Android Studio Giraffe Essentials

Java Edition

Android Studio Giraffe Essentials – Java Edition

ISBN-13: 978-1-951442-74-3

© 2023 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

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 The Android Studio setup wizard 5
2.5 Installing additional Android SDK packages 6
2.6 Installing the Android SDK Command-line Tools 9
2.6.1 Windows 8.1 10
2.6.2 Windows 10 11
2.6.3 Windows 11 11
2.6.4 Linux 11
2.6.5 macOS 11
2.7 Android Studio memory management 11
2.8 Updating Android Studio and the SDK 12
2.9 Summary 13

3. Creating an Example Android App in Android Studio 15

3.1 About the Project 15
3.2 Creating a New Android Project 15
3.3 Creating an Activity 16
3.4 Defining the Project and SDK Settings 16
3.5 Enabling the New Android Studio UI 17
3.6 Modifying the Example Application 18
3.7 Modifying the User Interface 19
3.8 Reviewing the Layout and Resource Files 25
3.9 Adding Interaction 28
3.10 Summary 29

4. Creating an Android Virtual Device (AVD) in Android Studio 31

4.1 About Android Virtual Devices 31
4.2 Starting the Emulator 33
4.3 Running the Application in the AVD 34
4.4 Running on Multiple Devices 35
4.5 Stopping a Running Application 36
4.6 Supporting Dark Theme 36
4.7 Running the Emulator in a Separate Window 37

4.8 Enabling the Device Frame 40
4.9 Summary 41

5. Using and Configuring the Android Studio AVD Emulator 43

5.1 The Emulator Environment 43
5.2 Emulator Toolbar Options 43
5.3 Working in Zoom Mode 45
5.4 Resizing the Emulator Window 45
5.5 Extended Control Options 45
5.5.1 Location 46
5.5.2 Displays 46
5.5.3 Cellular 46
5.5.4 Battery 46
5.5.5 Camera 46
5.5.6 Phone 46
5.5.7 Directional Pad 46
5.5.8 Microphone 46
5.5.9 Fingerprint 46
5.5.10 Virtual Sensors 47
5.5.11 Snapshots 47
5.5.12 Record and Playback 47
5.5.13 Google Play 47
5.5.14 Settings 47
5.5.15 Help 47
5.6 Working with Snapshots 47
5.7 Configuring Fingerprint Emulation 48
5.8 The Emulator in Tool Window Mode 50
5.9 Creating a Resizable Emulator 50
5.10 Summary 51

6. A Tour of the Android Studio User Interface 53

6.1 The Welcome Screen 53
6.2 The Menu Bar 54
6.3 The Main Window 54
6.4 The Tool Windows 56
6.5 The Tool Window Menus 59
6.6 Android Studio Keyboard Shortcuts 59
6.7 Switcher and Recent Files Navigation 60
6.8 Changing the Android Studio Theme 61
6.9 Summary 62

7. Testing Android Studio Apps on a Physical Android Device 63

7.1 An Overview of the Android Debug Bridge (ADB) 63
7.2 Enabling USB Debugging ADB on Android Devices 63
7.2.1 macOS ADB Configuration 64
7.2.2 Windows ADB Configuration 65
7.2.3 Linux adb Configuration 66
7.3 Resolving USB Connection Issues 66
7.4 Enabling Wireless Debugging on Android Devices 67
7.5 Testing the adb Connection 69
7.6 Device Mirroring 69

7.7 Summary 69

8. The Basics of the Android Studio Code Editor 71

8.1 The Android Studio Editor 71
8.2 Splitting the Editor Window 74
8.3 Code Completion 74
8.4 Statement Completion 76
8.5 Parameter Information 76
8.6 Parameter Name Hints 76
8.7 Code Generation 77
8.8 Code Folding 78
8.9 Quick Documentation Lookup 79
8.10 Code Reformatting 79
8.11 Finding Sample Code 80
8.12 Live Templates 80
8.13 Summary 81

9. An Overview of the Android Architecture 83

9.1 The Android Software Stack 83
9.2 The Linux Kernel 84
9.3 Android Runtime – ART 84
9.4 Android Libraries 84
9.4.1 C/C++ Libraries 85
9.5 Application Framework 85
9.6 Applications 86
9.7 Summary 86

10. The Anatomy of an Android App 87

10.1 Android Activities 87
10.2 Android Fragments 87
10.3 Android Intents 88
10.4 Broadcast Intents 88
10.5 Broadcast Receivers 88
10.6 Android Services 88
10.7 Content Providers 89
10.8 The Application Manifest 89
10.9 Application Resources 89
10.10 Application Context 89
10.11 Summary 89

