`** object, you can post it by calling **`notify(Int, Notification)`** on the **`NotificationManager`** system service. The **`Int`** is the ID of the notification from your app.

First you need to add some plumbing code. Open `MainActivity.kt` and add a **`newIntent(Context)`** function. This function will return an **`Intent`** instance that can be used to start **`MainActivity`**. (Eventually, **`PollWorker`** will call **`MainActivity.newIntent(…)`**, wrap the resulting intent in a **`PendingIntent`**, and set that **`PendingIntent`** on a notification.)

### Listing 22.10 Adding **`newIntent(…)`** to **`MainActivity`** (`MainActivity.kt`)

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        ...
    }

    companion object {
        fun newIntent(context: Context): Intent {
            return Intent(context, MainActivity::class.java)
        }
    }
}
```

Now make **`PollWorker`** notify the user that a new result is ready by creating a **`Notification`** and calling **`NotificationManager.notify(Int, Notification)`**.

### Listing 22.11 Adding a notification (`PollWorker.kt`)

```
class PollWorker(
    private val context: Context,
    workerParameters: WorkerParameters
) : CoroutineWorker(context, workerParameters) {
    override suspend fun doWork(): Result {
        ...
        return try {
            val items = photoRepository.searchPhotos(query)

            if (items.isNotEmpty()) {
                val newResultId = items.first().id
                if (newResultId == lastId) {
                    Log.i(TAG, "Still have the same result: $newResultId")
                } else {
                    Log.i(TAG, "Got a new result: $newResultId")
                    preferencesRepository.setLastResultId(newResultId)
                    notifyUser()
                }
            }

            Result.success()
        } catch (ex: Exception) {
            ...
        }
    }

    private fun notifyUser() {
        val intent = MainActivity.newIntent(context)
        val pendingIntent = PendingIntent.getActivity(
            context,
            0,
            intent,
            PendingIntent.FLAG_IMMUTABLE
        )
        val resources = context.resources

        val notification = NotificationCompat
            .Builder(context, NOTIFICATION_CHANNEL_ID)
            .setTicker(resources.getString(R.string.new_pictures_title))
            .setSmallIcon(android.R.drawable.ic_menu_report_image)
            .setContentTitle(resources.getString(R.string.new_pictures_title))
            .setContentText(resources.getString(R.string.new_pictures_text))
            .setContentIntent(pendingIntent)
            .setAutoCancel(true)
            .build()

        NotificationManagerCompat.from(context).notify(0, notification)
    }
}
```

Let’s go over this from top to bottom.

You use the **NotificationCompat** class to easily support notifications on both pre-Oreo and Oreo-and-above devices. **NotificationCompat.Builder** accepts a channel ID and uses the ID to set the notification's channel if the user is running Oreo or above. If the user is running a pre-Oreo version of Android, **NotificationCompat.Builder** ignores the channel. (Note that the channel ID you pass here comes from the `NOTIFICATION_CHANNEL_ID` constant you added to **PhotoGalleryApplication**.)

In Listing 22.8, you checked the build version SDK before you created the channel, because there is no AndroidX API for creating a channel. You do not need to do that here, because **NotificationCompat** checks the build version for you, keeping your code clean and spiffy. This is one reason you should use AndroidX's version of the Android APIs whenever available.

Next you configure the ticker text and small icon by calling **setTicker(CharSequence)** and **setSmallIcon(Int)**. (The icon resource you are using is provided as part of the Android framework, denoted by the package name qualifier `android` in `android.R.drawable.ic_menu_report_image`, so you do not have to pull the icon image into your resource folder.)

After that, you configure the appearance of your **Notification** in the drawer itself. It is possible to customize your notification, but it is easier to use the standard look, which features an icon, a title, and a text area. The value from **setSmallIcon(Int)** will be used for the icon. To set the title and text, you call **setContentTitle(CharSequence)** and **setContentText(CharSequence)**.

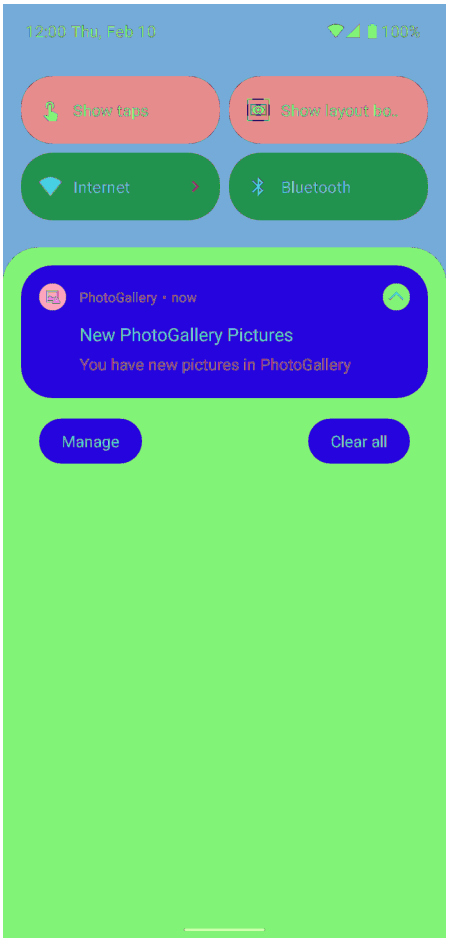
Next, you specify what happens when the user presses your **Notification**. This is done using a **PendingIntent** object. The **PendingIntent** you pass into **setContentIntent(PendingIntent)** will be fired when the user presses your **Notification** in the drawer. Calling **setAutoCancel(true)** tweaks that behavior a little bit: The notification will also be deleted from the notification drawer when the user presses it.

Finally, you get an instance of **NotificationManager** from the current context (`NotificationManagerCompat.from`) and call **NotificationManager.notify(…)** to post your notification.

The integer parameter you pass to **notify(…)** is an identifier for your notification. It should be unique across your application, but it is reusable. A notification will replace another notification with the same ID that is still in the notification drawer. If there is no existing notification with the ID, the system will show a new notification. This is how you would implement a progress bar or other dynamic visuals.

And that is it. Run your app, and you should eventually see a notification icon appear in the status bar (Figure 22.4). (You will want to clear any search terms to speed things along.)

Figure 22.4 New photos notification



## Providing User Control over Polling

Some users may not want your app to run in the background. An important control to provide users is the ability to enable and disable background polling.

For PhotoGallery, you will add a menu item to the app bar that will toggle your worker when selected. You will also update your work request to run your worker periodically instead of just once.

To toggle your worker, you first need to determine whether the worker is currently running. To do this, supplement your **`PreferencesRepository`** to store a flag indicating whether the worker is enabled.

Listing 22.12 Saving **`Worker`** state (`PreferencesRepository.kt`)

```
class PreferencesRepository private constructor(
    private val dataStore: DataStore<Preferences>
) {
    ...
    suspend fun setLastResultId(lastResultId: String) {
        dataStore.edit {
            it[PREF_LAST_RESULT_ID] = lastResultId
        }
    }

    val isPolling: Flow<Boolean> = dataStore.data.map {
        it[PREF_IS_POLLING] ?: false
    }.distinctUntilChanged()

    suspend fun setPolling(isPolling: Boolean) {
        dataStore.edit {
            it[PREF_IS_POLLING] = isPolling
        }
    }

    companion object {
        private val SEARCH_QUERY_KEY = stringPreferencesKey("search_query")
        private val PREF_LAST_RESULT_ID = stringPreferencesKey("lastResultId")
        private val PREF_IS_POLLING = booleanPreferencesKey("isPolling")
        private var INSTANCE: PreferencesRepository? = null
        ...
    }
}
```

Next, add the string resources your options menu item needs. You will need two strings, one to prompt the user to enable polling and one to prompt them to disable it.

Listing 22.13 Adding poll-toggling resources (`res/values/strings.xml`)

```
<resources>
    ...
    <string name="new_pictures_text">You have new pictures in PhotoGallery.</string>
    <string name="start_polling">Start polling</string>
    <string name="stop_polling">Stop polling</string>
</resources>
```

With your strings in place, open up your `res/menu/fragment_photo_gallery.xml` menu file and add a new item for your polling toggle.

### Listing 22.14 Adding a poll-toggling item (`res/menu/fragment_photo_gallery.xml`)

```
<?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/menu_item_clear"
          android:title="@string/clear_search"
          app:showAsAction="never" />

    <item android:id="@+id/menu_item_toggle_polling"
          android:title="@string/start_polling"
          app:showAsAction="ifRoom|withText"/>
</menu>
```

The default text for this item is the `start_polling` string. You will need to update this text if the worker is already running. Start by getting a reference to your new menu item in **`PhotoGalleryFragment`**.

### Listing 22.15 Accessing the menu item (`PhotoGalleryFragment.kt`)

```
class PhotoGalleryFragment : Fragment() {
    ...
    private var searchView: SearchView? = null
    private var pollingMenuItem: MenuItem? = null
    ...
    override fun onCreateOptionsMenu(menu: Menu, inflater: MenuInflater) {
        super.onCreateOptionsMenu(menu, inflater)
        inflater.inflate(R.menu.fragment_photo_gallery, menu)

        val searchItem: MenuItem = menu.findItem(R.id.menu_item_search)
        searchView = searchItem.actionView as? SearchView
        pollingMenuItem = menu.findItem(R.id.menu_item_toggle_polling)

        searchView?.setOnQueryTextListener(object : SearchView.OnQueryTextListener {
            ...
        })
    }
    ...
    override fun onDestroyOptionsMenu() {
        super.onDestroyOptionsMenu()
        searchView = null
        pollingMenuItem = null
    }
    ...
}
```

Next, include whether the worker is running in your **PhotoGalleryUiState** by collecting the latest value from the isPolling property on the **PreferencesRepository** class. Also, add a function to toggle the property.

### Listing 22.16 Adding more data to **PhotoGalleryUiState** (PhotoGalleryViewModel.kt)

```
class PhotoGalleryViewModel : ViewModel() {
    ...
    init {
        viewModelScope.launch {
            preferencesRepository.storedQuery.collectLatest { storedQuery ->
                ...
            }
        }

        viewModelScope.launch {
            preferencesRepository.isPolling.collect { isPolling ->
                _uiState.update { it.copy(isPolling = isPolling) }
            }
        }
    }

    fun setQuery(query: String) {
        viewModelScope.launch { preferencesRepository.setStoredQuery(query) }
    }

    fun toggleIsPolling() {
        viewModelScope.launch {
            preferencesRepository.setPolling(!uiState.value.isPolling)
        }
    }
    ...
}

data class PhotoGalleryUiState(
    val images: List<GalleryItem> = listOf(),
    val query: String = "",
    val isPolling: Boolean = false,
)
```

Open `PhotoGalleryFragment.kt` and update your menu item text whenever you receive a new **`PhotoGalleryUiState`** value. Do that work in a separate private function named **`updatePollingState()`**. You will add some more code to that function in just a second.

### Listing 22.17 Setting correct menu item text (`PhotoGalleryFragment.kt`)

```
class PhotoGalleryFragment : Fragment() {
    ...
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        viewLifecycleOwner.lifecycleScope.launch {
            viewLifecycleOwner.repeatOnLifecycle(Lifecycle.State.STARTED) {
                photoGalleryViewModel.uiState.collect { state ->
                    binding.photoGrid.adapter = PhotoListAdapter(state.images)
                    searchView?.setQuery(state.query, false)
                    updatePollingState(state.isPolling)
                }
            }
        }
    }
    ...
    override fun onDestroyOptionsMenu() {
        ...
    }

    private fun updatePollingState(isPolling: Boolean) {
        val toggleItemTitle = if (isPolling) {
            R.string.stop_polling
        } else {
            R.string.start_polling
        }
        pollingMenuItem?.setTitle(toggleItemTitle)
    }
}
```

Now, call the newly created **`toggleIsPolling()`** on your **`PhotoGalleryViewModel`** whenever your menu item is pressed.

### Listing 22.18 Handling menu item presses (`PhotoGalleryFragment.kt`)

```
class PhotoGalleryFragment : Fragment() {
    ...
    override fun onOptionsItemSelected(item: MenuItem): Boolean {
        return when (item.itemId) {
            R.id.menu_item_clear -> {
                photoGalleryViewModel.setQuery("")
                true
            }
            R.id.menu_item_toggle_polling -> {
                photoGalleryViewModel.toggleIsPolling()
                true
            }
            else -> super.onOptionsItemSelected(item)
        }
    }
    ...
}
```

Finally, delete the **`OneTimeWorkRequest`** logic from the **`onCreate(…)`** function, since it is no longer needed. Instead, add code to the new **`updatePollingState()`** to update the background work. If the worker is not running, create a new **`PeriodicWorkRequest`** and schedule it with the **`WorkManager`**. If the worker is running, stop it.

Listing 22.19 Handling poll-toggling item clicks (`PhotoGalleryFragment.kt`)

```
private const val TAG = "PhotoGalleryFragment"
private const val POLL_WORK = "POLL_WORK"

class PhotoGalleryFragment : Fragment() {
    ...
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setHasOptionsMenu(true)

        val constraints = Constraints.Builder()
            .setRequiredNetworkType(NetworkType.UNMETERED)
            .build()
        val workRequest = OneTimeWorkRequest
            .Builder(PollWorker::class.java)
            .setConstraints(constraints)
            .build()
        WorkManager.getInstance(requireContext())
            .enqueue(workRequest)
    }
    ...
    private fun updatePollingState(isPolling: Boolean) {
        val toggleItemTitle = if (isPolling) {
            R.string.stop_polling
        } else {
            R.string.start_polling
        }
        pollingMenuItem?.setTitle(toggleItemTitle)

        if (isPolling) {
            val constraints = Constraints.Builder()
                .setRequiredNetworkType(NetworkType.UNMETERED)
                .build()
            val periodicRequest =
                PeriodicWorkRequestBuilder<PollWorker>(15, TimeUnit.MINUTES)
                    .setConstraints(constraints)
                    .build()
            WorkManager.getInstance(requireContext()).enqueueUniquePeriodicWork(
                POLL_WORK,
                ExistingPeriodicWorkPolicy.KEEP,
                periodicRequest
            )
        } else {
            WorkManager.getInstance(requireContext()).cancelUniqueWork(POLL_WORK)
        }
    }
}
```

If you are given a choice when importing **`TimeUnit`**, select java.util.concurrent.TimeUnit.

Focus first on the `else` block you added here. If the worker is currently *not* running, then you schedule a new work request with the **`WorkManager`**. In this case, you are using the **`PeriodicWorkRequest`** class to make your worker reschedule itself on an interval. The work request uses a builder, like the **`OneTimeWorkRequest`** you used previously. The builder needs the worker class to run as well as the interval it should use to execute the worker.

If you are thinking that 15 minutes is a long time for an interval, you are right. However, if you tried to enter a smaller interval value, you would find that your worker still executes on a 15-minute interval. This is the minimum interval allowed for a **`PeriodicWorkRequest`** so that the system is not tied up running the same work request all the time. This saves system resources – and the user's battery life.

The **`PeriodicWorkRequest`** builder accepts constraints, just like the one-time request, so you add the unmetered network requirement. To schedule the work request, you use the **`WorkManager`** class, but this time you use the **`enqueueUniquePeriodicWork(…)`** function. This function takes in a **`String`** name, a policy, and your work request. The name allows you to uniquely identify the request, which is useful when you want to cancel it.

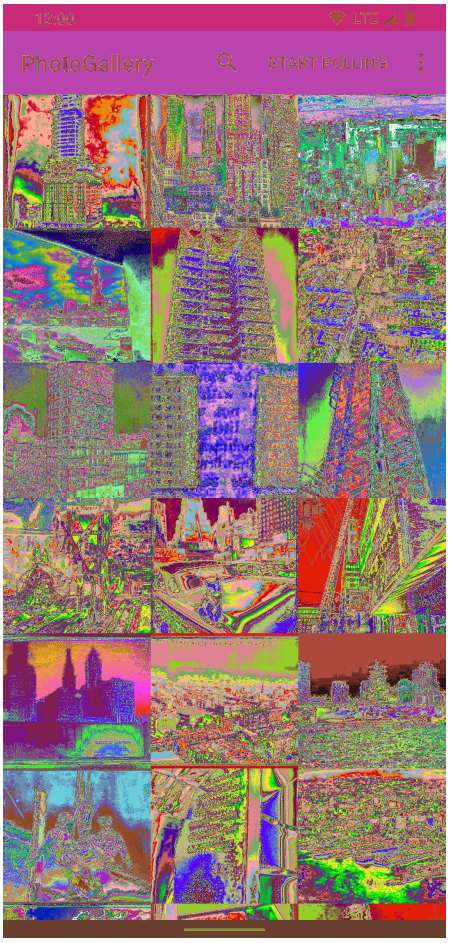
The existing work policy tells the work manager what to do if you have already scheduled a work request with a particular name. In this case you use the `KEEP` option, which discards your new request in favor of the one that already exists. The other option is `REPLACE`, which, as the name implies, will replace the existing work request with the new one.

If the worker is already running, then you need to tell the **`WorkManager`** to cancel the work request. In this case, you call the **`cancelUniqueWork(…)`** function with the `"POLL_WORK"` name to remove the periodic work request.

Run the application. You should see your new menu item to toggle polling. If you do not want to wait for the 15-minute interval, you can disable the polling, wait a few seconds, then enable polling to rerun the work request.

PhotoGallery can now keep the user up to date with the latest images automatically, even when the app is not running (Figure 22.5).

Figure 22.5 The end result



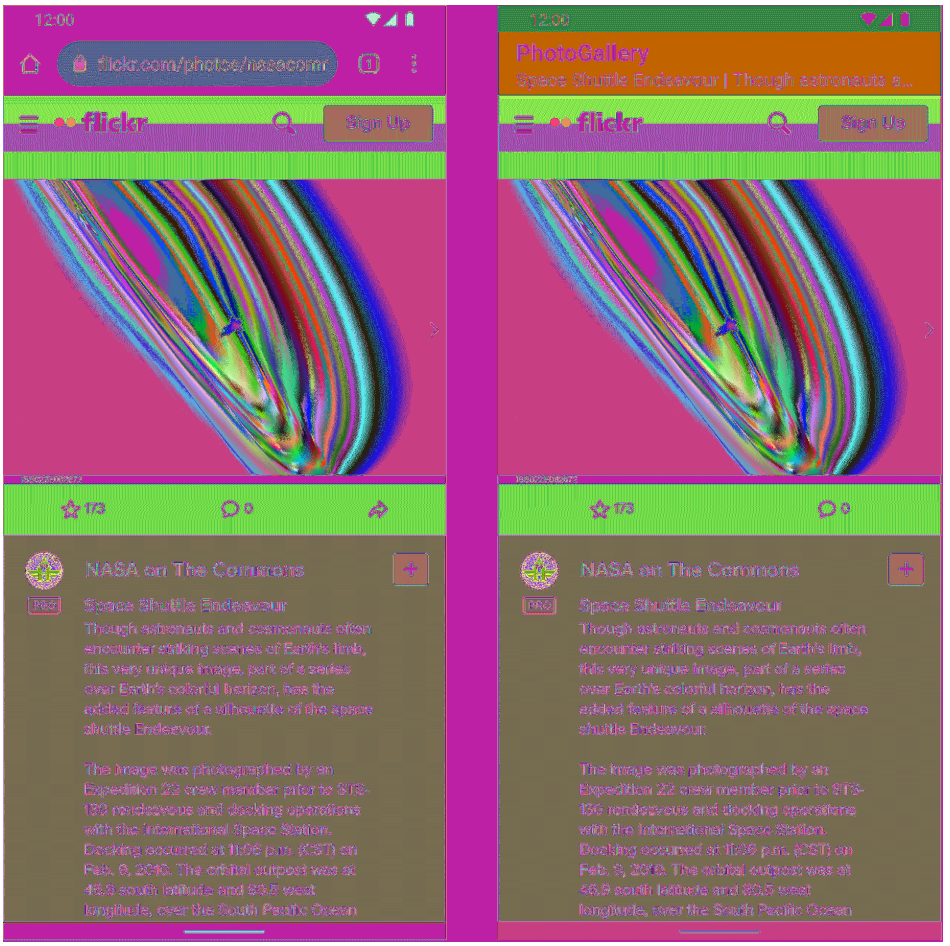

In the next chapter, you will finish your work on PhotoGallery by allowing users to open a photo's page on Flickr.

# 23

# Browsing the Web and WebView

Each photo you get from Flickr has a page associated with it. In this chapter, you will finish your work on PhotoGallery by updating it so that users can press a photo to see its Flickr page. You will learn two different ways to integrate web content into your apps, shown in Figure 23.1. The first works with the device's browser app (left), and the second uses a **`WebView`** to display web content within PhotoGallery (right).

Figure 23.1 Web content: two different approaches



## One Last Bit of Flickr Data

No matter how you choose to open Flickr's photo page, you need to get its URL first. If you look at the JSON data you are currently receiving for each photo, you can see that the photo page is not part of those results.

```
{
  "photos": {
    ...,
    "photo": [
      {
        "id": "9452133594",
        "owner": "44494372@N05",
        "secret": "d6d20af93e",
        "server": "7365",
        "farm": 8,
        "title": "Low and Wisoff at Work",
        "ispublic": 1,
        "isfriend": 0,
        "isfamily": 0,
        "url_s":"https://farm8.staticflickr.com/7365/9452133594_d6d20af93e_m.jpg"
      }, ...
    ]
  },
  "stat": "ok"
}
```

(Recall that `url_s` is the URL for the small version of the photo, not the full-size photo.)

You might think that you are in for some more JSON request writing. Fortunately, that is not the case. If you look at the Web Page URLs section of Flickr's documentation at `flickr.com/services/api/misc.urls.html`, you will see that you can create the URL for an individual photo's page like so:

```
https://www.flickr.com/photos/user-id/photo-id
```

The `photo-id` in the URL is the same as the value of the `id` attribute from your JSON data. You are already stashing that in `id` in **`GalleryItem`**. What about `user-id`? If you poke around the documentation, you will find that the `owner` attribute in your JSON data is the user ID. So if you pull out the `owner` attribute, you should be able to build the URL from your photo JSON data:

```
https://www.flickr.com/photos/owner/id
```

Update **`GalleryItem`** to put this plan into action.

### Listing 23.1 Adding code for the photo page (`GalleryItem.kt`)

```
@JsonClass(generateAdapter = true)
data class GalleryItem(
    val title: String,
    val id: String,
    @Json(name = "url_s") val url: String,
    val owner: String
) {
    val photoPageUri: Uri
        get() = Uri.parse("https://www.flickr.com/photos/")
            .buildUpon()
            .appendPath(owner)
            .appendPath(id)
            .build()
}
```

To determine the photo URL, you create a new `owner` property and add a computed property called `photoPageUri` to generate photo page URLs as discussed above. Because Moshi is translating your JSON responses into `GalleryItems` on your behalf, you can start using the `photoPageUri` property immediately, without any other code changes.

## The Easy Way: Implicit Intents

You will browse to this URL first by using your old friend the implicit intent. This intent will start up the browser with your photo URL.

The first step is to make your app listen for presses on an item in the **`RecyclerView`**. Update **`PhotoViewHolder`** to pass in a lambda expression that will be invoked with the **`Crime`**'s new `photoPageUri` property being passed in. Invoke the lambda expression when the root view is clicked.

### Listing 23.2 Firing an implicit intent when an item is pressed (`PhotoListAdapter.kt`)

```
class PhotoViewHolder(
    private val binding: ListItemGalleryBinding
) : RecyclerView.ViewHolder(binding.root) {
    fun bind(galleryItem: GalleryItem, onItemClicked: (Uri) -> Unit) {
        binding.itemImageView.load(galleryItem.url) {
            placeholder(R.drawable.bill_up_close)
        }
        binding.root.setOnClickListener { onItemClicked(galleryItem.photoPageUri) }
    }
}
```

Next, pass that same lambda expression into **`PhotoListAdapter`** as a constructor parameter and use it when binding a **`PhotoViewHolder`** in **`onBindViewHolder()`**.

### Listing 23.3 Binding **`PhotoViewHolder`** (`PhotoListAdapter.kt`)

```
class PhotoListAdapter(
    private val galleryItems: List<GalleryItem>,
    private val onItemClicked: (Uri) -> Unit
) : RecyclerView.Adapter<PhotoViewHolder>() {
    override fun onCreateViewHolder(
        parent: ViewGroup,
        viewType: Int
    ): PhotoViewHolder {
        ...
    }

    override fun onBindViewHolder(holder: PhotoViewHolder, position: Int) {
        val item = galleryItems[position]
        holder.bind(item, onItemClicked)
    }

    override fun getItemCount() = galleryItems.size
}
```

Finally, in **`PhotoGalleryFragment`**, pass in a lambda expression when creating an instance of **`PhotoListAdapter`**. Within that lambda expression, start an activity using an **`Intent`** containing that URL.

Listing 23.4 Starting your implicit intent (`PhotoGalleryFragment.kt`)

```
class PhotoGalleryFragment : Fragment() {
    ...
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        viewLifecycleOwner.lifecycleScope.launch {
            viewLifecycleOwner.repeatOnLifecycle(Lifecycle.State.STARTED) {
                photoGalleryViewModel.uiState.collect { state ->
                    ~~binding.photoGrid.adapter = PhotoListAdapter(state.images)~~
                    binding.photoGrid.adapter = PhotoListAdapter(
                        state.images
                    ) { photoPageUri ->
                        val intent = Intent(Intent.ACTION_VIEW, photoPageUri)
                        startActivity(intent)
                    }
                    searchView?.setQuery(state.query, false)
                    updatePollingState(state.isPolling)
                }
            }
        }
    }
    ...
}
```

That is it. Start up PhotoGallery and press a photo. Your browser app should pop up and load the photo page for the item you pressed (similar to the image on the left in Figure 23.1).

## The Harder Way: WebView

Using an implicit intent to display the photo page is easy and effective. But what if you do not want your app to open the browser?

Often, you want to display web content within your own activities instead of heading off to the browser. You may want to display HTML that you generate yourself, or you may want to lock down the browser somehow. For apps that include help documentation, it is common to implement it as a web page so that it is easy to update. Opening a web browser to a help web page does not look professional, and it prevents you from customizing behavior or integrating that web page into your own UI.

When you want to present web content within your own UI, you use the **`WebView`** class. We are calling this the "harder" way here, but it is pretty darned easy. (Anything is hard compared to using implicit intents.)

The first step is to create a new activity and fragment to display the **`WebView`**. Start, as usual, by defining a layout file: `res/layout/fragment_photo_page.xml`. Make **`ConstraintLayout`** the top-level layout. In the design view, drag a **`WebView`** into the **`ConstraintLayout`** as a child. (You will find **`WebView`** under the Widgets section.)

Once the **`WebView`** is added, add a constraint for every side to its parent. That gives you the following constraints:

- from the top of the **`WebView`** to the top of its parent
- from the bottom of the **`WebView`** to the bottom of its parent
- from the left of the **`WebView`** to the left of its parent
- from the right of the **`WebView`** to the right of its parent

Finally, change the height and width to 0 dp (match constraint) and change all the margins to `0`. Oh, and give your **`WebView`** an ID: `web_view`.

You may be thinking, "That **`ConstraintLayout`** is not useful." True enough – for the moment. You will fill it out later in the chapter with additional "chrome."

Next, get the rudiments of your fragment set up. Create **`PhotoPageFragment`**. You will need to inflate and bind your layout. All the work that this fragment does will occur in the **`onCreateView()`** function, so you do not need to hold on to a reference to the `binding` this time.

### Listing 23.5 Setting up your web browser fragment (`PhotoPageFragment.kt`)

```
class PhotoPageFragment : Fragment() {
    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View {
        val binding = FragmentPhotoPageBinding.inflate(
            inflater,
            container,
            false
        )

        return binding.root
    }
}
```

For now, this is little more than a skeleton. You will fill it out a bit more in a moment. But first, you need to set up the framework to navigate between fragments.

You will follow the same steps you used back in Chapter 13. We will walk you through the process quickly here; refer to that chapter if you need a refresher on any of the steps.

Start by setting up your Gradle build settings. You will use the Safe Args plugin again, so open up the `build.gradle` file labeled (Project: PhotoGallery) and include Safe Args in the list of plugins:

### Listing 23.6 Including the Safe Args plugin (`build.gradle`)

```
plugins {
    id 'com.android.application' version '7.1.2' apply false
    id 'com.android.library' version '7.1.2' apply false
    id 'org.jetbrains.kotlin.android' version '1.6.10' apply false
    id 'org.jetbrains.kotlin.kapt' version '1.6.10' apply false
    id 'androidx.navigation.safeargs.kotlin' version '2.4.1' apply false
}
...
```

Next, open the `app/build.gradle` file and enable the plugin. Also, include the two dependencies you need to enable fragment navigation.

### Listing 23.7 Configuring your app's build settings (`app/build.gradle`)

```
plugins {
    id 'com.android.application'
    id 'org.jetbrains.kotlin.android'
    id 'org.jetbrains.kotlin.kapt'
    id 'androidx.navigation.safeargs'
}
...
dependencies {
    ...
    implementation 'androidx.work:work-runtime-ktx:2.7.1'
    implementation 'androidx.navigation:navigation-fragment-ktx:2.4.1'
    implementation 'androidx.navigation:navigation-ui-ktx:2.4.1'
    ...
}
```

Sync your Gradle files. With the dependencies set up, create your `nav_graph.xml` file. In your navigation graph, you will need to handle a few tasks:

- Add both **`PhotoGalleryFragment`** and **`PhotoPageFragment`** as destinations. **`PhotoGalleryFragment`** will be your starting destination.
- Define a navigation action from **`PhotoGalleryFragment`** to **`PhotoPageFragment`**. Name the ID for this action `@+id/show_photo`.
- Add an argument for the **`PhotoPageFragment`** destination. Its name will be `photoPageUri`, and its type will be `android.net.Uri` (which is parcelable).

After you have completed those three steps, the code for your `nav_graph.xml` file will look like this:

```
<?xml version="1.0" encoding="utf-8"?>
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/nav_graph"
    app:startDestination="@id/photoGalleryFragment">

    <fragment
        android:id="@+id/photoGalleryFragment"
        android:name="com.bignerdranch.android.photogallery.PhotoGalleryFragment"
        android:label="PhotoGalleryFragment" >
        <action
            android:id="@+id/show_photo"
            app:destination="@id/photoPageFragment" />
    </fragment>
    <fragment
        android:id="@+id/photoPageFragment"
        android:name="com.bignerdranch.android.photogallery.PhotoPageFragment"
        android:label="PhotoPageFragment" >
        <argument
            android:name="photoPageUri"
            app:argType="android.net.Uri" />
    </fragment>
</navigation>
```

The last step to complete the navigation setup is to add a **NavHostFragment** inside the **FragmentContainerView** within your activity_main.xml file. Configure the same XML attributes as before.

### Listing 23.8 Adding a **NavHostFragment** (activity_main.xml)

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.fragment.app.FragmentContainerView
    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:id="@+id/fragment_container"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:name="com.bignerdranch.android.photogallery.PhotoGalleryFragment"
    android:name="androidx.navigation.fragment.NavHostFragment"
    app:defaultNavHost="true"
    app:navGraph="@navigation/nav_graph"
    tools:context=".MainActivity" />
```

With that, navigation is set up. Now, switch up your code in **PhotoGalleryFragment** to navigate to your new fragment instead of the implicit intent.

### Listing 23.9 Switching to launch your activity (PhotoGalleryFragment.kt)

```
class PhotoGalleryFragment : Fragment() {
    ...
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        viewLifecycleOwner.lifecycleScope.launch {
            viewLifecycleOwner.repeatOnLifecycle(Lifecycle.State.STARTED) {
                photoGalleryViewModel.uiState.collect { state ->
                    binding.photoGrid.adapter = PhotoListAdapter(
                        state.images
                    ) { photoPageUri ->
                        val intent = Intent(Intent.ACTION_VIEW, photoPageUri)
                        startActivity(intent)
                        findNavController().navigate(
                            PhotoGalleryFragmentDirections.showPhoto(
                                photoPageUri
                            )
                        )
                    }
                    searchView?.setQuery(state.query, false)
                    updatePollingState(state.isPolling)
                }
            }
        }
    }
    ...
}
```

Run PhotoGallery and press a picture. You should see a new empty screen pop up.

OK, now to get to the meat and make your fragment actually do something. You need to do three things to get your **WebView** to successfully display a Flickr photo page. The first one is straightforward – you need to tell it what URL to load.

The second thing you need to do is enable JavaScript. By default, JavaScript is off. You do not always need to have it on, but for Flickr, you do. (If you run Android Lint, it gives you a warning for doing this. It is worried about cross-site scripting attacks. You can suppress this Lint warning by annotating **onCreateView(…)** with `@SuppressLint("SetJavaScriptEnabled")`.)

Finally, you need to provide a default implementation of a class called **WebViewClient**. **WebViewClient** is used to respond to rendering events on a **WebView**. We will discuss this class a bit more after you enter the code.

Make these changes in **PhotoPageFragment**.

### Listing 23.10 Loading the URL into **WebView** (PhotoPageFragment.kt)

```
class PhotoPageFragment : Fragment() {
    private val args: PhotoPageFragmentArgs by navArgs()

    @SuppressLint("SetJavaScriptEnabled")
    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View {
        val binding = FragmentPhotoPageBinding.inflate(
            inflater,
            container,
            false
        )

        binding.apply {
            webView.apply {
                settings.javaScriptEnabled = true
                webViewClient = WebViewClient()
                loadUrl(args.photoPageUri.toString())
            }
        }

        return binding.root
    }
    ...
}
```

Loading the URL has to be done after configuring the **WebView**, so you do that last. Before that, you turn JavaScript on by accessing the `settings` property to get an instance of **WebSettings** and then setting `WebSettings.javaScriptEnabled = true`. **WebSettings** is the first of the three ways you can modify your **WebView**. It has various properties you can set, like the user agent string and text size.

After that, you add a **WebViewClient** to your **WebView**. To find out why, let's first address what happens without a **WebViewClient**.

A new URL can be loaded in a couple of different ways: The page can tell you to go to another URL on its own (a redirect), or the user can click a link. Without a **`WebViewClient`**, **`WebView`** will ask the activity manager to find an appropriate activity to load the new URL.

This is not what you want to have happen. Many sites (including Flickr's photo pages) immediately redirect to a mobile version of the same site when you load them from a phone browser. There is not much point to making your own view of the page if it is going to fire an implicit intent anyway when that happens.

If, on the other hand, you provide your own **`WebViewClient`** to your **`WebView`**, the process works differently. Instead of asking the activity manager what to do, **`WebView`** asks your **`WebViewClient`**. And in the default **`WebViewClient`** implementation, it says, "Go load the URL yourself!" So the page will appear in your **`WebView`**.

Run PhotoGallery, press an item, and you should see the item's photo page displayed in the **`WebView`** (like the image on the right in Figure 23.1).

## WebChromeClient

Since you are taking the time to create your own **`WebView`**, spruce it up a bit by adding a progress bar and updating the app bar's subtitle with the title of the loaded page. These decorations and the UI outside the **`WebView`** are referred to as *chrome* (not to be confused with the Google Chrome web browser). Crack open `fragment_photo_page.xml` once again.

In the design view, drag in a **`ProgressBar`** as a second child for your **`ConstraintLayout`**. Use the ProgressBar (Horizontal) version of **`ProgressBar`**. Delete the **`WebView`**'s top constraint, and then make its height fixed so that you can easily work with its constraint handles.

With that done, create the following constraints:

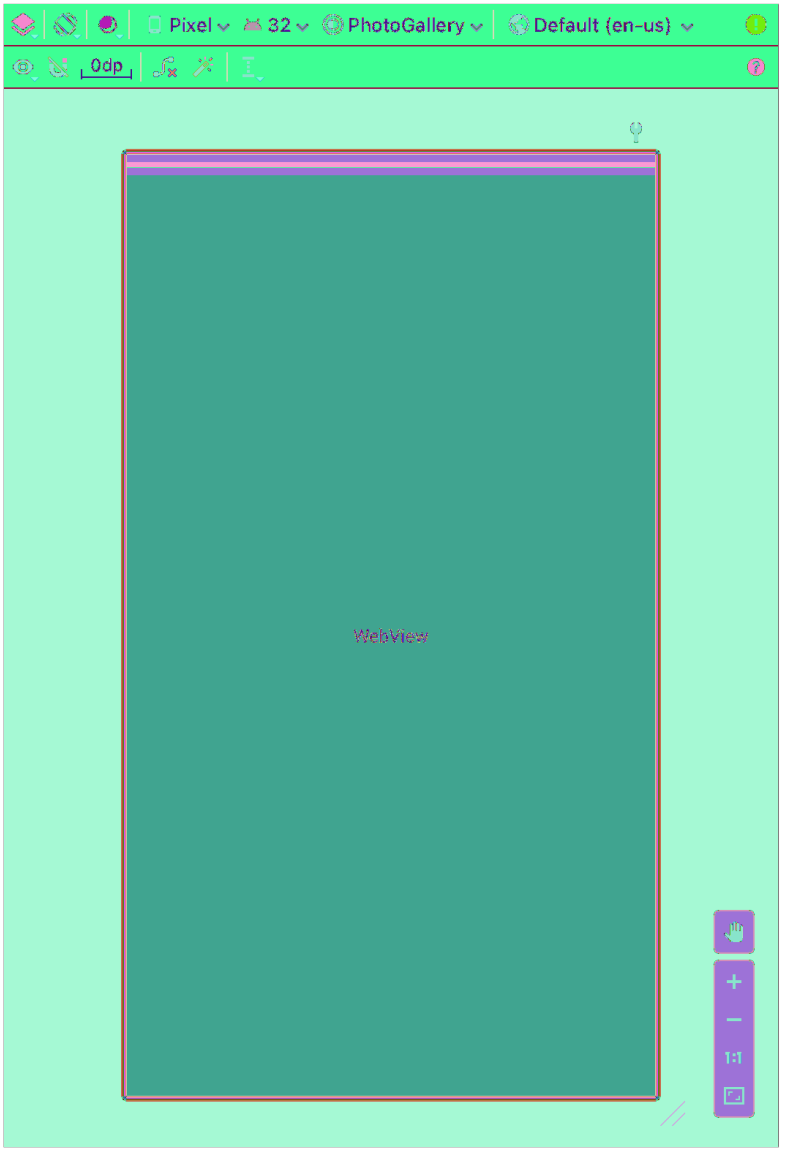
- from the **`ProgressBar`** to the top, right, and left of its parent
- from the **`WebView`**'s top to the bottom of the **`ProgressBar`**

Now, in the attributes window, change the height of the **`WebView`** back to 0 dp (match constraint) and change the **`ProgressBar`**'s height to wrap_content and width to 0 dp (match constraint).

With the **`ProgressBar`** selected, notice the two settings below layout_width and layout_height in the attributes window. They are both labeled visibility, but the second one has a wrench icon next to it. Change the first visibility to gone, and change the tool visibility (with the wrench icon) to visible. The first of these settings will hide the **`ProgressBar`** once the app is running on a device, and the second makes the progress bar visible in the layout preview. Finally, rename the **`ProgressBar`**'s ID to progress_bar.

Your result will look like Figure 23.2.

Figure 23.2 Adding a progress bar




To hook up the **`ProgressBar`**, you will use the second callback on **`WebView`**, which is **`WebChromeClient`**. **`Web`*`View`*`Client`** is an interface for responding to rendering events; **`Web`*`Chrome`*`Client`** is an interface for reacting to events that should change elements of chrome around the browser. This includes JavaScript alerts, favicons – and updates for loading progress and the title of the current page.

Hook it up in **`onCreateView(…)`**.

Listing 23.11 Using **`WebChromeClient`** (`PhotoPageFragment.kt`)

```
class PhotoPageFragment : Fragment() {
    ...
    @SuppressLint("SetJavaScriptEnabled")
    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        ...
        binding.apply {
            progressBar.max = 100

            webView.apply {
                settings.javaScriptEnabled = true
                webViewClient = WebViewClient()
                loadUrl(args.photoPageUri.toString())

                webChromeClient = object : WebChromeClient() {
                    override fun onProgressChanged(
                        webView: WebView,
                        newProgress: Int
                    ) {
                        if (newProgress == 100) {
                            progressBar.visibility = View.GONE
                        } else {
                            progressBar.visibility = View.VISIBLE
                            progressBar.progress = newProgress
                        }
                    }

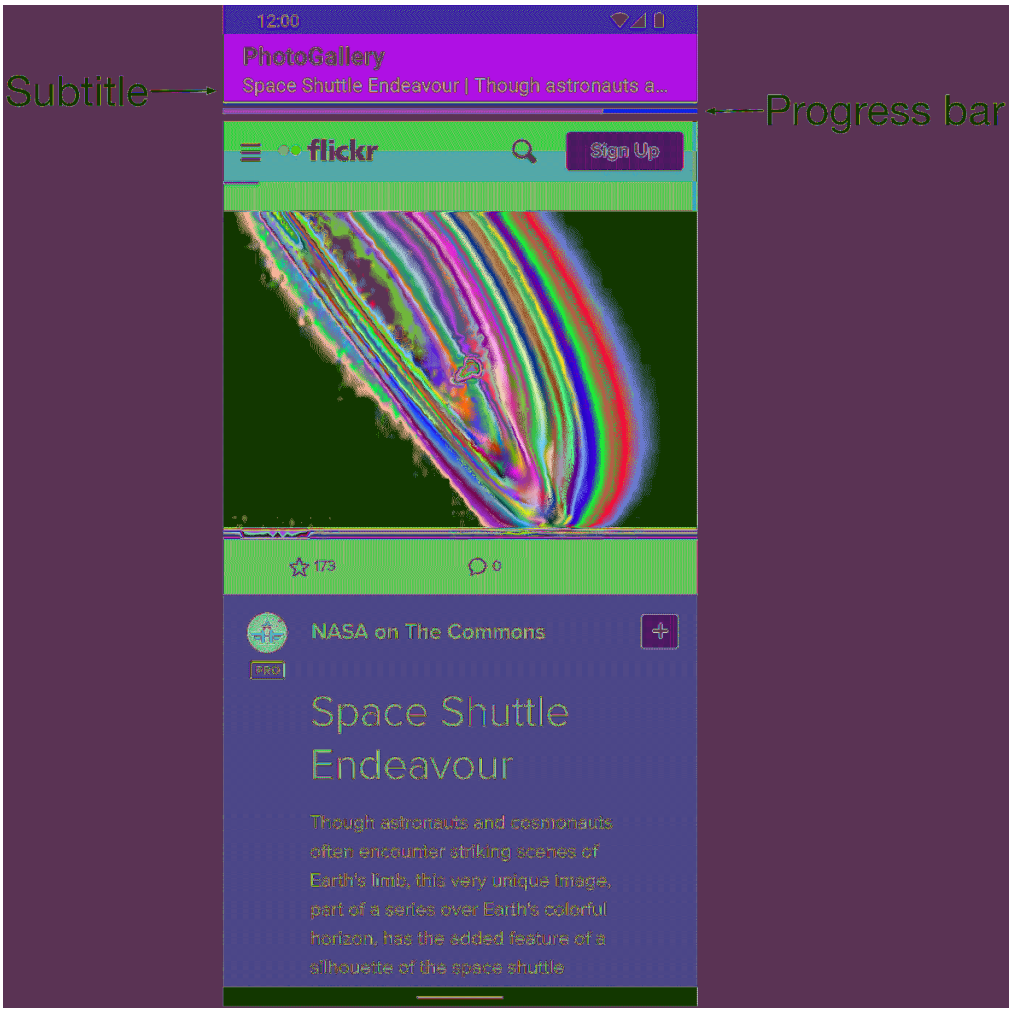
                    override fun onReceivedTitle(
                        view: WebView?,
                        title: String?
                    ) {
                        val parent = requireActivity() as AppCompatActivity
                        parent.supportActionBar?.subtitle = title
                    }
                }
            }
        }

        return binding.root
    }
    ...
}
```

Progress updates and title updates each have their own callback function, **`onProgressChanged(WebView, Int)`** and **`onReceivedTitle(WebView, String)`**. The progress you receive from **`onProgressChanged(WebView, Int)`** is an integer from 0 to 100. If it is 100, you know that the page is done loading, so you hide the **`ProgressBar`** by setting its visibility to `View.GONE`.

