

Android Studio Jellyfish Essentials



Kotlin Edition

Android Studio Jellyfish Essentials

Kotlin Edition

Android Studio Jellyfish Essentials – Kotlin Edition

ISBN: 978-1-951442-92-7

© 2024 Neil Smyth / Payload Media, Inc. All Rights Reserved.

This book is provided for personal use only. Unauthorized use, reproduction and/or distribution strictly prohibited. All rights reserved.

The content of this book is provided for informational purposes only. Neither the publisher nor the author offers any warranties or representation, express or implied, with regard to the accuracy of information contained in this book, nor do they accept any liability for any loss or damage arising from any errors or omissions.

This book contains trademarked terms that are used solely for editorial purposes and to the benefit of the respective trademark owner. The terms used within this book are not intended as infringement of any trademarks.

Rev: 1.0



<https://www.payloadbooks.com>

Table of Contents

1. Introduction	1
1.1 Downloading the Code Samples	1
1.2 Feedback	1
1.3 Errata	2
2. Setting up an Android Studio Development Environment	3
2.1 System requirements	3
2.2 Downloading the Android Studio package	3
2.3 Installing Android Studio	4
2.3.1 Installation on Windows	4
2.3.2 Installation on macOS	4
2.3.3 Installation on Linux	5
2.4 Installing additional Android SDK packages	5
2.5 Installing the Android SDK Command-line Tools	8
2.5.1 Windows 8.1	9
2.5.2 Windows 10	10
2.5.3 Windows 11	10
2.5.4 Linux	10
2.5.5 macOS	10
2.6 Android Studio memory management	10
2.7 Updating Android Studio and the SDK	11
2.8 Summary	12
3. Creating an Example Android App in Android Studio	13
3.1 About the Project	13
3.2 Creating a New Android Project	13
3.3 Creating an Activity	14
3.4 Defining the Project and SDK Settings	14
3.5 Modifying the Example Application	15
3.6 Modifying the User Interface	16
3.7 Reviewing the Layout and Resource Files	22
3.8 Adding Interaction	25
3.9 Summary	26
4. Creating an Android Virtual Device (AVD) in Android Studio	27
4.1 About Android Virtual Devices	27
4.2 Starting the Emulator	29
4.3 Running the Application in the AVD	30
4.4 Running on Multiple Devices	31
4.5 Stopping a Running Application	32
4.6 Supporting Dark Theme	32
4.7 Running the Emulator in a Separate Window	33
4.8 Removing the Device Frame	36
4.9 Summary	38

5. Using and Configuring the Android Studio AVD Emulator	39
5.1 The Emulator Environment	39
5.2 Emulator Toolbar Options	39
5.3 Working in Zoom Mode	41
5.4 Resizing the Emulator Window.....	41
5.5 Extended Control Options.....	41
5.5.1 Location.....	42
5.5.2 Displays.....	42
5.5.3 Cellular	42
5.5.4 Battery.....	42
5.5.5 Camera.....	42
5.5.6 Phone.....	42
5.5.7 Directional Pad.....	42
5.5.8 Microphone.....	42
5.5.9 Fingerprint	42
5.5.10 Virtual Sensors	43
5.5.11 Snapshots.....	43
5.5.12 Record and Playback	43
5.5.13 Google Play	43
5.5.14 Settings	43
5.5.15 Help.....	43
5.6 Working with Snapshots.....	43
5.7 Configuring Fingerprint Emulation	44
5.8 The Emulator in Tool Window Mode.....	45
5.9 Creating a Resizable Emulator.....	46
5.10 Summary	48
6. A Tour of the Android Studio User Interface	49
6.1 The Welcome Screen	49
6.2 The Menu Bar	50
6.3 The Main Window	50
6.4 The Tool Windows	52
6.5 The Tool Window Menus.....	55
6.6 Android Studio Keyboard Shortcuts	56
6.7 Switcher and Recent Files Navigation	56
6.8 Changing the Android Studio Theme	57
6.9 Summary	58
7. Testing Android Studio Apps on a Physical Android Device.....	59
7.1 An Overview of the Android Debug Bridge (ADB).....	59
7.2 Enabling USB Debugging ADB on Android Devices.....	59
7.2.1 macOS ADB Configuration.....	60
7.2.2 Windows ADB Configuration.....	61
7.2.3 Linux adb Configuration.....	62
7.3 Resolving USB Connection Issues	62
7.4 Enabling Wireless Debugging on Android Devices	63
7.5 Testing the adb Connection.....	65
7.6 Device Mirroring.....	65
7.7 Summary	65

8. The Basics of the Android Studio Code Editor	67
8.1 The Android Studio Editor.....	67
8.2 Splitting the Editor Window.....	70
8.3 Code Completion.....	70
8.4 Statement Completion.....	72
8.5 Parameter Information.....	72
8.6 Parameter Name Hints.....	72
8.7 Code Generation.....	72
8.8 Code Folding.....	74
8.9 Quick Documentation Lookup.....	75
8.10 Code Reformatting.....	75
8.11 Finding Sample Code.....	76
8.12 Live Templates.....	76
8.13 Summary.....	77
9. An Overview of the Android Architecture	79
9.1 The Android Software Stack.....	79
9.2 The Linux Kernel.....	80
9.3 Hardware Abstraction Layer.....	80
9.4 Android Runtime – ART.....	80
9.5 Android Libraries.....	80
9.5.1 C/C++ Libraries.....	81
9.6 Application Framework.....	82
9.7 Applications.....	82
9.8 Summary.....	82
10. The Anatomy of an Android App	83
10.1 Android Activities.....	83
10.2 Android Fragments.....	83
10.3 Android Intents.....	84
10.4 Broadcast Intents.....	84
10.5 Broadcast Receivers.....	84
10.6 Android Services.....	84
10.7 Content Providers.....	85
10.8 The Application Manifest.....	85
10.9 Application Resources.....	85
10.10 Application Context.....	85
10.11 Summary.....	85
11. An Introduction to Kotlin	87
11.1 What is Kotlin?.....	87
11.2 Kotlin and Java.....	87
11.3 Converting from Java to Kotlin.....	87
11.4 Kotlin and Android Studio.....	88
11.5 Experimenting with Kotlin.....	88
11.6 Semi-colons in Kotlin.....	89
11.7 Summary.....	89
12. Kotlin Data Types, Variables, and Nullability	91

Table of Contents

12.1 Kotlin Data Types.....	91
12.1.1 Integer Data Types	92
12.1.2 Floating-Point Data Types	92
12.1.3 Boolean Data Type.....	92
12.1.4 Character Data Type.....	92
12.1.5 String Data Type.....	92
12.1.6 Escape Sequences	93
12.2 Mutable Variables.....	94
12.3 Immutable Variables	94
12.4 Declaring Mutable and Immutable Variables.....	94
12.5 Data Types are Objects	94
12.6 Type Annotations and Type Inference	95
12.7 Nullable Type.....	96
12.8 The Safe Call Operator	96
12.9 Not-Null Assertion.....	97
12.10 Nullable Types and the let Function.....	97
12.11 Late Initialization (lateinit)	98
12.12 The Elvis Operator	99
12.13 Type Casting and Type Checking	99
12.14 Summary.....	100
13. Kotlin Operators and Expressions	101
13.1 Expression Syntax in Kotlin.....	101
13.2 The Basic Assignment Operator.....	101
13.3 Kotlin Arithmetic Operators	101
13.4 Augmented Assignment Operators	102
13.5 Increment and Decrement Operators	102
13.6 Equality Operators.....	103
13.7 Boolean Logical Operators	103
13.8 Range Operator	104
13.9 Bitwise Operators.....	104
13.9.1 Bitwise Inversion	104
13.9.2 Bitwise AND	105
13.9.3 Bitwise OR.....	105
13.9.4 Bitwise XOR.....	105
13.9.5 Bitwise Left Shift.....	106
13.9.6 Bitwise Right Shift.....	106
13.10 Summary.....	107
14. Kotlin Control Flow	109
14.1 Looping Control flow	109
14.1.1 The Kotlin <i>for-in</i> Statement.....	109
14.1.2 The <i>while</i> Loop	110
14.1.3 The <i>do ... while</i> loop	111
14.1.4 Breaking from Loops.....	111
14.1.5 The <i>continue</i> Statement	112
14.1.6 Break and Continue Labels.....	112
14.2 Conditional Control Flow.....	113
14.2.1 Using the <i>if</i> Expressions	113

14.2.2 Using <i>if... else ...</i> Expressions	114
14.2.3 Using <i>if... else if...</i> Expressions	114
14.2.4 Using the <i>when</i> Statement.....	114
14.3 Summary	115
15. An Overview of Kotlin Functions and Lambdas	117
15.1 What is a Function?	117
15.2 How to Declare a Kotlin Function	117
15.3 Calling a Kotlin Function.....	118
15.4 Single Expression Functions.....	118
15.5 Local Functions	118
15.6 Handling Return Values	119
15.7 Declaring Default Function Parameters.....	119
15.8 Variable Number of Function Parameters	119
15.9 Lambda Expressions	120
15.10 Higher-order Functions	121
15.11 Summary	122
16. The Basics of Object Oriented Programming in Kotlin	123
16.1 What is an Object?	123
16.2 What is a Class?	123
16.3 Declaring a Kotlin Class.....	123
16.4 Adding Properties to a Class.....	124
16.5 Defining Methods	124
16.6 Declaring and Initializing a Class Instance.....	124
16.7 Primary and Secondary Constructors.....	124
16.8 Initializer Blocks.....	127
16.9 Calling Methods and Accessing Properties	127
16.10 Custom Accessors	127
16.11 Nested and Inner Classes	128
16.12 Companion Objects.....	129
16.13 Summary	131
17. An Introduction to Kotlin Inheritance and Subclassing.....	133
17.1 Inheritance, Classes and Subclasses.....	133
17.2 Subclassing Syntax	133
17.3 A Kotlin Inheritance Example.....	134
17.4 Extending the Functionality of a Subclass	135
17.5 Overriding Inherited Methods.....	136
17.6 Adding a Custom Secondary Constructor.....	137
17.7 Using the SavingsAccount Class	137
17.8 Summary	137
18. An Overview of Android View Binding.....	139
18.1 Find View by Id	139
18.2 View Binding	139
18.3 Converting the AndroidSample project.....	140
18.4 Enabling View Binding.....	140
18.5 Using View Binding	140
18.6 Choosing an Option	142

Table of Contents

18.7 View Binding in the Book Examples	142
18.8 Migrating a Project to View Binding	142
18.9 Summary	143
19. Understanding Android Application and Activity Lifecycles	145
19.1 Android Applications and Resource Management	145
19.2 Android Process States	145
19.2.1 Foreground Process	146
19.2.2 Visible Process	146
19.2.3 Service Process	146
19.2.4 Background Process.....	146
19.2.5 Empty Process	147
19.3 Inter-Process Dependencies	147
19.4 The Activity Lifecycle.....	147
19.5 The Activity Stack.....	147
19.6 Activity States	148
19.7 Configuration Changes	148
19.8 Handling State Change.....	149
19.9 Summary	149
20. Handling Android Activity State Changes.....	151
20.1 New vs. Old Lifecycle Techniques.....	151
20.2 The Activity and Fragment Classes.....	151
20.3 Dynamic State vs. Persistent State.....	153
20.4 The Android Lifecycle Methods.....	153
20.5 Lifetimes	155
20.6 Foldable Devices and Multi-Resume	156
20.7 Disabling Configuration Change Restarts	156
20.8 Lifecycle Method Limitations.....	156
20.9 Summary	157
21. Android Activity State Changes by Example	159
21.1 Creating the State Change Example Project	159
21.2 Designing the User Interface	160
21.3 Overriding the Activity Lifecycle Methods	161
21.4 Filtering the Logcat Panel.....	163
21.5 Running the Application.....	164
21.6 Experimenting with the Activity.....	165
21.7 Summary	166
22. Saving and Restoring the State of an Android Activity	167
22.1 Saving Dynamic State	167
22.2 Default Saving of User Interface State	167
22.3 The Bundle Class	168
22.4 Saving the State.....	169
22.5 Restoring the State	170
22.6 Testing the Application.....	170
22.7 Summary	170
23. Understanding Android Views, View Groups and Layouts	171

23.1 Designing for Different Android Devices	171
23.2 Views and View Groups	171
23.3 Android Layout Managers	171
23.4 The View Hierarchy	173
23.5 Creating User Interfaces	174
23.6 Summary	174
24. A Guide to the Android Studio Layout Editor	175
24.1 Basic vs. Empty Views Activity Templates	175
24.2 The Android Studio Layout Editor	179
24.3 Design Mode	179
24.4 The Palette	180
24.5 Design Mode and Layout Views	181
24.6 Night Mode	182
24.7 Code Mode	182
24.8 Split Mode	183
24.9 Setting Attributes	183
24.10 Transforms	185
24.11 Tools Visibility Toggles	186
24.12 Converting Views	187
24.13 Displaying Sample Data	188
24.14 Creating a Custom Device Definition	189
24.15 Changing the Current Device	189
24.16 Layout Validation	190
24.17 Summary	191
25. A Guide to the Android ConstraintLayout	193
25.1 How ConstraintLayout Works	193
25.1.1 Constraints	193
25.1.2 Margins	194
25.1.3 Opposing Constraints	194
25.1.4 Constraint Bias	195
25.1.5 Chains	196
25.1.6 Chain Styles	196
25.2 Baseline Alignment	197
25.3 Configuring Widget Dimensions	197
25.4 Guideline Helper	198
25.5 Group Helper	198
25.6 Barrier Helper	198
25.7 Flow Helper	200
25.8 Ratios	201
25.9 ConstraintLayout Advantages	201
25.10 ConstraintLayout Availability	202
25.11 Summary	202
26. A Guide to Using ConstraintLayout in Android Studio	203
26.1 Design and Layout Views	203
26.2 Autoconnect Mode	205
26.3 Inference Mode	205

Table of Contents

26.4 Manipulating Constraints Manually.....	205
26.5 Adding Constraints in the Inspector	207
26.6 Viewing Constraints in the Attributes Window.....	207
26.7 Deleting Constraints.....	208
26.8 Adjusting Constraint Bias	209
26.9 Understanding ConstraintLayout Margins.....	209
26.10 The Importance of Opposing Constraints and Bias	211
26.11 Configuring Widget Dimensions.....	213
26.12 Design Time Tools Positioning.....	214
26.13 Adding Guidelines	215
26.14 Adding Barriers	217
26.15 Adding a Group.....	218
26.16 Working with the Flow Helper	219
26.17 Widget Group Alignment and Distribution.....	219
26.18 Converting other Layouts to ConstraintLayout.....	221
26.19 Summary	221
27. Working with ConstraintLayout Chains and Ratios in Android Studio	223
27.1 Creating a Chain.....	223
27.2 Changing the Chain Style	225
27.3 Spread Inside Chain Style.....	226
27.4 Packed Chain Style.....	226
27.5 Packed Chain Style with Bias.....	226
27.6 Weighted Chain	226
27.7 Working with Ratios	227
27.8 Summary	229
28. An Android Studio Layout Editor ConstraintLayout Tutorial	231
28.1 An Android Studio Layout Editor Tool Example	231
28.2 Preparing the Layout Editor Environment	231
28.3 Adding the Widgets to the User Interface.....	232
28.4 Adding the Constraints	235
28.5 Testing the Layout	237
28.6 Using the Layout Inspector	237
28.7 Summary	238
29. Manual XML Layout Design in Android Studio	239
29.1 Manually Creating an XML Layout	239
29.2 Manual XML vs. Visual Layout Design.....	242
29.3 Summary	242
30. Managing Constraints using Constraint Sets.....	243
30.1 Kotlin Code vs. XML Layout Files.....	243
30.2 Creating Views.....	243
30.3 View Attributes.....	244
30.4 Constraint Sets.....	244
30.4.1 Establishing Connections.....	244
30.4.2 Applying Constraints to a Layout	244
30.4.3 Parent Constraint Connections.....	244
30.4.4 Sizing Constraints	245

30.4.5 Constraint Bias	245
30.4.6 Alignment Constraints	245
30.4.7 Copying and Applying Constraint Sets	245
30.4.8 ConstraintLayout Chains	245
30.4.9 Guidelines	246
30.4.10 Removing Constraints	246
30.4.11 Scaling	246
30.4.12 Rotation	247
30.5 Summary	247
31. An Android ConstraintSet Tutorial.....	249
31.1 Creating the Example Project in Android Studio	249
31.2 Adding Views to an Activity	249
31.3 Setting View Attributes	250
31.4 Creating View IDs	251
31.5 Configuring the Constraint Set	252
31.6 Adding the EditText View	253
31.7 Converting Density Independent Pixels (dp) to Pixels (px)	254
31.8 Summary	255
32. A Guide to Using Apply Changes in Android Studio	257
32.1 Introducing Apply Changes	257
32.2 Understanding Apply Changes Options	257
32.3 Using Apply Changes	258
32.4 Configuring Apply Changes Fallback Settings	259
32.5 An Apply Changes Tutorial	259
32.6 Using Apply Code Changes	259
32.7 Using Apply Changes and Restart Activity	260
32.8 Using Run App	260
32.9 Summary	260
33. A Guide to Gradle Version Catalogs	261
33.1 Library and Plugin Dependencies	261
33.2 Project Gradle Build File	261
33.3 Module Gradle Build Files	261
33.4 Version Catalog File	262
33.5 Adding Dependencies	263
33.6 Library Updates	264
33.7 Summary	264
34. An Overview and Example of Android Event Handling	265
34.1 Understanding Android Events	265
34.2 Using the android:onClick Resource	265
34.3 Event Listeners and Callback Methods	266
34.4 An Event Handling Example	266
34.5 Designing the User Interface	267
34.6 The Event Listener and Callback Method	267
34.7 Consuming Events	269
34.8 Summary	270

35. Android Touch and Multi-touch Event Handling	271
35.1 Intercepting Touch Events	271
35.2 The MotionEvent Object	272
35.3 Understanding Touch Actions.....	272
35.4 Handling Multiple Touches	272
35.5 An Example Multi-Touch Application	273
35.6 Designing the Activity User Interface	273
35.7 Implementing the Touch Event Listener.....	273
35.8 Running the Example Application.....	276
35.9 Summary	276
36. Detecting Common Gestures Using the Android Gesture Detector Class	277
36.1 Implementing Common Gesture Detection.....	277
36.2 Creating an Example Gesture Detection Project	278
36.3 Implementing the Listener Class.....	278
36.4 Creating the GestureDetector Instance	280
36.5 Implementing the onTouchEvent() Method.....	280
36.6 Testing the Application.....	281
36.7 Summary	281
37. Implementing Custom Gesture and Pinch Recognition on Android	283
37.1 The Android Gesture Builder Application.....	283
37.2 The GestureOverlayView Class	283
37.3 Detecting Gestures.....	283
37.4 Identifying Specific Gestures	283
37.5 Installing and Running the Gesture Builder Application	283
37.6 Creating a Gestures File	284
37.7 Creating the Example Project.....	284
37.8 Extracting the Gestures File from the SD Card	284
37.9 Adding the Gestures File to the Project	285
37.10 Designing the User Interface	285
37.11 Loading the Gestures File	286
37.12 Registering the Event Listener.....	287
37.13 Implementing the onGesturePerformed Method.....	287
37.14 Testing the Application.....	288
37.15 Configuring the GestureOverlayView.....	288
37.16 Intercepting Gestures.....	288
37.17 Detecting Pinch Gestures.....	289
37.18 A Pinch Gesture Example Project.....	289
37.19 Summary	291
38. An Introduction to Android Fragments.....	293
38.1 What is a Fragment?	293
38.2 Creating a Fragment	293
38.3 Adding a Fragment to an Activity using the Layout XML File.....	294
38.4 Adding and Managing Fragments in Code	296
38.5 Handling Fragment Events	297
38.6 Implementing Fragment Communication.....	297
38.7 Summary	299

39. Using Fragments in Android Studio - An Example	301
39.1 About the Example Fragment Application	301
39.2 Creating the Example Project	301
39.3 Creating the First Fragment Layout.....	301
39.4 Migrating a Fragment to View Binding	303
39.5 Adding the Second Fragment	304
39.6 Adding the Fragments to the Activity	305
39.7 Making the Toolbar Fragment Talk to the Activity	306
39.8 Making the Activity Talk to the Text Fragment	309
39.9 Testing the Application.....	310
39.10 Summary	310
40. Modern Android App Architecture with Jetpack	311
40.1 What is Android Jetpack?	311
40.2 The “Old” Architecture	311
40.3 Modern Android Architecture	311
40.4 The ViewModel Component	312
40.5 The LiveData Component	312
40.6 ViewModel Saved State.....	313
40.7 LiveData and Data Binding.....	313
40.8 Android Lifecycles	314
40.9 Repository Modules.....	314
40.10 Summary	315
41. An Android ViewModel Tutorial	317
41.1 About the Project	317
41.2 Creating the ViewModel Example Project.....	317
41.3 Removing Unwanted Project Elements.....	317
41.4 Designing the Fragment Layout.....	318
41.5 Implementing the View Model.....	319
41.6 Associating the Fragment with the View Model.....	320
41.7 Modifying the Fragment	321
41.8 Accessing the ViewModel Data.....	321
41.9 Testing the Project.....	322
41.10 Summary	322
42. An Android Jetpack LiveData Tutorial	323
42.1 LiveData - A Recap	323
42.2 Adding LiveData to the ViewModel	323
42.3 Implementing the Observer.....	325
42.4 Summary	326
43. An Overview of Android Jetpack Data Binding	327
43.1 An Overview of Data Binding.....	327
43.2 The Key Components of Data Binding	327
43.2.1 The Project Build Configuration	327
43.2.2 The Data Binding Layout File.....	328
43.2.3 The Layout File Data Element	329
43.2.4 The Binding Classes	330

Table of Contents

43.2.5 Data Binding Variable Configuration.....	330
43.2.6 Binding Expressions (One-Way).....	331
43.2.7 Binding Expressions (Two-Way).....	332
43.2.8 Event and Listener Bindings.....	332
43.3 Summary.....	333
44. An Android Jetpack Data Binding Tutorial.....	335
44.1 Removing the Redundant Code.....	335
44.2 Enabling Data Binding.....	336
44.3 Adding the Layout Element.....	337
44.4 Adding the Data Element to Layout File.....	338
44.5 Working with the Binding Class.....	338
44.6 Assigning the ViewModel Instance to the Data Binding Variable.....	339
44.7 Adding Binding Expressions.....	340
44.8 Adding the Conversion Method.....	340
44.9 Adding a Listener Binding.....	341
44.10 Testing the App.....	341
44.11 Summary.....	341
45. An Android ViewModel Saved State Tutorial.....	343
45.1 Understanding ViewModel State Saving.....	343
45.2 Implementing ViewModel State Saving.....	343
45.3 Saving and Restoring State.....	344
45.4 Adding Saved State Support to the ViewModelDemo Project.....	345
45.5 Summary.....	346
46. Working with Android Lifecycle-Aware Components.....	347
46.1 Lifecycle Awareness.....	347
46.2 Lifecycle Owners.....	347
46.3 Lifecycle Observers.....	348
46.4 Lifecycle States and Events.....	348
46.5 Summary.....	349
47. An Android Jetpack Lifecycle Awareness Tutorial.....	351
47.1 Creating the Example Lifecycle Project.....	351
47.2 Creating a Lifecycle Observer.....	351
47.3 Adding the Observer.....	352
47.4 Testing the Observer.....	353
47.5 Creating a Lifecycle Owner.....	353
47.6 Testing the Custom Lifecycle Owner.....	355
47.7 Summary.....	355
48. An Overview of the Navigation Architecture Component.....	357
48.1 Understanding Navigation.....	357
48.2 Declaring a Navigation Host.....	358
48.3 The Navigation Graph.....	360
48.4 Accessing the Navigation Controller.....	361
48.5 Triggering a Navigation Action.....	361
48.6 Passing Arguments.....	362
48.7 Summary.....	362

49. An Android Jetpack Navigation Component Tutorial	363
49.1 Creating the NavigationDemo Project	363
49.2 Adding Navigation to the Build Configuration.....	363
49.3 Creating the Navigation Graph Resource File.....	364
49.4 Declaring a Navigation Host.....	365
49.5 Adding Navigation Destinations.....	366
49.6 Designing the Destination Fragment Layouts.....	368
49.7 Adding an Action to the Navigation Graph.....	370
49.8 Implement the OnFragmentManagerInteractionListener	371
49.9 Adding View Binding Support to the Destination Fragments	372
49.10 Triggering the Action	373
49.11 Passing Data Using Safeargs	373
49.12 Summary	376
50. An Introduction to MotionLayout.....	377
50.1 An Overview of MotionLayout	377
50.2 MotionLayout	377
50.3 MotionScene	377
50.4 Configuring ConstraintSets	378
50.5 Custom Attributes.....	379
50.6 Triggering an Animation.....	381
50.7 Arc Motion.....	382
50.8 Keyframes.....	382
50.8.1 Attribute Keyframes.....	382
50.8.2 Position Keyframes	383
50.9 Time Linearity	386
50.10 KeyTrigger	386
50.11 Cycle and Time Cycle Keyframes	387
50.12 Starting an Animation from Code.....	387
50.13 Summary	388
51. An Android MotionLayout Editor Tutorial.....	389
51.1 Creating the MotionLayoutDemo Project	389
51.2 ConstraintLayout to MotionLayout Conversion	389
51.3 Configuring Start and End Constraints	391
51.4 Previewing the MotionLayout Animation	394
51.5 Adding an OnClick Gesture	394
51.6 Adding an Attribute Keyframe to the Transition.....	396
51.7 Adding a CustomAttribute to a Transition.....	398
51.8 Adding Position Keyframes	400
51.9 Summary	402
52. A MotionLayout KeyCycle Tutorial	403
52.1 An Overview of Cycle Keyframes	403
52.2 Using the Cycle Editor	407
52.3 Creating the KeyCycleDemo Project.....	408
52.4 Configuring the Start and End Constraints.....	408
52.5 Creating the Cycles	410
52.6 Previewing the Animation	412