11. An Overview of Android View Binding 91

11.1 Find View by Id 91
11.2 View Binding 91
11.3 Converting the AndroidSample project 92
11.4 Enabling View Binding 92
11.5 Using View Binding 92
11.6 Choosing an Option 93
11.7 View Binding in the Book Examples 93
11.8 Migrating a Project to View Binding 94
11.9 Summary 94

12. Understanding Android Application and Activity Lifecycles 97

12.1 Android Applications and Resource Management 97
12.2 Android Process States 97
12.2.1 Foreground Process 98
12.2.2 Visible Process 98
12.2.3 Service Process 98
12.2.4 Background Process 98
12.2.5 Empty Process 99
12.3 Inter-Process Dependencies 99
12.4 The Activity Lifecycle 99
12.5 The Activity Stack 99
12.6 Activity States 100
12.7 Configuration Changes 100
12.8 Handling State Change 101
12.9 Summary 101

13. Handling Android Activity State Changes 103

13.1 New vs. Old Lifecycle Techniques 103
13.2 The Activity and Fragment Classes 103
13.3 Dynamic State vs. Persistent State 105
13.4 The Android Lifecycle Methods 106
13.5 Lifetimes 107
13.6 Foldable Devices and Multi-Resume 108
13.7 Disabling Configuration Change Restarts 108
13.8 Lifecycle Method Limitations 109
13.9 Summary 109

14. Android Activity State Changes by Example 111

14.1 Creating the State Change Example Project 111
14.2 Designing the User Interface 112
14.3 Overriding the Activity Lifecycle Methods 113
14.4 Filtering the Logcat Panel 115
14.5 Running the Application 117
14.6 Experimenting with the Activity 117
14.7 Summary 118

15. Saving and Restoring the State of an Android Activity 119

15.1 Saving Dynamic State 119
15.2 Default Saving of User Interface State 119
15.3 The Bundle Class 120
15.4 Saving the State 121
15.5 Restoring the State 122
15.6 Testing the Application 122
15.7 Summary 122

16. Understanding Android Views, View Groups and Layouts 125

16.1 Designing for Different Android Devices 125
16.2 Views and View Groups 125
16.3 Android Layout Managers 125
16.4 The View Hierarchy 127
16.5 Creating User Interfaces 128
16.6 Summary 128

17. A Guide to the Android Studio Layout Editor Tool 129

17.1 Basic vs. Empty Views Activity Templates 129
17.2 The Android Studio Layout Editor 133
17.3 Design Mode 133
17.4 The Palette 134
17.5 Design Mode and Layout Views 135
17.6 Night Mode 136
17.7 Code Mode 136
17.8 Split Mode 137
17.9 Setting Attributes 138
17.10 Transforms 139
17.11 Tools Visibility Toggles 140
17.12 Converting Views 141
17.13 Displaying Sample Data 142
17.14 Creating a Custom Device Definition 143
17.15 Changing the Current Device 143
17.16 Layout Validation 144
17.17 Summary 145

18. A Guide to the Android ConstraintLayout 147

18.1 How ConstraintLayout Works 147
18.1.1 Constraints 147
18.1.2 Margins 148
18.1.3 Opposing Constraints 148
18.1.4 Constraint Bias 149
18.1.5 Chains 150
18.1.6 Chain Styles 150
18.2 Baseline Alignment 151
18.3 Configuring Widget Dimensions 151
18.4 Guideline Helper 152
18.5 Group Helper 152
18.6 Barrier Helper 152
18.7 Flow Helper 154
18.8 Ratios 155
18.9 ConstraintLayout Advantages 155
18.10 ConstraintLayout Availability 156
18.11 Summary 156

19. A Guide to Using ConstraintLayout in Android Studio 157

19.1 Design and Layout Views 157
19.2 Autoconnect Mode 159
19.3 Inference Mode 159
19.4 Manipulating Constraints Manually 159
19.5 Adding Constraints in the Inspector 161
19.6 Viewing Constraints in the Attributes Window 161
19.7 Deleting Constraints 162
19.8 Adjusting Constraint Bias 163
19.9 Understanding ConstraintLayout Margins 163
19.10 The Importance of Opposing Constraints and Bias 165
19.11 Configuring Widget Dimensions 167

19.12 Design Time Tools Positioning 168
19.13 Adding Guidelines 169
19.14 Adding Barriers 171
19.15 Adding a Group 172
19.16 Working with the Flow Helper 173
19.17 Widget Group Alignment and Distribution 173
19.18 Converting other Layouts to ConstraintLayout 175
19.19 Summary 175