Run PhotoGallery to test your changes. It should look like Figure 23.3.

Figure 23.3 Fancy **`WebView`**




When you press a photo, **`PhotoPageFragment`** pops up. A progress bar displays as the page loads, and a subtitle reflecting the title received in **`onReceivedTitle(…)`** appears in the app bar. Once the page is loaded, the progress bar disappears.

## WebView vs a Custom UI

So there you have two ways to handle opening a photo's Flickr page from your app. There is, of course, a third option: You could create a custom UI to display the photo and its description.

A UI built natively (without **`WebView`**) would give you full control over how your app looks and behaves. Also, native UIs often feel more responsive and consistent to users. But there are a number of advantages to displaying web content instead of rolling out your own custom UI.

Displaying Flickr's site in a **`WebView`** lets you integrate a large feature much more quickly. You do not need to worry about fetching image descriptions, user account names, or other photo metadata to build out this UI. You can simply leverage what Flickr has already made available.

Another advantage to displaying web content is that the web content can change without you having to update your application. For example, if you need to display a privacy policy or terms of service in your app, you can choose to show a website instead of hardcoding the document into your application. That way, any changes can simply be pushed to a website instead of as an app update.

PhotoGallery is now complete. In the next two chapters, you will build two small apps as you learn about responding to touch events and creating animations.

## For the More Curious: WebView Updates

**`WebView`** is based on the Chromium open-source project. It shares the same rendering engine used by the Chrome for Android app, meaning pages should look and behave consistently across the two. (However, **`WebView`** does not have all the features Chrome for Android does. You can see a table comparing them at `https://developer.chrome.com/docs/multidevice/webview/`.)

Because it is based on Chromium, **`WebView`** stays up to date on web standards and JavaScript. From a development perspective, one of the most exciting features is the support for remote debugging of **`WebView`** using Chrome DevTools (which can be enabled by calling **`WebView.setWebContentsDebuggingEnabled()`**).

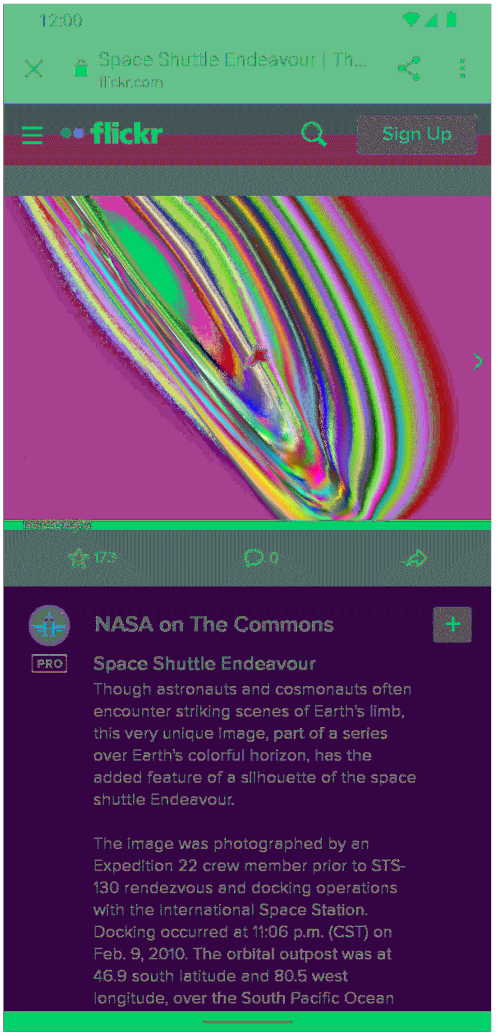
The Chromium layer of **`WebView`** is updated automatically from the Google Play Store. Users no longer have to wait for new releases of Android to receive security updates (and new features). So you can rest easy, knowing that Google works to keep the **`WebView`** components up to date.

## For the More Curious: Chrome Custom Tabs (Another Easy Way)

There is yet another way to display web content that is a hybrid of the two methods you used in this chapter. Chrome Custom Tabs (`developer.chrome.com/docs/android/custom-tabs/`) let you launch the Chrome web browser in a way that feels native to your application. You can configure its appearance to make it look like part of your app and feel like the user has never left your app.

Figure 23.4 shows an example of a custom tab. You can see that the result looks like a mix of Google Chrome and your **`PhotoPageActivity`**.

Figure 23.4 A Chrome custom tab



When you use a custom tab, it behaves very similar to launching Chrome. The browser instance even has access to information like the user's saved passwords, browser cache, and cookies from the full Chrome browser. This means that if the user had logged into Flickr in Chrome, then they would also be logged into Flickr in every custom tab. With **`WebView`**, the user would have to log into Flickr in both Chrome and PhotoGallery.

The downside to using a custom tab instead of a **`WebView`** is that you do not have as much control over the content you are displaying. For example, you cannot choose to use custom tabs in only the top half of your screen or to add navigation buttons to the bottom of a custom tab.

To start using Chrome Custom Tabs, you add this dependency:

```
implementation 'androidx.browser:browser:1.3.0'
```

You can then launch a custom tab. For example, in PhotoGallery you could launch a custom tab instead of **`PhotoPageFragment`**:

```
class PhotoGalleryFragment : Fragment() {
    ...
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        viewLifecycleOwner.lifecycleScope.launch {
            viewLifecycleOwner.repeatOnLifecycle(Lifecycle.State.STARTED) {
                photoGalleryViewModel.uiState.collect { state ->
                    binding.photoGrid.adapter = PhotoListAdapter(
                        state.images
                    ) { photoPageUri ->
                        findNavController().navigate(
                            PhotoGalleryFragmentDirections.showPhoto(
                                photoPageUri
                            )
                        )

                        CustomTabsIntent.Builder()
                            .setToolbarColor(ContextCompat.getColor(
                                requireContext(), R.color.colorPrimary))
                            .setShowTitle(true)
                            .build()
                            .launchUrl(requireContext(), photoPageUri)
                    }
                    searchView?.setQuery(state.query, false)
                    updatePollingState(state.isPolling)
                }
            }
        }
    }
    ...
}
```

With this change, a user who clicks a photo would see a custom tab like the one shown in Figure 23.4. (If the user did not have Chrome version 45 or higher installed, then PhotoGallery would fall back to using the system browser. The result would be just like when you used an implicit intent at the beginning of this chapter.)

## Challenge: Using the Back Button for Browser History

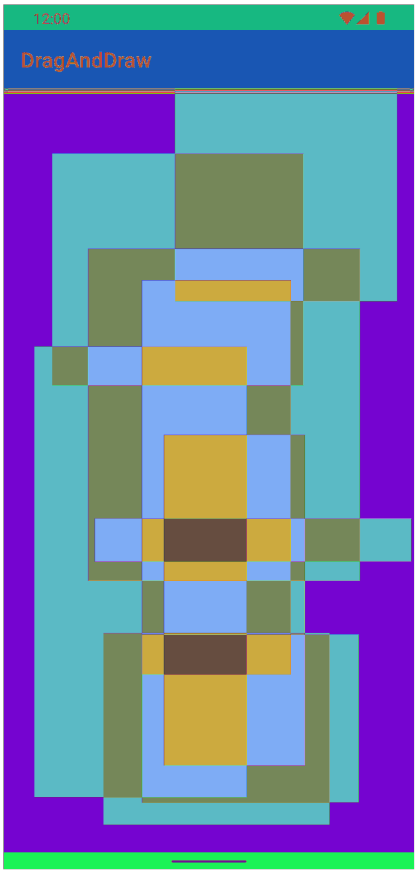
You may have noticed that you can follow other links within the **`WebView`** once you launch **`PhotoPageFragment`**. However, no matter how many links you follow, the Back button always brings you immediately back to **`PhotoGalleryFragment`**. What if you instead want the Back button to bring users through their browsing history within the **`WebView`**?

Implement this behavior by adding a callback to the activity's `onBackPressedDispatcher` property. Within that callback, use a combination of **`WebView`**'s browsing history functions (**`WebView.canGoBack()`** and **`WebView.goBack()`**) to do the right thing. If there are items in the **`WebView`**'s browsing history, go back to the previous item. Otherwise, allow the Back button to behave as it usually does by calling through to **`activity?.onBackPressed()`**.

# 24
# Custom Views and Touch Events

In this chapter, you will learn how to handle touch events by writing a custom subclass of **`View`** named **`BoxDrawingView`**. The **`BoxDrawingView`** class will be the star of a new project named DragAndDraw and will draw boxes in response to the user touching the screen and dragging. The finished product will look like Figure 24.1.

Figure 24.1 Boxes drawn in many shapes and sizes



## Setting Up the DragAndDraw Project

Create a new project named DragAndDraw and select API 24: Android 7.0 (Nougat) from the Minimum SDK dropdown.

Your newly generated **`MainActivity`** will host a **`BoxDrawingView`**, the custom view that you are going to write. All the drawing and touch-event handling will be implemented in **`BoxDrawingView`**.

## Creating a Custom View

Android provides many excellent standard views, but sometimes you need a custom view that presents visuals that are unique to your app.

While there are all kinds of custom views, you can shoehorn them into two broad categories:

*simple* — A simple view may be complicated inside; what makes it "simple" is that it has no child views. A simple view will almost always perform custom rendering.

*composite* — Composite views are composed of other view objects. Composite views typically manage child views but do not perform custom rendering. Instead, rendering is delegated to each child view.

There are three steps to follow when creating a custom view:

1. Pick a superclass. For a simple custom view, **`View`** is a blank canvas, so it is the most common choice. For a composite custom view, choose an appropriate layout class, such as **`FrameLayout`**.
2. Subclass this class and override the constructors from the superclass.
3. Override other key functions to customize behavior.

## Creating BoxDrawingView

**BoxDrawingView** will be a simple view and a direct subclass of **View**.

Create a new class named **BoxDrawingView** and make **View** its superclass. In BoxDrawingView.kt, add a constructor that takes in a **Context** object and a nullable **AttributeSet** with a default of null.

### Listing 24.1 Initial implementation for **BoxDrawingView** (BoxDrawingView.kt)

```
class BoxDrawingView(
    context: Context,
    attrs: AttributeSet? = null
) : View(context, attrs) {

}
```

Providing the null default value for the attribute set effectively provides two constructors for your view. Two constructors are needed, because your view could be instantiated in code or from a layout file. Views instantiated from a layout file receive an instance of **AttributeSet** containing the XML attributes that were specified in XML. Even if you do not plan on using both constructors, it is good practice to include them.

Next, update your res/layout/activity_main.xml layout file to use your new view.

### Listing 24.2 Adding **BoxDrawingView** to the layout (res/layout/activity_main.xml)

```
<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="com.bignerdranch.android.draganddraw.MainActivity">
</androidx.constraintlayout.widget.ConstraintLayout>
<com.bignerdranch.android.draganddraw.BoxDrawingView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

You must use **BoxDrawingView**'s fully qualified class name so that the layout inflater can find it. The inflater works through a layout file creating **View** instances. If the element name is an unqualified class name, then the inflater looks for a class with that name in the android.view and android.widget packages. If the class lives somewhere else, then the layout inflater will not find it, and your app will crash.

So, for custom classes and other classes that live outside of android.view and android.widget, you must always specify the fully qualified class name.

Run DragAndDraw to confirm that all the connections are correct. All you will see is an empty view (Figure 24.2).

Figure 24.2 **`BoxDrawingView`** with no boxes




The next step is to get **`BoxDrawingView`** listening for touch events and using the information from them to draw boxes on the screen.

## Handling Touch Events

One way to listen for touch events is to set a touch event listener using the following **`View`** function:

```
fun setOnTouchListener(l: View.OnTouchListener)
```

This function works the same way as **`setOnClickListener(View.OnClickListener)`**. You provide an implementation of **`View.OnTouchListener`**, and your listener will be called every time a touch event happens.

However, because you are subclassing **`View`**, you can take a shortcut and override this **`View`** function instead:

```
override fun onTouchEvent(event: MotionEvent): Boolean
```

This function receives an instance of **`MotionEvent`**, a class that describes the touch event, including its location and its *action*. The action describes the stage of the event:

| Action constants | Description |
|---|---|
| `ACTION_DOWN` | user's finger touches the screen |
| `ACTION_MOVE` | user's finger moves on the screen |
| `ACTION_UP` | user's finger lifts off the screen |
| `ACTION_CANCEL` | a parent view intercepts the touch event |

In your implementation of **`onTouchEvent(MotionEvent)`**, you can check the value by accessing the `action` class property on the `event`.

Let's get to it. In `BoxDrawingView.kt`, add a log tag and then an implementation of **`onTouchEvent(MotionEvent)`** that logs a message for each of the four actions.

Listing 24.3 Implementing **`BoxDrawingView`** (`BoxDrawingView.kt`)

```
private const val TAG = "BoxDrawingView"

class BoxDrawingView(
    context: Context,
    attrs: AttributeSet? = null
) : View(context, attrs) {

    override fun onTouchEvent(event: MotionEvent): Boolean {
        val current = PointF(event.x, event.y)
        var action = ""
        when (event.action) {
            MotionEvent.ACTION_DOWN -> {
                action = "ACTION_DOWN"
            }
            MotionEvent.ACTION_MOVE -> {
                action = "ACTION_MOVE"
            }
            MotionEvent.ACTION_UP -> {
                action = "ACTION_UP"
            }
            MotionEvent.ACTION_CANCEL -> {
                action = "ACTION_CANCEL"
            }
        }

        Log.i(TAG, "$action at x=${current.x}, y=${current.y}")

        return true
    }
}
```

Notice that you package your X and Y coordinates in a **`PointF`** object. You want to pass these two values together as you go through the rest of the chapter. **`PointF`** is a container class provided by Android that does this for you.

Run DragAndDraw again and pull up Logcat. Touch the screen and drag your finger. (On the emulator, click and drag.) You should see a report of the X and Y coordinates of every touch action that **`BoxDrawingView`** receives.

## Tracking across motion events

**BoxDrawingView** is intended to draw boxes on the screen, not just log coordinates. There are a few problems to solve to get there.

First, to define a box, you need two points: the start point (where the finger was initially placed) and the end point (where the finger currently is). So defining a box requires keeping track of data from more than one **MotionEvent**. You will store this data in a **Box** object.

Create a class named **Box** to represent the data that defines a single box.

Listing 24.4 Adding **Box** (Box.kt)

```
data class Box(val start: PointF) {

    var end: PointF = start

    val left: Float
        get() = Math.min(start.x, end.x)

    val right: Float
        get() = Math.max(start.x, end.x)

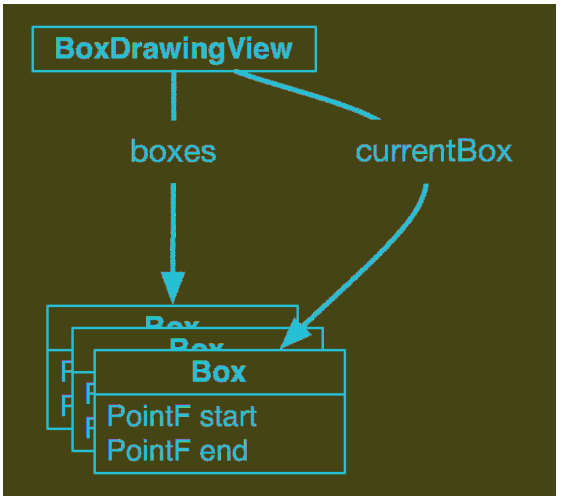
    val top: Float
        get() = Math.min(start.y, end.y)

    val bottom: Float
        get() = Math.max(start.y, end.y)

}
```

When the user touches **BoxDrawingView**, a new **Box** will be created and added to a list of existing boxes (Figure 24.3).

Figure 24.3 Objects in DragAndDraw



Back in **BoxDrawingView**, use your new **Box** object to track your drawing state.

Listing 24.5 Tracking **Box**es (BoxDrawingView.kt)

```
class BoxDrawingView(
    context: Context,
    attrs: AttributeSet? = null
) : View(context, attrs) {

    private var currentBox: Box? = null
    private val boxes = mutableListOf<Box>()

    override fun onTouchEvent(event: MotionEvent): Boolean {
        val current = PointF(event.x, event.y)
        var action = ""
        when (event.action) {
            MotionEvent.ACTION_DOWN -> {
                action = "ACTION_DOWN"
                // Reset drawing state
                currentBox = Box(current).also {
                    boxes.add(it)
                }
            }
            MotionEvent.ACTION_MOVE -> {
                action = "ACTION_MOVE"
                updateCurrentBox(current)
            }
            MotionEvent.ACTION_UP -> {
                action = "ACTION_UP"
                updateCurrentBox(current)
                currentBox = null
            }
            MotionEvent.ACTION_CANCEL -> {
                action = "ACTION_CANCEL"
                currentBox = null
            }
        }

        Log.i(TAG, "$action at x=${current.x}, y=${current.y}")

        return true
    }

    private fun updateCurrentBox(current: PointF) {
        currentBox?.let {
            it.end = current
            invalidate()
        }
    }
}
```

Any time an `ACTION_DOWN` motion event is received, you set `currentBox` to be a new **Box** with its origin as the event's location. This new **Box** is added to the list of boxes. (In the next section, when you implement custom drawing, **BoxDrawingView** will draw every **Box** within this list to the screen.)

As the user's finger moves around the screen, you update `currentBox.end`. Then, when the touch is canceled or when the user's finger leaves the screen, you update the current box with the final reported location and null out `currentBox` to end your draw motion. The **`Box`** is complete; it is stored safely in the list but will no longer be updated about motion events.

The call to **`invalidate()`** in the **`updateCurrentBox()`** function forces **`BoxDrawingView`** to redraw itself so that the user can see the box while dragging across the screen. Which brings you to the next step: drawing the boxes to the screen.

## Rendering Inside onDraw(Canvas)

When your application is launched, all its views are *invalid*. This means that they have not drawn anything to the screen. To fix this situation, Android calls the top-level **`View`**'s **`draw()`** function. This causes that view to draw itself, which causes its children to draw themselves. Those children's children then draw themselves, and so on down the hierarchy. When all the views in the hierarchy have drawn themselves, the top-level **`View`** is no longer invalid.

You can also manually specify that a view is invalid, even if it is currently onscreen. This will cause the system to redraw the view with any necessary updates. You will mark the **`BoxDrawingView`** as invalid any time the user creates a new box or resizes a box by moving their finger. This will ensure that users can see what their boxes look like as they create them.

To hook into this drawing, you override the following **`View`** function:

```
protected fun onDraw(canvas: Canvas)
```

The call to **`invalidate()`** that you make in response to `ACTION_MOVE` in **`onTouchEvent(MotionEvent)`** makes the **`BoxDrawingView`** invalid again. This causes it to redraw itself and will cause **`onDraw(Canvas)`** to be called again.

Now, consider the **`Canvas`** parameter. **`Canvas`** and **`Paint`** are the two main drawing classes in Android:

- The **`Canvas`** class has all the drawing operations you perform. The functions you call on **`Canvas`** determine where and what you draw – a line, a circle, a word, or a rectangle.
- The **`Paint`** class determines how these operations are done. The functions you call on **`Paint`** specify characteristics – whether shapes are filled, which font text is drawn in, and what color lines are.

In `BoxDrawingView.kt`, create two **`Paint`** objects when the **`BoxDrawingView`** is initialized.

Listing 24.6 Creating your paint (`BoxDrawingView.kt`)

```
class BoxDrawingView(
    context: Context,
    attrs: AttributeSet? = null
) : View(context, attrs) {

    private var currentBox: Box? = null
    private val boxes = mutableListOf<Box>()
    private val boxPaint = Paint().apply {
        color = 0x22ff0000.toInt()
    }
    private val backgroundPaint = Paint().apply {
        color = 0xfff8efe0.toInt()
    }
    ...
}
```

Armed with paint, you can now draw your boxes to the screen.

Listing 24.7 Overriding **`onDraw(Canvas)`** (`BoxDrawingView.kt`)

```
class BoxDrawingView(context: Context, attrs: AttributeSet? = null) :
        View(context, attrs)
    ...
    override fun onDraw(canvas: Canvas) {
        // Fill the background
        canvas.drawPaint(backgroundPaint)

        boxes.forEach { box ->
            canvas.drawRect(box.left, box.top, box.right, box.bottom, boxPaint)
        }
    }
}
```

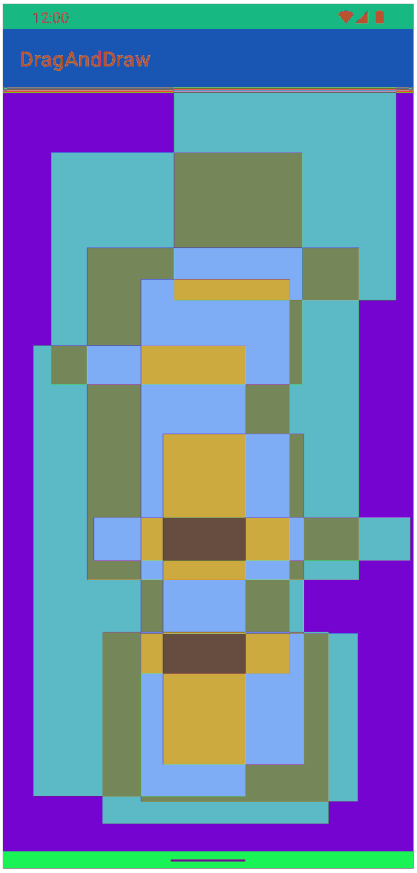
The first part of this code is straightforward: Using your off-white background paint, you fill the canvas with a backdrop for your boxes.

Then, for each box in your list of boxes, you determine what the left, right, top, and bottom of the box should be by looking at the two points for the box. On Android, the origin is the top-left corner, so the left and top values will be the minimum values, and the bottom and right values will be the maximum values.

After calculating these values, you call **`Canvas.drawRect(…)`** to draw a red rectangle onto the screen.

Run DragAndDraw and draw some red rectangles (Figure 24.4).

Figure 24.4 An expression of programmerly emotion




And that is it. You have now created a view that captures its own touch events and performs its own drawing.

## For the More Curious: Detecting Gestures

Another option for handling touch events is to use a **`GestureDetectorCompat`** object (`developer.android.com/reference/androidx/core/view/GestureDetectorCompat`). Instead of adding logic to detect events like a swipe or a fling, the **`GestureDetectorCompat`** has listeners that do the heavy lifting and notify you when a particular event occurs.

Many cases do not require the full control provided by overriding the **`onTouch`** function, so using the **`GestureDetectorCompat`** instead is a great choice.

## Challenge: Saving State

Figure out how to persist your boxes across orientation changes from within your **`View`**. This can be done with the following **`View`** functions:

```
protected fun onSaveInstanceState(): Parcelable
protected fun onRestoreInstanceState(state: Parcelable)
```

These functions work like **`Activity`** and **`Fragment`**'s **`onSaveInstanceState(Bundle)`** and **`ViewModel`**'s **`SavedStateHandle`**, but with a few key differences. First, they will only be called if your **`View`** has an ID. Second, instead of taking in a **`Bundle`**, they return and process an object that implements the **`Parcelable`** interface.

Since **`Bundle`** implements the **`Parcelable`** interface, you can still use that here. You could store your boxes' data by storing them within a **`Bundle`**.

You could also try out the Parcelize Kotlin compiler plugin to help you generate code that implements the **`Parcelable`** for you. Parcelize is used in Chapter 27; you can flip ahead to see how to incorporate and use it.

Finally, you must also maintain the saved state of **`BoxDrawingView`**'s parent, the **`View`** class. Save the result of **`super.onSaveInstanceState()`** in your new **`Bundle`** and send that result to the superclass when calling **`super.onRestoreInstanceState(Parcelable)`**.

## Challenge: Rotating Boxes

For a harder challenge, make it so that you can use a second finger to rotate your rectangles. To do this, you will need to handle multiple pointers in your **`MotionEvent`** handling code. You will also need to rotate your canvas.

When dealing with multiple touches, you need these extra ideas:

| | |
|---|---|
| *pointer index* | tells you which pointer in the current set of pointers the event is for |
| *pointer ID* | gives you a unique ID for a specific finger in a gesture |

The pointer index may change, but the pointer ID will not.

For more details, check out the documentation for the following **`MotionEvent`** functions:

```
final fun getActionMasked(): Int
final fun getActionIndex(): Int
final fun getPointerId(pointerIndex: Int): Int
final fun getX(pointerIndex: Int): Float
final fun getY(pointerIndex: Int): Float
```

Also look at the documentation for the `ACTION_POINTER_UP` and `ACTION_POINTER_DOWN` constants.

## Challenge: Accessibility Support

Built-in views provide support for accessibility options like TalkBack and Switch Access. Creating your own views places the responsibility on you as the developer to make sure your app is accessible. As a final challenge for this chapter, make your **`BoxDrawingView`** describable with TalkBack for screen readers.

There are several ways you can approach this. You could provide an overall summary of the view and tell the user how much of the view is covered in boxes. Alternatively, you could also make each box an accessible element and have it describe its location on the screen to the user. Refer to Chapter 19 for more information on making your apps accessible.

# 25
# Property Animation

For an app to be functional, all you need to do is write your code correctly so that it does not crash. For an app to be a joy to use, though, you need to give it more love than that. You need to make it feel like a real, physical phenomenon playing out on a phone or tablet's screen.

Real things move. To make your UI move, you *animate* its elements into new positions.

In this chapter, you will write an app called Sunset that shows a scene of the sun in the sky. When you press the scene, the sun will slide down below the horizon and the sky will change colors, like a sunset.

## Building the Scene

The first step is to build the scene that will be animated. Create a new project called Sunset. Make sure that your minimum API level is set to 24.

Before setting up anything else, open the `app/build.gradle` file. As in other apps, you are going to use View Binding to help you out in Sunset.

Listing 25.1 Setting up View Binding (`app/build.gradle`)

```
...
android {
    ...
    kotlinOptions {
        jvmTarget = '1.8'
    }
    buildFeatures {
        viewBinding true
    }
}
```

Do not forget to sync your changes with Gradle.

A sunset by the sea should be colorful, so it will help to start by naming a few colors. Open the `colors.xml` file in your `res/values` folder and add the following values to it.

### Listing 25.2 Adding sunset colors (`res/values/colors.xml`)

```
<resources>
    ...
    <color name="teal_700">#FF018786</color>
    <color name="black">#FF000000</color>
    <color name="white">#FFFFFFFF</color>

    <color name="bright_sun">#fcfcb7</color>
    <color name="blue_sky">#1e7ac7</color>
    <color name="sunset_sky">#ec8100</color>
    <color name="night_sky">#05192e</color>
    <color name="sea">#224869</color>
</resources>
```

Rectangular views will make for a fine impression of the sky and the sea. But, outside of Minecraft, people will not buy a rectangular sun, no matter how much you argue in favor of its technical simplicity. So, in the `res/drawable/` folder, add an oval shape drawable for a circular sun called `sun.xml`.

### Listing 25.3 Adding a sun XML drawable (`res/drawable/sun.xml`)

```
<shape xmlns:android="http://schemas.android.com/apk/res/android"
        android:shape="oval">
    <solid android:color="@color/bright_sun" />
</shape>
```

When you display this oval in a square view, you will get a circle. People will nod their heads in approval and then think about the real sun up in the sky.

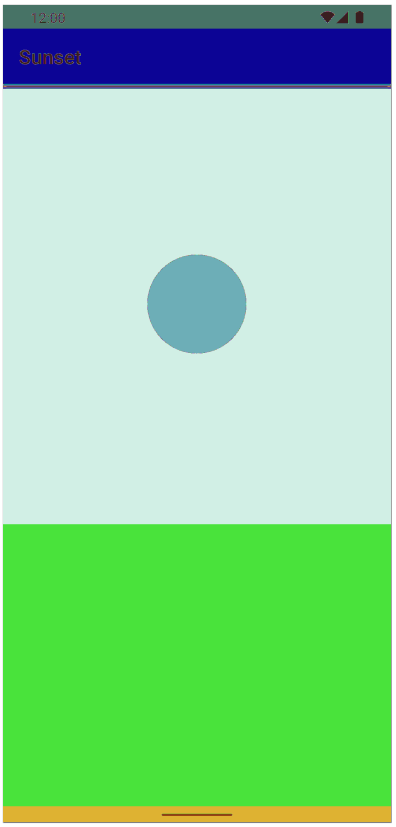
Next, build the entire scene out in a layout file. Open `res/layout/activity_main.xml`, delete the current contents, and add the following.

Listing 25.4 Setting up the layout (`res/layout/activity_main.xml`)

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
              android:id="@+id/scene"
              android:orientation="vertical"
              android:layout_width="match_parent"
              android:layout_height="match_parent">
    <FrameLayout
            android:id="@+id/sky"
            android:layout_width="match_parent"
            android:layout_height="0dp"
            android:layout_weight="0.61"
            android:background="@color/blue_sky">
        <ImageView
                android:id="@+id/sun"
                android:layout_width="100dp"
                android:layout_height="100dp"
                android:layout_gravity="center"
                android:src="@drawable/sun" />
    </FrameLayout>
    <View
            android:layout_width="match_parent"
            android:layout_height="0dp"
            android:layout_weight="0.39"
            android:background="@color/sea" />
</LinearLayout>
```

Check out the preview. You should see a daytime scene of the sun in a blue sky over a dark blue sea. Take a moment to run Sunset to make sure everything is hooked up correctly before moving on. It should look like Figure 25.1. Ahhh.

Figure 25.1 Before sunset



# Simple Property Animation

Before you start animating, you will want to inflate and bind your layout in **`MainActivity`**, in the **`onCreate(…)`** function.

### Listing 25.5 Inflating and binding the layout (`MainActivity.kt`)

```
class MainActivity : AppCompatActivity() {

    private lateinit var binding: ActivityMainBinding

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)
    }
}
```

Now, it is time to animate the sun down below the horizon. Here is the plan: Smoothly move `binding.sun` so that its top is right at the edge of the bottom of the sky. Since the bottom of the sky and the top of the sea are the same, the sun will be hidden behind the sea view. You will do this by *translating* the location of the top of `binding.sun` to the bottom of its parent.

The reason the sun view moves behind the sea is not immediately apparent. This has to do with the draw order of the views. Views are drawn in the order in which they are declared in the layout, so views declared later in the layout are drawn on top of those further up.

In this case, since the sun view is declared before the sea view, the sea view is on top of the sun view. When the sun animates past the sea, it will appear to go behind the sea.

The first step is to find where the animation should start and end. Write this first step in a new function called **`startAnimation()`**.

### Listing 25.6 Getting the start and end values (`MainActivity.kt`)

```
class MainActivity : AppCompatActivity() {
    ...
    override fun onCreate(savedInstanceState: Bundle?) {
        ...
    }

    private fun startAnimation() {
        val sunYStart = binding.sun.top.toFloat()
        val sunYEnd = binding.sky.height.toFloat()
    }
}
```

The `top` property is one of four properties on **`View`** that return the *local layout rect* for that view: `top`, `bottom`, `right`, and `left`. A rect (short for rectangle) is the rectangular bounding box for the view, which is specified by those four properties. A view's local layout rect specifies the position and size of that view in relation to its parent, as determined when the view was laid out.

It is possible to change the location of the view onscreen by modifying these values, but it is not recommended. They are reset every time a layout pass occurs, so they tend not to hold their value.

In any event, the animation will start with the top of the view at its current location. It needs to end with the top at the bottom of `binding.sun`'s parent, `binding.sky`. To get it there, it should be as far down as `binding.sky` is tall, which you find by calling **`height.toFloat()`**. The `height` property's value is the same as `bottom` minus `top`.

Now that you know where the animation should start and end, create and run an **`ObjectAnimator`** to perform it.

### Listing 25.7 Creating a sun animator (`MainActivity.kt`)

```
...
private fun startAnimation() {
    val sunYStart = binding.sun.top.toFloat()
    val sunYEnd = binding.sky.height.toFloat()

    val heightAnimator = ObjectAnimator
        .ofFloat(binding.sun, "y", sunYStart, sunYEnd)
        .setDuration(3000)

    heightAnimator.start()
}
...
```

We will come back to how **`ObjectAnimator`** works in a moment. First, hook up **`startAnimation()`** so that it is called every time the user presses anywhere in the scene.

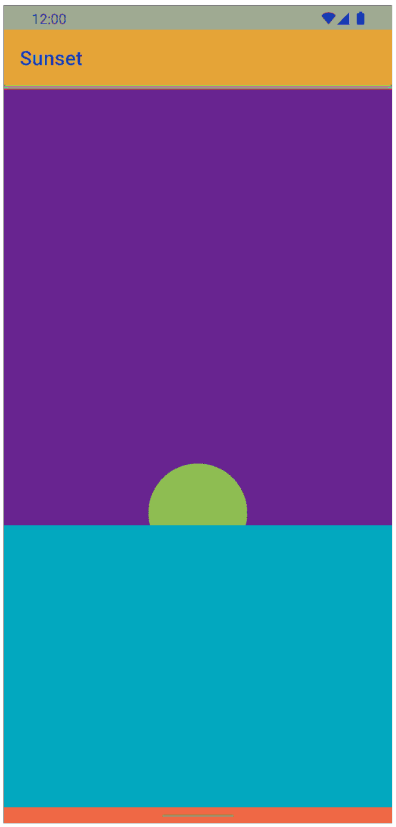
### Listing 25.8 Starting animation on press (`MainActivity.kt`)

```
...
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    binding = ActivityMainBinding.inflate(layoutInflater)
    setContentView(binding.root)

    binding.scene.setOnClickListener {
        startAnimation()
    }
}
...
```

Run Sunset and press anywhere in the scene to run the animation (Figure 25.2).

Figure 25.2 Setting sun




You should see the sun move below the horizon.

Here is how it works: **`ObjectAnimator`** is a *property animator*. Instead of knowing specifically about how to move a view around the screen, a property animator repeatedly calls property setter functions with different values.

For example, imagine the Y coordinate of the top of the sun is 120.00 and the Y coordinate of the bottom of the sky is 360.00. The code that you just wrote would create an **`ObjectAnimator`** in the call to **`ObjectAnimator.ofFloat(binding.sun, "y", sunYStart, sunYEnd)`**. When that **`ObjectAnimator`** is started, it would repeatedly call **`binding.sun.setY(Float)`** with values starting at `120.00` and moving up. Like this:

```
binding.sun.setY(120.00)
binding.sun.setY(121.33)
binding.sun.setY(122.67)
binding.sun.setY(124.00)
binding.sun.setY(125.33)
...
```

… and so on, until it finally calls **`binding.sun.setY(360.00)`**. This process of finding values between a starting and ending point is called *interpolation*. Between each interpolated value, a little time passes, which makes it look like the view is moving.

## View transformation properties

Property animators are great, but with them alone it would be impossible to animate a view as easily as you just did. Modern Android property animation works in concert with *transformation properties*.

We said earlier that your view has a local layout rect, which is the position and size it is assigned in the layout process. You can move the view around after that by setting additional properties on the view, called transformation properties.

You have three properties to rotate the view (`rotation`, `pivotX`, and `pivotY`, shown in Figure 25.3), two properties to scale the view vertically and horizontally (`scaleX` and `scaleY`, shown in Figure 25.4), and two properties to move the view around the screen (`translationX` and `translationY`, shown in Figure 25.5).

Figure 25.3 View rotation

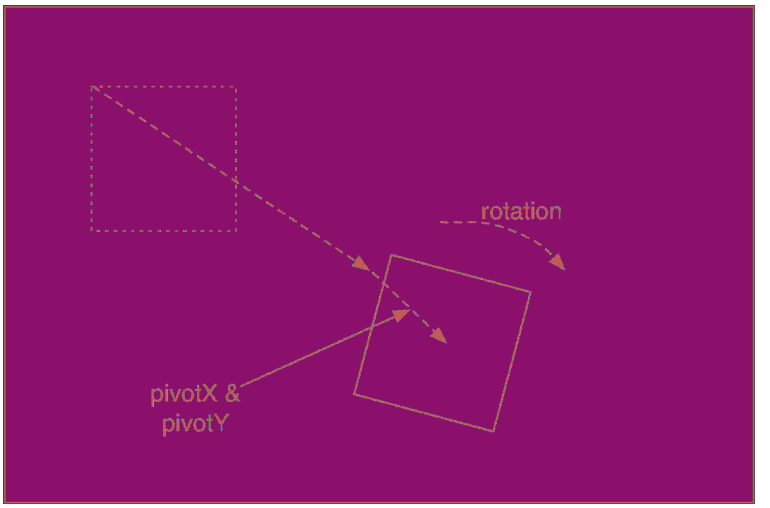


Figure 25.4 View scaling

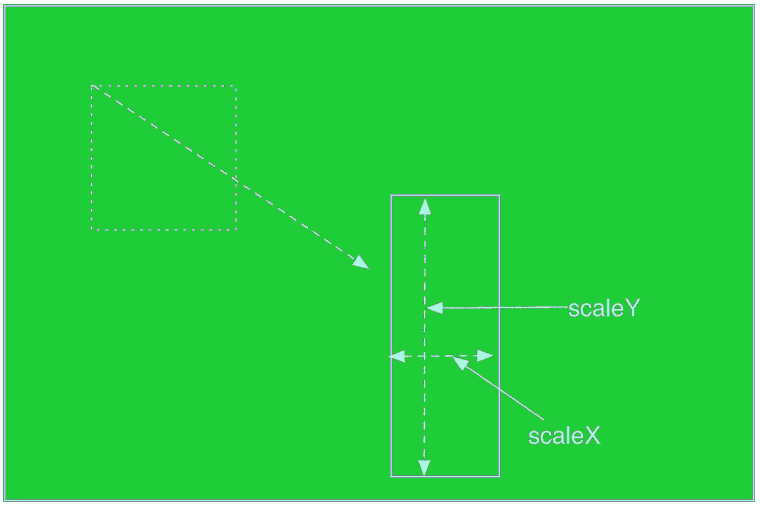
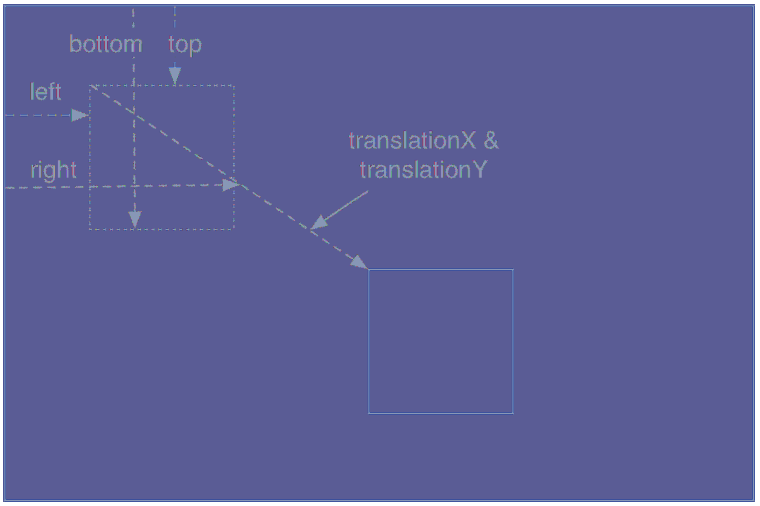



Figure 25.5 View translation




All these properties can be fetched and modified. For example, if you wanted to know the current value of `translationX`, you would invoke `view.translationX`. If you wanted to set it, you would invoke `view.translationX = Float`.

So what does the `y` property do? The `x` and `y` properties are conveniences built on top of local layout coordinates and the transformation properties. They allow you to write code that simply says, "Put this view at this X coordinate and this Y coordinate." Under the hood, these properties will modify `translationX` or `translationY` to put the view where you want it to be. That means that setting `binding.sun.y = 50` really means this:

```
binding.sun.translationY = 50 - binding.sun.top
```

## Using different interpolators

Your animation, while pretty, is abrupt. If the sun was really sitting there perfectly still in the sky, it would take a moment for it to accelerate into the animation you see. To add this sensation of acceleration, all you need to do is use a **`TimeInterpolator`**. **`TimeInterpolator`** has one role: to change the way your animation goes from point A to point B.

Use an **`AccelerateInterpolator`** in **`startAnimation()`** to make your sun speed up a bit at the beginning.

### Listing 25.9 Adding acceleration (`MainActivity.kt`)

```
private fun startAnimation() {
    val sunYStart = binding.sun.top.toFloat()
    val sunYEnd = binding.sky.height.toFloat()

    val heightAnimator = ObjectAnimator
        .ofFloat(binding.sun, "y", sunYStart, sunYEnd)
        .setDuration(3000)
    heightAnimator.interpolator = AccelerateInterpolator()

    heightAnimator.start()
}
```

Run Sunset one more time and press to see your animation. Your sun should now start moving slowly and accelerate to a quicker pace as it moves toward the horizon.

There are a lot of styles of motion you might want to use in your app, so there are a lot of different **`TimeInterpolator`**s. To see all the interpolators that ship with Android, look at the Known indirect subclasses section in the reference documentation for **`TimeInterpolator`**.

## Color evaluation

Now that your sun is animating down, let's animate the sky to a sunset-appropriate color. Pull the colors you defined in `colors.xml` into properties using a lazy delegate.

### Listing 25.10 Pulling out sunset colors (`MainActivity.kt`)

```
class MainActivity : AppCompatActivity() {

    private lateinit var binding: ActivityMainBinding

    private val blueSkyColor: Int by lazy {
        ContextCompat.getColor(this, R.color.blue_sky)
    }
    private val sunsetSkyColor: Int by lazy {
        ContextCompat.getColor(this, R.color.sunset_sky)
    }
    private val nightSkyColor: Int by lazy {
        ContextCompat.getColor(this, R.color.night_sky)
    }
    ...
}
```

Now add an animation to **`startAnimation()`** to animate the sky from `blueSkyColor` to `sunsetSkyColor`.

### Listing 25.11 Animating sky colors (`MainActivity.kt`)

```
private fun startAnimation() {
    val sunYStart = binding.sun.top.toFloat()
    val sunYEnd = binding.sky.height.toFloat()

    val heightAnimator = ObjectAnimator
        .ofFloat(binding.sun, "y", sunYStart, sunYEnd)
        .setDuration(3000)
    heightAnimator.interpolator = AccelerateInterpolator()

    val sunsetSkyAnimator = ObjectAnimator
        .ofInt(binding.sky, "backgroundColor", blueSkyColor, sunsetSkyColor)
        .setDuration(3000)

    heightAnimator.start()
    sunsetSkyAnimator.start()
}
```

This seems like it is headed in the right direction, but if you run it you will see that something is amiss. Instead of moving smoothly from blue to orange, the colors will kaleidoscope wildly.

The reason this happens is that a color integer is not a simple number. It is four smaller numbers schlupped together into one `Int`. So for **`ObjectAnimator`** to properly evaluate which color is halfway between blue and orange, it needs to know how that works.

When **`ObjectAnimator`**'s normal understanding of how to find values between the start and end is insufficient, you can provide a subclass of **`TypeEvaluator`** to fix things. A **`TypeEvaluator`** is an object that tells **`ObjectAnimator`** what value is, say, a quarter of the way between a start value and an end value. Android provides a subclass of **`TypeEvaluator`** called **`ArgbEvaluator`** that will do the trick here.