Table of Contents

52.7 Adding the KeyFrameSet to the MotionScene	412
52.8 Summary	414
53. Working with the Floating Action Button and Snackbar	415
53.1 The Material Design.....	415
53.2 The Design Library	415
53.3 The Floating Action Button (FAB)	415
53.4 The Snackbar.....	416
53.5 Creating the Example Project.....	417
53.6 Reviewing the Project	417
53.7 Removing Navigation Features.....	418
53.8 Changing the Floating Action Button	418
53.9 Adding an Action to the Snackbar.....	420
53.10 Summary.....	420
54. Creating a Tabbed Interface using the TabLayout Component	421
54.1 An Introduction to the ViewPager2	421
54.2 An Overview of the TabLayout Component	421
54.3 Creating the TabLayoutDemo Project.....	422
54.4 Creating the First Fragment.....	422
54.5 Duplicating the Fragments.....	424
54.6 Adding the TabLayout and ViewPager2.....	425
54.7 Performing the Initialization Tasks.....	427
54.8 Testing the Application.....	429
54.9 Customizing the TabLayout.....	429
54.10 Summary.....	430
55. Working with the RecyclerView and CardView Widgets	431
55.1 An Overview of the RecyclerView	431
55.2 An Overview of the CardView	433
55.3 Summary	434
56. An Android RecyclerView and CardView Tutorial	435
56.1 Creating the CardDemo Project.....	435
56.2 Modifying the Basic Views Activity Project	435
56.3 Designing the CardView Layout	436
56.4 Adding the RecyclerView.....	437
56.5 Adding the Image Files.....	437
56.6 Creating the RecyclerView Adapter.....	437
56.7 Initializing the RecyclerView Component.....	439
56.8 Testing the Application.....	440
56.9 Responding to Card Selections.....	441
56.10 Summary.....	442
57. Working with the AppBar and Collapsing Toolbar Layouts	443
57.1 The Anatomy of an AppBar	443
57.2 The Example Project	444
57.3 Coordinating the RecyclerView and Toolbar	444
57.4 Introducing the Collapsing Toolbar Layout	446
57.5 Changing the Title and Scrim Color	449

57.6 Summary	450
58. An Overview of Android Intents	451
58.1 An Overview of Intents	451
58.2 Explicit Intents.....	451
58.3 Returning Data from an Activity	452
58.4 Implicit Intents	453
58.5 Using Intent Filters.....	454
58.6 Automatic Link Verification	454
58.7 Manually Enabling Links	457
58.8 Checking Intent Availability	458
58.9 Summary	459
59. Android Explicit Intents – A Worked Example	461
59.1 Creating the Explicit Intent Example Application	461
59.2 Designing the User Interface Layout for MainActivity	461
59.3 Creating the Second Activity Class.....	462
59.4 Designing the User Interface Layout for SecondActivity	463
59.5 Reviewing the Application Manifest File	463
59.6 Creating the Intent	464
59.7 Extracting Intent Data	465
59.8 Launching SecondActivity as a Sub-Activity.....	466
59.9 Returning Data from a Sub-Activity.....	467
59.10 Testing the Application.....	467
59.11 Summary	467
60. Android Implicit Intents – A Worked Example	469
60.1 Creating the Android Studio Implicit Intent Example Project	469
60.2 Designing the User Interface	469
60.3 Creating the Implicit Intent	470
60.4 Adding a Second Matching Activity	470
60.5 Adding the Web View to the UI.....	471
60.6 Obtaining the Intent URL	471
60.7 Modifying the MyWebView Project Manifest File	473
60.8 Installing the MyWebView Package on a Device	474
60.9 Testing the Application.....	475
60.10 Manually Enabling the Link	475
60.11 Automatic Link Verification	477
60.12 Summary	479
61. Android Broadcast Intents and Broadcast Receivers	481
61.1 An Overview of Broadcast Intents	481
61.2 An Overview of Broadcast Receivers	482
61.3 Obtaining Results from a Broadcast	483
61.4 Sticky Broadcast Intents	483
61.5 The Broadcast Intent Example.....	483
61.6 Creating the Example Application	484
61.7 Creating and Sending the Broadcast Intent	484
61.8 Creating the Broadcast Receiver	485
61.9 Registering the Broadcast Receiver.....	486

Table of Contents

61.10 Testing the Broadcast Example	486
61.11 Listening for System Broadcasts.....	487
61.12 Summary.....	487
62. An Introduction to Kotlin Coroutines.....	489
62.1 What are Coroutines?.....	489
62.2 Threads vs. Coroutines.....	489
62.3 Coroutine Scope.....	490
62.4 Suspend Functions.....	490
62.5 Coroutine Dispatchers.....	490
62.6 Coroutine Builders.....	491
62.7 Jobs.....	491
62.8 Coroutines – Suspending and Resuming.....	492
62.9 Returning Results from a Coroutine	493
62.10 Using withContext.....	493
62.11 Coroutine Channel Communication	495
62.12 Summary.....	496
63. An Android Kotlin Coroutines Tutorial.....	497
63.1 Creating the Coroutine Example Application.....	497
63.2 Designing the User Interface	497
63.3 Implementing the SeekBar.....	498
63.4 Adding the Suspend Function.....	499
63.5 Implementing the launchCoroutines Method.....	500
63.6 Testing the App.....	500
63.7 Summary	501
64. An Overview of Android Services.....	503
64.1 Intent Service.....	503
64.2 Bound Service.....	503
64.3 The Anatomy of a Service	504
64.4 Controlling Destroyed Service Restart Options.....	504
64.5 Declaring a Service in the Manifest File.....	504
64.6 Starting a Service Running on System Startup.....	505
64.7 Summary	506
65. Android Local Bound Services – A Worked Example.....	507
65.1 Understanding Bound Services.....	507
65.2 Bound Service Interaction Options.....	507
65.3 A Local Bound Service Example.....	507
65.4 Adding a Bound Service to the Project	508
65.5 Implementing the Binder	508
65.6 Binding the Client to the Service	510
65.7 Completing the Example.....	511
65.8 Testing the Application.....	512
65.9 Summary	513
66. Android Remote Bound Services – A Worked Example	515
66.1 Client to Remote Service Communication.....	515
66.2 Creating the Example Application.....	515

66.3 Designing the User Interface	515
66.4 Implementing the Remote Bound Service	515
66.5 Configuring a Remote Service in the Manifest File	517
66.6 Launching and Binding to the Remote Service	517
66.7 Sending a Message to the Remote Service	519
66.8 Summary	519
67. An Introduction to Kotlin Flow	521
67.1 Understanding Flows	521
67.2 Creating the Sample Project	521
67.3 Adding the Kotlin Lifecycle Library	522
67.4 Declaring a Flow	522
67.5 Emitting Flow Data	523
67.6 Collecting Flow Data	523
67.7 Adding a Flow Buffer	525
67.8 Transforming Data with Intermediaries	526
67.9 Terminal Flow Operators	528
67.10 Flow Flattening	528
67.11 Combining Multiple Flows	530
67.12 Hot and Cold Flows	531
67.13 StateFlow	531
67.14 SharedFlow	532
67.15 Summary	534
68. An Android SharedFlow Tutorial	535
68.1 About the Project	535
68.2 Creating the SharedFlowDemo Project	535
68.3 Adding the Lifecycle Libraries	535
68.4 Designing the User Interface Layout	536
68.5 Adding the List Row Layout	536
68.6 Adding the RecyclerView Adapter	537
68.7 Adding the ViewModel	537
68.8 Configuring the ViewModelProvider	538
68.9 Collecting the Flow Values	539
68.10 Testing the SharedFlowDemo App	540
68.11 Handling Flows in the Background	541
68.12 Summary	543
69. An Overview of Android SQLite Databases	545
69.1 Understanding Database Tables	545
69.2 Introducing Database Schema	545
69.3 Columns and Data Types	545
69.4 Database Rows	546
69.5 Introducing Primary Keys	546
69.6 What is SQLite?	546
69.7 Structured Query Language (SQL)	546
69.8 Trying SQLite on an Android Virtual Device (AVD)	547
69.9 Android SQLite Classes	548
69.9.1 Cursor	549

Table of Contents

69.9.2 SQLiteDatabase	549
69.9.3 SQLiteOpenHelper	549
69.9.4 ContentValues.....	550
69.10 The Android Room Persistence Library.....	550
69.11 Summary.....	550
70. An Android SQLite Database Tutorial	551
70.1 About the Database Example.....	551
70.2 Creating the SQLDemo Project.....	551
70.3 Designing the User interface	551
70.4 Creating the Data Model.....	552
70.5 Implementing the Data Handler.....	552
70.6 The Add Handler Method.....	554
70.7 The Query Handler Method	555
70.8 The Delete Handler Method	555
70.9 Implementing the Activity Event Methods.....	556
70.10 Testing the Application.....	557
70.11 Summary.....	557
71. Understanding Android Content Providers.....	559
71.1 What is a Content Provider?.....	559
71.2 The Content Provider	559
71.2.1 onCreate()	559
71.2.2 query()	559
71.2.3 insert()	559
71.2.4 update()	560
71.2.5 delete()	560
71.2.6 getType()	560
71.3 The Content URI	560
71.4 The Content Resolver	560
71.5 The <provider> Manifest Element	561
71.6 Summary	561
72. An Android Content Provider Tutorial.....	563
72.1 Copying the SQLDemo Project.....	563
72.2 Adding the Content Provider Package.....	563
72.3 Creating the Content Provider Class.....	564
72.4 Constructing the Authority and Content URI.....	565
72.5 Implementing URI Matching in the Content Provider.....	566
72.6 Implementing the Content Provider onCreate() Method	567
72.7 Implementing the Content Provider insert() Method	567
72.8 Implementing the Content Provider query() Method	568
72.9 Implementing the Content Provider update() Method	569
72.10 Implementing the Content Provider delete() Method.....	570
72.11 Declaring the Content Provider in the Manifest File	571
72.12 Modifying the Database Handler.....	572
72.13 Summary.....	574
73. An Android Content Provider Client Tutorial.....	575
73.1 Creating the SQLDemoClient Project.....	575

73.2 Designing the User interface	575
73.3 Accessing the Content Provider	575
73.4 Adding the Query Permission.....	576
73.5 Testing the Project.....	577
73.6 Summary	577
74. The Android Room Persistence Library	579
74.1 Revisiting Modern App Architecture	579
74.2 Key Elements of Room Database Persistence.....	579
74.2.1 Repository	580
74.2.2 Room Database	580
74.2.3 Data Access Object (DAO)	580
74.2.4 Entities.....	580
74.2.5 SQLite Database	580
74.3 Understanding Entities.....	581
74.4 Data Access Objects.....	583
74.5 The Room Database.....	584
74.6 The Repository.....	585
74.7 In-Memory Databases.....	586
74.8 Database Inspector.....	586
74.9 Summary	587
75. An Android TableLayout and TableRow Tutorial	589
75.1 The TableLayout and TableRow Layout Views.....	589
75.2 Creating the Room Database Project	590
75.3 Converting to a LinearLayout.....	590
75.4 Adding the TableLayout to the User Interface.....	591
75.5 Configuring the TableRows	592
75.6 Adding the Button Bar to the Layout	593
75.7 Adding the RecyclerView.....	594
75.8 Adjusting the Layout Margins.....	595
75.9 Summary	595
76. An Android Room Database and Repository Tutorial.....	597
76.1 About the RoomDemo Project.....	597
76.2 Modifying the Build Configuration.....	597
76.3 Building the Entity	598
76.4 Creating the Data Access Object.....	600
76.5 Adding the Room Database.....	601
76.6 Adding the Repository	602
76.7 Adding the ViewModel	605
76.8 Creating the Product Item Layout	606
76.9 Adding the RecyclerView Adapter.....	606
76.10 Preparing the Main Activity	607
76.11 Adding the Button Listeners.....	608
76.12 Adding LiveData Observers	609
76.13 Initializing the RecyclerView.....	610
76.14 Testing the RoomDemo App.....	610
76.15 Using the Database Inspector.....	610

Table of Contents

76.16 Summary	611
77. Video Playback on Android using the VideoView and MediaController Classes	613
77.1 Introducing the Android VideoView Class	613
77.2 Introducing the Android MediaController Class	614
77.3 Creating the Video Playback Example	614
77.4 Designing the VideoPlayer Layout	614
77.5 Downloading the Video File	615
77.6 Configuring the VideoView	615
77.7 Adding the MediaController to the Video View	617
77.8 Setting up the onPreparedListener	617
77.9 Summary	618
78. Android Picture-in-Picture Mode	619
78.1 Picture-in-Picture Features	619
78.2 Enabling Picture-in-Picture Mode	620
78.3 Configuring Picture-in-Picture Parameters	620
78.4 Entering Picture-in-Picture Mode	621
78.5 Detecting Picture-in-Picture Mode Changes	621
78.6 Adding Picture-in-Picture Actions	621
78.7 Summary	622
79. An Android Picture-in-Picture Tutorial	623
79.1 Adding Picture-in-Picture Support to the Manifest	623
79.2 Adding a Picture-in-Picture Button	623
79.3 Entering Picture-in-Picture Mode	624
79.4 Detecting Picture-in-Picture Mode Changes	625
79.5 Adding a Broadcast Receiver	625
79.6 Adding the PiP Action	626
79.7 Testing the Picture-in-Picture Action	629
79.8 Summary	629
80. Making Runtime Permission Requests in Android	631
80.1 Understanding Normal and Dangerous Permissions	631
80.2 Creating the Permissions Example Project	633
80.3 Checking for a Permission	633
80.4 Requesting Permission at Runtime	635
80.5 Providing a Rationale for the Permission Request	636
80.6 Testing the Permissions App	637
80.7 Summary	638
81. Android Audio Recording and Playback using MediaPlayer and MediaRecorder	639
81.1 Playing Audio	639
81.2 Recording Audio and Video using the MediaRecorder Class	640
81.3 About the Example Project	641
81.4 Creating the AudioApp Project	641
81.5 Designing the User Interface	641
81.6 Checking for Microphone Availability	642
81.7 Initializing the Activity	643
81.8 Implementing the recordAudio() Method	644

81.9 Implementing the stopAudio() Method.....	644
81.10 Implementing the playAudio() method.....	645
81.11 Configuring and Requesting Permissions	645
81.12 Testing the Application.....	647
81.13 Summary.....	647
82. An Android Notifications Tutorial	649
82.1 An Overview of Notifications.....	649
82.2 Creating the NotifyDemo Project.....	651
82.3 Designing the User Interface	651
82.4 Creating the Second Activity.....	651
82.5 Creating a Notification Channel	652
82.6 Requesting Notification Permission	653
82.7 Creating and Issuing a Notification.....	656
82.8 Launching an Activity from a Notification.....	658
82.9 Adding Actions to a Notification	660
82.10 Bundled Notifications.....	660
82.11 Summary.....	662
83. An Android Direct Reply Notification Tutorial	663
83.1 Creating the DirectReply Project.....	663
83.2 Designing the User Interface	663
83.3 Requesting Notification Permission	664
83.4 Creating the Notification Channel.....	665
83.5 Building the RemoteInput Object.....	666
83.6 Creating the PendingIntent.....	667
83.7 Creating the Reply Action.....	668
83.8 Receiving Direct Reply Input.....	669
83.9 Updating the Notification	670
83.10 Summary.....	671
84. Working with the Google Maps Android API in Android Studio	673
84.1 The Elements of the Google Maps Android API	673
84.2 Creating the Google Maps Project.....	674
84.3 Creating a Google Cloud Billing Account	674
84.4 Creating a New Google Cloud Project	675
84.5 Enabling the Google Maps SDK.....	676
84.6 Generating a Google Maps API Key.....	677
84.7 Adding the API Key to the Android Studio Project.....	678
84.8 Testing the Application.....	678
84.9 Understanding Geocoding and Reverse Geocoding.....	678
84.10 Adding a Map to an Application.....	680
84.11 Requesting Current Location Permission.....	680
84.12 Displaying the User's Current Location	682
84.13 Changing the Map Type.....	683
84.14 Displaying Map Controls to the User.....	684
84.15 Handling Map Gesture Interaction.....	684
84.15.1 Map Zooming Gestures.....	684
84.15.2 Map Scrolling/Panning Gestures	685

Table of Contents

84.15.3 Map Tilt Gestures.....	685
84.15.4 Map Rotation Gestures.....	685
84.16 Creating Map Markers.....	685
84.17 Controlling the Map Camera	686
84.18 Summary.....	687
85. Printing with the Android Printing Framework	689
85.1 The Android Printing Architecture	689
85.2 The Print Service Plugins	689
85.3 Google Cloud Print.....	690
85.4 Printing to Google Drive.....	690
85.5 Save as PDF	691
85.6 Printing from Android Devices	691
85.7 Options for Building Print Support into Android Apps.....	692
85.7.1 Image Printing.....	692
85.7.2 Creating and Printing HTML Content	693
85.7.3 Printing a Web Page.....	694
85.7.4 Printing a Custom Document	695
85.8 Summary	695
86. An Android HTML and Web Content Printing Example	697
86.1 Creating the HTML Printing Example Application	697
86.2 Printing Dynamic HTML Content	697
86.3 Creating the Web Page Printing Example.....	700
86.4 Removing the Floating Action Button	700
86.5 Removing Navigation Features.....	700
86.6 Designing the User Interface Layout	701
86.7 Accessing the WebView from the Main Activity	702
86.8 Loading the Web Page into the WebView.....	702
86.9 Adding the Print Menu Option.....	703
86.10 Summary.....	705
87. A Guide to Android Custom Document Printing.....	707
87.1 An Overview of Android Custom Document Printing	707
87.1.1 Custom Print Adapters.....	707
87.2 Preparing the Custom Document Printing Project.....	708
87.3 Designing the UI	708
87.4 Creating the Custom Print Adapter.....	709
87.5 Implementing the onLayout() Callback Method	710
87.6 Implementing the onWrite() Callback Method	713
87.7 Checking a Page is in Range	715
87.8 Drawing the Content on the Page Canvas	716
87.9 Starting the Print Job	718
87.10 Testing the Application.....	719
87.11 Summary.....	719
88. An Introduction to Android App Links.....	721
88.1 An Overview of Android App Links	721
88.2 App Link Intent Filters	721
88.3 Handling App Link Intents	722

88.4 Associating the App with a Website.....	722
88.5 Summary	723
89. An Android Studio App Links Tutorial	725
89.1 About the Example App	725
89.2 The Database Schema	725
89.3 Loading and Running the Project.....	725
89.4 Adding the URL Mapping.....	727
89.5 Adding the Intent Filter.....	730
89.6 Adding Intent Handling Code.....	730
89.7 Testing the App.....	733
89.8 Creating the Digital Asset Links File	733
89.9 Testing the App Link.....	734
89.10 Summary	734
90. An Android Biometric Authentication Tutorial.....	735
90.1 An Overview of Biometric Authentication.....	735
90.2 Creating the Biometric Authentication Project	735
90.3 Configuring Device Fingerprint Authentication	736
90.4 Adding the Biometric Permission to the Manifest File.....	736
90.5 Designing the User Interface	737
90.6 Adding a Toast Convenience Method.....	737
90.7 Checking the Security Settings.....	738
90.8 Configuring the Authentication Callbacks.....	739
90.9 Adding the CancellationSignal.....	740
90.10 Starting the Biometric Prompt	740
90.11 Testing the Project.....	741
90.12 Summary	742
91. Creating, Testing, and Uploading an Android App Bundle	743
91.1 The Release Preparation Process.....	743
91.2 Android App Bundles.....	743
91.3 Register for a Google Play Developer Console Account.....	744
91.4 Configuring the App in the Console	745
91.5 Enabling Google Play App Signing.....	746
91.6 Creating a Keystore File	746
91.7 Creating the Android App Bundle.....	747
91.8 Generating Test APK Files	749
91.9 Uploading the App Bundle to the Google Play Developer Console.....	750
91.10 Exploring the App Bundle	751
91.11 Managing Testers	752
91.12 Rolling the App Out for Testing.....	752
91.13 Uploading New App Bundle Revisions.....	753
91.14 Analyzing the App Bundle File	754
91.15 Summary	755
92. An Overview of Android In-App Billing	757
92.1 Preparing a Project for In-App Purchasing	757
92.2 Creating In-App Products and Subscriptions	757
92.3 Billing Client Initialization.....	758

Table of Contents

92.4 Connecting to the Google Play Billing Library.....	759
92.5 Querying Available Products.....	759
92.6 Starting the Purchase Process.....	760
92.7 Completing the Purchase.....	760
92.8 Querying Previous Purchases.....	761
92.9 Summary.....	762
93. An Android In-App Purchasing Tutorial.....	763
93.1 About the In-App Purchasing Example Project.....	763
93.2 Creating the InAppPurchase Project.....	763
93.3 Adding Libraries to the Project.....	763
93.4 Designing the User Interface.....	764
93.5 Adding the App to the Google Play Store.....	765
93.6 Creating an In-App Product.....	765
93.7 Enabling License Testers.....	766
93.8 Initializing the Billing Client.....	767
93.9 Querying the Product.....	768
93.10 Launching the Purchase Flow.....	769
93.11 Handling Purchase Updates.....	770
93.12 Consuming the Product.....	771
93.13 Restoring a Previous Purchase.....	772
93.14 Testing the App.....	773
93.15 Troubleshooting.....	774
93.16 Summary.....	774
94. Accessing Cloud Storage using the Android Storage Access Framework.....	775
94.1 The Storage Access Framework.....	775
94.2 Working with the Storage Access Framework.....	776
94.3 Filtering Picker File Listings.....	776
94.4 Handling Intent Results.....	777
94.5 Reading the Content of a File.....	777
94.6 Writing Content to a File.....	778
94.7 Deleting a File.....	779
94.8 Gaining Persistent Access to a File.....	779
94.9 Summary.....	779
95. An Android Storage Access Framework Example.....	781
95.1 About the Storage Access Framework Example.....	781
95.2 Creating the Storage Access Framework Example.....	781
95.3 Designing the User Interface.....	781
95.4 Adding the Activity Launchers.....	782
95.5 Creating a New Storage File.....	783
95.6 Saving to a Storage File.....	784
95.7 Opening and Reading a Storage File.....	785
95.8 Testing the Storage Access Application.....	786
95.9 Summary.....	788
96. An Android Studio Primary/Detail Flow Tutorial.....	789
96.1 The Primary/Detail Flow.....	789
96.2 Creating a Primary/Detail Flow Activity.....	790

96.3 Adding the Primary/Detail Flow Activity.....	790
96.4 Modifying the Primary/Detail Flow Template.....	791
96.5 Changing the Content Model.....	791
96.6 Changing the Detail Pane.....	793
96.7 Modifying the ItemDetailFragment Class.....	794
96.8 Modifying the ItemListFragment Class.....	795
96.9 Adding Manifest Permissions.....	795
96.10 Running the Application.....	796
96.11 Summary.....	796
97. Working with Material Design 3 Theming.....	797
97.1 Material Design 2 vs. Material Design 3.....	797
97.2 Understanding Material Design Theming.....	797
97.3 Material Design 3 Theming.....	797
97.4 Building a Custom Theme.....	799
97.5 Summary.....	800
98. A Material Design 3 Theming and Dynamic Color Tutorial.....	801
98.1 Creating the ThemeDemo Project.....	801
98.2 Designing the User Interface.....	801
98.3 Building a New Theme.....	803
98.4 Adding the Theme to the Project.....	804
98.5 Enabling Dynamic Color Support.....	805
98.6 Previewing Dynamic Colors.....	806
98.7 Summary.....	807
99. An Overview of Gradle in Android Studio.....	809
99.1 An Overview of Gradle.....	809
99.2 Gradle and Android Studio.....	809
99.2.1 Sensible Defaults.....	809
99.2.2 Dependencies.....	809
99.2.3 Build Variants.....	810
99.2.4 Manifest Entries.....	810
99.2.5 APK Signing.....	810
99.2.6 ProGuard Support.....	810
99.3 The Property and Settings Gradle Build File.....	810
99.4 The Top-level Gradle Build File.....	811
99.5 Module Level Gradle Build Files.....	812
99.6 Configuring Signing Settings in the Build File.....	814
99.7 Running Gradle Tasks from the Command Line.....	815
99.8 Summary.....	816
Index.....	817

1. Introduction

This book, fully updated for Android Studio Jellyfish (2023.3.1) and the new UI, teaches you how to develop Android-based applications using the Kotlin programming language.

This book begins with the basics and outlines how to set up an Android development and testing environment, followed by an introduction to programming in Kotlin, including data types, control flow, functions, lambdas, and object-oriented programming. Asynchronous programming using Kotlin coroutines and flow is also covered in detail.