20. Working with ConstraintLayout Chains and Ratios in Android Studio 177

20.1 Creating a Chain 177
20.2 Changing the Chain Style 179
20.3 Spread Inside Chain Style 180
20.4 Packed Chain Style 180
20.5 Packed Chain Style with Bias 180
20.6 Weighted Chain 180
20.7 Working with Ratios 181
20.8 Summary 183

21. An Android Studio Layout Editor ConstraintLayout Tutorial 185

21.1 An Android Studio Layout Editor Tool Example 185
21.2 Preparing the Layout Editor Environment 185
21.3 Adding the Widgets to the User Interface 186
21.4 Adding the Constraints 189
21.5 Testing the Layout 191
21.6 Using the Layout Inspector 191
21.7 Summary 192

22. Manual XML Layout Design in Android Studio 193

22.1 Manually Creating an XML Layout 193
22.2 Manual XML vs. Visual Layout Design 196
22.3 Summary 196

23. Managing Constraints using Constraint Sets 197

23.1 Java Code vs. XML Layout Files 197
23.2 Creating Views 197
23.3 View Attributes 198
23.4 Constraint Sets 198
23.4.1 Establishing Connections 198
23.4.2 Applying Constraints to a Layout 198
23.4.3 Parent Constraint Connections 198
23.4.4 Sizing Constraints 199
23.4.5 Constraint Bias 199
23.4.6 Alignment Constraints 199
23.4.7 Copying and Applying Constraint Sets 199
23.4.8 ConstraintLayout Chains 199
23.4.9 Guidelines 200
23.4.10 Removing Constraints 200
23.4.11 Scaling 200
23.4.12 Rotation 201
23.5 Summary 201

24. An Android ConstraintSet Tutorial 203

24.1 Creating the Example Project in Android Studio 203
24.2 Adding Views to an Activity 203
24.3 Setting View Attributes 204
24.4 Creating View IDs 205
24.5 Configuring the Constraint Set 206
24.6 Adding the EditText View 207
24.7 Converting Density Independent Pixels (dp) to Pixels (px) 208
24.8 Summary 209

25. A Guide to Using Apply Changes in Android Studio 211

25.1 Introducing Apply Changes 211
25.2 Understanding Apply Changes Options 211
25.3 Using Apply Changes 212
25.4 Configuring Apply Changes Fallback Settings 213
25.5 An Apply Changes Tutorial 213
25.6 Using Apply Code Changes 213
25.7 Using Apply Changes and Restart Activity 214
25.8 Using Run App 214
25.9 Summary 214

26. An Overview and Example of Android Event Handling 215

26.1 Understanding Android Events 215
26.2 Using the android:onClick Resource 215
26.3 Event Listeners and Callback Methods 216
26.4 An Event Handling Example 216
26.5 Designing the User Interface 217
26.6 The Event Listener and Callback Method 217
26.7 Consuming Events 219
26.8 Summary 220

27. Android Touch and Multi-touch Event Handling 221

27.1 Intercepting Touch Events 221
27.2 The MotionEvent Object 221
27.3 Understanding Touch Actions 222
27.4 Handling Multiple Touches 222
27.5 An Example Multi-Touch Application 222
27.6 Designing the Activity User Interface 223
27.7 Implementing the Touch Event Listener 223
27.8 Running the Example Application 226
27.9 Summary 227

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

28.1 Implementing Common Gesture Detection 229
28.2 Creating an Example Gesture Detection Project 230
28.3 Implementing the Listener Class 230
28.4 Creating the GestureDetectorCompat Instance 232
28.5 Implementing the onTouchEvent() Method 233
28.6 Testing the Application 233
28.7 Summary 234

29. Implementing Custom Gesture and Pinch Recognition on Android 235

29.1 The Android Gesture Builder Application 235
29.2 The GestureOverlayView Class 235
29.3 Detecting Gestures 235
29.4 Identifying Specific Gestures 235
29.5 Installing and Running the Gesture Builder Application 235
29.6 Creating a Gestures File 236
29.7 Creating the Example Project 236
29.8 Extracting the Gestures File from the SD Card 236
29.9 Adding the Gestures File to the Project 237
29.10 Designing the User Interface 237
29.11 Loading the Gestures File 238
29.12 Registering the Event Listener 239
29.13 Implementing the onGesturePerformed Method 239
29.14 Testing the Application 240
29.15 Configuring the GestureOverlayView 240
29.16 Intercepting Gestures 241
29.17 Detecting Pinch Gestures 241
29.18 A Pinch Gesture Example Project 241
29.19 Summary 244