### Listing 25.12 Providing **`ArgbEvaluator`** (`MainActivity.kt`)

```
private fun startAnimation() {
    val sunYStart = binding.sun.top.toFloat()
    val sunYEnd = binding.sky.height.toFloat()

    val heightAnimator = ObjectAnimator
        .ofFloat(binding.sun, "y", sunYStart, sunYEnd)
        .setDuration(3000)
    heightAnimator.interpolator = AccelerateInterpolator()

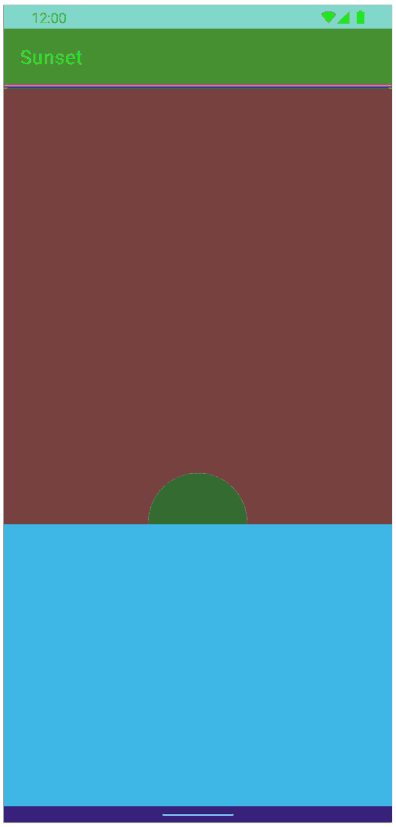
    val sunsetSkyAnimator = ObjectAnimator
        .ofInt(binding.sky, "backgroundColor", blueSkyColor, sunsetSkyColor)
        .setDuration(3000)
    sunsetSkyAnimator.setEvaluator(ArgbEvaluator())

    heightAnimator.start()
    sunsetSkyAnimator.start()
}
```

(There are multiple versions of **`ArgbEvaluator`**; import the `android.animation` version.)

Run your animation once again, and you should see the sky fade to a beautiful orange color (Figure 25.6).

Figure 25.6 Sunset color




## Playing Animators Together

If all you need to do is kick off a few animations at the same time, then your job is simple: Call **`start()`** on them all at the same time. They will all animate in sync with one another.

For more sophisticated animation choreography, this will not do the trick. For example, to complete the illusion of a sunset, it would be nice to show the sky turning from orange to a midnight blue after the sun goes down.

This can be done by using an **`AnimatorListener`**, which tells you when an animation completes. So you could write a listener that waits until the end of the first animation, at which time you can start the second night sky animation. But that is a huge hassle and requires a lot of listeners. It is much easier to use an **`AnimatorSet`**.

First, build out the night sky animation and delete your old animation start code.

### Listing 25.13 Building the night animation (`MainActivity.kt`)

```
private fun startAnimation() {
    val sunYStart = binding.sun.top.toFloat()
    val sunYEnd = binding.sky.height.toFloat()

    val heightAnimator = ObjectAnimator
        .ofFloat(binding.sun, "y", sunYStart, sunYEnd)
        .setDuration(3000)
    heightAnimator.interpolator = AccelerateInterpolator()

    val sunsetSkyAnimator = ObjectAnimator
        .ofInt(binding.sky, "backgroundColor", blueSkyColor, sunsetSkyColor)
        .setDuration(3000)
    sunsetSkyAnimator.setEvaluator(ArgbEvaluator())

    val nightSkyAnimator = ObjectAnimator
        .ofInt(binding.sky, "backgroundColor", sunsetSkyColor, nightSkyColor)
        .setDuration(1500)
    nightSkyAnimator.setEvaluator(ArgbEvaluator())

    ~~heightAnimator.start()~~
    ~~sunsetSkyAnimator.start()~~
}
```

And then build and run an **`AnimatorSet`**.

### Listing 25.14 Building an animator set (`MainActivity.kt`)

```
private fun startAnimation() {
    ...
    val nightSkyAnimator = ObjectAnimator
        .ofInt(binding.sky, "backgroundColor", sunsetSkyColor, nightSkyColor)
        .setDuration(1500)
    nightSkyAnimator.setEvaluator(ArgbEvaluator())

    val animatorSet = AnimatorSet()
    animatorSet.play(heightAnimator)
        .with(sunsetSkyAnimator)
        .before(nightSkyAnimator)
    animatorSet.start()
}
```

An **`AnimatorSet`** is nothing more than a set of animations that can be played together. There are a few ways to build one, but the easiest way is to use the **`play(Animator)`** function you are using here.

When you call **`play(Animator)`**, you get an **`AnimatorSet.Builder`**, which allows you to build a chain of instructions. The **`Animator`** passed into **`play(Animator)`** is the "subject" of the chain. So the chain of calls you wrote here could be described as, "Play `heightAnimator` with `sunsetSkyAnimator`; also, play `heightAnimator` before `nightSkyAnimator`." For complicated **`AnimatorSet`**s, you may find it necessary to call **`play(Animator)`** a few times, which is perfectly fine.

Run your app one more time and savor the soothing sunset you have created. Magic.

# For the More Curious: Other Animation APIs

While property animation is the most broadly useful tool in the animation toolbox, it is not the only one. Whether or not you are using them, it is a good idea to know about the other tools out there.

## Legacy animation tools

One set of tools is the classes living in the `android.view.animation` package. This should not be confused with the newer `android.animation` package, which was introduced in Honeycomb.

This is the legacy animation framework, which you should mainly know about so that you can ignore it. If you see the word "animaTION" in the class name instead of "animaTOR", that is a good sign that it is a legacy tool you should ignore.

## Transitions

Android 4.4 introduced the transitions framework, which enables fancy transitions between view hierarchies. For example, you might define a transition that explodes a small view in one activity into a zoomed-in version of that view in another activity.

The basic idea of the transitions framework is that you can define scenes, which represent the state of a view hierarchy at some point, and transitions between those scenes. Scenes can be described in layout XML files, and transitions can be described in animation XML files.

When an activity is already running, as in this chapter, the transitions framework is not that useful. This is where the property animation framework shines. However, the property animation framework is not good at animating a layout as it is coming onscreen.

Take CriminalIntent's crime pictures as an example. If you were to try to implement a "zoom" animation to the zoomed-in dialog of an image, you would have to figure out where the original image was and where the new image would be on the dialog. **`ObjectAnimator`** cannot achieve an effect like that without a lot of work. In that case, you would want to use the transitions framework instead.

## Challenges

For the first challenge, add the ability to *reverse* the sunset after it is completed, so your user can press for a sunset, and then press a second time to get a sunrise. You will need to build another **`AnimatorSet`** to do this – **`AnimatorSet`**s cannot be run in reverse.

For a second challenge, add a continuing animation to the sun. Make it pulsate with heat, or give it a spinning halo of rays. (You can use the **`setRepeatCount(Int)`** function on **`ObjectAnimator`** to make your animation repeat itself.)

Another good challenge would be to have a reflection of the sun in the water.

Your final challenge is to add the ability to press to reverse the sunset scene while it is still happening. So if your user presses the scene while the sun is halfway down, it will go right back up again seamlessly. Likewise, if your user presses the scene while transitioning to night, it will smoothly transition right back to a sunrise.

# 26

# Introduction to Jetpack Compose

Throughout this book, you have built UIs using **`View`** classes and XML layout files. These APIs are provided by the Android OS and are part of the Android framework UI toolkit. Colloquially, we refer to these APIs as "framework views."

Building UIs with framework views has been the standard for making an Android app since the first release of the OS. But in recent times, the framework view system has left much to be desired. For starters, it is built into the OS itself. This means that getting the latest features requires users to update their entire OS, which is not always an option. It also requires developers to bump their apps' minimum SDK level, leaving behind users who are not able to upgrade.

Also, Android's framework UI toolkit is based around ideas like the view hierarchy, view classes that extend from one another, and updating the state of your view manually, line by line. Meanwhile, many front-end UI frameworks have moved on to more modern approaches that make building UIs easier and more streamlined.

To address both of these issues, Google has created a new UI toolkit called *Jetpack Compose*. Jetpack Compose replaces the built-in framework UI toolkit. It is part of the Jetpack suite of libraries, so a UI built with Compose is entirely separate from the Android OS. And because it is separate, you can get updates to Compose just as you would any external library.

Compose is designed in Kotlin (and, in fact, is exclusively available in Kotlin) and is a *declarative* UI framework. The benefits of a declarative UI toolkit will become apparent in the next chapter, when you learn about UI state in Compose. To give you a teaser, though: Compose automatically updates your UI when your application state changes. You *declare* your UI how you want it to appear at all times, and Compose will make it so.

This marks a radical departure from what you are used to. Jetpack Compose does not let you store a reference to any of your UI elements, which means no View Binding, no imperative UI updates – even the time-honored **`findViewById()`** function is not available in Compose. Initially, you might find Compose a bit tricky to reason about, since it requires you to think about your UI differently. However, as you will see in the next chapter, Compose plays well with the modern Android programming paradigms you have used throughout this book like making your UI state observable and updating UI elements reactively.

In our opinion, Jetpack Compose offers a more elegant and concise set of tools to build UIs than what is available in the framework UI toolkit. Google is also emphasizing Jetpack Compose, and we expect that many apps in the future will exclusively use Compose and leave framework views behind.

You may be wondering, "If Compose is the latest and greatest, why bother learning about the framework UI toolkit as well?" We are glad you asked.

Jetpack Compose hit version 1.0 and went stable in the summer of 2021. Since then, the Android development community has begun to transition to Compose – but a transition of this size takes time. If you are just starting your journey as an Android developer, you likely need to be familiar with *both* UI frameworks, as many existing apps, libraries, code snippets, and examples still rely on framework views and will for some time.

Over the next four chapters, we will walk you through the basics of building UIs in Compose. We will not be able to cover every feature Compose offers, but you will end with a solid foundation to build UIs in Compose. You can find more information about Jetpack Compose on its documentation page, `developer.android.com/jetpack/compose/documentation`.

Although the UI code in this project will be different than what you have been seeing, fear not. You are still working on Android – everything you have learned so far will be helpful as you venture into this new territory. Let's get started.

## Creating a Compose Project

To get your feet wet with Compose, you will be creating an app for a pizza delivery service that allows users to customize their pizzas' toppings. This app will be called Coda Pizza. (In music, a *coda* brings a piece to an end, just as your study of Jetpack Compose will conclude your journey through this book.) The finished product will look like Figure 26.1. In this chapter, you will focus on building out the scrollable list of toppings.

Figure 26.1 The finished product



Android Studio offers a template to create an empty Compose app, but you will not use it for Coda Pizza. The Compose templates in Android Studio include a fair amount of code that would just get in your way – plus they are likely to change as Compose evolves.

Instead, we will walk you through setting up a new project and then adding Jetpack Compose. This will allow you to explore Compose in more detail, and the steps involved in setting Compose up will be helpful if you find yourself migrating an existing app away from framework views. (Outside of this book, we encourage you to use the Compose templates for new apps once you have mastered the basics.)

Create a new Android Studio project with the name Coda Pizza and the package name `com.bignerdranch.android.codapizza`. Be sure to use the Empty Activity template, as you have done before. Set the Minimum SDK to 24, and save the project wherever you would like.

With your new project open, your first task is to add Jetpack Compose. This is a multistep process. Compose is enabled in the **`buildFeatures`** block, like View Binding, but you also have to specify the Compose compiler version and add several dependencies. Delve into your `app/build.gradle` file (the one labeled (Module: Coda_Pizza.app)) and make these changes now.

### Listing 26.1 Becoming a composer (`app/build.gradle`)

```
...
android {
    ...
    buildTypes {
        ...
    }

    buildFeatures {
        compose true
    }

    composeOptions {
        kotlinCompilerExtensionVersion '1.1.1'
    }

    compileOptions {
        ...
    }
    ...
}

dependencies {
    ...
    implementation 'androidx.constraintlayout:constraintlayout:2.1.3'

    implementation 'androidx.compose.foundation:foundation:1.1.1'
    implementation 'androidx.compose.runtime:runtime:1.1.1'
    implementation 'androidx.compose.ui:ui:1.1.1'
    implementation 'androidx.compose.ui:ui-tooling:1.1.1'
    implementation 'androidx.compose.material:material:1.1.1'
    implementation 'androidx.activity:activity-compose:1.4.0'

    testImplementation 'junit:junit:4.13.2'
    ...
}
```

Because Compose is built on the latest features in Kotlin, it has specific requirements about which version of Kotlin it supports. Compose 1.1.1 requires Kotlin 1.6.10 – *exactly*. Double-check that your project's `build.gradle` file (the one labeled (Project: Coda_Pizza)) specifies this version, otherwise you will run into build errors.

### Listing 26.2 Matching the Kotlin compiler version (`build.gradle`)

```
plugins {
    id 'com.android.application' version '7.1.2' apply false
    id 'com.android.library' version '7.1.2' apply false
    id 'org.jetbrains.kotlin.android' version '1.6.10' apply false
}
...
```

When you are done, sync your Gradle files to apply these changes.

Next, it is time to delete some code. Coda Pizza will be 100% Compose, so spend a moment to remove the current layout code. Start by removing the call to **`setContentView`** in **`MainActivity`**.

### Listing 26.3 Removing the content view (`MainActivity.kt`)

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

Then, delete the `activity_main.xml` file from your `layout` resources folder.

## Composing Your First UI

With your framework views out of the way, you are ready to write your first Compose UI. The default layout in the Empty Activity project template you have used throughout this book includes an empty **`Activity`** with the text "Hello World!" For your first Compose UI, you will remake this layout without any framework views. (Printing "Hello World!" to the screen is a time-honored coding tradition.)

To populate an activity with a Compose UI, you use a function called **`setContent`**. This function accepts a lambda expression, which is where you have access to Compose UI elements, called *composables*. Use the **`Text`** composable to show the text "Hello World!"

### Listing 26.4 Writing a Compose UI (`MainActivity.kt`)

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            Text(text = "Hello World!")
        }
    }
}
```

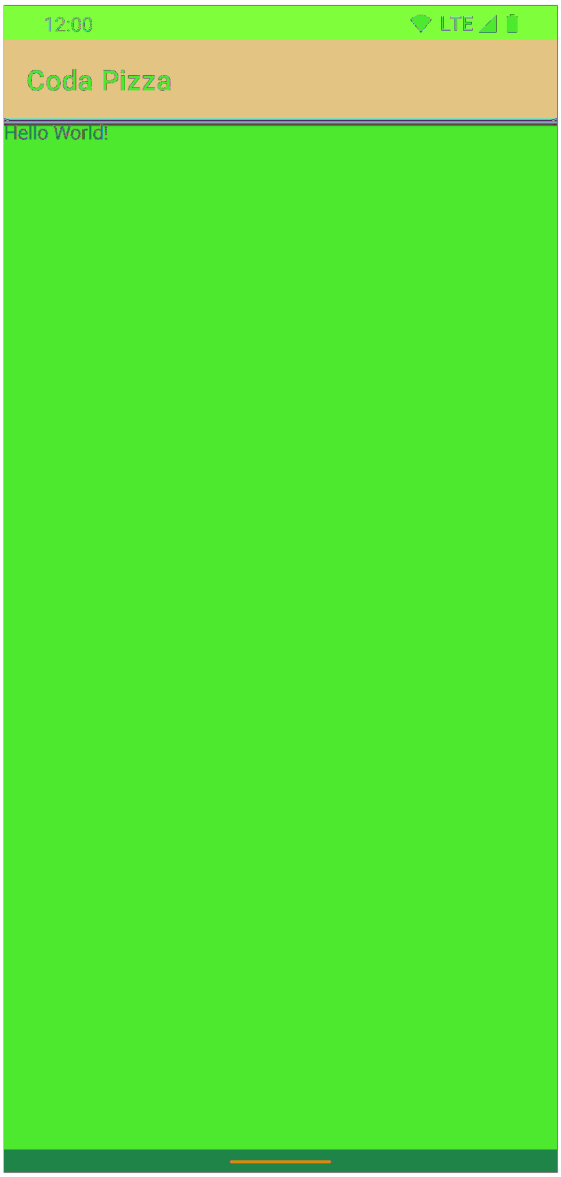
You will need to add two import statements for the code you just entered. For **`setContent`**, add an import for `androidx.activity.compose.setContent`, since you are setting an activity's content. For the **`Text`** composable, add an import for `androidx.compose.material.Text`.

When using Jetpack Compose, you will find that there are many import statements to juggle, and it is not always obvious which you should choose from Android Studio's list of suggestions. In general, your Compose imports will start with the `androidx.compose` package. Also, if you are looking to import a composable function, you can quickly identify them by looking closely at the icon in Android Studio's list of suggested imports. Imports for composables will be marked with  .

Run Coda Pizza and confirm that your text appears as shown in Figure 26.2.

(Coda Pizza is not yet set up to handle dark mode, so be sure to disable the dark theme if you are using it on your test device. If you do not, Coda Pizza will show black text against a black background, making your text appear invisible. After Chapter 29, Coda Pizza will be legible in both modes.)

Figure 26.2 Hello, Compose!




Although this example is rudimentary, notice how concise it is compared to the old fashioned way of building UIs. Putting text onscreen took only a line of code, and all your view code is in Kotlin – no more jumping in and out of XML.

## Layouts in Compose

Time to begin building out the views that Coda Pizza will present to its users. Let's focus on the scrollable list of toppings. First, as you did for your **`RecyclerView`** in CriminalIntent, you will construct a cell that will appear for each topping choice. You will come back to the actual scrolling behavior at the end of this chapter.

The cell will have three elements: the name of the topping, a checkbox indicating whether the topping is on the pizza, and a description of where the topping will appear on the pizza (the left half, the right half, or the whole pizza). Start with the two **`Text`** elements.

In Chapter 2, we told you that flat (non-nested) layouts are faster for the OS to measure and lay out. That is true for framework views, but Compose's efficiency makes it no longer a concern. Composables can be nested to create layouts as complex as you want. For example, to arrange elements vertically from top to bottom, you can place them inside a **`Column`** composable.

**`Column`** is analogous to a **`LinearLayout`** with the `vertical` orientation. It accepts a lambda, and the composables added to the lambda will be arranged from top to bottom. Try it out now.

### Listing 26.5 The **`Column`** composable (`MainActivity.kt`)

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            Text(text = "Hello World!")
            Column {
                Text(
                    text = "Pineapple"
                )

                Text(
                    text = "Whole pizza"
                )
            }
        }
    }
}
```

Run the app after making this change. You should see Pineapple at the top left of the screen, with the text Whole pizza underneath it.

Next, shift your attention to the checkbox. The checkbox will appear to the left of the two text elements. You can accomplish this using a **`Row`**, which behaves like a **`Column`** but lays its content out from left to right (or, if the user's device is set to a right-to-left language, from right to left).

Your **`Row`** will contain the **`Column`** of text plus a **`Checkbox`** composable. You will leave the behavior of the checkbox unimplemented for now, and we will revisit it in the next chapter.

### Listing 26.6 The **`Row`** composable (`MainActivity.kt`)

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            Row {
                Checkbox(
                    checked = true,
                    onCheckedChange = { /* TODO */ }
                )

                Column {
                    Text(
                        text = "Pineapple"
                    )

                    Text(
                        text = "Whole pizza"
                    )
                }
            }
        }
    }
}
```

Run the app again. You should now see that a checked checkbox appears in the top-left corner of the app with the text to its right (Figure 26.3). (If you press the checkbox, its state will not change. This is expected, and we will explain why in the next chapter when we talk about state in Jetpack Compose.)

### Figure 26.3 Rows and columns



## Composable Functions

Before you create the scrollable list of toppings, there is a bit of housekeeping to take care of. Right now, your entire UI is defined in your **`Activity`**. This can get unwieldy quickly, especially for large applications. You can break your UI into smaller chunks by refactoring your Compose code into functions.

Composables' names, like **`Row`** and **`Column`**, begin with capital letters – just like the names of framework views like **`Button`** and **`ImageView`**. But composables are not classes, like views: They are functions.

Remember when we said you cannot get a reference to a Compose UI element or call **`findViewById`** on one? No classes means there is nothing that can be referenced. At runtime, with some help from the Compose compiler, composable functions effectively turn into draw commands.

Compose comes with many prefab composables for basic components like buttons, switches, and text input fields (in addition to the ones you have already seen), but you can also write your own composable functions. Although the built-in composables are often simple, your own composables can combine other composables and be as simple or as complex as you want.

Try writing your own composable now by converting the content inside your **`setContent`** function into its own composable. You can make this change manually, or you can use Android Studio's built-in refactoring tools to make the change automatically. Start by highlighting the code in **`setContent`**'s lambda:

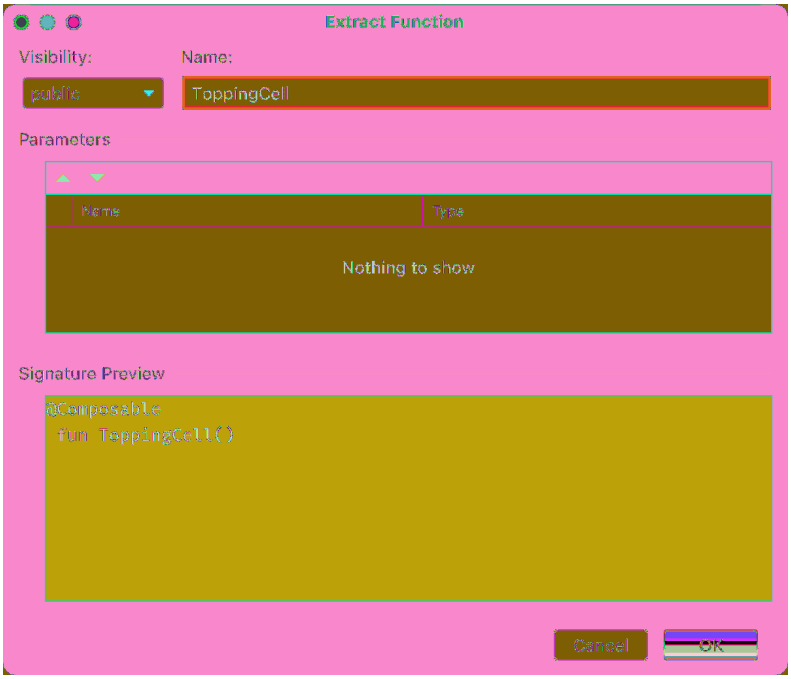
```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            Row {
                Checkbox(
                    checked = true,
                    onCheckedChange = { /* TODO */ }
                )

                Column {
                    Text(
                        text = "Pineapple"
                    )

                    Text(
                        text = "Whole pizza"
                    )
                }
            }
        }
    }
}
```

Next, right-click the code and select Refactor → Function.... The Extract Function dialog will appear. Set the function's visibility to `public` and name the new function **`ToppingCell`** (Figure 26.4).

Figure 26.4 Extracting a composable function




Click OK to perform the refactor. Android Studio will extract the highlighted code into its own function. Your updated code should match the following:

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            ToppingCell()
        }
    }

    @Composable
    fun ToppingCell() {
        Row {
            Checkbox(
                checked = true,
                onCheckedChange = { /* TODO */ }
            )

            Column {
                Text(
                    text = "Pineapple"
                )

                Text(
                    text = "Whole pizza"
                )
            }
        }
    }
}
```

The new **ToppingCell** function looks almost identical to any Kotlin function. In fact, there is only one difference: the `@Composable` annotation. When a function is annotated with `@Composable`, it becomes a composable function. Composable functions can call other composable functions and can add elements onscreen when invoked. Composable functions can also call regular functions, but regular functions cannot call a composable function. (**setContent** is an exception to this rule. It can use composable functions because it is responsible for creating the composition itself.)

Notice that you named your composable **ToppingCell**, not **toppingCell**. As you have seen, it is conventional for the names of composable functions to start with a capital letter, and we recommend following this pattern.

Run the app and confirm that nothing has changed after your refactor. You should still see a checkbox and two lines of text in the top-left corner.

There is one more bit of cleanup to take care of before moving on. Although you have organized your UI into a smaller function, it is still a function on your **MainActivity** class. The composable does not access any information in your activity, so it can be declared in its own file to keep your activity small.

Create a new package called `ui` under the `com.bignerdranch.android.codapizza` package. Inside your new package, create a new file called `ToppingCell.kt`, then copy and paste your **ToppingCell** function into this file.

### Listing 26.7 Putting **ToppingCell** in its own file (`ToppingCell.kt`)

```
@Composable
fun ToppingCell() {
    Row {
        Checkbox(
            checked = true,
            onCheckedChange = { /* TODO */ }
        )
        Column {
            Text(
                text = "Pineapple"
            )

            Text(
                text = "Whole pizza"
            )
        }
    }
}
```

Now you can delete the implementation of **`ToppingCell`** from **`MainActivity`**. You will need to add an import for your relocated **`ToppingCell`** function after making this change. (Remember, it is in the `com.bignerdranch.android.codapizza.ui` package.)

Listing 26.8 Using a composable defined in another file (`MainActivity.kt`)

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            ToppingCell()
        }
    }

    @Composable
    fun ToppingCell() {
        Row {
            Checkbox(
                checked = true,
                onCheckedChange = { /* TODO */ }
            )
            Column {
                Text(
                    text = "Pineapple"
                )

                Text(
                    text = "Whole pizza"
                )
            }
        }
    }
}
```

## Previewing Composables

If you are a fan of Android Studio's design view for XML layouts, you may be wondering if you can preview your Compose layouts the same way. Preview functionality is available for Compose, but Android Studio needs a little help: You must opt in to previews for each composable. Do this now for your **`ToppingCell`** composable by annotating it with the `@Preview` annotation.

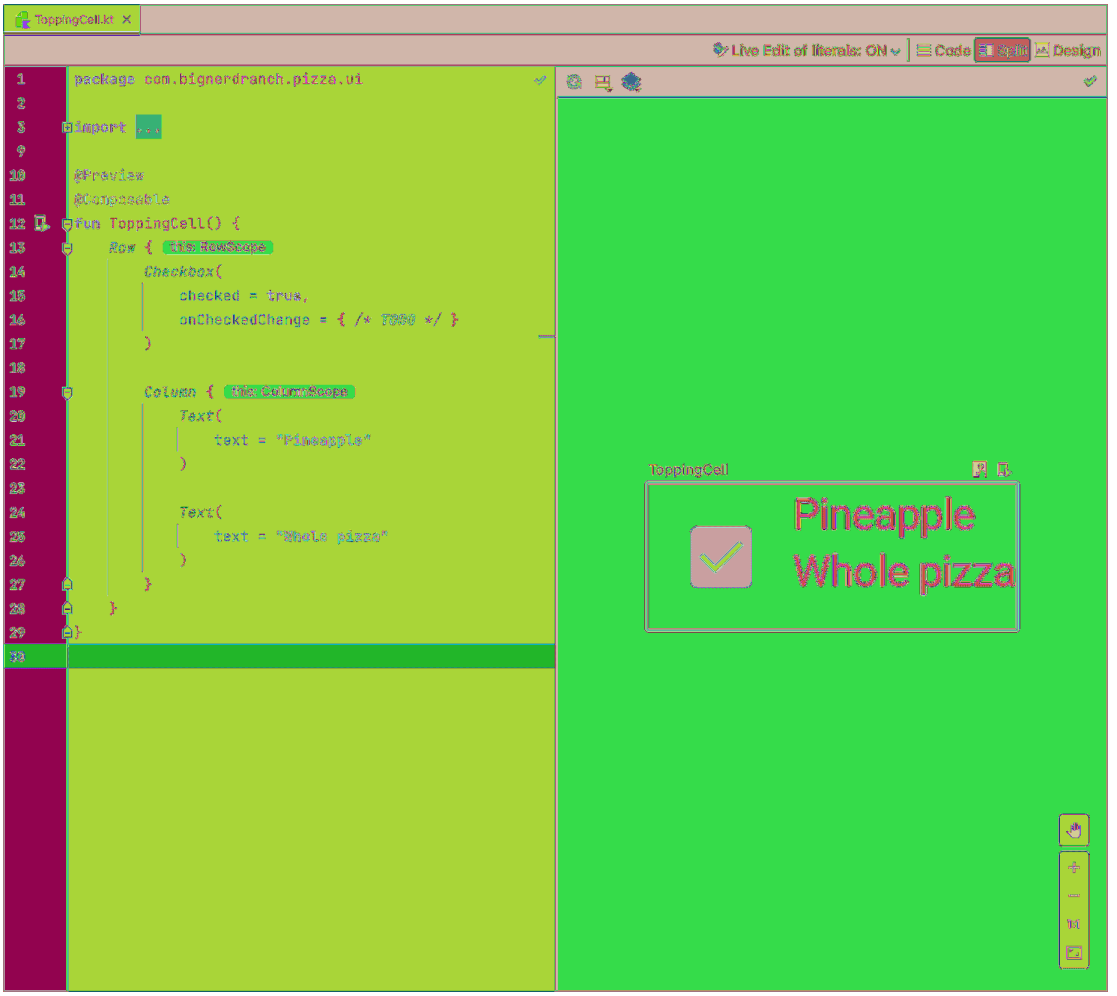
Listing 26.9 Enabling previews on a composable (`ToppingCell.kt`)

```
@Preview
@Composable
fun ToppingCell() {
    ...
}
```

Android Studio uses your project's compiled code to generate previews of composables. This means that your project must be built to show changes you have made in the preview. Build your project now by pressing the  Build icon in Android Studio's toolbar.

When the build finishes, click the Split tab with `ToppingCell.kt` open in the editor. You should see the preview in the right side of the editor (Figure 26.5). (If you do not, check that your project has built successfully, with no errors.)

Figure 26.5 Previewing a composable




As you saw with XML layouts, the preview matches what the user will see when your composable is placed onscreen. Remember that Android Studio needs to recompile your code before it can update the preview, so your changes will not appear instantly the way they did with XML layouts.

The `@Preview` annotation has one very noteworthy limitation: By default, it is not able to show previews for composables that have parameters (unless each parameter has a default value). When this happens, you need to specify the values to use for those arguments. There is a mechanism for doing this with the `@Preview` annotation, but we find it to be cumbersome to set up. Instead, many developers create a separate preview function. This dedicated preview function can pass the desired inputs to the composable being previewed while specifying no inputs of its own.

In just a moment, you are going to add parameters to your **`ToppingCell`** composable. To avoid breaking its preview, preemptively add a separate preview function in the same file:

### Listing 26.10 A dedicated preview composable (`ToppingCell.kt`)

```
@Preview
@Composable
private fun ToppingCellPreview() {
    ToppingCell()
}

~~@Preview~~
@Composable
fun ToppingCell() {
    ...
}
```

This preview composable uses the `private` visibility modifier, allowing you to define the preview without exposing it for use in your production code. Refresh the preview by rebuilding the project or by clicking the  Build Refresh button in the preview window. The preview should look identical except for being labeled ToppingCellPreview instead of ToppingCell.

# Customizing Composables

Coda Pizza is off to a great start. You are ready to start making your composables look just the way you want them to. Previously, you accomplished this using XML attributes. In Compose, function parameters take the place of the attributes that you are accustomed to in XML.

You have already seen a few parameters on the built-in composables you have been using: `text` for the **`Text`** composable and `checked` and `onCheckedChange` for **`Checkbox`**. You are also free to add parameters to your own composables.

## Declaring inputs on a composable function

Think about the **`ToppingCell`** composable. It will need to take in three pieces of information: the name of the topping, the placement of the topping, and what to do when the topping is clicked. Currently, these values are hardcoded – the topping is always pineapple, it is placed on the whole pizza, and nothing happens when you try to edit the topping. This will upset opponents of pineapple on pizza, so it is time to make your toppings more flexible.

The set of toppings and the options for the position of toppings will both have a fixed set of values. Instead of representing these using **`String`**s, enums are a better fit. Also, the hardcoded strings you have been using would not be easy to localize. Jetpack Compose supports loading from your string resources, and it is a good idea to use them.

So start by defining some string resources:

Listing 26.11 Adding string resources (`strings.xml`)

```
<resources>
    <string name="app_name">Coda Pizza</string>

    <string name="placement_none">None</string>
    <string name="placement_left">Left half</string>
    <string name="placement_right">Right half</string>
    <string name="placement_all">Whole pizza</string>

    <string name="topping_basil">Basil</string>
    <string name="topping_mushroom">Mushrooms</string>
    <string name="topping_olive">Olives</string>
    <string name="topping_peppers">Peppers</string>
    <string name="topping_pepperoni">Pepperoni</string>
    <string name="topping_pineapple">Pineapple</string>
</resources>
```

Next, create a new package called `com.bignerdranch.android.codapizza.model` to store the model classes you will use to define and represent a pizza. Create a new file in this package called `ToppingPlacement.kt` and define an enum to specify which part of a pizza a topping is present on.

Give the enum three cases: the whole pizza, the left half of the pizza, and the right half of the pizza. If a topping is not present on the pizza, you can represent that with a `null` value instead.

### Listing 26.12 Specifying topping locations (`ToppingPlacement.kt`)

```
enum class ToppingPlacement(
    @StringRes val label: Int
) {
    Left(R.string.placement_left),
    Right(R.string.placement_right),
    All(R.string.placement_all)
}
```

(The `@StringRes` annotation is not required, but it helps Android Lint verify at compile time that constructor calls provide a valid string resource ID.)

Next, define another enum to specify all the toppings that a customer can add to their pizza. Place this enum in a new file called `Topping.kt` in the `model` package, and populate it as shown:

### Listing 26.13 Specifying toppings (`Topping.kt`)

```
enum class Topping(
    @StringRes val toppingName: Int
) {
    Basil(
        toppingName = R.string.topping_basil
    ),

    Mushroom(
        toppingName = R.string.topping_mushroom
    ),

    Olive(
        toppingName = R.string.topping_olive
    ),

    Peppers(
        toppingName = R.string.topping_peppers
    ),

    Pepperoni(
        toppingName = R.string.topping_pepperoni
    ),

    Pineapple(
        toppingName = R.string.topping_pineapple
    )
}
```

With the models in place, you are ready to add parameters to **`ToppingCell`**. You will add three parameters: a `topping`, a nullable `placement`, and an `onClickTopping` callback. Be sure to provide values for these parameters in your preview composable, otherwise you will get a compiler error.

### Listing 26.14 Adding parameters to a composable (`ToppingCell.kt`)

```
@Preview
@Composable
private fun ToppingCellPreview() {
    ToppingCell(
        topping = Topping.Pepperoni,
        placement = ToppingPlacement.Left,
        onClickTopping = {}
    )
}

@Composable
fun ToppingCell(
    topping: Topping,
    placement: ToppingPlacement?,
    onClickTopping: () -> Unit
) {
    ...
}
```

You will also need to update **`MainActivity`** to provide these arguments when it calls **`ToppingCell`**. Currently, **`MainActivity`** has a compiler error, which will prevent the preview from updating. Fix this now by specifying the required arguments for **`ToppingCell`**. You will revisit the `onClickTopping` callback later. For now, implement it with an empty lambda.

### Listing 26.15 Fixing the compiler error (`MainActivity.kt`)

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            ToppingCell(
                topping = Topping.Pepperoni,
                placement = ToppingPlacement.Left,
                onClickTopping = {}
            )
        }
    }
}
```

Return to `ToppingCell.kt` and build the project to update the preview. Thanks to the changes you just made to **`ToppingCellPreview`**, you might expect the preview to show pepperoni on just the left side of the pizza. However, it still shows pineapple on the whole pizza. This is because you have not yet used the new inputs in your **`ToppingCell`**. Let's change that.

### Resources in Compose

Start with the name of the topping. With the framework views you have seen before, you used the **`Context.getString(Int)`** function to turn a string resource into a **`String`** object you could show onscreen. In Compose, you can accomplish the same thing using the **`stringResource(Int)`** function. Take it for a spin.

Listing 26.16 Using string resources in Compose (`ToppingCell.kt`)

```
...
@Composable
fun ToppingCell(
    topping: Topping,
    placement: ToppingPlacement?,
    onClickTopping: () -> Unit
) {
    Row {
        Checkbox(
            checked = true,
            onCheckedChange = { /* TODO */ }
        )

        Column {
            Text(
                ~~text = "Pineapple"~~
                text = stringResource(topping.toppingName)
            )

            Text(
                text = "Whole pizza"
            )
        }
    }
}
```

Build and refresh the preview. You should see that the topping name changes from the hardcoded Pineapple string to the Pepperoni string from your string resources. (If you wanted, you could also specify a specific string resource instead of accessing it in a variable. The same string lookup you just wrote could also be written as `stringResource(R.string.pepperoni)`, but you instead read it from the `topping` parameter to keep your composable dynamic.)

### Control flow in a composable

Next, shift your attention to the placement text. This is a bit trickier because the `placement` input is nullable. A null value indicates that the topping is not on the pizza. In that case, the second text should not be visible and the **`Checkbox`** should not be checked.

To add this null check, you can wrap the second **`Text`** in an `if` statement. If the topping is present, this `if` statement will execute and add the label to the UI. Otherwise, the `if` statement will be skipped, and only one **`Text`** will end up onscreen.

Go ahead and make this change now. While you are at it, update the `checked` input to **`Checkbox`** to check whether the topping is present on the pizza.

Listing 26.17 `if` statements in a composable (`ToppingCell.kt`)

```
...
@Composable
fun ToppingCell(
    topping: Topping,
    placement: ToppingPlacement?,
    onClickTopping: () -> Unit
) {
    Row {
        Checkbox(
            checked = true,
            checked = (placement != null),
            onCheckedChange = { /* TODO */ }
        )

        Column {
            Text(
                text = stringResource(topping.toppingName)
            )

            if (placement != null) {
                Text(
                    text = "Whole pizza"
                    text = stringResource(placement.label)
                )
            }
        }
    }
}
```

Refresh the preview once more and confirm that the placement text has updated to Left half, matching the value specified in **`ToppingCellPreview`**.

To confirm that your **`ToppingCell`** is appearing as expected when the topping is not present, you will need to update your preview function to specify a `null` input for the `placement`. You could adjust your existing preview to change the `placement` argument, but it can be helpful to preview several versions of a composable at the same time.

Create a second preview function to show what **`ToppingCell`** looks like when the topping is not added to the pizza. Give your two preview functions distinct names to clarify what they are previewing.

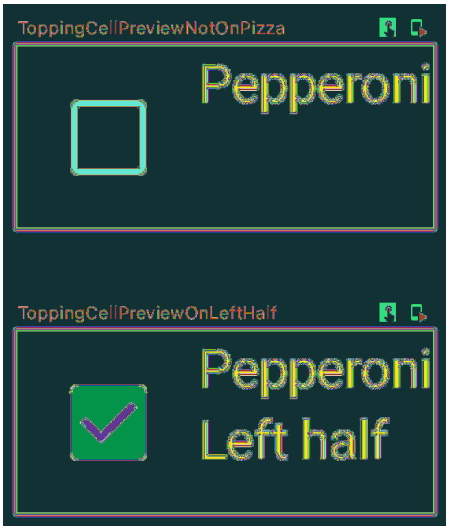
Listing 26.18 Adding another preview (`ToppingCell.kt`)

```
@Preview
@Composable
private fun ToppingCellPreviewNotOnPizza() {
    ToppingCell(
        topping = Topping.Pepperoni,
        placement = null,
        onClickTopping = {}
    )
}

@Preview
@Composable
private fun ToppingCellPreviewOnLeftHalf() {
    ToppingCell(
        topping = Topping.Pepperoni,
        placement = ToppingPlacement.Left,
        onClickTopping = {}
    )
}
...
```

Refresh the preview. You will now see two previews. In the one labeled ToppingCellPreviewNotOnPizza, only the “Pepperoni” label appears in the cell and the checkbox is unchecked (Figure 26.6).

Figure 26.6 No pepperoni, please



You have just observed the effects of control flow inside a composable. Because composables are functions, you can call them however you would like – including conditionally. Here, the **`Text`** composable was not invoked, so it is not drawn onscreen.

You can accomplish something similar with framework views by setting their visibility to `gone`. But with a framework view, the **`View`** itself would still be there, just not contributing to what is drawn onscreen. In Compose, the composable is simply not invoked. It does not exist at all.

`if` statements are not the only control flows you can use in a composable function. Composable functions are, at their core, merely Kotlin functions, so any syntax you can use in other functions can appear in a composable. `when` expressions, `for` loops, and `while` loops are all fair game in your composables, to name a few examples.

## Aligning elements in a row

Take another look at the preview of your topping cell. You may have noticed that in the unselected state, it looks a bit awkward because the checkbox and the text are not vertically aligned. Worry not, there is another parameter you can specify to beautify this layout.

The **`Row`** and **`Column`** composables specify their own parameters that you can use to adjust the layout of their children. For a **`Row`**, you can use the `Alignment` parameter to adjust how its children are positioned vertically. (A **`Column`**'s `Alignment` will adjust the horizontal positioning of its children.)

By default, **`Row`**'s vertical alignment is set to `Alignment.Top`, meaning that the top of each composable will be at the top of the row. To center all items in the composable, set its alignment to `Alignment.CenterVertically`.

### Listing 26.19 Specifying alignment (`ToppingCell.kt`)

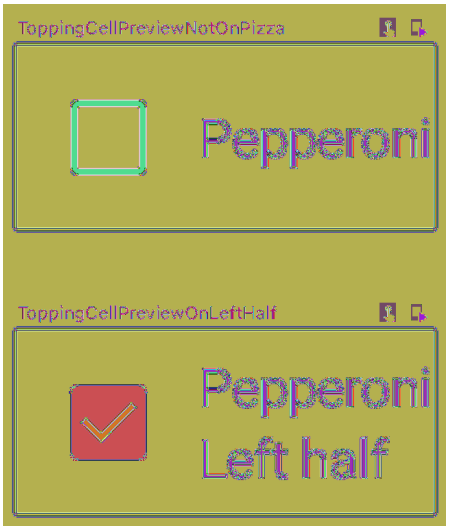
```
...
@Composable
fun ToppingCell(
    topping: Topping,
    placement: ToppingPlacement?,
    onClickTopping: () -> Unit
) {
    Row(
        verticalAlignment = Alignment.CenterVertically
    ) {
        ...
    }
}
```

Be sure to import `androidx.compose.ui.Alignment` from the options provided.

By the way: Do not confuse the `Alignment` parameter with the `Arrangement` parameter, which specifies how the extra horizontal space of a **`Row`** (or vertical space, for a **`Column`**) should be placed relative to its children.

Refresh the preview again and confirm that the topping name and checkbox are vertically aligned (Figure 26.7).

Figure 26.7 Aligning the contents of a row



## Specifying text styles

Composable parameters are useful for arranging your content and setting values to display. They also serve an important role in styling your UI.

In Chapter 9, you set the `android:textAppearance` attribute to `?attr/textAppearanceHeadline5` to apply built-in styling to text elements in XML. In Compose, you can accomplish the same thing by setting the `style` parameter of the **`Text`** composable. Like the framework toolkit, Compose also has built-in text styles accessible through the **`MaterialTheme`** object. Take them for a spin now, applying the `body1` style to the name of the topping and `body2` to the placement of the topping.

When entering this code, be sure to choose the **`MaterialTheme`** object when prompted, not the **`MaterialTheme`** composable function. Their imports are the same, so no need to worry if you choose the wrong one initially – just note that Android Studio will autocomplete different code. You will see how the **`MaterialTheme`** function works in Chapter 29.

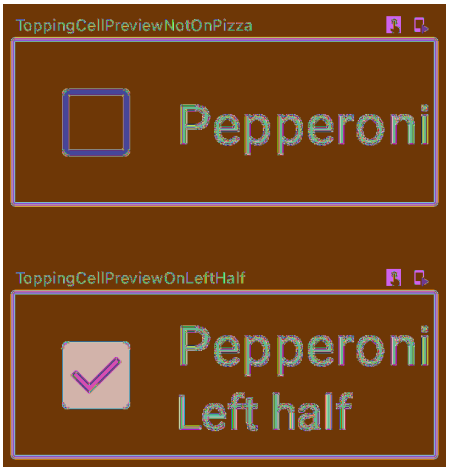
Listing 26.20 Setting text styles (`ToppingCell.kt`)

```
...
@Composable
fun ToppingCell(
    ...
) {
    Row(
        verticalAlignment = Alignment.CenterVertically
    ) {
        ...
        Column {
            Text(
                text = stringResource(topping.toppingName),
                style = MaterialTheme.typography.body1
            )

            if (placement != null) {
                Text(
                    text = stringResource(placement.label),
                    style = MaterialTheme.typography.body2
                )
            }
        }
    }
}
```

Update the previews by pressing the Build & Refresh label in the banner that says The preview is out of date or by building the project. You will see that the first line of text is larger than the second (Figure 26.8). The difference is subtle, but we promise – they are different sizes.