Chapters also cover the Android Architecture Components, including view models, lifecycle management, Room database access, content providers, the Database Inspector, app navigation, live data, and data binding.

More advanced topics such as intents are also covered, as are touch screen handling, gesture recognition, and the recording and playback of audio. This book edition also covers printing, transitions, and foldable device support.

The concepts of material design are also covered in detail, including the use of floating action buttons, Snackbars, tabbed interfaces, card views, navigation drawers, and collapsing toolbars.

Other key features of Android Studio and Android are also covered in detail, including the Layout Editor, the `ConstraintLayout` and `ConstraintSet` classes, MotionLayout Editor, view binding, constraint chains, barriers, and direct reply notifications.

Chapters also cover advanced features of Android Studio, such as App Links, Gradle build configuration, in-app billing, and submitting apps to the Google Play Developer Console.

Assuming you already have some programming experience, are ready to download Android Studio and the Android SDK, have access to a Windows, Mac, or Linux system, and have ideas for some apps to develop, you are ready to get started.

1.1 Downloading the Code Samples

The source code and Android Studio project files for the examples contained in this book are available for download at:

<https://www.payloadbooks.com/product/jellyfishkotlin/>

The steps to load a project from the code samples into Android Studio are as follows:

1. From the Welcome to Android Studio dialog, click on the Open button option.
2. In the project selection dialog, navigate to and select the folder containing the project to be imported and click on OK.

1.2 Feedback

We want you to be satisfied with your purchase of this book. If you find any errors in the book, or have any comments, questions or concerns please contact us at info@payloadbooks.com.

1.3 Errata

While we make every effort to ensure the accuracy of the content of this book, it is inevitable that a book covering a subject area of this size and complexity may include some errors and oversights. Any known issues with the book will be outlined, together with solutions, at the following URL:

<https://www.payloadbooks.com/jellyfishkotlin>

If you find an error not listed in the errata, please let us know by emailing our technical support team at *info@payloadbooks.com*. They are there to help you and will work to resolve any problems you may encounter.

3. Creating an Example Android App in Android Studio

The preceding chapters of this book have explained how to configure an environment suitable for developing Android applications using the Android Studio IDE. Before moving on to slightly more advanced topics, now is a good time to validate that all required development packages are installed and functioning correctly. The best way to achieve this goal is to create an Android application and compile and run it. This chapter will cover creating an Android application project using Android Studio. Once the project has been created, a later chapter will explore using the Android emulator environment to perform a test run of the application.

3.1 About the Project

The project created in this chapter takes the form of a rudimentary currency conversion calculator (so simple, in fact, that it only converts from dollars to euros and does so using an estimated conversion rate). The project will also use one of the most basic Android Studio project templates. This simplicity allows us to introduce some key aspects of Android app development without overwhelming the beginner by introducing too many concepts, such as the recommended app architecture and Android architecture components, at once. When following the tutorial in this chapter, rest assured that the techniques and code used in this initial example project will be covered in much greater detail later.

3.2 Creating a New Android Project

The first step in the application development process is to create a new project within the Android Studio environment. Begin, therefore, by launching Android Studio so that the “Welcome to Android Studio” screen appears as illustrated in Figure 3-1:

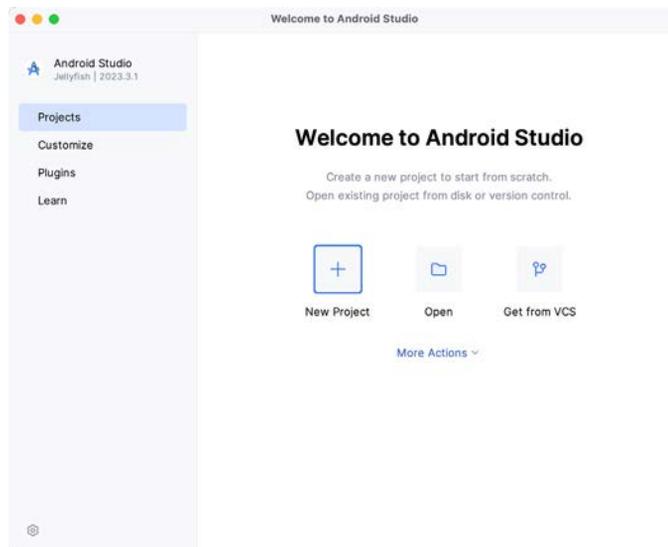


Figure 3-1

Creating an Example Android App in Android Studio

Once this window appears, Android Studio is ready for a new project to be created. To create the new project, click on the *New Project* option to display the first screen of the *New Project* wizard.

3.3 Creating an Activity

The next step is to define the type of initial activity to be created for the application. Options are available to create projects for Phone and Tablet, Wear OS, Television, or Automotive. A range of different activity types is available when developing Android applications, many of which will be covered extensively in later chapters. For this example, however, select the *Phone and Tablet* option from the Templates panel, followed by the option to create an *Empty Views Activity*. The Empty Views Activity option creates a template user interface consisting of a single *TextView* object.

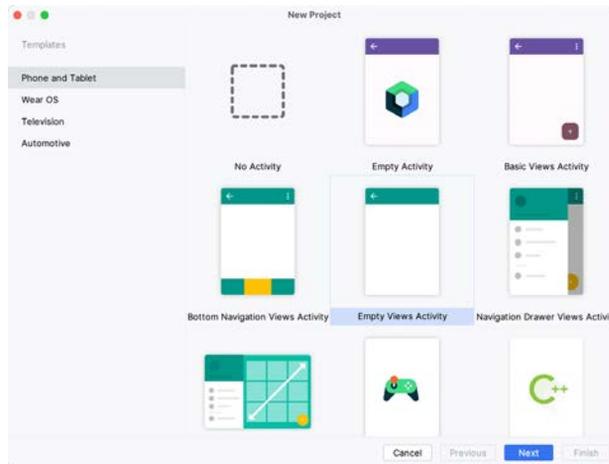


Figure 3-2

With the Empty Views Activity option selected, click *Next* to continue with the project configuration.

3.4 Defining the Project and SDK Settings

In the project configuration window (Figure 3-3), set the *Name* field to *AndroidSample*. The application name is the name by which the application will be referenced and identified within Android Studio and is also the name that would be used if the completed application were to go on sale in the Google Play store.

The *Package name* uniquely identifies the application within the Android application ecosystem. Although this can be set to any string that uniquely identifies your app, it is traditionally based on the reversed URL of your domain name followed by the application's name. For example, if your domain is *www.mycompany.com*, and the application has been named *AndroidSample*, then the package name might be specified as follows:

```
com.mycompany.androidsample
```

If you do not have a domain name, you can enter any other string into the Company Domain field, or you may use *example.com* for testing, though this will need to be changed before an application can be published:

```
com.example.androidsample
```

The *Save location* setting will default to a location in the folder named *AndroidStudioProjects* located in your home directory and may be changed by clicking on the folder icon to the right of the text field containing the current path setting.

Set the minimum SDK setting to API 26 (Oreo; Android 8.0). This minimum SDK will be used in most projects created in this book unless a necessary feature is only available in a more recent version. The objective here is to

build an app using the latest Android SDK while retaining compatibility with devices running older versions of Android (in this case, as far back as Android 8.0). The text beneath the Minimum SDK setting will outline the percentage of Android devices currently in use on which the app will run. Click on the *Help me choose* button (highlighted in Figure 3-3) to see a full breakdown of the various Android versions still in use:

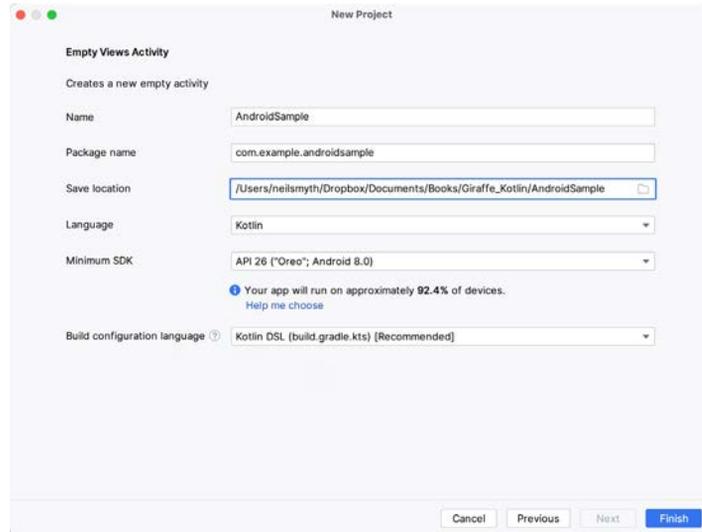


Figure 3-3

Finally, change the *Language* menu to *Kotlin* and select *Kotlin DSL (build.gradle.kts)* as the build configuration language before clicking *Finish* to create the project.

3.5 Modifying the Example Application

Once the project has been created, the main window will appear containing our *AndroidSample* project, as illustrated in Figure 3-4 below:

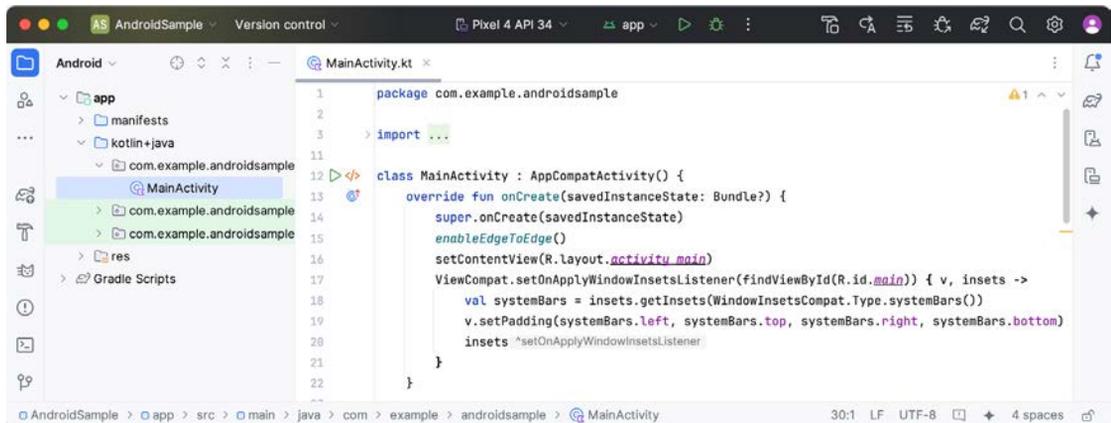


Figure 3-4

The newly created project and references to associated files are listed in the *Project* tool window on the left side of the main project window. The Project tool window has several modes in which information can be displayed. By default, this panel should be in *Android* mode. This setting is controlled by the menu at the top of the panel as highlighted in Figure 3-5. If the panel is not currently in Android mode, use the menu to switch mode:

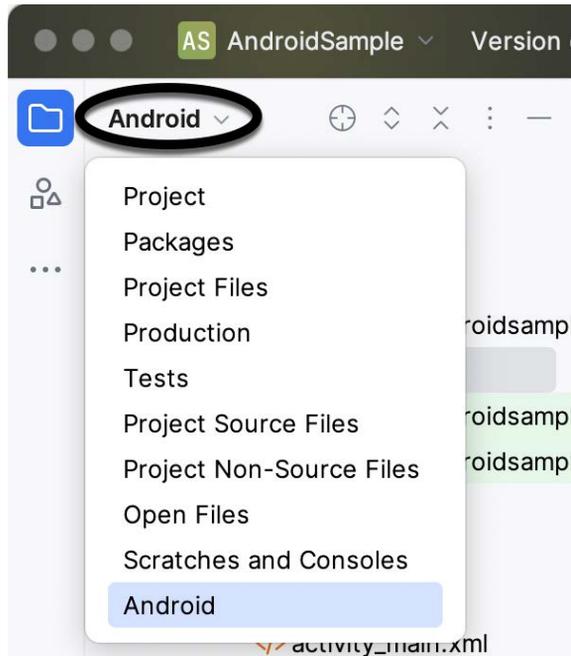


Figure 3-5

3.6 Modifying the User Interface

The user interface design for our activity is stored in a file named *activity_main.xml* which, in turn, is located under *app* -> *res* -> *layout* in the Project tool window file hierarchy. Once located in the Project tool window, double-click on the file to load it into the user interface Layout Editor tool, which will appear in the center panel of the Android Studio main window:

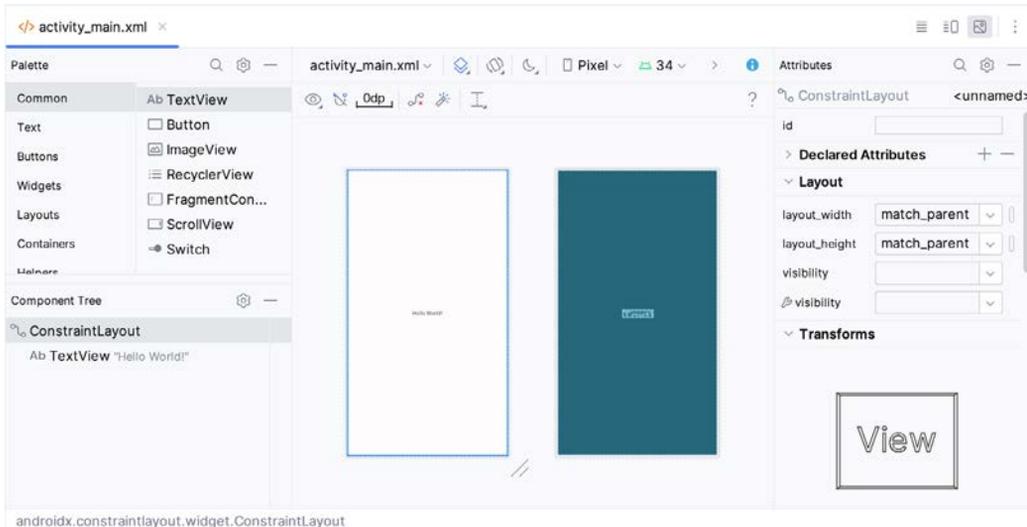


Figure 3-6

In the toolbar across the top of the Layout Editor window is a menu (currently set to *Pixel* in the above figure) which is reflected in the visual representation of the device within the Layout Editor panel. A range of other

device options are available by clicking on this menu.

Use the System UI Mode button () to turn Night mode on and off for the device screen layout. To change the orientation of the device representation between landscape and portrait, use the drop-down menu showing the  icon.

As we can see in the device screen, the content layout already includes a label that displays a “Hello World!” message. Running down the left-hand side of the panel is a palette containing different categories of user interface components that may be used to construct a user interface, such as buttons, labels, and text fields. However, it should be noted that not all user interface components are visible to the user. One such category consists of *layouts*. Android supports a variety of layouts that provide different levels of control over how visual user interface components are positioned and managed on the screen. Though it is difficult to tell from looking at the visual representation of the user interface, the current design has been created using a `ConstraintLayout`. This can be confirmed by reviewing the information in the *Component Tree* panel, which, by default, is located in the lower left-hand corner of the Layout Editor panel and is shown in Figure 3-7:

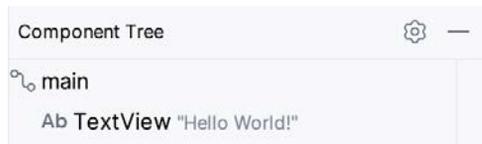


Figure 3-7

As we can see from the component tree hierarchy, the user interface layout consists of a `ConstraintLayout` parent called *main* and a `TextView` child object.

Before proceeding, check that the Layout Editor’s Autoconnect mode is enabled. This means that as components are added to the layout, the Layout Editor will automatically add constraints to ensure the components are correctly positioned for different screen sizes and device orientations (a topic that will be covered in much greater detail in future chapters). The Autoconnect button appears in the Layout Editor toolbar and is represented by a U-shaped icon. When disabled, the icon appears with a diagonal line through it (Figure 3-8). If necessary, re-enable Autoconnect mode by clicking on this button.



Figure 3-8

The next step in modifying the application is to add some additional components to the layout, the first of which will be a `Button` for the user to press to initiate the currency conversion.

The Palette panel consists of two columns, with the left-hand column containing a list of view component categories. The right-hand column lists the components contained within the currently selected category. In Figure 3-9, for example, the `Button` view is currently selected within the `Buttons` category:

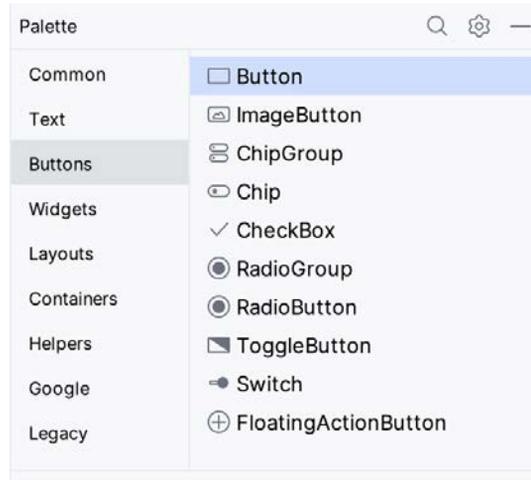


Figure 3-9

Click and drag the *Button* object from the Buttons list and drop it in the horizontal center of the user interface design so that it is positioned beneath the existing *TextView* widget:

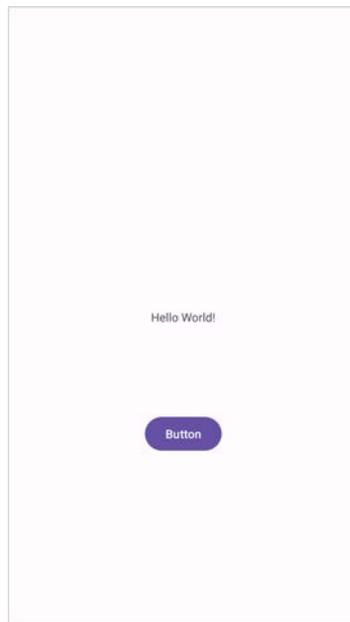


Figure 3-10

The next step is to change the text currently displayed by the *Button* component. The panel located to the right of the design area is the Attributes panel. This panel displays the attributes assigned to the currently selected component in the layout. Within this panel, locate the *text* property in the Common Attributes section and change the current value from “Button” to “Convert”, as shown in Figure 3-11:



Figure 3-11

The second text property with a wrench next to it allows a text property to be set, which only appears within the Layout Editor tool but is not shown at runtime. This is useful for testing how a visual component and the layout will behave with different settings without running the app repeatedly.

Just in case the Autoconnect system failed to set all of the layout connections, click on the Infer Constraints button (Figure 3-12) to add any missing constraints to the layout:

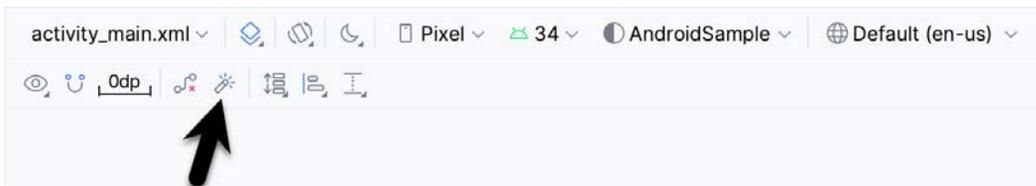


Figure 3-12

It is important to explain the warning button in the top right-hand corner of the Layout Editor tool, as indicated in Figure 3-13. This warning indicates potential problems with the layout. For details on any problems, click on the button:



Figure 3-13

When clicked, the Problems tool window (Figure 3-14) will appear, describing the nature of the problems:

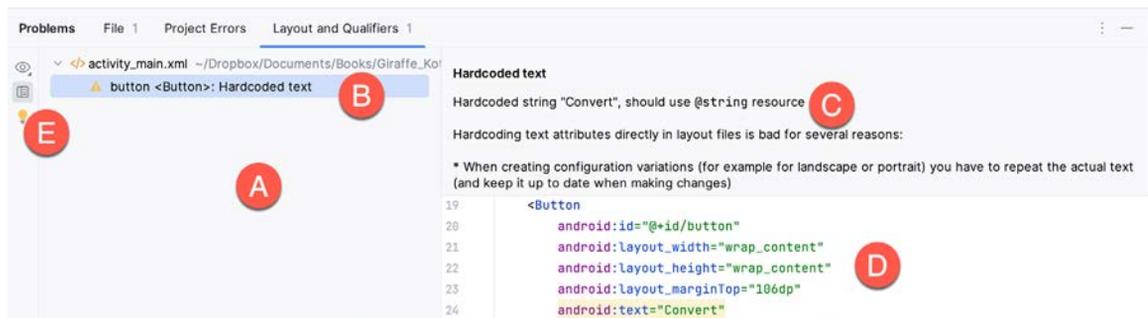


Figure 3-14

This tool window is divided into two panels. The left panel (marked A in the above figure) lists issues detected

Creating an Example Android App in Android Studio

within the layout file. In our example, only the following problem is listed:

```
button <Button>: Hardcoded text
```

When an item is selected from the list (B), the right-hand panel will update to provide additional detail on the problem (C). In this case, the explanation reads as follows:

```
Hardcoded string "Convert", should use @string resource
```

The tool window also includes a preview editor (D), allowing manual corrections to be made to the layout file.

This I18N message informs us that a potential issue exists concerning the future internationalization of the project (“I18N” comes from the fact that the word “internationalization” begins with an “I”, ends with an “N” and has 18 letters in between). The warning reminds us that attributes and values such as text strings should be stored as *resources* wherever possible when developing Android applications. Doing so enables changes to the appearance of the application to be made by modifying resource files instead of changing the application source code. This can be especially valuable when translating a user interface to a different spoken language. If all of the text in a user interface is contained in a single resource file, for example, that file can be given to a translator, who will then perform the translation work and return the translated file for inclusion in the application. This enables multiple languages to be targeted without the necessity for any source code changes to be made. In this instance, we are going to create a new resource named *convert_string* and assign to it the string “Convert”.

Begin by clicking on the Show Quick Fixes button (E) and selecting the *Extract string resource* option from the menu, as shown in Figure 3-15:

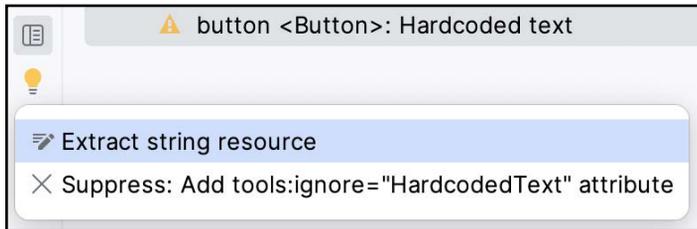


Figure 3-15

After selecting this option, the *Extract Resource* panel (Figure 3-16) will appear. Within this panel, change the resource name field to *convert_string* and leave the resource value set to *Convert* before clicking on the OK button:

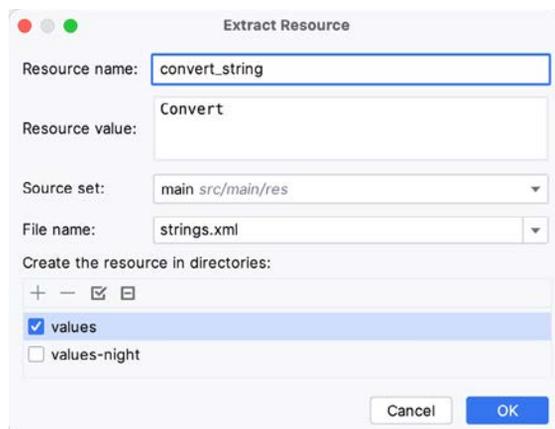


Figure 3-16

The next widget to be added is an EditText widget, into which the user will enter the dollar amount to be converted. From the Palette panel, select the Text category and click and drag a Number (Decimal) component onto the layout so that it is centered horizontally and positioned above the existing TextView widget. With the widget selected, use the Attributes tools window to set the *hint* property to “dollars”. Click on the warning icon and extract the string to a resource named *dollars_hint*.