30. An Introduction to Android Fragments 245

30.1 What is a Fragment? 245
30.2 Creating a Fragment 245
30.3 Adding a Fragment to an Activity using the Layout XML File 246
30.4 Adding and Managing Fragments in Code 248
30.5 Handling Fragment Events 249
30.6 Implementing Fragment Communication 250
30.7 Summary 251

31. Using Fragments in Android Studio - An Example 253

31.1 About the Example Fragment Application 253
31.2 Creating the Example Project 253
31.3 Creating the First Fragment Layout 253
31.4 Migrating a Fragment to View Binding 255
31.5 Adding the Second Fragment 256
31.6 Adding the Fragments to the Activity 257
31.7 Making the Toolbar Fragment Talk to the Activity 258
31.8 Making the Activity Talk to the Text Fragment 261
31.9 Testing the Application 262
31.10 Summary 263

32. Modern Android App Architecture with Jetpack 265

32.1 What is Android Jetpack? 265
32.2 The “Old” Architecture 265
32.3 Modern Android Architecture 265
32.4 The ViewModel Component 266
32.5 The LiveData Component 266
32.6 ViewModel Saved State 267
32.7 LiveData and Data Binding 267

32.8 Android Lifecycles 268
32.9 Repository Modules 268
32.10 Summary 269

33. An Android ViewModel Tutorial 271

33.1 About the Project 271
33.2 Creating the ViewModel Example Project 271
33.3 Removing Unwanted Project Elements 271
33.4 Designing the Fragment Layout 272
33.5 Implementing the View Model 273
33.6 Associating the Fragment with the View Model 274
33.7 Modifying the Fragment 275
33.8 Accessing the ViewModel Data 276
33.9 Testing the Project 276
33.10 Summary 277

34. An Android Jetpack LiveData Tutorial 279

34.1 LiveData - A Recap 279
34.2 Adding LiveData to the ViewModel 279
34.3 Implementing the Observer 281
34.4 Summary 283

35. An Overview of Android Jetpack Data Binding 285

35.1 An Overview of Data Binding 285
35.2 The Key Components of Data Binding 285
35.2.1 The Project Build Configuration 285
35.2.2 The Data Binding Layout File 286
35.2.3 The Layout File Data Element 287
35.2.4 The Binding Classes 288
35.2.5 Data Binding Variable Configuration 288
35.2.6 Binding Expressions (One-Way) 289
35.2.7 Binding Expressions (Two-Way) 290
35.2.8 Event and Listener Bindings 290
35.3 Summary 291

36. An Android Jetpack Data Binding Tutorial 293

36.1 Removing the Redundant Code 293
36.2 Enabling Data Binding 294
36.3 Adding the Layout Element 295
36.4 Adding the Data Element to Layout File 296
36.5 Working with the Binding Class 297
36.6 Assigning the ViewModel Instance to the Data Binding Variable 298
36.7 Adding Binding Expressions 298
36.8 Adding the Conversion Method 299
36.9 Adding a Listener Binding 299
36.10 Testing the App 300
36.11 Summary 300

37. An Android ViewModel Saved State Tutorial 301

37.1 Understanding ViewModel State Saving 301
37.2 Implementing ViewModel State Saving 301

37.3 Saving and Restoring State 303
37.4 Adding Saved State Support to the ViewModelDemo Project 303
37.5 Summary 305

38. Working with Android Lifecycle-Aware Components 307

38.1 Lifecycle Awareness 307
38.2 Lifecycle Owners 307
38.3 Lifecycle Observers 308
38.4 Lifecycle States and Events 309
38.5 Summary 310

39. An Android Jetpack Lifecycle Awareness Tutorial 311

39.1 Creating the Example Lifecycle Project 311
39.2 Creating a Lifecycle Observer 311
39.3 Adding the Observer 313
39.4 Testing the Observer 313
39.5 Creating a Lifecycle Owner 313
39.6 Testing the Custom Lifecycle Owner 315
39.7 Summary 316

40. An Overview of the Navigation Architecture Component 317

40.1 Understanding Navigation 317
40.2 Declaring a Navigation Host 318
40.3 The Navigation Graph 320
40.4 Accessing the Navigation Controller 321
40.5 Triggering a Navigation Action 321
40.6 Passing Arguments 322
40.7 Summary 322

