Android Studio Jellyfish Essentials

Java Edition

Neil Smyth

Payload publishing

Android Studio Jellyfish Essentials

Java Edition

Android Studio Jellyfish Essentials – Java Edition

ISBN: 978-1-951442-86-6

© 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.

Contents

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 ... 73
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 Overview of Android View Binding ... 87
11.1 Find View by Id ... 87
11.2 View Binding ... 87
11.3 Converting the AndroidSample project ... 88
11.4 Enabling View Binding ... 88
11.5 Using View Binding ... 88
11.6 Choosing an Option ... 90
11.7 View Binding in the Book Examples ... 90
11.8 Migrating a Project to View Binding ... 90
11.9 Summary ... 91

12. Understanding Android Application and Activity Lifecycles 93

12.1 Android Applications and Resource Management........ 93
12.2 Android Process States........ 93
12.2.1 Foreground Process........ 94
12.2.2 Visible Process........ 94
12.2.3 Service Process........ 94
12.2.4 Background Process........ 94
12.2.5 Empty Process........ 95
12.3 Inter-Process Dependencies........ 95
12.4 The Activity Lifecycle........ 95
12.5 The Activity Stack........ 95
12.6 Activity States........ 96
12.7 Configuration Changes........ 96
12.8 Handling State Change........ 97
12.9 Summary........ 97

13. Handling Android Activity State Changes........ 99

13.1 New vs. Old Lifecycle Techniques........ 99
13.2 The Activity and Fragment Classes........ 99
13.3 Dynamic State vs. Persistent State........ 101
13.4 The Android Lifecycle Methods........ 102
13.5 Lifetimes........ 103
13.6 Foldable Devices and Multi-Resume........ 104
13.7 Disabling Configuration Change Restarts........ 104
13.8 Lifecycle Method Limitations........ 105
13.9 Summary........ 105

14. Android Activity State Changes by Example........ 107

14.1 Creating the State Change Example Project........ 107
14.2 Designing the User Interface........ 108
14.3 Overriding the Activity Lifecycle Methods........ 109
14.4 Filtering the Logcat Panel........ 111
14.5 Running the Application........ 113
14.6 Experimenting with the Activity........ 113
14.7 Summary........ 114

15. Saving and Restoring the State of an Android Activity........ 115

15.1 Saving Dynamic State........ 115
15.2 Default Saving of User Interface State........ 115
15.3 The Bundle Class........ 116
15.4 Saving the State........ 117
15.5 Restoring the State........ 118
15.6 Testing the Application........ 118
15.7 Summary........ 118

16. Understanding Android Views, View Groups and Layouts........ 121

16.1 Designing for Different Android Devices........ 121
16.2 Views and View Groups........ 121
16.3 Android Layout Managers........ 121

16.4 The View Hierarchy 123
16.5 Creating User Interfaces 124
16.6 Summary 124

17. A Guide to the Android Studio Layout Editor Tool 125

17.1 Basic vs. Empty Views Activity Templates 125
17.2 The Android Studio Layout Editor 129
17.3 Design Mode 129
17.4 The Palette 130
17.5 Design Mode and Layout Views 131
17.6 Night Mode 132
17.7 Code Mode 132
17.8 Split Mode 133
17.9 Setting Attributes 133
17.10 Transforms 135
17.11 Tools Visibility Toggles 136
17.12 Converting Views 137
17.13 Displaying Sample Data 138
17.14 Creating a Custom Device Definition 139
17.15 Changing the Current Device 139
17.16 Layout Validation 140
17.17 Summary 141

18. A Guide to the Android ConstraintLayout 143

18.1 How ConstraintLayout Works 143
18.1.1 Constraints 143
18.1.2 Margins 144
18.1.3 Opposing Constraints 144
18.1.4 Constraint Bias 145
18.1.5 Chains 146
18.1.6 Chain Styles 146
18.2 Baseline Alignment 147
18.3 Configuring Widget Dimensions 147
18.4 Guideline Helper 148
18.5 Group Helper 148
18.6 Barrier Helper 148
18.7 Flow Helper 150
18.8 Ratios 151
18.9 ConstraintLayout Advantages 151
18.10 ConstraintLayout Availability 152
18.11 Summary 152

19. A Guide to Using ConstraintLayout in Android Studio 153

19.1 Design and Layout Views 153
19.2 Autoconnect Mode 155
19.3 Inference Mode 155
19.4 Manipulating Constraints Manually 155
19.5 Adding Constraints in the Inspector 157
19.6 Viewing Constraints in the Attributes Window 157

19.7 Deleting Constraints 158
19.8 Adjusting Constraint Bias 159
19.9 Understanding ConstraintLayout Margins 159
19.10 The Importance of Opposing Constraints and Bias 161
19.11 Configuring Widget Dimensions 163
19.12 Design Time Tools Positioning 164
19.13 Adding Guidelines 165
19.14 Adding Barriers 167
19.15 Adding a Group 168
19.16 Working with the Flow Helper 169
19.17 Widget Group Alignment and Distribution 169
19.18 Converting other Layouts to ConstraintLayout 171
19.19 Summary 171

20. Working with ConstraintLayout Chains and Ratios in Android Studio 173

20.1 Creating a Chain 173
20.2 Changing the Chain Style 175
20.3 Spread Inside Chain Style 176
20.4 Packed Chain Style 176
20.5 Packed Chain Style with Bias 176
20.6 Weighted Chain 176
20.7 Working with Ratios 177
20.8 Summary 179

21. An Android Studio Layout Editor ConstraintLayout Tutorial 181

21.1 An Android Studio Layout Editor Tool Example 181
21.2 Preparing the Layout Editor Environment 181
21.3 Adding the Widgets to the User Interface 182
21.4 Adding the Constraints 185
21.5 Testing the Layout 187
21.6 Using the Layout Inspector 187
21.7 Summary 188

22. Manual XML Layout Design in Android Studio 189

22.1 Manually Creating an XML Layout 189
22.2 Manual XML vs. Visual Layout Design 192
22.3 Summary 192

23. Managing Constraints using Constraint Sets 193

23.1 Java Code vs. XML Layout Files 193
23.2 Creating Views 193
23.3 View Attributes 194
23.4 Constraint Sets 194
23.4.1 Establishing Connections 194
23.4.2 Applying Constraints to a Layout 194
23.4.3 Parent Constraint Connections 194
23.4.4 Sizing Constraints 195
23.4.5 Constraint Bias 195
23.4.6 Alignment Constraints 195
23.4.7 Copying and Applying Constraint Sets 195

23.4.8 ConstraintLayout Chains ... 195
23.4.9 Guidelines ... 196
23.4.10 Removing Constraints ... 196
23.4.11 Scaling ... 196
23.4.12 Rotation ... 197
23.5 Summary ... 197

24. An Android ConstraintSet Tutorial ... 199

24.1 Creating the Example Project in Android Studio ... 199
24.2 Adding Views to an Activity ... 199
24.3 Setting View Attributes ... 201
24.4 Creating View IDs ... 201
24.5 Configuring the Constraint Set ... 202
24.6 Adding the EditText View ... 203
24.7 Converting Density Independent Pixels (dp) to Pixels (px) ... 205
24.8 Summary ... 206

25. A Guide to Using Apply Changes in Android Studio ... 207

25.1 Introducing Apply Changes ... 207
25.2 Understanding Apply Changes Options ... 207
25.3 Using Apply Changes ... 208
25.4 Configuring Apply Changes Fallback Settings ... 209
25.5 An Apply Changes Tutorial ... 209
25.6 Using Apply Code Changes ... 209
25.7 Using Apply Changes and Restart Activity ... 210
25.8 Using Run App ... 210
25.9 Summary ... 210

26. A Guide to Gradle Version Catalogs ... 211

26.1 Library and Plugin Dependencies ... 211
26.2 Project Gradle Build File ... 211
26.3 Module Gradle Build Files ... 211
26.4 Version Catalog File ... 212
26.5 Adding Dependencies ... 213
26.6 Library Updates ... 214
26.7 Summary ... 214

27. An Overview and Example of Android Event Handling ... 215

27.1 Understanding Android Events ... 215
27.2 Using the android:onClick Resource ... 215
27.3 Event Listeners and Callback Methods ... 216
27.4 An Event Handling Example ... 216
27.5 Designing the User Interface ... 217
27.6 The Event Listener and Callback Method ... 217
27.7 Consuming Events ... 219
27.8 Summary ... 220

28. Android Touch and Multi-touch Event Handling ... 221

28.1 Intercepting Touch Events ... 221
28.2 The MotionEvent Object ... 221

28.3 Understanding Touch Actions....222
28.4 Handling Multiple Touches....222
28.5 An Example Multi-Touch Application....222
28.6 Designing the Activity User Interface....223
28.7 Implementing the Touch Event Listener....223
28.8 Running the Example Application....226
28.9 Summary....227

29. Detecting Common Gestures Using the Android Gesture Detector Class....229

29.1 Implementing Common Gesture Detection....229
29.2 Creating an Example Gesture Detection Project....230
29.3 Implementing the Listener Class....230
29.4 Creating the GestureDetectorCompat Instance....232
29.5 Implementing the onTouchEvent() Method....233
29.6 Testing the Application....233
29.7 Summary....234

30. Implementing Custom Gesture and Pinch Recognition on Android....235

30.1 The Android Gesture Builder Application....235
30.2 The GestureOverlayView Class....235
30.3 Detecting Gestures....235
30.4 Identifying Specific Gestures....235
30.5 Installing and Running the Gesture Builder Application....235
30.6 Creating a Gestures File....236
30.7 Creating the Example Project....236
30.8 Extracting the Gestures File from the SD Card....236
30.9 Adding the Gestures File to the Project....237
30.10 Designing the User Interface....237
30.11 Loading the Gestures File....238
30.12 Registering the Event Listener....239
30.13 Implementing the onGesturePerformed Method....239
30.14 Testing the Application....240
30.15 Configuring the GestureOverlayView....240
30.16 Intercepting Gestures....241
30.17 Detecting Pinch Gestures....241
30.18 A Pinch Gesture Example Project....241
30.19 Summary....243

31. An Introduction to Android Fragments....245

31.1 What is a Fragment?....245
31.2 Creating a Fragment....245
31.3 Adding a Fragment to an Activity using the Layout XML File....246
31.4 Adding and Managing Fragments in Code....248
31.5 Handling Fragment Events....249
31.6 Implementing Fragment Communication....250
31.7 Summary....251

32. Using Fragments in Android Studio - An Example....253

32.1 About the Example Fragment Application....253
32.2 Creating the Example Project....253

32.3 Creating the First Fragment Layout ... 253
32.4 Migrating a Fragment to View Binding ... 255
32.5 Adding the Second Fragment ... 256
32.6 Adding the Fragments to the Activity ... 257
32.7 Making the Toolbar Fragment Talk to the Activity ... 258
32.8 Making the Activity Talk to the Text Fragment ... 261
32.9 Testing the Application ... 262
32.10 Summary ... 263

33. Modern Android App Architecture with Jetpack ... 265

33.1 What is Android Jetpack? ... 265
33.2 The "Old" Architecture ... 265
33.3 Modern Android Architecture ... 265
33.4 The ViewModel Component ... 266
33.5 The LiveData Component ... 266
33.6 ViewModel Saved State ... 267
33.7 LiveData and Data Binding ... 267
33.8 Android Lifecycles ... 268
33.9 Repository Modules ... 268
33.10 Summary ... 269

34. An Android ViewModel Tutorial ... 271

34.1 About the Project ... 271
34.2 Creating the ViewModel Example Project ... 271
34.3 Removing Unwanted Project Elements ... 271
34.4 Designing the Fragment Layout ... 272
34.5 Implementing the View Model ... 273
34.6 Associating the Fragment with the View Model ... 274
34.7 Modifying the Fragment ... 275
34.8 Accessing the ViewModel Data ... 276
34.9 Testing the Project ... 276
34.10 Summary ... 277

35. An Android Jetpack LiveData Tutorial ... 279

35.1 LiveData - A Recap ... 279
35.2 Adding LiveData to the ViewModel ... 279
35.3 Implementing the Observer ... 281
35.4 Summary ... 283

36. An Overview of Android Jetpack Data Binding ... 285

36.1 An Overview of Data Binding ... 285
36.2 The Key Components of Data Binding ... 285
36.2.1 The Project Build Configuration ... 285
36.2.2 The Data Binding Layout File ... 286
36.2.3 The Layout File Data Element ... 287
36.2.4 The Binding Classes ... 288
36.2.5 Data Binding Variable Configuration ... 288
36.2.6 Binding Expressions (One-Way) ... 289
36.2.7 Binding Expressions (Two-Way) ... 290
36.2.8 Event and Listener Bindings ... 290

36.3 Summary ... 291

37. An Android Jetpack Data Binding Tutorial ... 293

37.1 Removing the Redundant Code ... 293
37.2 Enabling Data Binding ... 294
37.3 Adding the Layout Element ... 295
37.4 Adding the Data Element to Layout File ... 296
37.5 Working with the Binding Class ... 296
37.6 Assigning the ViewModel Instance to the Data Binding Variable ... 297
37.7 Adding Binding Expressions ... 298
37.8 Adding the Conversion Method ... 299
37.9 Adding a Listener Binding ... 299
37.10 Testing the App ... 299
37.11 Summary ... 300

38. An Android ViewModel Saved State Tutorial ... 301

38.1 Understanding ViewModel State Saving ... 301
38.2 Implementing ViewModel State Saving ... 301
38.3 Saving and Restoring State ... 303
38.4 Adding Saved State Support to the ViewModelDemo Project ... 303
38.5 Summary ... 304

39. Working with Android Lifecycle-Aware Components ... 305

39.1 Lifecycle Awareness ... 305
39.2 Lifecycle Owners ... 305
39.3 Lifecycle Observers ... 306
39.4 Lifecycle States and Events ... 307
39.5 Summary ... 308

40. An Android Jetpack Lifecycle Awareness Tutorial ... 309

40.1 Creating the Example Lifecycle Project ... 309
40.2 Creating a Lifecycle Observer ... 309
40.3 Adding the Observer ... 311
40.4 Testing the Observer ... 311
40.5 Creating a Lifecycle Owner ... 311
40.6 Testing the Custom Lifecycle Owner ... 313
40.7 Summary ... 313

41. An Overview of the Navigation Architecture Component ... 315

41.1 Understanding Navigation ... 315
41.2 Declaring a Navigation Host ... 316
41.3 The Navigation Graph ... 318
41.4 Accessing the Navigation Controller ... 319
41.5 Triggering a Navigation Action ... 319
41.6 Passing Arguments ... 320
41.7 Summary ... 320

42. An Android Jetpack Navigation Component Tutorial ... 321

42.1 Creating the NavigationDemo Project ... 321
42.2 Adding Navigation to the Build Configuration ... 321

42.3 Creating the Navigation Graph Resource File 322
42.4 Declaring a Navigation Host 323
42.5 Adding Navigation Destinations 324
42.6 Designing the Destination Fragment Layouts 326
42.7 Adding an Action to the Navigation Graph 328
42.8 Implement the OnFragmentInteractionListener 329
42.9 Adding View Binding Support to the Destination Fragments 330
42.10 Triggering the Action 331
42.11 Passing Data Using Safeargs 332
42.12 Summary 335

43. An Introduction to MotionLayout 337

43.1 An Overview of MotionLayout 337
43.2 MotionLayout 337
43.3 MotionScene 337
43.4 Configuring ConstraintSets 338
43.5 Custom Attributes 339
43.6 Triggering an Animation 341
43.7 Arc Motion 342
43.8 Keyframes 342
43.8.1 Attribute Keyframes 342
43.8.2 Position Keyframes 343
43.9 Time Linearity 346
43.10 KeyTrigger 346
43.11 Cycle and Time Cycle Keyframes 347
43.12 Starting an Animation from Code 347
43.13 Summary 348

44. An Android MotionLayout Editor Tutorial 349

44.1 Creating the MotionLayoutDemo Project 349
44.2 ConstraintLayout to MotionLayout Conversion 349
44.3 Configuring Start and End Constraints 351
44.4 Previewing the MotionLayout Animation 354
44.5 Adding an OnClick Gesture 354
44.6 Adding an Attribute Keyframe to the Transition 356
44.7 Adding a CustomAttribute to a Transition 358
44.8 Adding Position Keyframes 360
44.9 Summary 362

45. A MotionLayout KeyCycle Tutorial 363

45.1 An Overview of Cycle Keyframes 363
45.2 Using the Cycle Editor 367
45.3 Creating the KeyCycleDemo Project 368
45.4 Configuring the Start and End Constraints 368
45.5 Creating the Cycles 370
45.6 Previewing the Animation 372
45.7 Adding the KeyFrameSet to the MotionScene 372
45.8 Summary 374

46. Working with the Floating Action Button and Snackbar 375

46.1 The Material Design ... 375
46.2 The Design Library ... 375
46.3 The Floating Action Button (FAB) ... 375
46.4 The Snackbar ... 376
46.5 Creating the Example Project ... 377
46.6 Reviewing the Project ... 377
46.7 Removing Navigation Features ... 378
46.8 Changing the Floating Action Button ... 378
46.9 Adding an Action to the Snackbar ... 380
46.10 Summary ... 380

47. Creating a Tabbed Interface using the TabLayout Component ... 381

47.1 An Introduction to the ViewPager2 ... 381
47.2 An Overview of the TabLayout Component ... 381
47.3 Creating the TabLayoutDemo Project ... 382
47.4 Creating the First Fragment ... 383
47.5 Duplicating the Fragments ... 384
47.6 Adding the TabLayout and ViewPager2 ... 385
47.7 Performing the Initialization Tasks ... 387
47.8 Testing the Application ... 389
47.9 Customizing the TabLayout ... 389
47.10 Summary ... 391

48. Working with the RecyclerView and CardView Widgets ... 393

48.1 An Overview of the RecyclerView ... 393
48.2 An Overview of the CardView ... 395
48.3 Summary ... 396

49. An Android RecyclerView and CardView Tutorial ... 397

49.1 Creating the CardDemo Project ... 397
49.2 Modifying the Basic Views Activity Project ... 397
49.3 Designing the CardView Layout ... 398
49.4 Adding the RecyclerView ... 399
49.5 Adding the Image Files ... 399
49.6 Creating the RecyclerView Adapter ... 400
49.7 Initializing the RecyclerView Component ... 402
49.8 Testing the Application ... 403
49.9 Responding to Card Selections ... 403
49.10 Summary ... 405

50. A Layout Editor Sample Data Tutorial ... 407

50.1 Adding Sample Data to a Project ... 407
50.2 Using Custom Sample Data ... 411
50.3 Summary ... 414

51. Working with the AppBar and Collapsing Toolbar Layouts ... 415

51.1 The Anatomy of an AppBar ... 415
51.2 The Example Project ... 416
51.3 Coordinating the RecyclerView and Toolbar ... 416
51.4 Introducing the Collapsing Toolbar Layout ... 418

51.5 Changing the Title and Scrim Color 421
51.6 Summary 422

52. An Android Studio Primary/Detail Flow Tutorial 423

52.1 The Primary/Detail Flow 423
52.2 Creating a Primary/Detail Flow Activity 424
52.3 Adding the Primary/Detail Flow Activity 424
52.4 Modifying the Primary/Detail Flow Template 425
52.5 Changing the Content Model 425
52.6 Changing the Detail Pane 427
52.7 Modifying the ItemDetailFragment Class 428
52.8 Modifying the ItemListFragment Class 429
52.9 Adding Manifest Permissions 430
52.10 Running the Application 430
52.11 Summary 430