The code written later in this chapter will need to access the dollar value entered by the user into the EditText field. It will do this by referencing the id assigned to the widget in the user interface layout. The default id assigned to the widget by Android Studio can be viewed and changed from within the Attributes tool window when the widget is selected in the layout, as shown in Figure 3-17:

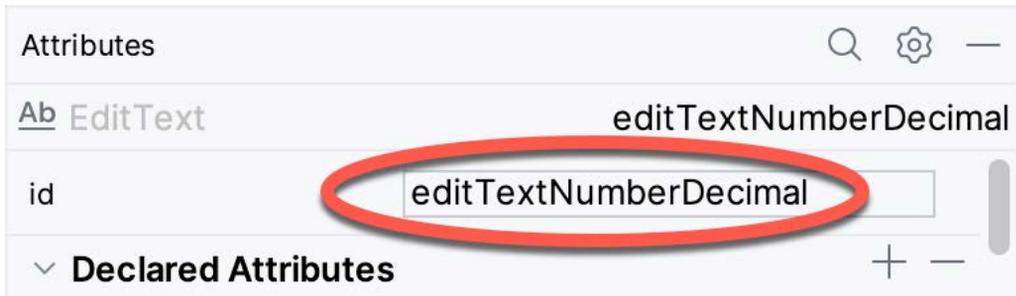


Figure 3-17

Change the id to *dollarText* and, in the Rename dialog, click on the *Refactor* button. This ensures that any references elsewhere within the project to the old id are automatically updated to use the new id:

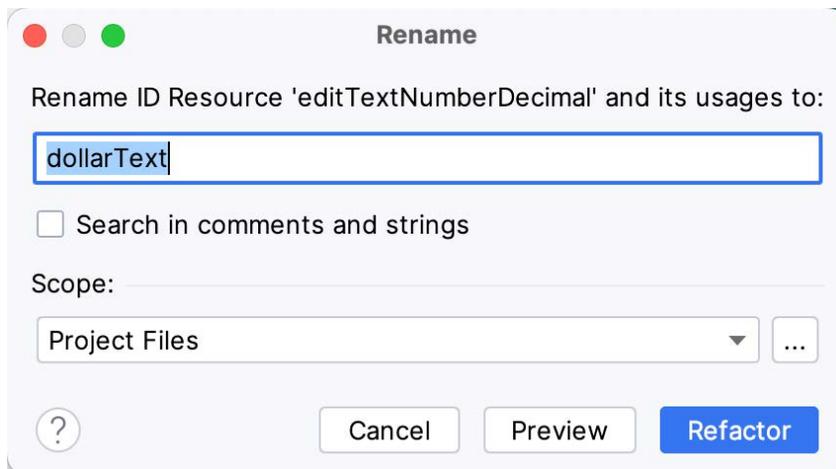


Figure 3-18

Repeat the steps to set the id of the TextView widget to *textView*, if necessary.

Add any missing layout constraints by clicking on the *Infer Constraints* button. At this point, the layout should resemble that shown in Figure 3-19:

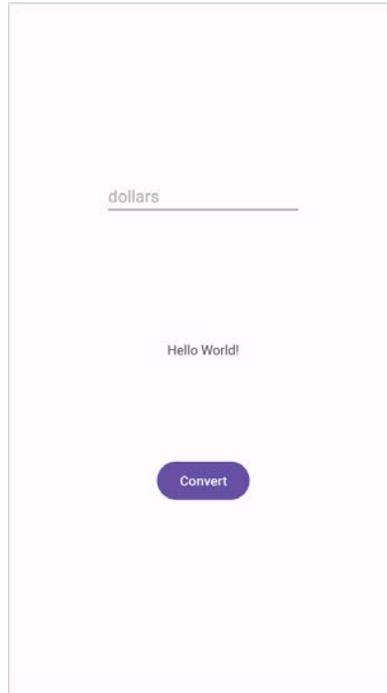


Figure 3-19

3.7 Reviewing the Layout and Resource Files

Before moving on to the next step, we will look at some internal aspects of user interface design and resource handling. In the previous section, we changed the user interface by modifying the `activity_main.xml` file using the Layout Editor tool. In fact, all that the Layout Editor was doing was providing a user-friendly way to edit the underlying XML content of the file. In practice, there is no reason why you cannot modify the XML directly to make user interface changes, and, in some instances, this may actually be quicker than using the Layout Editor tool. In the top right-hand corner of the Layout Editor panel are the View Modes buttons marked A through C in Figure 3-20 below:

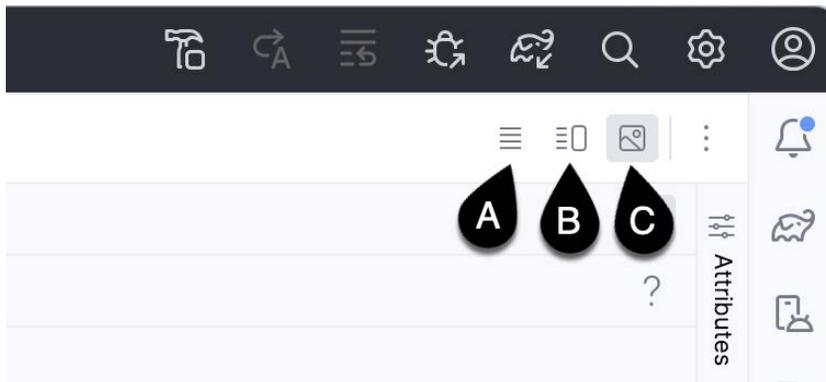


Figure 3-20

By default, the editor will be in *Design* mode (button C), whereby only the visual representation of the layout is displayed. In *Code* mode (A), the editor will display the XML for the layout, while in *Split* mode (B), both the layout and XML are displayed, as shown in Figure 3-21:

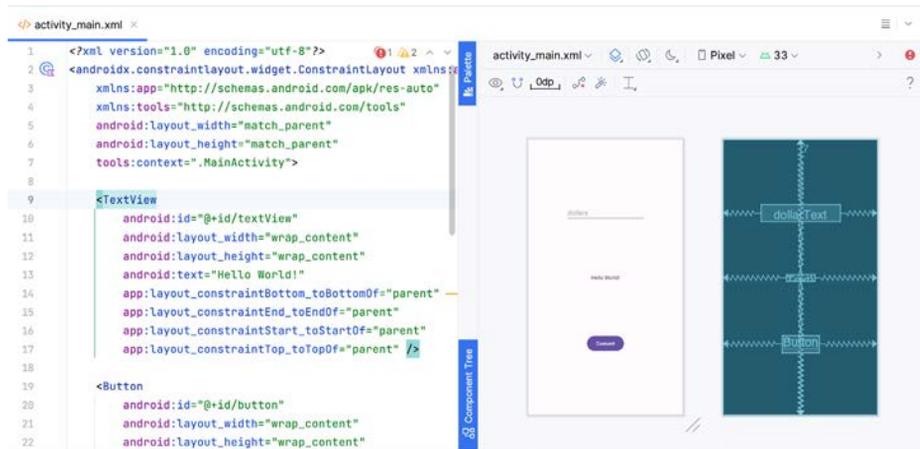


Figure 3-21

The button to the left of the View Modes button (marked B in Figure 3-20 above) is used to toggle between Code and Split modes quickly.

As can be seen from the structure of the XML file, the user interface consists of the `ConstraintLayout` component, which in turn, is the parent of the `TextView`, `Button`, and `EditText` objects. We can also see, for example, that the `text` property of the `Button` is set to our `convert_string` resource. Although complexity and content vary, all user interface layouts are structured in this hierarchical, XML-based way.

As changes are made to the XML layout, these will be reflected in the layout canvas. The layout may also be modified visually from within the layout canvas panel, with the changes appearing in the XML listing. To see this in action, switch to Split mode and modify the XML layout to change the background color of the `ConstraintLayout` to a shade of red as follows:

```
<?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:id="@+id/main"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity"
    android:background="#ff2438" >
    .
    .
</androidx.constraintlayout.widget.ConstraintLayout>
```

Note that the layout color changes in real-time to match the new setting in the XML file. Note also that a small red square appears in the XML editor's left margin (also called the *gutter*) next to the line containing the color setting. This is a visual cue to the fact that the color red has been set on a property. Clicking on the red square will display a color chooser allowing a different color to be selected:

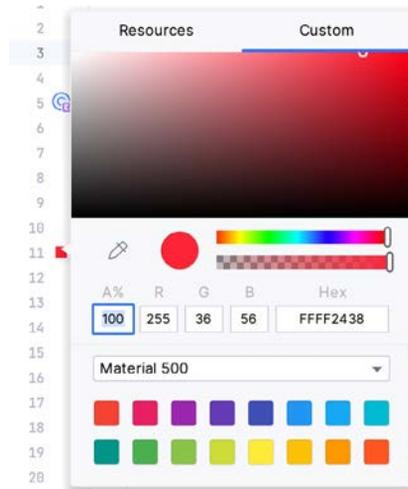


Figure 3-22

Before proceeding, delete the background property from the layout file so that the background returns to the default setting.

Finally, use the Project panel to locate the *app -> res -> values -> strings.xml* file and double-click on it to load it into the editor. Currently, the XML should read as follows:

```
<resources>
    <string name="app_name">AndroidSample</string>
    <string name="convert_string">Convert</string>
    <string name="dollars_hint">dollars</string>
</resources>
```

To demonstrate resources in action, change the string value currently assigned to the *convert_string* resource to “Convert to Euros” and then return to the Layout Editor tool by selecting the tab for the layout file in the editor panel. Note that the layout has picked up the new resource value for the string.

There is also a quick way to access the value of a resource referenced in an XML file. With the Layout Editor tool in Split or Code mode, click on the “@string/convert_string” property setting so that it highlights, and then press Ctrl-B on the keyboard (Cmd-B on macOS). Android Studio will subsequently open the *strings.xml* file and take you to the line in that file where this resource is declared. Use this opportunity to revert the string resource to the original “Convert” text and to add the following additional entry for a string resource that will be referenced later in the app code:

```
<resources>
    <string name="app_name">AndroidSample</string>
    <string name="convert_string">Convert</string>
    <string name="dollars_hint">dollars</string>
    <string name="no_value_string">No Value</string>
</resources>
```

Resource strings may also be edited using the Android Studio Translations Editor by clicking on the *Open editor* link in the top right-hand corner of the editor window. This will display the Translation Editor in the main panel of the Android Studio window:

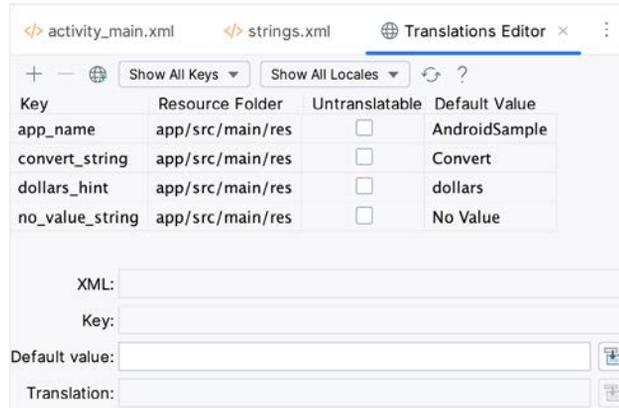


Figure 3-23

This editor allows the strings assigned to resource keys to be edited and for translations for multiple languages to be managed.

3.8 Adding Interaction

The final step in this example project is to make the app interactive so that when the user enters a dollar value into the EditText field and clicks the convert button, the converted euro value appears on the TextView. This involves the implementation of some event handling on the Button widget. Specifically, the Button needs to be configured so that a method in the app code is called when an *onClick* event is triggered. Event handling can be implemented in several ways and is covered in a later chapter entitled “*An Overview and Example of Android Event Handling*”. Return the layout editor to Design mode, select the Button widget in the layout editor, refer to the Attributes tool window, and specify a method named *convertCurrency* as shown below:



Figure 3-24

Next, double-click on the *MainActivity.kt* file in the Project tool window (*app* -> *kotlin+java* -> *<package name>* -> *MainActivity*) to load it into the code editor and add the code for the *convertCurrency* method to the class file so that it reads as follows, noting that it is also necessary to import some additional Android packages:

```
package com.example.androidsample

import android.os.Bundle
import androidx.activity.enableEdgeToEdge
import androidx.appcompat.app.AppCompatActivity
import androidx.core.view.ViewCompat
import androidx.core.view.WindowInsetsCompat
import android.view.View
import android.widget.EditText
import android.widget.TextView
```

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

fun convertCurrency(view: View) {
    val dollarText: EditText = findViewById(R.id.dollarText)
    val textView: TextView = findViewById(R.id.textView)

    if (dollarText.text.isNotEmpty()) {
        val dollarValue = dollarText.text.toString().toFloat()
        val euroValue = dollarValue * 0.85f
        textView.text = euroValue.toString()
    } else {
        textView.text = getString(R.string.no_value_string)
    }
}
```

The method begins by obtaining references to the EditText and TextView objects by making a call to a method named `findViewById`, passing through the id assigned within the layout file. A check is then made to ensure that the user has entered a dollar value, and if so, that value is extracted, converted from a String to a floating point value, and converted to euros. Finally, the result is displayed on the TextView widget.

If any of this is unclear, rest assured that these concepts will be covered in greater detail in later chapters. In particular, the topic of accessing widgets from within code using `findViewById` and an introduction to an alternative technique referred to as *view binding* will be covered in the chapter entitled “*An Overview of Android View Binding*”.

3.9 Summary

While not excessively complex, several steps are involved in setting up an Android development environment. Having performed those steps, it is worth working through an example to ensure the environment is correctly installed and configured. In this chapter, we have created an example application and then used the Android Studio Layout Editor tool to modify the user interface layout. In doing so, we explored the importance of using resources wherever possible, particularly string values, and briefly touched on layouts. Next, we looked at the underlying XML used to store Android application user interface designs.

Finally, an `onClick` event was added to a Button connected to a method implemented to extract the user input from the EditText component, convert it from dollars to euros and then display the result on the TextView.

With the app ready for testing, the steps necessary to set up an emulator for testing purposes will be covered in detail in the next chapter.

14. Kotlin Control Flow

Regardless of the programming language used, application development is largely an exercise in applying logic, and much of the art of programming involves writing code that makes decisions based on one or more criteria. Such decisions define which code gets executed, how many times it is executed and, conversely, which code gets by-passed when the program is executing. This is often referred to as *control flow* since it controls the *flow* of program execution. Control flow typically falls into the categories of *looping control* (how often code is executed) and *conditional control flow* (whether or not code is executed). This chapter is intended to provide an introductory overview of both types of control flow in Kotlin.

14.1 Looping Control flow

This chapter will begin by looking at control flow in the form of loops. Loops are essentially sequences of Kotlin statements which are to be executed repeatedly until a specified condition is met. The first looping statement we will explore is the *for* loop.

14.1.1 The Kotlin *for-in* Statement

The for-in loop is used to iterate over a sequence of items contained in a collection or number range.

The syntax of the for-in loop is as follows:

```
for variable name in collection or range {  
    // code to be executed  
}
```

In this syntax, *variable name* is the name to be used for a variable that will contain the current item from the collection or range through which the loop is iterating. The code in the body of the loop will typically use this name as a reference to the current item in the loop cycle. The *collection* or *range* references the item through which the loop is iterating. This could, for example, be an array of string values, a range operator or even a string of characters.

Consider, for example, the following for-in loop construct:

```
for (index in 1..5) {  
    println("Value of index is $index")  
}
```

The loop begins by stating that the current item is to be assigned to a constant named *index*. The statement then declares a closed range operator to indicate that the for loop is to iterate through a range of numbers, starting at 1 and ending at 5. The body of the loop prints out a message to the console indicating the current value assigned to the *index* constant, resulting in the following output:

```
Value of index is 1  
Value of index is 2  
Value of index is 3  
Value of index is 4  
Value of index is 5
```

The for-in loop is of particular benefit when working with collections such as arrays. In fact, the for-in loop can be used to iterate through any object that contains more than one item. The following loop, for example, outputs

Kotlin Control Flow

each of the characters in the specified string:

```
for (index in "Hello") {
    println("Value of index is $index")
}
```

The operation of a for-in loop may be configured using the *downTo* and *until* functions. The *downTo* function causes the for loop to work backwards through the specified collection until the specified number is reached. The following for loop counts backwards from 100 until the number 90 is reached:

```
for (index in 100 downTo 90) {
    print("$index.. ")
}
```

When executed, the above loop will generate the following output:

```
100.. 99.. 98.. 97.. 96.. 95.. 94.. 93.. 92.. 91.. 90..
```

The *until* function operates in much the same way with the exception that counting starts from the bottom of the collection range and works up until (but not including) the specified end point (a concept referred to as a half closed range):

```
for (index in 1 until 10) {
    print("$index.. ")
}
```

The output from the above code will range from the start value of 1 through to 9:

```
1.. 2.. 3.. 4.. 5.. 6.. 7.. 8.. 9..
```

The increment used on each iteration through the loop may also be defined using the *step* function as follows:

```
for (index in 0 until 100 step 10) {
    print("$index.. ")
}
```

The above code will result in the following console output:

```
0.. 10.. 20.. 30.. 40.. 50.. 60.. 70.. 80.. 90..
```

14.1.2 The *while* Loop

The Kotlin *for* loop described previously works well when it is known in advance how many times a particular task needs to be repeated in a program. There will, however, be instances where code needs to be repeated until a certain condition is met, with no way of knowing in advance how many repetitions are going to be needed to meet that criteria. To address this need, Kotlin includes the *while* loop.

Essentially, the while loop repeats a set of tasks while a specified condition is met. The *while* loop syntax is defined as follows:

```
while condition {
    // Kotlin statements go here
}
```

In the above syntax, *condition* is an expression that will return either *true* or *false* and the *// Kotlin statements go here* comment represents the code to be executed while the condition expression is true. For example:

```
var myCount = 0
```

```
while (myCount < 100) {
```

```

    myCount++
    println(myCount)
}

```

In the above example, the *while* expression will evaluate whether the *myCount* variable is less than 100. If it is already greater than 100, the code in the braces is skipped and the loop exits without performing any tasks.

If, on the other hand, *myCount* is not greater than 100 the code in the braces is executed and the loop returns to the *while* statement and repeats the evaluation of *myCount*. This process repeats until the value of *myCount* is greater than 100, at which point the loop exits.

14.1.3 The *do ... while* loop

It is often helpful to think of the *do ... while* loop as an inverted *while* loop. The *while* loop evaluates an expression before executing the code contained in the body of the loop. If the expression evaluates to *false* on the first check then the code is not executed. The *do ... while* loop, on the other hand, is provided for situations where you know that the code contained in the body of the loop will *always* need to be executed at least once. For example, you may want to keep stepping through the items in an array until a specific item is found. You know that you have to at least check the first item in the array to have any hope of finding the entry you need. The syntax for the *do ... while* loop is as follows:

```

do {
    // Kotlin statements here
} while conditional expression

```

In the *do ... while* example below the loop will continue until the value of a variable named *i* equals 0:

```

var i = 10

do {
    i--
    println(i)
} while (i > 0)

```

14.1.4 Breaking from Loops

Having created a loop, it is possible that under certain conditions you might want to break out of the loop before the completion criteria have been met (particularly if you have created an infinite loop). One such example might involve continually checking for activity on a network socket. Once activity has been detected it will most likely be necessary to break out of the monitoring loop and perform some other task.

For the purpose of breaking out of a loop, Kotlin provides the *break* statement which breaks out of the current loop and resumes execution at the code directly after the loop. For example:

```

var j = 10

for (i in 0..100)
{
    j += j

    if (j > 100) {
        break
    }
}

```

Kotlin Control Flow

```
        println("j = $j")
    }
```

In the above example the loop will continue to execute until the value of *j* exceeds 100 at which point the loop will exit and execution will continue with the next line of code after the loop.

14.1.5 The *continue* Statement

The *continue* statement causes all remaining code statements in a loop to be skipped, and execution to be returned to the top of the loop. In the following example, the *println* function is only called when the value of variable *i* is an even number:

```
var i = 1

while (i < 20)
{
    i += 1

    if (i % 2 != 0) {
        continue
    }

    println("i = $i")
}
```

The *continue* statement in the above example will cause the *println* call to be skipped unless the value of *i* can be divided by 2 with no remainder. If the *continue* statement is triggered, execution will skip to the top of the while loop and the statements in the body of the loop will be repeated (until the value of *i* exceeds 19).

14.1.6 Break and Continue Labels

Kotlin expressions may be assigned a label by preceding the expression with a label name followed by the @ sign. This label may then be referenced when using *break* and *continue* statements to designate where execution is to resume. This is particularly useful when breaking out of nested loops. The following code contains a for loop nested within another for loop. The inner loop contains a *break* statement which is executed when the value of *j* reaches 10:

```
for (i in 1..100) {

    println("Outer loop i = $i")

    for (j in 1..100) {
        println("Inner loop j = $j")
        if (j == 10) break
    }
}
```

As currently implemented, the *break* statement will exit the inner for loop but execution will resume at the top of the outer for loop. Suppose, however, that the *break* statement is required to also exit the outer loop. This can be achieved by assigning a label to the outer loop and referencing that label with the *break* statement as follows:

```
outerloop@ for (i in 1..100) {
```

```
println("Outer loop i = $i")

for (j in 1..100) {

    println("Inner loop j = $j")

    if (j == 10) break@outerloop
}
}
```

Now when the value assigned to variable `j` reaches 10 the `break` statement will break out of both loops and resume execution at the line of code immediately following the outer loop.

14.2 Conditional Control Flow

In the previous chapter we looked at how to use logical expressions in Kotlin to determine whether something is *true* or *false*. Since programming is largely an exercise in applying logic, much of the art of programming involves writing code that makes decisions based on one or more criteria. Such decisions define which code gets executed and, conversely, which code gets by-passed when the program is executing.

14.2.1 Using the *if* Expressions

The *if* expression is perhaps the most basic of control flow options available to the Kotlin programmer. Programmers who are familiar with C, Swift, C++ or Java will immediately be comfortable using Kotlin *if* statements, although there are some subtle differences.

The basic syntax of the Kotlin *if* expression is as follows:

```
if (boolean expression) {
    // Kotlin code to be performed when expression evaluates to true
}
```

Unlike some other programming languages, it is important to note that the braces are optional in Kotlin if only one line of code is associated with the *if* expression. In fact, in this scenario, the statement is often placed on the same line as the *if* expression.

Essentially if the *Boolean expression* evaluates to *true* then the code in the body of the statement is executed. If, on the other hand, the expression evaluates to *false* the code in the body of the statement is skipped.

For example, if a decision needs to be made depending on whether one value is greater than another, we would write code similar to the following:

```
val x = 10

if (x > 9) println("x is greater than 9!")
```

Clearly, `x` is indeed greater than 9 causing the message to appear in the console panel.

At this point it is important to notice that we have been referring to the *if* expression instead of the *if* statement. The reason for this is that unlike the *if* statement in other programming languages, the Kotlin *if* returns a result. This allows *if* constructs to be used within expressions. As an example, a typical *if* expression to identify the largest of two numbers and assign the result to a variable might read as follows:

```
if (x > y)
    largest = x
else
```

Kotlin Control Flow

```
largest = y
```

The same result can be achieved using the *if* statement within an expression using the following syntax:

```
variable = if (condition) return_val_1 else return_val_2
```

The original example can, therefore be re-written as follows:

```
val largest = if (x > y) x else y
```

The technique is not limited to returning the values contained within the condition. The following example is also a valid use of *if* in an expression, in this case assigning a string value to the variable:

```
val largest = if (x > y) "x is greatest" else "y is greatest"  
println(largest)
```

For those familiar with programming languages such as Java, this feature allows code constructs similar to ternary statements to be implemented in Kotlin.

14.2.2 Using *if... else ...* Expressions

The next variation of the *if* expression allows us to also specify some code to perform if the expression in the *if* expression evaluates to *false*. The syntax for this construct is as follows:

```
if (boolean expression) {  
    // Code to be executed if expression is true  
} else {  
    // Code to be executed if expression is false  
}
```

The braces are, once again, optional if only one line of code is to be executed.

Using the above syntax, we can now extend our previous example to display a different message if the comparison expression evaluates to be *false*:

```
val x = 10  
  
if (x > 9) println("x is greater than 9!")  
    else println("x is less than 9!")
```

In this case, the second `println` statement will execute if the value of `x` was less than 9.

14.2.3 Using *if... else if... else if... else if...* Expressions

So far we have looked at *if* statements which make decisions based on the result of a single logical expression. Sometimes it becomes necessary to make decisions based on a number of different criteria. For this purpose, we can use the *if... else if... else if... else if...* construct, an example of which is as follows:

```
var x = 9  
  
if (x == 10) println("x is 10")  
    else if (x == 9) println("x is 9")  
        else if (x == 8) println("x is 8")  
            else println("x is less than 8")  
}
```

14.2.4 Using the *when* Statement

The Kotlin *when* statement provides a cleaner alternative to the *if... else if... else if...* construct and uses the following syntax:

```
when (value) {  
    match1 -> // code to be executed on match  
    match2 -> // code to be executed on match  
    .  
    .  
    else -> // default code to executed if no match  
}
```

Using this syntax, the previous *if... else if...* construct can be rewritten to use the *when* statement:

```
when (x) {  
    10 -> println ("x is 10")  
    9 -> println("x is 9")  
    8 -> println("x is 8")  
    else -> println("x is less than 8")  
}
```

The *when* statement is similar to the *switch* statement found in many other programming languages.

14.3 Summary

The term *control flow* is used to describe the logic that dictates the execution path that is taken through the source code of an application as it runs. This chapter has looked at the two types of control flow provided by Kotlin (looping and conditional) and explored the various Kotlin constructs that are available to implement both forms of control flow logic.

62. An Introduction to Kotlin Coroutines

When an Android application is first started, the runtime system creates a single thread in which all components will run by default. This thread is generally referred to as the *main thread*. The primary role of the main thread is to handle the user interface in terms of event handling and interaction with views in the user interface. Any additional components started within the application will, by default, also run on the main thread.

Any code within an application that performs a time-consuming task using the main thread will cause the entire application to appear to lock up until the task is completed. This typically results in the operating system displaying an “Application is not responding” warning to the user. This is far from the desired behavior for any application. Fortunately, Kotlin provides a lightweight alternative in the form of Coroutines. This chapter will introduce Coroutines, including terminology such as dispatchers, coroutine scope, suspend functions, coroutine builders, and structured concurrency. The chapter will also explore channel-based communication between coroutines.

62.1 What are Coroutines?

Coroutines are blocks of code that execute asynchronously without blocking the thread from which they are launched. Coroutines can be implemented without worrying about building complex `AsyncTask` implementations or directly managing multiple threads. Because of the way they are implemented, coroutines are much more efficient and less resource intensive than using traditional multi-threading options. Coroutines also make for code that is much easier to write, understand and maintain since it allows code to be written sequentially without having to write callbacks to handle thread-related events and results.

Although a relatively recent addition to Kotlin, there is nothing new or innovative about coroutines. Coroutines, in one form or another, have existed in programming languages since the 1960s and are based on a model known as Communicating Sequential Processes (CSP). Though it does so efficiently, Kotlin still uses multi-threading behind the scenes.

62.2 Threads vs. Coroutines

A problem with threads is that they are a finite resource and expensive in terms of CPU capabilities and system overhead. In the background, much work is involved in creating, scheduling, and destroying a thread. Although modern CPUs can run large numbers of threads, the actual number of threads that can be run in parallel at any one time is limited by the number of CPU cores (though newer CPUs have 8 cores, most Android devices contain CPUs with 4 cores). When more threads are required than there are CPU cores, the system has to perform thread scheduling to decide how the execution of these threads is to be shared between the available cores.

To avoid these overheads, instead of starting a new thread for each coroutine and destroying it when the coroutine exits, Kotlin maintains a pool of active threads and manages how coroutines are assigned to those threads. When an active coroutine is suspended, the Kotlin runtime saves it, and another coroutine resumes to take its place. When the coroutine is resumed, it is restored to an existing unoccupied thread within the pool to continue executing until it either completes or is suspended. Using this approach, a limited number of threads are used efficiently to execute asynchronous tasks with the potential to perform large numbers of concurrent

tasks without the inherent performance degeneration that would occur using standard multi-threading.

62.3 Coroutine Scope

All coroutines must run within a specific scope, allowing them to be managed as groups instead of as individual ones. This is particularly important when canceling and cleaning up coroutines, for example, when a Fragment or Activity is destroyed, and ensuring that coroutines do not “leak” (in other words, continue running in the background when the app no longer needs them). By assigning coroutines to a scope, they can, for example, all be canceled in bulk when they are no longer needed.

Kotlin and Android provide built-in scopes and the option to create custom scopes using the `CoroutineScope` class. The built-in scopes can be summarized as follows:

- **GlobalScope** – `GlobalScope` is used to launch top-level coroutines tied to the entire application lifecycle. Since this has the potential for coroutines in this scope to continue running when not needed (for example, when an Activity exits), use of this scope is not recommended for Android applications. Coroutines running in `GlobalScope` are considered to be using *unstructured concurrency*.
- **ViewModelScope** – Provided specifically for `ViewModel` instances when using the Jetpack architecture `ViewModel` component. Coroutines launched in this scope from within a `ViewModel` instance are automatically canceled by the Kotlin runtime system when the corresponding `ViewModel` instance is destroyed.
- **LifecycleScope** – Every lifecycle owner has associated with it a `LifecycleScope`. This scope is canceled when the corresponding lifecycle owner is destroyed, making it particularly useful for launching coroutines from within activities and fragments.

For all other requirements, a custom scope will likely be used. The following code, for example, creates a custom scope named *myCoroutineScope*:

```
private val myCoroutineScope = CoroutineScope(Dispatchers.Main)
```

The `myCoroutineScope` declares the dispatcher that will be used to run coroutines (though this can be overridden) and must be referenced each time a coroutine is started if it is to be included within the scope. All of the running coroutines in a scope can be canceled via a call to the `cancel()` method of the scope instance:

```
myCoroutineScope.cancel()
```

62.4 Suspend Functions

A suspend function is a special type of Kotlin function that contains the code of a coroutine. It is declared using the Kotlin `suspend` keyword, which indicates to Kotlin that the function can be paused and resumed later, allowing long-running computations to execute without blocking the main thread.

The following is an example suspend function:

```
suspend fun mySlowTask() {  
    // Perform long-running tasks here  
}
```

62.5 Coroutine Dispatchers

Kotlin maintains threads for different types of asynchronous activity, and when launching a coroutine, it will be necessary to select the appropriate dispatcher from the following options:

- **Dispatchers.Main** – Runs the coroutine on the main thread and is suitable for coroutines that need to make changes to the UI and as a general-purpose option for performing lightweight tasks.
- **Dispatchers.IO** – Recommended for coroutines that perform network, disk, or database operations.

- **Dispatchers.Default** – Intended for CPU-intensive tasks such as sorting data or performing complex calculations.

The dispatcher is responsible for assigning coroutines to appropriate threads and suspending and resuming the coroutine during its lifecycle. In addition to the predefined dispatchers, it is also possible to create dispatchers for your own custom thread pools.

62.6 Coroutine Builders

The coroutine builders bring together all of the components covered so far and launch the coroutines so that they start executing. For this purpose, Kotlin provides the following six builders:

- **launch** – Starts a coroutine without blocking the current thread and does not return a result to the caller. Use this builder when calling a suspend function from within a traditional function and when the results of the coroutine do not need to be handled (sometimes referred to as “fire and forget” coroutines).
- **async** – Starts a coroutine and allows the caller to wait for a result using the `await()` function without blocking the current thread. Use `async` when you have multiple coroutines that need to run in parallel. The `async` builder can only be used from within another suspend function.
- **withContext** – Allows a coroutine to be launched in a different context from that used by the parent coroutine. Using this builder, a coroutine running using the `Main` context could launch a child coroutine in the `Default` context. The `withContext` builder also provides a useful alternative to `async` when returning results from a coroutine.
- **coroutineScope** – The `coroutineScope` builder is ideal for situations where a suspend function launches multiple coroutines that will run in parallel and where some action must occur only when all the coroutines reach completion. If those coroutines are launched using the `coroutineScope` builder, the calling function will not return until all child coroutines have completed. When using `coroutineScope`, a failure in any coroutine will cancel all other coroutines.
- **supervisorScope** – Similar to the `coroutineScope` outlined above, except that a failure in one child does not result in the cancellation of the other coroutines.
- **runBlocking** – Starts a coroutine and blocks the current thread until the coroutine reaches completion. This is typically the exact opposite of what is wanted from coroutines but is useful for testing code and when integrating legacy code and libraries. Otherwise to be avoided.

62.7 Jobs

Each call to a coroutine builder, such as `launch` or `async`, returns a `Job` instance which can, in turn, be used to track and manage the lifecycle of the corresponding coroutine. Subsequent builder calls from within the coroutine create new `Job` instances, which will become children of the immediate parent `Job`, forming a parent-child relationship tree where canceling a parent `Job` will recursively cancel all its children. Canceling a child does not, however, cancel the parent, though an uncaught exception within a child created using the `launch` builder may result in the cancellation of the parent (this is not the case for children created using the `async` builder, which encapsulates the exception in the result returned to the parent).

The status of a coroutine can be identified by accessing the `isActive`, `isCompleted`, and `isCancelled` properties of the associated `Job` object. In addition to these properties, several methods are also available on a `Job` instance. For example, a `Job` and all of its children may be canceled by calling the `cancel()` method of the `Job` object, while a call to the `cancelChildren()` method will cancel all child coroutines.

The `join()` method can be called to suspend the coroutine associated with the job until all of its child jobs have completed. To perform this task and cancel the `Job` once all child jobs have completed, call the `cancelAndJoin()`

method.

This hierarchical Job structure, together with coroutine scopes, form the foundation of structured concurrency, which aims to ensure that coroutines do not run longer than required without manually keeping references to each coroutine.

62.8 Coroutines – Suspending and Resuming

It helps to see some coroutine examples in action to understand coroutine suspension better. To start with, let's assume a simple Android app containing a button that, when clicked, calls a function named *startTask()*. This function calls a suspend function named *performSlowTask()* using the Main coroutine dispatcher. The code for this might read as follows:

```
private val myCoroutineScope = CoroutineScope(Dispatchers.Main)

fun startTask(view: View) {
    myCoroutineScope.launch(Dispatchers.Main) {
        performSlowTask()
    }
}
```

In the above code, a custom scope is declared and referenced in the call to the launch builder, which, in turn, calls the *performSlowTask()* suspend function. Since *startTask()* is not a suspend function, the coroutine must be started using the launch builder instead of the async builder.

Next, we can declare the *performSlowTask()* suspend function as follows:

```
suspend fun performSlowTask() {
    Log.i(TAG, "performSlowTask before")
    delay(5_000) // simulates long-running task
    Log.i(TAG, "performSlowTask after")
}
```

As implemented, all the function does is output diagnostic messages before and after performing a 5-second delay, simulating a long-running task. While the 5-second delay is in effect, the user interface will continue to be responsive because the main thread is not being blocked. To understand why it helps to explore what is happening behind the scenes.

First, the *startTask()* function is executed and launches the *performSlowTask()* suspend function as a coroutine. This function then calls the Kotlin *delay()* function passing through a time value. The built-in Kotlin *delay()* function is implemented as a suspend function, so it is also launched as a coroutine by the Kotlin runtime environment. The code execution has now reached what is referred to as a suspend point which will cause the *performSlowTask()* coroutine to be suspended while the delay coroutine is running. This frees up the thread on which *performSlowTask()* was running and returns control to the main thread so that the UI is unaffected.

Once the *delay()* function reaches completion, the suspended coroutine will be resumed and restored to a thread from the pool where it can display the Log message and return to the *startTask()* function.

When working with coroutines in Android Studio suspend points within the code editor are marked as shown in the figure below:



Figure 62-1

62.9 Returning Results from a Coroutine

The above example ran a suspend function as a coroutine but did not demonstrate how to return results. However, suppose the *performSlowTask()* function is required to return a string value to be displayed to the user via a *TextView* object.

To do this, we must rewrite the suspend function to return a *Deferred* object. A *Deferred* object is a commitment to provide a value at some point in the future. By calling the *await()* function on the *Deferred* object, the Kotlin runtime will deliver the value when the coroutine returns it. The code in our *startTask()* function might, therefore, be rewritten as follows:

```
fun startTask(view: View) {
    coroutineScope.launch(Dispatchers.Main) {
        statusText.text = performSlowTask().await()
    }
}
```

The problem now is that we are having to use the launch builder to start the coroutine since *startTask()* is not a suspend function. As outlined earlier in this chapter, it is only possible to return results when using the *async* builder. To get around this, we have to adapt the suspend function to use the *async* builder to start another coroutine that returns a *Deferred* result:

```
suspend fun performSlowTask(): Deferred<String> =
    coroutineScope.async(Dispatchers.Default) {
        Log.i(TAG, "performSlowTask before")
        delay(5_000)
        Log.i(TAG, "performSlowTask after")
        return@async "Finished"
    }
```

When the app runs, the “Finished” result string will be displayed on the *TextView* object when the *performSlowTask()* coroutine completes. Once again, the wait for the result will occur in the background without blocking the main thread.

62.10 Using withContext

As we have seen, coroutines are launched within a specified scope and using a specific dispatcher. By default, any child coroutines will inherit the same dispatcher as that used by the parent. Consider the following code

An Introduction to Kotlin Coroutines

designed to call multiple functions from within a suspend function:

```
fun startTask(view: View) {

    coroutineScope.launch(Dispatchers.Main) {
        performTasks()
    }
}

suspend fun performTasks() {
    performTask1()
    performTask2()
    performTask3()
}

suspend fun performTask1() {
    Log.i(TAG, "Task 1 ${Thread.currentThread().name}")
}

suspend fun performTask2() {
    Log.i(TAG, "Task 2 ${Thread.currentThread().name}")
}

suspend fun performTask3() {
    Log.i(TAG, "Task 3 ${Thread.currentThread().name}")
}
```

Since the *performTasks()* function was launched using the Main dispatcher, all three functions will default to the main thread. To prove this, the functions have been written to output the name of the thread in which they are running. On execution, the Logcat panel will contain the following output:

```
Task 1 main
Task 2 main
Task 3 main
```

However, imagine that the *performTask2()* function performs network-intensive operations more suited to the IO dispatcher. This can easily be achieved using the *withContext* launcher, which allows the context of a coroutine to be changed while still staying in the same coroutine scope. The following change switches the *performTask2()* coroutine to an IO thread:

```
suspend fun performTasks() {
    performTask1()
    withContext(Dispatchers.IO) { performTask2() }
    performTask3()
}
```

When executed, the output will read as follows, indicating that the Task 2 coroutine is no longer on the main thread:

```
Task 1 main
Task 2 DefaultDispatcher-worker-1
```

Task 3 main

The `withContext` builder also provides an interesting alternative to using the `async` builder and the `Deferred` object `await()` call when returning a result. Using `withContext`, the code from the previous section can be rewritten as follows:

```
fun startTask(view: View) {

    coroutineScope.launch(Dispatchers.Main) {
        statusText.text = performSlowTask()
    }
}

suspend fun performSlowTask(): String =
    withContext(Dispatchers.Main) {
        Log.i(TAG, "performSlowTask before")
        delay(5_000)
        Log.i(TAG, "performSlowTask after")

        return@withContext "Finished"
    }
}
```

62.11 Coroutine Channel Communication

Channels provide a simple way to implement communication between coroutines, including streams of data. In the simplest form, this involves the creation of a `Channel` instance and calling the `send()` method to send the data. Once sent, transmitted data can be received in another coroutine via a call to the `receive()` method of the same `Channel` instance.

The following code, for example, passes six integers from one coroutine to another:

```
.
.
import kotlinx.coroutines.channels.*
.
.
val channel = Channel<Int>()

suspend fun channelDemo() {
    coroutineScope.launch(Dispatchers.Main) { performTask1() }
    coroutineScope.launch(Dispatchers.Main) { performTask2() }
}

suspend fun performTask1() {
    (1..6).forEach {
        channel.send(it)
    }
}
```

An Introduction to Kotlin Coroutines

```
suspend fun performTask2() {  
    repeat(6) {  
        Log.d(TAG, "Received: ${channel.receive()}")  
    }  
}
```

When executed, the following logcat output will be generated:

```
Received: 1  
Received: 2  
Received: 3  
Received: 4  
Received: 5  
Received: 6
```

62.12 Summary

Kotlin coroutines provide a simpler and more efficient approach to performing asynchronous tasks than traditional multi-threading. Coroutines allow asynchronous tasks to be implemented in a structured way without implementing the callbacks associated with typical thread-based tasks. This chapter has introduced the basic concepts of coroutines, including jobs, scope, builders, suspend functions, structured concurrency, and channel-based communication.

71. Understanding Android Content Providers

The previous chapter worked on creating an example application designed to store data using a SQLite database. When implemented this way, the data is private to the application and, as such, inaccessible to other applications running on the same device. While this may be the desired behavior for many application types, situations will inevitably arise whereby the data stored on behalf of an application could benefit other applications. A prime example is the data stored by the built-in Contacts application on an Android device. While the Contacts application is primarily responsible for managing the user's address book details, this data is also made accessible to any other applications needing access. This data sharing between Android applications is achieved through implementing *content providers*.

71.1 What is a Content Provider?

A content provider provides access to structured data between different Android applications. This data is exposed to applications either as tables of data (in much the same way as a SQLite database) or as a handle to a file. This essentially involves the implementation of a client/server arrangement whereby the application seeking access to the data is the client and the content provider is the server, performing actions and returning results on behalf of the client.

A successful content provider implementation involves several elements, each of which will be covered in detail in the remainder of this chapter.

71.2 The Content Provider

A content provider is created as a subclass of the *android.content.ContentProvider* class. Typically, the application responsible for managing the data to be shared will implement a content provider to facilitate sharing of that data with other applications.

Creating a content provider involves implementing methods to manage the data on behalf of other client applications. These methods are as follows:

71.2.1 onCreate()

This method is called when the content provider is first created and should be used to perform any initialization tasks required by the content provider.

71.2.2 query()

This method will be called when a client requests that data be retrieved from the content provider. This method identifies the data to be retrieved (single or multiple rows), performs the data extraction, and returns the results wrapped in a Cursor object.

71.2.3 insert()

This method is called when a new row needs to be inserted into the provider database. This method must identify the destination for the data, perform the insertion and return the full URI of the newly added row.

71.2.4 update()

The method called when existing rows need to be updated on behalf of the client. The method uses the arguments passed through to update the appropriate table rows and return the number of rows updated as a result of the operation.

71.2.5 delete()

Called when rows are to be deleted from a table. This method deletes the designated rows and returns a count of the number of rows deleted.

71.2.6 getType()

Returns the MIME type of the data stored by the content provider.

It is important when implementing these methods in a content provider to keep in mind that, with the exception of the *onCreate()* method, they can be called from many processes simultaneously and must, therefore, be thread safe.

Once a content provider has been implemented, the issue that then arises is how the provider is identified within the Android system. This is where the *content URI* comes into play.

71.3 The Content URI

An Android device will potentially contain several content providers. The system must, therefore, provide some way of identifying one provider from another. Similarly, a single content provider may provide access to multiple forms of content (typically in the form of database tables). Client applications, therefore, need a way to specify the underlying data for which access is required. This is achieved using content URIs.

The content URI is used to identify specific data within a specific content provider. The Authority section of the URI identifies the content provider and usually takes the form of the package name of the content provider. For example:

```
com.example.mydbapp.myprovider
```

A specific database table within the provider data structure may be referenced by appending the table name to the authority. For example, the following URI references a table named *products* within the content provider:

```
com.example.mydbapp.myprovider/products
```

Similarly, a specific row within the specified table may be referenced by appending the row ID to the URI. The following URI, for example, references the row in the *products* table in which the value stored in the ID column equals 3:

```
com.example.mydbapp.myprovider/products/3
```

When implementing the insert, query, update and delete methods in the content provider, it will be the responsibility of these methods to identify whether the incoming URI is targeting a specific row in a table, or references multiple rows, and act accordingly. This can potentially be a complex task given that a URI can extend to multiple levels. This process can, however, be eased significantly using the *UriMatcher* class, as will be outlined in the next chapter.

71.4 The Content Resolver

Access to a content provider is achieved via a *ContentResolver* object. An application can obtain a reference to its content resolver by calling the *getContentResolver()* method of the application context.

The content resolver object contains a set of methods that mirror those of the content provider (insert, query, delete etc.). The application simply makes calls to the methods, specifying the URI of the content on which the

operation is to be performed. The content resolver and content provider objects then communicate to perform the requested task on behalf of the application.

71.5 The <provider> Manifest Element

For a content provider to be visible within an Android system, it must be declared within the Android manifest file for the application in which it resides. This is achieved using the <provider> element, which must contain the following items:

- **android:authority** – The full authority URI of the content provider. For example `com.example.mydbapp.mydbapp.myprovider`.
- **android:name** – The name of the class that implements the content provider. In most cases, this will use the same value as the authority.

Similarly, the <provider> element may be used to define the permissions that must be held by client applications in order to qualify for access to the underlying data. If no permissions are declared, the default behavior is for permission to be allowed for all applications.

Permissions can be set to cover the entire content provider, or limited to specific tables and records.

71.6 Summary

The data belonging to an application is typically private to the application and inaccessible to other applications. Setting up a content provider is necessary when the data needs to be shared. This chapter has covered the basic elements that combine to enable data sharing between applications and outlined the concepts of the content provider, content URI, and content resolver.

In the next chapter, the SQLDemo project created previously will be extended to make the underlying customer data available via a content provider.

97. Working with Material Design 3 Theming

The appearance of an Android app is intended to conform to a set of guidelines defined by Material Design. Google developed Material Design to provide a level of design consistency between different apps while also allowing app developers to include their own branding in terms of color, typography, and shape choices (a concept referred to as Material theming). In addition to design guidelines, Material Design also includes a set of UI components for use when designing user interface layouts, many of which we have used throughout this book.

This chapter will provide an overview of how theming works within an Android Studio project and explore how the default design configurations provided for newly created projects can be modified to meet your branding requirements.

97.1 Material Design 2 vs. Material Design 3

Before beginning, it is important to note that Google is transitioning from Material Design 2 to Material Design 3 and that Android Studio Jellyfish projects default to Material Design 3. Material Design 3 provides the basis for Material You, a feature introduced in Android 12 that allows an app to automatically adjust theme elements to complement preferences configured by the user on the device. For example, dynamic color support provided by Material Design 3 allows the colors used in apps to adapt automatically to match the user's wallpaper selection.

97.2 Understanding Material Design Theming

We know that Android app user interfaces are created by assembling components such as layouts, text fields, and buttons. These components appear using default colors unless we specifically override a color attribute in the XML layout resource file or by writing code. The project's theme defines these default colors. The theme consists of a set of color slots (declared in *themes.xml* files) which are assigned color values (declared in the *colors.xml* file). Each UI component is programmed internally to use theme color slots as the default color for specific attributes (such as the foreground and background colors of the Text widget). It follows, therefore, that we can change the application-wide theme of an app by changing the colors assigned to specific theme slots. When the app runs, the new default colors will be used for all widgets when the user interface is rendered.

97.3 Material Design 3 Theming

Before exploring Material Design 3, we must consider how it is used in an Android Studio project. The theme used by an application project is declared as a property of the *application* element within the *AndroidManifest.xml* file, for example:

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

    <application
    .
    .
```

Working with Material Design 3 Theming

```
android:supportsRtl="true"  
android:theme="@style/Theme.MyDemoApp"  
tools:targetApi="31">  
<activity
```

As previously discussed, all of the files associated with the project theme are contained within the *colors.xml* and *themes.xml* files located in the *res -> values* folder, as shown in Figure 97-1:

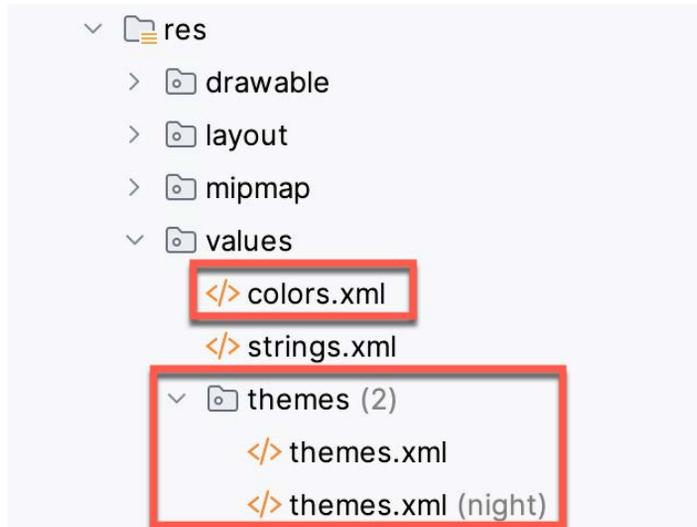


Figure 97-1

The theme itself is declared in the two *themes.xml* files located in the *themes* folder. These resource files declare different color palettes containing Material Theme color slots for use when the device is in light or dark (night) mode. Note that the style name property in each file must match that referenced in the *AndroidManifest.xml* file, for example:

```
<resources xmlns:tools="http://schemas.android.com/tools">  
  <!-- Base application theme. -->  
  <style name="Base.Theme.MyDemoApp" parent="Theme.Material3.DayNight.  
NoActionBar">  
    <!-- Customize your light theme here. -->  
    <!-- <item name="colorPrimary">@color/my_light_primary</item> -->  
  </style>  
  
  <style name="Theme.MyDemoApp" parent="Base.Theme.MyDemoApp" />  
</resources>
```

These color slots (also referred to as *color attributes*) are used by the Material components to set colors when they are rendered on the screen. For example, the *colorPrimary* color slot is used as the background color for the Material Button component.

Color slots in MD3 are grouped as Primary, Secondary, Tertiary, Error, Background, and Surface. These slots are further divided into pairs consisting of a *base color* and an "on" *base color*. This generally translates to the background and foreground colors of a Material component.

The particular group used for coloring will differ between widgets. A Material Button widget, for example, will use the *colorPrimary* base color for the background color and *colorOnPrimary* for its content (i.e., the text or icon it displays). The *FloatingActionButton* component, on the other hand, uses *colorPrimaryContainer* as the background color and *colorOnPrimaryContainer* for the foreground. The correct group for a specific widget type can usually be identified quickly by changing color settings in the theme files and reviewing the rendering in the layout editor.