Figure 26.8 Text with style



# The Compose Modifier

What about attributes like background color, margins, padding, and click listeners? In the framework view system, these attributes are inherited, making them accessible on every subclass of **`View`**. But composable functions do not have the luxury of inheritance.

Instead, Compose defines a separate type called **`Modifier`** where it defines general-purpose customizations that can be set on any composable. Modifiers can be chained and combined as desired. Between modifiers and a composable's function parameters, you can perform all the customizations you have used so far with framework views.

To modify a composable, pass a **`Modifier`** into the composable's `modifier` parameter. You can obtain an empty modifier to build on by referencing the **`Modifier`** object first. Then, you can chain a sequence of modifiers together to create a final **`Modifier`** object to set on the composable.

## The padding modifier

Start by adding padding to the entire cell with the **`padding`** modifier. Set the vertical padding to 4dp and the horizontal padding to 16dp.

Listing 26.21 Adding padding (`ToppingCell.kt`)

```
...
@Composable
fun ToppingCell(
    topping: Topping,
    placement: ToppingPlacement?,
    onClickTopping: () -> Unit
) {
    Row(
        verticalAlignment = Alignment.CenterVertically,
        modifier = Modifier
            .padding(vertical = 4.dp, horizontal = 16.dp)
    ) {
        ...
    }
}
```

When prompted, be sure to choose the import for `androidx.compose.ui.Modifier`.

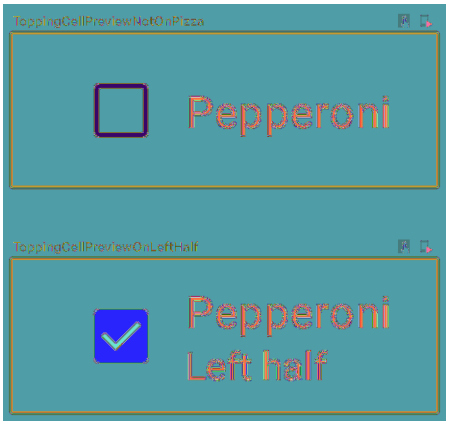
In case you are curious, there are a few other ways of specifying padding amounts. Some commonly used overloads allow you to specify the same padding amount on all sides (`Modifier.padding(all = 16.dp)`), or for all four sides independently (`Modifier.padding(top = 4.dp, bottom = 4.dp, start = 16.dp, end = 16.dp)`).

Recall from Chapter 11 that `dp` units are ideal for specifying margins and padding. Composables and modifiers specify which unit they expect a dimension to be specified in. The **`.dp`** extension property you are using returns a **`Dp`** object, which adds additional type safety to make sure you are using the correct units. There is also an **`.sp`** extension to specify an **`Sp`** value for text sizes.

Unlike framework views, these units are not interchangeable – if a Compose API needs a text size, it requests an **`Sp`** instance specifically. This also means that if you do not specify a unit, you will see a compiler error, because **`Int`** cannot be converted to **`Dp`** or **`Sp`** automatically.

Build and refresh your preview. You should see some additional spacing around your composable (Figure 26.9).

Figure 26.9 Room to breathe




By the way, you may also see borders around elements in your composable previews. These represent the bounds of objects in your composables, and can help you visualize how and why items are being positioned as they are. You can hover over a preview to reveal these bounds and click one of the items to navigate to the composable corresponding to that element.

## Chaining modifiers and modifier ordering

If you want to further customize the appearance of your composables, you can chain modifiers. But beware: The order of modifiers matters. To see why, try adding a background to the **`ToppingCell`**. Place the **`background`** modifier after the **`padding`**.

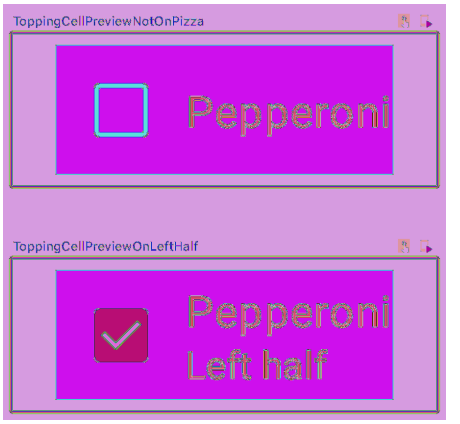
Listing 26.22 Adding a background (`ToppingCell.kt`)

```
...
@Composable
fun ToppingCell(
    topping: Topping,
    placement: ToppingPlacement?,
    onClickTopping: () -> Unit
) {
    Row(
        verticalAlignment = Alignment.CenterVertically,
        modifier = Modifier
            .padding(vertical = 4.dp, horizontal = 16.dp)
            .background(Color.Cyan)
    ) {
        ...
    }
}
```

When using the **`Color`** class, be sure to choose the import for `androidx.compose.ui.graphics`. Compose specifies its own class for colors (much like it does for **`Dp`** and **`Sp`**) and includes constants for a few colors as a convenience.

Where do you think the background will appear? Build and refresh your composable preview to see for yourself (Figure 26.10). Does this match your expectation?

Figure 26.10 Padding the background

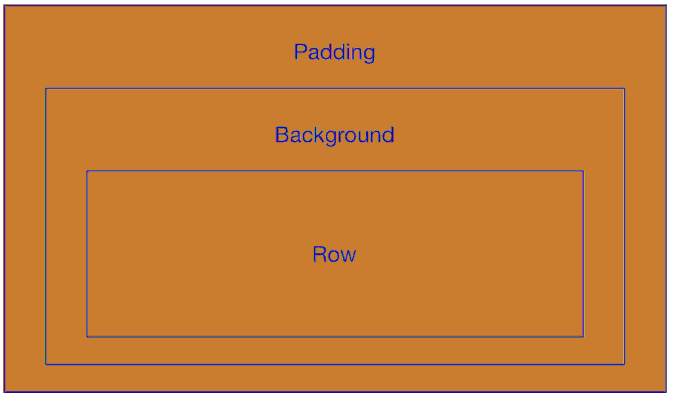



With this code, the background appeared *inside* the padding. Why?

As we said: In Compose, the order of modifiers matters. Modifiers are invoked from top to bottom. Each makes its contribution to the appearance of the composable, and then the next modifier is applied *inside* it. Once all modifiers have been applied, the content of the composable is placed inside the final modifier.

Looking at your current code, this means that the padding is added first, and then the background is added inside the padding. Finally, the **`Row`** and its contents are placed inside the background. Figure 26.11 illustrates how Compose is treating your **`Row`** and the two modifiers.

Figure 26.11 How Compose sees your modifiers



Now try moving the background modifier so that it comes before the padding modifier.

Listing 26.23 Reordering the background modifier (`ToppingCell.kt`)

```
...
@Composable
fun ToppingCell(
    topping: Topping,
    placement: ToppingPlacement?,
    onClickTopping: () -> Unit
) {
    Row(
        verticalAlignment = Alignment.CenterVertically,
        modifier = Modifier
            .background(Color.Cyan)
            .padding(vertical = 4.dp, horizontal = 16.dp)
            .background(Color.Cyan)
    ) {
        ...
    }
}
```

Build and refresh the preview. You will now see that the background fills the entire composable – padding included – and the rest of the content remains inside the padding (Figure 26.12).

Figure 26.12 Backgrounds with padding



The padding is placed inside the background now because the background is added first and the padding is added second (Figure 26.13).

Figure 26.13 How Compose sees your reordered modifiers




Sometimes, the order of your modifiers does not matter. For example, if you had several instances of the **`padding`** modifier to specify the padding amounts for the top, bottom, and sides of a composable, it would not matter what order you declared them in. But for many combinations of modifiers, you need to be careful about ordering.

If you ever find that one of your composables is not appearing as expected, think about the order of your modifiers and make sure they are not incorrectly affecting one another.

Remove the background, as it is a bit garish and will not be part of the final UI.

Listing 26.24 Removing the background (`ToppingCell.kt`)

```
...
@Composable
fun ToppingCell(
    topping: Topping,
    placement: ToppingPlacement?,
    onClickTopping: () -> Unit
) {
    Row(
        verticalAlignment = Alignment.CenterVertically,
        modifier = Modifier
            .background(Color.Cyan)
            .padding(vertical = 4.dp, horizontal = 16.dp)
    ) {
        ...
    }
}
```

## The clickable modifier

Another crucial modifier is **`clickable`**, which is analogous to the venerable **`setOnClickListener`** method. The **`clickable`** modifier makes a composable clickable, and it accepts a lambda expression to define what to do when the view is pressed. Try it out now to make pressing the **`Row`** invoke the `onClickTopping` callback.

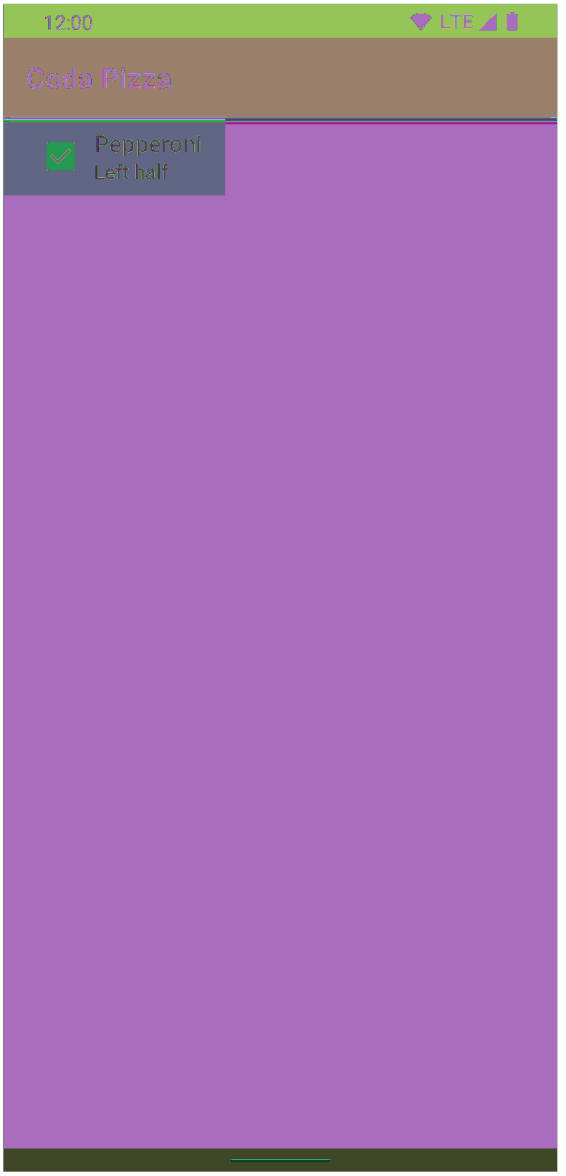
Be sure to add the **`clickable`** modifier as the *first* modifier. You want the entire composable (padding included) to be clickable. Also, besides defining click behavior, the **`clickable`** modifier darkens the background of the composable when it is pressed to indicate that it is being selected. You want this effect to extend through the padding, which is another reason to put the modifier first.

Listing 26.25 Making a composable clickable (`ToppingCell.kt`)

```
...
@Composable
fun ToppingCell(
    topping: Topping,
    placement: ToppingPlacement?,
    onClickTopping: () -> Unit
) {
    Row(
        verticalAlignment = Alignment.CenterVertically,
        modifier = Modifier
            .clickable { onClickTopping() }
            .padding(vertical = 4.dp, horizontal = 16.dp)
    ) {
        ...
    }
}
```

Run Coda Pizza in an emulator. You will see the pepperoni topping displayed in the top-left corner of your activity, matching the preview. Try clicking the topping. (Be sure to click near the text Pepperoni.) You have not specified any behavior to take place after the click, but you will see the background darken, indicating that the click was recognized (Figure 26.14).

Figure 26.14 Interacting with a clickable composable



## Sizing composables

Currently, the only clickable area is near the label of the topping. If you try to click to the right of the topping cell, nothing will happen. (Try it for yourself.) This probably does not match your users' expectations – you should be able to click anywhere along the width of the screen to interact with the cell.

The reason that the clickable area does not fill the width of the screen is that your **`ToppingCell`** composable is only taking up as much space as it needs to to display its content. Effectively, its dimensions are implicitly set to wrap its content.

One way to make your composable consume all the available width is to make your **`Column`** take up all the remaining width in its container. You can do this with the **`weight`** modifier.

The **`weight`** modifier is a bit special in that it can only be used when your composable is placed inside another composable that supports weights – like **`Row`** and **`Column`**. The weight modifier behaves the same as the `layout_weight` attribute on a **`LinearLayout`**: Any extra space will be divvied up proportionally to views in the layout based on their weight. If only one of the composables specifies a weight, all the extra space will go to that composable.

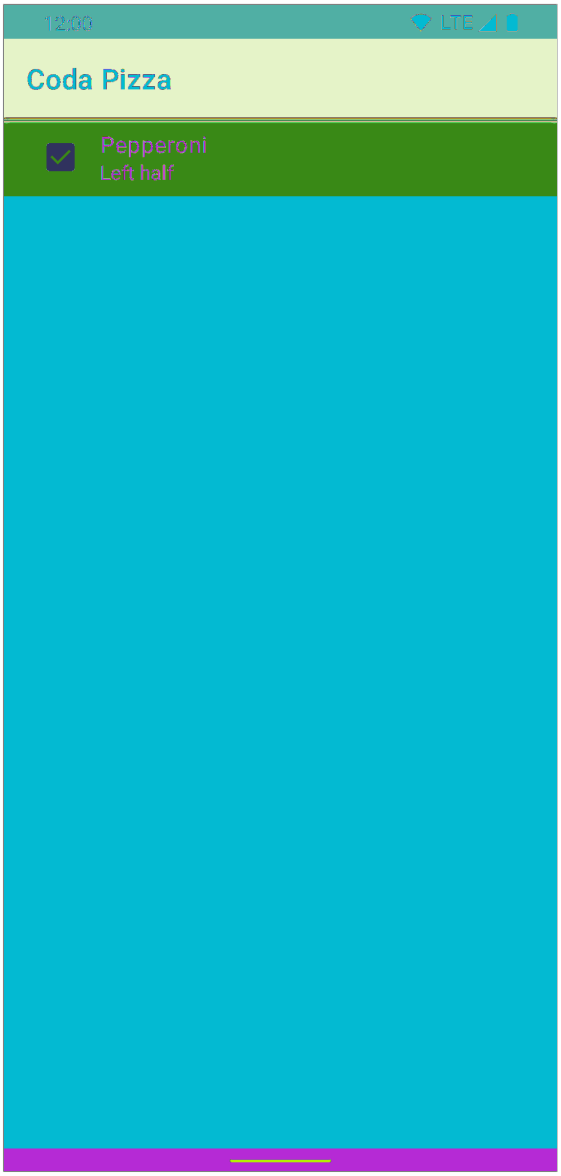
Try it out now. While you are adding a modifier to your **`Column`**, include some padding, which will give your checkbox and text some more breathing room.

### Listing 26.26 Using the `weight` modifier (`ToppingCell.kt`)

```
...
@Composable
fun ToppingCell(
    topping: Topping,
    placement: ToppingPlacement?,
    onClickTopping: () -> Unit
) {
    Row(
        ...
    ) {
        ...
        Column(
            modifier = Modifier.weight(1f, fill = true)
                .padding(start = 4.dp)
        ) {
            ...
        }
    }
}
```

Rerun Coda Pizza in the emulator to see the changes. Press the right half of the screen next to the Pepperoni text. You should see that the touch indication appears for the topping cell, even though you are clicking the empty space next to it. You should also see that the touch indication fills the full width of the screen (Figure 26.15).

Figure 26.15 Interacting with a weighted clickable composable




We have just scratched the surface of what is available with the **`Modifier`** API. For example, you can more explicitly tell a composable how much space to take up with modifiers like **`wrapContentHeight`** and **`fillParentWidth`**. You can also specify dimension with modifiers like **`size`**, and you can get more creative about how to constrain a composable's size with modifiers like **`aspectRatio`** and **`sizeIn`**.

You can find a list of all the built-in modifiers at `developer.android.com/jetpack/compose/modifiers-list`. We encourage you to experiment and combine modifiers to build more complex UIs. You will likely find that modifiers are more predictable, flexible, and concise than what is available in the framework view classes.

## Specifying a modifier parameter

Modifiers are a crucial part of customizing a composable, and they allow you to specify many common customizations for your UI elements. They are so important, in fact, that we recommend that *every* composable UI element that you define accept an optional **`Modifier`** input.

Even if you do not think you need to specify any modifiers on a composable, it is better to have the option readily available than to have to add it later if you change your mind. For **`ToppingCell`**, you may decide later that you want to add a background, change its padding, or set a size for the composable. In fact, later in this chapter, you will need to tell your **`ToppingCell`** how wide it should be.

Currently, the only way to change these attributes is to modify **`ToppingCell`** itself. But any changes made to the composable directly will appear everywhere you use **`ToppingCell`** in your app, which is not ideal. What if **`ToppingCell`** is used in multiple places and needs to be a different size in each place? To open the door to future customizations, you will add a **`Modifier`** parameter to **`ToppingCell`**.

To avoid requiring a **`Modifier`** instance for every single usage, give this parameter a default value of `Modifier` – the `Modifier` object that you have been building off of so far that represents an empty set of modifications. The official convention for the `modifier` parameter is to place it after your required parameters and before any other optional parameters.

To use the `modifier` parameter, pass it to your outer composable – your **`Row`**, in this case. Make these changes now (and be sure to change the capitalized `Modifier` to the lowercase `modifier`):

Listing 26.27 Allowing modifications (`ToppingCell.kt`)

```
...
@Composable
fun ToppingCell(
    topping: Topping,
    placement: ToppingPlacement?,
    modifier: Modifier = Modifier,
    onClickTopping: () -> Unit
) {
    Row(
        verticalAlignment = Alignment.CenterVertically,
        modifier = Modifier modifier
            .clickable { onClickTopping() }
            .padding(vertical = 4.dp, horizontal = 16.dp)
    ) {
        ...
    }
}
```

And that is it! We will revisit your lingering `TODO` in the next chapter when we discuss state, but for now you have finished implementing **`ToppingCell`**. It is ready to be used in your application – say, in a scrollable list.

# Building Screens with Composables

You have seen a number of composables up to this point, with a range of complexity. On the simple end, you have seen atomic components like **`Row`** and **`Text`**. And you have built **`ToppingCell`**, a more complex composable that is built on top of other composables.

It is also possible to create a composable that renders the entire screen. In fact, the **`setContent`** function you called in **`MainActivity`** is a lambda that does just that. Because there are no limitations about what a single composable can do, you do not need another component like a **`Fragment`** to perform navigation. (You can, however, enlist the help of the AndroidX Navigation library, which has a Jetpack Compose flavor.)

Under Jetpack Compose, composables of all sizes are *the* building blocks of your UIs.

Right now, Coda Pizza only displays a single topping. Eventually, you will display several toppings, but you will not want to be constantly opening your activity's code to modify its UI. Instead, you can extract its content into a separate file. The result of this change will be that your **`MainActivity`** will call a single content composable, leaving the activity itself sparse with code.

To get started, define a new file called `PizzaBuilderScreen.kt` in the `com.bignerdranch.android.codapizza.ui` package. Define a new composable function called **`PizzaBuilderScreen`** in this file. **`PizzaBuilderScreen`** will be a composable that draws *all* of the main content inside the activity. If your code had navigation or other logic, you could place it in this composable.

Remember to give your new composable a **`Modifier`** parameter. Also, add the `@Preview` annotation to add a quick preview to the function. (You will not add any arguments to this function, so a separate preview function is not necessary.)

### Listing 26.28 Defining a screen (`PizzaBuilderScreen.kt`)

```
@Preview
@Composable
fun PizzaBuilderScreen(
    modifier: Modifier = Modifier
) {

}
```

Here, we are using the convention of ending the composable's name with "Screen" to indicate that it fills the entire viewport of the window and represents a distinct portion of the app's UI. Apps can have many screens if they need to display several different UIs and navigate between them.

Your **`PizzaBuilderScreen`** will display several elements by the time Coda Pizza is complete: the app bar, the pizza customization preview, the list of toppings, and the PLACE ORDER button. For now, you will focus on two of these elements: the list of toppings and the PLACE ORDER button.

Before adding any new composables, take a brief detour to your `strings.xml` file to add the string resource you will use for the PLACE ORDER button.

### Listing 26.29 The "place order" label (`strings.xml`)

```
<resources>
    <string name="app_name">Coda Pizza</string>

    <string name="place_order_button">Place Order</string>

    <string name="placement_none">None</string>
    ...
</resources>
```

Next, create two new private composables inside `PizzaBuilderScreen.kt`: one for the toppings list and one for the PLACE ORDER button. For the toppings list composable, call **`ToppingCell`** once for the time being. You will implement the scrolling list behavior in the next section. For the PLACE ORDER button, use the **`Button`** composable.

The **`Button`** composable takes in two required inputs: an `onClick` callback and a set of composable children to place inside the button. If you wanted to, you could add an icon or any other composables to spice up your button, but for now you will stick to just the **`Text`** composable.

It is conventional for button labels on Android to appear in all caps. This happens by default when using the framework **`Button`** view, but does not happen automatically in Compose. To respect this convention, manually capitalize your string using the **`toUpperCase`** function.

### Listing 26.30 Defining content to put onscreen (`PizzaBuilderScreen.kt`)

```
@Preview
@Composable
fun PizzaBuilderScreen(
    modifier: Modifier = Modifier
) {

}

@Composable
private fun ToppingsList(
    modifier: Modifier = Modifier
) {
    ToppingCell(
        topping = Topping.Pepperoni,
        placement = ToppingPlacement.Left,
        onClickTopping = {},
        modifier = modifier
    )
}

@Composable
private fun OrderButton(
    modifier: Modifier = Modifier
) {
    Button(
        modifier = modifier,
        onClick = {
            // TODO
        }
    ) {
        Text(
            text = stringResource(R.string.place_order_button)
                .toUpperCase(Locale.current)
        )
    }
}
```

If prompted, be sure to choose the import for `androidx.compose.material.Button` as well as the imports for `androidx.compose.ui.text.toUpperCase` and `androidx.compose.ui.text.intl.Locale`.

Now you can place these composables in your **PizzaBuilderScreen** to set their position onscreen. Add a **Column** to **PizzaBuilderScreen** to place the **ToppingsList** at the top of the screen and the **OrderButton** at the bottom of the screen. The **ToppingsList** should fill all of the available height, so set its weight to 1 via its modifier. Also, make the **OrderButton** take up the full width of the screen, with 16dp of padding around it.

### Listing 26.31 Placing content in the **PizzaBuilderScreen** (PizzaBuilderScreen.kt)

```
@Preview
@Composable
fun PizzaBuilderScreen(
    modifier: Modifier = Modifier
) {
    Column(
        modifier = modifier
    ) {
        ToppingsList(
            modifier = Modifier
                .fillMaxWidth()
                .weight(1f, fill = true)
        )

        OrderButton(
            modifier = Modifier
                .fillMaxWidth()
                .padding(16.dp)
        )
    }
}
...
```

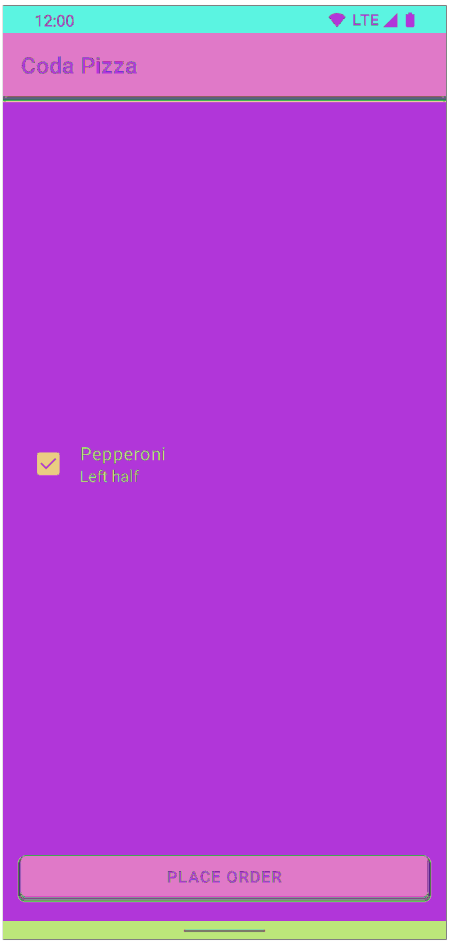
With **PizzaBuilderScreen** ready, you can now update your **MainActivity**'s **setContent** block to delegate to this function instead of creating the UI itself.

### Listing 26.32 Using the **PizzaBuilderScreen** composable (MainActivity.kt)

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            ToppingCell(
                topping = Topping.Pepperoni,
                placement = ToppingPlacement.Left
            )
            PizzaBuilderScreen()
        }
    }
}
```

Run Coda Pizza. The app should look like Figure 26.16, with the PLACE ORDER at the bottom of the screen and the topping information in the center of the screen. The topping cell is centered on the screen because you have set it to fill the height of the screen and it is arranging its content to be centered within its bounds. Although the centering behavior is a bit awkward, your **ToppingsList** composable is being correctly placed to take up the leftover height of the screen for when you give it a final implementation.

Figure 26.16 **`PizzaBuilderScreen`** in action




## Scrollable Lists with LazyColumn

The last step in the major scaffolding for Coda Pizza is to turn your **`ToppingsList`** composable into, well, a *list*. Right now it displays a single topping, but you want it to show all the available toppings in Coda Pizza's menu. Previously, you would use **`RecyclerView`** to achieve this goal, which generally involves a ritual of creating an adapter, view holder, and layout manager for even the most basic of lists.

In Compose, scrollable lists of items are created using **`LazyColumn`** (or **`LazyRow`**, if you want to scroll horizontally instead of vertically). At a very high level, **`LazyColumn`** behaves like **`RecyclerView`**: It only does work for the composables that will be drawn onscreen, and it can scroll through enormous lists with high efficiency. Unlike **`RecyclerView`**, a **`LazyColumn`** can be created with just a few lines of code.

Because you do not have the overhead of view objects or the ability to store references to composables, there is no **`ViewHolder`** class to create. And because you have the power of Compose's layout engine, an adapter is not necessary. The adapter's role is to turn indices into views and to recycle those views with new information, but Compose can spin up and tear down composables so efficiently on its own that **`LazyColumn`** only needs to know what to display at a given position. It will take care of the rest.

**`LazyColumn`** has one required parameter: a lambda expression to specify the contents of the list. Beware, though – unlike most of the other lambdas you have seen in this chapter, the lambda you pass to **`LazyColumn`** is *not* a `@Composable` function, meaning that you cannot directly add composable content to the list.

Instead, you add elements to the list by calling functions like **`item`**, **`items`**, or **`itemsIndexed`** inside **`LazyColumn`**'s lambda. Each of these builder functions accepts its own lambda to create a composable that defines what will be drawn for its position or positions in the list. You can add as many or as few items as you want, and you can easily combine datasets if desired.

For Coda Pizza, you will use a **`LazyColumn`** to display all the values of the **`Topping`** enum. Inside the **`LazyColumn`**, use the **`items`** function and pass a list of the **`Topping`** values to specify that they should appear in the list. In the lambda for the **`items`** builder, take the `topping` passed to the lambda and use it to create a row for that topping by calling your **`ToppingCell`**.

Make this change in `PizzaBuilderScreen.kt` now, deleting the **`ToppingCell`** you added as a placeholder.

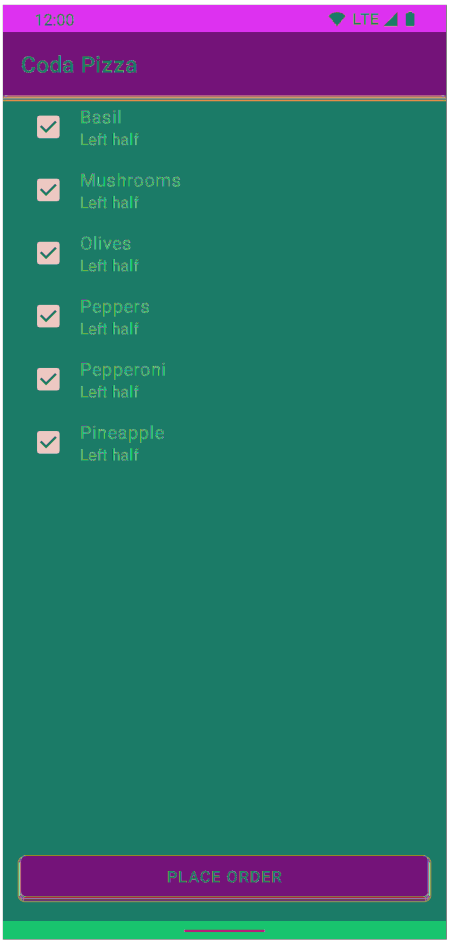
### Listing 26.33 Using **`LazyColumn`** to show the list of toppings (`PizzaBuilderScreen.kt`)

```
...
@Composable
private fun ToppingsList(
    modifier: Modifier = Modifier
) {
    ToppingCell(
        topping = Topping.Pepperoni,
        placement = ToppingPlacement.Left,
        onClickTopping = {},
        modifier = modifier
    )
    LazyColumn(
        modifier = modifier
    ) {
        items(Topping.values()) { topping ->
            ToppingCell(
                topping = topping,
                placement = ToppingPlacement.Left,
                onClickTopping = {
                    // TODO
                }
            )
        }
    }
}
...
```

When entering this code, be sure to import the **`items`** function, if Android Studio does not do so automatically. The import for this function is `androidx.compose.foundation.lazy.items`. You may see an error about a type mismatch if you do not have this import statement.

Run Coda Pizza. You will see all the toppings listed in the order that they are declared in your **`Topping`** enum. They will all be set to appear on the left half of the pizza (Figure 26.17).

Figure 26.17 **`LazyColumn`** in action




The list still needs a bit of work – it is not possible to change the placement of a topping, for example – but spend a moment to marvel at your code. These few lines of code are all it takes to implement the **`ToppingsList`** composable, and your code can easily be adapted to add more content to the list or to change the appearance of its contents. Achieving this level of conciseness with **`RecyclerView`** is simply not possible. This is a testament to how Jetpack Compose makes it easier to build Android apps.

You have accomplished quite a bit in this chapter, from making your way through the fundamentals of Compose's layout system through giving Coda Pizza a solid foundation to build on as you continue to explore Jetpack Compose. In the next chapter, you will make your UI elements interactive, update your composables to react to user input, and explore how Jetpack Compose handles application state.

## For the More Curious: Live Literals

In addition to previews, Jetpack Compose has several tricks up its sleeve to let you quickly iterate on UI designs. One of these is a feature called *live literals*. When live literals are enabled and you run your Compose app through Android Studio, any changes you make to hardcoded values ("literals") in your Compose UI will automatically be pushed to the app as it is running. Your UI will refresh with new values as they are typed, letting you preview changes instantly, without recompiling.

Live literals only work for simple hardcoded values like **`Int`**, **`Boolean`**, and **`String`**. They also must be enabled in the IDE. To check if you have live literals enabled, open Android Studio's preferences with Android Studio → Preferences.... The option for live literals is under the Editor section in the left half of the preferences window, on a page called Live Edit of literals. Navigate to this page and ensure Enable Live Edit of literals is checked to turn on live literals. Then re-launch your app to enable the feature.

Try out live literals by changing your padding values while Coda Pizza is running. Use larger and smaller values and watch as your content instantly shifts to match the new measurements you define.

Any other code you modify – like adding or removing modifiers entirely – will not update your UI until you rebuild and run Coda Pizza again; only your literals get updated automatically. Despite this limitation, the instantaneous nature of these updates makes live literals a great tool for putting the finishing touches on your UI's appearance.

When you are finished experimenting with live literals, make sure to revert any changes you made to Coda Pizza before proceeding to the next chapter.

# 27

# UI State in Jetpack Compose

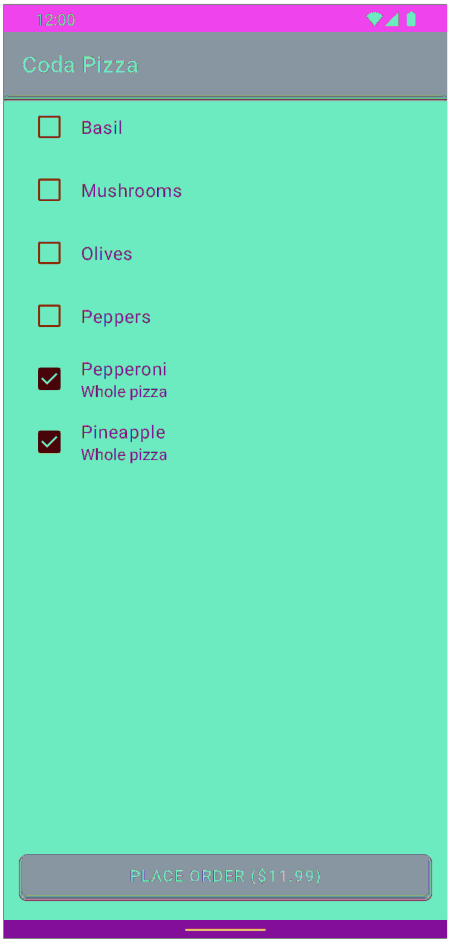
Coda Pizza is off to a good start. In the last chapter, you built out a scrollable list, created a composable to act as the cell for toppings, and set up the screen's layout, including an order button at the bottom of the page. But your app is missing something: Users cannot interact with it (other than scrolling the list of toppings). If users press a checkbox, it does not change from checked to unchecked or vice versa.

This is not what you are used to with framework views. Think back to CriminalIntent, where you used a **`CheckBox`** view to track whether a crime was solved. As soon as you added the checkbox to your UI, it responded to click events by toggling its state, with no additional code needed.

In this chapter, you will learn why your current code does not respond to user inputs, and you will see how to incorporate UI state into your composables. By the end of this chapter, Coda Pizza will let users place and remove toppings on their pizzas. Pressing either the checkbox or the topping's cell will toggle whether the topping is on the pizza. You will also update the PLACE ORDER button to include the price for the pizza based on the number of toppings.

When you finish with this chapter, Coda Pizza will look like Figure 27.1.

Figure 27.1 Coda Pizza, now with state




In the next chapter, you will go a step further and display a dialog to ask the user where they want their topping to be placed.

## Philosophies of State

Let's start by discussing how framework views approach state. The apps you worked on earlier in this book actually had two representations of UI state. The first was built by you, generally defined with data classes and managed by **`ViewModel`**s. This kind of state is often called *application state* or *app state*, because this state controls the behavior of your app as a whole and is how your app sees the world. Your Compose app will also have app state, and it will be defined similarly.

The second representation of UI state in framework views lived inside the **`View`**s themselves. Think about the framework **`CheckBox`** and **`EditText`** views. When the user presses a **`CheckBox`** or enters new text in an **`EditText`**, that input immediately updates the UI, whether you want it to or not.

The fragmentation of state with framework views means that one of the important responsibilities of your UI code so far has been keeping your app state and **`View`** state in sync. If you are using a unidirectional data flow architecture, for example, your app will take in events, update its model, generate new UI values, and then update the UI. But when views have their own state, it is possible for events to short-circuit this flow and update the UI without your consent – possibly incorrectly.

So, when using framework views, you have to ensure that your UI updates as expected, which sometimes means immediately undoing state changes the view makes on itself. This back and forth shuffling of data is hard to manage and is the cause of many UI bugs in Android apps.

Compose is different. In an app created with Compose, UI state is stored in only one place. In fact, Compose does not store any UI state of its own. Instead, you are in complete control of your UI state. Take the **`Checkbox`** composable you are using in Coda Pizza:

```
Checkbox(
    checked = true,
    onCheckedChange = { /* TODO */ }
)
```

If this were a framework view, you would likely expect `checked` to set the initial checked state of the checkbox and `onCheckedChange` to be called whenever it changes between states. But in Compose, the semantics are a little different.

The `checked` parameter defines whether the **`Checkbox`** is *currently* checked. You hardcoded this value to `true`, so the checkbox is permanently in the checked state – unless you change your code. As you might guess, this is not really how developers design their checkbox composables. Later in this chapter, you will instead set this parameter to a variable, allowing your composable to update when the variable providing its value is reassigned.

Meanwhile, `onCheckedChange` is called whenever the **`Checkbox`** *requests* that its checked state change. In practice, this means that `onCheckedChange` is called each time the user interacts with the **`Checkbox`**, indicating that the user wants to toggle the box's checked state.

Generally, this lambda is defined so that it updates the input for `checked` with the new state – but it does not have to. Which is good, because in the finished Coda Pizza, you want the user to pick where the topping goes – you do not want to immediately toggle the state of the **`Checkbox`**.

All of this explains why your checkboxes are currently ignoring user input: Right now, you do nothing when the component requests that its state change, so the checked state of the boxes never changes.

By the way, this philosophy of state is why Compose is referred to as a *declarative* UI toolkit. You declare how you want your UI to appear; Compose does the heavy lifting of not only setting your UI up the first time but also keeping it up to date as its state changes. Your composables will have a live connection to any state they reference, and changes to their state will update any consumers of that state object with no extra effort on your part.

## Defining Your UI State

The first step in adding state to a Compose application is to define the models you will use to store it. For Coda Pizza, you need a place to hold the state of which toppings are selected. Create a new file called `Pizza.kt` in the `com.bignerdranch.android.codapizza.model` package. In this file, create a new data class called **`Pizza`**.

Give your data class one property: the toppings on the pizza, a **`Map<Topping, ToppingPlacement>`**. If a topping is present on the pizza, it will be added to this map as a key. The value will be the topping's position on the pizza. If a topping is not on the pizza, it will not have an entry in this map.

Listing 27.1 The **`Pizza`** class (`Pizza.kt`)

```
data class Pizza(
    val toppings: Map<Topping, ToppingPlacement> = emptyMap()
)
```

Representing your pizza this way makes it easy to determine whether and where a topping is on the pizza. It also prevents you from making unsupported combinations, like adding two instances of pepperoni to the entire pizza. (Coda Pizza does not have an option to change the quantity of a topping, only its placement.)

# Updating UIs with MutableState

With your **`Pizza`** model ready for use, you can begin incorporating UI state into your application. To get your bearings with how state behaves in Jetpack Compose, define a file-level property in `PizzaBuilderScreen.kt` to track selected toppings. Assign the `pizza` property's initial value to include some toppings by default: pepperoni and pineapple on the whole pizza. After defining your pizza state, also update your **`ToppingsList`** composable to determine the `placement` of a topping based on the `pizza` property.

Listing 27.2 Declaring state (`PizzaBuilderScreen.kt`)

```
private var pizza =
    Pizza(
        toppings = mapOf(
            Topping.Pepperoni to ToppingPlacement.All,
            Topping.Pineapple to ToppingPlacement.All
        )
    )
...
@Composable
private fun ToppingsList(
    modifier: Modifier = Modifier
) {
    LazyColumn(
        modifier = modifier
    ) {
        items(Topping.values()) { topping ->
            ToppingCell(
                topping = topping,
                placement = ToppingPlacement.Left,
                placement = pizza.toppings[topping],
                onClickTopping = {
                    // TODO
                }
            )
        }
    }
}
...
```

Run Coda Pizza. You will see the list of toppings, as before, but only the rows for pepperoni and pineapple will be checked, matching the values you specified in the `pizza` property. If you click a topping, it still will not change anything onscreen, because you have not reassigned the state.

In a moment, you will update the `onClickTopping` lambda in your **`ToppingsList`** composable so that topping selections can be changed. First, add a function to the **`Pizza`** class to make it easier to add and remove toppings.

Define a function called **`withTopping`** that returns a copy of the pizza with a given topping. The function should also accept a **`ToppingPlacement`** to indicate where the topping is being placed. If a `null` value is sent for the placement, the new pizza should have that topping removed. Use the **`copy`** function to make the updated **`Pizza`** instance:

### Listing 27.3 Easy pizza changes (`Pizza.kt`)

```
data class Pizza(
    val toppings: Map<Topping, ToppingPlacement> = emptyMap()
) {
    fun withTopping(topping: Topping, placement: ToppingPlacement?): Pizza {
        return copy(
            toppings = if (placement == null) {
                toppings - topping
            } else {
                toppings + (topping to placement)
            }
        )
    }
}
```

Why create copies of your **`Pizza`** objects instead of making the `toppings` parameter a `var` or a **`MutableMap`**? As you will see shortly, Compose is aware of changes to your UI state – but only when your state object itself is reassigned. If a property of a UI state object changes, the UI will not update as expected. This can cause problems in your application. For this reason, we recommend making UI state classes only contain `val` properties.

With this helper function in place, you can implement your `onClickTopping` lambda. Do so now, setting the `pizza` property to an updated pizza. For now, keep the implementation simple: If the topping is on the whole pizza, your lambda should remove it; otherwise, it should be added to the whole pizza. To watch as your `pizza` is modified, also add a custom setter to the `pizza` property to print a log message each time your pizza's state is reassigned.

### Listing 27.4 Updating UI state (`PizzaBuilderScreen.kt`)

```
private var pizza =
    Pizza(
        toppings = mapOf(
            Topping.Pepperoni to ToppingPlacement.All,
            Topping.Pineapple to ToppingPlacement.Left
        )
    )
    set(value) {
        Log.d("PizzaBuilderScreen", "Reassigned pizza to $value")
        field = value
    }
...
@Composable
private fun ToppingsList(
    modifier: Modifier = Modifier
) {
    LazyColumn(
        modifier = modifier
    ) {
        items(Topping.values()) { topping ->
            ToppingCell(
                topping = topping,
                placement = pizza.toppings[topping],
                onClickTopping = {
                    // TODO
                    val isOnPizza = pizza.toppings[topping] != null
                    pizza = pizza.withTopping(
                        topping = topping,
                        placement = if (isOnPizza) {
                            null
                        } else {
                            ToppingPlacement.All
                        }
                    )
                }
            )
        }
    }
}
...
```

Run Coda Pizza and press the cell labeled Pineapple to remove pineapple from your pizza. (You will need to press the cell itself – not the checkbox.) In Logcat, you should see the message reporting that the state has changed along with the new value of your `pizza` property:

```
D/PizzaBuilderScreen: Reassigned pizza to Pizza(toppings={Pepperoni=All})
```

But despite the `pizza` state having changed, as the log shows, your UI did not update. It still shows pineapple as being on the whole pizza. This is in line with what you might expect based on your experience with framework UIs: Updating your application state without telling the UI it needs to update results in a stale UI. However, as you will see shortly, Jetpack Compose can be made aware of this reassignment so that it updates your UI automatically.

Before you wire up Compose to update your UIs automatically, do another quick test to confirm that your application state has correctly changed. With Coda Pizza still running, rotate your emulator or device to trigger a configuration change. This will cause your activity to be re-created, as you have seen before, which in turn means that your Compose UI will get rebuilt. Because your state is defined as a global variable, it will survive this configuration change and be used when your UI is redrawn. You will see that after rotating, the list of toppings updates to match your `pizza` state – no more pineapple.

Currently, your composable functions only know how to set up their initial state. The variables that define your UI state can change all they want, but your UI does not yet have a way to know about these changes, so your composables never update with fresh data. To fix this issue, you need a mechanism to tell your composable functions when and how to update.

In your time with the framework UI toolkit, you have been responsible for figuring out when changes to your application state should cause updates in your UI. But Compose updates your UI state by observing your application state. To allow this observation to happen, you need a **`MutableState`**.

**`MutableState`** (like its read-only sibling, **`State`**) is a wrapper object that keeps track of a single value. Whenever the value inside one of these state objects is reassigned, Compose is immediately notified of the change. Every composable that accesses the state object will then automatically update with the new value held in the state object.

Because your `pizza` state is not tracked with a state object, Compose cannot do anything in response to its value changing. Fix this by storing your pizza inside a **`MutableState`** instance. To create a **`MutableState`** object, use the **`mutableStateOf`** function, which requires an initial value. Pass an empty **`Pizza`** for this parameter to ensure that users start customizing their pizza from a clean slate. Also, remove the custom setter you added, as you no longer need the log message.

You can make this change without altering any of your usages of `pizza` by delegating your property to the mutable state value with the `by` keyword you have seen before. Delegation using `by` makes the property look like a normal property syntactically – but reads and writes will go through the **`MutableState`** so that Compose can keep track of state changes.

Type slowly when entering the delegation syntax. There are two functions you need to import to allow this syntax, which Android Studio can sometimes be finicky about. The full import statements for these two functions are shown in Listing 27.5.

Listing 27.5 Storing values in a **`MutableState`** (`PizzaBuilderScreen.kt`)

```
...
import androidx.compose.runtime.getValue
import androidx.compose.runtime.setValue

private var pizza =
    Pizza(
        toppings = mapOf(
            Topping.Pepperoni to ToppingPlacement.All,
            Topping.Pineapple to ToppingPlacement.Left
        )
    )
    set(value) {
        Log.d("PizzaBuilderScreen", "Reassigned pizza to $value")
        field = value
    }

private var pizza by mutableStateOf(Pizza())
...
```

Run Coda Pizza. Initially, all your toppings will be deselected, matching the empty **`Pizza`** state you use when initializing your application. Try clicking the cell for any of your toppings (again – click the cell itself, not the checkbox).

You will see that the topping you click is toggled: Its checkbox will become ticked and the label Whole pizza will appear under the topping name. Click it again, and the check mark and placement text will disappear.

Now click a checkbox. You will see that it registers the click with a visual touch indication (a dark circle), but it will not change from checked to unchecked or vice versa. Time to fix that.

The click behavior for the checkbox will be the same as the behavior for the cell itself: It will toggle the presence of the topping on the pizza. In Chapter 28, when you add a dialog to ask where on the pizza the topping should be placed, the two will still behave identically – both will show the dialog.

**`ToppingCell`** already has everything it needs to implement this behavior. Implement the `onCheckedChange` lambda for its **`Checkbox`** by calling the same `onClickTopping` lambda you used for your **`clickable`** modifier.

### Listing 27.6 Implementing the **`Checkbox`** (`ToppingCell.kt`)

```
...
@Composable
fun ToppingCell(
    topping: Topping,
    placement: ToppingPlacement?,
    onClickTopping: () -> Unit,
    modifier: Modifier = Modifier
) {
    Row(
        verticalAlignment = Alignment.CenterVertically,
        modifier = modifier
            .clickable { onClickTopping() }
            .padding(vertical = 4.dp, horizontal = 16.dp)
    ) {
        Checkbox(
            checked = (placement != null),
            ~~onCheckedChange = { /* TODO */ }~~
            onCheckedChange = { onClickTopping() }
        )
        ...
    }
}
```

Run Coda Pizza again and click a checkbox. This time, you will notice that the checkbox toggles the presence of the topping, just like the cell does. And no matter where you click, the checkbox or the topping cell, both elements of the UI update at once. There is no intermediate step where you need to remember to update the placement text when the checkbox changes state, and the two will never be out of sync.

## Recomposition

Compared to your work in the framework UI toolkit to keep your app state and UI state in sync, Compose's automatic UI updates could seem a bit magical. Allow us to dispel some of the magic.

To see how Compose updates your UI firsthand, add a log statement to your **`ToppingCell`** composable to print a message each time the function is invoked.

Listing 27.7 Pulling back the curtain (`ToppingCell.kt`)

```
...
@Composable
fun ToppingCell(
    topping: Topping,
    placement: ToppingPlacement?,
    onClickTopping: () -> Unit,
    modifier: Modifier = Modifier
) {
    Log.d("ToppingCell", "Called ToppingCell for $topping")
    Row(
        ...
    ) {
        ...
    }
}
```

Run Coda Pizza and open Logcat. Right after Coda Pizza starts, you will see that your new log message has printed several times – once for each topping that the **`LazyColumn`** placed onscreen. (You might see these logs print again as your UI settles. That is OK. The important thing is that **`ToppingCell`** gets called for each of your toppings when the list is rendered.)