41. An Android Jetpack Navigation Component Tutorial 323

41.1 Creating the NavigationDemo Project 323
41.2 Adding Navigation to the Build Configuration 323
41.3 Creating the Navigation Graph Resource File 324
41.4 Declaring a Navigation Host 325
41.5 Adding Navigation Destinations 326
41.6 Designing the Destination Fragment Layouts 328
41.7 Adding an Action to the Navigation Graph 329
41.8 Implement the OnFragmentInteractionListener 331
41.9 Adding View Binding Support to the Destination Fragments 332
41.10 Triggering the Action 332
41.11 Passing Data Using Safeargs 333
41.12 Summary 336

42. An Introduction to MotionLayout 337

42.1 An Overview of MotionLayout 337
42.2 MotionLayout 337
42.3 MotionScene 337
42.4 Configuring ConstraintSets 338
42.5 Custom Attributes 339
42.6 Triggering an Animation 341
42.7 Arc Motion 342

42.8 Keyframes ... 342
42.8.1 Attribute Keyframes ... 342
42.8.2 Position Keyframes ... 343
42.9 Time Linearity ... 346
42.10 KeyTrigger ... 346
42.11 Cycle and Time Cycle Keyframes ... 347
42.12 Starting an Animation from Code ... 347

43. An Android MotionLayout Editor Tutorial ... 349

43.1 Creating the MotionLayoutDemo Project ... 349
43.2 ConstraintLayout to MotionLayout Conversion ... 349
43.3 Configuring Start and End Constraints ... 351
43.4 Previewing the MotionLayout Animation ... 354
43.5 Adding an OnClick Gesture ... 354
43.6 Adding an Attribute Keyframe to the Transition ... 356
43.7 Adding a CustomAttribute to a Transition ... 358
43.8 Adding Position Keyframes ... 360
43.9 Summary ... 362

44. A MotionLayout KeyCycle Tutorial ... 363

44.1 An Overview of Cycle Keyframes ... 363
44.2 Using the Cycle Editor ... 367
44.3 Creating the KeyCycleDemo Project ... 368
44.4 Configuring the Start and End Constraints ... 368
44.5 Creating the Cycles ... 370
44.6 Previewing the Animation ... 372
44.7 Adding the KeyFrameSet to the MotionScene ... 372
44.8 Summary ... 374

45. Working with the Floating Action Button and Snackbar ... 375

45.1 The Material Design ... 375
45.2 The Design Library ... 375
45.3 The Floating Action Button (FAB) ... 375
45.4 The Snackbar ... 376
45.5 Creating the Example Project ... 377
45.6 Reviewing the Project ... 377
45.7 Removing Navigation Features ... 378
45.8 Changing the Floating Action Button ... 378
45.9 Adding an Action to the Snackbar ... 380
45.10 Summary ... 380

46. Creating a Tabbed Interface using the TabLayout Component ... 383

46.1 An Introduction to the ViewPager2 ... 383
46.2 An Overview of the TabLayout Component ... 383
46.3 Creating the TabLayoutDemo Project ... 384
46.4 Creating the First Fragment ... 385
46.5 Duplicating the Fragments ... 386
46.6 Adding the TabLayout and ViewPager2 ... 387
46.7 Creating the Pager Adapter ... 388
46.8 Performing the Initialization Tasks ... 389
46.9 Testing the Application ... 391

46.10 Customizing the TabLayout 392
46.11 Summary 393

47. Working with the RecyclerView and CardView Widgets 395

47.1 An Overview of the RecyclerView 395
47.2 An Overview of the CardView 397
47.3 Summary 398

48. An Android RecyclerView and CardView Tutorial 399

48.1 Creating the CardDemo Project 399
48.2 Modifying the Basic Views Activity Project 399
48.3 Designing the CardView Layout 400
48.4 Adding the RecyclerView 401
48.5 Adding the Image Files 401
48.6 Creating the RecyclerView Adapter 402
48.7 Initializing the RecyclerView Component 404
48.8 Testing the Application 405
48.9 Responding to Card Selections 406
48.10 Summary 407

49. A Layout Editor Sample Data Tutorial 409

49.1 Adding Sample Data to a Project 409
49.2 Using Custom Sample Data 413
49.3 Summary 416

50. Working with the AppBar and Collapsing Toolbar Layouts 417

50.1 The Anatomy of an AppBar 417
50.2 The Example Project 418
50.3 Coordinating the RecyclerView and Toolbar 418
50.4 Introducing the Collapsing Toolbar Layout 420
50.5 Changing the Title and Scrim Color 423
50.6 Summary 424