Suppose that we need to change *colorPrimary* to red. We achieve this by adding a new entry to the *colors.xml* file for the red color and then assigning it to the *colorPrimary* slot in the *themes.xml* file. The *colorPrimary* slot in an MD3 theme night, therefore, read as follows:

```
<resources xmlns:tools="http://schemas.android.com/tools">
    <!-- Base application theme. -->
    <style name="Base.Theme.MyDemoApp" parent="Theme.Material3.DayNight.NoActionBar">
        <item name="colorPrimary">@color/my_bright_primary</item>
    </style>

    <style name="Theme.MyDemoApp" parent="Base.Theme.MyDemoApp" />
</resources>
```

This color is then declared in the *colors.xml* file:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
.
.
    <color name="my_bright_primary">#FC0505</color>
</resources>
```

97.4 Building a Custom Theme

As we have seen, the coding work in implementing a theme is relatively simple. The difficult part, however, is often choosing complementary colors to make up the theme. Fortunately, Google has developed a tool that makes it easy to design custom color themes for your apps. This tool is called the Material Theme Builder and is available at:

<https://m3.material.io/theme-builder#/custom>

On the custom screen (Figure 97-2), make a color selection for the primary color key (A) by clicking on the color circle to display the color selection dialog. Once a color has been selected, the preview (B) will change to reflect the recommended colors for all MD3 color slots, along with example app interfaces and widgets. In addition, you can override the generated colors for the Secondary, Tertiary, and Neutral slots by clicking on the corresponding color circles to display the color selection dialog.

The area marked B displays example app interfaces, light and dark color scheme charts, and widgets that update to preview your color selections. Since the panel is longer than the typical browser window, you must scroll down to see all the information.

To incorporate the theme into your design, click the Export button (C) and select the Android View (XML) option. Once downloaded, the *colors.xml* and *themes.xml* files can be used to replace the existing files in your project. Note that the theme name in the two exported *themes.xml* files must be changed to match your project.

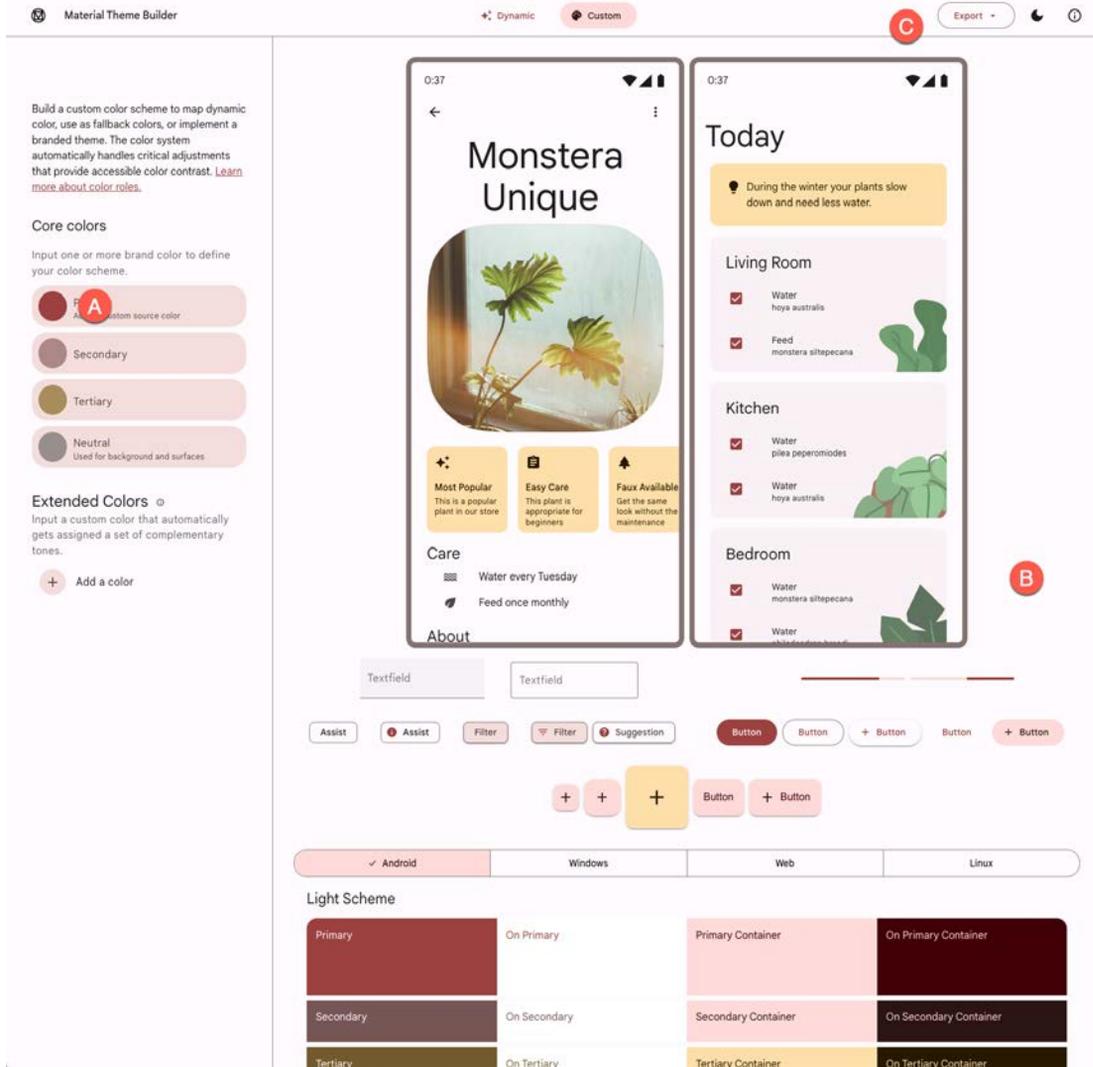


Figure 97-2

97.5 Summary

Material Design provides guidelines and components defining how Android apps appear. Individual branding can be applied to an app by designing themes that specify the colors, fonts, and shapes used when displaying the app. Google recently introduced Material Design 3, which replaces Material Design 2 and supports the new features of Material You, including dynamic colors. Google also provides the Material Theme Builder for designing your own themes, which eases the task of choosing complementary theme colors. Once this tool has been used to design a theme, the corresponding files can be exported and used within an Android Studio project.

98. A Material Design 3 Theming and Dynamic Color Tutorial

This chapter will show you how to create a new Material Design 3 theme using the Material Theme Builder tool and integrate it into an Android Studio project. The tutorial will also demonstrate how to add support for and test dynamic theme colors to an app.

98.1 Creating the ThemeDemo Project

Select the *New Project* option from the welcome screen and, within the resulting new project dialog, choose the Empty Views Activity template before clicking on the Next button.

Enter *ThemeDemo* into the Name field and specify *com.ebookfrenzy.themedemo* as the package name. Before clicking on the Finish button, change the Minimum API level setting to API 26: Android 8.0 (Oreo) and the Language menu to Kotlin.

98.2 Designing the User Interface

The main activity will consist of a simple layout containing some user interface components that will enable us to see the effects of the theming work performed later in the chapter. For information on MD3 components, refer to the following web page:

<https://material.io/blog/migrating-material-3>

The layout will be designed within the *activity_main.xml* file, which currently contains a single Text view. Open this file in the layout editor, delete the Text view, turn off Autoconnect mode (marked A in Figure 98-1), and click on the button to clear all constraints from the layout (B).

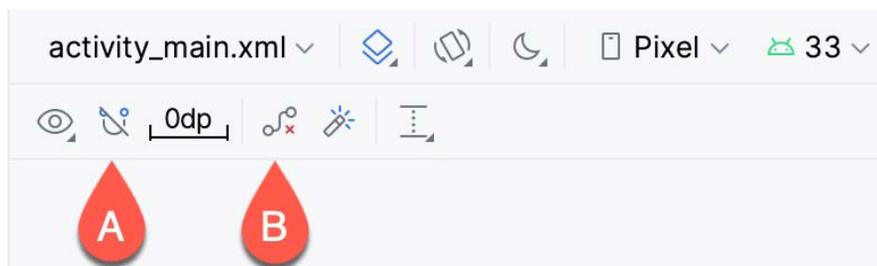


Figure 98-1

From the Buttons section of the Palette, drag Chip, CheckBox, Switch, and Button views onto the layout canvas. Next, drag a FloatingActionButton onto the layout canvas to position it beneath the Button component. When prompted to choose an icon to appear on the FloatingActionButton, select the *ic_lock_power_off* icon from within the resource tool window, as illustrated in Figure 98-2:

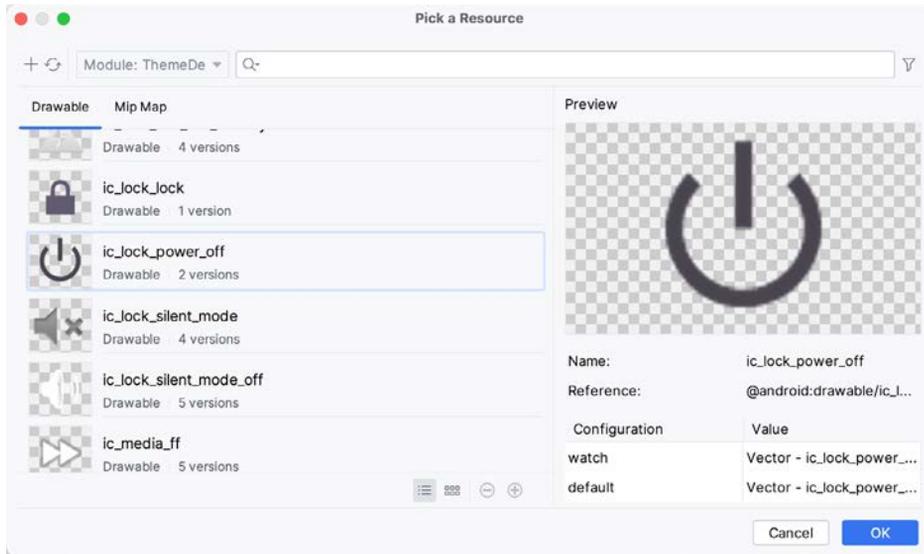


Figure 98-2

Change the text attribute for the Chip widget to “This is my chip” and set the chipIcon attribute to `@android:drawable/ic_btn_speak_now` so that the layout resembles that shown to the left in Figure 98-3:

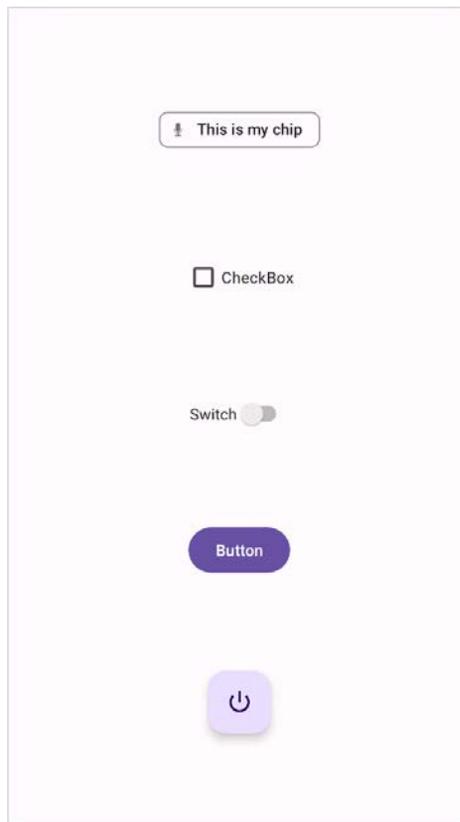


Figure 98-3

To set up the constraints, select all the components, right-click on the Chip view, and select *Chains* -> *Create Vertical Chain* from the resulting menu. Repeat this step, this time selecting the *Center* -> *Horizontally in Parent* menu option.

Compile and run the app on a device or emulator and verify that the user interface matches that shown in Figure 98-3 above. The next step is to create a custom theme and apply it to the project.

98.3 Building a New Theme

Begin by opening a browser window and navigating to the following URL to access the builder tool:

<https://m3.material.io/theme-builder#/custom>

Once you have loaded the builder, select a wallpaper followed by the Custom button at the top of the screen, and then click on the Primary color circle in the Core Colors section to display the color selector. From the color selector, choose any color as the basis for your theme:

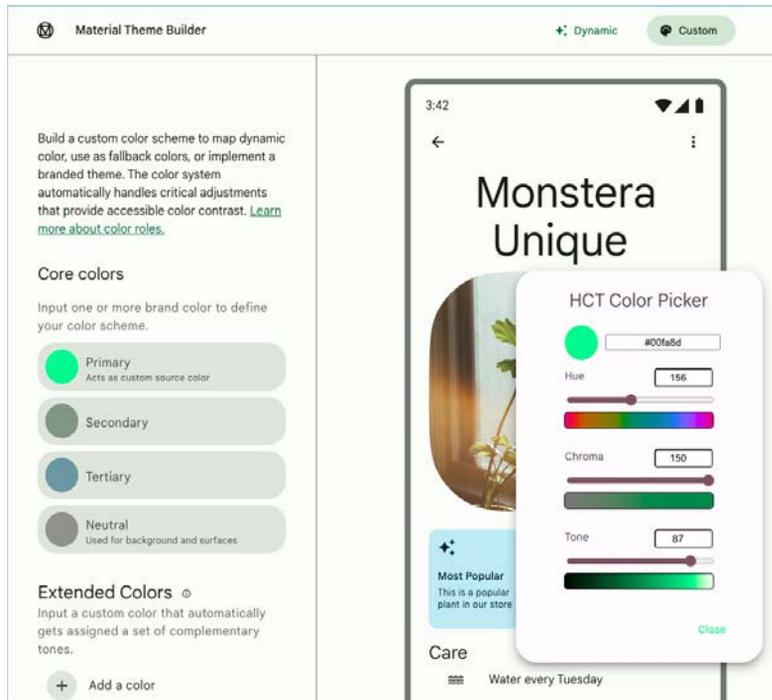


Figure 98-4

Review the color scheme in the Your Theme panel and make any necessary color adjustments using the Core colors settings until you are happy with the color slots. Once the theme is ready, click the Export button in the top right-hand corner and select the *Android Views (XML)* option. When prompted, save the file to a suitable location on your computer filesystem. The theme will be saved as a compressed file named *material-theme.zip*.

Using the appropriate tool for your operating system, unpack the theme file, which should contain the following folders and files in a folder named *material-theme*:

- **values/colors.xml** - The color definitions.
- **values/themes.xml** - The theme for the light mode.
- **values-night/themes.xml** - The theme for dark mode.

Now that the theme files have been generated, they need to be integrated into the Android Studio project.

98.4 Adding the Theme to the Project

Before adding the new theme to the project, we first need to remove the old theme files. This is easier if the Project tool window is in Project Files mode. To switch mode, use the menu at the top of the tool Project tool window as shown below and select the Project Files option:

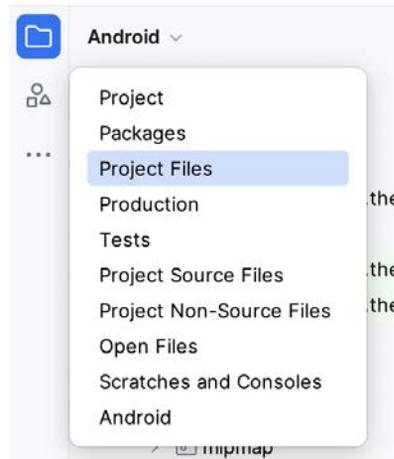


Figure 98-5

With Project Files mode selected, navigate to the `app -> src -> main -> res -> values` folder and select and delete the `colors.xml` and `themes.xml` files. Also, delete the `themes.xml` file located in the `values-night` folder.

Open the filesystem navigation tool for your operating system, locate the `colors.xml` and `themes.xml` files in the `values` folder of the new material theme, and copy and paste them into the `values` folder within the Project tool window. Repeat this step to copy the `themes.xml` file in the `values-night` folder, this time pasting it into the `values-night` folder.

Switch the Project tool window back to Android mode, at which point the value resource files section should match Figure 98-6:



Figure 98-6

Next, modify the light `themes.xml` file to match the current project as follows:

```
<resources>
  <style name="Base.Theme.ThemeDemo" parent="Theme.Material3.Light.NoActionBar">
    <item name="colorPrimary">@color/md_theme_light_primary</item>
    <item name="colorOnPrimary">@color/md_theme_light_onPrimary</item>
```

```

.
.
</style>

<style name="Theme.ThemeDemo" parent="Base.Theme.ThemeDemo" />

</resources>

```

Repeat these steps to make the same modifications to the *themes.xml (night)* file.

Return to the *activity_main.xml* file or rerun the app to confirm that the user interface is rendered using the custom theme colors.

98.5 Enabling Dynamic Color Support

The app will need to be run on a device or emulator running Android 12 or later with the correct Wallpaper settings to test dynamic colors. On the device or emulator, launch the Settings app and select *Wallpaper & style* from the list of options. On the wallpaper settings screen, click the option to change the wallpaper (marked A in Figure 98-7) and select a wallpaper image containing colors that differ significantly from the colors in your theme. Once selected, assign the wallpaper to the Home screen.

Return to the Wallpaper & styles screen and make sure that the *Wallpaper colors* option is selected (B) before choosing an option from the color scheme buttons (C). As each option is clicked, the wallpaper example will change to reflect the selection:

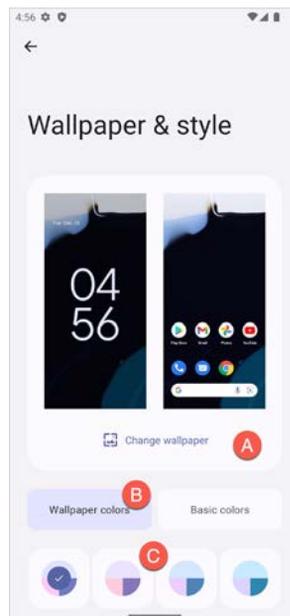


Figure 98-7

To enable dynamic colors, we need to call the *applyToActivitiesIfAvailable()* method of the *DynamicColors* class. To enable dynamic color support for the entire app, this needs to be called from within the *onCreate()* method of a custom *Application* instance. Begin by adding a new Kotlin class file to the project under *app -> kotlin+java -> com.ebookfrenzy.themedemo* named *ThemeDemoApplication.kt* and modifying it so that it reads as follows:

```
package com.ebookfrenzy.themedemo
```

```
import android.app.Application
import com.google.android.material.color.DynamicColors

class ThemeDemoApplication: Application() {
    override fun onCreate() {
        super.onCreate()
        DynamicColors.applyToActivitiesIfAvailable(this)
    }
}
```

With the custom Application class created, we must configure the project to use this class instead of the default Application instance. To do this, edit the *AndroidManifest.xml* file and add an *android:name* element referencing the new class:

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

    <application
        android:name=".ThemeDemoApplication"
        android:allowBackup="true"
        .
        .
```

Build and run the app and note that the layout uses a theme matching the wallpaper color. Place the ThemeDemo app into the background, return to the *Wallpaper & styles* settings screen, and choose a different wallpaper. Bring the ThemeDemo app to the foreground again. At this point, it will have dynamically adapted to match the new wallpaper.

98.6 Previewing Dynamic Colors

Dynamic color behavior can also be previewed within the Android Studio layout editor. To try this, open the *activity_main.xml* file, click on the theme menu, and select the More Themes option, as shown in Figure 98-8:

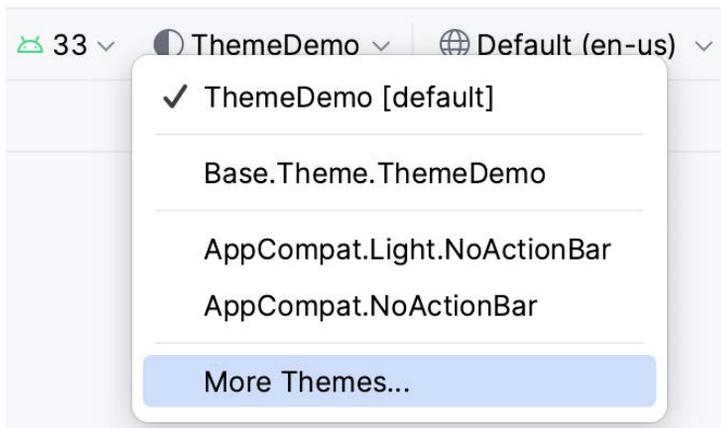


Figure 98-8

Next, use the search field in the theme selection dialog to list dynamic themes:

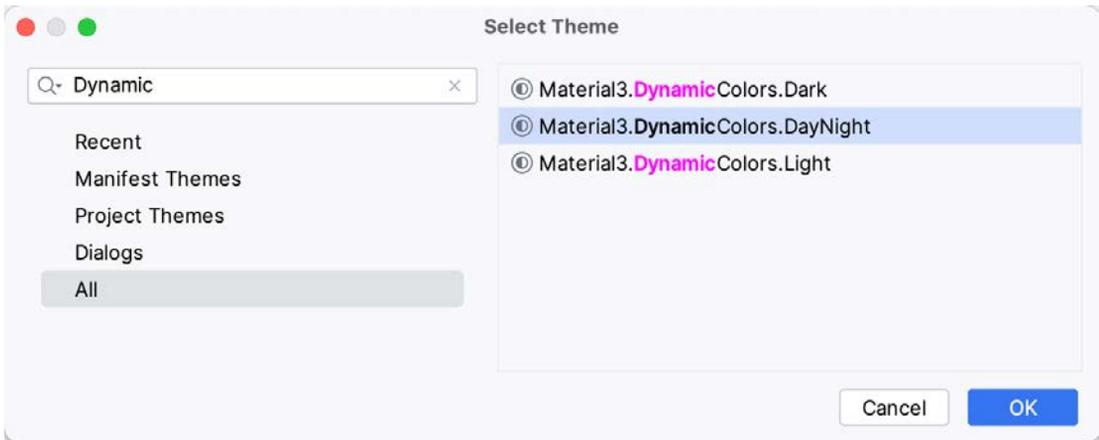


Figure 98-9

Select the *Material3.DynamicColors.DayNight* theme before clicking on the OK button. On returning to the layout editor, select the System UI Mode menu and choose one of the wallpaper options as highlighted in Figure 98-10:

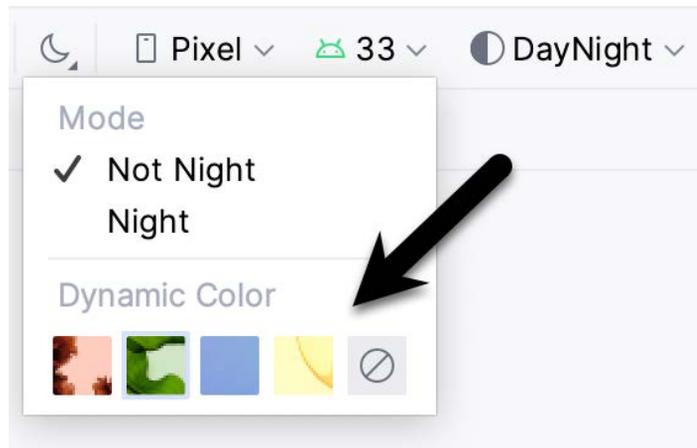


Figure 98-10

Once a wallpaper has been selected, the colors of the components in the layout will change accordingly.

98.7 Summary

In this chapter, we have used the Material Theme Builder to design a new theme and explained the steps to integrate the generated theme files into an Android Studio project. Finally, the chapter demonstrated how to implement and use the Material You dynamic colors feature.

99. An Overview of Gradle in Android Studio

In the “A Guide to Gradle Version Catalogs” chapter, we introduced the library version catalog and explained how the Gradle build system relies on it to ensure that projects are built using the correct libraries and versions. Aside from some modifications to the version catalog and library decencies in the intervening chapters, it has been taken for granted that Android Studio will take the necessary steps to compile and run the application projects that have been created. Android Studio has been achieving this in the background using a system known as *Gradle*.

It is time to look at how Gradle is used to compile and package an application project’s various elements and begin exploring how to configure this system when more advanced requirements are needed for building projects in Android Studio.

99.1 An Overview of Gradle

Gradle is an automated build toolkit that allows how projects are built to be configured and managed through a set of build configuration files. This includes defining how a project will be built, what dependencies need to be fulfilled to build successfully, and what the build process’s end result (or results) should be.

The strength of Gradle lies in the flexibility that it provides to the developer. The Gradle system is a self-contained, command-line-based environment that can be integrated into other environments using plugins. In the case of Android Studio, Gradle integration is provided through the appropriately named Android Studio Plugin.

Although the Android Studio Plug-in allows Gradle tasks to be initiated and managed from within Android Studio, the Gradle command-line wrapper can still be used to build Android Studio-based projects, including on systems on which Android Studio is not installed.

The configuration rules to build a project are declared in Gradle build files and scripts based on the Groovy programming language.

99.2 Gradle and Android Studio

Gradle brings many powerful features to building Android application projects. Some of the key features are as follows:

99.2.1 Sensible Defaults

Gradle implements a concept referred to as *convention over configuration*. This means that Gradle has a predefined set of sensible default configuration settings that will be used unless settings in the build files override them. This means that builds can be performed with the minimum configuration required by the developer. Changes to the build files are only needed when the default configuration does not meet your build needs.

99.2.2 Dependencies

Another key area of Gradle functionality is that of dependencies. Consider, for example, a module within an Android Studio project which triggers an intent to load another module in the project. The first module has, in effect, a dependency on the second module since the application will fail to build if the second module cannot be located and launched at runtime. This dependency can be declared in the Gradle build file for the first module

so that the second module is included in the application build, or an error flagged if the second module cannot be found or built. Other examples of dependencies are libraries and JAR files on which the project depends to compile and run.

Gradle dependencies can be categorized as *local* or *remote*. A local dependency references an item that is present on the local file system of the computer system on which the build is being performed. A remote dependency refers to an item that is present on a remote server (typically referred to as a *repository*).

Remote dependencies are handled for Android Studio projects using another project management tool named *Maven*. If a remote dependency is declared in a Gradle build file using Maven syntax, then the dependency will be downloaded automatically from the designated repository and included in the build process. The following dependency declaration, for example, causes the AppCompatActivity library to be added to the project from the Google repository:

```
implementation(libs.androidx.appcompat)
```

99.2.3 Build Variants

In addition to dependencies, Gradle also provides *build variant* support for Android Studio projects. This allows multiple variations of an application to be built from a single project. Android runs on many different devices encompassing a range of processor types and screen sizes. To target as wide a range of device types and sizes as possible, it will often be necessary to build several variants of an application (for example, one with a user interface for phones and another for tablet-sized screens). Through the use of Gradle, this is now possible in Android Studio.