53. An Overview of Android Services 431

53.1 Intent Service 431
53.2 Bound Service 431
53.3 The Anatomy of a Service 432
53.4 Controlling Destroyed Service Restart Options 432
53.5 Declaring a Service in the Manifest File 432
53.6 Starting a Service Running on System Startup 433
53.7 Summary 434

54. An Overview of Android Intents 435

54.1 An Overview of Intents 435
54.2 Explicit Intents 435
54.3 Returning Data from an Activity 436
54.4 Implicit Intents 437
54.5 Using Intent Filters 438
54.6 Automatic Link Verification 438
54.7 Manually Enabling Links 441
54.8 Checking Intent Availability 442
54.9 Summary 443

55. Android Explicit Intents – A Worked Example 445

55.1 Creating the Explicit Intent Example Application 445
55.2 Designing the User Interface Layout for MainActivity 445
55.3 Creating the Second Activity Class 446
55.4 Designing the User Interface Layout for SecondActivity 447
55.5 Reviewing the Application Manifest File 447
55.6 Creating the Intent 448
55.7 Extracting Intent Data 449
55.8 Launching SecondActivity as a Sub-Activity 450
55.9 Returning Data from a Sub-Activity 451
55.10 Testing the Application 451
55.11 Summary 452

56. Android Implicit Intents – A Worked Example 453

56.1 Creating the Android Studio Implicit Intent Example Project 453
56.2 Designing the User Interface 453
56.3 Creating the Implicit Intent 454
56.4 Adding a Second Matching Activity 454
56.5 Adding the Web View to the UI 455
56.6 Obtaining the Intent URL 455
56.7 Modifying the MyWebView Project Manifest File 457
56.8 Installing the MyWebView Package on a Device 458
56.9 Testing the Application 459
56.10 Manually Enabling the Link 459
56.11 Automatic Link Verification 461
56.12 Summary 463

57. Android Broadcast Intents and Broadcast Receivers 465

57.1 An Overview of Broadcast Intents 465
57.2 An Overview of Broadcast Receivers 466
57.3 Obtaining Results from a Broadcast 467
57.4 Sticky Broadcast Intents 467
57.5 The Broadcast Intent Example 468
57.6 Creating the Example Application 468
57.7 Creating and Sending the Broadcast Intent 468
57.8 Creating the Broadcast Receiver 469
57.9 Registering the Broadcast Receiver 470
57.10 Testing the Broadcast Example 471
57.11 Listening for System Broadcasts 471
57.12 Summary 472

58. Android Local Bound Services – A Worked Example 473

58.1 Understanding Bound Services 473
58.2 Bound Service Interaction Options 473
58.3 A Local Bound Service Example 473
58.4 Adding a Bound Service to the Project 474
58.5 Implementing the Binder 474
58.6 Binding the Client to the Service 477
58.7 Completing the Example 478
58.8 Testing the Application 479
58.9 Summary 479

59. Android Remote Bound Services – A Worked Example 481

59.1 Client to Remote Service Communication 481
59.2 Creating the Example Application 481
59.3 Designing the User Interface 481
59.4 Implementing the Remote Bound Service 481
59.5 Configuring a Remote Service in the Manifest File 483
59.6 Launching and Binding to the Remote Service 484
59.7 Sending a Message to the Remote Service 485
59.8 Summary 486

60. An Overview of Java Threads, Handlers and Executors 487

60.1 The Application Main Thread 487

60.2 Thread Handlers ... 487
60.3 A Threading Example ... 487
60.4 Building the App ... 488
60.5 Creating a New Thread ... 489
60.6 Implementing a Thread Handler ... 490
60.7 Passing a Message to the Handler ... 492
60.8 Java Executor Concurrency ... 492
60.9 Working with Runnable Tasks ... 493
60.10 Shutting down an Executor Service ... 494
60.11 Working with Callable Tasks and Futures ... 494
60.12 Handling a Future Result ... 496
60.13 Scheduling Tasks ... 497
60.14 Summary ... 498

61. Making Runtime Permission Requests in Android ... 499

61.1 Understanding Normal and Dangerous Permissions ... 499
61.2 Creating the Permissions Example Project ... 501
61.3 Checking for a Permission ... 501
61.4 Requesting Permission at Runtime ... 503
61.5 Providing a Rationale for the Permission Request ... 504
61.6 Testing the Permissions App ... 506
61.7 Summary ... 506

62. An Android Notifications Tutorial ... 507

62.1 An Overview of Notifications ... 507
62.2 Creating the NotifyDemo Project ... 509
62.3 Designing the User Interface ... 509
62.4 Creating the Second Activity ... 509
62.5 Creating a Notification Channel ... 510
62.6 Requesting Notification Permission ... 511
62.7 Creating and Issuing a Notification ... 514
62.8 Launching an Activity from a Notification ... 516
62.9 Adding Actions to a Notification ... 518
62.10 Bundled Notifications ... 519
62.11 Summary ... 521

63. An Android Direct Reply Notification Tutorial ... 523

63.1 Creating the DirectReply Project ... 523
63.2 Designing the User Interface ... 523
63.3 Requesting Notification Permission ... 524
63.4 Creating the Notification Channel ... 525
63.5 Building the RemoteInput Object ... 526
63.6 Creating the PendingIntent ... 527
63.7 Creating the Reply Action ... 528
63.8 Receiving Direct Reply Input ... 530
63.9 Updating the Notification ... 531
63.10 Summary ... 533

64. Foldable Devices and Multi-Window Support ... 535

64.1 Foldables and Multi-Window Support ... 535

64.2 Using a Foldable Emulator....536
64.3 Entering Multi-Window Mode....537
64.4 Enabling and using Freeform Support....538
64.5 Checking for Freeform Support....538
64.6 Enabling Multi-Window Support in an App....538
64.7 Specifying Multi-Window Attributes....539
64.8 Detecting Multi-Window Mode in an Activity....540
64.9 Receiving Multi-Window Notifications....540
64.10 Launching an Activity in Multi-Window Mode....541
64.11 Configuring Freeform Activity Size and Position....541
64.12 Summary....542

65. An Overview of Android SQLite Databases....543

65.1 Understanding Database Tables....543
65.2 Introducing Database Schema....543
65.3 Columns and Data Types....543
65.4 Database Rows....544
65.5 Introducing Primary Keys....544
65.6 What is SQLite?....544
65.7 Structured Query Language (SQL)....544
65.8 Trying SQLite on an Android Virtual Device (AVD)....545
65.9 Android SQLite Classes....546
65.9.1 Cursor....547
65.9.2 SQLiteDatabase....547
65.9.3 SQLiteOpenHelper....547
65.9.4 ContentValues....548
65.10 The Android Room Persistence Library....548
65.11 Summary....548

66. An Android SQLite Database Tutorial....549

66.1 About the Database Example....549
66.2 Creating the SQLDemo Project....549
66.3 Designing the User interface....549
66.4 Creating the Data Model....550
66.5 Implementing the Data Handler....551
66.6 The Add Handler Method....553
66.7 The Query Handler Method....553
66.8 The Delete Handler Method....554
66.9 Implementing the Activity Event Methods....554
66.10 Testing the Application....556
66.11 Summary....556

67. Understanding Android Content Providers....557

67.1 What is a Content Provider?....557
67.2 The Content Provider....557
67.2.1 onCreate()....557
67.2.2 query()....557
67.2.3 insert()....557
67.2.4 update()....558

67.2.5 delete() ... 558
67.2.6 getType() ... 558
67.3 The Content URI ... 558
67.4 The Content Resolver ... 558
67.5 The <provider> Manifest Element ... 559
67.6 Summary ... 559

68. An Android Content Provider Tutorial ... 561

68.1 Copying the SQLDemo Project ... 561
68.2 Adding the Content Provider Package ... 561
68.3 Creating the Content Provider Class ... 562
68.4 Constructing the Authority and Content URI ... 563
68.5 Implementing URI Matching in the Content Provider ... 564
68.6 Implementing the Content Provider onCreate() Method ... 565
68.7 Implementing the Content Provider insert() Method ... 565
68.8 Implementing the Content Provider query() Method ... 566
68.9 Implementing the Content Provider update() Method ... 567
68.10 Implementing the Content Provider delete() Method ... 569
68.11 Declaring the Content Provider in the Manifest File ... 570
68.12 Modifying the Database Handler ... 571
68.13 Summary ... 573

69. An Android Content Provider Client Tutorial ... 575

69.1 Creating the SQLDemoClient Project ... 575
69.2 Designing the User interface ... 575
69.3 Accessing the Content Provider ... 575
69.4 Adding the Query Permission ... 576
69.5 Testing the Project ... 577
69.6 Summary ... 577

70. The Android Room Persistence Library ... 579

70.1 Revisiting Modern App Architecture ... 579
70.2 Key Elements of Room Database Persistence ... 579
70.2.1 Repository ... 580
70.2.2 Room Database ... 580
70.2.3 Data Access Object (DAO) ... 580
70.2.4 Entities ... 580
70.2.5 SQLite Database ... 580
70.3 Understanding Entities ... 581
70.4 Data Access Objects ... 584
70.5 The Room Database ... 585
70.6 The Repository ... 586
70.7 In-Memory Databases ... 587
70.8 Database Inspector ... 587
70.9 Summary ... 587

71. An Android TableLayout and TableRow Tutorial ... 589

71.1 The TableLayout and TableRow Layout Views ... 589
71.2 Creating the Room Database Project ... 590
71.3 Converting to a LinearLayout ... 590

71.4 Adding the TableLayout to the User Interface 591
71.5 Configuring the TableRows 592
71.6 Adding the Button Bar to the Layout 593
71.7 Adding the RecyclerView 594
71.8 Adjusting the Layout Margins 595
71.9 Summary 595

72. An Android Room Database and Repository Tutorial 597

72.1 About the RoomDemo Project 597
72.2 Modifying the Build Configuration 597
72.3 Building the Entity 598
72.4 Creating the Data Access Object 600
72.5 Adding the Room Database 601
72.6 Adding the Repository 601
72.7 Adding the ViewModel 604
72.8 Creating the Product Item Layout 605
72.9 Adding the RecyclerView Adapter 606
72.10 Preparing the Main Activity 607
72.11 Adding the Button Listeners 608
72.12 Adding LiveData Observers 609
72.13 Initializing the RecyclerView 610
72.14 Testing the RoomDemo App 610
72.15 Using the Database Inspector 610
72.16 Summary 611

73. Accessing Cloud Storage using the Android Storage Access Framework 613

73.1 The Storage Access Framework 613
73.2 Working with the Storage Access Framework 614
73.3 Filtering Picker File Listings 614
73.4 Handling Intent Results 615
73.5 Reading the Content of a File 615
73.6 Writing Content to a File 616
73.7 Deleting a File 617
73.8 Gaining Persistent Access to a File 617
73.9 Summary 617

74. An Android Storage Access Framework Example 619

74.1 About the Storage Access Framework Example 619
74.2 Creating the Storage Access Framework Example 619
74.3 Designing the User Interface 619
74.4 Adding the Activity Launchers 620
74.5 Creating a New Storage File 622
74.6 Saving to a Storage File 622
74.7 Opening and Reading a Storage File 624
74.8 Testing the Storage Access Application 625
74.9 Summary 626

75. Video Playback on Android using the VideoView and MediaController Classes 627

75.1 Introducing the Android VideoView Class 627
75.2 Introducing the Android MediaController Class 628

75.3 Creating the Video Playback Example 628
75.4 Designing the VideoPlayer Layout 628
75.5 Downloading the Video File 629
75.6 Configuring the VideoView 629
75.7 Adding the MediaController to the Video View 631
75.8 Setting up the onPreparedListener 631
75.9 Summary 632

76. Android Picture-in-Picture Mode 633

76.1 Picture-in-Picture Features 633
76.2 Enabling Picture-in-Picture Mode 634
76.3 Configuring Picture-in-Picture Parameters 634
76.4 Entering Picture-in-Picture Mode 635
76.5 Detecting Picture-in-Picture Mode Changes 635
76.6 Adding Picture-in-Picture Actions 636
76.7 Summary 636

77. An Android Picture-in-Picture Tutorial 639

77.1 Adding Picture-in-Picture Support to the Manifest 639
77.2 Adding a Picture-in-Picture Button 639
77.3 Entering Picture-in-Picture Mode 640
77.4 Detecting Picture-in-Picture Mode Changes 641
77.5 Adding a Broadcast Receiver 641
77.6 Adding the PiP Action 642
77.7 Testing the Picture-in-Picture Action 645
77.8 Summary 646

78. Android Audio Recording and Playback using MediaPlayer and MediaRecorder 647

78.1 Playing Audio 647
78.2 Recording Audio and Video using the MediaRecorder Class 648
78.3 About the Example Project 649
78.4 Creating the AudioApp Project 649
78.5 Designing the User Interface 649
78.6 Checking for Microphone Availability 650
78.7 Initializing the Activity 651
78.8 Implementing the recordAudio() Method 652
78.9 Implementing the stopAudio() Method 652
78.10 Implementing the playAudio() method 653
78.11 Configuring and Requesting Permissions 653
78.12 Testing the Application 655
78.13 Summary 656

79. Working with the Google Maps Android API in Android Studio 657

79.1 The Elements of the Google Maps Android API 657
79.2 Creating the Google Maps Project 658
79.3 Creating a Google Cloud Billing Account 658
79.4 Creating a New Google Cloud Project 659
79.5 Enabling the Google Maps SDK 660
79.6 Generating a Google Maps API Key 661
79.7 Adding the API Key to the Android Studio Project 662

79.8 Testing the Application ... 662
79.9 Understanding Geocoding and Reverse Geocoding ... 662
79.10 Adding a Map to an Application ... 664
79.11 Requesting Current Location Permission ... 664
79.12 Displaying the User's Current Location ... 666
79.13 Changing the Map Type ... 667
79.14 Displaying Map Controls to the User ... 668
79.15 Handling Map Gesture Interaction ... 669
79.15.1 Map Zooming Gestures ... 669
79.15.2 Map Scrolling/Panning Gestures ... 669
79.15.3 Map Tilt Gestures ... 669
79.15.4 Map Rotation Gestures ... 669
79.16 Creating Map Markers ... 670
79.17 Controlling the Map Camera ... 671
79.18 Summary ... 672

80. Printing with the Android Printing Framework ... 673

80.1 The Android Printing Architecture ... 673
80.2 The Print Service Plugins ... 673
80.3 Google Cloud Print ... 674
80.4 Printing to Google Drive ... 674
80.5 Save as PDF ... 675
80.6 Printing from Android Devices ... 675
80.7 Options for Building Print Support into Android Apps ... 676
80.7.1 Image Printing ... 676
80.7.2 Creating and Printing HTML Content ... 677
80.7.3 Printing a Web Page ... 678
80.7.4 Printing a Custom Document ... 679
80.8 Summary ... 679

81. An Android HTML and Web Content Printing Example ... 681

81.1 Creating the HTML Printing Example Application ... 681
81.2 Printing Dynamic HTML Content ... 681
81.3 Creating the Web Page Printing Example ... 684
81.4 Removing the Floating Action Button ... 684
81.5 Removing Navigation Features ... 684
81.6 Designing the User Interface Layout ... 686
81.7 Accessing the WebView from the Main Activity ... 686
81.8 Loading the Web Page into the WebView ... 687
81.9 Adding the Print Menu Option ... 688
81.10 Summary ... 690

82. A Guide to Android Custom Document Printing ... 691

82.1 An Overview of Android Custom Document Printing ... 691
82.1.1 Custom Print Adapters ... 691
82.2 Preparing the Custom Document Printing Project ... 692
82.3 Designing the UI ... 692
82.4 Creating the Custom Print Adapter ... 693
82.5 Implementing the onLayout() Callback Method ... 694

82.6 Implementing the onWrite() Callback Method ... 697
82.7 Checking a Page is in Range ... 699
82.8 Drawing the Content on the Page Canvas ... 700
82.9 Starting the Print Job ... 702
82.10 Testing the Application ... 703
82.11 Summary ... 703

83. An Introduction to Android App Links ... 705

83.1 An Overview of Android App Links ... 705
83.2 App Link Intent Filters ... 705
83.3 Handling App Link Intents ... 706
83.4 Associating the App with a Website ... 706
83.5 Summary ... 707

84. An Android Studio App Links Tutorial ... 709

84.1 About the Example App ... 709
84.2 The Database Schema ... 709
84.3 Loading and Running the Project ... 709
84.4 Adding the URL Mapping ... 711
84.5 Adding the Intent Filter ... 714
84.6 Adding Intent Handling Code ... 714
84.7 Testing the App ... 717
84.8 Creating the Digital Asset Links File ... 717
84.9 Testing the App Link ... 718
84.10 Summary ... 718

85. An Android Biometric Authentication Tutorial ... 719

85.1 An Overview of Biometric Authentication ... 719
85.2 Creating the Biometric Authentication Project ... 719
85.3 Configuring Device Fingerprint Authentication ... 720
85.4 Adding the Biometric Permission to the Manifest File ... 720
85.5 Designing the User Interface ... 721
85.6 Adding a Toast Convenience Method ... 721
85.7 Checking the Security Settings ... 722
85.8 Configuring the Authentication Callbacks ... 723
85.9 Adding the CancellationSignal ... 724
85.10 Starting the Biometric Prompt ... 724
85.11 Testing the Project ... 725
85.12 Summary ... 726

86. Creating, Testing, and Uploading an Android App Bundle ... 727

86.1 The Release Preparation Process ... 727
86.2 Android App Bundles ... 727
86.3 Register for a Google Play Developer Console Account ... 728
86.4 Configuring the App in the Console ... 729
86.5 Enabling Google Play App Signing ... 730
86.6 Creating a Keystore File ... 730
86.7 Creating the Android App Bundle ... 731
86.8 Generating Test APK Files ... 733
86.9 Uploading the App Bundle to the Google Play Developer Console ... 734

86.10 Exploring the App Bundle 735
86.11 Managing Testers 736
86.12 Rolling the App Out for Testing 736
86.13 Uploading New App Bundle Revisions 737
86.14 Analyzing the App Bundle File 738
86.15 Summary 739

87. An Overview of Android In-App Billing 741

87.1 Preparing a Project for In-App Purchasing 741
87.2 Creating In-App Products and Subscriptions 741
87.3 Billing Client Initialization 742
87.4 Connecting to the Google Play Billing Library 743
87.5 Querying Available Products 744
87.6 Starting the Purchase Process 744
87.7 Completing the Purchase 745
87.8 Querying Previous Purchases 746
87.9 Summary 747

88. An Android In-App Purchasing Tutorial 749

88.1 About the In-App Purchasing Example Project 749
88.2 Creating the InAppPurchase Project 749
88.3 Adding Libraries to the Project 749
88.4 Designing the User Interface 750
88.5 Adding the App to the Google Play Store 751
88.6 Creating an In-App Product 751
88.7 Enabling License Testers 752
88.8 Initializing the Billing Client 753
88.9 Querying the Product 754
88.10 Launching the Purchase Flow 756
88.11 Handling Purchase Updates 756
88.12 Consuming the Product 757
88.13 Restoring a Previous Purchase 758
88.14 Testing the App 759
88.15 Troubleshooting 760
88.16 Summary 760

89. Creating and Managing Overflow Menus on Android 761

89.1 The Overflow Menu 761
89.2 Creating an Overflow Menu 761
89.3 Displaying an Overflow Menu 762
89.4 Responding to Menu Item Selections 762
89.5 Creating Checkable Item Groups 763
89.6 Menus and the Android Studio Menu Editor 764
89.7 Creating the Example Project 765
89.8 Designing the Menu 765
89.9 Modifying the onOptionsItemSelected() Method 767
89.10 Testing the Application 768
89.11 Summary 769

90. Working with Material Design 3 Theming 771

90.1 Material Design 2 vs. Material Design 3 ... 771
90.2 Understanding Material Design Theming ... 771
90.3 Material Design 3 Theming ... 771
90.4 Building a Custom Theme ... 773
90.5 Summary ... 774

91. A Material Design 3 Theming and Dynamic Color Tutorial ... 775

91.1 Creating the ThemeDemo Project ... 775
91.2 Designing the User Interface ... 775
91.3 Building a New Theme ... 777
91.4 Adding the Theme to the Project ... 778
91.5 Enabling Dynamic Color Support ... 779
91.6 Previewing Dynamic Colors ... 780
91.7 Summary ... 781

92. An Overview of Gradle in Android Studio ... 783

92.1 An Overview of Gradle ... 783
92.2 Gradle and Android Studio ... 783
92.2.1 Sensible Defaults ... 783
92.2.2 Dependencies ... 783
92.2.3 Build Variants ... 784
92.2.4 Manifest Entries ... 784
92.2.5 APK Signing ... 784
92.2.6 ProGuard Support ... 784
92.3 The Property and Settings Gradle Build File ... 784
92.4 The Top-level Gradle Build File ... 785
92.5 Module Level Gradle Build Files ... 786
92.6 Configuring Signing Settings in the Build File ... 788
92.7 Running Gradle Tasks from the Command Line ... 789
92.8 Summary ... 790

Index ... 791

Chapter 1

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 Java programming language.