51. An Android Studio Primary/Detail Flow Tutorial 425

51.1 The Primary/Detail Flow 425
51.2 Creating a Primary/Detail Flow Activity 426
51.3 Adding the Primary/Detail Flow Activity 426
51.4 Modifying the Primary/Detail Flow Template 427
51.5 Changing the Content Model 427
51.6 Changing the Detail Pane 429
51.7 Modifying the ItemDetailFragment Class 430
51.8 Modifying the ItemListFragment Class 431
51.9 Adding Manifest Permissions 432
51.10 Running the Application 432
51.11 Summary 433

52. An Overview of Android Services 435

52.1 Intent Service 435
52.2 Bound Service 435
52.3 The Anatomy of a Service 436
52.4 Controlling Destroyed Service Restart Options 436
52.5 Declaring a Service in the Manifest File 436

52.6 Starting a Service Running on System Startup 437
52.7 Summary 438

53. An Overview of Android Intents 439

53.1 An Overview of Intents 439
53.2 Explicit Intents 439
53.3 Returning Data from an Activity 440
53.4 Implicit Intents 441
53.5 Using Intent Filters 442
53.6 Automatic Link Verification 442
53.7 Manually Enabling Links 445
53.8 Checking Intent Availability 446
53.9 Summary 447

54. Android Explicit Intents – A Worked Example 449

54.1 Creating the Explicit Intent Example Application 449
54.2 Designing the User Interface Layout for MainActivity 449
54.3 Creating the Second Activity Class 450
54.4 Designing the User Interface Layout for SecondActivity 451
54.5 Reviewing the Application Manifest File 451
54.6 Creating the Intent 452
54.7 Extracting Intent Data 453
54.8 Launching SecondActivity as a Sub-Activity 454
54.9 Returning Data from a Sub-Activity 455
54.10 Testing the Application 455
54.11 Summary 455

55. Android Implicit Intents – A Worked Example 457

55.1 Creating the Android Studio Implicit Intent Example Project 457
55.2 Designing the User Interface 457
55.3 Creating the Implicit Intent 458
55.4 Adding a Second Matching Activity 459
55.5 Adding the Web View to the UI 459
55.6 Obtaining the Intent URL 460
55.7 Modifying the MyWebView Project Manifest File 461
55.8 Installing the MyWebView Package on a Device 462
55.9 Testing the Application 463
55.10 Manually Enabling the Link 463
55.11 Automatic Link Verification 465
55.12 Summary 467

56. Android Broadcast Intents and Broadcast Receivers 469

56.1 An Overview of Broadcast Intents 469
56.2 An Overview of Broadcast Receivers 470
56.3 Obtaining Results from a Broadcast 471
56.4 Sticky Broadcast Intents 471
56.5 The Broadcast Intent Example 472
56.6 Creating the Example Application 472
56.7 Creating and Sending the Broadcast Intent 472
56.8 Creating the Broadcast Receiver 473
56.9 Registering the Broadcast Receiver 474

56.10 Testing the Broadcast Example 475
56.11 Listening for System Broadcasts 475
56.12 Summary 476

57. Android Local Bound Services – A Worked Example 477

57.1 Understanding Bound Services 477
57.2 Bound Service Interaction Options 477
57.3 A Local Bound Service Example 477
57.4 Adding a Bound Service to the Project 478
57.5 Implementing the Binder 478
57.6 Binding the Client to the Service 481
57.7 Completing the Example 482
57.8 Testing the Application 483
57.9 Summary 483

58. Android Remote Bound Services – A Worked Example 485

58.1 Client to Remote Service Communication 485
58.2 Creating the Example Application 485
58.3 Designing the User Interface 485
58.4 Implementing the Remote Bound Service 485
58.5 Configuring a Remote Service in the Manifest File 487
58.6 Launching and Binding to the Remote Service 488
58.7 Sending a Message to the Remote Service 489
58.8 Summary 490

59. A Basic Overview of Java Threads, Handlers and Executors 491

59.1 The Application Main Thread 491
59.2 Thread Handlers 491
59.3 A Threading Example 491
59.4 Building the App 492
59.5 Creating a New Thread 493
59.6 Implementing a Thread Handler 494
59.7 Passing a Message to the Handler 496
59.8 Java Executor Concurrency 496
59.9 Working with Runnable Tasks 497
59.10 Shutting down an Executor Service 498
59.11 Working with Callable Tasks and Futures 498
59.12 Handling a Future Result 500
59.13 Scheduling Tasks 501
59.14 Summary 502

60. Making Runtime Permission Requests in Android 503

60.1 Understanding Normal and Dangerous Permissions 503
60.2 Creating the Permissions Example Project 505
60.3 Checking for a Permission 505
60.4 Requesting Permission at Runtime 507
60.5 Providing a Rationale for the Permission Request 508
60.6 Testing the Permissions App 510
60.7 Summary 510