99.2.4 Manifest Entries

Each Android Studio project has associated with it an *AndroidManifest.xml* file containing configuration details about the application. Several manifest entries can be specified in Gradle build files which are then auto-generated into the manifest file when the project is built. This capability complements the build variants feature, allowing elements such as the application version number, application ID, and SDK version information to be configured differently for each build variant.

99.2.5 APK Signing

The chapter “*Creating, Testing, and Uploading an Android App Bundle*” covered creating a signed release APK file using the Android Studio environment. It is also possible to include the signing information entered through the Android Studio user interface within a Gradle build file to generate signed APK files from the command line.

99.2.6 ProGuard Support

ProGuard is a tool included with Android Studio that optimizes, shrinks, and obfuscates Java byte code to make it more efficient and harder to reverse engineer (the method by which others can identify the logic of an application through analysis of the compiled Java byte code). The Gradle build files allow you to control whether or not ProGuard is run on your application when it is built.

99.3 The Property and Settings Gradle Build File

The gradle build configuration consists of configuration, property, and settings files. The *gradle.properties* file, for example, contains mostly esoteric settings relating to the command-line flags used by the Java Virtual Machine (JVM), whether or not the project uses the AndroidX libraries and Kotlin coding style support. As a typical user, it is unlikely that you will need to change any of these settings in this file.

The *settings.gradle.kts* file, on the other hand, defines which online repositories are to be searched when the build system needs to download and install any additional libraries and plugins required to build the project and the project name. A typical *settings.gradle.kts* file will read as follows:

```
pluginManagement {
```

```

    repositories {
        google()
        mavenCentral()
        gradlePluginPortal()
    }
}
dependencyResolutionManagement {
    repositoriesMode.set(RepositoriesMode.FAIL_ON_PROJECT_REPOS)
    repositories {
        google()
        mavenCentral()
    }
}

rootProject.name = "ThemeDemo"
include(":app")

```

As with the *gradle.properties* file, it is unlikely that changes will need to be made to this file.

99.4 The Top-level Gradle Build File

A completed Android Studio project contains everything needed to build an Android application and consists of modules, libraries, manifest files, and Gradle build files.

Each project contains one top-level Gradle build file. This file is listed as *build.gradle.kts* (*Project: <project name>*) and can be found in the project tool window as highlighted in Figure 99-1:

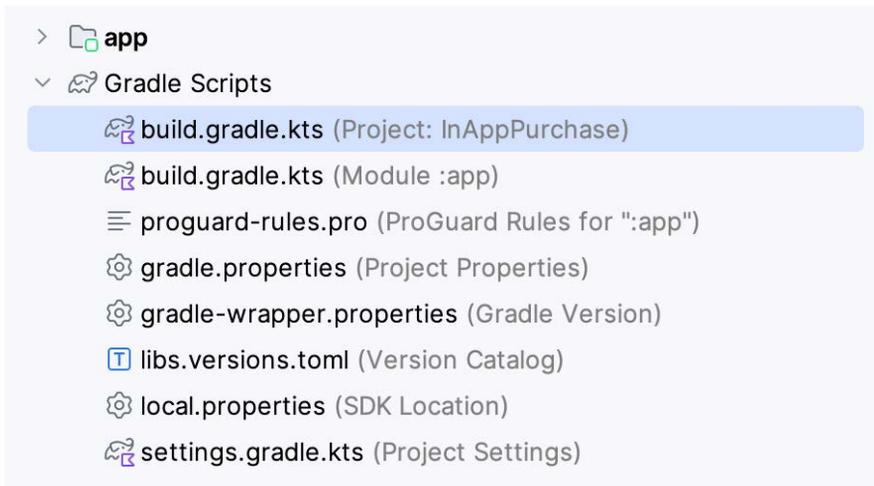


Figure 99-1

By default, the contents of the top-level Gradle build file reads as follows:

```

plugins {
    alias(libs.plugins.androidApplication) apply false
    alias(libs.plugins.jetbrainsKotlinAndroid) apply false
}

```

In most situations, making any changes to this build file is unnecessary.

99.5 Module Level Gradle Build Files

An Android Studio application project is made up of one or more modules. Take, for example, a hypothetical application project named GradleDemo which contains modules named Module1 and Module2, respectively. In this scenario, each module will require its own Gradle build file. In terms of the project structure, these would be located as follows:

- Module1/build.gradle.kts
- Module2/build.gradle.kts

By default, the Module1 *build.gradle.kts* file would resemble that of the following listing:

```
plugins {
    alias(libs.plugins.androidApplication)
    alias(libs.plugins.jetbrainsKotlinAndroid)
}

android {
    namespace = "com.example.gradlesample"
    compileSdk = 34

    defaultConfig {
        applicationId = "com.example.gradlesample"
        minSdk = 26
        targetSdk = 34
        versionCode = 1
        versionName = "1.0"

        testInstrumentationRunner = "androidx.test.runner.AndroidJUnitRunner"
    }

    buildTypes {
        release {
            isMinifyEnabled = false
            proguardFiles(
                getDefaultProguardFile("proguard-android-optimize.txt"),
                "proguard-rules.pro"
            )
        }
    }

    compileOptions {
        sourceCompatibility = JavaVersion.VERSION_1_8
        targetCompatibility = JavaVersion.VERSION_1_8
    }

    kotlinOptions {
        jvmTarget = "1.8"
    }
}
```

```

    }
}

dependencies {

    implementation(libs.androidx.core.ktx)
    implementation(libs.androidx.appcompat)
    implementation(libs.material)
    implementation(libs.androidx.activity)
    implementation(libs.androidx.constraintlayout)
    testImplementation(libs.junit)
    androidTestImplementation(libs.androidx.junit)
    androidTestImplementation(libs.androidx.espresso.core)
}

```

As is evident from the file content, the build file begins by declaring the use of the Gradle Android application and Kotlin plug-ins:

```

plugins {
    alias(libs.plugins.androidApplication)
    alias(libs.plugins.jetbrainsKotlinAndroid)
}

```

The *android* section of the file declares the project namespace and then states the version of the SDK to be used when building Module1.

```

android {
    namespace = "com.example.gradlesample"
    compileSdk = 34
}

```

The items declared in the *defaultConfig* section define elements to be generated into the module's *AndroidManifest.xml* file during the build. These settings, which may be modified in the build file, are taken from the settings entered within Android Studio when the module was first created:

```

defaultConfig {
    applicationId = "com.example.gradlesample"
    minSdk = 26
    targetSdk = 34
    versionCode = 1
    versionName = "1.0"

    testInstrumentationRunner = "androidx.test.runner.AndroidJUnitRunner"
}

```

The *buildTypes* section contains instructions on whether and how to run ProGuard on the APK file when a release version of the application is built:

```

buildTypes {
    release {
        isMinifyEnabled = false
        proguardFiles(

```

An Overview of Gradle in Android Studio

```
        getDefaultProguardFile("proguard-android-optimize.txt"),
        "proguard-rules.pro"
    )
}
}
```

As currently configured, ProGuard will not be run when Module1 is built. To enable ProGuard, the *minifyEnabled* entry must be changed from *false* to *true*. The *proguard-rules.pro* file can be found in the module directory of the project. Changes made to this file override the default settings in the *proguard-android.txt* file, which is located in the Android SDK installation directory under *sdk/tools/proguard*.

Since no debug buildType is declared in this file, the defaults will be used (built without ProGuard, signed with a debug key, and debug symbols enabled).

An additional section, entitled *productFlavors*, may also be included in the module build file to enable multiple build variants to be created.

Next, directives are included to specify the version of the Java compiler to be used when building the project:

```
compileOptions {
    sourceCompatibility JavaVersion.VERSION_1_8
    targetCompatibility JavaVersion.VERSION_1_8
}
kotlinOptions {
    jvmTarget = "1.8"
}
```

Finally, the dependencies section lists any local and remote dependencies on which the module depends. The dependency lines in the above example file designate the Android libraries that need to be included from the Android Repository:

```
dependencies {

    implementation(libs.androidx.core.ktx)
    implementation(libs.androidx.appcompat)
    implementation(libs.material)
    .
    .
}
```

Note that the dependency declarations include version numbers to indicate which library version should be included.

99.6 Configuring Signing Settings in the Build File

The “*Creating, Testing, and Uploading an Android App Bundle*” chapter of this book covered the steps involved in setting up keys and generating a signed release APK file using the Android Studio user interface. These settings may also be declared within a *signingConfigs* section of the *build.gradle.kts* file. For example:

```
.
.
    defaultConfig {
.
.
```

```

    }
    signingConfigs {
        release {
            storeFile file("keystore.release")
            storePassword "your keystore password here"
            keyAlias "your key alias here"
            keyPassword "your key password here"
        }
    }
    buildTypes {
    .
    .
    }

```

The above example embeds the key password information directly into the build file. An alternative to this approach is to extract these values from system environment variables:

```

signingConfigs {
    release {
        storeFile file("keystore.release")
        storePassword System.getenv("KEYSTOREPASSWD")
        keyAlias "your key alias here"
        keyPassword System.getenv("KEYPASSWD")
    }
}

```

Yet another approach is to configure the build file so that Gradle prompts for the passwords to be entered during the build process:

```

signingConfigs {
    release {
        storeFile file("keystore.release")
        storePassword System.console().readLine(
            "\nEnter Keystore password: ")
        keyAlias "your key alias here"
        keyPassword System.console().readLine("\nEnter Key password: ")
    }
}

```

99.7 Running Gradle Tasks from the Command Line

Each Android Studio project contains a Gradle wrapper tool to invoke Gradle tasks from the command line. This tool is located in the root directory of each project folder. While this wrapper is executable on Windows systems, it may need to have execute permission enabled on Linux and macOS before it can be used. To enable execute permission, open a terminal window, change directory to the project folder for which the wrapper is needed, and execute the following command:

```
chmod +x gradlew
```

Once the file has execute permissions, the location of the file will either need to be added to your \$PATH environment variable or the name prefixed by ./ to run. For example:

An Overview of Gradle in Android Studio

```
./gradlew tasks
```

Gradle views project building in terms of several different tasks. A full listing of tasks that are available for the current project can be obtained by running the following command from within the project directory (remembering to prefix the command with a `./` if running on macOS or Linux):

```
gradlew tasks
```

To build a debug release of the project suitable for device or emulator testing, use the `assembleDebug` option:

```
gradlew assembleDebug
```

Alternatively, to build a release version of the application:

```
gradlew assembleRelease
```

99.8 Summary

For the most part, Android Studio performs application builds in the background without any intervention from the developer. This build process is handled using the Gradle system, an automated build toolkit designed to allow how projects are built to be configured and managed through a set of build configuration files. While the default behavior of Gradle is adequate for many basic project build requirements, the need to configure the build process is inevitable with more complex projects. This chapter has provided an overview of the Gradle build system and configuration files within the context of an Android Studio project.

Index

Symbols

? 97
 <application> 504
 <fragment> 295
 <fragment> element 295
 <provider> 561
 <receiver> 482
 <service> 504, 510, 517
 :: operator 99
 .well-known folder 455, 478, 722

A

AbsoluteLayout 172
 ACCESS_COARSE_LOCATION permission 632
 ACCESS_FINE_LOCATION permission 632
 acknowledgePurchase() method 761
 ACTION_CREATE_DOCUMENT 783
 ACTION_CREATE_INTENT 784
 ACTION_DOWN 272
 ACTION_MOVE 272
 ACTION_OPEN_DOCUMENT intent 776
 ACTION_POINTER_DOWN 272
 ACTION_POINTER_UP 272
 ACTION_UP 272
 ACTION_VIEW 473
 Active / Running state 148
 Activity 83, 151
 adding views in Java code 249
 class 151
 creation 14
 Entire Lifetime 155
 Foreground Lifetime 155
 lifecycle methods 153
 lifecycles 145
 returning data from 452

 state change example 159
 state changes 151
 states 148
 Visible Lifetime 155
 Activity Lifecycle 147
 Activity Manager 82
 ActivityResultLauncher 453
 Activity Stack 147
 Actual screen pixels 240
 adb
 command-line tool 59
 connection testing 65
 device pairing 63
 enabling on Android devices 59
 Linux configuration 62
 list devices 59
 macOS configuration 60
 overview 59
 restart server 60
 testing connection 65
 WiFi debugging 63
 Windows configuration 61
 Wireless debugging 63
 Wireless pairing 63
 addCategory() method 481
 addMarker() method 685
 addView() method 243
 ADD_VOICEMAIL permission 632
 android
 exported 505
 gestureColor 288
 layout_behavior property 445
 onClick 297
 process 505, 517
 uncertainGestureColor 288
 Android
 Activity 83
 architecture 79
 events 265

Index

- intents 84
- onClick Resource 265
- runtime 80
- SDK Packages 5
- android.app 80
- Android Architecture Components 311
- android.content 80
- android.content.Intent 451
- android.database 80
- Android Debug Bridge. *See* ADB
- Android Development
 - System Requirements 3
- Android Devices
 - designing for different 171
- android.graphics 81
- android.hardware 81
- android.intent.action 487
- android.intent.action.BOOT_COMPLETED 505
- android.intent.action.MAIN 473
- android.intent.category.LAUNCHER 473
- Android Libraries 80
- android.media 81
- Android Monitor tool window 32
- Android Native Development Kit 281
- android.net 81
- android.opengl 81
- android.os 81
- android.permission.RECORD_AUDIO 641
- android.print 81
- Android Project
 - create new 13
- android.provider 81
- Android SDK Location
 - identifying 9
- Android SDK Manager 7, 9
- Android SDK Packages
 - version requirements 7
- Android SDK Tools
 - command-line access 8
 - Linux 10
 - macOS 10
 - Windows 7 9
- Windows 8 9
- Android Software Stack 79
- Android Storage Access Framework 776
- Android Studio
 - changing theme 57
 - downloading 3
 - Editor Window 52
 - installation 4
 - Linux installation 5
 - macOS installation 4
 - Navigation Bar 51
 - Project tool window 52
 - Status Bar 52
 - Toolbar 51
 - Tool window bars 52
 - tool windows 52
 - updating 11
 - Welcome Screen 49
 - Windows installation 4
- android.text 81
- android.util 81
- android.view 81
- android.view.View 174
- android.view.ViewGroup 171, 174
- Android Virtual Device. *See* AVD
 - overview 27
- Android Virtual Device Manager 27
- android.webkit 81
- android.widget 81
- AndroidX libraries 810
- API Key 677
- APK analyzer 754
- APK file 747
- APK File
 - analyzing 754
- APK Signing 810
- APK Wizard dialog 746
- App Architecture
 - modern 311
- AppBar
 - anatomy of 443
- AppBarScrollingViewBehavior 445

- App Bundles 743
 - creating 747
 - overview 743
 - revisions 753
 - uploading 750
- AppCompatActivity class 152
- App Inspector 53
- Application
 - stopping 32
- Application Context 85
- Application Framework 82
- Application Manifest 85
- Application Resources 85
- App Link
 - Adding Intent Filter 730
 - Digital Asset Links file 722, 455
 - Intent Filter Handling 730
 - Intent Filters 721
 - Intent Handling 722
 - Testing 734
 - URL Mapping 727
- App Links 721
 - auto verification 454
 - autoVerify 455
 - overview 721
- Apply Changes 257
 - Apply Changes and Restart Activity 257
 - Apply Code Changes 257
 - fallback settings 259
 - options 257
 - Run App 257
 - tutorial 259
- applyToActivitiesIfAvailable() method 805
- Architecture Components 311
- ART 80
- as 99
- as? 99
- asFlow() builder 523
- assetlinks.json , 722, 455
- asSharedFlow() 532
- asStateFlow() 531
- async 491

- Attribute Keyframes 382
- Audio
 - supported formats 639
- Audio Playback 639
- Audio Recording 639
- Auto Blocker 60
- Autoconnect Mode 205
- Automatic Link Verification 454, 477
- autoVerify 455, 730
- AVD
 - Change posture 47
 - cold boot 44
 - command-line creation 27
 - creation 27
 - device frame 36
 - Display mode 46
 - launch in tool window 36
 - overview 27
 - quickboot 44
 - Resizable 46
 - running an application 30
 - Snapshots 43
 - standalone 33
 - starting 29
 - Startup size and orientation 30

B

- Background Process 146
- Barriers 198
 - adding 217
 - constrained views 198
- Baseline Alignment 197
- beginTransaction() method 296
- BillingClient 762
 - acknowledgePurchase() method 761
 - consumeAsync() method 761
 - getPurchaseState() method 760
 - initialization 758, 767
 - launchBillingFlow() method 760
 - queryProductDetailsAsync() method 759
 - queryPurchasesAsync() method 762
- BillingResult 774

Index

- getDebugMessage() 774
- Binding Expressions 331
 - one-way 331
 - two-way 332
- BIND_JOB_SERVICE permission 505
- bindService() method 503, 507, 511
- Biometric Authentication 735
 - callbacks 739
 - overview 735
 - tutorial 735
- Biometric Prompt 740
- BitmapFactory 778
- Bitwise AND 105
- Bitwise Inversion 104
- Bitwise Left Shift 106
- Bitwise OR 105
- Bitwise Right Shift 106
- Bitwise XOR 105
- black activity 14
- Blank template 175
- Blueprint view 203
- BODY_SENSORS permission 632
- Boolean 92
- Bound Service 503, 507
 - adding to a project 508
 - Implementing the Binder 508
 - Interaction options 507
- BoundService class 509
- Broadcast Intent 481
 - example 483
 - overview 84, 481
 - sending 484
 - Sticky 483
- Broadcast Receiver 481
 - adding to manifest file 486
 - creation 485
 - overview 84, 482
- BroadcastReceiver class 482
- BroadcastReceiver superclass 485
- BufferedReader object 786
- buffer() operator 525
- Build Variants , 54

- tool window 54
- Bundle class 168
- Bundled Notifications 660

C

- Calendar permissions 632
- CALL_PHONE permission 632
- CAMERA permission 632
- Camera permissions 632
- CameraUpdateFactory class
 - methods 686
- cancelAndJoin() 491
- cancelChildren() 491
- CancellationSignal 740
- Canvas class 716
- CardView
 - layout file 433
 - responding to selection of 441
- CardView class 433
- CATEGORY_OPENABLE 776
- C/C++ Libraries 81
- Chain bias 226
- chain head 196
- chains 196
- Chains
 - creation of 223
- Chain style
 - changing 225
- chain styles 196
- Char 92
- CheckBox 171
- checkSelfPermission() method 636
- Circle class 673
- Code completion 70
- Code Editor
 - basics 67
 - Code completion 70
 - Code Generation 72
 - Code Reformatting 75
 - Document Tabs 68
 - Editing area 68
 - Gutter Area 68

- Live Templates 76
- Splitting 70
- Statement Completion 72
- Status Bar 69
- Code Generation 72
- Code Reformatting 75
- code samples
 - download 1
- cold boot 44
- Cold flows 531
- CollapsingToolBarLayout
 - example 446
 - introduction 446
 - parallax mode 446
 - pin mode 446
 - setting scrim color 449
 - setting title 449
 - with image 446
- collectLatest() operator 524
- Color class 717
- COLOR_MODE_COLOR 692, 712
- COLOR_MODE_MONOCHROME 692, 712
- combine() operator 530
- Common Gestures 277
 - detection 277
- Communicating Sequential Processes 489
- Companion Objects 129
- Component tree 17
- conflate() operator 525
- Constraint Bias 195
 - adjusting 209
- ConstraintLayout
 - advantages of 201
 - Availability 202
 - Barriers 198
 - Baseline Alignment 197
 - chain bias 226
 - chain head 196
 - chains 196
 - chain styles 196
 - Constraint Bias 195
 - Constraints 193
 - conversion to 221
 - convert to MotionLayout 389
 - deleting constraints 208
 - guidelines 215
 - Guidelines 198
 - manual constraint manipulation 205
 - Margins 194, 209
 - Opposing Constraints 194, 211
 - overview of 193
 - Packed chain 197, 226
 - ratios 201, 227
 - Spread chain 196
 - Spread inside 226
 - Spread inside chain 196
 - tutorial 231
 - using in Android Studio 203
 - Weighted chain 196, 226
 - Widget Dimensions 197, 213
 - Widget Group Alignment 219
- ConstraintLayout chains
 - creation of 223
 - in layout editor 223
- ConstraintLayout Chain style
 - changing 225
- Constraints
 - deleting 208
- ConstraintSet
 - addToHorizontalChain() method 246
 - addToVerticalChain() method 246
 - alignment constraints 245
 - apply to layout 244
 - applyTo() method 244
 - centerHorizontally() method 245
 - centerVertically() method 245
 - chains 245
 - clear() method 246
 - clone() method 245
 - connect() method 244
 - connect to parent 244
 - constraint bias 245
 - copying constraints 245
 - create 244

Index

- create connection 244
- createHorizontalChain() method 245
- createVerticalChain() method 245
- guidelines 246
- removeFromHorizontalChain() method 246
- removeFromVerticalChain() method 246
- removing constraints 246
- rotation 247
- scaling 246
- setGuidelineBegin() method 246
- setGuidelineEnd() method 246
- setGuidelinePercent() method 246
- setHorizontalBias() method 245
- setRotationX() method 247
- setRotationY() method 247
- setScaleX() method 246
- setScaleY() method 246
- setTransformPivot() method 247
- setTransformPivotX() method 247
- setTransformPivotY() method 247
- setVerticalBias() method 245
- sizing constraints 245
- tutorial 249
- view IDs 251
- ConstraintSet class 243, 244
- Constraint Sets 244
- ConstraintSets
 - configuring 378
- consumeAsync() method 761
- ConsumeParams 771
- Contacts permissions 632
- container view 171
- Content Provider 82, 559, 575
 - <provider> 561
 - accessing 575
 - Authority 565
 - client tutorial 575
 - ContentProvider class 559
 - Content Resolver 560
 - ContentResolver 572
 - content URI 560
 - Content URI 565, 575
 - ContentValues 567
 - delete() 560, 570
 - getType() 560
 - insert() 559, 567
 - onCreate() 559, 567
 - overview 85
 - query() 559, 568
 - tutorial 563
 - update() 560, 569
 - UriMatcher 566
 - UriMatcher class 560
- ContentProvider class 559
- Content Resolver 560
 - getContentResolver() 560
- ContentResolver 572
 - getContentResolver() 560
- content URI 560
- Content URI 560, 565
- ContentValues 567
- Context class 85
- CoordinatorLayout 172, 445
- Coroutine Builders 491
 - async 491
 - coroutineScope 491
 - launch 491
 - runBlocking 491
 - supervisorScope 491
 - withContext 491
- Coroutine Dispatchers 490
- Coroutines 489, 521
 - channel communication 495
 - GlobalScope 490
 - returning results 493
 - Suspend Functions 490
 - suspending 492
 - tutorial 497
 - ViewModelScope 490
 - vs. Threads 489
- coroutineScope 491
- Coroutine Scope 490
- createPrintDocumentAdapter() method 707
- Custom Accessors 127

- Custom Attribute 379
- Custom Document Printing 695, 707
- Custom Gesture
 - recognition 283
- Custom Print Adapter
 - implementation 709
- Custom Print Adapters 707
- Custom Theme
 - building 799
- Cycle Editor 407
- Cycle Keyframe 387
- Cycle Keyframes
 - overview 403

D

- dangerous permissions
 - list of 632
- Dark Theme 32
 - enable on device 32
- Data Access Object (DAO) 580
- Database Inspector 586, 610
 - live updates 610
 - SQL query 610
- Database Rows 546
- Database Schema 545
- Database Tables 545
- Data binding
 - binding expressions 331
- Data Binding 313
 - binding classes 330
 - enabling 336
 - event and listener binding 332
 - key components 327
 - overview 327
 - tutorial 335
 - variables 330
 - with LiveData 313
- DDMS 32
- Debugging
 - enabling on device 59
- debug.keystore file 455, 477
- Default Function Parameters 119

- DefaultLifecycleObserver 348, 351
- deltaRelative 384
- Density-independent pixels 239
- Density Independent Pixels
 - converting to pixels 254
- Device Definition
 - custom 189
- Device File Explorer 54
- device frame 36
- Device Mirroring 65
 - enabling 65
- device pairing 63
- Digital Asset Links file 722, 455, 455
- Direct Reply Input 669
- Dispatchers.Default 491
- Dispatchers.IO 490
- Dispatchers.Main 490
- document provider 775
- dp 239
- DROP_LATEST 533
- DROP_OLDEST 533
- Dynamic Colors
 - applyToActivitiesIfAvailable() method 805
 - enabling in Android 805
- Dynamic State 153
 - saving 167

E

- Elvis Operator 99
- Empty Process 147
- Empty template 175
- Emulator
 - battery 42
 - cellular configuration 42
 - configuring fingerprints 44
 - directional pad 42
 - extended control options 41
 - Extended controls 41
 - fingerprint 42
 - location configuration 42
 - phone settings 42
 - Resizable 46