This book begins with the basics and outlines how to set up an Android development and testing environment, followed by an overview of areas such as tool windows, the code editor, and the Layout Editor tool. An introduction to the architecture of Android is followed by an in-depth look at the design of Android applications and user interfaces using the Android Studio environment.

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 Java 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/jellyfishjava

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/jellyfishjava

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.

Chapter 3

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:

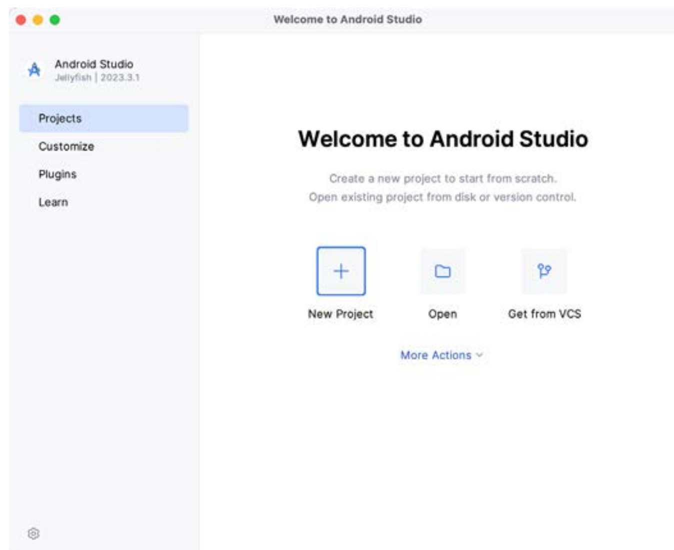

Figure 3-1

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.

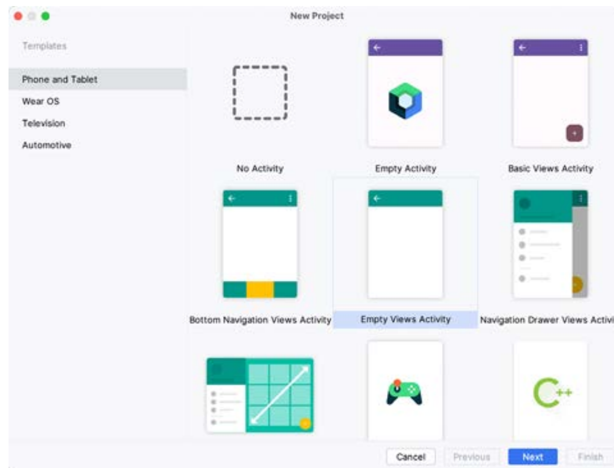

Figure 3-2

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:

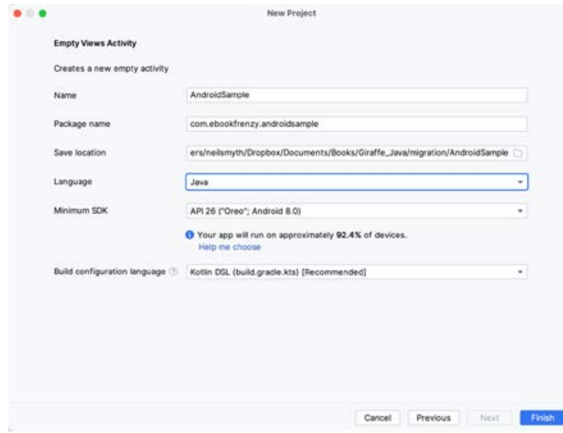

Figure 3-3

Finally, change the *Language* menu to *Java* 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:

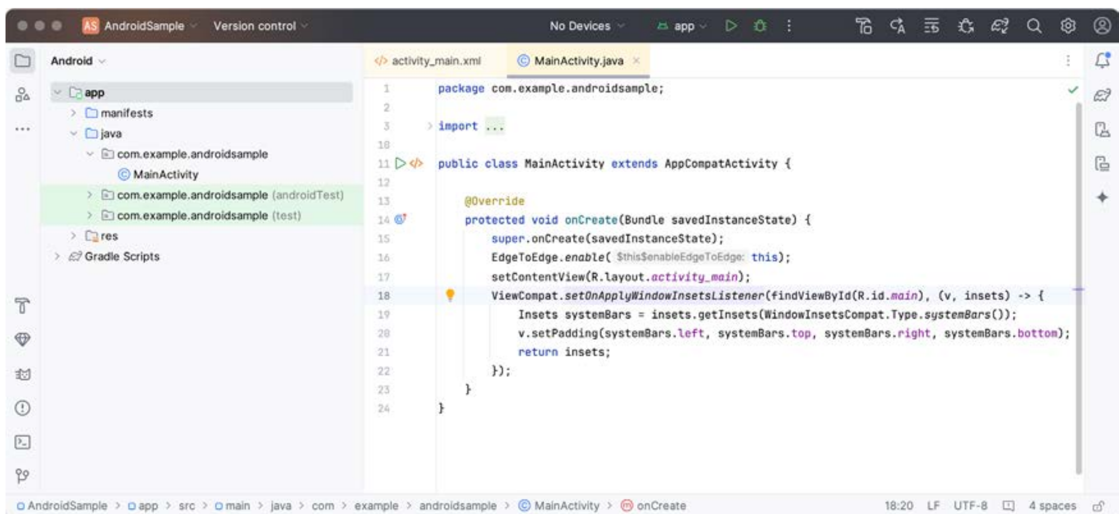

Figure 3-4

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:

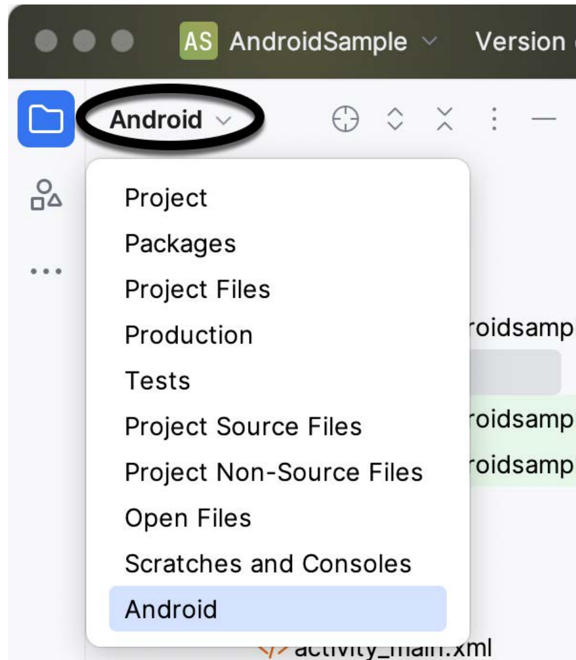

Figure 3-5

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:

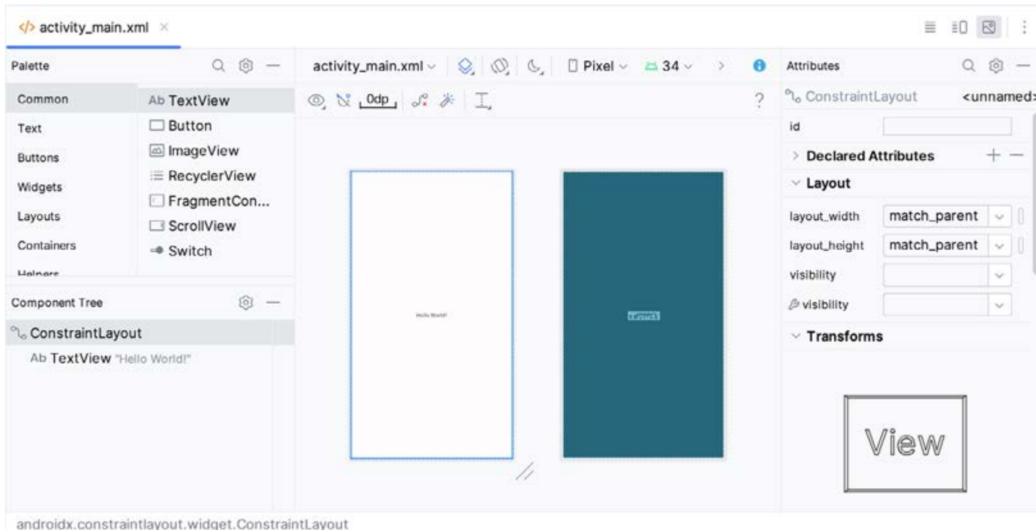

Figure 3-6

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:

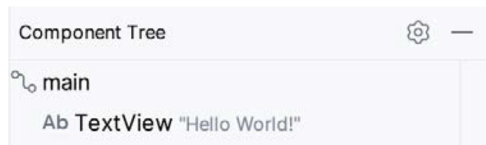

Figure 3-7

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:

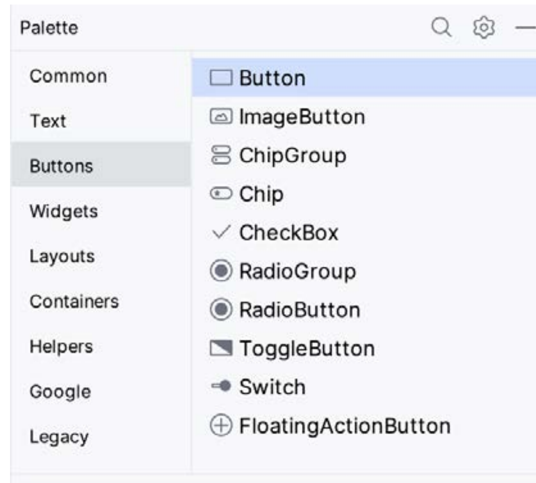

Figure 3-9

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:

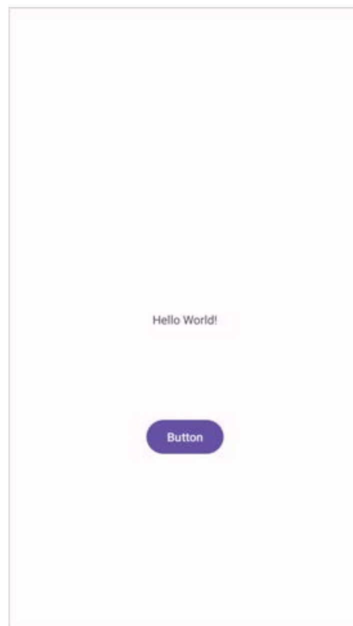

Figure 3-10

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:

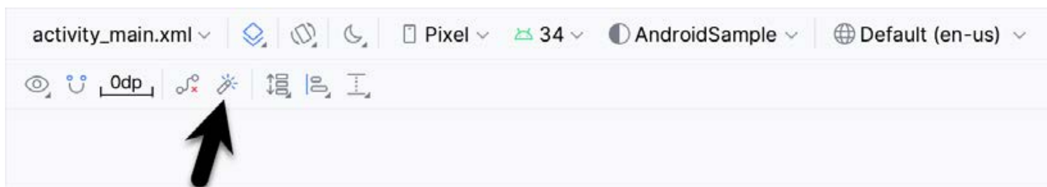

Figure 3-12

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:

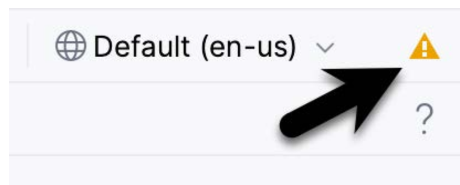

Figure 3-13

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

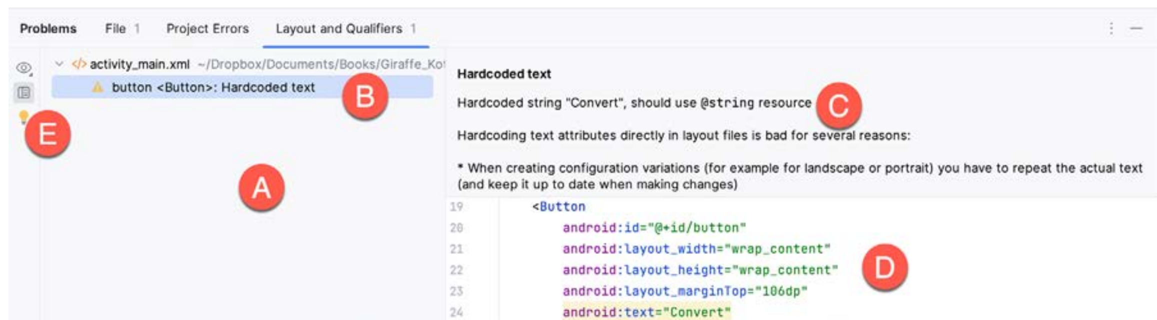

Figure 3-14

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

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:

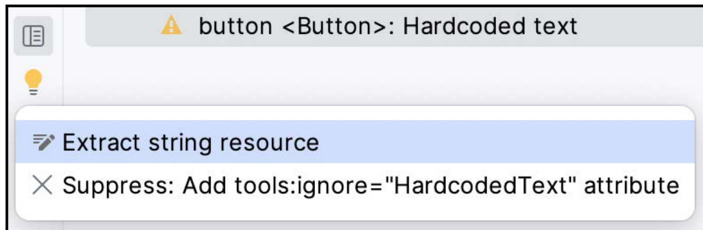

Figure 3-15

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:

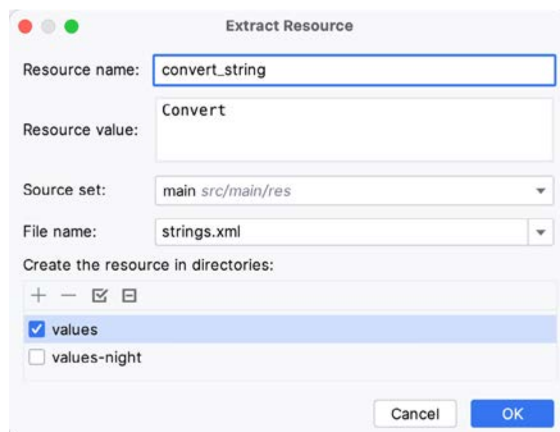

Figure 3-16

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:

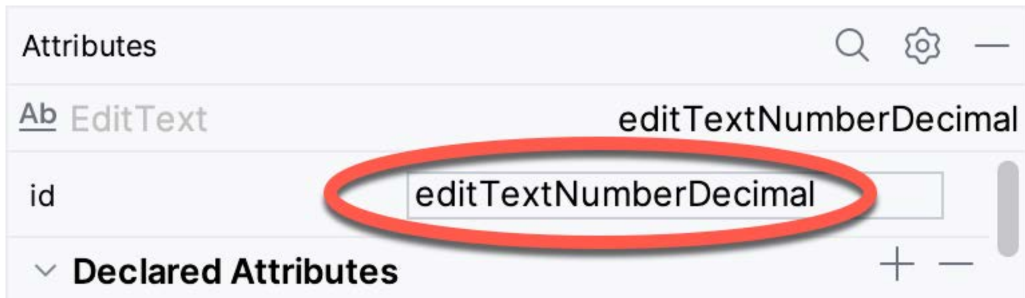

Figure 3-17

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:

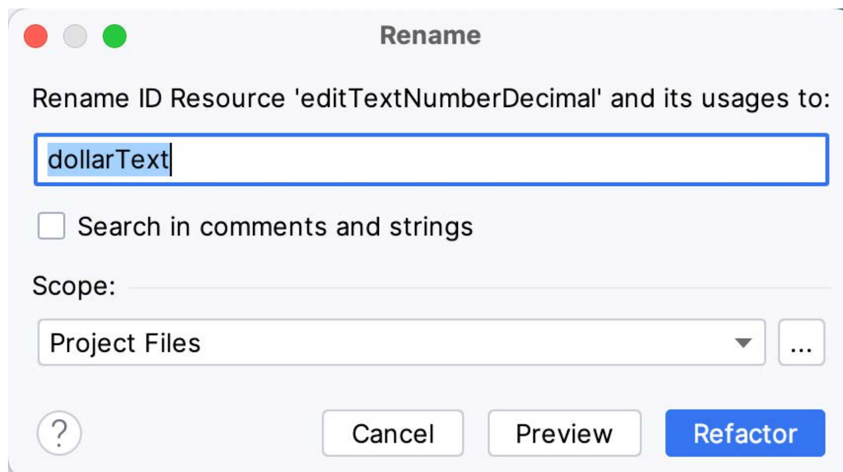

Figure 3-18

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:

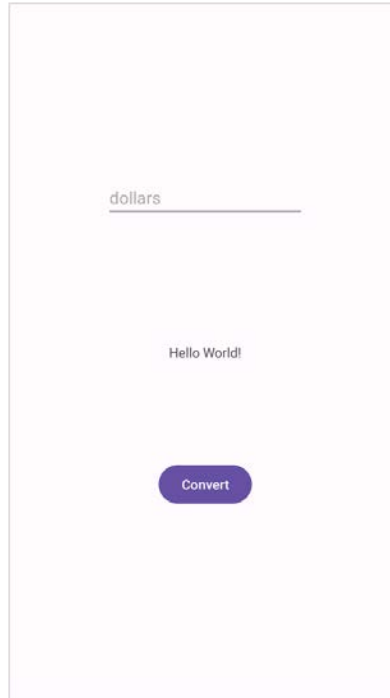

Figure 3-19

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:

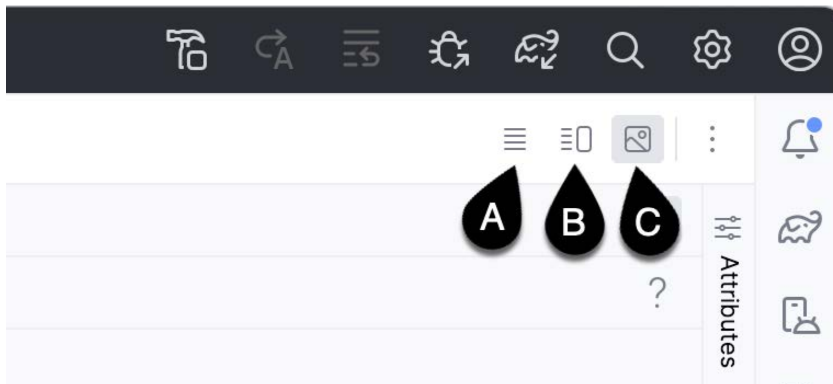

Figure 3-20

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:

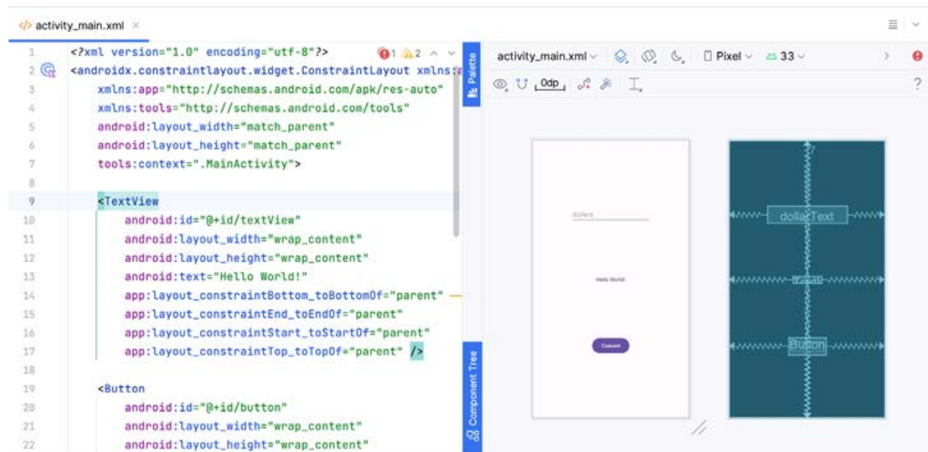

Figure 3-21

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:

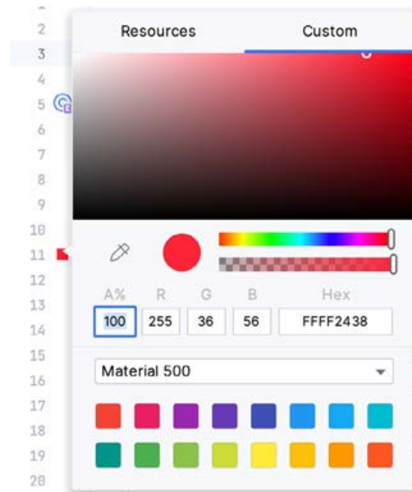

Figure 3-22

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:

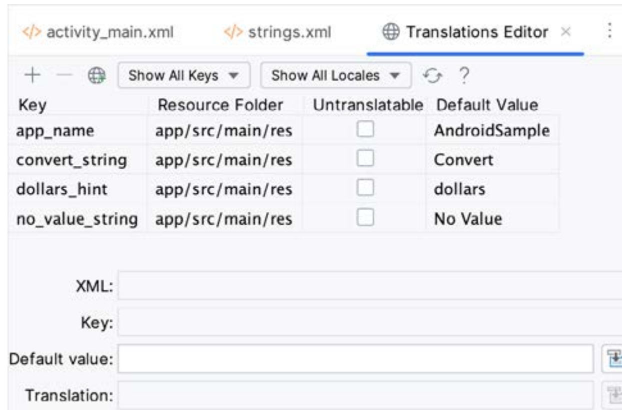

Figure 3-23

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:

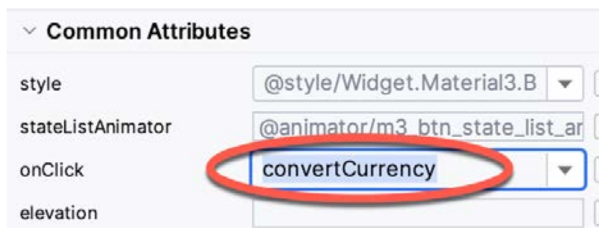

Figure 3-24

Next, double-click on the *MainActivity.java* file in the Project tool window (*app -> 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 android.view.View;
import android.widget.EditText;
import android.widget.TextView;
import java.util.Locale;

public class MainActivity extends AppCompatActivity {
```

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

    public void convertCurrency(View view) {

        EditText dollarText = findViewById(R.id.dollarText);
        TextView textView = findViewById(R.id.textView);

        if (!dollarText.getText().toString().isEmpty()) {

            float dollarValue = Float.parseFloat(dollarText.getText().toString());
            float euroValue = dollarValue * 0.85F;
            textView.setText(String.format(Locale.ENGLISH,"%.2f", euroValue));
        } else {
            textView.setText(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 findByViewId 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.

Chapter 20

20. Working with ConstraintLayout Chains and Ratios in Android Studio

The previous chapters have introduced the key features of the ConstraintLayout class and outlined the best practices for ConstraintLayout-based user interface design within the Android Studio Layout Editor. Although the concepts of ConstraintLayout chains and ratios were outlined in the chapter entitled *"A Guide to the Android ConstraintLayout"*, we have not yet addressed how to use these features within the Layout Editor. Therefore, this chapter's focus is to provide practical steps on how to create and manage chains and ratios when using the ConstraintLayout class.

20.1 Creating a Chain

Chains may be implemented by adding a few lines to an activity's XML layout resource file or by using some chain-specific features of the Layout Editor.

Consider a layout consisting of three Button widgets constrained to be positioned in the top-left, top-center, and top-right of the ConstraintLayout parent, as illustrated in Figure 20-1:

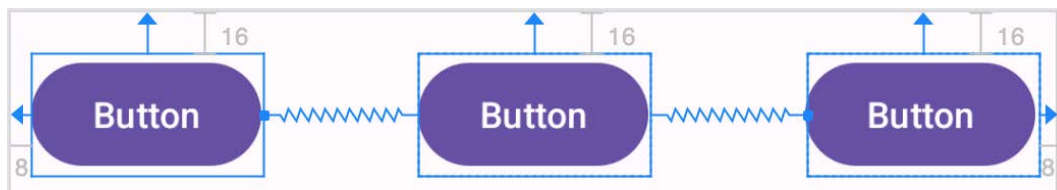

Figure 20-1

To represent such a layout, the XML resource layout file might contain the following entries for the button widgets:

```
<Button
    android:id="@+id/button1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginStart="8dp"
    android:layout_marginTop="16dp"
    android:text="Button"
    app:layout_constraintHorizontal_bias="0.5"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />

<Button
    android:id="@+id/button2"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
```

```
    android:layout_marginEnd="8dp"
    android:layout_marginStart="8dp"
    android:layout_marginTop="16dp"
    android:text="Button"
    app:layout_constraintHorizontal_bias="0.5"
    app:layout_constraintEnd_toStartOf="@+id/button3"
    app:layout_constraintStart_toEndOf="@+id/button1"
    app:layout_constraintTop_toTopOf="parent" />

<Button
    android:id="@+id/button3"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginEnd="8dp"
    android:layout_marginTop="16dp"
    android:text="Button"
    app:layout_constraintHorizontal_bias="0.5"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
```

As currently configured, there are no bi-directional constraints to group these widgets into a chain. To address this, additional constraints need to be added from the right-hand side of button1 to the left side of button2 and from the left side of button3 to the right side of button2 as follows:

```
<Button
    android:id="@+id/button1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginStart="8dp"
    android:layout_marginTop="16dp"
    android:text="Button"
    app:layout_constraintHorizontal_bias="0.5"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintEnd_toStartOf="@+id/button2" />

<Button
    android:id="@+id/button2"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginEnd="8dp"
    android:layout_marginStart="8dp"
    android:layout_marginTop="16dp"
    android:text="Button"
    app:layout_constraintHorizontal_bias="0.5"
    app:layout_constraintEnd_toStartOf="@+id/button3"
    app:layout_constraintStart_toEndOf="@+id/button1"
```

```
    app:layout_constraintTop_toTopOf="parent" />

<Button
    android:id="@+id/button3"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginEnd="8dp"
    android:layout_marginTop="16dp"
    android:text="Button"
    app:layout_constraintHorizontal_bias="0.5"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintStart_toEndOf="@+id/button2" />
```

With these changes, the widgets now have bi-directional horizontal constraints configured. This constitutes a ConstraintLayout chain represented visually within the Layout Editor by chain connections, as shown in Figure 20-2 below. Note that the chain has defaulted to the *spread* chain style in this configuration.

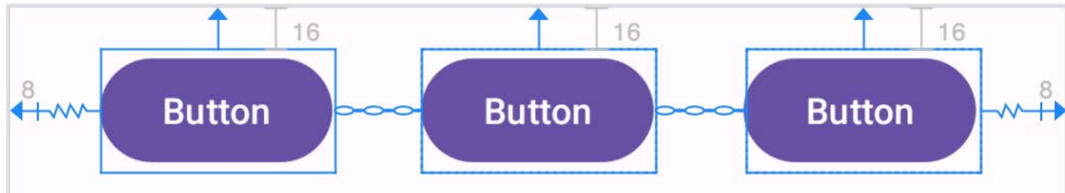

Figure 20-2

A chain may also be created by right-clicking on one of the views and selecting the *Chains -> Create Horizontal Chain* or *Chains -> Create Vertical Chain* menu options.

20.2 Changing the Chain Style

If no chain style is configured, the ConstraintLayout will default to the spread chain style. The chain style can be altered by right-clicking any of the widgets in the chain and selecting the *Cycle Chain Mode* menu option. Each time the menu option is clicked, the style will switch to another setting in the order of spread, spread inside, and packed.

Alternatively, the style may be specified in the Attributes tool window unfolding the *layout_constraints* property and changing either the *horizontal_chainStyle* or *vertical_chainStyle* property depending on the orientation of the chain:

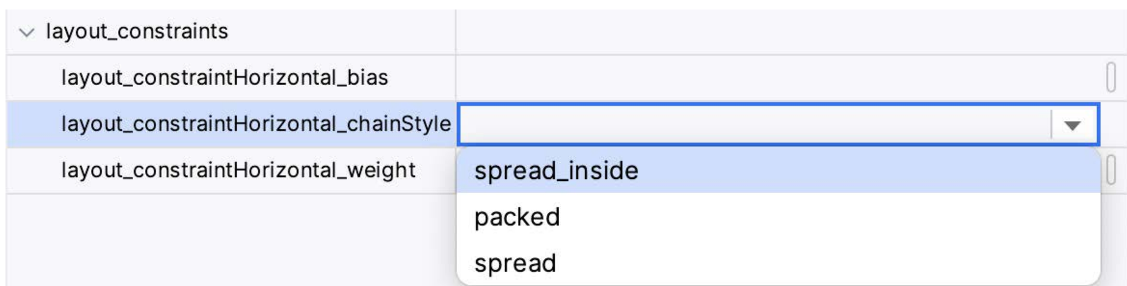

Figure 20-3

20.3 Spread Inside Chain Style

Figure 20-4 illustrates the effect of changing the chain style to the *spread inside* chain style using the above techniques:

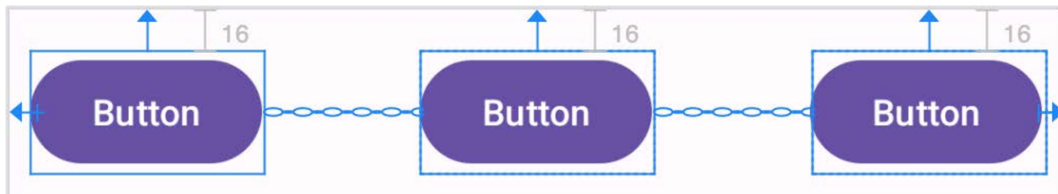

Figure 20-4

20.4 Packed Chain Style

Using the same technique, changing the chain style property to *packed* causes the layout to change, as shown in Figure 20-5:

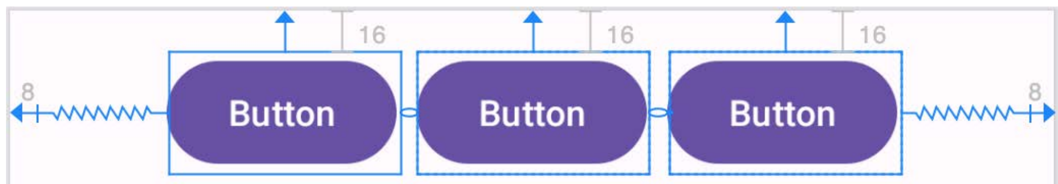

Figure 20-5

20.5 Packed Chain Style with Bias

The positioning of the packed chain may be influenced by applying a bias value. The bias can be between 0.0 and 1.0, with 0.5 representing the parent's center. Bias is controlled by selecting the chain head widget and assigning a value to the *layout_constraintHorizontal_bias* or *layout_constraintVertical_bias* attribute in the Attributes panel. Figure 20-6 shows a packed chain with a horizontal bias setting of 0.2:

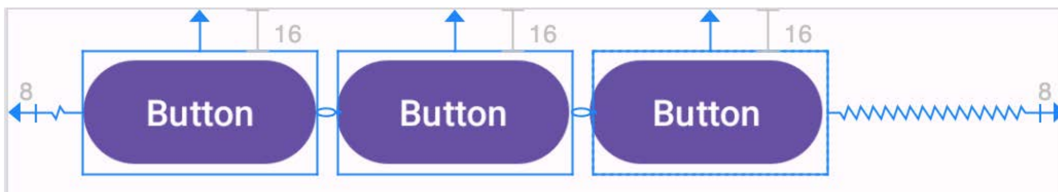

Figure 20-6

20.6 Weighted Chain

The final area of chains to explore involves weighting the individual widgets to control how much space each widget in the chain occupies within the available space. A weighted chain may only be implemented using the spread chain style, and any widget within the chain that responds to the weight property must have the corresponding dimension property (height for a vertical chain and width for a horizontal chain) configured for *match constraint* mode. Match constraint mode for a widget dimension may be configured by selecting the widget, displaying the Attributes panel, and changing the dimension to *match_constraint* (equivalent to 0dp). In Figure 20-7, for example, the *layout_width* constraint for a button has been set to *match_constraint (0dp)* to indicate that the width of the widget is to be determined based on the prevailing constraint settings:

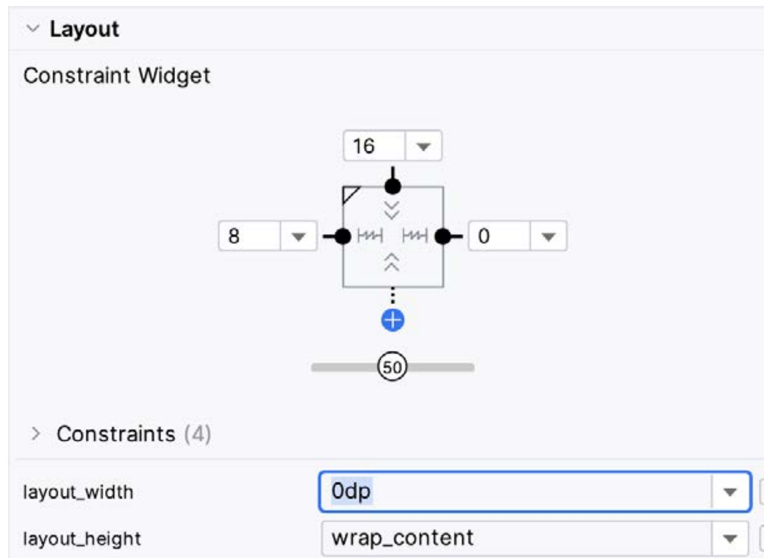

Figure 20-7

Assuming that the spread chain style has been selected and all three buttons have been configured such that the width dimension is set to match the constraints, the widgets in the chain will expand equally to fill the available space:

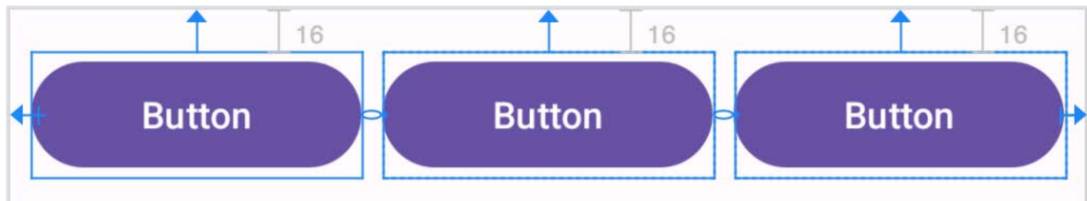

Figure 20-8

The amount of space occupied by each widget relative to the other widgets in the chain can be controlled by adding weight properties to the widgets. Figure 20-9 shows the effect of setting the *layout_constraintHorizontal_weight* property to 4 on button1, and to 2 on both button2 and button3:

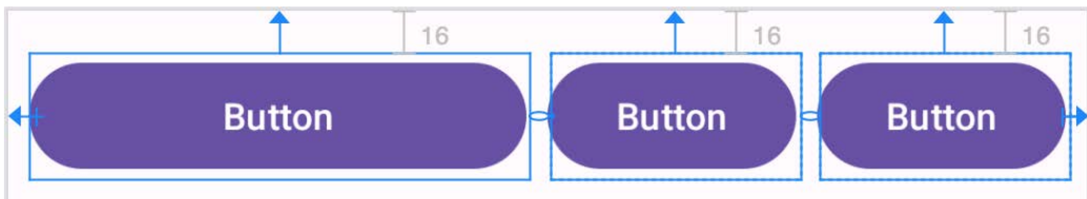

Figure 20-9

As a result of these weighting values, button1 occupies half of the space (4/8), while button2 and button3 each occupy one-quarter (2/8) of the space.

20.7 Working with Ratios

ConstraintLayout ratios allow one widget dimension to be sized relative to the widget's other dimension (also referred to as aspect ratio). For example, an aspect ratio setting could be applied to an ImageView to ensure that its width is always twice its height.

A dimension ratio constraint is configured by setting the constrained dimension to match constraint mode and configuring the *layout_constraintDimensionRatio* attribute on that widget to the required ratio. This ratio value may be specified as a float value or a *width:height* ratio setting. The following XML excerpt, for example, configures a ratio of 2:1 on an ImageView widget:

```
<ImageView
        android:layout_width="0dp"
        android:layout_height="100dp"
        android:id="@+id/imageView"
        app:layout_constraintDimensionRatio="2:1" />
```

The above example demonstrates how to configure a ratio when only one dimension is set to *match constraint*. A ratio may also be applied when both dimensions are set to match constraint mode. This involves specifying the ratio preceded with either an H or a W to indicate which of the dimensions is constrained relative to the other.

Consider, for example, the following XML excerpt for an ImageView object:

```
<ImageView
    android:layout_width="0dp"
    android:layout_height="0dp"
    android:id="@+id/imageView"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintDimensionRatio="W,1:3" />
```

In the above example, the height will be defined subject to the constraints applied to it. In this case, constraints have been configured such that it is attached to the top and bottom of the parent view, essentially stretching the widget to fill the entire height of the parent. On the other hand, the width dimension has been constrained to be one-third of the ImageView's height dimension. Consequently, whatever size screen or orientation the layout appears on, the ImageView will always be the same height as the parent and the width one-third of that height.

The same results may also be achieved without manually editing the XML resource file. Whenever a widget dimension is set to match constraint mode, a ratio control toggle appears in the Inspector area of the property panel. Figure 20-10, for example, shows the layout width and height attributes of a button widget set to match constraint mode and 100dp respectively, and highlights the ratio control toggle in the widget sizing preview:

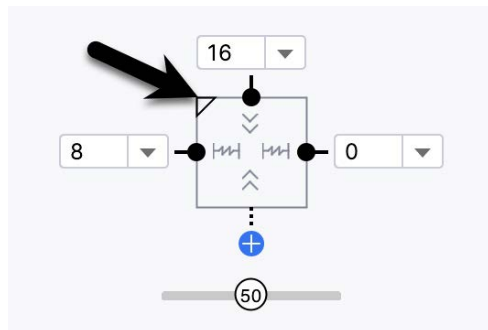

Figure 20-10

By default, the ratio sizing control is toggled off. Clicking on the control enables the ratio constraint and displays an additional field where the ratio may be changed:

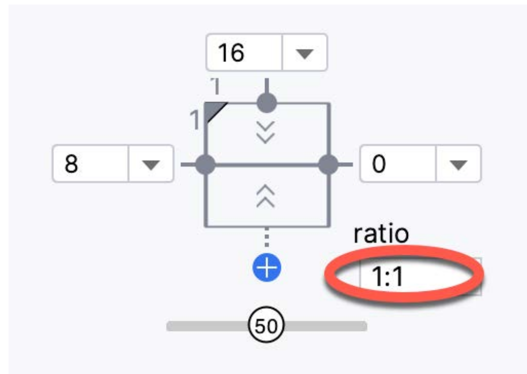

Figure 20-11

20.8 Summary

Both chains and ratios are powerful features of the ConstraintLayout class intended to provide additional options for designing flexible and responsive user interface layouts within Android applications. As outlined in this chapter, the Android Studio Layout Editor has been enhanced to make it easier to use these features during the user interface design process.

Chapter 27

27. An Overview and Example of Android Event Handling

Much has been covered in the previous chapters relating to the design of user interfaces for Android applications. However, an area that has yet to be covered involves how a user's interaction with the user interface triggers the underlying activity to perform a task. In other words, from the previous chapters, we know how to create a user interface containing a button view but not how to make something happen within the application when the user touches it.

Therefore, this chapter's primary objective is to provide an overview of event handling in Android applications together with an Android Studio-based example project.

27.1 Understanding Android Events

Android events can take various forms but are usually generated in response to an external action. The most common form of events, particularly for devices such as tablets and smartphones, involve some form of interaction with the touch screen. Such events fall into the category of *input events*.

The Android framework maintains an *event queue* into which events are placed as they occur. Events are then removed from the queue on a first-in, first-out (FIFO) basis. In the case of an input event, such as a touch on the screen, the event is passed to the view positioned at the location on the screen where the touch took place. In addition to the event notification, the view is also passed a range of information (depending on the event type) about the nature of the event, such as the coordinates of the point of contact between the user's fingertip and the screen.

To handle an event that has been passed, the view must have an *event listener* in place. The Android View class, from which all user interface components are derived, contains a range of event listener interfaces, each containing an abstract declaration for a callback method. To be able to respond to an event of a particular type, a view must register the appropriate event listener and implement the corresponding callback. For example, if a button is to respond to a *click* event (the equivalent of the user touching and releasing the button view as though clicking on a physical button), it must both register the *View.onClickListener* event listener (via a call to the target view's *setOnClickListener()* method) and implement the corresponding *onClick()* callback method. If a "click" event is detected on the screen at the location of the button view, the Android framework will call the *onClick()* method of that view when that event is removed from the event queue. It is, of course, within the implementation of the onClick() callback method that any tasks or other methods called in response to the button click should be performed.

27.2 Using the android:onClick Resource

Before exploring event listeners in more detail, it is worth noting that a shortcut is available when all that is required is for a callback method to be called when a user "clicks" on a button view in the user interface. Consider a user interface layout containing a button view named *button1* with the requirement that when the user touches the button, a method called *buttonClick()* declared in the activity class is called. All that is required to implement this behavior is to write the *buttonClick()* method (which takes as an argument a reference to the view that triggered the click event) and add a single line to the declaration of the button view in the XML file. For example:

```
<Button
        android:id="@+id/button1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:onClick="buttonClick"
        android:text="Click me" />
```

This provides a simple way to capture click events. It does not, however, provide the range of options offered by event handlers, which is the topic of the rest of this chapter. As outlined in later chapters, the onClick property also has limitations in layouts involving fragments. When working within Android Studio Layout Editor, the onClick property can be found and configured in the Attributes panel when a suitable view type is selected in the device screen layout.

27.3 Event Listeners and Callback Methods

In the example activity outlined later in this chapter, the steps involved in registering an event listener and implementing the callback method will be covered in detail. Before doing so, however, it is worth taking some time to outline the event listeners available in the Android framework and the callback methods associated with each one.

- **onClickListener** – Used to detect click style events whereby the user touches and then releases an area of the device display occupied by a view. Corresponds to the *onClick()* callback method, which is passed a reference to the view that received the event as an argument.

- **onLongClickListener** – Used to detect when the user maintains the touch over a view for an extended period. Corresponds to the *onLongClick()* callback method, which is passed as an argument the view that received the event.

- **onTouchListener** – Used to detect any contact with the touch screen, including individual or multiple touches and gesture motions. Corresponding with the *onTouch()* callback, this topic will be covered in greater detail in the chapter entitled *"Android Touch and Multi-touch Event Handling"*. The callback method is passed as arguments the view that received the event and a MotionEvent object.

- **onCreateContextMenuListener** – Listens for the creation of a context menu as the result of a long click. Corresponds to the *onCreateContextMenu()* callback method. The callback is passed the menu, the view that received the event and a menu context object.

- **onFocusChangeListener** – Detects when focus moves away from the current view due to interaction with a trackball or navigation key. Corresponds to the *onFocusChange()* callback method, which is passed the view that received the event and a Boolean value to indicate whether focus was gained or lost.

- **onKeyListener** – Used to detect when a key on a device is pressed while a view has focus. Corresponds to the *onKey()* callback method. It is passed as arguments the view that received the event, the KeyCode of the physical key that was pressed, and a KeyEvent object.

27.4 An Event Handling Example

In the remainder of this chapter, we will create an Android Studio project designed to demonstrate the implementation of an event listener and corresponding callback method to detect when the user has clicked on a button. The code within the callback method will update a text view to indicate that the event has been processed.

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

Enter *EventExample* into the Name field and specify *com.ebookfrenzy.eventexample* 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 Java. Using the steps outlined in section *11.8 Migrating a Project to View Binding*, convert the project to use view binding.

27.5 Designing the User Interface

The user interface layout for the *MainActivity* class in this example will consist of a ConstraintLayout, a Button, and a TextView, as illustrated in Figure 27-1.

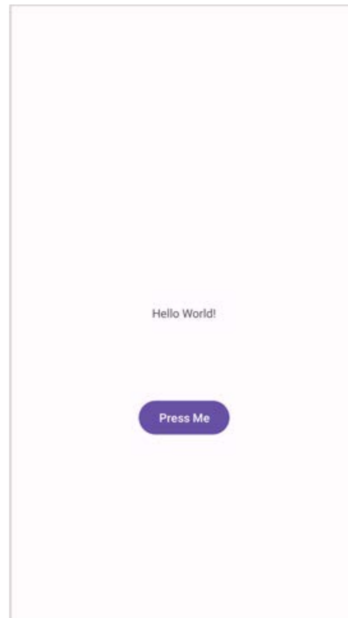

Figure 27-1

Locate and select the *activity_main.xml* file created by Android Studio (located in the Project tool window under *app -> res -> layouts*) and double-click on it to load it into the Layout Editor tool.

Ensure that Autoconnect is enabled, then drag a Button widget from the palette and move it so that it is positioned in the horizontal center of the layout and beneath the existing TextView widget. When correctly positioned, drop the widget into place so that the autoconnect system adds appropriate constraints.

Select the "Hello World!" TextView widget and use the Attributes panel to set the ID to *statusText*. Repeat this step to change the ID of the Button widget to *myButton*.

Add any missing constraints by clicking on the *Infer Constraints* button in the layout editor toolbar.

With the Button widget selected, use the Attributes panel to set the text property to Press Me. Extract the text string on the button to a resource named *press_me*.

With the user interface layout completed, the next step is registering the event listener and callback method.

27.6 The Event Listener and Callback Method

For this example, an *onClickListener* needs to be registered for the *myButton* view. This is achieved by calling the *setOnClickListener()* method of the button view, passing through a new *onClickListener* object as an argument, and implementing the *onClick()* callback method. Since this task only needs to be performed when the activity is created, a good location is the *onCreate()* method of the MainActivity class.

If the *MainActivity.java* file is already open within an editor session, select it by clicking on the tab in the editor panel. Alternatively, locate it within the Project tool window by navigating to (*app* -> *kotlin+java* -> *com.ebookfrenzy.eventexample* -> *MainActivity*) and double-click on it to load it into the code editor. Once loaded, locate the template *onCreate()* method and modify it to obtain a reference to the button view, register the event listener, and implement the *onClick()* callback method:

```
package com.ebookfrenzy.eventexample;

import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;
import android.view.View;
import android.widget.Button;

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
.
.
        setContentView(view);

        binding.myButton.setOnClickListener(
                new Button.OnClickListener() {
                    public void onClick(View v) {

                    }
                }
        );
    }
.
.
}
```

The above code has registered the event listener on the button and implemented the *onClick()* method. If the application were to be run at this point, however, there would be no indication that the event listener installed on the button was working since there is, as yet, no code implemented within the body of the *onClick()* callback method. The goal for the example is to have a message appear on the TextView when the button is clicked, so some further code changes need to be made:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
.
.
       binding.myButton.setOnClickListener(
                new Button.OnClickListener() {
                    public void onClick(View v) {
```

```
                        binding.statusText.setText("Button clicked");
                    }
                }
        );
}
```

Complete this tutorial phase by compiling and running the application on either an AVD emulator or a physical Android device. On touching and releasing the button view (otherwise known as "clicking"), the text view should change to display the "Button clicked" text.

27.7 Consuming Events

The detection of standard clicks (as opposed to long clicks) on views is a straightforward case of event handling. The example will now be extended to include the detection of long click events, which occur when the user clicks and holds a view on the screen and, in doing so, cover the topic of event consumption.

Consider the code for the *onClick()* method in the above section of this chapter. The callback is declared as *void* and, as such, does not return a value to the Android framework after it has finished executing.

On the other hand, the code assigned to the *onLongClickListener* is required to return a Boolean value to the Android framework. The purpose of this return value is to indicate to the Android runtime whether or not the callback has consumed the event. If the callback returns a true value, the framework discards the event. If, on the other hand, the callback returns a false value, the Android framework will consider the event still to be active and pass it along to the next matching event listener registered on the same view.

As with many programming concepts, this is best demonstrated with an example. The first step is to add an event listener and callback method for long clicks to the button view in the example activity:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
.
.
        binding.myButton.setOnLongClickListener(
                new Button.OnLongClickListener() {
                     public boolean onLongClick(View v) {
                       binding.statusText.setText("Long button click");
                       return true;
                }
              }
        );
    }
}
```

When a long click is detected, the *onLongClick()* callback method will display "Long button click" on the text view. Note, however, that the callback method returns a *true* value to indicate that it has consumed the event. Run the application and press and hold the Button view until the "Long button click" text appears in the text view. On releasing the button, the text view displays the "Long button click" text indicating that the onClick listener code was not called.

Next, modify the code so that the onLongClick listener now returns a *false* value:

```
button.setOnLongClickListener(
    new Button.OnLongClickListener() {
```

```
        public boolean onLongClick(View v) {
                TextView myTextView = findViewById(R.id.myTextView);
                myTextView.setText("Long button click");
                return false;
        }
    }
);
```

Once again, compile and run the application and perform a long click on the button until the long click message appears. However, after releasing the button this time, note that the onClick listener is also triggered, and the text changes to "Button clicked". This is because the *false* value returned by the *onLongClick* listener code indicated to the Android framework that the event was not consumed by the method and was eligible to be passed on to the next registered listener on the view. In this case, the runtime ascertained that the onClickListener on the button was also interested in events of this type and subsequently called the *onClick* listener code.

27.8 Summary

A user interface is of little practical use if the views it contains do not do anything in response to user interaction. Android bridges the gap between the user interface and the back-end code of the application through the concepts of event listeners and callback methods. The Android View class defines a set of event listeners which can be registered on view objects. Each event listener also has associated with it a callback method.

When an event takes place on a view in a user interface, that event is placed into an event queue and handled on a first-in, first-out basis by the Android runtime. If the view on which the event took place has registered a listener that matches the type of event, the corresponding callback method is called. This code then performs any tasks required by the activity before returning. Some callback methods are required to return a Boolean value to indicate whether the event needs to be passed on to other event listeners registered on the view or discarded by the system.

Now that the basics of event handling have been covered, the next chapter will explore touch events with a particular emphasis on handling multiple touches.

Chapter 43

43. An Introduction to MotionLayout

The MotionLayout class provides an easy way to add animation effects to the views of a user interface layout. This chapter will begin by providing an overview of MotionLayout and introduce the concepts of MotionScenes, Transitions, and Keyframes. Once these basics have been covered, the next two chapters (entitled *"An Android MotionLayout Editor Tutorial"* and *"A MotionLayout KeyCycle Tutorial"*) will provide additional detail and examples of MotionLayout animation in action through the creation of example projects.

43.1 An Overview of MotionLayout

MotionLayout is a layout container, the primary purpose of which is to animate the transition of views within a layout from one state to another. MotionLayout could, for example, animate the motion of an ImageView instance from the top left-hand corner of the screen to the bottom right-hand corner over a specified time. In addition to the position of a view, other attribute changes may also be animated, such as the color, size, or rotation angle. These state changes can also be interpolated (such that a view moves, rotates, and changes size throughout the animation).

The motion of a view using MotionLayout may be performed in a straight line between two points or implemented to follow a path comprising intermediate points at different positions between the start and end points. MotionLayout also supports using touches and swipes to initiate and control animation.

MotionLayout animations are declared entirely in XML and do not typically require writing code. These XML declarations may be implemented manually in the Android Studio code editor, visually using the MotionLayout editor, or combining both approaches.

43.2 MotionLayout

When implementing animation, the ConstraintLayout container typically used in a user interface must first be converted to a MotionLayout instance (a task which can be achieved by right-clicking on the ConstraintLayout in the layout editor and selecting the *Convert to MotionLayout* menu option). MotionLayout also requires at least version 2.0.0 of the ConstraintLayout library.

Unsurprisingly since it is a subclass of ConstraintLayout, MotionLayout supports all of the layout features of the ConstraintLayout. Therefore, a user interface layout can be similarly designed when using MotionLayout for views that do not require animation.

For views that are to be animated, two ConstraintSets are declared, defining the appearance and location of the view at the start and end of the animation. A *transition* declaration defines *keyframes* to apply additional effects to the target view between these start and end states and click and swipe handlers used to start and control the animation.

The start and end ConstraintSets and the transitions are declared within a MotionScene XML file.

43.3 MotionScene

As we have seen in earlier chapters, an XML layout file contains the information necessary to configure the appearance and layout behavior of the static views presented to the user, and this is still the case when using MotionLayout. For non-static views (in other words, the views that will be animated), those views are still declared within the layout file, but the start, end, and transition declarations related to those views are stored in a separate XML file referred to as the MotionScene file (so called because all of the declarations are defined

within a MotionScene element). This file is imported into the layout XML file and contains the start and end ConstraintSets and Transition declarations (a single file can contain multiple ConstraintSet pairs and Transition declarations, allowing different animations to be targeted to specific views within the user interface layout).

The following listing shows a template for a MotionScene file:

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

    <Transition
        motion:constraintSetEnd="@+id/end"
        motion:constraintSetStart="@id/start"
        motion:duration="1000">
       <KeyFrameSet>
       </KeyFrameSet>
    </Transition>

    <ConstraintSet android:id="@+id/start">
    </ConstraintSet>

    <ConstraintSet android:id="@+id/end">
    </ConstraintSet>
</MotionScene>
```

In the above XML, ConstraintSets named *start* and *end* (though any name can be used) have been declared, which, at this point, are yet to contain any constraint elements. The Transition element defines that these ConstraintSets represent the animation start and end points and contain an empty KeyFrameSet element ready to be populated with additional animation keyframe entries. The Transition element also includes a millisecond duration property to control the running time of the animation.

ConstraintSets do not have to imply the motion of a view. It is possible to have the start and end sets declare the same location on the screen and then use the transition to animate other property changes, such as scale and rotation angle.

ConstraintSets do not have to imply the motion of a view. It is possible, for example, to have the start and end sets declare the same location on the screen and then use the transition to animate other property changes, such as scale and rotation angle.

43.4 Configuring ConstraintSets

The ConstraintSets in the MotionScene file allow the full set of ConstraintLayout settings to be applied to a view regarding positioning, sizing, and relation to the parent and other views. In addition, the following attributes may also be included within the ConstraintSet declarations:

- alpha
- visibility
- elevation
- rotation

- rotationX
- rotationY
- translationX
- translationY
- translationZ
- scaleX
- scaleY

For example, to rotate the view by 180° during the animation, the following could be declared within the start and end constraints:

```
<ConstraintSet android:id="@+id/start">
    <Constraint
.
.
        motion:layout_constraintStart_toStartOf="parent"
        android:rotation="0">
    </Constraint>
</ConstraintSet>

<ConstraintSet android:id="@+id/end">
    <Constraint
.
.
        motion:layout_constraintBottom_toBottomOf="parent"
        android:rotation="180">
    </Constraint>
</ConstraintSet>
```

The above changes tell MotionLayout that the view is to start at 0° and then, during the animation, rotate a full 180° before coming to rest upside-down.

43.5 Custom Attributes

In addition to the standard attributes listed above, it is possible to specify a range of *custom attributes* (declared using CustomAttribute). In fact, just about any property available on the view type can be specified as a custom attribute for inclusion in an animation. To identify the attribute's name, find the getter/setter name from the documentation for the target view class, remove the get/set prefix, and lower the case of the first remaining character. For example, to change the background color of a Button view in code, we might call the *setBackgroundColor()* setter method as follows:

```
myButton.setBackgroundColor(Color.RED)
```

When setting this attribute in a constraint set or keyframe, the attribute name will be *backgroundColor*. In addition to the attribute name, the value must also be declared using the appropriate type from the following list of options:

- **motion:customBoolean** - Boolean attribute values.

- **motion:customColorValue** - Color attribute values.
- **motion:customDimension** - Dimension attribute values.
- **motion:customFloatValue** - Floating point attribute values.
- **motion:customIntegerValue** - Integer attribute values.
- **motion:customStringValue** - String attribute values

For example, a color setting will need to be assigned using the *customColorValue* type:

```
<CustomAttribute
    motion:attributeName="backgroundColor"
    motion:customColorValue="#43CC76" />
```

The following excerpt from a MotionScene file, for example, declares start and end constraints for a view in addition to changing the background color from green to red:

```
.
.
   <ConstraintSet android:id="@+id/start">
        <Constraint
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            motion:layout_editor_absoluteX="21dp"
            android:id="@+id/button"
            motion:layout_constraintTop_toTopOf="parent"
            motion:layout_constraintStart_toStartOf="parent" >
            <CustomAttribute
                motion:attributeName="backgroundColor"
                motion:customColorValue="#33CC33" />
        </Constraint>
    </ConstraintSet>

    <ConstraintSet android:id="@+id/end">
        <Constraint
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            motion:layout_editor_absoluteY="21dp"
            android:id="@+id/button"
            motion:layout_constraintEnd_toEndOf="parent"
            motion:layout_constraintBottom_toBottomOf="parent" >
            <CustomAttribute
                motion:attributeName="backgroundColor"
                motion:customColorValue="#F80A1F" />
        </Constraint>
    </ConstraintSet>
.
.
```

43.6 Triggering an Animation

Without some event to tell MotionLayout to start the animation, none of the settings in the MotionScene file will affect the layout (except that the view will be positioned based on the setting in the start ConstraintSet).