```
D/ToppingCell: Called ToppingCell for Basil
D/ToppingCell: Called ToppingCell for Mushroom
D/ToppingCell: Called ToppingCell for Olive
D/ToppingCell: Called ToppingCell for Peppers
D/ToppingCell: Called ToppingCell for Pepperoni
D/ToppingCell: Called ToppingCell for Pineapple
```

Toggle the pepperoni topping by clicking its checkbox or name and take a close look at Logcat:

```
D/ToppingCell: Called ToppingCell for Basil
D/ToppingCell: Called ToppingCell for Mushroom
D/ToppingCell: Called ToppingCell for Olive
D/ToppingCell: Called ToppingCell for Peppers
D/ToppingCell: Called ToppingCell for Pepperoni
D/ToppingCell: Called ToppingCell for Pineapple
D/ToppingCell: Called ToppingCell for Pepperoni
```

You have just witnessed *recomposition* in action. Recomposition is the technique that Compose uses to update your UI when your state changes.

Compose keeps track of which composables are using which state. When a composable's state or any of its inputs change, Compose needs to update the composable for its new state. It does this by recomposing the composable.

When a composable is recomposed, the Compose runtime invokes the function again with its new inputs. The function is executed from the beginning, and whatever UI is created by this recomposition will replace whatever the composable had shown previously. Every expression in a function being recomposed gets called again – including your log expressions, in this case.

Here, Compose knows that the **`ToppingCell`** for pepperoni uses the `pizza` state to set both its checkbox and its topping placement label. Clicking the checkbox or cell modifies your `pizza` property with a new value that has an updated toppings map. Compose sees this reassignment, determines that this change affects the **`ToppingCell`** composable for pepperoni, and invokes the function again.

Compose has many tricks up its sleeve to avoid unnecessary work when your UI is recomposed. Here, only the inputs to the **`ToppingCell`** for pepperoni changed, so it will be the only function that gets recomposed. None of the other **`ToppingCell`**s were recomposed, nor were your **`PizzaBuilderScreen`** and **`OrderButton`** composables, so they are not called again.

Because your composables can be invoked at any time, you need to be careful about what you do inside a composable. A composable should be a *pure* function, meaning that it should have no *side effects* during its composition. A side effect is any operation that causes a change somewhere outside the function in question. For example, writing a value to a database or to a variable defined outside of the function itself would be side effects, because these operations impact the behavior of other parts of your application.

Side effects are dangerous in composables. Because you never know when your composable will be recomposed, you cannot control when or how many times an operation happens. If you are not careful, you could easily get into a situation where composition triggers recomposition – possibly in an endless cycle.

It is OK – and expected – to have side effects in a callback, like a click listener in response to user input. But a side effect should never appear inside the composition itself.

(Having said that, Compose does offer mechanisms to safely host side effects inside a composable. The details are outside the scope of this book, but if you are interested you can find more information at `developer.android.com/jetpack/compose/side-effects`.)

Recomposition is a crucial part of how Compose works: Any time a Compose UI changes, there is a corresponding recomposition.

Recomposition can also happen without changing the UI. If any of the inputs to your composables change, they will always recompose – even if the UI they present is not affected. In this case, recomposition will be imperceptible to users.

Now that you have uncovered some of Compose's magic, you can remove the log that prints when **`ToppingCell`** is called.

### Listing 27.8 Ending the magic show (`ToppingCell.kt`)

```
...
@Composable
fun ToppingCell(
    topping: Topping,
    placement: ToppingPlacement?,
    onClickTopping: () -> Unit,
    modifier: Modifier = Modifier
) {
    ~~Log.d("ToppingCell", "Called ToppingCell for $topping")~~
    Row(
        ...
    ) {
        ...
    }
}
```

## remember

Currently, the `pizza` state is defined as a global variable. This is not ideal, since global variables in general can become tricky to manage and maintain. It would be better if the state were owned by the composable itself. Because composables are nothing more than functions, the only place to encapsulate state is inside the function itself.

This presents an issue: Because Compose will call your entire composable function from the ground up every time it recomposes your UI, any local variables you declare inside a composable function will be lost between compositions. If you tried to store state inside a composable, it would reset each time the composable is invoked – which is not an ideal mechanism for storing UI state.

To address this issue, you will use a function called **`remember`**. **`remember`** takes in a lambda expression as its argument. On the first composition, the lambda is invoked to generate the remembered value, which the function then returns.

On subsequent compositions, **`remember`** immediately returns the value from the previous composition. This allows you to persist information across compositions, which is imperative for any information that cannot be derived from the composable's inputs.

Composables can have any number of remembered values. Also, Compose keeps track of which instances of a composable have remembered which values. If you have several instances of the same composable, they will each remember their own values.

**`remember`** is frequently used with **`mutableStateOf`** to define state for a composable. In fact, Android Studio will flag your code with an error if you attempt to call **`mutableStateOf`** inside a composable without wrapping it in a **`remember`** block.

Armed with this knowledge, you are ready to move your `pizza` property. Right now, the only composable that needs your pizza state is the **`ToppingsList`**, making it an ideal candidate to store this state.

This will not be the final place that your pizza state is stored – in fact, you will find yourself needing to store this state in a different composable very soon. However, it is fairly common as your application evolves to move where your state is defined – so much so that it is a rite of passage for new Compose developers. Luckily, this kind of refactoring is straightforward in Compose.

Relocate your `pizza` state into your **`ToppingsList`** composable (but do not let it get too comfortable).

### Listing 27.9 Storing state inside a composable (`PizzaBuilderScreen.kt`)

```
~~private var pizza by mutableStateOf(Pizza())~~
...
@Composable
private fun ToppingsList(
    modifier: Modifier = Modifier
) {
    var pizza by remember { mutableStateOf(Pizza()) }

    LazyColumn(
        modifier = modifier
    ) {
        ...
    }
}
```

Although this code might look like you are delegating onto the **`remember`** function, keep in mind that **`remember`** will be returning a **`MutableState<Pizza>`**. *This* is the value that the `pizza` variable will delegate to, and it behaves exactly the same as the state delegation you set up before.

Run Coda Pizza again. You should see the same behavior as before, but now you have rid `PizzaBuilderScreen.kt` of a global variable that could add complications later.

## State Hoisting

For your Coda Pizza app, you want the PLACE ORDER button to display the price of a pizza based on its toppings: A plain cheese pizza costs $9.99, and each topping adds $1.00 for the whole pizza or $0.50 for half the pizza. Define a computed property called `price` on your **`Pizza`** class to keep track of this price information.

### Listing 27.10 Pricing pizzas (`Pizza.kt`)

```
import com.bignerdranch.android.codapizza.model.ToppingPlacement.*

data class Pizza(
    val toppings: Map<Topping, ToppingPlacement> = emptyMap()
) {
    val price: Double
        get() = 9.99 + toppings.asSequence()
            .sumOf { (_, toppingPlacement) ->
                when (toppingPlacement) {
                    Left, Right -> 0.5
                    All -> 1.0
                }
            }
    ...
}
```

Make sure you import your **`Left`**, **`Right`**, and **`All ToppingPlacement`**s, not the Compose constants with the same names.

Now, you need to make the `pizza` state accessible in your **`OrderButton`** composable.

Currently, this state is held by **`ToppingsList`**, and there is not a great way to share information with **`OrderButton`**, because the two composables are siblings within **`PizzaBuilderScreen`**. From Compose's perspective, these two composables are entirely unrelated; they cannot communicate with one another directly.

You need to move the state where both **`ToppingsList`** and **`OrderButton`** can access it – like their shared parent, **`PizzaBuilderScreen`**. (We did warn you this was coming.)

The need to move state out of a composable and up to the composable's caller is so common in development with Compose that it has a name: *state hoisting*. The pattern for hoisting state involves removing state from a composable and defining it instead as a parameter of the composable. If the composable also needs to update the state, it should take in a lambda expression that will be called with information about the change being made.

Think about your **`Checkbox`**. It does not hold any state itself; instead, it accesses the same `pizza` state that is currently held in **`ToppingsList`**. This, remember, is how your UI elements stay so effortlessly in sync.

In fact, **`Checkbox`** follows the state hoisting pattern. The same is true of the other commonly used built-in composables that change appearance in response to user input (like **`TextField`**, **`Switch`**, and **`Slider`**). This lets composables that depend on these components maintain complete control of their children's behavior. By hoisting state out of a composable, you end up with a flexible component whose behavior can be customized.

By the way, although state hoisting is a great tool for making more generalized components, do not stress about ensuring that *all* your composables are stateless. Many composables can effectively hold their own state, and you are free to decide where your UI state is held. And if you change your mind, you can easily refactor your code to incorporate state hoisting – as you will see momentarily.

To hoist the `pizza` state out of **`ToppingsList`**, you will make three changes: First, you will move the declaration of the `pizza` state into **`PizzaBuilderScreen`**. Second, you will define two new parameters on **`ToppingsList`**: a **`Pizza`** object to display in the list and a lambda expression that will be called when the pizza is modified to supply a new value for the pizza. With these arguments in place, you will then update any writes to the `pizza` property with calls to your lambda.

Make these changes now to hoist your `pizza` state (Listing 27.11).

### Listing 27.11 Pizza hoisting (`PizzaBuilderScreen.kt`)

```
@Preview
@Composable
fun PizzaBuilderScreen(
    modifier: Modifier = Modifier
) {
    var pizza by remember { mutableStateOf(Pizza()) }

    Column(
        modifier = modifier
    ) {
        ToppingsList(
            pizza = pizza,
            onEditPizza = { pizza = it },
            modifier = Modifier
                .fillMaxWidth()
                .weight(1f, fill = true)
        )
        ...
    }
}

@Composable
private fun ToppingsList(
    pizza: Pizza,
    onEditPizza: (Pizza) -> Unit,
    modifier: Modifier = Modifier
) {
    var pizza by remember { mutableStateOf(Pizza()) }

    LazyColumn(
        modifier = modifier
    ) {
        items(Topping.values()) { topping ->
            ToppingCell(
                topping = topping,
                placement = pizza.toppings[topping],
                onClickTopping = {
                    val isOnPizza = pizza.toppings[topping] != null
                    pizza = onEditPizza(pizza.withTopping(
                        topping = topping,
                        placement = if (isOnPizza) {
                            null
                        } else {
                            ToppingPlacement.All
                        }
                    ))
                }
            )
        }
    }
}
...
```

Run Coda Pizza and confirm that its behavior has not changed.

With the pizza state now owned by **PizzaBuilderScreen**, you are ready to show price information in your **OrderButton**. Start by updating your string resource file to include a spot for the price to appear, using a format string placeholder.

### Listing 27.12 Adding a price tag (strings.xml)

```
<resources>
    ...
    <string name="place_order_button">Place Order (%1$s)</string>
    ...
</resources>
```

Next, pass a pizza instance from **PizzaBuilderScreen** to **OrderButton** and use an instance of **NumberFormat** to convert the price property to a formatted string that you can display.

### Listing 27.13 Showing pizza prices (PizzaBuilderScreen.kt)

```
@Preview
@Composable
fun PizzaBuilderScreen(
    modifier: Modifier = Modifier
) {
    ...
    Column(
        modifier = modifier
    ) {
        ...
        OrderButton(
            pizza = pizza,
            modifier = Modifier
                .fillMaxWidth()
                .padding(16.dp)
        )
    }
}
...
@Composable
private fun OrderButton(
    pizza: Pizza,
    modifier: Modifier = Modifier
) {
    Button(
        modifier = modifier,
        onClick = {
            // TODO
        }
    ) {
        val currencyFormatter = NumberFormat.getCurrencyInstance()
        val price = currencyFormatter.format(pizza.price)
        Text(
            text = stringResource(R.string.place_order_button, price)
                .toUpperCase(Locale.current)
        )
    }
}
```

Run Coda Pizza and test your new feature: When no toppings are added, the order button's text should read PLACE ORDER ($9.99). For each topping you add to the pizza, the price will increase by $1, the price of a topping placed on the entire pizza.

Notice that your **`ToppingsList`** composable can automatically update the **`OrderButton`** composable. By simply editing the state that backs the UI, every consumer of the UI state instantly receives its latest value. There is no extra effort on your part to track down all the reasons a certain UI element might need to be updated.

By the way, **`NumberFormat`** objects are slightly expensive to allocate, so discarding them between compositions is a bit wasteful. This is another practical application of **`remember`**, which you can use to keep resources available between recompositions. Wrap your **`NumberFormat`** in a **`remember`** block to try this out for yourself.

### Listing 27.14 **`remember`**ing a **`NumberFormatter`** (`PizzaBuilderScreen.kt`)

```
...
@Composable
private fun OrderButton(
    pizza: Pizza,
    modifier: Modifier = Modifier
) {
    Button(
        modifier = modifier,
        onClick = {
            // TODO
        }
    ) {
        val currencyFormatter = remember { NumberFormat.getCurrencyInstance() }
        val price = currencyFormatter.format(pizza.price)
        Text(
            text = stringResource(R.string.place_order_button, price)
                .toUpperCase(Locale.current)
        )
    }
}
```

Run Coda Pizza again. The behavior of your app will not change – and because this overhead is fairly small and modern phones are very fast, the performance difference should be imperceptible. But you can now rest easy knowing that you are not unnecessarily throwing out work that you will need to redo on the next composition.

## State and Configuration Changes

Currently, Coda Pizza has a small problem, and it will instantly be familiar to you. Run Coda Pizza and add a topping or two to the pizza. Then rotate your device or emulator.

Yep. The topping selections are lost after the configuration change. **`remember`** persists state across recompositions, but it has its limits: When your **`Activity`** is destroyed and re-created, so is your composition. Because the composition is discarded, it will restart from a blank slate when your **`Activity`** is re-created and calls **`setContent`**.

This was not an issue when you declared the `pizza` state as a global variable. But now, because your state is associated with your composition hierarchy – and, by extension, your activity – it must obey the rules of the activity lifecycle. Every variable stored using **`remember`** will be lost after a configuration change (and process death), just like values stored in your **`Activity`**.

In your time with the framework UI toolkit, you saw two approaches to solve this problem: the `savedInstanceState` bundle and **`ViewModel`**. Although **`ViewModel`**s can be a great tool for managing UI state and can be used in Jetpack Compose, they require more setup than is warranted for your needs in Coda Pizza. `savedInstanceState` will be your solution, and you will access it using a variation of **`remember`** called **`rememberSaveable`**.

The "saveable" portion of this function's name refers to the fact that any remembered value is also automatically written to your **`Activity`**'s `savedInstanceState` bundle when it is destroyed. Remembered values that were saved can be restored when your composition is re-created, so values remembered in this way also survive configuration changes.

**`rememberSaveable`** is called in the same way as **`remember`**, using a lambda to perform its initialization. Try it out now. (This change will introduce a problem in your code, which we will explain next.)

### Listing 27.15 Remembering and saving (`PizzaBuilderScreen.kt`)

```
@Preview
@Composable
fun PizzaBuilderScreen(
    modifier: Modifier = Modifier
) {
    var pizza by remember Saveable { mutableStateOf(Pizza()) }
    ...
}
...
```

Run Coda Pizza. It will crash with the following exception:

```
IllegalArgumentException: MutableState containing Pizza(toppings={}) cannot
be saved using the current SaveableStateRegistry. The default implementation
only supports types which can be stored inside the Bundle. Please consider
implementing a custom Saver for this class and pass it as a stateSaver
parameter to rememberSaveable().
```

Coda Pizza has crashed because it is attempting to write a **`Pizza`** to a **`Bundle`**. But **`Bundle`**s are restricted in the types they can store: Only instances of **`Serializable`**, **`Parcelable`**, and basic types like **`String`**, **`Int`**, **`Long`**, **`Float`**, and **`Double`** are allowed in a **`Bundle`**. To fix this crash, you need to convert **`Pizza`** into a type that can be added to a **`Bundle`**.

## Parcelable and Parcelize

The most effective way to let a **Pizza** fit into a **Bundle** is to make it a **Parcelable** class. **Parcelable** is an interface provided by the Android OS that allows a class to be converted into and read out of a **Parcel** object. **Parcel**s allow for compact storage of objects and are ideal for use in a **Bundle**.

The process of manually implementing the **Parcelable** interface is a bit complex – plus there are limitations about which data types can appear in the **Parcel**. Luckily, there is a plugin to help. With a bit of setup, you can have a **Parcelable** implementation automatically generated for you.

To do this, you will need to add a new plugin to your project called Parcelize, which takes care of generating **Parcelable** implementations for you at build time. Because Parcelize is a compiler plugin, your **Parcelable** implementations will always stay up to date with your class definitions, preventing errors when converting a **Parcel** back into the original object.

To add the Parcelize plugin, first register it with your project by adding its plugin ID and version to your `build.gradle` file labeled (Project: Coda_Pizza).

### Listing 27.16 Adding the Parcelize plugin (`build.gradle`)

```
plugins {
    id 'com.android.application' version '7.1.2' apply false
    id 'com.android.library' version '7.1.2' apply false
    id 'org.jetbrains.kotlin.android' version '1.6.10' apply false
    id 'org.jetbrains.kotlin.plugin.parcelize' version '1.6.10' apply false
}
...
```

Next, apply this plugin to your application by registering it in the `app/build.gradle` file.

### Listing 27.17 Enabling the Parcelize plugin (`app/build.gradle`)

```
plugins {
    id 'com.android.application'
    id 'org.jetbrains.kotlin.android'
    id 'org.jetbrains.kotlin.plugin.parcelize'
}
...
```

After making these changes to your build configuration files, do not forget to click the Sync Now button to make Android Studio aware of your changes.

After the sync completes, you are ready to make your **Pizza** class implement the **Parcelable** interface. With the help of Parcelize, you can accomplish this with two small changes. First, annotate the **Pizza** class with `@Parcelize`. Second, make **Pizza** implement the **Parcelable** interface.

### Listing 27.18 Parcelizing pizza (`Pizza.kt`)

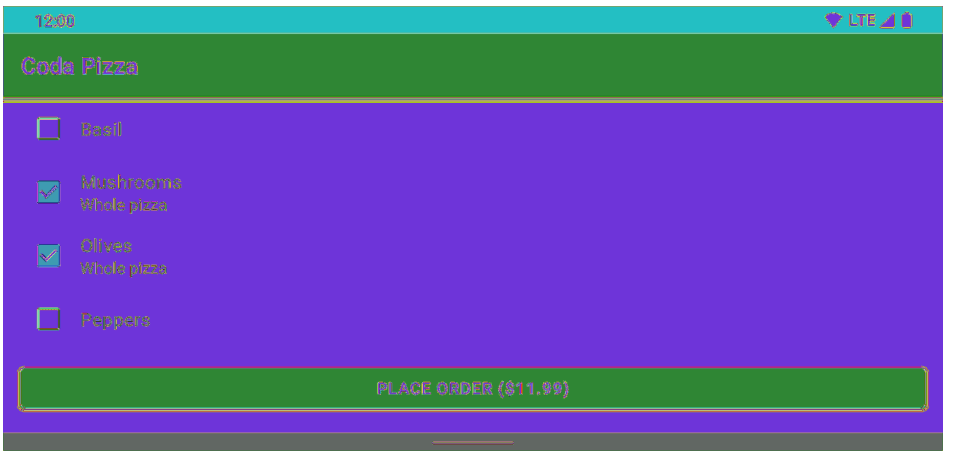
```
@Parcelize
data class Pizza(
    val toppings: Map<Topping, ToppingPlacement> = emptyMap()
) : Parcelable {
    ...
}
```

(If prompted, be sure to choose the import for `kotlinx.parcelize.Parcelize` instead of `kotlinx.android.parcel.Parcelize`. The `kotlinx.android` package is a relic of the now deprecated Kotlin Android Extensions plugin.)

**Parcelable** contains a few functions that all implementers must define. As you type this code, you may notice that errors about missing function overrides disappear as soon as you annotate the class with `@Parcelize`. Parcelize will automatically provide the entire implementation for this interface with no extra effort on your part.

Run Coda Pizza. This time, it will not crash, and you will be presented with the familiar list of toppings. Add some toppings to the pizza and rotate the emulator or device. The state should survive the configuration change, and you should see the same selection of toppings – no matter how many times you rotate the phone or what other configuration changes Coda Pizza encounters (Figure 27.2). And because your state is stored in the `savedInstanceState` bundle, it will even survive process death.

### Figure 27.2 Saving pizzas across configuration changes




State is a crucial part of any application, and in the world of Compose, it is entirely in your hands. But the discussion of state does not end there. In the next chapter, you will incorporate a dialog into Coda Pizza and see how Compose's philosophy of state lets you achieve a wide variety of app behaviors.

# For the More Curious: Coroutines, Flow, and Compose

Coda Pizza is a small application to get your feet wet with Compose. But what about bigger apps that need to store data or access a web service?

The Kotlin coroutines you met in Chapter 12 are a great fit for Compose, thanks to a few APIs that translate between the declarative and asynchronous worlds. In fact, Compose itself uses coroutines in a few of its APIs. You can see an example of one of these APIs in the next section (the section called For the More Curious: Scrolling State).

In Chapter 12, you also explored using **`StateFlow`** to manage UI state. At a high level, **`StateFlow`** and Compose's **`State`** classes do roughly the same thing: They emit a sequence of values over time and can be observed. With **`StateFlow`**, the observation is explicit – you call **`collect`** and specify what to do with each emission. Compose's **`State`** class works in a similar way, but the observation is implicit. Any time the value changes, all consumers are recomposed to get the update.

The easiest way to use values from a **`Flow`** inside a composable is to first convert the flow into a **`State`** object. You can do this with the **`collectAsState`** function. **`collectAsState`** will collect all the items from a flow and relay them to a **`State`** that can be used for composition. **`collectAsState`** does not need to be called from a coroutine scope and is itself a composable function. If you used the repository pattern in Coda Pizza, your **`PizzaBuilderScreen`** composable might obtain the `pizza` state like this:

```
@Composable
fun PizzaBuilderScreen(
    repository: PizzaRepository,
    modifier: Modifier = Modifier
) {
    val pizzaFlow: Flow <Pizza> = repository.getCustomizedPizza()
    val pizza: Pizza by pizzaFlow.collectAsState()
    ...
}
```

**`collectAsState`** returns a **`State`**, not a **`MutableState`**, so you cannot write values back to the flow. (This is also why `pizza` is declared as a `val` instead of a `var` in this example.) If you need to send updates back to the repository, you will need to call into it.

Chances are, any functions on **`PizzaRepository`** that update the pizza will be suspending functions. Your composables are not currently set up to launch coroutines, since they are not associated with a coroutine scope. To fix this, you would need to obtain a coroutine scope to launch coroutines from, which you can do using the **`rememberCoroutineScope`** function.

**`rememberCoroutineScope`** creates a coroutine scope and remembers it for future compositions. If the composable is removed from the composition in the future, the coroutine scope will be canceled. (By the way, composable functions cannot also be suspending functions, so you will always have an explicit coroutine scope when using a coroutine from a composable.)

One way to set up **`ToppingCell`** with a suspending call into a repository might look like this:

```
val coroutineScope = rememberCoroutineScope()
ToppingsList(
    pizza = pizza,
    onEditPizza = { updatedPizza ->
        coroutineScope.launch {
            pizzaRepository.setPizza(updatedPizza)
        }
    }
)
```

Notice that the call to **`launch`** happens inside the callback, not the composable itself. Although the coroutine scope will be remembered, composition still behaves as normal, meaning that calling **`launch`** during composition will cause the coroutine to be re-launched on each composition. Because of this, you should never call **`launch`** inside a composable directly; instead, **`remember`** the coroutine for use elsewhere in the composable.

With these integrations between Compose and coroutines, you can drive your front end with a robust data back end, like you did in CriminalIntent and PhotoGallery. Compose also has integrations for popular reactive libraries, including RxJava and LiveData. We encourage you to experiment with these integrations to apply some of the patterns you have seen previously to the world of Compose.

## For the More Curious: Scrolling State

Jetpack Compose's ideas about state and recomposition are woven throughout the framework. In fact, Coda Pizza was leveraging these two concepts even before you added state of your own.

Think about your **`LazyColumn`**. It needs to keep track of its scroll position, which it does automatically. But it also uses the state hoisting pattern to allow its parent to manage the scrollable state. How does it do this? To see for yourself, take a look at its signature:

```
@Composable
fun LazyColumn(
    modifier: Modifier = Modifier,
    state: LazyListState = rememberLazyListState(),
    ...
)
```

**`rememberLazyListState`** does two things: It creates a **`LazyListState`** object with an initial position at the beginning of the list, and it remembers that state via the **`remember`** function. For many lists, you do not need to think about this behavior – the default, automatically managed scrolling state will simply do the right thing. But if you ever need to read or control the scroll position of the **`LazyColumn`**, the option is always available.

To take the scroll state into your own hands, you can create your own **`LazyListState`** and pass it in as the `state` parameter. The code to do so might look like this:

```
val listState = rememberLazyListState()
LazyColumn(
    state = listState
) {
    // Add items to the LazyColumn
}
```

This code effectively does the same thing as the default, automatically managed `state` parameter, but you now have a reference to the state being used. This lets you both read the scroll state and modify it, if desired, which you can do like this:

```
// Determine whether the user is currently scrolled to the top of the list
val isAtTopOfList = (listState.firstVisibleItemIndex == 0) &&
    (listState.firstVisibleItemScrollOffset == 0)

// Scroll to the top of the list from the current scroll position
coroutineScope.launch {
    // Suspends until the scroll animation finishes
    listState.scrollToItem(index = 0, scrollOffset = 0)
}
```

The **`LazyListState`** backs its scroll position properties with **`State`** objects, meaning that the scroll position can be observed and trigger recompositions just as you have seen for other state. Most of the built-in composables will explicitly require state, but other composables that have an implicit or self-managed state will still offer some mechanism to read and control the state, as you saw with **`LazyColumn`**.

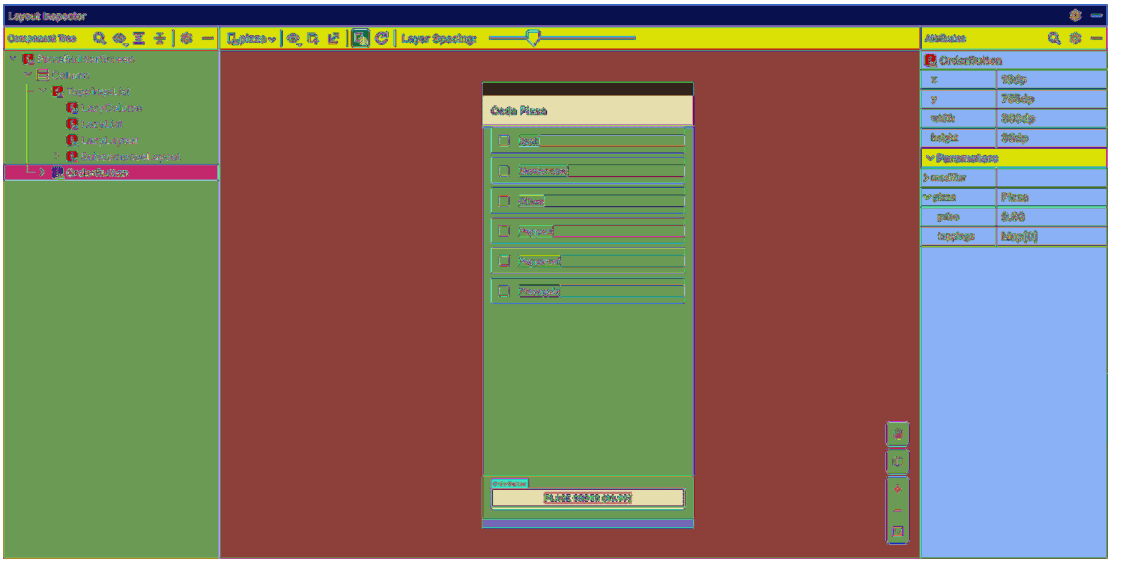
These design paradigms make the built-in composables highly flexible, and you can apply the same ideas to your own composables if you need to make flexible, reusable components like the ones included with Compose.

## For the More Curious: Inspecting Compose Layouts

Sometimes you will need to debug your Compose UIs just as you have debugged traditional UIs using the framework view system. In a challenge near the end of Chapter 5, you explored the layout inspector. The layout inspector allows you to see the configuration of your views, including their nesting, attributes, and position. You can also break the view into layers to see exactly what is being rendered by a given view.

The layout inspector (as well as many other Android UI debugging tools) fully supports Jetpack Compose. Try it out for yourself by opening the layout inspector with its menu option, Tools → Layout Inspector, then running Coda Pizza. The layout inspector will open in the bottom of the IDE (Figure 27.3).

Figure 27.3 Inspecting your Compose UI




Explore the component tree on the left side of the layout inspector. You will see all of the composables in your layout, represented as a hierarchy of nodes. Double-clicking a node will take you to the composable call that contributed the element to your UI.

Find and select your **`OrderButton`** composable in the tree. The screen preview in the center of the layout inspector will be marked with the bounding box of the button, and the attributes window on the right will update to show the composable's attributes. In the Parameters section, you will see all the inputs that were passed to the composable.

You cannot use the layout inspector to edit the attributes of a composable (nor the attributes of a **`View`**), but getting a better understanding of your layout can be invaluable for debugging.

Try experimenting with the other techniques you have learned to debug your apps. You will find that many or all of them still work in the declarative world of Jetpack Compose.

# 28

# Showing Dialogs with Jetpack Compose

Understanding how Jetpack Compose treats state and recomposition are crucial to effectively using Compose. The ramifications of state changes in your code being the trigger for UI updates extend throughout the framework. A great example of this is how Compose handles dialogs.

When you want to display a dialog in the framework UI toolkit, you use a class like **`AlertDialog`** or **`DatePickerDialog`** – ideally, wrapped in a **`DialogFragment`**. To display one of these dialogs, you call the appropriate **`show`** function. The dialog then does whatever it wants to do and dismisses itself. If there is a result it needs to send back to whatever component summoned it, it needs to safely transfer the data back to that location – often with a fair amount of orchestration.

In Compose, dialogs follow the same rule of declarative UIs that you have seen before: If you say that a dialog should be shown, it will be shown. You are in control of when the dialog disappears – the dialog can request that it be dismissed, but the call site gets the final say.

Because the state of whether the dialog is visible is known by the composable that hosts the dialog, there is no need to wrap the dialog in another container to manage its lifecycle. And because the dialog is directly managed by another composable, transferring data between the two is as simple as declaring a lambda expression to serve as a callback – no need to set up a finicky line of communication between two components.

In Coda Pizza, you will use a dialog to ask your users where they would like a topping to be placed on their pizza. The finished dialog will look like Figure 28.1, and it will appear any time the user selects a topping from the list.

Figure 28.1 The final dialog



# Your First Dialog in Compose

There are several types of dialogs available in Compose, including an **`AlertDialog`** composable that mimics the appearance of its framework counterpart. You will be using **`Dialog`**, which is more agnostic about its content and will let you build a more custom UI. Regardless of which dialog flavor you choose, the semantics are largely the same – especially when it comes to how the dialog's state is managed.

On its own, the **`Dialog`** function renders an empty window, so you will need to create a new composable that builds off of **`Dialog`** and provides the dialog's view. Building this UI will require a fair amount of code, so your composable for this dialog should be in its own file to keep your code organized.

Create a new file in the `ui` package called `ToppingPlacementDialog.kt`. In your new file, define a composable function called **`ToppingPlacementDialog`** that calls the **`Dialog`** function.

**`Dialog`** requires two inputs: an `onDismissRequest` lambda and a content lambda. We will revisit the role of `onDismissRequest` later – for now, leave it empty. To begin your work on the content, create a placeholder UI to show in the dialog: a boring red box. You will use this to make sure you have everything set up before adding more functionality. Use a **`Box`** composable and make it visible by setting its background color to `Color.Red` and its width and height to 64dp:

### Listing 28.1 Painting the dialog red (`ToppingPlacementDialog.kt`)

```
@Composable
fun ToppingPlacementDialog() {
    Dialog(onDismissRequest = { /* TODO */ }) {
        Box(
            modifier = Modifier
                .background(Color.Red)
                .size(64.dp)
        )
    }
}
```

(Be sure to add the import for `androidx.compose.ui.window.Dialog` instead of `android.app.Dialog`.)

To see your dialog in action, you need to call **`ToppingPlacementDialog`**. Because **`Dialog`** appears in its own window, it does not matter where in your composition this function call appears. The result will always be the same: a fullscreen dialog with the specified content. In this case, we recommend showing the dialog from **`ToppingsList`**.

**`ToppingsList`** is a good candidate for managing the dialog because it is a convenient place to manage the dialog's state. When any of the **`ToppingCell`**s in its **`LazyColumn`** are clicked, the dialog should be shown. **`ToppingsList`** already has visibility into how **`ToppingCell`**s are created, making this a small change.

You do not want to store this state too far up your composition hierarchy, since each level above **`ToppingsList`** means that you need another pair of parameters to access and change the state. No other component in Coda Pizza will need to be aware of this state, so managing it directly in **`ToppingsList`** will prevent unnecessary clutter in your code and make it easier to read.

Add a call to **`ToppingPlacementDialog`** in **`ToppingsList`**.

### Listing 28.2 Showing a dialog (`PizzaBuilderScreen.kt`)

```
...
@Composable
private fun ToppingsList(
    pizza: Pizza,
    onEditPizza: (Pizza) -> Unit,
    modifier: Modifier = Modifier
) {
      ToppingPlacementDialog()

      LazyColumn(
          modifier = modifier
      ) {
          ...
      }
  }
  ...
```

What do you think will happen when this code executes? To find out, run Coda Pizza. When the app launches, you will see your familiar list of toppings behind a dark overlay with a red square in the center of the window (Figure 28.2).

Figure 28.2 A test dialog




The red square is your dialog, and the dark overlay is being added by the system (the same overlay appears behind dialogs in the framework UI toolkit, as you may have noticed). With the dialog open, try dismissing it – either with the Back button or by clicking outside the dialog (on the dark backdrop). Despite your efforts, you will be unable to dismiss it.

## Dismissing the Dialog

Whenever a dialog is part of your composition, it will be shown onscreen. You have not told your **ToppingPlacementDialog** when it should stop being displayed, so attempts to dismiss it will do nothing. And none of Compose's dialog functions has a parameter to set the dialog's visibility, so you will need another way to dismiss your dialog.

In Chapter 26, you made another UI element that was visible only some of the time: the placement label in your **ToppingCell**. You use an `if` statement so that the **Text** composable is only invoked when you want it to be visible. You will use the same approach to show and hide the dialog.

With the dialog's visibility controlled by an `if` statement, the job of the `onDismissRequest` lambda will be to update the condition set by the `if` statement so that the **Dialog** function will not be invoked again when your UI is recomposed. This means that, for your dialog's visibility to be managed correctly, **ToppingPlacementDialog** will need to forward its requests to be dismissed.

Declare a new parameter called `onDismissRequest`, mirroring the parameter from the base **Dialog** composable, and pass it to your **Dialog**.

### Listing 28.3 Forwarding dismiss requests (`ToppingPlacementDialog.kt`)

```
@Composable
fun ToppingPlacementDialog(
    onDismissRequest: () -> Unit
) {
    Dialog(onDismissRequest = { /* TODO */ } onDismissRequest) {
        Box(
            modifier = Modifier
                .background(Color.Red)
                .size(64.dp)
        )
    }
}
```

With this parameter in place, **ToppingPlacementDialog** now has everything it needs to have its visibility managed. To track whether the dialog should be visible, define a new **MutableState** property and surround your call to **ToppingPlacementDialog** in an `if` statement that checks this state. You will want the dialog state to persist across configurations, so use **rememberSaveable** instead of **remember**.

Your new state will be driven by two events: When a topping is clicked, the dialog should be shown. When the dialog requests to be dismissed, it should be hidden. Drive this state by setting your `onClickTopping` and `onDismissRequest` callback implementations with an update to your state.

Listing 28.4 Managing your dialog's state (`PizzaBuilderScreen.kt`)

```
...
@Composable
private fun ToppingsList(
    pizza: Pizza,
    onEditPizza: (Pizza) -> Unit,
    modifier: Modifier = Modifier
) {
      var showToppingPlacementDialog by rememberSaveable { mutableStateOf(false) }

      if (showToppingPlacementDialog) {
          ToppingPlacementDialog(
              onDismissRequest = {
                  showToppingPlacementDialog = false
              }
          )
      }

      LazyColumn(
          modifier = modifier
      ) {
          items(Topping.values()) { topping ->
              ToppingCell(
                  topping = topping,
                  placement = pizza.toppings[topping],
                  onClickTopping = {
                      val isOnPizza = pizza.toppings[topping] != null
                      onEditPizza(pizza.withTopping(
                          topping = topping,
                          placement = if (isOnPizza) {
                              null
                          } else {
                              ToppingPlacement.All
                          }
                      ))
                      showToppingPlacementDialog = true
                  }
              )
          }
      }
  }
  ...
```

By the way, although you could technically use Kotlin's trailing lambda syntax to omit the parameter name and parentheses after **`ToppingPlacementDialog`**, we do not recommend it. Callbacks for composables are most effectively identified by their name, and it can be difficult to determine what a lambda does in Compose when used with the trailing lambda syntax.

We recommend only using the trailing lambda syntax with a composable when you are passing in its main content. If the parameter name is anything besides `content`, the conventional name for the "primary" content of a composable, the trailing lambda syntax can remove a label that is important in understanding how your UI will appear.

Run Coda Pizza. Now, the app displays the toppings list, as before. Clicking a topping shows the placeholder dialog, which can now be dismissed by clicking outside the dialog or by pressing the Back button. The dialog is not dismissed if the user presses the red square itself, which lets the user interact with the dialog without dismissing it.

## Setting the Dialog's Content

Now that the dialog can show and hide itself, you can focus on its content. Start by adding a string resource to show in the dialog.

Listing 28.5 Asking important questions (`strings.xml`)

```
<resources>
  ...
  <string name="place_order_button">Place order (%1$s)</string>

  <string name="placement_prompt">Where do you want %1$s on your pizza?</string>
  <string name="placement_none">None</string>
  <string name="placement_left">Left half</string>
  <string name="placement_right">Right half</string>
  <string name="placement_all">Whole pizza</string>
  ...
</resources>
```

Now you are ready to start building the real UI to appear in the dialog. Remove the temporary **`Box`** and replace it with a **`Card`**. **`Card`** includes a background, drop shadow, and rounded corners – exactly what you need for your dialog. Its children are stacked on top of one another (like a **`FrameLayout`**'s), so you will only include one direct child in the **`Card`**.

In your **`Card`**, add a **`Text`** with the prompt you just declared. Place the **`Text`** in a **`Column`**, because you will need to add buttons underneath the prompt shortly. You will also need to add a new parameter to accept the name of the topping being added to the pizza.

Listing 28.6 Asking the right question (`ToppingPlacementDialog.kt`)