Index

- resize 41
- rotate 40
- Screen Record 43
- Snapshots 43
- starting 29
- take screenshot 40
- toolbar 39
- toolbar options 39
- tool window mode 45
- Virtual Sensors 43
- zoom 40
- enablePendingPurchases() method 761
- enabling ADB support 59
- Escape Sequences 93
- Event Handling 265
 - example 266
- Event Listener 267
- Event Listeners 266
- Events
 - consuming 269
- execSQL() 554
- explicit
 - intent 84
- explicit intent 451
- Explicit Intent 451
- Extended Control
 - options 41
- F**
- Files
 - switching between 68
- filter() operator 526
- findPointerIndex() method 272
- findViewById() 139
- Fingerprint
 - emulation 44
- Fingerprint authentication
 - device configuration 736
 - permission 736
 - steps to implement 735
- Fingerprint Authentication
 - overview 735
 - tutorial 735
- FLAG_INCLUDE_STOPPED_PACKAGES 481
- flatMapConcat() operator 529
- flatMapMerge() operator 529
- flexible space area 443
- Float 92
- floating action button 14, 176
 - changing appearance of 418
 - margins 416
 - removing 177
 - sizes 416
- Flow 521
 - asFlow() builder 523
 - asSharedFlow() 532
 - asStateFlow() 531
 - background handling 541
 - buffering 525
 - buffer() operator 525
 - cold 531
 - collect() 523
 - collecting data 523
 - collectLatest() operator 524
 - combine() operator 530
 - conflate() operator 525
 - declaring 522
 - emit() 523
 - emitting data 523
 - filter() operator 526
 - flatMapConcat() operator 529
 - flatMapMerge() operator 529
 - flattening 528
 - flowOf() builder 523
 - flow of flows 528
 - fold() operator 528
 - hot 531
 - intermediate operators 526
 - library requirements 522
 - map() operator 526
 - MutableSharedFlow 532
 - MutableStateFlow 531
 - onEach() operator 530
 - reduce() operator 528

- repeatOnLifecycle 542
 - SharedFlow 532
 - single() operator 525
 - StateFlow 531
 - terminal flow operators 528
 - transform() operator 527
 - try/finally 524
 - zip() operator 530
 - flowOf() builder 523
 - flow of flows 528
 - Flow operators 526
 - Flows
 - combining 530
 - Introduction to 521
 - Foldable Devices 156
 - multi-resume 156
 - Foreground Process 146
 - Forward-geocoding 679
 - Fragment
 - creation 293
 - event handling 297
 - XML file 294
 - FragmentActivity class 152
 - Fragment Communication 297
 - Fragments 293
 - adding in code 296
 - duplicating 424
 - example 301
 - overview 293
 - FragmentStateAdapter class 427
 - FrameLayout 172
 - Function Parameters
 - variable number of 119
 - Functions 117
- G**
- Geocoder object 680
 - Geocoding 678
 - Gesture Builder Application 283
 - building and running 283
 - Gesture Detector class 277
 - GestureDetectorCompat
 - instance creation 280
 - GestureDetectorCompat class 277
 - GestureDetector.OnDoubleTapListener 277, 278
 - GestureDetector.OnGestureListener 278
 - GestureLibrary 283
 - GestureOverlayView 283
 - configuring color 288
 - configuring multiple strokes 288
 - GestureOverlayView class 283
 - GesturePerformedListener 283
 - Gestures
 - interception of 288
 - Gestures File
 - creation 284
 - extract from SD card 284
 - loading into application 286
 - GET_ACCOUNTS permission 632
 - getAction() method 487
 - getContentResolver() 560
 - getDebugMessage() 774
 - getFromLocation() method 680
 - getId() method 244
 - getIntent() method 452
 - getPointerCount() method 272
 - getPointerId() method 272
 - getPurchaseState() method 760
 - getService() method 511
 - getWritableDatabase() 554
 - GlobalScope 490
 - GNU/Linux 80
 - Google Cloud
 - billing account 674
 - new project 675
 - Google Cloud Print 690
 - Google Drive 776
 - printing to 690
 - GoogleMap 673
 - map types 683
 - GoogleMap.MAP_TYPE_HYBRID 683
 - GoogleMap.MAP_TYPE_NONE 683
 - GoogleMap.MAP_TYPE_NORMAL 683
 - GoogleMap.MAP_TYPE_SATELLITE 683

Index

GoogleMap.MAP_TYPE_TERRAIN 683

Google Maps Android API 673

Controlling the Map Camera 686

displaying controls 684

Map Markers 685

overview 673

Google Maps SDK 673

API Key 677

Credentials 677

enabling 676

Maps SDK for Android 677

Google Play App Signing 746

Google Play Console 765

Creating an in-app product 765

License Testers 766

Google Play Developer Console 744

Gradle

APK signing settings 814

Build Variants 810

command line tasks 815

dependencies 809

Manifest Entries 810

overview 809

sensible defaults 809

Gradle Build File

top level 811

Gradle Build Files

module level 812

gradle.properties file 810

GridLayout 172

GridLayoutManager 431

H

HAL 80

Handler class 516

Hardware Abstraction Layer 80

Higher-order Functions 121

Hot flows 531

HP Print Services Plugin 689

HTML printing 693

HTML Printing

example 697

I

IBinder 503, 509

IBinder object 507, 516

Image Printing 692

Immutable Variables 94

implicit

intent 84

implicit intent 451

Implicit Intent 453

Implicit Intents

example 469

importance hierarchy 145

in 239

INAPP 762

In-App Products 757

In-App Purchasing 763

acknowledgePurchase() method 761

BillingClient 758

BillingResult 774

consumeAsync() method 761

ConsumeParams 771

Consuming purchases 771

enablePendingPurchases() method 761

getPurchaseState() method 760

launchBillingFlow() method 760

Libraries 763

newBuilder() method 758

onBillingServiceDisconnected() callback 768

onBillingServiceDisconnected() method 759

onBillingSetupFinished() listener 768

onProductDetailsResponse() callback 768

Overview 757

ProductDetail 760

ProductDetails 769

products 757

ProductType 762

Purchase Flow 769

PurchaseResponseListener 762

PurchasesUpdatedListener 760

PurchaseUpdatedListener 770

purchase updates 770

queryProductDetailsAsync() 768

- queryProductDetailsAsync() method 759
 - queryPurchasesAsync() 772
 - queryPurchasesAsync() method 762
 - runOnUiThread() 769
 - subscriptions 757
 - tutorial 763
 - Initializer Blocks 127
 - In-Memory Database 586
 - Inner Classes 128
 - IntelliJ IDEA 87
 - Intent 84
 - explicit 84
 - implicit 84
 - Intent Availability
 - checking for 458
 - Intent.CATEGORY_OPENABLE 784
 - Intent Filters 454
 - App Link 721
 - Intents 451
 - ActivityResultLauncher 453
 - overview 451
 - registerForActivityResult() 453, 466
 - Intent Service 503
 - Intent URL 471
 - intermediate flow operators 526
 - is 99
 - isInitialized property 99
- J**
- Java
 - convert to Kotlin 87
 - Java Native Interface 81
 - JetBrains 87
 - Jetpack 311
 - overview 311
 - JobIntentService 503
 - BIND_JOB_SERVICE permission 505
 - onHandleWork() method 503
 - join() 491
- K**
- KeyAttribute 382
 - Keyboard Shortcuts 56
 - KeyCycle 403
 - Cycle Editor 407
 - tutorial 403
 - Keyframe 396
 - Keyframes 382
 - KeyFrameSet 412
 - KeyPosition 383
 - deltaRelative 384
 - parentRelative 383
 - pathRelative 384
 - Keystore File
 - creation 746
 - KeyTimeCycle 403
 - keytool 455
 - KeyTrigger 386
 - Killed state 148
 - Kotlin
 - accessing class properties 127
 - and Java 87
 - arithmetic operators 101
 - assignment operator 101
 - augmented assignment operators 102
 - bitwise operators 104
 - Boolean 92
 - break 112
 - breaking from loops 111
 - calling class methods 127
 - Char 92
 - class declaration 123
 - class initialization 124
 - class properties 124
 - Companion Objects 129
 - conditional control flow 113
 - continue labels 112
 - continue statement 112
 - control flow 109
 - convert from Java 87
 - Custom Accessors 127
 - data types 91
 - decrement operator 102
 - Default Function Parameters 119

Index

- defining class methods 124
- do ... while loop 111
- Elvis Operator 99
- equality operators 103
- Escape Sequences 93
- expression syntax 101
- Float 92
- Flow 521
- for-in statement 109
- function calling 118
- Functions 117
- Higher-order Functions 121
- if ... else ... expressions 114
- if expressions 113
- Immutable Variables 94
- increment operator 102
- inheritance 133
- Initializer Blocks 127
- Inner Classes 128
- introduction 87
- Lambda Expressions 120
- let Function 97
- Local Functions 118
- logical operators 103
- looping 109
- Mutable Variables 94
- Not-Null Assertion 97
- Nullable Type 96
- Overriding inherited methods 136
- playground 88
- Primary Constructor 124
- properties 127
- range operator 104
- Safe Call Operator 96
- Secondary Constructors 124
- Single Expression Functions 118
- String 92
- subclassing 133
- Type Annotations 95
- Type Casting 99
- Type Checking 99
- Type Inference 95

- variable parameters 119
- when statement 114
- while loop 110

L

- Lambda Expressions 120
- lateinit 98
- Late Initialization 98
- launch 491
- launchBillingFlow() method 760
- layout_collapseMode
 - parallax 448
 - pin 448
- layout_constraintDimensionRatio 228
- layout_constraintHorizontal_bias 226
- layout_constraintVertical_bias 226
- layout editor
 - ConstraintLayout chains 223
- Layout Editor 16, 231
 - Autoconnect Mode 205
 - code mode 182
 - Component Tree 179
 - design mode 179
 - device screen 179
 - example project 231
 - Inference Mode 205
 - palette 179
 - properties panel 180
 - Sample Data 188
 - Setting Properties 183
 - toolbar 180
 - user interface design 231
 - view conversion 187
- Layout Editor Tool
 - changing orientation 17
 - overview 179
- Layout Inspector 54
- Layout Managers 171
- LayoutResultCallback object 713
- Layouts 171
- layout_scrollFlags
 - enterAlwaysCollapsed mode 445

- enterAlways mode 445
 - exitUntilCollapsed mode 445
 - scroll mode 445
 - Layout Validation 190
 - let Function 97
 - libc 81
 - libs.versions.toml file 262
 - License Testers 766
 - Lifecycle
 - awareness 347
 - components 314
 - observers 348
 - owners 347
 - states and events 348
 - tutorial 351
 - Lifecycle-Aware Components 347
 - Lifecycle library 522
 - Lifecycle Methods 153
 - Lifecycle Observer 351
 - creating a 351
 - Lifecycle Owner
 - creating a 353
 - Lifecycles
 - modern 314
 - Lifecycle.State.CREATED 543
 - Lifecycle.State.DESTROYED 543
 - Lifecycle.State.INITIALIZED 543
 - Lifecycle.State.RESUMED 543
 - Lifecycle.State.STARTED 543
 - LinearLayout 172
 - LinearLayoutManager 431
 - LinearLayoutManager layout 439
 - Linux Kernel 80
 - list devices 59
 - LiveData 312, 323
 - adding to ViewModel 323
 - observer 325
 - tutorial 323
 - Live Templates 76
 - Local Bound Service 507
 - example 507
 - Local Functions 118
 - Location Manager 82
 - Location permission 632
 - Logcat
 - tool window 54
 - LogCat
 - enabling 163
- ## M
- MANAGE_EXTERNAL_STORAGE 633
 - adb enabling 633
 - testing 633
 - Manifest File
 - permissions 473
 - map() operator 526
 - Maps 673
 - MapView 673
 - adding to a layout 680
 - Marker class 673
 - Master/Detail Flow
 - creation 790
 - two pane mode 789
 - match_parent properties 239
 - Material design 415
 - Material Design 2 797
 - Material Design 2 Theming 797
 - Material Design 3 797
 - Material Theme Builder 799
 - Material You 797
 - measureTimeMillis() function 525
 - MediaController
 - adding to VideoView instance 617
 - MediaController class 614
 - methods 614
 - MediaPlayer class 639
 - methods 639
 - MediaRecorder class 639
 - methods 640
 - recording audio 640
 - Memory Indicator 69
 - Messenger object 516
 - Microphone
 - checking for availability 642

Index

Microphone permissions 632
mm 239
MotionEvent 271, 272, 291
 getActionMasked() 272
MotionLayout 377
 arc motion 382
 Attribute Keyframes 382
 ConstraintSets 378
 Custom Attribute 398
 Custom Attributes 379
 Cycle Editor 407
 Editor 389
 KeyAttribute 382
 KeyCycle 403
 Keyframes 382
 KeyFrameSet 412
 KeyPosition 383
 KeyTimeCycle 403
 KeyTrigger 386
 OnClick 381, 394
 OnSwipe 381
 overview 377
 Position Keyframes 383
 previewing animation 394
 Trigger Keyframe 386
 Tutorial 389
MotionScene
 ConstraintSets 378
 Custom Attributes 379
 file 378
 overview 377
 transition 378
moveCamera() method 686
multiple devices
 testing app on 31
Multiple Touches
 handling 272
multi-resume 156
Multi-Touch
 example 273
Multi-touch Event Handling 271
multi-window support 156

MutableSharedFlow 532
MutableStateFlow 531
Mutable Variables 94
My Location Layer 673

N

Navigation 357
 adding destinations 366
 overview 357
 pass data with safeargs 373
 passing arguments 362
 stack 357
 tutorial 363
Navigation Action
 triggering 361
Navigation Architecture Component 357
Navigation Component
 tutorial 363
Navigation Controller
 accessing 361
Navigation Graph 360, 364
 adding actions 370
 creating a 364
Navigation Host 358
 declaring 365
newBuilder() method 758
normal permissions 631
Notification
 adding actions 660
 Direct Reply Input 669
 issuing a basic 656
 launch activity from a 658
 PendingIntent 666
 Reply Action 668
 updating direct reply 670
Notifications
 bundled 660
 overview 649
Notifications Manager 82
Not-Null Assertion 97
Nullable Type 96

O

Observer
 implementing a LiveData 325
 onAttach() method 298
 onBillingServiceDisconnected() callback 768
 onBillingServiceDisconnected() method 759
 onBillingSetupFinished() listener 768
 onBind() method 504, 507, 515
 onBindViewHolder() method 439
 OnClick 381
 onClickListener 266, 267, 270
 onClick() method 265
 onCreateContextMenuListener 266
 onCreate() method 146, 153, 504
 onCreateView() method 154
 onDestroy() method 154, 504
 onDoubleTap() method 277
 onDown() method 277
 onEach() operator 530
 onFling() method 277
 onFocusChangeListener 266
 OnFragmentInteractionListener
 implementation 371
 onGesturePerformed() method 283
 onHandleWork() method 504
 onKeyListener 266
 onLayoutFailed() method 713
 onLayoutFinished() method 713
 onLongClickListener 266
 onLongPress() method 277
 onMapReady() method 682
 onPageFinished() callback 698
 onPause() method 154
 onProductDetailsResponse() callback 768
 onReceive() method 146, 482, 483, 485
 onRequestPermissionsResult() method 635, 646, 654, 664
 onRestart() method 153
 onRestoreInstanceState() method 154
 onResume() method 146, 154
 onSaveInstanceState() method 154
 onScaleBegin() method 289
 onScaleEnd() method 289

onScale() method 289
 onScroll() method 277
 OnSeekBarChangeListener 308
 onServiceConnected() method 507, 510, 517
 onServiceDisconnected() method 507, 510, 517
 onShowPress() method 277
 onSingleTapUp() method 277
 onStartCommand() method 504
 onStart() method 154
 onStop() method 154
 onTouchEvent() method 277, 289
 onTouchListener 266
 onTouch() method 272
 onUpgrade() 554
 onViewCreated() method 154
 onViewStatusRestored() method 154
 openFileDescriptor() method 776
 OpenJDK 3

P

Package Explorer 15
 Package Manager 82
 PackageManager class 642
 PackageManager.FEATURE_MICROPHONE 642
 PackageManager.PERMISSION_DENIED 633
 PackageManager.PERMISSION_GRANTED 633
 Package Name 14
 Packed chain 197, 226
 PageRange 714, 715
 Paint class 717
 parentRelative 383
 parent view 173
 pathRelative 384
 Paused state 148
 PdfDocument 695
 PdfDocument.Page 707, 714
 PendingIntent class 666
 Permission
 checking for 633
 permissions
 normal 631
 Persistent State 153

Index

Phone permissions 632
picker 775
Pinch Gesture
 detection 289
 example 289
Pinch Gesture Recognition 283
Position Keyframes 383
POST_NOTIFICATIONS permission 632, 664
Primary Constructor 124
PrintAttributes 712
PrintDocumentAdapter 695, 707
Printing
 color 692
 monochrome 692
Printing framework
 architecture 689
Printing Framework 689
Print Job
 starting 718
PrintManager service 699
Problems
 tool window 54, 55
process
 priority 145
 state 145
PROCESS_OUTGOING_CALLS permission 632
Process States 145
ProductDetail 760
ProductDetails 769
ProductType 762
Profiler
 tool window 55
ProgressBar 171
proguard-rules.pro file 814
ProGuard Support 810
Project Name 14
Project tool window 15, 53
pt 239
PurchaseResponseListener 762
PurchasesUpdatedListener 760
PurchaseUpdatedListener 770
putExtra() method 451, 481

px 240

Q

queryProductDetailsAsync() 768
queryPurchasesAsync() 772
quickboot snapshot 44
Quick Documentation 75

R

RadioButton 171
Range Operator 104
ratios 227
READ_CALENDAR permission 632
READ_CALL_LOG permission 632
READ_CONTACTS permission 632
READ_EXTERNAL_STORAGE permission 633
READ_PHONE_STATE permission 632
READ_SMS permission 632
RECEIVE_MMS permission 632
RECEIVE_SMS permission 632
RECEIVE_WAP_PUSH permission 632
Recent Files Navigation 56
RECORD_AUDIO permission 632
Recording Audio
 permission 641
RecyclerView 431
 adding to layout file 432
 LayoutManager 431
 initializing 439
 LayoutManager 431
 StaggeredLayoutManager 431
RecyclerView Adapter
 creation of 437
RecyclerView.Adapter 432, 438
 getItemCount() method 432
 onBindViewHolder() method 432
 onCreateViewHolder() method 432
RecyclerView.ViewHolder
 getAdapterPosition() method 442
reduce() operator 528
registerForActivityResult() 453
registerForActivityResult() method 452, 466

- registerReceiver() method 483
- RelativeLayout 172
- releasePersistableUriPermission() method 779
- Release Preparation 743
- Remote Bound Service 515
 - client communication 515
 - implementation 515
 - manifest file declaration 517
- RemoteInput.Builder() method 666
- RemoteInput Object 666
- Remote Service
 - launching and binding 517
 - sending a message 519
- repeatOnLifecycle 542
- Repository
 - tutorial 597
- Repository Modules 314
- Resizable Emulator 46
- Resource
 - string creation 20
- Resource File 22
- Resource Management 145
- Resource Manager 53, 82
- result receiver 483
- Reverse-geocoding 679
- Reverse Geocoding 678
- Room
 - Data Access Object (DAO) 580
 - entities 580, 581
 - In-Memory Database 586
 - Repository 580
- Room Database 580
 - tutorial 597
- Room Database Persistence 579
- Room Persistence Library 550, 579
- root element 171
- root view 173
- Run
 - tool window 53
- runBlocking 491
- Running Devices
 - tool window 65

- runOnUiThread() 769

S

- safeargs 373
- Safe Call Operator 96
- Sample Data 188
- Saved State 313, 343
- SavedStateHandle 344
 - contains() method 345
 - keys() method 345
 - remove() method 345
- Saved State module 343
- SavedStateViewModelFactory 344
- ScaleGestureDetector class 289
- Scale-independent 239
- SDK Packages 5
- Secondary Constructors 124
- Secure Sockets Layer (SSL) 81
- SeekBar 301
- sendBroadcast() method 481, 483
- sendOrderedBroadcast() method 481, 483
- SEND_SMS permission 632
- sendStickyBroadcast() method 481
- Sensor permissions 632
- Service
 - anatomy 504
 - launch at system start 505
 - manifest file entry 504
 - overview 84
 - run in separate process 505
- ServiceConnection class 517
- Service Process 146
- Service Restart Options 504
- setAudioEncoder() method 640
- setAudioSource() method 640
- setBackgroundColor() 244
- setCompassEnabled() method 684
- setContentView() method 243, 249
- setId() method 244
- setMyLocationButtonEnabled() method 684
- setOnClickListener() method 265, 267
- setOnDoubleTapListener() method 277, 280

Index

- setOutputFile() method 640
- setOutputFormat() method 640
- setResult() method 453
- setText() method 170
- settings.gradle file 810
- settings.gradle.kts file 810
- setTransition() 387
- setVideoSource() method 640
- SHA-256 certificate fingerprint 455
- SharedFlow 532, 535
 - backgroundn handling 541
 - DROP_LATEST 533
 - DROP_OLDEST 533
 - in ViewModel 537
 - repeatOnLifecycle 542
 - SUSPEND 533
 - tutorial 535
- shouldOverrideUrlLoading() method 698
- SimpleOnScaleGestureListener 289
- SimpleOnScaleGestureListener class 290
- single() operator 525
- SMS permissions 632
- Snackbar 415, 416, 417
- Snapshots
 - emulator 43
- sp 239
- Spread chain 196
- Spread inside 226
- Spread inside chain 196
- SQL 546
- SQL CREATE 554
- SQLite 545
 - AVD command-line use 547
 - Columns and Data Types 545
 - overview 546
 - Primary keys 546
 - tutorial 551
- SQLiteDatabase 554
- SQLiteOpenHelper 552
- SQL SELECT 555
- StaggeredGridLayoutManager 431
- startActivity() method 451
- startForeground() method 146
- START_NOT_STICKY 504
- START_REDELIVER_INTENT 504
- START_STICKY 504
- State
 - restoring 170
- State Change
 - handling 149
- StateFlow 531
- Statement Completion 72
- Status Bar Widgets 69
 - Memory Indicator 69
- Sticky Broadcast Intents 483
- Stopped state 148
- Storage Access Framework 775
 - ACTION_CREATE_DOCUMENT 776
 - ACTION_OPEN_DOCUMENT 776
 - deleting a file 779
 - example 781
 - file creation 783
 - file filtering 776
 - file reading 777
 - file writing 778
 - intents 776
 - MIME Types 777
 - Persistent Access 779
 - picker 775
- Storage permissions 633
- String 92
- StringBuilder object 786
- strings.xml file 24
- Structure
 - tool window 55
- Structured Query Language 546
- Structure tool window 55
- SUBS 762
- subscriptions 757
- supervisorScope 491
- SupportMapFragment class 673
- SUSPEND 533
- Suspend Functions 490
- Switcher 56

System Broadcasts 487

system requirements 3

T

TabLayout

adding to layout 425

app

tabGravity property 430

tabMode property 430

example 422

fixed mode 429

getItemCount() method 421

overview 421

TableLayout 172, 589

TableRow 589

Telephony Manager 82

Templates

blank vs. empty 175

Terminal

tool window 54

terminal flow operators 528

Theme

building a custom 799

Theming 797

tutorial 801

Time Cycle Keyframes 387

TODO

tool window 55

ToolBarListener 298

tools

layout 295

Tool window bars 52

Tool windows 52

Touch Actions 272

Touch Event Listener

implementation 273

Touch Events

intercepting 271

Touch handling 271

transform() operator 527

try/finally 524

Type Annotations 95

Type Casting 99

Type Checking 99

Type Inference 95

U

UiSettings class 673

unbindService() method 503

unregisterReceiver() method 483

upload key 746

UriMatcher 560, 566

UriMatcher class 560

URL Mapping 727

USB connection issues

resolving 62

USE_BIOMETRIC 736

user interface state 153

USE_SIP permission 632

V

Version catalog 261

dependencies 263

libraries 263

libs.versions.toml file 262

plugins 263

versions 263

Video Playback 613

VideoView class 613

methods 613

supported formats 613

view bindings

enabling 140

using 140

View class

setting properties 250

view conversion 187

ViewGroup 171

View Groups 171

View Hierarchy 173

ViewHolder class 432

sample implementation 438

ViewModel

adding LiveData 323

Index

- data access 321
- overview 312
- saved state 343
- Saved State 313, 343
- tutorial 317
- ViewModelProvider 320
- ViewModel Saved State 343
- ViewModelScope 490
- ViewPager
 - adding to layout 425
 - example 422
- Views 171
 - Java creation 243
- View System 82
- Virtual Device Configuration dialog 28
- Virtual Sensors 43
- Visible Process 146

W

- WebViewClient 693, 698
- WebView view 471
- Weighted chain 196, 226
- Welcome screen 49
- while Loop 110
- Widget Dimensions 197
- Widget Group Alignment 219
- Widgets palette 232
- WiFi debugging 63
- Wireless debugging 63
- Wireless pairing 63
- withContext 491, 493
- wrap_content properties 241
- WRITE_CALENDAR permission 632
- WRITE_CALL_LOG permission 632
- WRITE_CONTACTS permission 632
- WRITE_EXTERNAL_STORAGE permission 633

X

- XML Layout File
 - manual creation 239
 - vs. Java Code 243

Z

- zip() operator 530