The animation can be configured to start in response to either screen tap (OnClick) or swipe motion (OnSwipe) gesture. The OnClick handler causes the animation to start and run until completion, while OnSwipe will synchronize the animation to move back and forth along the timeline to match the touch motion. The OnSwipe handler will also respond to "flinging" motions on the screen. The OnSwipe handler also provides options to configure how the animation reacts to dragging in different directions and the side of the target view to which the swipe is to be anchored. This allows, for example, left-ward dragging motions to move a view in the corresponding direction while preventing an upward motion from causing a view to move sideways (unless, of course, that is the required behavior).

The OnSwipe and OnClick declarations are contained within the Transition element of a MotionScene file. In both cases, the view id must be specified. For example, to implement an OnSwipe handler responding to downward drag motions anchored to the bottom edge of a view named *button*, the following XML would be placed in the Transition element:

```
.
.
<Transition
    motion:constraintSetEnd="@+id/end"
    motion:constraintSetStart="@id/start"
    motion:duration="1000">
   <KeyFrameSet>
   </KeyFrameSet>
    <OnSwipe
        motion:touchAnchorId="@+id/button"
        motion:dragDirection="dragDown"
        motion:touchAnchorSide="bottom" />
</Transition>
.
.
```

Alternatively, to add an OnClick handler to the same button:

```
<OnClick motion:targetId="@id/button"
    motion:clickAction="toggle" />
```

In the above example, the action has been set to *toggle* mode. This mode and the other available options can be summarized as follows:

- **toggle** - Animates to the opposite state. For example, if the view is currently at the transition start point, it will transition to the end point, and vice versa.
- **jumpToStart** - Changes immediately to the start state without animation.
- **jumpToEnd** - Changes immediately to the end state without animation.
- **transitionToStart** - Transitions with animation to the start state.
- **transitionToEnd** - Transitions with animation to the end state.

43.7 Arc Motion

By default, a movement of view position will travel in a straight line between the start and end points. To change the motion to an arc path, use the *pathMotionArc* attribute as follows within the start constraint, configured with either a *startHorizontal* or *startVertical* setting to define whether the arc is to be concave or convex:

```
<ConstraintSet android:id="@+id/start">
    <Constraint
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        motion:layout_editor_absoluteX="21dp"
        android:id="@+id/button"
        motion:layout_constraintTop_toTopOf="parent"
        motion:layout_constraintStart_toStartOf="parent"
        motion:pathMotionArc="startVertical" >
```

Figure 43-1 illustrates startVertical and startHorizontal arcs in comparison to the default straight line motion:

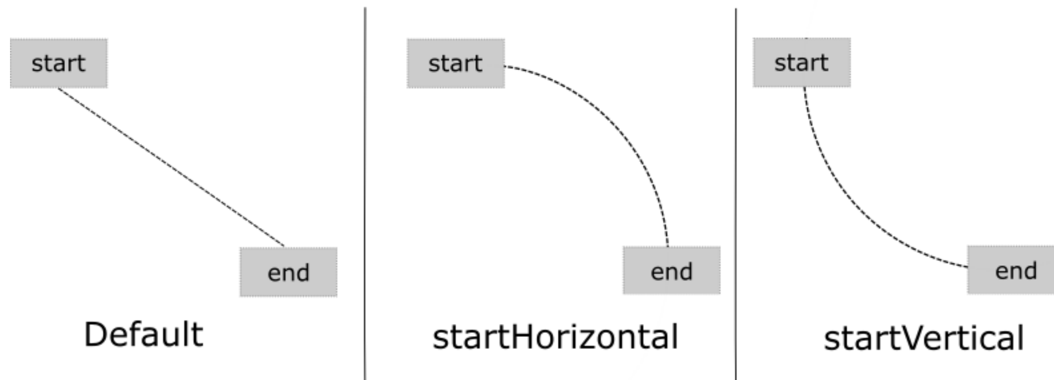

Figure 43-1

43.8 Keyframes

All of the ConstraintSet attributes outlined so far only apply to the start and end points of the animation. In other words, if the rotation property were set to 180° on the end point, the rotation would begin when the animation starts and complete when the end point is reached. It is not, therefore, possible to configure the rotation to reach the full 180° at a point 50% of the way through the animation and then rotate back to the original orientation by the end. Fortunately, this type of effect is available using Keyframes.

Keyframes are used to define intermediate points during the animation at which state changes are to occur. Keyframes could, for example, be declared such that the background color of a view is to have transitioned to blue at a point 50% of the way through the animation, green at the 75% point, and then back to the original color by the end of the animation. Keyframes are implemented within the Transition element of the MotionScene file embedded into the KeyFrameSet element.

MotionLayout supports several types of Keyframe which can be summarized as follows:

43.8.1 Attribute Keyframes

Attribute Keyframes (declared using KeyAttribute) allow view attributes to be changed at intermediate points in the animation timeline. KeyAttribute supports the attributes listed above for ConstraintSets combined with the ability to specify where the change will take effect in the animation timeline. For example, the following

Keyframe declaration will gradually cause the button view to double in size horizontally (scaleX) and vertically (scaleY), reaching full size at 50% through the timeline. For the remainder of the timeline, the view will decrease in size to its original dimensions:

```
<Transition
    motion:constraintSetEnd="@+id/end"
    motion:constraintSetStart="@id/start"
    motion:duration="1000">
   <KeyFrameSet>
       <KeyAttribute
           motion:motionTarget="@+id/button"
           motion:framePosition="50"
           android:scaleX="2.0" />
       <KeyAttribute
           motion:motionTarget="@+id/button"
           motion:framePosition="50"
           android:scaleY="2.0" />
   </KeyFrameSet>
```

43.8.2 Position Keyframes

Position keyframes (KeyPosition) modify the path followed by a view as it moves between the start and end locations. By placing key positions at different points on the timeline, a path of just about any level of complexity can be applied to an animation. Positions are declared using x and y coordinates combined with the corresponding points in the transition timeline. These coordinates must be declared relative to one of the following coordinate systems:

- **parentRelative** - The x and y coordinates are relative to the parent container where the coordinates are specified as a percentage (represented as a value between 0.0 and 1.0):

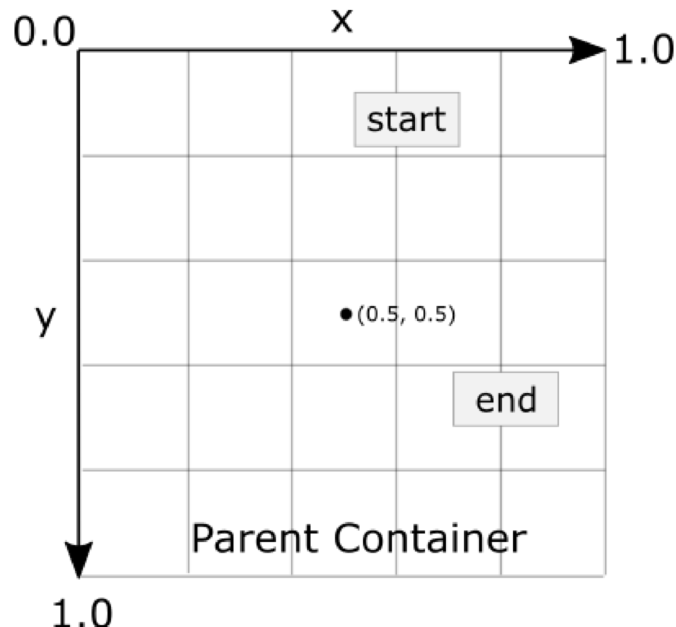

Figure 43-2

- **deltaRelative** - Instead of relative to the parent, the x and y coordinates are relative to the start and end positions. For example, the start point is (0, 0) the end point (1, 1). Keep in mind that the x and y coordinates can be negative values):

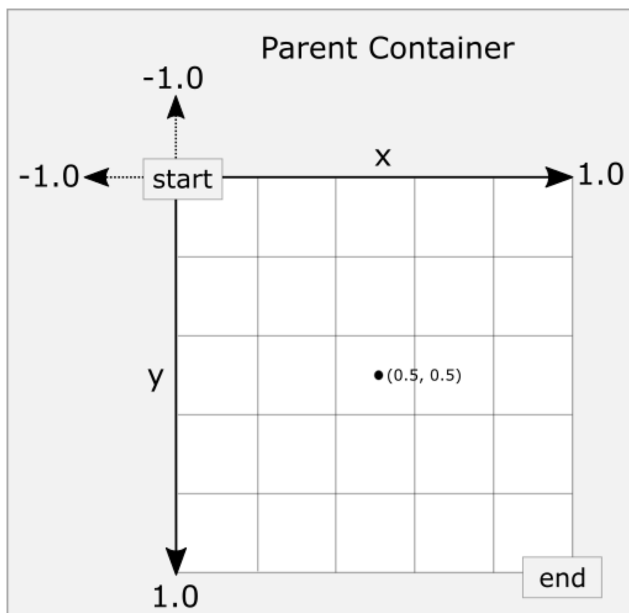

Figure 43-3

- **pathRelative** - The x and y coordinates are relative to the path, where the straight line between the start and end points serves as the graph's X-axis. Once again, coordinates are represented as a percentage (0.0 to 1.0). This is similar to the deltaRelative coordinate space but takes into consideration the angle of the path. Once again coordinates may be negative:

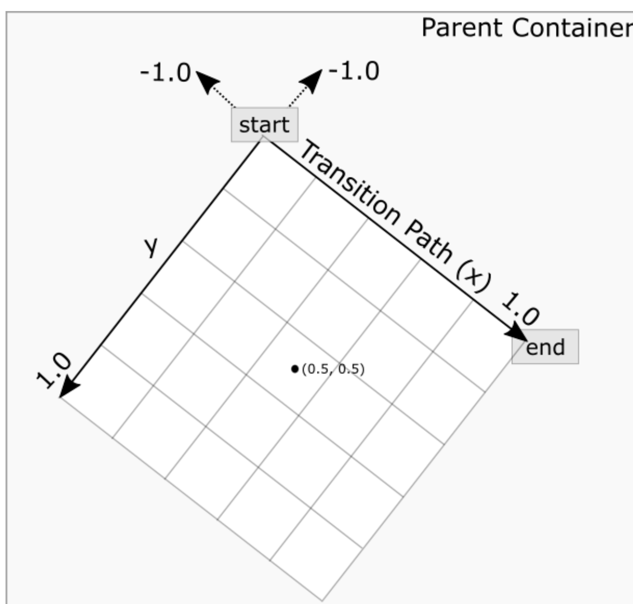

Figure 43-4

As an example, the following ConstraintSets declare start and end points on either side of a device screen. By default, a view transition using these points would move in a straight line across the screen, as illustrated in Figure 43-5:

Figure 43-5

Suppose, however, that the view is required to follow a path similar to that shown in Figure 43-6 below:

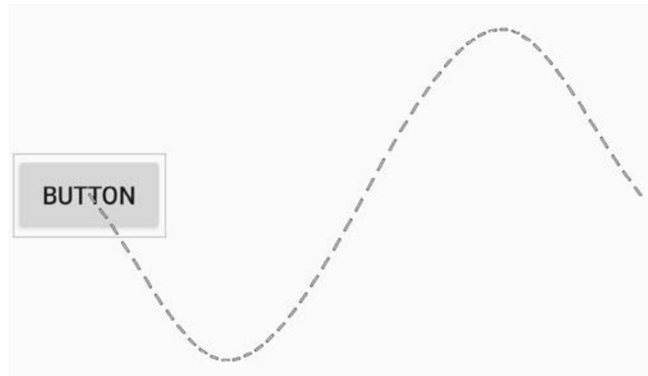

Figure 43-6

To achieve this, keyframe position points could be declared within the transition as follows:

```
<KeyPosition
   motion:motionTarget="@+id/button"
   motion:framePosition="25"
   motion:keyPositionType="pathRelative"
   motion:percentY="0.3"
   motion:percentX="0.25"/>

<KeyPosition
   motion:motionTarget="@+id/button"
   motion:framePosition="75"
   motion:keyPositionType="pathRelative"
   motion:percentY="-0.3"
   motion:percentX="0.75"/>
```

The above elements create keyframe position points 25% and 75% through the path using the pathRelative coordinate system. The first position is placed at coordinates (0.25, 0.3) and the second at (0.75, -0.3). These position keyframes can be visualized as illustrated in Figure 43-7 below:

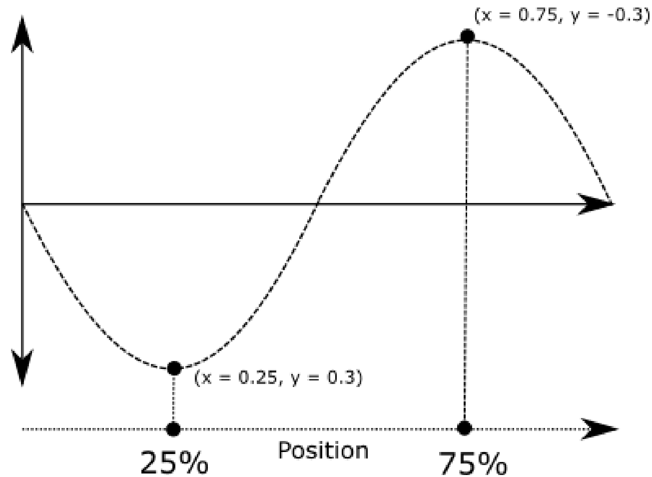

Figure 43-7

43.9 Time Linearity

Without additional settings, the animations outlined above will be performed at a constant speed. To vary the animation speed (for example, so that it accelerates and then decelerates), the transition easing attribute (transitionEasing) can be used within a ConstraintSet or Keyframe.

For complex easing requirements, the linearity can be defined by plotting points on a cubic Bézier curve, for example:

```
.
.
            motion:layout_constraintBottom_toBottomOf="parent"
            motion:transitionEasing="cubic(0.2, 0.7, 0.3, 1)"
            android:rotation="360">
.
.
```

If you are unfamiliar with Bézier curves, consider using the curve generator online at the following URL:

https://cubic-bezier.com/

For most requirements, however, easing can be specified using the built-in *standard*, *accelerate* and *decelerate* values:

```
.
.
            motion:layout_constraintBottom_toBottomOf="parent"
            motion:transitionEasing="decelerate"
            android:rotation="360">
.
.
```

43.10 KeyTrigger

The trigger keyframe (KeyTrigger) allows a method on a view to be called when the animation reaches a specified frame position within the animation timeline. This also takes into consideration the direction of the

animations. For example, different methods can be called depending on whether the animation runs forward or backward. Consider a button that is to be made visible when the animation moves beyond 20% of the timeline. The KeyTrigger would be implemented within the KeyFrameSet of the Transition element as follows using the *onPositiveCross* property:

```
.
.
    <KeyFrameSet>
           <KeyTrigger
               motion:framePosition="20"
               motion:onPositiveCross="show"
               motion:motionTarget="@id/button"/>
.
.
```

Similarly, if the same button is to be hidden when the animation is reversed and drops below 10%, a second key trigger could be added using the *onNegativeCross* property:

```
<KeyTrigger
     motion:framePosition="10"
     motion:onNegativeCross="show"
     motion:motionTarget="@id/button2"/>
```

If the animation is using toggle action, use the *onCross* property:

```
<KeyTrigger
     motion:framePosition="10"
     motion:onCross="show"
     motion:motionTarget="@id/button2"/>
```

43.11 Cycle and Time Cycle Keyframes

While position keyframes can be used to add intermediate state changes into the animation, this would quickly become cumbersome if large numbers of repetitive positions and changes needed to be implemented. For situations where state changes need to be performed repetitively with predictable changes, MotionLayout includes the Cycle and Time Cycle keyframes. The chapter entitled *"A MotionLayout KeyCycle Tutorial"* will cover this topic in detail.

43.12 Starting an Animation from Code

So far in this chapter, we have only looked at controlling an animation using the OnSwipe and OnClick handlers. It is also possible to start an animation from within code by calling methods on the MotionLayout instance. The following code, for example, runs the transition from start to end with a duration of 2000ms for a layout named *motionLayout*:

```
motionLayout.setTransitionDuration(2000);
motionLayout.transitionToEnd();
```

In the absence of additional settings, the start and end states used for the animation will be those declared in the Transition declaration of the MotionScene file. To use specific start and end constraint sets, reference them by id in a call to the *setTransition()* method of the MotionLayout instance:

```
motionLayout.setTransition(R.id.myStart, R.id.myEnd);
motionLayout.transitionToEnd();
```

To monitor the state of an animation while it is running, add a transition listener to the MotionLayout instance

as follows:

```
motionLayout.setTransitionListener(transitionListener);

MotionLayout.TransitionListener transitionListener =
                        new MotionLayout.TransitionListener() {
    @Override
    public void onTransitionStarted(MotionLayout motionLayout,
                                      int startId, int endId) {
            // Called when the transition starts
    }

    @Override
    public void onTransitionChange(MotionLayout motionLayout, int startId,
                                      int endId, float progress) {
            // Called each time a property changes. Track progress value to find
            // current position
    }

    @Override
    public void onTransitionCompleted(MotionLayout motionLayout, int currentId) {
           // Called when the transition is complete
    }

    @Override
    public void onTransitionTrigger(MotionLayout motionLayout, int triggerId,
                                     boolean positive, float progress) {
            // Called when a trigger keyframe threshold is crossed
    }
};
```

43.13 Summary

MotionLayout is a subclass of ConstraintLayout designed specifically to add animation effects to the views in user interface layouts. MotionLayout works by animating the transition of a view between two states defined by start and end constraint sets. Additional animation effects may be added between these start and end points using keyframes.

Animations may be triggered via OnClick or OnSwipe handlers or programmatically via method calls on the MotionLayout instance.

Chapter 87

87. An Overview of Android In-App Billing

n the early days of mobile applications for operating systems such as Android and iOS, the most common method for earning revenue was to charge an upfront fee to download and install the application. Another revenue opportunity was soon introduced by embedding advertising within applications. The most common and lucrative option is to charge the user for purchasing items from within the application after installing it. This typically takes the form of access to a higher level in a game, acquiring virtual goods or currency, or subscribing to premium content in the digital edition of a magazine or newspaper.

Google supports integrating in-app purchasing through the Google Play In-App Billing API and the Play Console. This chapter will provide an overview of in-app billing and outline how to integrate in-app billing into your Android projects. Once these topics have been explored, the next chapter will walk you through creating an example app that includes in-app purchasing features.

87.1 Preparing a Project for In-App Purchasing

Building in-app purchasing into an app will require a Google Play Developer Console account, details of which were covered previously in the *"Creating, Testing and Uploading an Android App Bundle"* chapter. You must also register a Google merchant account. These settings can be found by navigating to *Setup -> Payments profile* in the Play Console. Note that merchant registration is not available in all countries. For details, refer to the following page:

https://support.google.com/googleplay/android-developer/answer/9306917

The app must then be uploaded to the console and enabled for in-app purchasing. However, the console will not activate in-app purchasing support for an app unless the Google Play Billing Library has been added to the module-level *build.gradle.kts* file:

```
dependencies {
.
.
    implementation(libs.billingclient)
.
.
}
```

Once the build file has been modified and the app bundle uploaded to the console, the next step is to add in-app products or subscriptions for the user to purchase.