```
@Composable
fun ToppingPlacementDialog(
    topping: Topping,
    onDismissRequest: () -> Unit
) {
    Dialog(onDismissRequest = onDismissRequest) {
        Box(
            modifier = Modifier
                .background(Color.Red)
                .size(64.dp)
        )
        Card {
            Column {
                val toppingName = stringResource(topping.toppingName)
                Text(
                    text = stringResource(R.string.placement_prompt, toppingName),
                    style = MaterialTheme.typography.subtitle1,
                    textAlign = TextAlign.Center,
                    modifier = Modifier.padding(24.dp)
                )
            }
        }
    }
}
```

Because you have added a new parameter to **`ToppingPlacementDialog`**, **`ToppingsList`** will now have a compiler error. Your dialog needs to know not only whether it should be visible but also what content it should show. More specifically, your dialog needs to know which topping was selected, not just that *a* topping was selected.

To keep track of this information, you will need to be a bit more clever about the dialog state you store. Instead of keeping track of whether the dialog should appear, your state can instead track which topping you are in the process of putting on the pizza.

If the user has not selected a topping, this state should be null to indicate that no topping was selected. Otherwise, the most recently selected topping can be remembered and shown in the dialog. Make this change now, replacing your current `showToppingPlacementDialog` state.

### Listing 28.7 Smarter state (`PizzaBuilderScreen.kt`)

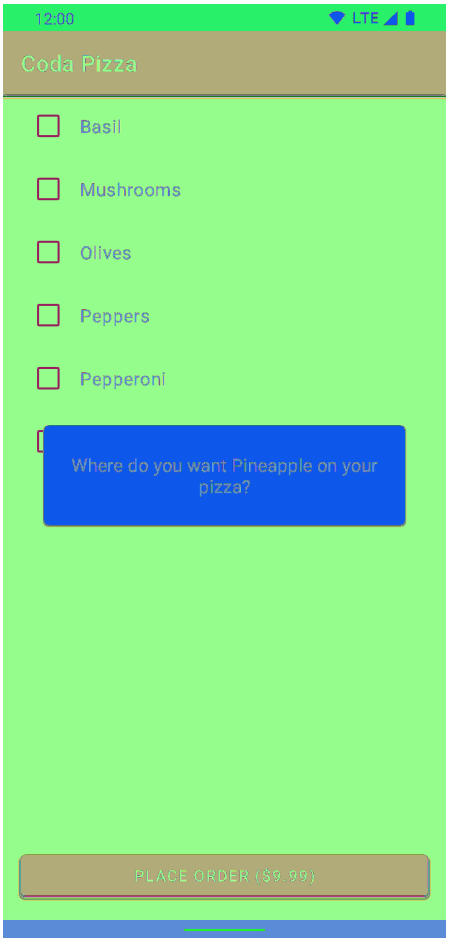
```
...
@Composable
private fun ToppingsList(
    pizza: Pizza,
    onEditPizza: (Pizza) -> Unit,
    modifier: Modifier = Modifier
) {
    var showToppingPlacementDialog by rememberSaveable { mutableStateOf(false) }
    var toppingBeingAdded by rememberSaveable { mutableStateOf<Topping?>(null) }

    if (showToppingPlacementDialog) {
    toppingBeingAdded?.let { topping ->
        ToppingPlacementDialog(
            topping = topping,
            onDismissRequest = {
                showToppingPlacementDialog = false
                toppingBeingAdded = null
            }
        )
    }

    LazyColumn(
        modifier = modifier
    ) {
        items(Topping.values()) { topping ->
            ToppingCell(
                topping = topping,
                placement = pizza.toppings[topping],
                onClickTopping = {
                    showToppingPlacementDialog = true
                    toppingBeingAdded = topping
                }
            )
        }
    }
}
...
```

Run Coda Pizza once again and select any of the toppings. You will now see a dialog – one that actually looks more like a dialog this time – asking the user about the topping they just selected (Figure 28.3). Dismissal will work as it has before, but because there are no placement options yet, your users are still confined to the dullness of cheese pizzas.

Figure 28.3 The beginning of a dialog



Time to add those topping placement options. You will add four options to this dialog: Whole pizza, Left half, Right half, and None. There are several composables you could use to create these options, but **`TextButton`** is a great fit for your needs.

Each of the buttons you will add to this dialog will require a similar set of customizations. They will all need to fill the width of the dialog, they will all have 8dp of padding, and they will all pull their labels from the topping's string resource. To make these buttons a bit easier to add, start by declaring a **`ToppingPlacementOption`** composable in `ToppingPlacementDialog.kt`. You will use this composable to add the choices to your dialog.

### Listing 28.8 Defining a reusable button (`ToppingPlacementDialog.kt`)

```
@Composable
fun ToppingPlacementDialog(
    topping: Topping,
    onDismissRequest: () -> Unit
) {
    ...
}

@Composable
private fun ToppingPlacementOption(
    @StringRes placementName: Int,
    onClick: () -> Unit,
    modifier: Modifier = Modifier
) {
    TextButton(
        onClick = onClick,
        modifier = modifier.fillMaxWidth()
    ) {
        Text(
            text = stringResource(placementName),
            modifier = Modifier.padding(8.dp)
        )
    }
}
```

Much like **`Button`**, **`TextButton`** accepts a lambda to define the label of the button. This means that, despite its name, it is possible to place something like an icon inside a **`TextButton`**. **`TextButton`** simply has a few optimizations that make it ideal for hosting **`Text`**, like automatically setting the text color with an appropriate button color. But even with these handy default customizations, you do not want to duplicate this hierarchy for each button you want to add to the dialog. With your **`ToppingPlacementOption`** composable, adding a button to the dialog is a single function call away.

Next, you can declare the four buttons in the dialog. You could declare them one by one – but remember that you have control flow at your disposal: You can use a loop to add several items onscreen at once. Try it out now by iterating over all values of **`ToppingPlacement`**. (Leave each button's `onClick` callback blank for now.) Remember that you did not add the "none" option as a case to **`ToppingPlacement`**, so you will need to manually add the fourth option to this dialog.

### Listing 28.9 Adding options (`ToppingPlacementDialog.kt`)

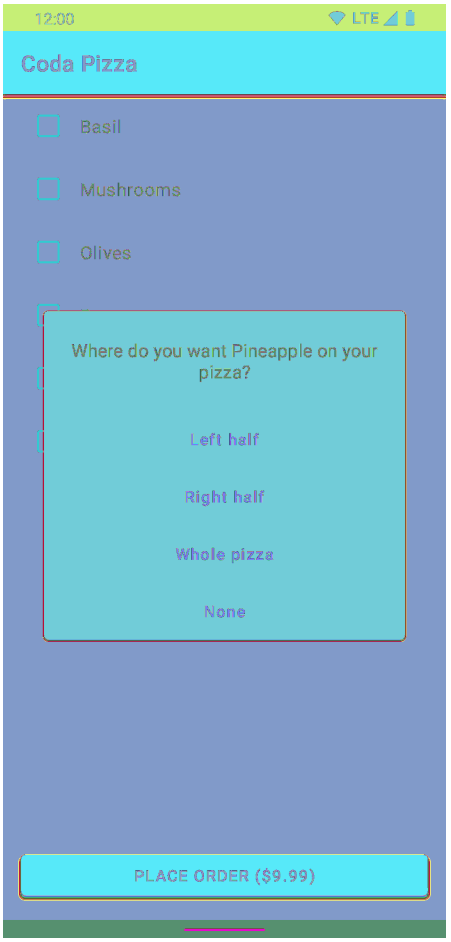
```
@Composable
fun ToppingPlacementDialog(
    topping: Topping,
    onDismissRequest: () -> Unit
) {
    Dialog(onDismissRequest = onDismissRequest) {
        Card {
            Column {
                val toppingName = stringResource(topping.toppingName)
                Text(
                    ...
                )

                ToppingPlacement.values().forEach { placement ->
                    ToppingPlacementOption(
                        placementName = placement.label,
                        onClick = { /* TODO */ }
                    )
                }

                ToppingPlacementOption(
                    placementName = R.string.placement_none,
                    onClick = { /* TODO */ }
                )
            }
        }
    }
}
...
```

Run Coda Pizza. When you click a topping, you will now be presented with the full list of options for placing the topping (Figure 28.4).

Figure 28.4 Do you want to build a pizza?



## Sending Results from a Dialog

Your final task is to wire up all the buttons in your dialog so they can correctly update the user's pizza. Currently, the options in this dialog do nothing more than offer confirmation that they were, in fact, pressed.

Each of the options in this dialog should do two things when pressed: It should notify the creator of the dialog which choice was selected, and it should dismiss the dialog. You already have everything you need to make the options dismiss the dialog. The `onDismissRequest` callback can be reused after selecting an option to indicate that the dialog is requesting to be dismissed.

Update your two empty `onClick` callbacks with this behavior.

### Listing 28.10 Dismissing the dialog (`ToppingPlacementDialog.kt`)

```
@Composable
fun ToppingPlacementDialog(
    topping: Topping,
    onDismissRequest: () -> Unit
) {
    Dialog(onDismissRequest = onDismissRequest) {
        Card {
            Column {
                ...
                ToppingPlacement.values().forEach { placement ->
                    ToppingPlacementOption(
                        placementName = placement.label,
                        onClick = { /* TODO */ }
                            onDismissRequest()
                        }
                    )
                }

                ToppingPlacementOption(
                    placementName = R.string.placement_none,
                    onClick = { /* TODO */ }
                        onDismissRequest()
                    }
                )
            }
        }
    }
}
...
```

Run Coda Pizza. Click any of the toppings and select any of the four placements. Although the pizza itself will not change, notice that the dialog is dismissed.

To update the pizza, you will again use the state hoisting pattern you used in the **`ToppingsList`**. **`ToppingPlacementDialog`** is not in control of the pizza state, but its parent is. To modify the state, **`ToppingPlacementDialog`** will need to take in another lambda to request a change to the pizza.

Add a function parameter called `onSetToppingPlacement`. This parameter will be a lambda that passes the selected **`ToppingPlacement`** value (or a null value, if None was selected). After you have this parameter in place, invoke it in each button's `onClick` callback before dismissing the dialog.

### Listing 28.11 Sending results back (`ToppingPlacementDialog.kt`)

```
@Composable
fun ToppingPlacementDialog(
    topping: Topping,
    onSetToppingPlacement: (placement: ToppingPlacement?) -> Unit,
    onDismissRequest: () -> Unit
) {
    Dialog(onDismissRequest = onDismissRequest) {
        Card {
            Column {
                ...
                ToppingPlacement.values().forEach { placement ->
                    ToppingPlacementOption(
                        placementName = placement.label,
                        onClick = {
                            onSetToppingPlacement(placement)
                            onDismissRequest()
                        }
                    )
                }

                ToppingPlacementOption(
                    placementName = R.string.placement_none,
                    onClick = {
                        onSetToppingPlacement(null)
                        onDismissRequest()
                    }
                )
            }
        }
    }
}
...
```

To use this returned value, you will also need to update **`ToppingsList`** to handle the topping placement selection. **`ToppingsList`** will then delegate to its `onEditPizza` callback so that **`PizzaBuilderScreen`** can commit the change.

### Listing 28.12 Handling the result (`PizzaBuilderScreen.kt`)

```
...
@Composable
private fun ToppingsList(
    pizza: Pizza,
    onEditPizza: (Pizza) -> Unit,
    modifier: Modifier = Modifier
) {
    var toppingBeingAdded by rememberSaveable { mutableStateOf<Topping?>(null) }

    toppingBeingAdded?.let { topping ->
        ToppingPlacementDialog(
            topping = topping,
            onSetToppingPlacement = { placement ->
                onEditPizza(pizza.withTopping(topping, placement))
            },
            onDismissRequest = {
                toppingBeingAdded = null
            }
        )
    }
    ...
}
...
```

Your dialog is now complete. Run Coda Pizza and customize a pizza to your heart's content. Spend a moment admiring the power of choice, and notice that your UI – including the PLACE ORDER button – automatically updates, all at once, for each topping you change.

This is the power of declarative programming in Jetpack Compose: You did nothing to tell any component onscreen to update, nor did you specify where a change would be coming from. But because you wrapped your values in **`State`** objects, Compose takes care of your UI updates, regardless of how or why the UI needs to change. And because dialogs are simply composables, you have the full flexibility to communicate directly to them without jumping through any hoops.

In the next chapter, you will finish your work on Coda Pizza by adding a pizza preview image and customizing some of the app's visual elements.

## Challenge: Pizza Sizes and Drop-Down Menus

For this challenge, you will expand the customization options available in Coda Pizza. Currently, it is only possible to order a pizza in a single size. You will change that by adding another UI element to prompt the user to choose a pizza size.

For some UI interactions, a dialog can be a bit intrusive. It forces your users to interact with a specific message and hides the rest of your application's UI. As an alternative, you can use a drop-down menu to show a set of options that blocks a much smaller portion of your UI. You saw a drop-down menu in Chapter 15 when menu items in your app bar were relegated into an overflow menu.

Creating a drop-down menu in Compose is similar to how you created a **`Dialog`**. A dropdown can be shown using the **`DropdownMenu`** composable. We have copied its signature below, and you can find its full documentation with the rest of the Material composables at `developer.android.com/reference/kotlin/androidx/compose/material/package-summary`.

```
@Composable
fun DropdownMenu(
    expanded: Boolean,
    onDismissRequest: () -> Unit,
    modifier: Modifier = Modifier,
    offset: DpOffset = DpOffset(0.dp, 0.dp),
    properties: PopupProperties = PopupProperties(focusable = true),
    content: @Composable ColumnScope.() -> Unit
)
```

There are two notable differences between how you use a **`DropdownMenu`** and how you use a **`Dialog`**. First, **`DropdownMenu`** specifies an `expanded` parameter, which controls whether the menu is expanded (visible) or collapsed (hidden). This means that you do not need to wrap your usage of **`DropdownMenu`** in an `if` statement, like you did with **`Dialog`**.

Second, where the **`DropdownMenu`** is drawn onscreen is directly affected by where it is placed in your composition hierarchy. A **`Dialog`** always fills the full size of your app and draws over all other composables. But **`DropdownMenu`** is *anchored* to its parent composable, meaning that it will appear in the same area of your screen as the composable that hosts the menu. It is conventional for menus on Android to expand outward from and on top of the UI element that caused the menu to appear. Keep this in mind when you are deciding where to nest your **`DropdownMenu`**.

Take **`DropdownMenu`** for a spin by adding a dropdown near the top of the screen that lets the user change their pizza's size. You will also find the **`DropdownMenuItem`** composable handy, for adding choices into your drop-down menu. Give your customers four size options: small, medium, large, and extra large. Smaller pizza sizes should be less expensive than larger pizzas, and your pizzas should be large by default. (You are free to decide Coda Pizza's exact prices for this challenge.)

You will need to define new UI state to track the selected size as part of this challenge. We recommend defining a new enum called **`Size`** to declare the size options and adding a size property to your **`Pizza`** data class to track the user's size selection.

# 29
# Theming Compose UIs

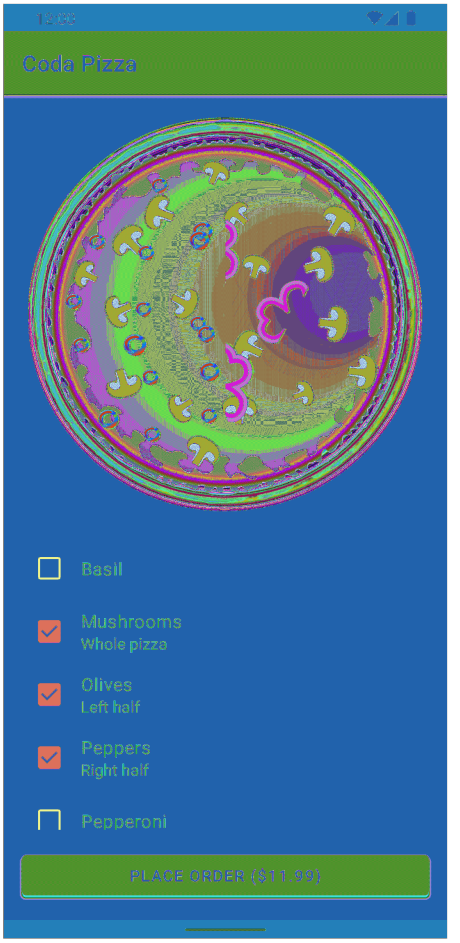
Coda Pizza's users can now customize their pizzas, but the app itself is still the equivalent of plain cheese: It gets the job done, but it has no particular personality. In the last chapter of this series on the basics of Jetpack Compose, you will spend some time polishing Coda Pizza and adding some visual flair.

You will start by adding a preview of the user's customized pizza. It will be placed above your list of toppings and will automatically update every time the user changes the toppings on their pizza. You will also learn how to specify a theme in Compose, which works differently than the application themes you have seen in a `themes.xml` file.

Finally, you will wrap up your journey through Jetpack Compose by reflecting on what Coda Pizza does *not* have compared to the other apps you have built in the past – and what Compose means for the future of Android development.

When you finish, Coda Pizza will look like Figure 29.1.

Figure 29.1 Coda Pizza's final form



## Images

Coda Pizza offers 4,096 combinations of toppings for your users to select. Creating a preview image for each combination is not feasible, so you will need to generate the preview for the user's pizza on the fly. You will do this by combining several images, layering them on top of each other to get the final result.

The preview will start with an image of a plain cheese pizza. For each topping the user adds, you will overlay an image of that topping on the base image. If a topping is only on half of the pizza, you will crop the image so that it only appears on the correct half.

Start by importing the images you will use to build your pizza previews. If you do not already have them, download the solutions to the exercises in this book from `www.bignerdranch.com/android-5e-solutions`. Unzip this archive and locate the solution for Coda Pizza in the `28. Theming Compose UIs/Solution/CodaPizza` folder. Navigate to `app/src/main/res/drawable` and copy the `.webp` files in that folder into the `app/src/main/res/drawable` folder of your project.

The images you just copied to your project are `.webp`s instead of `.xml` vectors, like you have used before. Vectors are a great choice for UI elements and simple designs, but not all images can be effectively expressed as a vector. Complex images and photographs, in particular, are either impossible to represent as vectors or can lead to performance issues if they are used as vectors.

Your pizza preview images fall into this category, so you need another format, like `.webp`. By the way, Android also supports other image formats in your resources, including `.png` and `.jpeg` files. (You might recall that you used a `.png` for PhotoGallery's placeholder image.) We have chosen `.webp` here because it offers smaller file sizes.

With your assets in place, you can begin to create the composable that will show pizza previews. Initially, you will only show the base image. Later, you will programmatically draw other toppings on top of this image.

To display images in Compose, you use the **`Image`** composable. When calling **`Image`**, you provide a **`Painter`** to specify the image you want to display. **`Painter`** is analogous to the **`Drawable`** class you have used before. It declares something that can be "painted" to the screen, like a vector image, bitmap image, solid color, or gradient.

You can obtain a **`Painter`** for one of your drawable resources by calling **`painterResource`**. Much like the **`stringResource`** function you saw earlier, **`painterResource`** will trigger Compose to take care of querying your resources, loading the right image, and converting it into a **`Painter`** that can be used with your **`Image`**.

Create a new file in the `ui` package called `PizzaHeroImage.kt`. (A *hero image* is a large image placed prominently at the top of a page.) Define a new composable called **`PizzaHeroImage`** that will show pizza previews. The **`PizzaHeroImage`** should have two arguments: a **`Pizza`** and the compulsory **`Modifier`** argument. Use **`Image`** to show the image of a plain pizza, and give your new composable a preview function to see your changes in Android Studio.

### Listing 29.1 Let me imagine it... (`PizzaHeroImage.kt`)

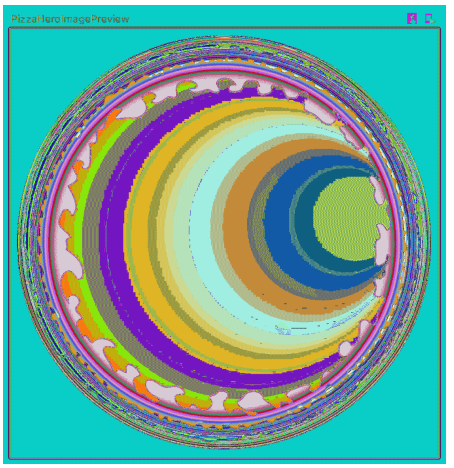
```
@Preview
@Composable
private fun PizzaHeroImagePreview() {
    PizzaHeroImage(
        pizza = Pizza(
            toppings = mapOf(
                Topping.Pineapple to ToppingPlacement.All,
                Topping.Pepperoni to ToppingPlacement.Left,
                Topping.Basil to ToppingPlacement.Right
            )
        )
    )
}

@Composable
fun PizzaHeroImage(
    pizza: Pizza,
    modifier: Modifier = Modifier
) {
    Image(
        painter = painterResource(R.drawable.pizza_crust),
        contentDescription = null,
        modifier = modifier
    )
}
```

Be sure to import **`Image`** from the `androidx.compose.foundation` package.

Change to the split view in your editor and build your project to update the preview. When the build completes, you should see the image of a plain pizza in the preview (Figure 29.2).

### Figure 29.2 Plain pizza



## Image’s contentDescription

When you added your **`Image`**, you also had to specify a `contentDescription`. Like the `android:contentDescription` XML attribute you learned about in Chapter 19, this argument is used for accessibility: It gives screen readers text to read out. But unlike `android:contentDescription`, this parameter is mandatory and must always be provided.

You set the `contentDescription` to `null` initially. But you should respect this parameter’s purpose and provide a content description for your image. Start by adding a string resource to describe the image.

### Listing 29.2 Defining a content description (`strings.xml`)

```
<resources>
    ...
    <string name="pizza_preview">Pizza preview</string>
</resources>
```

With the string resource in place, you can specify the content description of the **`Image`**.

### Listing 29.3 Describing the content (`PizzaHeroImage.kt`)

```
@Composable
fun PizzaHeroImage(
    pizza: Pizza,
    modifier: Modifier = Modifier
) {
    Image(
        painter = painterResource(R.drawable.pizza_crust),
        contentDescription = null,
        contentDescription = stringResource(R.string.pizza_preview),
        modifier = modifier
    )
}
```

Having a content description in place will not change your app’s appearance, but it does make your app more accessible to users who rely on screen readers.

## Adding more images

Next, you will work on stacking toppings on the base pizza to form the final preview. The first step is to track which image to draw when a topping is placed on a pizza.

Add a new property to your **`Topping`** enum called `pizzaOverlayImage`. This will keep track of which image to use for each topping. After you add the property, give each case of the enum a value to associate the toppings with their images.

### Listing 29.4 Associating the preview images (`Topping.kt`)

```
enum class Topping(
    @StringRes val toppingName: Int,
    @DrawableRes val pizzaOverlayImage: Int
) {
    Basil(
        toppingName = R.string.topping_basil,
        pizzaOverlayImage = R.drawable.topping_basil
    ),

    Mushroom(
        toppingName = R.string.topping_mushroom,
        pizzaOverlayImage = R.drawable.topping_mushroom
    ),

    Olive(
        toppingName = R.string.topping_olive,
        pizzaOverlayImage = R.drawable.topping_olive
    ),

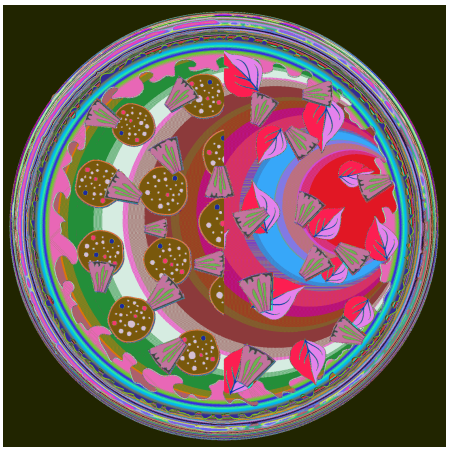
    Peppers(
        toppingName = R.string.topping_peppers,
        pizzaOverlayImage = R.drawable.topping_peppers
    ),

    Pepperoni(
        toppingName = R.string.topping_pepperoni,
        pizzaOverlayImage = R.drawable.topping_pepperoni
    ),

    Pineapple(
        toppingName = R.string.topping_pineapple,
        pizzaOverlayImage = R.drawable.topping_pineapple
    )
}
```

You are now ready to add toppings to your pizza previews. Ultimately, your layout preview will show toppings matching what you specified in **`PizzaHeroImagePreview`**: pepperoni on the left half, pineapple on the whole pizza, and basil on the right half (Figure 29.3).

Figure 29.3 Pizza preview goal



To do this, you will layer more **`Image`** composables – one for each topping on the pizza – on top of the base pizza **`Image`**.

Start by wrapping your base **`Image`** in a **`Box`**. Unlike the **`Column`** and **`Row`** composables, which place their content one after the other, the **`Box`** composable stacks its content. Then, use a `for` loop to add a new **`Image`** for each topping. For now, make each topping appear on the whole pizza.

Listing 29.5 An image for each topping (`PizzaHeroImage.kt`)

```
...
@Composable
fun PizzaHeroImage(
    pizza: Pizza,
    modifier: Modifier = Modifier
) {
    Box(
        modifier = modifier
    ) {
        Image(
            painter = painterResource(R.drawable.pizza_crust),
            contentDescription = stringResource(R.string.pizza_preview),
            modifier = modifier
        )

        pizza.toppings.forEach { (topping, placement) ->
            Image(
                painter = painterResource(topping.pizzaOverlayImage),
                contentDescription = null,
                modifier = Modifier.focusable(false)
            )
        }
    }
}
```

For your new **`Image`** composables, you use the **`focusable`** modifier to disable focus. This tells screen readers to ignore your topping images so that they do not see the pizza preview as multiple components. Because you disable focus, there is no need to specify a `contentDescription`.

Refresh your previews. You should see pepperoni, pineapple, and basil, all on the whole pizza, centered on top of the cheese. So far, so good.

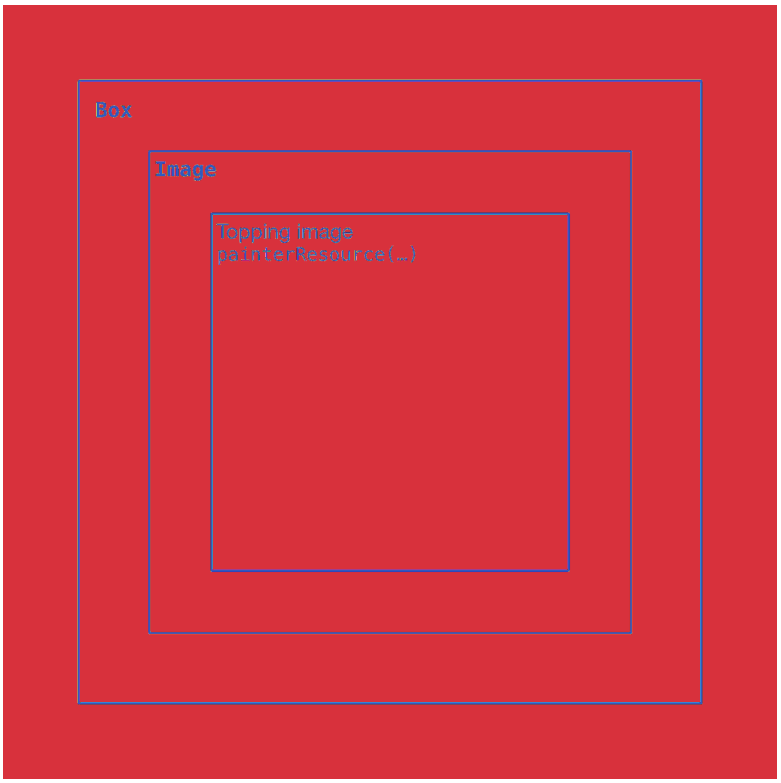
## Customizing the Image composable

Your topping **`Image`**s are all contained in the same **`Box`**. Because they are the same size, they are stacked on top of one another, resulting in what appears to be a single pizza image with the desired toppings. This setup works perfectly if your toppings are all placed on the entire pizza – but Coda Pizza does not limit users to full-pizza toppings. Your next task is to show only half of a topping layer when the topping is on half of the pizza.

Before you tackle this task, think about the structure of your topping images: Each image that is presented onscreen is drawn by a **`Painter`** and is hosted in an **`Image`** composable. The **`Image`** has its own bounds, as does the **`Painter`**. And, remember, the **`Painter`** *is* the topping image you display in your UI.

Figure 29.4 shows the relationship between your topping images and your **`Image`** composables. Keep this in mind as you tackle Coda Pizza's next feature, which will require customizing how your **`Image`** composables display their images.

Figure 29.4 Layers of pizza



Although the topping image's bounds are currently the same as its **`Image`** container's, this will need to change soon.

With the theory out of the way, it is time to make your previews accurately show toppings that are on only half of the pizza. At a high level, this will require four steps:

- Set the **`Image`** composable's size to the full height of the pizza and half of its width.
- Crop the image of your topping inside the **`Image`**'s bounds so that only half of the topping layer is visible.
- Align the topping image in the bounds of the crop to ensure that the correct half of the topping is visible.
- Arrange the **`Image`** composable so that it appears on the correct half of the pizza.

### aspectRatio

Start by setting the size of the topping **`Image`**. By default, an **`Image`**'s size is determined by the image being displayed. If you want to display an **`Image`** at a different size, you can use a modifier to alter its size. There are several modifiers that will do this, including the **`size`** modifier to set an exact size for the image.

But the size of the pizza preview will be dynamic, as it will fit the width of the user's device. So you do not want to hardcode the size of the image. Instead, you need to set the size of each topping layer relative to the size of its container. One way to accomplish this is with the help of aspect ratios.

Aspect ratio compares a rectangle's width to its height. The pizza preview has an aspect ratio of 1:1 – it is a perfect square, as wide as it is tall. Toppings that appear on half of the pizza will have the same height as the preview, but half of the width. This means that a topping image's aspect ratio should be 1:2 when it is on half of the pizza – it will be twice as tall as it is wide.

To set a composable's aspect ratio, use the **`aspectRatio`** modifier. If a topping is on half of the pizza, set its aspect ratio to 1:2. Aspect ratios are passed as floating point values instead of ratios, so the 1:2 aspect ratio is specified as `0.5`. For toppings placed on the entire pizza, set the aspect ratio to 1:1 by passing in `1.0`. To ensure your base pizza is always a perfect square, set its aspect ratio to 1:1 as well. Finally, make sure that the pizza crust always fills the full bounds of the **`PizzaHeroImage`** by adding the **`fillMaxSize`** modifier.

Listing 29.6 Setting aspect ratios (`PizzaHeroImage.kt`)

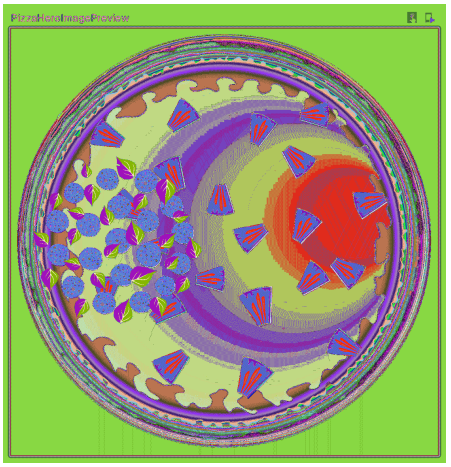
```
...
@Composable
fun PizzaHeroImage(
    pizza: Pizza,
    modifier: Modifier = Modifier
) {
    Box(
        modifier = modifier
            .aspectRatio(1f)
    ) {
        Image(
            painter = painterResource(R.drawable.pizza_crust),
            contentDescription = stringResource(R.string.pizza_preview),
            modifier = Modifier.fillMaxSize()
        )

        pizza.toppings.forEach { (topping, placement) ->
            Image(
                painter = painterResource(topping.pizzaOverlayImage),
                contentDescription = null,
                modifier = Modifier.focusable(false)
                    .aspectRatio(when (placement) {
                        Left, Right -> 0.5f
                        All -> 1.0f
                    })
            )
        }
    }
}
```

(To use your **`ToppingAlignment`** enum values without the `ToppingAlignment.` prefix, add an import for `com.bignerdranch.android.codapizza.model.ToppingPlacement.*` at the top of the file.)

Build your project to get an updated preview for PizzaHeroImagePreview. You will see that the pepperoni and basil toppings now appear at half of their original size, vertically centered and on the left half of the pizza (Figure 29.5).

Figure 29.5 Small toppings



Although the pepperoni and basil **`Image`** composables are, correctly, half their original size, they are scaling down their contents so that the entire image can be displayed. You will fix that next.

## contentScale

When the image you are displaying and the container that holds it do not have matching aspect ratios, Compose needs some strategy to handle the discrepancy. Here, the pepperoni and basil **`Image`**s have a ratio of 1:2, but the **`Box`** that contains them has a ratio of 1:1.

To tell Compose how to handle this difference, you set the **`Image`**'s `contentScale`. The default content scale is `Fit`, which scales the entire content image to fit the **`Image`**'s bounds, while preserving its original aspect ratio. This is not what you want, as you need to show the left or right half of the topping's image.

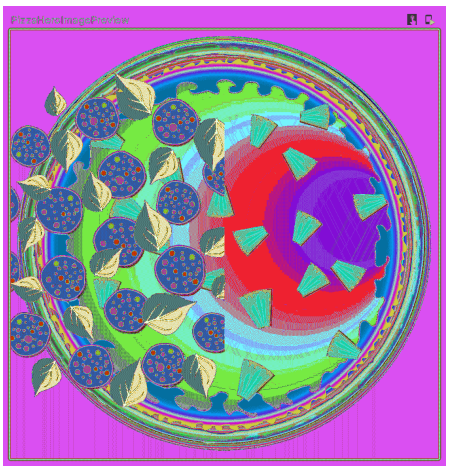
To change this behavior, you can add a `contentScale` argument when calling **`Image`**. Specifying the `Crop` behavior will scale the image to fit the bounds of the **`Image`**, cropping any excess that extends beyond the composable's bounds.

### Listing 29.7 Specifying a content scale (`PizzaHeroImage.kt`)

```
...
@Composable
fun PizzaHeroImage(
    pizza: Pizza,
    modifier: Modifier = Modifier
) {
    Box(
        ...
    ) {
        ...
        pizza.toppings.forEach { (topping, placement) ->
            Image(
                painter = painterResource(topping.pizzaOverlayImage),
                contentDescription = null,
                contentScale = ContentScale.Crop,
                modifier = Modifier.focusable(false)
                    ...
            )
        }
    }
}
```

Refresh the preview of PizzaHeroImagePreview. Now, the pepperoni and basil fill the height of the pizza preview – but they also extend off the edge of the pizza. Although your topping images are being cropped, you are not seeing their left halves – you are seeing their centers (Figure 29.6). This is because the images themselves (the **`Painter`**s) are centered within the bounds of their **`Image`** composables.

### Figure 29.6 Cropped toppings



### Image alignment

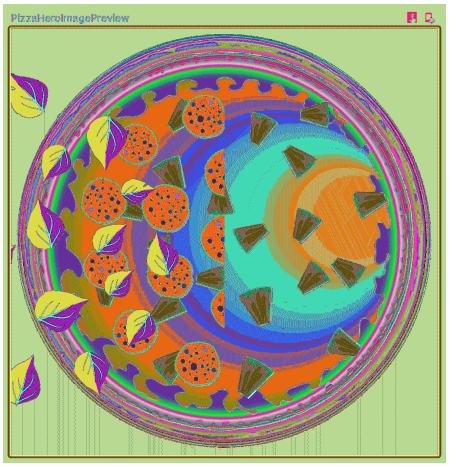
To change which portion of the image is shown and which portion gets cropped, you can set the `alignment` property on your **`Image`**. When a topping is placed on the left half of the pizza, you want to align the topping image's left edge with the left edge of the **`Image`**. You can do this by setting the **`Image`**'s alignment to `TopStart`, which aligns the top-left corner of its content with the top-left corner of the composable itself. Similarly, if a topping is on the right half of the pizza, you can use the `TopEnd` alignment to show the right half of the topping. If the topping is on both sides of the pizza, you can use the default `Center` alignment.

Listing 29.8 Aligning the image (`PizzaHeroImage.kt`)

```
...
@Composable
fun PizzaHeroImage(
    pizza: Pizza,
    modifier: Modifier = Modifier
) {
    Box(
        ...
    ) {
        ...
        pizza.toppings.forEach { (topping, placement) ->
            Image(
                painter = painterResource(topping.pizzaOverlayImage),
                contentDescription = null,
                contentScale = ContentScale.Crop,
                alignment = when (placement) {
                    Left -> Alignment.TopStart
                    Right -> Alignment.TopEnd
                    All -> Alignment.Center
                },
                modifier = Modifier.focusable(false)
                    ...
            )
        }
    }
}
```

Refresh the preview once again. The pepperoni is now properly placed on the pizza: It covers the left half of the pizza, without spilling over. But the basil is still misplaced. Although it is the correct size and shape to fill the right half, it is positioned over the left half of the pizza, spilling off the left edge of the crust (Figure 29.7).

Figure 29.7 Partially aligned toppings



### The align modifier

To place toppings on the right half of the pizza preview, you need to make one last change to your **`PizzaHeroImage`**. By setting the `alignment` parameter, you specified where you wanted the topping image to be painted inside the bounds of the **`Image`** composable. But the **`Image`** composable itself is always aligned to the top-left corner of its container, the **`Box`**.

To position the **`Image`** composable, you will need another tool: the **`align`** modifier. This modifier can be used to align the composable children of a **`Box`**. Much like the **`weight`** modifier you have used before, the **`align`** modifier is contextually available only when your content appears in a **`Box`**.

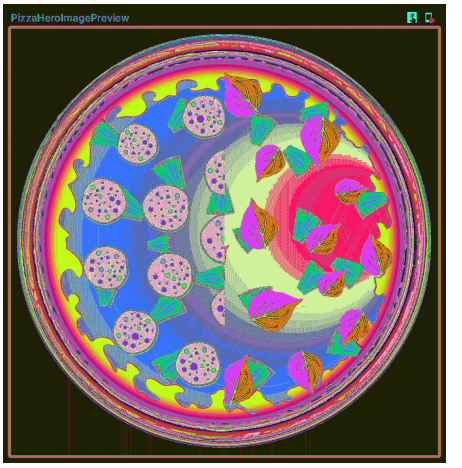
Set the alignment of your topping **`Image`** so that toppings on the right half of the pizza are aligned to the right edge of the **`Box`**, using the `CenterEnd` alignment. This will causes the image to appear at the end (right) of the **`Box`** and vertically centered. Although the toppings are already aligned correctly when placed on the left half or entire pizza, specify alignments for those cases as well – `CenterStart` and `Center`, respectively. This will allow you to build your modifier with a single fluent chain of function calls.

Listing 29.9 Aligning the toppings (`PizzaHeroImage.kt`)

```
...
@Composable
fun PizzaHeroImage(
    pizza: Pizza,
    modifier: Modifier = Modifier
) {
    Box(
        ...
    ) {
        ...
        pizza.toppings.forEach { (topping, placement) ->
            Image(
                painter = painterResource(topping.pizzaOverlayImage),
                contentDescription = null,
                contentScale = ContentScale.Crop,
                alignment = when (placement) {
                    Left -> Alignment.TopStart
                    Right -> Alignment.TopEnd
                    All -> Alignment.Center
                },
                modifier = Modifier.focusable(false)
                    .aspectRatio(when (placement) {
                        Left, Right -> 0.5f
                        All -> 1.0f
                    })
                    .align(when (placement) {
                        Left -> Alignment.CenterStart
                        Right -> Alignment.CenterEnd
                        All -> Alignment.Center
                    })
            )
        }
    }
}
```

Refresh your preview again. At last, the toppings on your pizza are accurately placed in the preview. You should see pineapple on the entire pizza, pepperoni on the left half, and basil on the right half. All the toppings should be correctly sized and vertically centered, and nothing should be outside of the crust (Figure 29.8).

Figure 29.8 The final preview



## Adding a header to LazyColumn

Your **`PizzaHeroImage`** is now complete, but you will not yet see the fruits of your labor if you run Coda Pizza. Allow your efforts to pay off by adding **`PizzaHeroImage`** as an item in your **`ToppingsList`**'s **`LazyColumn`**.

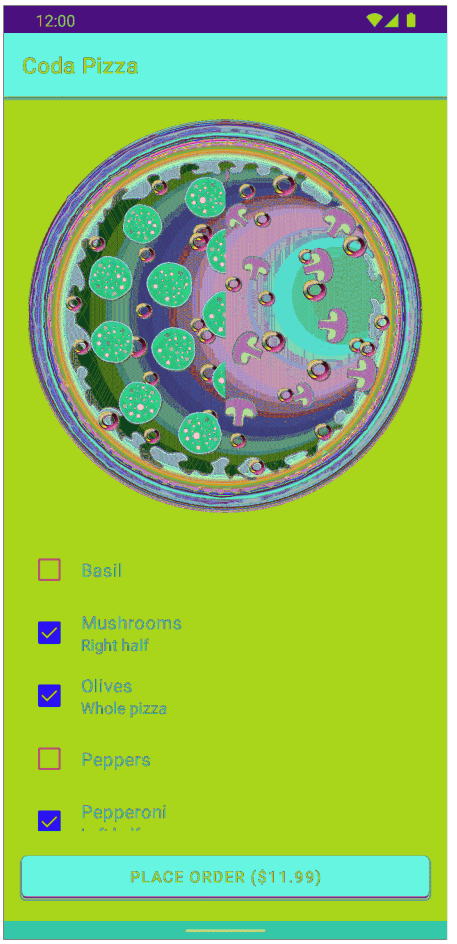
Listing 29.10 Adding more items to a LazyColumn (`PizzaBuilderScreen.kt`)

```
...
@Composable
private fun ToppingsList(
    pizza: Pizza,
    onEditPizza: (Pizza) -> Unit,
    modifier: Modifier = Modifier
) {
    ...
    LazyColumn(
        modifier = modifier
    ) {
        item {
            PizzaHeroImage(
                pizza = pizza,
                modifier = Modifier.padding(16.dp)
            )
        }

        items(Topping.values()) { topping ->
            ToppingCell(
                topping = topping,
                placement = pizza.toppings[topping],
                onClickTopping = {
                    toppingBeingAdded = topping
                }
            )
        }
    }
}
...
```

Run Coda Pizza. You will now see the pizza's preview proudly presented above your list of toppings. Add any toppings you want to the pizza. Because you have already tracked your **`Pizza`** object using the **`State`** class, your preview will automatically update, with no additional effort on your part (Figure 29.9). As you scroll through the list of toppings, the pizza preview will scroll with the other content.

Figure 29.9 Pizza preview in action




## MaterialTheme

With your pizza preview in place, it is time to add a fresh coat of paint to Coda Pizza. Currently, Coda Pizza is using the default theme, but you can add your own customizations to specify your application's colors, typographic styles, and shapes for various components like **`Button`**s and **`Card`**s. For Coda Pizza, you will stick to changing your app's colors, although the steps are similar for other customizations.

You saw themes for the first time in Chapter 11. The themes that you are familiar with were defined in XML and were used by your framework views for styling. But Compose has its own theming system that does not leverage XML styles or the theming system used by framework views.

Themes are stored in an object called `MaterialTheme`, which you used in Chapter 26 to set **`Text`** styles. Currently, you are using a default theme, which is where your application colors are coming from. Let's change that.

To change the values in your `MaterialTheme` object, you will use the **`MaterialTheme`** composable function. This function accepts parameters to change your theme's colors, typographic styles, and component shapes. The **`MaterialTheme`** composable also accepts a lambda expression, which is where your content will be placed. Any theme configuration you specify only affects content placed inside this lambda. This means that your theme should be specified very early in your composition.

Create a new file in your `ui` package called `AppTheme.kt`. This is where all of your theme information will be stored. In this file, create a new composable called **`AppTheme`**. This function will call **`MaterialTheme`**, passing all the theme attributes you want to use to customize Coda Pizza's appearance.

### Listing 29.11 Declaring a theme (`AppTheme.kt`)

```
@Composable
fun AppTheme(
    content: @Composable () -> Unit
) = MaterialTheme(
    colors = lightColors(
        primary = Color(0xFFB72A33),
        primaryVariant = Color(0xFFA6262E),
        secondary = Color(0xFF03C4DD),
        secondaryVariant = Color(0xFF03B2C9),
    )
) {
    content()
}
```

You set the `colors` property of your theme to be a palette with a light background and a few specific colors for Coda Pizza. There are several other colors you can specify, but you will rely on the defaults provided by **`lightColors`**. You also did not provide other styling information for your app's typography, so the defaults will be used.

For your theme to be used, you must wrap your application content in an **`AppTheme`** composable. Do so in **`MainActivity`**, right inside the **`setContent`** call, to ensure that the theme is applied to your entire application.

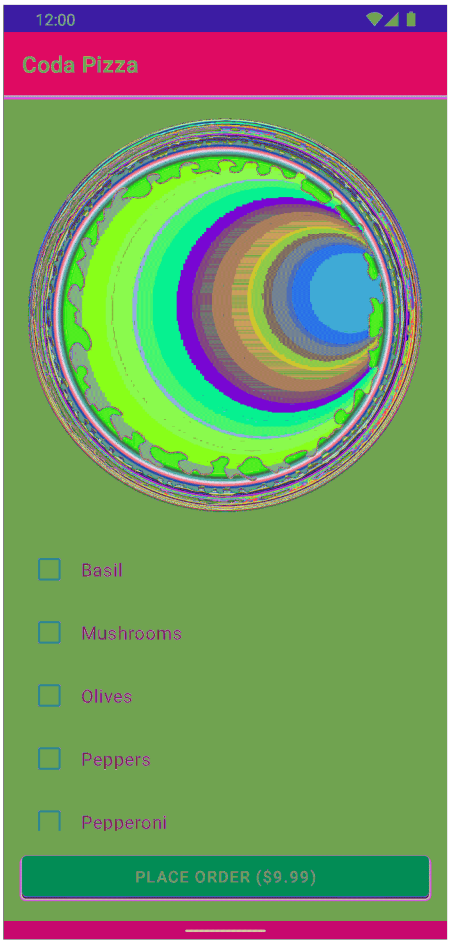
### Listing 29.12 Applying a theme (`MainActivity.kt`)

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            AppTheme {
                PizzaBuilderScreen()
            }
        }
    }
}
```

With that change, many UI elements in Compose are made aware of your theme. But you can also manually access these values in your composition through the `MaterialTheme` object. For example, to access your theme's primary color, you can call `MaterialTheme.colors.primary`.