61. An Android Notifications Tutorial 511

61.1 An Overview of Notifications 511
61.2 Creating the NotifyDemo Project 513
61.3 Designing the User Interface 513
61.4 Creating the Second Activity 513
61.5 Creating a Notification Channel 514
61.6 Requesting Notification Permission 515
61.7 Creating and Issuing a Notification 518
61.8 Launching an Activity from a Notification 520
61.9 Adding Actions to a Notification 522
61.10 Bundled Notifications 523
61.11 Summary 525

62. An Android Direct Reply Notification Tutorial 527

62.1 Creating the DirectReply Project 527
62.2 Designing the User Interface 527
62.3 Requesting Notification Permission 528
62.4 Creating the Notification Channel 529
62.5 Building the RemoteInput Object 530
62.6 Creating the PendingIntent 531
62.7 Creating the Reply Action 532
62.8 Receiving Direct Reply Input 534
62.9 Updating the Notification 535
62.10 Summary 537

63. Foldable Devices and Multi-Window Support 539

63.1 Foldables and Multi-Window Support 539
63.2 Using a Foldable Emulator 540
63.3 Entering Multi-Window Mode 541
63.4 Enabling and using Freeform Support 542
63.5 Checking for Freeform Support 542
63.6 Enabling Multi-Window Support in an App 542
63.7 Specifying Multi-Window Attributes 543
63.8 Detecting Multi-Window Mode in an Activity 544
63.9 Receiving Multi-Window Notifications 544
63.10 Launching an Activity in Multi-Window Mode 545
63.11 Configuring Freeform Activity Size and Position 545
63.12 Summary 546

64. An Overview of Android SQLite Databases 547

64.1 Understanding Database Tables 547
64.2 Introducing Database Schema 547
64.3 Columns and Data Types 547
64.4 Database Rows 548
64.5 Introducing Primary Keys 548
64.6 What is SQLite? 548
64.7 Structured Query Language (SQL) 548
64.8 Trying SQLite on an Android Virtual Device (AVD) 549
64.9 The Android Room Persistence Library 550
64.10 Summary 551

65. The Android Room Persistence Library 553

65.1 Revisiting Modern App Architecture 553
65.2 Key Elements of Room Database Persistence 553
65.2.1 Repository 554
65.2.2 Room Database 554
65.2.3 Data Access Object (DAO) 554
65.2.4 Entities 554
65.2.5 SQLite Database 554
65.3 Understanding Entities 555
65.4 Data Access Objects 558
65.5 The Room Database 559
65.6 The Repository 560
65.7 In-Memory Databases 561
65.8 Database Inspector 561
65.9 Summary 561

66. An Android TableLayout and TableRow Tutorial 563

66.1 The TableLayout and TableRow Layout Views 563
66.2 Creating the Room Database Project 564
66.3 Converting to a LinearLayout 564
66.4 Adding the TableLayout to the User Interface 565
66.5 Configuring the TableRows 566
66.6 Adding the Button Bar to the Layout 567
66.7 Adding the RecyclerView 568
66.8 Adjusting the Layout Margins 569
66.9 Summary 569

67. An Android Room Database and Repository Tutorial 571

67.1 About the RoomDemo Project 571
67.2 Modifying the Build Configuration 571
67.3 Building the Entity 571
67.4 Creating the Data Access Object 573
67.5 Adding the Room Database 574
67.6 Adding the Repository 575
67.7 Adding the ViewModel 578
67.8 Creating the Product Item Layout 579
67.9 Adding the RecyclerView Adapter 579
67.10 Preparing the Main Activity 581
67.11 Adding the Button Listeners 582
67.12 Adding LiveData Observers 583
67.13 Initializing the RecyclerView 584
67.14 Testing the RoomDemo App 584
67.15 Using the Database Inspector 584
67.16 Summary 585

68. Accessing Cloud Storage using the Android Storage Access Framework 587

68.1 The Storage Access Framework 587
68.2 Working with the Storage Access Framework 588
68.3 Filtering Picker File Listings 588
68.4 Handling Intent Results 589
68.5 Reading the Content of a File 589
68.6 Writing Content to a File 590

68.7 Deleting a File 591
68.8 Gaining Persistent Access to a File 591
68.9 Summary 591

69. An Android Storage Access Framework Example 593

69.1 About the Storage Access Framework Example 593
69.2 Creating the Storage Access Framework Example 593
69.3 Designing the User Interface 593
69.4 Adding the Activity Launchers 594
69.5 Creating a New Storage File 596
69.6 Saving to a Storage File 596
69.7 Opening and Reading a Storage File 598
69.8 Testing the Storage Access Application 599
69.9 Summary 600