87.2 Creating In-App Products and Subscriptions

Products and subscriptions are created and managed using the options listed beneath the Monetize section of the Play Console navigation panel, as highlighted in Figure 87-1 below:

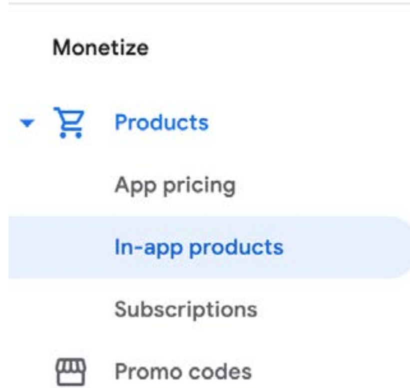

Figure 87-1

Each product or subscription needs an ID, title, description, and pricing information. Purchases fall into the categories of *consumable* (the item must be purchased each time it is required by the user, such as virtual currency in a game), *non-consumable* (only needs to be purchased once by the user, such as content access), and *subscription*-based. Consumable and non-consumable products are collectively referred to as *managed products*.

Subscriptions are useful for selling an item that needs to be renewed regularly, such as access to news content or the premium features of an app. When creating a subscription, a *base plan* specifies the price, renewal period (monthly, annually, etc.), and whether the subscription auto-renews. Users can also be given discount offers and the option of pre-purchasing a subscription.

87.3 Billing Client Initialization

Communication between your app and the Google Play Billing Library is handled by a BillingClient instance. In addition, BillingClient includes a set of methods that can be called to perform both synchronous and asynchronous billing-related activities. When the billing client is initialized, it will need to be provided with a reference to a PurchasesUpdatedListener callback handler. The client will call this handler to notify your app of the results of any purchasing activity. To avoid duplicate notifications, it is recommended to have only one BillingClient instance per app.

A BillingClient instance can be created using the *newBuilder()* method, passing through the current activity or fragment context. The purchase update handler is then assigned to the client via the *setListener()* method:

```
private final PurchasesUpdatedListener purchasesUpdatedListener =
                                     new PurchasesUpdatedListener() {
    @Override
    public void onPurchasesUpdated(BillingResult billingResult,
                                   List<Purchase> purchases) {

        if (billingResult.getResponseCode() ==
                BillingClient.BillingResponseCode.OK
                && purchases != null) {

            // Purchase(s) successful

            for (Purchase purchase : purchases) {
                // Process purchases
```

```
            }
        } else if (billingResult.getResponseCode() ==
                BillingClient.BillingResponseCode.USER_CANCELED) {
            // User cancelled purchase
        } else {
            // handle errors here
        }
    }
};

private BillingClient billingClient = BillingClient.newBuilder(context)
    .setListener(purchasesUpdatedListener)
    .enablePendingPurchases()
    .build();
```

87.4 Connecting to the Google Play Billing Library

After successfully creating the Billing Client, the next step is initializing a connection to the Google Play Billing Library. A call must be made to the *startConnection()* method of the billing client instance to establish this connection. Since the connection is performed asynchronously, a BillingClientStateListener must be implemented to receive a callback indicating whether the connection was successful. Code should also be added to override the *onBillingServiceDisconnected()* method. This is called if the connection to the Billing Library is lost and can be used to report the problem to the user and retry the connection.

Once the setup and connection tasks are complete, the BillingClient instance will make a call to the *onBillingSetupFinished()* method, which can be used to check that the client is ready:

```
billingClient.startConnection(new BillingClientStateListener() {

    @Override
    public void onBillingSetupFinished(
            @NonNull BillingResult billingResult) {

        if (billingResult.getResponseCode() ==
                BillingClient.BillingResponseCode.OK) {
            // Connection successful
        } else {
            // Connection failed
        }
    }

    @Override
    public void onBillingServiceDisconnected() {
        // Existing connection lost
    }
});
```

87.5 Querying Available Products

Once the billing environment is initialized and ready to go, the next step is to request the details of the products or subscriptions available for purchase. This is achieved by making a call to the *queryProductDetailsAsync()* method of the BillingClient and passing through an appropriately configured QueryProductDetailsParams instance containing the product ID and type (ProductType.SUBS for a subscription or ProductType.INAPP for a managed product):

```
QueryProductDetailsParams queryProductDetailsParams =
    QueryProductDetailsParams.newBuilder()
        .setProductList(
            ImmutableList.of(
                QueryProductDetailsParams.Product.newBuilder()
                    .setProductId("one_button_click")
                    .setProductType(BillingClient.ProductType.INAPP)
                    .build()))
        .build();

billingClient.queryProductDetailsAsync(queryProductDetailsParams,
    new ProductDetailsResponseListener() {
        public void onProductDetailsResponse(
                @NonNull BillingResult billingResult,
                @NonNull List<ProductDetails> productDetailsList) {

            if (!productDetailsList.isEmpty()) {
                // Process list of matching products
            } else {
                // No product matches found
            }
        }
    }
);
```

The *queryProductDetailsAsync()* method is passed a ProductDetailsResponseListener handler which, in turn, is called and passed a list of ProductDetail objects containing information about the matching products. For example, we can call methods on these objects to get information such as the product name, title, description, price, and offer details.

87.6 Starting the Purchase Process

Once a product or subscription has been queried and selected for purchase by the user, the purchase process is ready to be launched. We do this by calling the *launchBillingFlow()* method of the BillingClient, passing through as arguments the current activity and a BillingFlowParams instance configured with the ProductDetail object for the purchased item.

```
BillingFlowParams billingFlowParams =
    BillingFlowParams.newBuilder()
        .setProductDetailsParamsList(
            ImmutableList.of(
                BillingFlowParams.ProductDetailsParams.newBuilder()
```

```
                        .setProductDetails(productDetails)
                        .build()
            )
        )
        .build();

billingClient.launchBillingFlow(this, billingFlowParams);
```

The success or otherwise of the purchase operation will be reported via a call to the PurchasesUpdatedListener callback handler outlined earlier in the chapter.

87.7 Completing the Purchase

When purchases are successful, the PurchasesUpdatedListener handler will be passed a list containing a Purchase object for each item. You can verify that the item has been purchased by calling the *getPurchaseState()* method of the Purchase instance as follows:

```
if (purchase.getPurchaseState() == Purchase.PurchaseState.PURCHASED) {
    // Purchase completed.
} else if (purchase.getPurchaseState() == Purchase.PurchaseState.PENDING) {
    // Payment is still pending
}
```

Note that your app will only support pending purchases if a call is made to the *enablePendingPurchases()* method during initialization. A pending purchase will remain so until the user completes the payment process.

When the purchase of a non-consumable item is complete, it must be acknowledged to prevent a refund from being issued to the user. This requires the *purchase token* for the item, which is obtained via a call to the *getPurchaseToken()* method of the Purchase object. This token is used to create an AcknowledgePurchaseParams instance and an AcknowledgePurchaseResponseListener handler. Managed product purchases and subscriptions are acknowledged by calling the BillingClient's *acknowledgePurchase()* method as follows:

```
AcknowledgePurchaseParams acknowledgePurchaseParams =
        AcknowledgePurchaseParams.newBuilder()
                .setPurchaseToken(purchase.getPurchaseToken())
                .build();

AcknowledgePurchaseResponseListener acknowledgePurchaseResponseListener =
                   new AcknowledgePurchaseResponseListener() {

    @Override
    public void onAcknowledgePurchaseResponse(
                      @NonNull BillingResult billingResult) {
        billingClient.acknowledgePurchase(
                            acknowledgePurchaseParams,
                            acknowledgePurchaseResponseListener);
    }
};
```

For consumable purchases, you will need to notify Google Play when the item has been consumed so that it is available to be repurchased by the user. This requires a configured ConsumeParams instance containing a purchase token, a ConsumeResponseListener, and a call to the billing client's *consumeAsync()* method:

```
ConsumeParams consumeParams =
    ConsumeParams.newBuilder()
            .setPurchaseToken(purchase.getPurchaseToken())
            .build();

ConsumeResponseListener listener = new ConsumeResponseListener() {
    @Override
    public void onConsumeResponse(BillingResult billingResult,
                                  @NonNull String purchaseToken) {
        if (billingResult.getResponseCode() ==
                BillingClient.BillingResponseCode.OK) {
            // Purchase consumed successfully
        }
    }
};

billingClient.consumeAsync(consumeParams, listener);
```

87.8 Querying Previous Purchases

When working with in-app billing, checking whether a user has already purchased a product or subscription is a common requirement. A list of all the user's previous purchases of a specific type can be generated by calling the *queryPurchasesAsync()* method of the BillingClient instance and implementing a PurchaseResponseListener. The following code, for example, obtains a list of all previously purchased items that have not yet been consumed:

```
QueryPurchasesParams queryPurchasesParams =
        QueryPurchasesParams.newBuilder()
                .setProductType(BillingClient.ProductType.INAPP)
                .build();

billingClient.queryPurchasesAsync(queryPurchasesParams,
                       new PurchasesResponseListener() {
    @Override
    public void onQueryPurchasesResponse(@NonNull BillingResult billingResult,
              @NonNull List<Purchase> list) {
        // Process list of purchases
    }
});
```

To obtain a list of active subscriptions, change the ProductType value from INAPP to SUBS.

Alternatively, to obtain a list of the most recent purchases for each product, make a call to the BillingClient *queryPurchaseHistoryAsync()* method:

```
QueryPurchaseHistoryParams queryPurchaseHistoryParams =
        QueryPurchaseHistoryParams.newBuilder()
                .setProductType(BillingClient.ProductType.INAPP)
                .build();

billingClient.queryPurchaseHistoryAsync(queryPurchaseHistoryParams,
```

```
                new PurchaseHistoryResponseListener() {

    @Override
    public void onPurchaseHistoryResponse(@NonNull BillingResult billingResult,
                @NonNull List<PurchaseHistoryRecord> list) {
        // Process purchase history
    }
});
```

87.9 Summary

In-app purchases provide a way to generate revenue from within Android apps by selling virtual products and subscriptions to users. This chapter explored managed products and subscriptions and explained the difference between consumable and non-consumable products. In-app purchasing support is added to an app using the Google Play In-app Billing Library. It involves creating and initializing a billing client on which methods are called to perform tasks such as making purchases, listing available products, and consuming existing purchases. The next chapter contains a tutorial demonstrating the addition of in-app purchases to an Android Studio project.

Index

Symbols

<application> 432
<fragment> 247
<fragment> element 247
<menu> 761
<provider> 559
<receiver> 466
<service> 432, 476, 483
.well-known folder 439, 462, 706

A

AbsoluteLayout 122
ACCESS_COARSE_LOCATION permission 500
ACCESS_FINE_LOCATION permission 500
acknowledgePurchase() method 745
ACTION_CREATE_DOCUMENT 622
ACTION_CREATE_INTENT 622
ACTION_DOWN 222
ACTION_MOVE 222
ACTION_OPEN_DOCUMENT intent 614
ACTION_POINTER_DOWN 222
ACTION_POINTER_UP 222
ACTION_UP 222
ACTION_VIEW 457
Active / Running state 96
Activity 83, 99
  adding views in Java code 199
  class 99
  creation 14
  Entire Lifetime 103
  Foreground Lifetime 103
  lifecycle methods 102
  lifecycles 93
  returning data from 436
  state change example 107
  state changes 99
  states 96
  Visible Lifetime 103
Activity Lifecycle 95
Activity Manager 82
ActivityResultLauncher 437
Activity Stack 95
Actual screen pixels 190
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 465
addMarker() method 670
addView() method 193
ADD_VOICEMAIL permission 500
android
  checkableBehavior 763
  exported 433
  gestureColor 240
  layout_behavior property 417
  onClick 249
  orderInCategory 762
  process 433, 483
  uncertainGestureColor 240
Android
  Activity 83
  architecture 79

events 215
intents 84
onClick Resource 215
runtime 80
SDK Packages 5
android.app 80
Android Architecture Components 265
android.content 80
android.content.Intent 435
android.database 80
Android Debug Bridge. *See* ADB
Android Development
System Requirements 3
Android Devices
designing for different 121
android.graphics 81
android.hardware 81
android.intent.action 471
android.intent.action.BOOT_COMPLETED 433
android.intent.action.MAIN 457
android.intent.category.LAUNCHER 457
Android Libraries 80
android.media 81
Android Monitor tool window 32
Android Native Development Kit 81
android.net 81
android.opengl 81
android.os 81
android.permission.RECORD_AUDIO 649
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 614
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 124
android.view.ViewGroup 121, 124
Android Virtual Device. *See* AVD
overview 27
Android Virtual Device Manager 27
android.webkit 81
android.widget 81
AndroidX libraries 784
API Key 661
APK analyzer 738
APK file 731
APK File
analyzing 738
APK Signing 784
APK Wizard dialog 730
app
showAsAction 762
App Architecture
modern 265

AppBar
- anatomy of 415

appbar_scrolling_view_behavior 417
App Bundles 727
- creating 731
- overview 727
- revisions 737
- uploading 734

AppCompatActivity class 100
App Inspector 53
Application
- stopping 32

Application Context 85
Application Framework 82
Application Manifest 85
Application Resources 85
App Link
- Adding Intent Filter 714
- Digital Asset Links file 706, 439
- Intent Filter Handling 714
- Intent Filters 705
- Intent Handling 706
- Testing 718
- URL Mapping 711

App Links 705
- auto verification 438
- autoVerify 439
- overview 705

Apply Changes 207
- Apply Changes and Restart Activity 207
- Apply Code Changes 207
- fallback settings 209
- options 207
- Run App 207
- tutorial 209

applyToActivitiesIfAvailable() method 779
Architecture Components 265
ART 80
assetlinks.json , 706, 439
Attribute Keyframes 342
Audio
- supported formats 647

Audio Playback 647
Audio Recording 647
Auto Blocker 60
Autoconnect Mode 155
Automatic Link Verification 438, 461
autoVerify 439, 714
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 94
Barriers 148
- adding 167
- constrained views 148

Baseline Alignment 147
beginTransaction() method 248
BillingClient 746
- acknowledgePurchase() method 745
- consumeAsync() method 745
- getPurchaseState() method 745
- initialization 742, 753
- launchBillingFlow() method 744
- queryProductDetailsAsync() method 744
- queryPurchasesAsync() method 746

BillingResult 760
- getDebugMessage() 760

Binding Expressions 289
- one-way 289

two-way 290
BIND_JOB_SERVICE permission 433
bindService() method 431, 473, 478
Biometric Authentication 719
callbacks 723
overview 719
tutorial 719
Biometric Prompt 724
BitmapFactory 615
black activity 14
Blank template 125
Blueprint view 153
BODY_SENSORS permission 500
Bound Service 431, 473
adding to a project 474
Implementing the Binder 474
Interaction options 473
BoundService class 475
Broadcast Intent 465
example 468
overview 84, 465
sending 468
Sticky 467
Broadcast Receiver 465
adding to manifest file 470
creation 469
overview 84, 466
BroadcastReceiver class 466
BroadcastReceiver superclass 469
BufferedReader object 625
Build Variants , 54
tool window 54
Bundle class 116
Bundled Notifications 519

C

Calendar permissions 500
CALL_PHONE permission 500
CAMERA permission 500
Camera permissions 500
CameraUpdateFactory class
methods 671
CancellationSignal 724
Canvas class 700
CardView
layout file 395
responding to selection of 403
CardView class 395
CATEGORY_OPENABLE 614
C/C++ Libraries 81
Chain bias 176
chain head 146
chains 146
Chains
creation of 173
Chain style
changing 175
chain styles 146
CheckBox 121
checkSelfPermission() method 504
Circle class 657
Code completion 70
Code Editor
basics 67
Code completion 70
Code Generation 73
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 73
Code Reformatting 75
code samples
download 1
cold boot 44
CollapsingToolbarLayout
example 418
introduction 418
parallax mode 418
pin mode 418

setting scrim color 421
setting title 421
with image 418
Color class 701
COLOR_MODE_COLOR 676, 696
COLOR_MODE_MONOCHROME 676, 696
Common Gestures 229
detection 229
Component tree 17
Constraint Bias 145
adjusting 159
ConstraintLayout
advantages of 151
Availability 152
Barriers 148
Baseline Alignment 147
chain bias 176
chain head 146
chains 146
chain styles 146
Constraint Bias 145
Constraints 143
conversion to 171
convert to MotionLayout 349
deleting constraints 158
guidelines 165
Guidelines 148
manual constraint manipulation 155
Margins 144, 159
Opposing Constraints 144, 161
overview of 143
Packed chain 147, 176
ratios 151, 177
Spread chain 146
Spread inside 176
Spread inside chain 146
tutorial 181
using in Android Studio 153
Weighted chain 146, 176
Widget Dimensions 147, 163
Widget Group Alignment 169
ConstraintLayout chains
creation of 173
in layout editor 173
ConstraintLayout Chain style
changing 175
Constraints
deleting 158
ConstraintSet
addToHorizontalChain() method 196
addToVerticalChain() method 196
alignment constraints 195
apply to layout 194
applyTo() method 194
centerHorizontally() method 195
centerVertically() method 195
chains 195
clear() method 196
clone() method 195
connect() method 194
connect to parent 194
constraint bias 195
copying constraints 195
create 194
create connection 194
createHorizontalChain() method 195
createVerticalChain() method 195
guidelines 196
removeFromHorizontalChain() method 196
removeFromVerticalChain() method 196
removing constraints 196
rotation 197
scaling 196
setGuidelineBegin() method 196
setGuidelineEnd() method 196
setGuidelinePercent() method 196
setHorizonalBias() method 195
setRotationX() method 197
setRotationY() method 197
setScaleX() method 196
setScaleY() method 196
setTransformPivot() method 197
setTransformPivotX() method 197
setTransformPivotY() method 197

setVerticalBias() method 195
sizing constraints 195
tutorial 199
view IDs 201
ConstraintSet class 193, 194
Constraint Sets 194
ConstraintSets
configuring 338
consumeAsync() method 745
ConsumeParams 757
ConsumeResponseListener 745
Contacts permissions 500
container view 121
Content Provider 82, 557, 575
<provider> 559
accessing 575
Authority 563
client tutorial 575
ContentProvider class 557
Content Resolver 558
ContentResolver 571
content URI 558
Content URI 563, 575
ContentValues 565
delete() 558, 569
getType() 558
insert() 557, 565
onCreate() 557, 565
overview 85
query() 557, 566
tutorial 561
update() 558, 567
UriMatcher 564
UriMatcher class 558
ContentProvider class 557
Content Resolver 558
getContentResolver() 558
ContentResolver 571
getContentResolver() 558
content URI 558
Content URI 558, 563
ContentValues 565
Context class 85
CoordinatorLayout 122, 417
createPrintDocumentAdapter() method 691
Custom Attribute 339
Custom Document Printing 679, 691
Custom Gesture
recognition 235
Custom Print Adapter
implementation 693
Custom Print Adapters 691
Custom Theme
building 773
Cycle Editor 367
Cycle Keyframe 347
Cycle Keyframes
overview 363

D

dangerous permissions
list of 500
Dark Theme 32
enable on device 32
Data Access Object (DAO) 580
Data Access Objects (DAO) 584
Database Inspector 587, 610
live updates 611
SQL query 611
Database Rows 544
Database Schema 543
Database Tables 543
Data binding
binding expressions 289
Data Binding 267
binding classes 288
enabling 294
event and listener binding 290
key components 285
overview 285
tutorial 293
with LiveData 267
DDMS 32
Debugging

enabling on device 59
debug.keystore file 439, 461
DefaultLifecycleObserver 306, 309
deltaRelative 344
Density-independent pixels 189
Density Independent Pixels
converting to pixels 205
Device Definition
custom 139
Device File Explorer 54
device frame 36
Device Mirroring 65
enabling 65
device pairing 63
Digital Asset Links file 706, 439, 439
Direct Reply Input 530
document provider 613
dp 189
Dynamic Colors
applyToActivitiesIfAvailable() method 779
enabling in Android 779
Dynamic State 101
saving 115