Run Coda Pizza. Your PLACE ORDER button is now red, matching the value specified in your theme, but the app bar and status bar are still stubbornly purple, as shown in Figure 29.10. (We promise, the app bar and the button are different colors!)

Figure 29.10 A partially themed Coda Pizza




What gives?

Recall from Chapter 15 that your activities are automatically given an app bar when they extend from **`AppCompatActivity`** and use a style with a built-in app bar. Your **`MainActivity`** is doing just that. And because this app bar is provided as a framework view, it is unaware of any themes you created in your Compose code.

Although you could manually keep your Compose themes and framework themes in sync with one another, it would be better if your Compose theme were the source of truth for your application. Luckily, you can take the app bar into your own hands and render it using a composable. But before you can make this change, you need to remove the built-in app bar.

Speaking of removing things from your theme, there are a number of things in your project that you no longer need now that Compose is managing your styles. First, because your app's theme is defined in Compose, you do not need to specify theme attributes to define the same customizations. There are a few circumstances where you will still need to edit themes, even in 100% Compose apps – such as if you need to customize the color of system bars or set a custom splash screen for your app. But these issues will not come up for Coda Pizza.

Also, because your colors are defined entirely in your Compose code, Coda Pizza will not need its `colors.xml` file. Last, you do not need the Material Components library, since it only provides styling for framework views.

Start by tidying up your application theme. The project template you used to create Coda Pizza includes themes for both light mode and night mode. Jetpack Compose is in complete control of your application theme, so you do not need to provide a separate night mode theme for your application. Remove this theme variation by deleting the `res/values/themes/themes.xml` file labeled (night) in Android Studio.

Next, you need to remove the default app bar from your activity. To do this, you can change your theme to use a theme with the `NoActionBar` suffix. Use the AppCompat-provided `Theme.AppCompat` theme to remove your dependence on the Material Components library. At the same time, remove all the style declarations from your theme, which are unnecessary because these values are now set in your Compose theme.

### Listing 29.13 Removing framework styles (`themes.xml`)

```
<resources xmlns:tools="http://schemas.android.com/tools">
    <!-- Base application theme. -->
    <style name="Theme.CodaPizza"
        parent="Theme.MaterialComponents.DayNight.DarkActionBar">
    <style name="Theme.CodaPizza" parent="Theme.AppCompat.Light.NoActionBar">
        <!-- Primary brand color. -->
        <item name="colorPrimary">@color/purple_500</item>
        <item name="colorPrimaryVariant">@color/purple_700</item>
        <item name="colorOnPrimary">@color/white</item>
        <!-- Secondary brand color. -->
        <item name="colorSecondary">@color/teal_200</item>
        <item name="colorSecondaryVariant">@color/teal_700</item>
        <item name="colorOnSecondary">@color/black</item>
        <!-- Status bar color. -->
        <item name="android:statusBarColor" tools:targetApi="l">
            ?attr/colorPrimaryVariant</item>
        <!-- Customize your theme here. -->
    </style>
</resources>
```

Next, delete your `colors.xml` resource file. You will not need to access these colors, and you removed the only reference to them when you deleted your theme attributes.

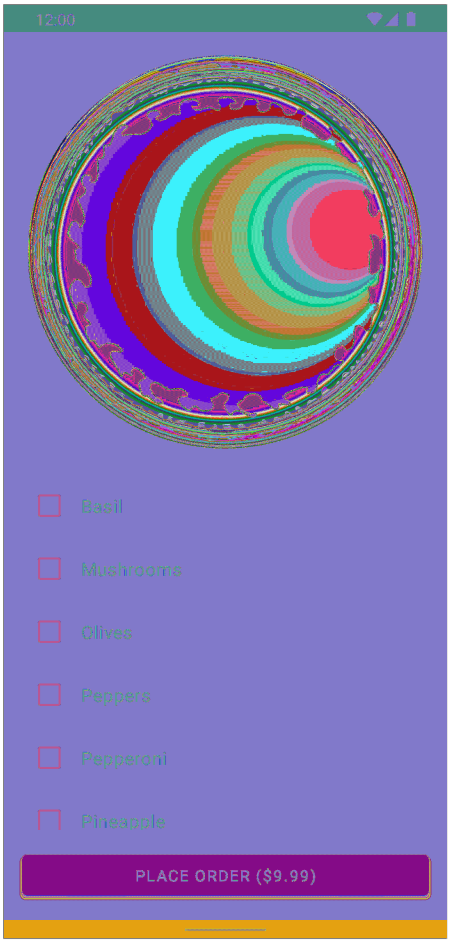
You no longer reference the Material Components library, so now you can remove it from your project. Removing unused dependencies reduces your application size, gets rid of unnecessary classes that clutter your IDE's autocompletion suggestions, and can improve compilation performance. Delete this dependency by taking a trip to your `app/build.gradle` file.

### Listing 29.14 Removing Material Components (`app/build.gradle`)

```
...
dependencies {
    implementation 'androidx.core:core-ktx:1.7.0'
    implementation 'androidx.appcompat:appcompat:1.4.1'
    implementation 'com.google.android.material:material:1.5.0'
    implementation 'androidx.constraintlayout:constraintlayout:2.1.3'
    ...
}
```

Remember to perform a Gradle sync after making these changes. When the sync completes, run Coda Pizza. The app bar has disappeared, and your composable content is now the only content that appears onscreen (Figure 29.11).

### Figure 29.11 A full-screen composable



## Scaffold and TopAppBar

App bars are an important part of your app, both visually and for navigation and menus. It is a good idea for Coda Pizza to include an app bar, even if it is just to show a title.

To reinstate your app bar, you will use the **`TopAppBar`** composable. **`TopAppBar`** accepts several composables as inputs, each of which can add content in a specific region of the app bar. You will only set the `title` parameter to show the application name, matching the default behavior of the automatically provided app bar – but with the benefit of using your Compose theme.

If you wanted, you could also provide a parameter for a `navigationIcon` to add an element to be the Up button. There is also a parameter called `actions`, which takes on the role of adding items to the right side of the app bar – like what you accomplished with a menu in the framework-provided app bar.

This pattern of accepting several composable lambdas as inputs is called *slotting*, as each of the components slots into a specific area of the app bar. Slotting makes composables much more flexible about what they can display than most framework views. You can pass any composable into any slot, which gives you complete control over what appears in each slot.

In the framework UI toolkit, you were limited to displaying a string as the title. And when you add a **`TopAppBar`**, you will typically pass a **`Text`** in for the `title` slot. But because the `title` argument takes in a composable, you have the power to use elements like an image, loading spinner, drop-down menu, or checkbox within that space – to name a few examples.

Take the **`TopAppBar`** composable for a spin, placing it at the top of your **`PizzaBuilderScreen`** composable.

### Listing 29.15 Adding an app bar (`PizzaBuilderScreen.kt`)

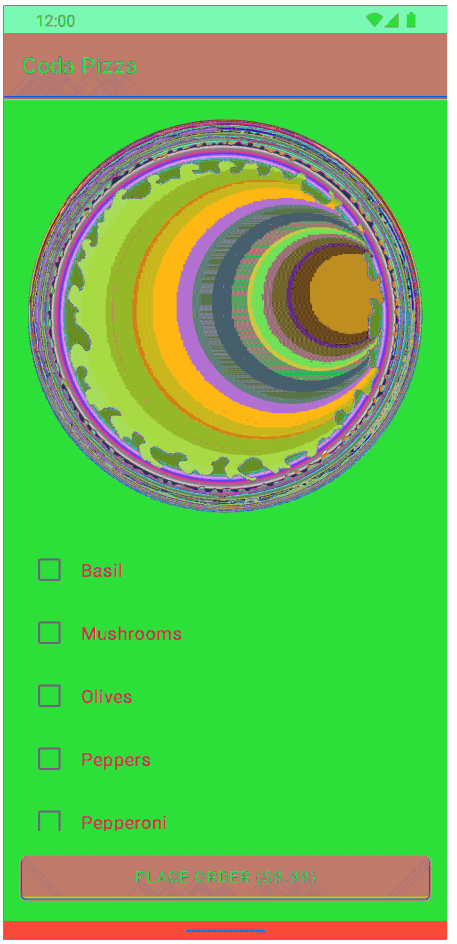
```
@Preview
@Composable
fun PizzaBuilderScreen(
    modifier: Modifier = Modifier
) {
    var pizza by rememberSaveable { mutableStateOf(Pizza()) }

    Column(
        modifier = modifier
    ) {
        TopAppBar(
            title = { Text(stringResource(R.string.app_name)) }
        )

        ToppingsList(
            ...
        )
        ...
    }
}
...
```

Run Coda Pizza. An app bar is back at the top of the screen – much like the one you previously had in Coda Pizza and the ones you have seen in the other apps you have built. But this time, the app bar is red, matching the color you set in your **`AppTheme`** composable (Figure 29.12).

Figure 29.12 TopAppBar and AppTheme in action




Your app bar is now functional, which is good. What is not so good is that **`PizzaBuilderScreen`**'s **`Column`** now has several elements in it. As your application grows in complexity, it can become harder to work with these large building blocks for your application's UI. Take the **`TopAppBar`**, for example. It *must* come first in this column if it is to be drawn at the top of the screen. If you accidentally add another composable before it, your app bar will appear toward the middle of the screen.

You can make the components in your UI easier to manage using another composable called **`Scaffold`**. **`Scaffold`** is designed to help lay out your application – it effectively acts as a skeleton for your app's layout. It uses the slotting pattern to define regions of your application where content can be labeled and consistently placed. There are two main slots you are interested in: the `topBar` slot and the `content` slot.

The `topBar` slot is designed for components like your **`TopAppBar`**. Composables placed in this slot always appear at the top of the screen, above your main content. The `content` slot, meanwhile, is for your app's primary content. There are other slots for elements like bottom bars and snackbars, each of which will always appear appropriately around your `content`.

Update **PizzaBuilderScreen** to use a **Scaffold**. You will still use a **Column** to lay out your toppings list and order button, but the **Scaffold** will separate the app bar from the content.

Listing 29.16 Using a scaffold (PizzaBuilderScreen.kt)

```
@Preview
@Composable
fun PizzaBuilderScreen(
    modifier: Modifier = Modifier
) {
    var pizza by rememberSaveable { mutableStateOf(Pizza()) }

    Column(
        modifier = modifier
    ) {
    Scaffold(
        modifier = modifier,
        topBar = {
            TopAppBar(
                title = { Text(stringResource(R.string.app_name)) }
            )
        },
        content = {
            Column {
                TopAppBar(
                    title = { Text(stringResource(R.string.app_name)) }
                )

                ToppingsList(
                    ...
                )

                OrderButton(
                    ...
                )
            }
        }
    )
}
```

Run Coda Pizza again. The app will look and behave the same, but your **TopAppBar** now has a guaranteed, designated space. **Scaffold** follows its own blueprints to position the content of its slots, so it does not matter how you specify the content of a slot or what order you put them in. The argument you provide for `topBar` appears at the top of the screen – always.

# CompositionLocal

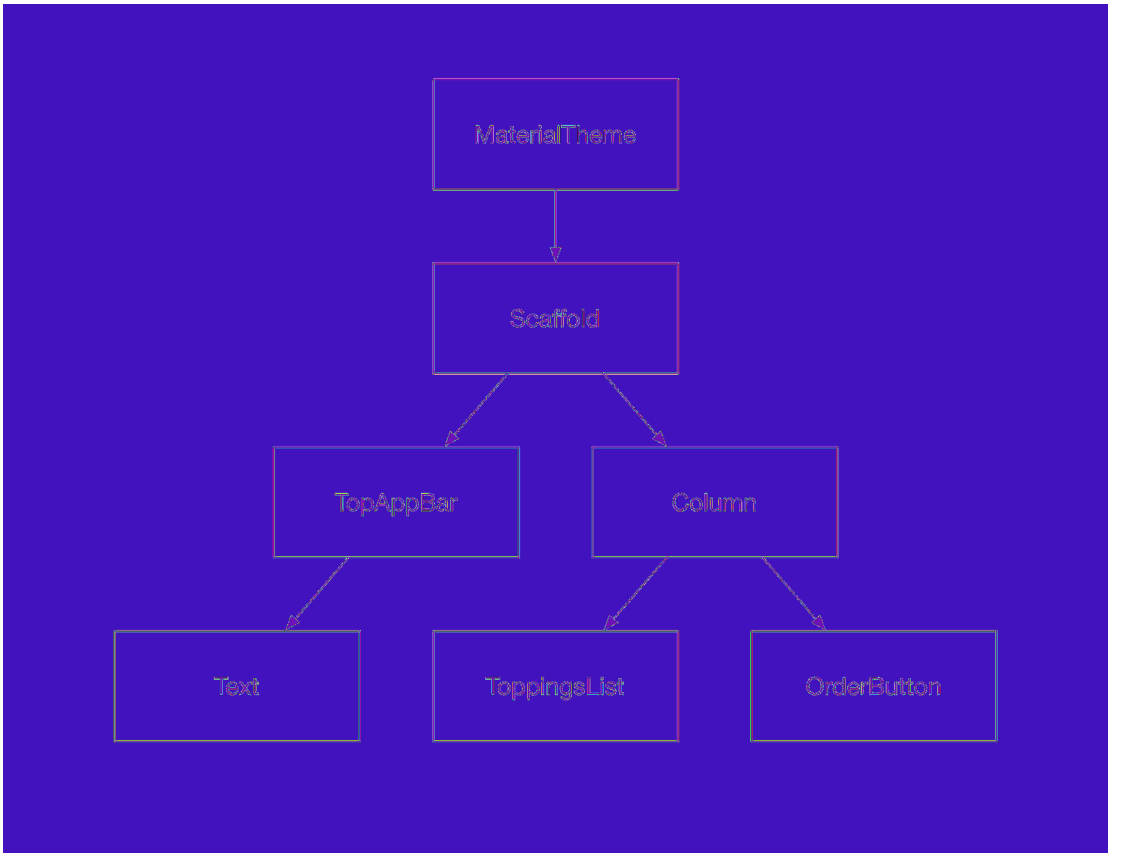
In this chapter, you have seen a few examples where nesting one composable inside another causes the inner composable's appearance to change. Using the **MaterialTheme** composable, for example, causes all components nested within it to be aware of your theme and use its colors.

When you added the **Text** to your **TopAppBar**, you might have noticed something interesting. Despite the fact that you did not specify any parameters besides the `text` itself, the **Text** was appropriately styled with the correct font size and color to appear in your app bar. How did this happen?

When you build a UI using Jetpack Compose, a composition hierarchy is created at runtime for all your composables – much like the view hierarchy for framework UIs. Whenever you call a composable function to add a UI element to your app, a corresponding node is added to this composition hierarchy. You cannot get a reference to this hierarchy or its nodes, but they are still there to organize your composables. (Whereas **`View`** objects are themselves the nodes in a framework UI's hierarchy, and you can get direct access to them whenever you like.)

For Coda Pizza, the top of the composition hierarchy looks something like Figure 29.13:

Figure 29.13 Coda Pizza's UI hierarchy




And **`ToppingsList`** and **`OrderButton`** have their own children, down to the **`Text`**s and **`Checkbox`** seen onscreen.

In Chapter 11, we warned you that nesting view hierarchies too deeply could degrade an app's performance. But although Coda Pizza's UI hierarchy is many layers deep, fear not. Jetpack Compose is substantially more efficient than Android's framework UI toolkit when it comes to managing, laying out, and drawing UI elements. Deeply nested layouts are not a performance concern with Compose in the way they were for the other apps you built – which is also why you did not reach for a tool like **`ConstraintLayout`** in Coda Pizza.

The top of this hierarchy is your **`MaterialTheme`** composable. As you have seen in this chapter, using the **`MaterialTheme`** composable sets the values that will be returned by the `MaterialTheme` object. And these values have a scope.

When you call the **`MaterialTheme`** composable, the theme values are stored and made accessible for all its children. Whenever a component needs to access a theme attribute, you use the `MaterialTheme` object and access the colors, shapes, or typography with code like `MaterialTheme.colors.primary`. Values referenced in this way are tracked by instances of a class called **`CompositionLocal`** so that every composable that is a child of the **`MaterialTheme`** node can access your theme information.

You placed the **`MaterialTheme`** composable at the root of your composition because you want the theme to affect everything you display in your composition. If you added a composable as a sibling to the **`MaterialTheme`**, it would be unaware of the themes you set elsewhere in the UI hierarchy and would use the default material theme.

If for some reason you want to have different themes for different parts of your composition, you can also nest one **`MaterialTheme`** composable inside another. The inner theme will override the theme values from the outer **`MaterialTheme`** composable, but only for the children of the inner **`MaterialTheme`**.

Back to the question of where the style of the **`Text`** in your **`TopAppBar`** is coming from. In addition to the theme attributes set in the `themes.xml` resource file and **`AppTheme`** composable (if you have one), some composables also specify preferred theme attributes that Compose will take into account. (These are officially termed *current* theme attributes, but we find that term unnecessarily confusing.)

For example, while your application has text color specifications, the **`TextButton`** composable you are using for the topping placement options in your dialog overrides this color specification, setting a style that works with the overall theme but is specific to its own environment. When the **`Text`** child of your **`TextButton`**s asks for a text color, it gets this overridden value, not the value from your theme.

**`TopAppBar`** does the same thing to its text, setting a color that works with the background color set by the app theme. In this way, "current" theme attributes like the ones set by **`TextButton`** and **`TopAppBar`** provide automatic localized theming that coordinates with the app's overall theme. The **`Text`** composable determines its styles by first looking at the text size, color, and so on set by its parent (or another direct ancestor) and then falling back to the theme for any styles that are not set.

Behind the scenes, these behaviors are all driven by the same **`CompositionLocal`** class we mentioned earlier.

`CompositionLocal`s are variables that are defined for a part of your composition hierarchy. When a `CompositionLocal` is defined, it is accessible to all of its composable children – and can be overridden deeper in the hierarchy if the same `CompositionLocal` is set again. Theme information is often defined this way – many children need to share and access theme information, and **`CompositionLocal`**s make that sharing easy.

Instances of **`CompositionLocal`** propagate theme information automatically. But you can also access `CompositionLocal` variables yourself to get many more values and resources associated with your composition. To see this in action, it is time to implement one last feature in Coda Pizza: the PLACE ORDER button.

Unfortunately, Coda Pizza will not result in a pizza being delivered to your address. But it can present you with a **`Toast`** (which, arguably, has many similarities to pizza). To set one up, you will replace your final TODO, which is lingering in **`OrderButton`**'s `onClick` callback.

First, prepare Coda Pizza to display a toast by adding a string resource for the message that will appear when the order is placed.

### Listing 29.17 A consolation toast's message (`strings.xml`)

```
<resources>
  <string name="app_name">Coda Pizza</string>

  <string name="place_order_button">Place Order (%1$s)</string>
  <string name="order_placed_toast">Order submitted!</string>
  ...
</resources>
```

To show the toast, you need to obtain a **`Context`**. You could accomplish this by adding a `context` parameter to **`OrderButton`** and passing your activity context all the way down your composition hierarchy, but that would be messy and would not scale well if you needed to access many properties.

Instead, Compose includes a `CompositionLocal` out of the box that stores the context that hosts your composable UI. You can use this `CompositionLocal` to access the context regardless of where you are in the composition.

To read the value of a `CompositionLocal` variable, you first obtain a reference to the corresponding **`CompositionLocal`** class itself. Then you can get the value of the variable for the current position in the composition hierarchy via its `current` property. The convention for naming a `CompositionLocal` is to use the prefix "Local" followed by the name or type of the variable being provided. So the composition's local context is stored in `LocalContext`.

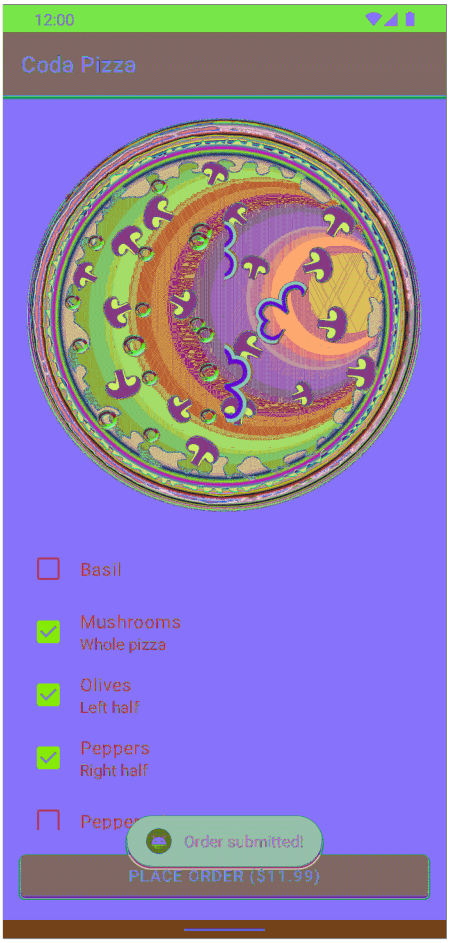
Using the `LocalContext` property, obtain a **`Context`**. Then, implement your **`OrderButton`**'s `onClick` lambda to show a toast. Because `CompositionLocal`s give you the *current* value of the variable, they can only be read inside the composition itself. This means that you must obtain the context outside the click listener, because your click listener cannot access the composition hierarchy.

### Listing 29.18 Using a **`Context`** inside a composable (`PizzaBuilderScreen.kt`)

```
...
@Composable
private fun OrderButton(
    pizza: Pizza,
    modifier: Modifier = Modifier
) {
    val context = LocalContext.current
    Button(
        modifier = modifier,
        onClick = {
            // TODO
            Toast.makeText(context, R.string.order_placed_toast, Toast.LENGTH_LONG)
                .show()
        }
    ) {
        val currencyFormatter = remember { NumberFormat.getCurrencyInstance() }
        val price = currencyFormatter.format(pizza.price)
        Text(text = stringResource(R.string.place_order_button, price))
    }
}
```

Run Coda Pizza and press the PLACE ORDER button. You should see a toast with the message Order submitted! near the bottom of the screen (Figure 29.14).

Figure 29.14 A toast to pizza




`CompositionLocal`s are a convenient way to get more information about your composition. Many `CompositionLocal`s are predefined and readily available should you need them. Some of these values, like your theme, allow you to easily customize portions of your UI hierarchy. Other `CompositionLocal`s, meanwhile, have values that do not change in the composition hierarchy and can give you information about the composition itself.

Using the built-in `CompositionLocal`s, you can access values including the **`Lifecycle`** of the component hosting your composition, the clipboard, and the size of the display. And because `CompositionLocal`s are tracked by Compose itself, you can access all these values without declaring new parameters. This flexibility makes `CompositionLocal`s a great choice for storing information you need to access sporadically throughout your UI.

You can also define your own `CompositionLocal`s, if you need to. This is not something you will do for Coda Pizza, but if you want to know more about this process, take a look at the section called For the More Curious: Creating Your Own CompositionLocals near the end of this chapter.

## Removing AppCompat

Coda Pizza is now fully operational, but there is one more change you can make to look toward the future. Every app you have built so far has relied on several Jetpack libraries, the most fundamental being the AppCompat library.

AppCompat back-ports many important UI behaviors to ensure consistency across versions of Android. It acts as the building block for many other Jetpack libraries, including ConstraintLayout and the Material Design Components. It also brings along many other tools you have used, including **`Fragment`**s.

Despite the importance of these components in the other apps you have built, these dependencies are designed for the world of the framework UI toolkit. Compose does not depend on AppCompat; it reinvents so many APIs that AppCompat does not provide the same value as it does for apps with framework views. In fact, AppCompat arguably does not provide *any* value if your UI exclusively uses Compose.

But AppCompat is still present in your application, increasing its size and adding more dependencies to download when your project builds. You can reclaim these resources and part ways with the framework views entirely by removing AppCompat from your project.

But do not jump straight to your `build.gradle` file to delete the dependency. There are still a few references to AppCompat you must remove first.

The first reference to AppCompat that you need to remove is in your **`MainActivity`**. Your **`MainActivity`** extends from **`AppCompatActivity`**, following the recommendation for all apps using the framework UI toolkit. In addition to back-porting behaviors to older versions of Android, **`AppCompatActivity`** also provides hooks that other Jetpack libraries, like ViewModel, require.

If you replace **`AppCompatActivity`** with the platform-provided **`Activity`** class, you will lose the ability to use several other Jetpack libraries, which is not ideal. Instead, you can use **`ComponentActivity`**, which exists in the middle ground between the base **`Activity`** class and the full-fledged **`AppCompatActivity`** class.

**`ComponentActivity`** exists outside AppCompat and provides hooks so that other libraries that need deeper access to your activity, such as the AndroidX Lifecycle library and ViewModel, can do what they need to do. Using **`ComponentActivity`** allows these integrations to continue working, while removing your dependence on the AppCompat library.

To migrate to **`ComponentActivity`**, update your **`MainActivity`** class to change which variation of **`Activity`** it extends from. Also, delete the import statement for **`AppCompatActivity`**.

### Listing 29.19 Removing AppCompatActivity (`MainActivity.kt`)

```
~~import androidx.appcompat.app.AppCompatActivity~~
...
class MainActivity : ~~AppCompatActivity()~~ ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            AppTheme {
                PizzaBuilderScreen()
            }
        }
    }
}
```

There is one final usage of AppCompat lingering in your project. It is in the application theme you specify for Coda Pizza. To remove this reference, you will need to change the theme that your application theme, `Theme.CodaPizza`, is based on.

AppCompat themes provide many customizations to the built-in themes provided by the platform to ensure consistency and to bring new features to the views you can use. But these benefits only apply to framework views, which are nowhere to be found in Coda Pizza.

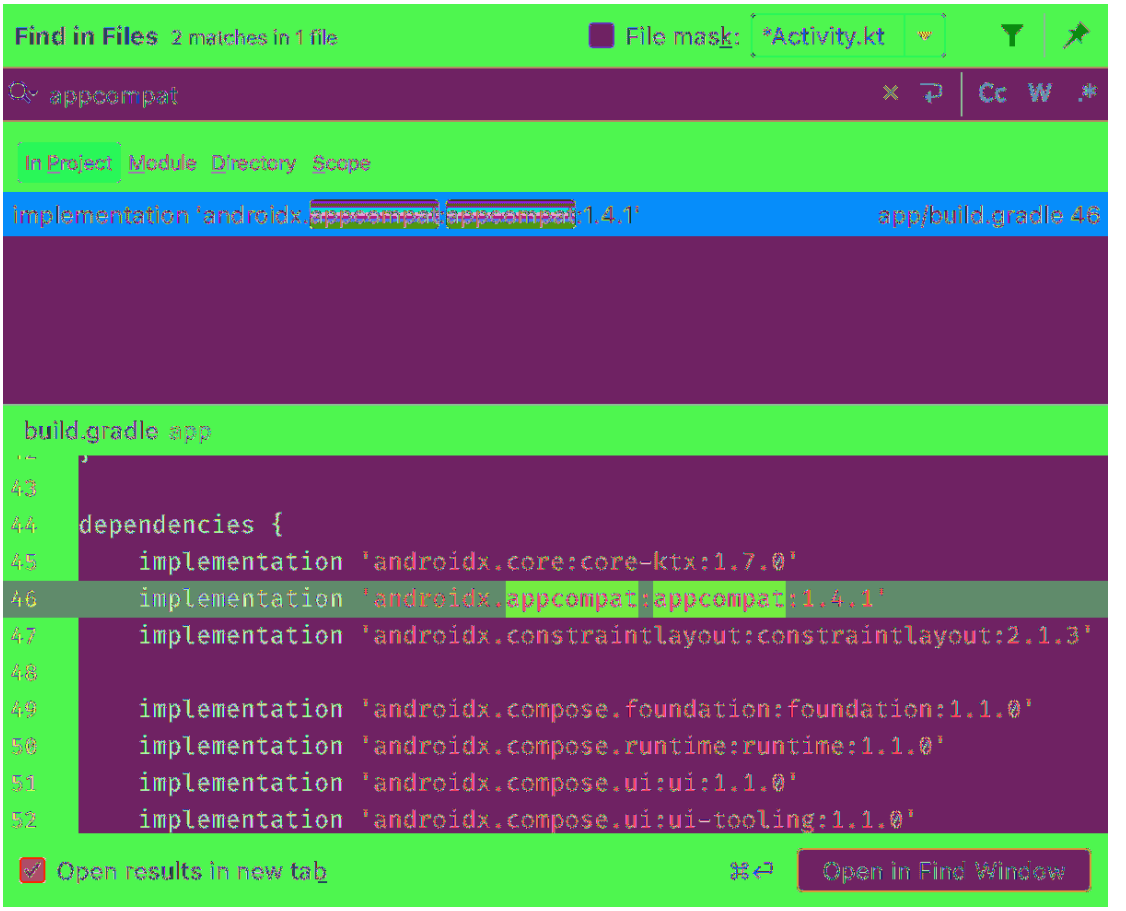
Because these customizations are unnecessary, you can safely remove the reference to `Theme.AppCompat` and replace it with a reference to the `Theme.Material` theme that ships with the platform. Although there may be discrepancies in this theme's appearance across versions, nothing in the theme will affect your Compose UI, making the differences negligible.

### Listing 29.20 Using a platform theme (`themes.xml`)

```
<resources xmlns:tools="http://schemas.android.com/tools">
    <!-- Base application theme. -->
    <style name="Theme.CodaPizza" parent="Theme.AppCompat.Light.NoActionBar">
    <style name="Theme.CodaPizza" parent="android:Theme.Material.NoActionBar">
    </style>
</resources>
```

At this point, Coda Pizza no longer references anything from the AppCompat library. To confirm this, open the Find in Files dialog by pressing Command-Shift-F (Ctrl-Shift-F). In the dialog, enter the query appcompat to search every file in your project for this term (Figure 29.15).

Figure 29.15 Looking for AppCompat




You will see one hit in your Gradle build file, but none of your Kotlin files should contain references. If you see any Kotlin files in these results, double-check that the code in these files matches the code in this book exactly, with no leftover references to AppCompat. You may also need to delete a few lingering import statements in your project if Android Studio did not automatically remove them.

After cleaning up any rogue references to AppCompat, your last step is to remove the venerable dependency from your project. While you are removing AppCompat's dependency, also remove the dependency for ConstraintLayout, which was automatically added with the blank project template.

### Listing 29.21 Saying goodbye to AppCompat (`app/build.gradle`)

```
...
dependencies {
    implementation 'androidx.core:core-ktx:1.7.0'
    implementation 'androidx.appcompat:appcompat:1.4.1'
    implementation 'androidx.constraintlayout:constraintlayout:2.1.3'
    ...
}
```

Sync your changes, then run Coda Pizza one last time. It should behave exactly as it did before – but now, Android Studio can build your project ever so slightly faster, and your application size will be smaller.

Although the benefits of removing AppCompat are somewhat small, this change signifies the start of a major paradigm shift powered by Jetpack Compose. Under Jetpack Compose, many of the tools you have become familiar with are no longer necessary. Components like **`RecyclerView`** and **`Fragment`** can be big sources of complexity in an Android app, but you do not need them in Compose.

We love Jetpack Compose for its ease of use and effectiveness at building UIs in Android apps. You have only scratched the surface of what Jetpack Compose can do, and we encourage you to experiment with it in your own apps. You will almost certainly encounter and create many framework views in your time as an Android developer, because they have been the only way to build UIs for more than a decade. But we expect that you will see fewer and fewer framework UIs over time as Compose brings about a renaissance in the world of Android development.

## For the More Curious: Accompanist

Jetpack Compose is stable, but still in its early days. For many apps, the Compose dependencies you used with Coda Pizza offer every component and API you need to build the UI of your dreams. For other apps, though, you might find that the Compose libraries do not *quite* have all the features you need.

Jetpack Compose version 1.1, for example, does not offer a built-in way to change the status bar or navigation bar colors, request permissions from a composable, or make your UI respond to display cutouts and notches, to name a few examples. But fear not: In addition to the mainstream Compose APIs, Google also offers a set of libraries to provide this functionality and more.

Accompanist is a constantly evolving set of libraries that offer functionality that is not yet built into the mainstream Compose dependencies. They offer a quick way for developers to access these features in their Compose apps. The goal for most Accompanist libraries is that they will eventually graduate out of Accompanist and into the official library they are part of.

For example, Accompanist previously included support for loading images with Coil in Compose, but that functionality has since been moved into Coil itself. Developing features this way allows the Compose team to more effectively design and experiment with these APIs.

Because of these evolutions, Accompanist leans toward being an experimental library. Regardless, we encourage you to take a look at Accompanist and see which of its features are useful for your app. These features are ready to go, and – despite the "experimental" designation – ready for use in production apps.

If you do choose to incorporate Accompanist, keep in mind that its APIs are likely to change over time. Features in Accompanist that make their way into the official Compose dependencies will eventually be removed, which will require you to make updates in your application.

It is too early to tell what the future of Compose looks like, but it looks like Accompanist will be a useful breeding ground for supporting a larger set of features within Compose. For more information on Accompanist, including its latest version and which features it can offer, see its official documentation at `google.github.io/accompanist`.

# For the More Curious: Creating Your Own CompositionLocals

In this chapter, you learned about several built-in `CompositionLocal`s and used the `LocalContext` to obtain a **`Context`** in your composable. If you want, you can also define your own `CompositionLocal`s.

Declaring a `CompositionLocal` is particularly useful when you want to give your composables access to new values without introducing an additional parameter. This works best when the information being provided applies to many composables and can be shared across an entire section of your composition hierarchy. Much like global variables, `CompositionLocal`s can be dangerous if used haphazardly. If a value should only be available to one composable, we recommend sticking to parameters.

Suppose your application needed to track analytics to see what features your users rely on most. You could create a class called **`AnalyticsManager`** to implement the analytics logging yourself. But many composables would likely need to report analytics, and you do not want to concern yourself with passing instances of **`AnalyticsManager`** through layer after layer of composables. That is where `CompositionLocal`s come in.

Making a `CompositionLocal` is a two-step process. First, you need to define the `CompositionLocal`. Second, you need to set the value of the `CompositionLocal` in your UI hierarchy.

`CompositionLocal`s are defined by creating a public, file-level property of type **`CompositionLocal`** – like, say, a `LocalAnalyticsManager`. This value basically acts as a key to get hold of the corresponding value. You can assign a value for this property using the **`compositionLocalOf`** function.

This function takes in a lambda to provide a default value for the `CompositionLocal`. For many `CompositionLocal`s, including your hypothetical `LocalAnalyticsManager`, there is no default value – a value must always be explicitly set in the composition itself. In these cases, you can simply throw an exception indicating that the `CompositionLocal` was read before it was set.

```
val LocalAnalyticsManager = compositionLocalOf<AnalyticsManager> {
    error("AnalyticsManager not set")
}
```

With the `CompositionLocal` defined, you can then specify its value at runtime. You do this using the **`CompositionLocalProvider`** composable. **`CompositionLocalProvider`** takes in a set of all the `CompositionLocal`s you want to specify, along with a value for each one. When a component requests one of the `CompositionLocal`s in the provider, the value you specify will be returned.

```
@Composable
fun PizzaBuilderScreen(
    analyticsManager: AnalyticsManager,
    modifier: Modifier = Modifier
) {
    CompositionLocalProvider(
        LocalAnalyticsManager provides analyticsManager
    ) {
        Scaffold(
            modifier = modifier,
            ...
        )
    }
}
```

You may be asking, "OK, but how does **PizzaBuilderScreen** obtain its analyticsManager?" This question has several answers, and in your own code, you will have to decide for yourself how to answer this question. If **AnalyticsManager** is easy to create, you may be able to instantiate it directly inside **PizzaBuilderScreen**. (If you do this, make sure to **remember** it!)

Alternatively, you could create this value elsewhere in your application – possibly as a singleton – and pass it through your composition hierarchy as an argument. Either approach is valid, and it is up to you and your team to decide how dependencies like analyticsManager should make their way through your code.

With the CompositionLocal and its provider in place, LocalAnalyticsManager is ready to be used. To obtain an **AnalyticsManager**, you call LocalAnalyticsManager.current inside a composable function. The Compose runtime will look at your composition hierarchy to find an appropriate provider for this value. Once a provider is found, the value that was set will be returned by the CompositionLocal.

If several providers are found, the closest parent in the hierarchy will be chosen and its value will be used. If no provider is found, the default value of the CompositionLocal (specified with the **compositionLocalOf**) will be provided.

Storing values this way allows for easy access throughout your composition hierarchy, but we recommend using CompositionLocals sparingly. They are great for accessing more general or widely used dependencies. But you can find yourself getting into trouble if you hold your application state in a CompositionLocal, as this makes it difficult to track down exactly where a value is coming from.

## Challenge: Animations

Jetpack Compose has many animation APIs to add pizzazz to UIs. You can find the full list at developer.android.com/jetpack/compose/animation. The same page also has tips to help you decide which function you should use to achieve a certain type of animation.

For this challenge, add some grandeur to Coda Pizza by incorporating animations into your UI. Currently, adding a topping to Coda Pizza causes your pizza's preview to change abruptly. Make this change more graceful by fading the topping onto the pizza. (Hint: Try using the **Crossfade** composable.)

For more of a challenge, add some excitement when placing an order. When the user presses the PLACE ORDER button, make their pizza preview spin in a complete circle.

This will require several changes to your code, including refactoring your **OrderButton** with a new lambda parameter to be called when an order is placed. You will also need to update your **ToppingsList** composable to accept information about the pizza preview's rotation. The pizza can be rotated using the **Modifier.rotate(Float)** modifier. There are several animation APIs that can drive this animation, but we recommend using either **animateFloatAsState** or **Animatable**.

# 30
# Afterword

Congratulations! You are at the end of this guide. Not everyone has the discipline to do what you have done – and learn what you have learned. Take a moment to give yourself a pat on the back.

Your hard work has paid off: You are now an Android developer.

## The Final Challenge

We have one last challenge for you: Become a *good* Android developer. Good developers are each good in their own way, so you must find your own path from here on out.

Where might you start? Here are some places we recommend:

*Write code.* Now. You will quickly forget what you have learned here if you do not apply it. Contribute to a project or write a simple application of your own. Whatever you do, waste no time: Write code.

*Learn.* You have learned a little bit about a lot of things in this book. Did any of them spark your imagination? Write some code to play around with your favorite thing. Find and read more documentation about it – or an entire book, if there is one. Also, check out the Android Developers YouTube channel (`youtube.com/user/androiddevelopers`) and listen to the Android Developers Backstage podcast (`androidbackstage.blogspot.com`) for Android updates from Google.

*Meet people.* Local meetups are a good place to meet like-minded developers. Lots of top-notch Android developers are active on Twitter. Attend Android conferences to meet other Android developers (like us!).

*Explore the open-source community.* Android development is exploding on `github.com`. When you find a cool library, see what other projects its contributors are committing to. Share your own code, too – you never know who will find it useful or interesting. We find the Android Weekly mailing list (`androidweekly.net`) to be a great way to see what is happening in the Android community.

## Shameless Plugs

You can find Bryan (`@bryansills`) and Big Nerd Ranch (`@bignerdranch`) on Twitter.

If you enjoyed this book, check out the other Big Nerd Ranch Guides at `bignerdranch.com/books`. We also have weeklong courses for developers, where we can help you learn this amount of stuff in only a week. And, of course, if you just need someone to write great code, we do contract programming, too. For more information, go to our website at `bignerdranch.com`.

## Thank You

Without readers like you, our work would not exist. Thank you for buying and reading our book.