E

Empty Process 95
Empty template 125
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
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 745
enabling ADB support 59
Event Handling 215
example 216
Event Listener 217
Event Listeners 216
Events
consuming 219
execSQL() 552
explicit
intent 84
explicit intent 435
Explicit Intent 435
Extended Control
options 41

F

Files
switching between 68
findPointerIndex() method 222
findViewById() 87
Fingerprint
emulation 44
Fingerprint authentication
device configuration 720
permission 720
steps to implement 719
Fingerprint Authentication
overview 719
tutorial 719
FLAG_INCLUDE_STOPPED_PACKAGES 465
flexible space area 415
floating action button 14, 126
changing appearance of 378
margins 376

removing 127
sizes 376
Foldable Devices 104
multi-resume 104
Foldable Emulator 536
Foldables 535
Foreground Process 94
Forward-geocoding 663
Fragment
creation 245
event handling 249
XML file 246
FragmentActivity class 101
Fragment Communication 250
Fragments 245
adding in code 248
duplicating 384
example 253
overview 245
FragmentStateAdapter class 387
FrameLayout 122

G

Geocoder object 664
Geocoding 662
Gesture Builder Application 235
building and running 235
Gesture Detector class 229
GestureDetectorCompat 232
instance creation 232
GestureDetectorCompat class 229
GestureDetector.OnDoubleTapListener 229, 230
GestureDetector.OnGestureListener 230
GestureLibrary 235
GestureOverlayView 235
configuring color 240
configuring multiple strokes 240
GestureOverlayView class 235
GesturePerformedListener 235
Gestures
interception of 241
Gestures File
creation 236
extract from SD card 236
loading into application 238
GET_ACCOUNTS permission 500
getAction() method 471
getContentResolver() 558
getDebugMessage() 760
getFromLocation() method 664
getId() method 194
getIntent() method 436
getItemId() method 763
getPointerCount() method 222
getPointerId() method 222
getPurchaseState() method 745
getService() method 477
getWritableDatabase() 553
GNU/Linux 80
Google Cloud
billing account 658
new project 659
Google Cloud Print 674
Google Drive 614
printing to 674
GoogleMap 657
map types 667
GoogleMap.MAP_TYPE_HYBRID 667
GoogleMap.MAP_TYPE_NONE 667
GoogleMap.MAP_TYPE_NORMAL 667
GoogleMap.MAP_TYPE_SATELLITE 667
GoogleMap.MAP_TYPE_TERRAIN 667
Google Maps Android API 657
Controlling the Map Camera 671
displaying controls 668
Map Markers 670
overview 657
Google Maps SDK 657
API Key 661
Credentials 661
enabling 660
Maps SDK for Android 661
Google Play App Signing 730
Google Play Console 751

Creating an in-app product 751
License Testers 752
Google Play Developer Console 728
Gradle
APK signing settings 788
Build Variants 784
command line tasks 789
dependencies 783
Manifest Entries 784
overview 783
sensible defaults 783
Gradle Build File
top level 785
Gradle Build Files
module level 786
gradle.properties file 784
GridLayout 122
GridLayoutManager 393

H

HAL 80
Handler class 482
Hardware Abstraction Layer 80
HP Print Services Plugin 673
HTML printing 677
HTML Printing
example 681

I

IBinder 431, 475
IBinder object 473, 483
Image Printing 676
implicit
intent 84
implicit intent 435
Implicit Intent 437
Implicit Intents
example 453
importance hierarchy 93
in 189
INAPP 746
In-App Products 741
In-App Purchasing 749
acknowledgePurchase() method 745
BillingClient 742
BillingResult 760
consumeAsync() method 745
ConsumeParams 757
ConsumeResponseListener 745
Consuming purchases 757
enablePendingPurchases() method 745
getPurchaseState() method 745
launchBillingFlow() method 744
Libraries 749
newBuilder() method 742
onBillingServiceDisconnected() callback 754
onBillingServiceDisconnected() method 743
onBillingSetupFinished() listener 754
onProductDetailsResponse() callback 754
Overview 741
ProductDetail 744
ProductDetails 755
products 741
ProductType 746
ProductType.INAPP 746
ProductType.SUBS 746
Purchase Flow 756
PurchaseResponseListener 746
PurchasesUpdatedListener 745
PurchaseUpdatedListener 756
purchase updates 756
queryProductDetailsAsync() 754
queryProductDetailsAsync() method 744
queryPurchasesAsync() 758
queryPurchasesAsync() method 746
runOnUiThread() 755
subscriptions 741
tutorial 749
In-Memory Database 587
Intent 84
explicit 84
implicit 84
Intent Availability
checking for 442

Intent.CATEGORY_OPENABLE 622
Intent Filters 438
  App Link 705
Intents 435
  ActivityResultLauncher 437
  overview 435
  registerForActivityResult() 450
Intent Service 431
Intent URL 455

J

Java Native Interface 81
Jetpack 265
  overview 265
JobIntentService 431
  BIND_JOB_SERVICE permission 433
  onHandleWork() method 431

K

KeyAttribute 342
Keyboard Shortcuts 56
KeyCycle 363
  Cycle Editor 367
  tutorial 363
Keyframe 356
Keyframes 342
KeyFrameSet 372
KeyPosition 343
  deltaRelative 344
  parentRelative 343
  pathRelative 344
Keystore File
  creation 730
KeyTimeCycle 363
keytool 439
KeyTrigger 346
Killed state 96

L

launchBillingFlow() method 744
layout_collapseMode
  parallax 420
  pin 420
layout_constraintDimentionRatio 178
layout_constraintHorizontal_bias 176
layout_constraintVertical_bias 176
layout editor
  ConstraintLayout chains 173
Layout Editor 16, 181
  Autoconnect Mode 155
  code mode 132
  Component Tree 130
  design mode 129
  device screen 130
  example project 181
  Inference Mode 155
  palette 130
  properties panel 130
  Sample Data 138
  Setting Properties 133
  toolbar 130
  user interface design 181
  view conversion 137
Layout Editor Tool
  changing orientation 17
  overview 129
Layout Inspector 54
Layout Managers 121
LayoutResultCallback object 696
Layouts 121
layout_scrollFlags
  enterAlwaysCollapsed mode 417
  enterAlways mode 417
  exitUntilCollapsed mode 417
  scroll mode 417
Layout Validation 140
libc 81
libs.versions.toml file 212
License Testers 752
Lifecycle
  awareness 305
  components 268
  owners 305
  states and events 307

tutorial 309
Lifecycle-Aware Components 305
Lifecycle Methods 102
Lifecycle Observer 309
creating a 309
Lifecycle Owner
creating a 311
Lifecycles
modern 268
LinearLayout 122
LinearLayoutManager 393
LinearLayoutManager layout 402
Linux Kernel 80
list devices 59
LiveData 266, 279
adding to ViewModel 279
observer 281
tutorial 279
Live Templates 76
Local Bound Service 473
example 473
Location Manager 82
Location permission 500
Logcat
tool window 54
LogCat
enabling 111

M

MANAGE_EXTERNAL_STORAGE 501
adb enabling 501
testing 501
Manifest File
permissions 457
Maps 657
MapView 657
adding to a layout 664
Marker class 657
Master/Detail Flow
creation 424
two pane mode 423
match_parent properties 189
Material design 375
Material Design 2 771
Material Design 2 Theming 771
Material Design 3 771
Material Theme Builder 773
Material You 771
MediaController
adding to VideoView instance 631
MediaController class 628
methods 628
MediaPlayer class 647
methods 647
MediaRecorder class 647
methods 648
recording audio 648
Memory Indicator 69
Menu Editor 764
Menu Item Selections 762
Menus 761
menu editor 764
Messenger object 483
Microphone
checking for availability 650
Microphone permissions 500
mm 189
MotionEvent 221, 222, 243
getActionMasked() 222
MotionLayout 337
arc motion 342
Attribute Keyframes 342
ConstraintSets 338
Custom Attribute 358
Custom Attributes 339
Cycle Editor 367
Editor 349
KeyAttribute 342
KeyCycle 363
Keyframes 342
KeyFrameSet 372
KeyPosition 343
KeyTimeCycle 363
KeyTrigger 346

OnClick 341, 354
OnSwipe 341
overview 337
Position Keyframes 343
previewing animation 354
Trigger Keyframe 346
Tutorial 349
MotionScene
ConstraintSets 338
Custom Attributes 339
file 338
overview 337
transition 338
moveCamera() method 671
multiple devices
testing app on 31
Multiple Touches
handling 222
multi-resume 104
Multi-Touch
example 222
Multi-touch Event Handling 221
Multi-Window
attributes 539
Multi-Window Mode
detecting 540
entering 537
launching activity into 541
Multi-Window Notifications 540
multi-window support 104
Multi-Window Support
enabling 538
My Location Layer 657

N

Navigation 315
adding destinations 324
overview 315
pass data with safeargs 332
passing arguments 320
stack 315
tutorial 321
Navigation Action
triggering 319
Navigation Architecture Component 315
Navigation Component
tutorial 321
Navigation Controller
accessing 319
Navigation Graph 318, 322
adding actions 328
creating a 322
Navigation Host 316
declaring 323
newBuilder() method 742
normal permissions 499
Notification
adding actions 518
Direct Reply Input 530
issuing a basic 514
launch activity from a 516
PendingIntent 526
Reply Action 528
updating direct reply 531
Notifications
bundled 519
overview 507
Notifications Manager 82

O

Observer
implementing a LiveData 281
onAttach() method 250
onBillingServiceDisconnected() callback 754
onBillingServiceDisconnected() method 743
onBillingSetupFinished() listener 754
onBind() method 432, 473, 481
onBindViewHolder() method 401
OnClick 341
onClickListener 216, 217, 220
onClick() method 215
onCreateContextMenuListener 216
onCreate() method 94, 102, 432
onCreateOptionsMenu() method 762

onCreateView() method 103
onDestroy() method 102, 432
onDoubleTap() method 229
onDown() method 229
onFling() method 229
onFocusChangeListener 216
OnFragmentInteractionListener
  implementation 329
onGesturePerformed() method 235
onHandleWork() method 432
onKeyListener 216
onLayoutFailed() method 696
onLayoutFinished() method 697
onLongClickListener 216
onLongClick() method 219
onLongPress() method 229
onMapReady() method 666
onOptionsItemSelected() method 762
onOptionsItemsSelected() method 767
onPageFinished() callback 682
onPause() method 102
onProductDetailsResponse() callback 754
onReceive() method 94, 466, 467, 469
onRequestPermissionsResult() method 503, 654, 512, 524
onRestart() method 102
onRestoreInstanceState() method 103
onResume() method 94, 102
onSaveInstanceState() method 103
onScaleBegin() method 241
onScaleEnd() method 241
onScale() method 241
onScroll() method 229
OnSeekBarChangeListener 260
onServiceConnected() method 473, 477, 484
onServiceDisconnected() method 473, 477, 484
onShowPress() method 229
onSingleTapUp() method 229
onStartCommand() method 432
onStart() method 102
onStop() method 102
onTouchEvent() method 229, 241
onTouchListener 216
onTouch() method 221
onUpgrade() 552
onViewCreated() method 103
onViewStatusRestored() method 103
openFileDescriptor() method 614
OpenJDK 3
Overflow Menu 761
  creation 761
  displaying 762
  overview 761
  XML file 761
Overflow Menus
  Checkable Item Groups 763

P

Package Explorer 15
Package Manager 82
PackageManager class 650
PackageManager.FEATURE_MICROPHONE 650
PackageManager.PERMISSION_DENIED 501
PackageManager.PERMISSION_GRANTED 501
Package Name 14
Packed chain 147, 176
PageRange 698, 699
Paint class 701
parentRelative 343
parent view 123
pathRelative 344
Paused state 96
PdfDocument 679
PdfDocument.Page 691, 698
PendingIntent class 526
Permission
  checking for 501
permissions
  normal 499
Persistent State 101
Phone permissions 500
picker 613
Pinch Gesture
  detection 241
  example 241

Pinch Gesture Recognition 235
Position Keyframes 343
POST_NOTIFICATIONS permission 500, 524
PrintAttributes 696
PrintDocumentAdapter 679, 691
Printing
  color 676
  monochrome 676
Printing framework
  architecture 673
Printing Framework 673
Print Job
  starting 702
PrintManager service 683
Problems
  tool window 54, 55
process
  priority 93
  state 93
PROCESS_OUTGOING_CALLS permission 500
Process States 93
ProductDetail 744
ProductDetails 755
ProductType 746
Profiler
  tool window 55
ProgressBar 121
proguard-rules.pro file 788
ProGuard Support 784
Project Name 14
Project tool window 15, 53
pt 189
PurchaseResponseListener 746
PurchasesUpdatedListener 745
PurchaseUpdatedListener 756
putExtra() method 435, 465
px 190

Q

queryProductDetailsAsync() 754
queryPurchasesAsync() 758
quickboot snapshot 44
Quick Documentation 75

R

RadioButton 121
ratios 177
READ_CALENDAR permission 500
READ_CALL_LOG permission 500
READ_CONTACTS permission 500
READ_EXTERNAL_STORAGE permission 501
READ_PHONE_STATE permission 500
READ_SMS permission 500
RECEIVE_MMS permission 500
RECEIVE_SMS permission 500
RECEIVE_WAP_PUSH permission 500
Recent Files Navigation 56
RECORD_AUDIO permission 500
Recording Audio
  permission 649
RecyclerView 393
  adding to layout file 394
  GridLayoutManager 393
  initializing 402
  LinearLayoutManager 393
  StaggeredGridLayoutManager 393
RecyclerView Adapter
  creation of 400
RecyclerView.Adapter 394, 400
  getItemCount() method 394
  onBindViewHolder() method 394
  onCreateViewHolder() method 394
RecyclerView.ViewHolder
  getAdapterPosition() method 404
registerForActivityResult() method 436, 450
registerReceiver() method 467
RelativeLayout 122
releasePersistableUriPermission() method 617
Release Preparation 727
Remote Bound Service 481
  client communication 481
  implementation 481
  manifest file declaration 483
RemoteInput.Builder() method 526

RemoteInput Object 526
Remote Service
- launching and binding 484
- sending a message 485

Repository
- tutorial 597

Repository Modules 268
Resizable Emulator 46
Resource
- string creation 20

Resource File 22
Resource Management 93
Resource Manager 53, 82
result receiver 467
Reverse-geocoding 663
Reverse Geocoding 662
Room
- Data Access Object (DAO) 580
- entities 580, 581
- In-Memory Database 587
- Repository 580

Room Database 580
- tutorial 597

Room Database Persistence 579
Room Persistence Library 548, 579
root element 121
root view 123
Run
- tool window 53

Running Devices
- tool window 65

runOnUiThread() 755

S

safeargs 332
Sample Data 138, 407
- tutorial 407

Saved State 267, 301
SavedStateHandle 302, 303
- contains() method 303
- keys() method 303
- remove() method 303

Saved State module 301
SavedStateViewModelFactory 302
ScaleGestureDetector class 241
Scale-independent 189
SDK Packages 5
Secure Sockets Layer (SSL) 81
SeekBar 253
sendBroadcast() method 465, 467
sendOrderedBroadcast() method 465, 467
SEND_SMS permission 500
sendStickyBroadcast() method 465
Sensor permissions 500
Service
- anatomy 432
- launch at system start 433
- manifest file entry 432
- overview 84
- run in separate process 433

ServiceConnection class 484
Service Process 94
Service Restart Options 432
setAudioEncoder() method 648
setAudioSource() method 648
setBackgroundColor() 194
setCompassEnabled() method 668
setContentView() method 193, 199
setId() method 194
setMyLocationButtonEnabled() method 669
setOnClickListener() method 215, 217
setOnDoubleTapListener() method 229, 232
setOutputFile() method 648
setOutputFormat() method 648
setResult() method 437
setRotateGesturesEnabled() method 669
setScrollGesturesEnabled() method 669
setText() method 118
setTiltGesturesEnabled() method 669
settings.gradle file 784
settings.gradle.kts file 784
setTransition() 347
setVideoSource() method 648
setZoomControlsEnabled() method 668, 669

SHA-256 certificate fingerprint 439
shouldOverrideUrlLoading() method 682
SimpleOnScaleGestureListener 241
SimpleOnScaleGestureListener class 243
SMS permissions 500
Snackbar 375, 376, 377
Snapshots
  emulator 43
sp 189
Spread chain 146
Spread inside 176
Spread inside chain 146
SQL 544
SQL CREATE 552
SQLite 543
  AVD command-line use 545
  Columns and Data Types 543
  overview 544
  Primary keys 544
  tutorial 549
SQLiteDatabase 552
SQLiteOpenHelper 550, 551
SQL SELECT 553, 554
StaggeredGridLayoutManager 393
startActivity() method 435
startForeground() method 94
START_NOT_STICKY 432
START_REDELIVER_INTENT 432
START_STICKY 432
State
  restoring 118
State Change
  handling 97
Statement Completion 72
Status Bar Widgets 69
  Memory Indicator 69
Sticky Broadcast Intents 467
Stopped state 96
Storage Access Framework 613
  ACTION_CREATE_DOCUMENT 614
  ACTION_OPEN_DOCUMENT 614
  deleting a file 617
  example 619
  file creation 622
  file filtering 614
  file reading 615
  file writing 616
  intents 614
  MIME Types 615
  Persistent Access 617
  picker 613
Storage permissions 501
StringBuilder object 625
strings.xml file 24
Structure
  tool window 55
Structured Query Language 544
Structure tool window 55
SUBS 746
subscriptions 741
SupportMapFragment class 657
Switcher 56
System Broadcasts 471
system requirements 3

T

TabLayout
  adding to layout 385
  app
    tabGravity property 390
    tabMode property 390
  example 382
  fixed mode 389
  getItemCount() method 381
  overview 381
TableLayout 122, 589
TableRow 589
Telephony Manager 82
Templates
  blank vs. empty 125
Terminal
  tool window 54
Theme
  building a custom 773

Theming 771
  tutorial 775
Time Cycle Keyframes 347
TODO
  tool window 55
ToolbarListener 250
tools
  layout 247
Tool window bars 52
Tool windows 52
Touch Actions 222
Touch Event Listener
  implementation 223
Touch Events
  intercepting 221
Touch handling 221

U

UiSettings class 657
unbindService() method 431
unregisterReceiver() method 467
upload key 730
UriMatcher 558, 564
UriMatcher class 558
URL Mapping 711
USB connection issues
  resolving 62
USE_BIOMETRIC 720
user interface state 101
USE_SIP permission 500

V

Version catalog 211
  dependencies 213
  libraries 212
  libs.versions.toml file 212
  plugins 212
  versions 212
Video Playback 627
VideoView class 627
  methods 627
  supported formats 627
view bindings
  enabling 88
  using 88
View class
  setting properties 201
view conversion 137
ViewGroup 121
View Groups 121
View Hierarchy 123
ViewHolder class 394
  sample implementation 401
ViewModel
  adding LiveData 279
  data access 276
  overview 266
  saved state 301
  Saved State 267, 301
  tutorial 271
ViewModelProvider 274
ViewModel Saved State 301
ViewPager
  adding to layout 385
  example 382
Views 121
  Java creation 193
View System 82
Virtual Device Configuration dialog 28
Virtual Sensors 43
Visible Process 94

W

WebViewClient 677, 682
WebView view 455
Weighted chain 146, 176
Welcome screen 49
Widget Dimensions 147
Widget Group Alignment 169
Widgets palette 182
WiFi debugging 63
Wireless debugging 63
Wireless pairing 63
wrap_content properties 191

WRITE_CALENDAR permission 500
WRITE_CALL_LOG permission 500
WRITE_CONTACTS permission 500
WRITE_EXTERNAL_STORAGE permission 501

X

XML Layout File
manual creation 189
vs. Java Code 193