# Index

## Symbols


`%1$s`, `%2$s`, etc. syntax, 325
`.apk` file, 30
`<meta-data>` tag, 352
`<uses-feature>` tag, 362
`?attr/` syntax, 224
`@+id`, 21
`@After` annotation (JUnit), 109
`@Before` annotation (JUnit), 109
`@Composable` annotation (Jetpack Compose), 538
`@Dao` annotation (Room), 245
`@Database` annotation (Room), 243
`@drawable/` syntax, 48
`@Entity` annotation (Room), 242
`@GET` annotation (Retrofit), 407
`@Insert` annotation (Room), 312
`@Json` annotation (Moshi), 420
`@JsonClass` annotation (Moshi), 420
`@Preview` annotation (Jetpack Compose), 539
`@PrimaryKey` annotation (Room), 242
`@Query` annotation
  in Retrofit, 440
  in Room, 245
`@RequiresApi` annotation, 144
`@RunWith(AndroidJUnit4::class)` annotation (JUnit), 106
`@string/` syntax, 15
`@StringRes` annotation, 34
`@style/` syntax, 223
`@Test` annotation (JUnit), 101
`@TypeConverter` annotation (Room), 244
`@TypeConverters` annotation (Room), 244
`@Update` annotation (Room), 285


## A


aapt2 (Android Asset Packing tool), 30
abstract classes, 243
**`AccelerateInterpolator`** class, 520
accessibility
  (see also TalkBack)
  about, 377, 389
  accessibility focus, 381
  Accessibility Scanner, 394
  accessibility services, 378
  `android:contentDescription` attribute, 386
  `android:focusable` attribute, 388
  `android:labelFor` attribute, 399
  `contentDescription` (**`Image`**) (Jetpack Compose), 615
  Explore by Touch, 382
  **`focusable`** modifier (Jetpack Compose), 618
  ticker text, 470
  for touch targets, 397
  `View.contentDescription`, 390
Accompanist library (Jetpack Compose), 644
action items (see menus)
action views, 442
**`ActionBar`** class, 318
**`ACTION_CALL`** category, 344
**`ACTION_DIAL`** category, 344
**`ACTION_PICK`** category, 334
**`ACTION_SEND`** category, 329
activities
  (see also **`Activity`** class, fragments)
  about, 3
  adding to project, 5, 113-135
  back stack, 135
  child, 114, 127
  configuration changes and, 64
  creating, 116
  declaring in manifest, 120
  finishing, 63
  hosting fragments, 154, 174
  launcher, 132, 137
  lifecycle, 55, 64
  lifecycle, and hosted fragments, 174
  overriding functions in, 56, 58
  passing data between, 123-132
  process death and, 78
  rotation and, 64
  stack, 133
  starting, defined, 114
  starting, in another application, 321
  starting, with **`startActivity(Intent)`**, 122
  states, 55
  UI flexibility and, 152
**`Activity`** class
  about, 3
  **`AppCompatActivity`** and **`ComponentActivity`** vs, 640
  as **`Context`** subclass, 25
  **`findNavController`**, 271

**getIntent()**, 126
lifecycle functions, 55-60
**onCreate(Bundle?)**, 19, 55
**onCreateOptionsMenu(Menu)**, 309
**onDestroy()**, 55
**onPause()**, 55
**onResume()**, 55
**onSaveInstanceState(Bundle)**, 82
**onStart()**, 55
**onStop()**, 55
overriding superclass functions, 58
result codes, 128
**setContentView(…)**, 19
**setResult(…)**, 128
**startActivity(Intent)**, 122, 329
**ActivityManager** class
back stack, 133, 135
starting activities, 122, 123, 129
**ActivityNotFoundException** class, 123
**ActivityResultContracts** class
about, 127
**PickContact**, 334
**StartActivityForResult**, 127
**TakePicture**, 354
**ActivityScenario** class, 109
**Adapter** class (**RecyclerView**)
about, 188
**getItemCount()**, 189
**ListAdapter** vs, 197
**notifyDataSetChanged(…)**, 197
**notifyItemInserted(…)**, 197
**notifyItemMoved(…)**, 197
**onBindViewHolder(…)**, 189
**onCreateViewHolder(…)**, 189
`adb` (Android Debug Bridge) driver, 50
**add(…)** function (**FragmentTransaction**), 172
**addMigrations(…)** function (Room), 324
`@After` annotation (JUnit), 109
**Alignment** interface (Jetpack Compose)
`.Center`, 623
`.TopEnd`, 623
`.TopStart`, 623
Android Asset Packing tool (aapt), 30
Android Debug Bridge (`adb`) driver, 50
Android developer documentation, 147, 148
Android Lint
as code inspector, 34
as static analyzer, 93
compatibility and, 143-146
running, 93
Android manifest (see manifest)
Android SDK Manager, xviii
Android Studio
adding Gradle plugins, 240
Android Lint code inspector, 34
assets, 250
assistant tool window, 8
build process, 30
build tool window, 8
code completion, 25, 164
creating activities, 116
creating classes, 34
creating menu files, 307
creating packages, 243
creating projects, 4-6
creating values resource files, 364
debugger, 88, 90
(see also debugging)
devices view, 50
editor, 8
installing, xviii
layout editor, 202, 375
(see also layouts)
Logcat, 29
(see also Logcat, logging)
previewing layout decorations, 18
project tool window, 7
project window, 7
shortcut for creating test classes, 103
shortcut to override functions, 188
tool windows, 7
Translations Editor, 368
variables view, 90
Android Virtual Device Manager, 26
Android Virtual Devices (AVDs), creating, 26
Android XML namespace, 14
**android.text.format.DateFormat** class, 227
**android.util.Log** class (see **Log** class)
`android.view.animation` package, 525
`android:authorities` attribute, 351
`android:contentDescription` attribute, 386
`android:exported` attribute, 351
`android:focusable` attribute, 388
`android:grantUriPermissions` attribute, 351
`android:id` attribute, 21
`android:labelFor` attribute, 399

android:layout_gravity attribute, 429
android:layout_height attribute, 15
**android:layout_margin** attribute, 226
android:layout_width attribute, 15
android:name attribute, 120, 169, 171
android:orientation attribute, 15
**android:padding** attribute, 226
android:required attribute, 362
android:scaleType attribute, 429
android:text attribute, 15
AndroidManifest.xml (see manifest)
AndroidX (see Jetpack libraries)
androidx.activity:activity-ktx architecture component library (androidx.lifecycle), 70
androidx.lifecycle:lifecycle-viewmodel-ktx architecture component library (androidx.lifecycle), 70
animation
  about, 511-526
  android.view.animation package, 525
  draw order of views, 515
  interpolation, 517
  property animation vs transitions framework, 525
  property animators, 517
  running multiple animators, 523, 524
  simple property animation, 515-523
  transformation properties, 518
    (see also transformation properties)
  transitions framework, 525
  translation, 515
**AnimatorListener** class, 523
**AnimatorSet** class
  about, 524
  **play(Animator)**, 524
.apk file, 30
app bar
  about, 303
  action views in, 442
  **app:showAsAction** attribute, 307
  AppCompat vs Jetpack Compose, 630
  default from **AppCompatActivity**, 304
  menu (see menus)
  previewing, 18
  terminology vs action bar, toolbar, 318
  themes and, 304
app features, declaring in manifest, 362
app namespace, 308
app/build.gradle file, 70
app/java directory, 19
app:actionViewClass attribute, 442
**app:showAsAction** attribute, 307
AppCompat foundation library
  about, 304
  app namespace, 308
  Jetpack Compose and, 640
**AppCompatActivity** class
  about, 19
  **Activity** and **ComponentActivity** vs, 640
application architecture
  Google Guide to App Architecture, 247
  single activity architecture, 260
**Application** class
  **onCreate()**, 248
  registering in manifest, 249
  subclassing, 248
application lifecycle, accessing, 248
assertions, 101
assets, 250
assistant tool window (Android Studio), 8
asynchronous code
  about, 230
  Jetpack Compose and, 589
?attr/ syntax, 224
attributes, 14
  (see also layout attributes, individual attribute names)
**AttributeSet** class, 501
autocompletion, 25
AVDs (Android Virtual Devices), creating, 26

## B

Back button, 137, 498
back stack, 133
background threads
  (see also threads)
  **doWork()** and, 461
  for asynchronous network requests, 410
  scheduling work on, 462, 478
  **Worker** and, 461
@Before annotation (JUnit), 109
**beginTransaction()** function (**FragmentManager**), 172
**Bitmap** class, 358

**BitmapFactory** class, **decodeFile(photoFil**...
...**e.getPath())**, 358
bitmaps
(see also images)
resizing, 359
scaling and displaying, 358-361
bottom property (**View**), 516
**Box** composable (Jetpack Compose)
about, 617
aligning children, 625
breakpoints, setting, 88-92
(see also debugging)
build errors, 22, 96
(see also debugging)
build process, 30
build tool window (Android Studio), 8
Build.VERSION.SDK_INT, 146
**Bundle** class, 586
buttons
adding icons, 47
adding IDs, 21
**Button** class, 11
**Button** composable, 563, 637
**ImageButton** class, 348
**TextButton** composable, 603
by keyword, 575

## C

**Calendar** class, 299
camera, 345-358
**cancelUniqueWork(…)** (**WorkManager**), 479
**Canvas** class, 506
**Card** composable (Jetpack Compose), 600
**Channel** class, 468
**CheckBox** class, 160
**Checkbox** composable (Jetpack Compose)
about, 535
implementing state changes, 573-577
choosers, creating, 332
Chrome Custom Tabs, 496
**Class** class, explicit intents and, 122
classes
(see also singletons)
abstract classes, 243
dependencies, 435
importing, 22
code completion, 25
Coil library, 431
color
**ArgbEvaluator** and, 522
resources, 512
**Column** composable (Jetpack Compose)
about, 534
Alignment parameter, 548
Arrangement parameter, 548
companion objects, 125
compatibility
Android Lint and, 143-146
configuration qualifiers and, 370
fragments and, 163
issues, 142
Jetpack libraries and, 163
minimum SDK version and, 141
using conditional code for, 144
compile SDK version, 142
**ComponentActivity** class, **Activity** and
**AppCompatActivity** vs, 640
components, 122
@Composable annotation, 538
composables
about, 532
@Composable annotation, 538
@Preview annotation, 539
accessing string resources in, 545
adding background, 553
adding padding, 552
aligning images in containers, 623
alignment in nested layouts, 548
**alignment** modifier, 625
as functions, 536
as pure functions, 579
**aspectRatio** modifier, 620
calling other functions, 538
composition hierarchy, 635
control flow in, 546
creating, 536
**fillMaxSize** modifier, 620
**focusable** modifier, 618
for app bars, 633
for checkboxes, 535
for column layouts, 534
for dialogs, 595, 600
for images, 613
for layering content, 617
for layouts, 634

for row layouts, 535
for scrollable lists, 565
for text, 532
for text buttons, 603
making clickable, 557
`modifier` parameter, 552
modifiers (see **`Modifier`** type (Jetpack Compose))
naming conventions, 536
nesting, 534
parameters for, 542-551
positioning within a **`Box`**, 625
previewing, 539
recomposition, 578, 579
remembering state, 580
scaling images to containers, 621
**`size`** modifier, 620
sizing, 559, 560
slotting, 633
state hoisting, 581
styles set by, 637
styling text, 550
trailing lambda syntax and, 599
Compose (see Jetpack Compose)
**`CompositionLocal`** class (Jetpack Compose)
about, 637-639
accessing `CompositionLocal` variables, 637
creating `CompositionLocal` variables, 645
`LocalContext`, 638
**`MaterialTheme`** and, 637
configuration changes
**`Application`** class and, 248
effect on activities, 64
effect on fragments, 165
networking and, 426
**`remember`** composable and, 586
state in Jetpack Compose and, 586
**`ViewModel`** and, 69
configuration qualifiers
Android versions and, 370
for language, 364
listed, 370
multiple, 373-375
order of precedence, 371-375
for screen size, 376
**`ConstraintLayout`** class
about, 201
converting layout to use, 203
**`Guideline`**, 227
**`MotionLayout`**, 227
constraints
about, 201
adding in layout editor, 211
in XML, 214
removing, 206
warnings when insufficient, 206
**`Constraints`** class, **`WorkRequest`** and, 463
contacts
getting data from, 337
permissions for, 339
container view IDs, 172
**`ContentProvider`** class
about, 337
**`FileProvider`** convenience class, 351, 352
for storing files shared among apps, 351
**`ContentResolver`** class, 337
**`ContentScale`** interface (Jetpack Compose)
`.Crop`, 621
`.Fit`, 621
**`Context`** class
explicit intents and, 122
**`fileList(…)`**, 350
functions for private files and directories, 350
**`getCacheDir(…)`**, 350
**`getDir(…)`**, 350
**`getFilesDir()`**, 350
`MODE_WORLD_READABLE`, 350
**`openFileInput(…)`**, 350
**`openFileOutput(…)`**, 350
resource IDs and, 25
contracts, 127
(see also **`ActivityResultContracts`** class)
conventions (see naming conventions)
**`Converter.Factory`** class (Retrofit), 409
converters
converter factories, 409
scalars converter, 409
coroutines
(see also **`Flow`** class, flows)
about, 230-239
**`Activity`** class and, 231
builders, 231
Coroutines library, 231
**`CoroutineScope`** class, 231
enabling, 231
**`Fragment`** class and, 231, 235

Jetpack Compose and, 589
**launch** function, 231
race conditions and, 235
Retrofit library and, 407
scope, 231
suspending functions, 230, 232
threads and, 230
**ViewModel** class and, 231
**createChooser(…)** function (**Intent**), 332
created activity state, 56
**Cursor** class, 337


## D


**d(…)** function (**Log**), 57, 68, 87
@Dao annotation (Room), 245
data access objects (DAOs), 245
data classes, 34
data persistence
using saved instance state, 78
using **ViewModel**, 72
with shared preferences, 448-454
@Database annotation (Room), 243
databases
(see also Room architecture component library)
accessing, 247
data access objects (DAOs), 245
database classes, 243
entities, 242
primary keys, 242
repository pattern, 247
saving changes to, 285-287
schemas, 257
Structured Query Language (SQL), 245
type conversion, 244
DataStore library, 448-451, 454
**Date** class, 299
**DateFormat** class, 227
**DatePickerDialog** class
about, 291, 299
configuration changes and, 292
wrapping in **DialogFragment**, 292
debug key, 30
debugging
(see also Android Lint, Android Studio)
about, 83
build errors, 96
crashes, 85, 86
logging stack traces vs setting breakpoints, 92
misbehaviors, 86
online help for, 96
running app with debugger, 88
stopping debugger, 91
using breakpoints, 88-92
declarative UI toolkit, defined, 571
default resources, 367
delegation, using by keyword, 575
density-independent pixel (dp), 49
dependencies
adding, 70
deleting, 632
dependency injection (DI) design pattern
about, 436
injectors, 258
design patterns
dependency injection (DI), 258, 436
factory software, 281
repository, 247, 413
unidirectional data flow pattern, 279
developer documentation, 147, 148
devices
configurations, 65
(see also configuration qualifiers)
configuring language settings, 364
enabling developer options, 79
hardware, 26
sandbox, 350
testing configurations, 375
virtual, 26
devices view (Android Studio), 50
**Dialog** class, 291
**Dialog** composable (Jetpack Compose), 595
**DialogFragment** class
about, 292
**onCreateDialog(Bundle?)**, 293
dialogs
adding to navigation graphs, 293
in framework UI toolkit, 291-296
in Jetpack Compose, 593-608
passing data to, 297
**DiffUtil** class, 197
dip (density-independent pixel, dp), 49
documentation, 147
**doOnLayout()** function (**View**), 359
dp (density-independent pixel), 49

**draw()** function (**View**), 506
@drawable/ syntax, 48
drawables, 48
drawing
  **Canvas**, 506
  in **onDraw(Canvas)**, 506
  **Paint**, 506

## E

**e(…)** function (**Log**), 68
editor (Android Studio)
  about, 8
  layout editor, 202
**EditText** class, 160
emulator
  creating a virtual device for, 26
  enabling developer options, 79
  installing, xviii
  Quick Settings, 54
  rotating, 48, 53
  running on, 26
  search queries on, 447
  simulating network types, 463
**enqueue(…)** function (**WorkManager**), 462
**enqueueUniquePeriodicWork(…)** (**WorkManager**), 479
@Entity annotation (Room), 242
errors
  (see also debugging, exceptions)
  Android Studio indicators, 22
  DEBUG log level, 68
  ERROR log level, 68
  INFO log level, 68
  VERBOSE log level, 68
  WARNING log level, 68
escape sequence (in strings), 38
event-driven applications, 23
**Exception** class, 87
exceptions
  (see also debugging, errors)
  about, 87
  **ActivityNotFoundException**, 123
  creating, 87
  **IllegalStateException**, 75, 248
  in Logcat, 29, 85
  java.lang exceptions in Kotlin code, 85
  **kotlin.RuntimeException**, 85
  logging, 68
  **Resources.NotFoundException**, 367
  type-aliasing and, 85
  **UninitializedPropertyAccessException**, 85, 423
explicit intents
  about, 123, 321
  creating, 122
  implicit intents vs, 123, 321, 327
Explore by Touch, 382
extras
  about, 124
  as key-value pairs, 124
  naming, 124
  putting, 124, 125
  retrieving, 125
  structure, 124

## F

factory software design pattern, 281
file storage
  authorities, 351
  granting write permission, 351
  private, 350
  shared between apps, 351
file types, for images, 613
**fileList(…)** function (**Context**), 350
**FileProvider** convenience class
  about, 351, 352
  **getUriForFile(…)**, 354
**findNavController** function (**Activity**, **Fragment**), 271
**Flow** class
  (see also flows)
  **collect {}** function, 254
  **MutableStateFlow**, 255
  **StateFlow**, 255
flows
  about, 253
  databases and, 253
fluent interface, defined, 172
format strings, 325
**Fragment** class
  **findNavController**, 271
  lifecycleScope, 235
  **onActivityCreated(Bundle?)**, 173
  **onAttach(Context?)**, 173

**onCreate(Bundle?)**, 165, 173
**onCreateOptionsMenu(…)**, 309
**onCreateView(…)**, 165, 173
**onDestroy()**, 173
**onDestroyView()**, 173
**onDetach()**, 173
**onOptionsItemSelected(MenuItem)**, 309, 313
**onPause()**, 173
**onResume()**, 173
**onStart()**, 173
**onStop()**, 173
**onViewCreated(…)**, 173
**setFragmentResultListener**, 300
**setHasOptionsMenu(Boolean)**, 309
**startActivity(Intent)**, 329
versions, 163
viewLifecycleScope, 235
visibility of lifecycle functions, 165
Fragment Results API, 297
fragment transactions, 172
(see also **FragmentTransaction** class)
**FragmentContainerView** class, 169
**FragmentManager** class
adding fragments, 171-174
**beginTransaction()**, 172
fragment lifecycle functions and, 173
responsibilities, 171
fragments
(see also **Fragment** class, **FragmentContainerView** class, **FragmentManager** class, **FragmentTransaction** class)
activities vs, 152
activity lifecycle and, 174
adding a fragment to an activity, 172
adding to **FragmentManager**, 171-174
as composable units, 152
compatibility and, 163
container view IDs, 172
creating, 160
hosting, 154, 169
implementing lifecycle functions, 164, 165
inflating layouts for, 165
Jetpack libraries and, 163
lifecycle, 164, 173
lifecycle functions, 173
memory management and, 174
reasons for, 152, 153
setting listeners in, 167
transactions, 171
UI flexibility and, 152
using Jetpack (androidx) version, 163
views and, 167
**FragmentTransaction** class
about, 171
**add(…)**, 172
functions
assertions, 101
pure, 579
side effects, 579
suspending with coroutines, 230, 232

## G

**GestureDetector** class, 509
@GET annotation (Retrofit), 407
**getAction()** function (**MotionEvent**), 503
**getBooleanExtra(…)** function (**Intent**), 125
**getCacheDir(…)** function (**Context**), 350
**getDir(…)** function (**Context**), 350
**getFilesDir()** function (**Context**), 350
**getIntent()** function (**Activity**), 126
**getUriForFile(…)** function (**FileProvider**), 354
**GlobalScope** class, 286
Gradle source sets, 99
**GridView** class, 196
**Guideline** class, 227

## H

hardware devices, 26
height property (**View**), 516
Home gesture, 61
HTTP networking (see networking)
HTTP request method annotations, 407

## I

**i(…)** function (**Log**), 68
**IllegalStateException** class, 75, 248
**Image** composable (Jetpack Compose)
about, 613
alignment property, 623
contentDescription, 615
contentScale property, 621
**ImageButton** class, 348
images

displaying with Coil library, 431
image types, 613
scaling and displaying bitmaps, 358-361
implicit intents
about, 123, 321
action, 327
**ACTION_CALL** category, 344
**ACTION_DIAL** category, 344
**ACTION_PICK** category, 334
**ACTION_SEND** category, 329
benefits of using, 322
categories, 327
data, 327
explicit intents vs, 123, 321, 327
extras, 328
FLAG_GRANT_READ_URI_PERMISSION flag, 339
for browsing web content, 484
**LAUNCHER** category, 132
**MAIN** category, 132
parts of, 327
inflating layouts, 19, 165
@Insert annotation (Room), 312
instrumented tests
about, 99
**ActivityScenario**, 109
creating, 106-112
JUnit framework, 101
**Intent** class
about, 327
constructors, 122, 329
**createChooser(…)**, 332
**getBooleanExtra(…)**, 125
**putExtra(…)**, 124
intent filters
about, 132
explained, 328
Intent.FLAG_GRANT_READ_URI_PERMISSION flag, 339
intents
(see also explicit intents, extras, implicit intents, **Intent** class, intent filters)
about, 122
checking for responding activities, 340
communicating with, 122, 123
companion objects and, 125
extras, 124
implicit vs explicit, 123, 321, 327
permissions and, 339
setting results, 128
**Interceptor** interface (OkHttp library), 438
interceptors, 438
interfaces, with a single abstract method (SAMs), 23
interpolators, 520
**invalidate()** function (**View**), 506

## J

Java Virtual Machine (JVM) tests
about, 99
creating, 103-106
JUnit framework, 101
JavaScript Object Notation (JSON) (see JSON (JavaScript Object Notation))
JavaScript, enabling, 490
javaScriptEnabled property (**WebSettings**), 490
Jetpack Compose
about, 527-533
@Composable annotation, 538
@Preview annotation, 539
accessibility in, 382, 615, 618
Accompanist library, 644
adding to a project, 530
AppCompat foundation library and, 640
AppCompat themes vs, 630
as a declarative toolkit, 571
asynchronous code and, 589
**Checkbox** composable, 535
**Column** composable, 534, 548
composable modifiers (see **Modifier** type (Jetpack Compose))
composables (see composables, individual composable names)
composition hierarchy, 635
**CompositionLocal** class, 637-639
(see also **CompositionLocal** class (Jetpack Compose))
coroutines and, 589
displaying images, 613
Kotlin versions and, 531
**LazyColumn** composable, 565
**LazyRow** composable, 565
live literals, 568
**MaterialTheme** composable, 629
**MaterialTheme** object, 550, 629
**Modifier** type, 552-561

(see also **`Modifier`** type (Jetpack Compose))
previewing layouts, 539
recomposition, 578, 579
**`remember`** composable, 580
**`rememberSaveable`** composable, 586
**`Row`** composable, 535, 548
**`Scaffold`** composable, 634
scrollable lists in, 565
**`setContent`**, 532
sizing UI elements, 559, 560
state hoisting, 581
state in, 571-588, 593-608
(see also state in Jetpack Compose, state in Jetpack Compose)
**`stringResource(Int)`**, 545
styles set by composables, 637
**`Text`** composable, 532, 550
themes, 629-632
**`TopAppBar`** composable, 633
trailing lambda syntax and, 599
Jetpack libraries
(see also libraries, individual library names)
about, 81
`androidx.activity` package, 70
`androidx.lifecycle` package, 70
for backward compatibility, 143
**`Job`** class, 236
JSON (JavaScript Object Notation)
about, 414
arrays, 419
deserializing, 419
Moshi library and, 419
objects, 419
`@Json` annotation, 420
`@JsonClass` annotation, 420
JUnit testing framework (see testing, instrumented tests, Java Virtual Machine (JVM) tests)

## K

Kotlin
coroutines (see coroutines)
enabling in an Android Studio project, 6
exceptions, compiled to java.lang exceptions, 85
functions public by default, 165
Kotlin annotation processing tool (`kapt`), 240
Kotlin files in `java` directory, 19
single abstract method interfaces (SAMs) and, 23
**`kotlin.RuntimeException`** class, 85

## L

language qualifiers, 364
language settings, device, 364
language-region qualifiers, 369
launcher activities, 132
**`LAUNCHER`** category (**`Intent`**), 132
layout attributes
`android:id`, 21
`android:layout_height`, 15
**`android:layout_margin`**, 226
`android:layout_width`, 15
`android:orientation`, 15
**`android:padding`**, 226
`android:text`, 15
layout constraints (see constraints)
layout editor (Android Studio), 202, 375
layout parameters, 214
**`LayoutInflater`** class, 30
**`LayoutManager`** class, 183
layouts
(see also constraints, layout attributes, layout editor (Android Studio))
about, 3
animating, 227
defining in XML, 12-15
design guidelines, 215
inflating, 19, 165
naming, 9
nested vs flat, 200
previewing (Jetpack Compose), 539
previewing (XML layouts), 17
previewing device configurations, 375
for property animation, 513
root element, 14
testing, 375
using guidelines, 227
view groups and, 10
view hierarchy and, 14
**`LazyColumn`** composable (Jetpack Compose)
about, 565
**`item`**, 566

**items**, 566
**itemsIndexed**, 566
**RecyclerView** vs, 565
state in, 591
**LazyRow** composable (Jetpack Compose)
about, 565
**item**, 566
**items**, 566
**itemsIndexed**, 566
**RecyclerView** vs, 565
`left` property (**View**), 516
libraries
adding to projects, 70
removing from project, 632
lifecycle callbacks, 56
**Lifecycle** class, **repeatOnLifecycle(…)**, 237
**LinearLayout** class, 11, 14
Lint (see Android Lint)
list-detail interfaces, 151
**ListAdapter** class
about, 197
**DiffUtil** and, 197
listeners
about, 23
as interfaces, 23
setting in fragments, 167
setting up, 23-25
lists
displaying, 177
getting item data, 188
in Jetpack Compose, 565
**ListView** class, 196
live literals, 568
local layout rect, 516
`LocalContext` property (**CompositionLocal**), 638
localization
about, 363
creating values resource files, 364
default resources and, 367
language qualifiers, 364
language-region qualifiers, 369
other configuration qualifiers and, 371
testing, 375
Translations Editor, 368
**Log** class
**d(…)**, 57, 68, 87
**e(…)**, 68
**i(…)**, 68
levels, 68
logging messages, 57
**v(…)**, 68
**w(…)**, 68
Logcat
(see also logging)
about, 59, 60
filtering, 29, 59, 73
logging messages, 57
setting log level, 85
wrapping output, 417
logging
exceptions, 87
messages, 57
stack traces, 87
`TAG` constant, 57

## M

**MAIN** category (**Intent**), 132
**makeText(…)** function (**Toast**), 25
manifest
about, 120
`<meta-data>` tag, 352
`<uses-feature>` tag, 362
adding network permissions, 411
adding **uses-permission**, 411
build process and, 30
declaring **Activity** in, 120
disclosing queries, 340
margins, 226
master-detail interfaces, 151
`match_parent`, 15
Material Components library, 305
**MaterialTheme** composable (Jetpack Compose)
about, 629
**CompositionLocal** and, 637
nested, 637
scope, 637
**MaterialTheme** object (Jetpack Compose)
about, 550, 629
text styles, 550
memory leaks, 75
**MenuItem** class, 313
menus
(see also app bar)
about, 305
action items, 305

**app:showAsAction** attribute, 307
creating, 309
creating XML file for, 307
defining in XML, 307
determining selected item, 313
overflow menu, 307
populating with items, 309
as resources, 307
responding to selections, 312
<meta-data> tag, 352
**Migration** classes (Room), 323
model classes, using data keyword, 34
**Modifier** type (Jetpack Compose)
about, 552
**alignment**, 625
**aspectRatio**, 620
**background**, 553
**clickable**, 557
**fillMaxSize**, 620
**focusable**, 618
ordering modifiers, 553
**padding**, 552
**size**, 620
**weight**, 559
Moshi library
about, 419-423
@Json annotation, 420
@JsonClass annotation, 420
motion events, handling, 502-506
**MotionEvent** class
about, 503
actions, 503
**getAction()**, 503
**MotionLayout** class, 227
multi-window (split screen) mode
activity states and, 67
multi-resume support, 67
**MutableState** interface (Jetpack Compose)
about, 575
**mutableStateOf**, 575
**mutableStateOf** function (**MutableState**), 575

## N

namespaces
Android resource XML, 14
app, 308
tools, 37, 264
naming conventions
classes, 8
composables, 536
extras, 124
file sharing authorities, 351
icons, 210
layouts, 9
menu files, 307
packages, 6
screens in Jetpack Compose, 562
test classes, 103
test functions, 101
unused parameters, 167
**NavController** class, 271
**NavHostFragment** class, 265, 271
navigation
(see also Navigation Jetpack component library)
**Activity** lifecycle and, 137
Back button, 137
navigation graphs (Navigation Jetpack component library)
about, 260
adding destinations, 262
adding dialogs, 293
creating, 261
defining actions, 267
hosting, 265
previews, 263, 288
Navigation Jetpack component library
about, 260
**findNavController**, 271
**NavController**, 271
**NavHostFragment**, 271
navigation graphs, 260
(see also navigation graphs (Navigation Jetpack component library))
performing navigations, 271
Safe Args Gradle plugin and, 273
networking
about, 402
configuration changes and, 426
limiting by network type, 463
permissions, 411
providing user control, 474
scheduling, 462, 478
nonexistent activity state, 56
**Notification** class

(see also notifications)
about, 470
**NotificationManager** and, 470
**NotificationCompat** class, 472
**NotificationManager** class
**Notification** and, 470
**notify(…)**, 470
notifications
about, 468-473
configuring, 472
notification channels, 468
**notify(…)** function (**NotificationManager**), 470

## O

**ObjectAnimator** class, 516
OkHttp HTTP client library, 408, 438
**onActivityCreated(Bundle?)** function (**Fragment**), 173
**onAttach(Context?)** function (**Fragment**), 173
**OnCheckedChangeListener** interface, 168
**onCleared()** function (**ViewModel**), 71
**OnClickListener** interface, 23
**onCreate()** function (**Application**), overriding, 248
**onCreate(Bundle?)** function (**Activity**), 19, 55
**onCreate(Bundle?)** function (**Fragment**), 165, 173
**onCreateDialog(Bundle?)** function (**DialogFragment**), 293
**onCreateOptionsMenu(Menu)** function (**Activity**), 309
**onCreateOptionsMenu(…)** function (**Fragment**), 309
**onCreateView(…)** function (**Fragment**), 165, 173
**onDestroy()** function (**Activity**), 55
**onDestroy()** function (**Fragment**), 173
**onDestroyView()** function (**Fragment**), 173
**onDetach()** function (**Fragment**), 173
**onDraw(Canvas)** function (**View**), 506
**OneTimeWorkRequest** class (**WorkRequest**), 462
**onOptionsItemSelected(MenuItem)** function (**Fragment**), 309, 313
**onPause()** function (**Activity**), 55
**onPause()** function (**Fragment**), 173
**onProgressChanged(…)** function (**WebChromeClient**), 493
**OnQueryTextListener(…)** interface (**SearchView**), 445
**onReceivedTitle(…)** function (**WebChromeClient**), 493
**onRestoreInstanceState(Parcelable)** function (**View**), 509
**onResume()** function (**Activity**), 55
**onResume()** function (**Fragment**), 173
**onSaveInstanceState()** function (**View**), 509
**onSaveInstanceState(Bundle)** function (**Activity**), 82
**onStart()** function (**Activity**), 55
**onStart()** function (**Fragment**), 173
**onStop()** function (**Activity**), 55
**onStop()** function (**Fragment**), 173
**onTouchEvent(MotionEvent)** function (**View**), 502
**OnTouchListener** interface (**View**), 502
**onViewCreated(…)** function (**Fragment**), 173
**openFileInput(…)** function (**Context**), 350
**openFileOutput(…)** function (**Context**), 350
overflow menu, 307
`override` keyword, 58
overriding functions, Android Studio shortcut, 164
overview screen, 61, 63

## P

**PackageManager** class
about, 341
**resolveActivity(…)**, 341
packages
creating, 243
naming, 6
padding, 226
**Paint** class, 506
**Painter** class (Jetpack Compose)
about, 613
**painterResource**, 613
**painterResource** function (**Painter**), 613
parameters, _ to denote unused, 167
**Parcelable** interface
about, 509, 586
implementing with Parcelize plugin, 587
**PeriodicWorkRequest** class (**WorkRequest**), 462, 478
permissions

adding to manifest, 411
`android:authorities` attribute, 351
`android:exported` attribute, 351
`android:grantUriPermissions` attribute, 351
for contacts database, 339
`Intent.FLAG_GRANT_READ_URI_PERMISSION` flag, 339
`INTERNET`, 411
normal, 411
Request App Permissions guide, 344
runtime, 344
**`PhotoView`** class, 359
**`PickContact`** (**`ActivityResultContracts`**), 334
placeholders (in format strings), 325
**`play(Animator)`** function (**`AnimatorSet`**), 524
**`PointF`** class, 503
presses (see touch events)
`@Preview` annotation, 539
`@PrimaryKey` annotation (Room), 242
process death, 78
progress indicator, hiding, 493
**`ProgressBar`** class, 491
project tool window (Android Studio), 7
project window (Android Studio), 7
projects
adding dependencies, 70
adding resources, 45
`app/java` directory, 19
configuring, 5
creating, 4-6
deleting dependencies, 632
layout, 9
`res/layout` directory, 20
`res/menu` directory, 307
`res/values` directory, 20
setting package name, 5
setting project name, 5
property animation (see animation)
property delegates, 72
protocol buffers, 448


## Q


queries
disclosing, 340
for apps, 340
`@Query` annotation
in Retrofit, 440
in Room, 245


## R


race conditions, 235
Recents gesture, 61
recomposition, 578
**`RecyclerView`** class
about, 181-194
animations, 196
as a **`ViewGroup`**, 184
creating views, 193
item views, 184
**`LayoutManager`** and, 183
**`LazyColumn`** or **`LazyRow`** vs, 565
**`ListAdapter`** and, 197
**`ListView`** and **`GridView`** vs, 196
**`setOnItemClickListener(…)`**, 484
**`ViewHolder`** and, 186
**`RecyclerView.Adapter`** class
about, 188
**`getItemCount()`**, 189
**`ListAdapter`** vs, 197
notifyDataSetChanged(…), 197
notifyItemInserted(…), 197
notifyItemMoved(…), 197
**`onBindViewHolder(…)`**, 189
**`onCreateViewHolder(…)`**, 189
release key, 30
**`remember`** composable (Jetpack Compose), 580
**`rememberSaveable`** composable (Jetpack Compose), 586
**`repeatOnLifecycle(…)`** function (**`Lifecycle`**), 237
repositories, 247, 413
repository design pattern, 247, 413
`@RequiresApi` annotation, 144
`res/layout` directory, 20
`res/menu` directory, 307
`res/values` directory, 16, 20
**`resolveActivity(…)`** function (**`PackageManager`**), 341
resource IDs
about, 20, 21
+ prefix in, 21
resources
(see also color, configuration qualifiers, drawables, layouts, menus, string resources)

about, 20
adding, 45
alternative, 371-375
assets vs, 250
default, 367
directories, 20
localizing, 363-366
referencing in XML, 47
**`Resources.NotFoundException`** class, 367
result code (**`Activity`**), 128
resumed activity state, 56, 67
Retrofit library
`@GET` annotation, 407
about, 406-413
**`baseUrl(…)`** function, 408
**`build()`** function, 408
**`Converter.Factory`** class, 409
coroutines and, 407
**`create()`** function, 408
defining an API interface, 407
HTTP request method annotations, 407
**`Retrofit.Builder()`** class, 408
scalars converter, 409
**`Retrofit.Builder()`** class (Retrofit)
about, 408
**`baseUrl(…)`**, 408
**`build()`**, 408
`right` property (**`View`**), 516
Room architecture component library
`@Dao` annotation, 245
`@Database` annotation, 243
`@Entity` annotation, 242
`@Insert` annotation, 312
`@PrimaryKey` annotation, 242
`@Query` annotation, 245
`@TypeConverter` annotation, 244
`@TypeConverters` annotation, 244
`@Update` annotation, 285
accessing a database, 247
adding database properties, 323, 324
**`addMigrations(…)`**, 324
defining a data access object (DAO), 245
defining a database class, 243
defining database entities, 242
defining database primary key, 242
instantiating a database, 249
**`Migration`** classes, 323
**`Room.databaseBuilder()`**, 249
setting up a database, 240-244
specifying type converters, 244
SQL commands, 245
SQLite in, 243
updating database version, 323, 324
rotation (see configuration changes)
**`Row`** composable (Jetpack Compose)
about, 535
`Alignment` parameter, 548
`Arrangement` parameter, 548
running on device, 50, 51
`@RunWith(AndroidJUnit4::class)` annotation (JUnit), 106


## S


Safe Args Gradle plugin, 273-278
SAMs (single abstract method interfaces), 23
sandbox, device, 350
saved instance state, 78
**`SavedStateHandle`** class, 78
**`Scaffold`** composable (Jetpack Compose)
about, 634
`content`, 634
`topBar`, 634
scale-independent pixel (sp), 49
scope, 72
screen pixel density, 48
screen size, determining, 376
SDK versions
(see also compatibility)
about, 142
configuration qualifiers and, 370
installing, xviii
updating, xix
search
about, 437-457
integrating into app, 437
user-initiated, 442-447
**`SearchView`** class
about, 442-447
bug, 447
**`OnQueryTextListener(…)`**, 445
responding to user interactions, 445
**`setContent`** function (Jetpack Compose), 532
**`setContentView(…)`** function (**`Activity`**), 19
**`setFragmentResultListener`** function (**`Fragment`**), 300

**setHasOptionsMenu(Boolean)** function (**Fragment**), 309
**setOnClickListener(OnClickListener)** function (**View**), 23
**setOnItemClickListener(…)** function (**RecyclerView**), 484
**setOnTouchListener(…)** function (**View**), 502
**setResult(…)** function (**Activity**), 128
**setText(Int)** function (**TextView**), 126
shared preferences, 448
**shouldOverrideUrlLoading(…)** function (**WebViewClient**), 490
**show()** function (**Toast**), 25
simulator (see emulator)
single abstract method interfaces (SAMs), 23
single activity architecture, 260
singletons
  about, 247
  activity/fragment lifecycles and, 247
  benefits and drawbacks, 258
source sets, 99
sp (scale-independent pixel), 49
stack traces
  in Logcat, 85
  logging, 87
**startActivity(Intent)** function (**Activity**), 122, 329
**startActivity(Intent)** function (**Fragment**), 329
**StartActivityForResult** (**ActivityResultContracts**), 127
started activity state, 56, 67
state in Jetpack Compose
  about, 571-588, 593-608
  changes to state objects vs properties, 573
  configuration changes and, 586
  delegation and, 575
  dialogs and UI state, 593-608
  **MutableState** and, 575
  responding to changes in application state, 575
`@string/` syntax, 15
string resources
  about, 15, 16
  `@StringRes` annotation, 34
  about, 15
  creating, 16
  referencing, 48
  `res/values/strings.xml`, 16
`@StringRes` annotation, 34
**stringResource(Int)** function (Jetpack Compose), 545
strings, format, 325
Structured Query Language (SQL), 245
`@style/` syntax, 223
styles
  `/res/values/themes.xml` file, 223
  creating custom styles, 223
  Material Design and, 223
`suspend` modifier, 234

## T

`TAG` constant, 57
**TakePicture** (**ActivityResultContracts**), 354
TalkBack
  about, 378
  Android views' inherent support, 382
  enabling, 379
  linear navigation by swiping, 382
  non-text elements and, 385-388
target SDK version, 142
`@Test` annotation (JUnit), 101
testing
  `@After` annotation (JUnit), 109
  `@Before` annotation (JUnit), 109
  `@RunWith(AndroidJUnit4::class)` annotation (JUnit), 106
  `@Test` annotation (JUnit), 101
  creating tests, 103-112
  instrumented, 99
    (see also instrumented tests)
  Java Virtual Machine (JVM), 99
    (see also Java Virtual Machine (JVM) tests)
  JUnit framework, 101
  running tests, 101
  setup, test, verify pattern, 105
  unit, 99
  using **ActivityScenario**, 109
**Text** composable (Jetpack Compose)
  about, 532
  `style` parameter, 550
  styles set by, 637
**TextButton** composable (Jetpack Compose), 603
**TextView** class
  example, 11
  **setText(Int)**, 126

tools:text and, 37
themes
(see also styles)
about, 223
?attr/ syntax for attributes, 224
app bar and, 304
AppCompat vs Jetpack Compose, 630
default, 305
in Jetpack Compose, 629-632
referencing theme attributes, 224
theme attributes, 223
threads
about, 230
background, 230
(see also background threads)
blocking, 230
coroutines and, 230
main (UI), 230, 410
ticker text, 470
**TimeInterpolator** class, 520
**Toast** class
**makeText(…)**, 25
**show()**, 25
toasts, 24, 25
tool windows (Android Studio), 7
toolbar (see app bar)
**Toolbar** class, 318
tools:layout attribute, 264
tools:listitem attribute, 288
tools:text attribute, 37
top property (**View**), 516
**TopAppBar** composable (Jetpack Compose), 633
touch events
action constants, 503
handling, 195, 502-506
handling with **GestureDetectorCompat**, 509
**MotionEvent** and, 503
recommended minimum size for touch targets, 397
transformation properties
pivotX, 518
pivotY, 518
rotation, 518
scaleX, 518
scaleY, 518
translationX, 518
translationY, 518
transitions framework, for animation, 525
Translations Editor, 368
@TypeConverter annotation (Room), 244
@TypeConverters annotation (Room), 244
**TypeEvaluator** class, 522


## U


unidirectional data flow pattern, 279-284
**UninitializedPropertyAccessException** class, 85, 423
unit testing, 99
@Update annotation (Room), 285
**Uri** class
creating shareable instances, 354
**FileProvider.getUriForFile(…)** and, 354
user interfaces
activities vs fragments in, 152
declarative vs imperative, 527
defined by layout, 3
flexibility in, 152
framework UI toolkit vs Jetpack Compose, 527
laying out, 9-17
<uses-feature> tag, 362


## V


**v(…)** function (**Log**), 68
variables view (Android Studio), 90
vector drawables, 45
**VectorDrawable** class, 45
View Binding
about, 39
memory management and, 174
**View** class
(see also views)
bottom, 516
**Button**, 11
**CheckBox**, 160
**doOnLayout()**, 359
**draw()**, 506
**EditText**, 160
height, 516
**invalidate()**, 506
left, 516
**LinearLayout**, 11, 14
**OnClickListener** interface, 23
**onDraw(Canvas)**, 506
**onRestoreInstanceState(Parcelable)**, 509

**onSaveInstanceState()**, 509
**onTouchEvent(MotionEvent)**, 502
**OnTouchListener** interface, 502
`right`, 516
**setOnClickListener(OnClickListener)**, 23
**setOnTouchListener(…)**, 502
subclasses, 10
`tag` property, 360
**TextView**, 11, 37
`top`, 516
**ViewGroup** class, 10, 14
**ViewHolder** class
about, 186
`itemView` property, 186
**ViewModel** class
about, 69-80
activity lifecycle and, 72
constructors, 281
fragment lifecycle and, 179, 426
**onCleared()**, 71
for storage, 179
**ViewModelProvider.Factory** interface, 281
**viewModels()** property delegate, 72
`viewModelScope`, 231
views
about, 10
action, 442
adding in layout editor, 209
attributes, 14
creating custom views, 500
creation by **RecyclerView**, 193
custom, 500-502
defining in XML, 12-15
downsides of framework UI toolkit, 527
draw order, 515
for buttons, 11
for checkboxes, 160
for displaying text, 11, 37
for laying out other views, 11
for text entry, 160
hierarchy, 14
invalid, 506
margins, 226
padding, 226
persisting, 509
references, 22
simple vs composite, 500
size settings, 207
TalkBack and, 382-388
`title` attribute and accessibility, 383
touch events and, 502-506
UI state in framework views, 570
using fully qualified name in layout, 501
view groups, 10
wiring up, 21
wiring up in fragments, 167
virtual devices (see emulator)

## W

**w(…)** function (**Log**), 68
web content
browsing via implicit intent, 484
displaying within an activity, 486
enabling JavaScript, 490
in Chrome Custom Tabs, 496
web rendering events, responding to, 490
**WebChromeClient** interface
about, 492
for enhancing appearance of **WebView**, 491
**onProgressChanged(…)**, 493
**onReceivedTitle(…)**, 493
**WebSettings** class, 490
**WebView** class
custom UI vs, 495
for presenting web content, 486
**WebViewClient** and, 490
**WebViewClient** class
about, 490
**shouldOverrideUrlLoading(…)**, 490
**WebView** and, 490
work requests
about, 462, 478
constraints for, 463
**Worker** class
about, 461
**doWork()**, 461
enabling and disabling, 474
scheduling with **WorkRequest**, 462, 478
WorkManager architecture component library
about, 460
**Constraints**, 463
**Worker**, 461
**WorkRequest**, 461
**WorkManager** class, **enqueue(…)**, 462
**WorkRequest** class

about, 461
**`Constraints`** and, 463
scheduling a **`Worker`**, 462, 478
subclasses, 462
`wrap_content`, 15

# X

XML
Android namespace, 14
referencing resources in, 47