70. Video Playback on Android using the VideoView and MediaController Classes 601

70.1 Introducing the Android VideoView Class 601
70.2 Introducing the Android MediaController Class 602
70.3 Creating the Video Playback Example 602
70.4 Designing the VideoPlayer Layout 602
70.5 Downloading the Video File 603
70.6 Configuring the VideoView 603
70.7 Adding the MediaController to the Video View 605
70.8 Setting up the onPreparedListener 606
70.9 Summary 606

71. Android Picture-in-Picture Mode 607

71.1 Picture-in-Picture Features 607
71.2 Enabling Picture-in-Picture Mode 608
71.3 Configuring Picture-in-Picture Parameters 608
71.4 Entering Picture-in-Picture Mode 609
71.5 Detecting Picture-in-Picture Mode Changes 609
71.6 Adding Picture-in-Picture Actions 610
71.7 Summary 610

72. An Android Picture-in-Picture Tutorial 613

72.1 Adding Picture-in-Picture Support to the Manifest 613
72.2 Adding a Picture-in-Picture Button 613
72.3 Entering Picture-in-Picture Mode 614
72.4 Detecting Picture-in-Picture Mode Changes 615
72.5 Adding a Broadcast Receiver 615
72.6 Adding the PiP Action 616
72.7 Testing the Picture-in-Picture Action 619
72.8 Summary 620

73. Android Audio Recording and Playback using MediaPlayer and MediaRecorder 621

73.1 Playing Audio 621
73.2 Recording Audio and Video using the MediaRecorder Class 622
73.3 About the Example Project 623
73.4 Creating the AudioApp Project 623
73.5 Designing the User Interface 623

73.6 Checking for Microphone Availability 624
73.7 Initializing the Activity 625
73.8 Implementing the recordAudio() Method 626
73.9 Implementing the stopAudio() Method 626
73.10 Implementing the playAudio() method 627
73.11 Configuring and Requesting Permissions 627
73.12 Testing the Application 629
73.13 Summary 630

74. Working with the Google Maps Android API in Android Studio 631

74.1 The Elements of the Google Maps Android API 631
74.2 Creating the Google Maps Project 632
74.3 Creating a Google Cloud Billing Account 632
74.4 Creating a New Google Cloud Project 633
74.5 Enabling the Google Maps SDK 634
74.6 Generating a Google Maps API Key 635
74.7 Adding the API Key to the Android Studio Project 636
74.8 Testing the Application 636
74.9 Understanding Geocoding and Reverse Geocoding 636
74.10 Adding a Map to an Application 638
74.11 Requesting Current Location Permission 638
74.12 Displaying the User’s Current Location 640
74.13 Changing the Map Type 641
74.14 Displaying Map Controls to the User 642
74.15 Handling Map Gesture Interaction 643
74.15.1 Map Zooming Gestures 643
74.15.2 Map Scrolling/Panning Gestures 643
74.15.3 Map Tilt Gestures 643
74.15.4 Map Rotation Gestures 643
74.16 Creating Map Markers 644
74.17 Controlling the Map Camera 645
74.18 Summary 646

75. Printing with the Android Printing Framework 647

75.1 The Android Printing Architecture 647
75.2 The Print Service Plugins 647
75.3 Google Cloud Print 648
75.4 Printing to Google Drive 648
75.5 Save as PDF 649
75.6 Printing from Android Devices 649
75.7 Options for Building Print Support into Android Apps 650
75.7.1 Image Printing 650
75.7.2 Creating and Printing HTML Content 651
75.7.3 Printing a Web Page 652
75.7.4 Printing a Custom Document 653
75.8 Summary 653

76. An Android HTML and Web Content Printing Example 655

76.1 Creating the HTML Printing Example Application 655
76.2 Printing Dynamic HTML Content 655
76.3 Creating the Web Page Printing Example 658

76.4 Removing the Floating Action Button ... 658
76.5 Removing Navigation Features ... 658
76.6 Designing the User Interface Layout ... 660
76.7 Accessing the WebView from the Main Activity ... 660
76.8 Loading the Web Page into the WebView ... 661
76.9 Adding the Print Menu Option ... 662
76.10 Summary ... 664

77. A Guide to Android Custom Document Printing ... 665

77.1 An Overview of Android Custom Document Printing ... 665
77.1.1 Custom Print Adapters ... 665
77.2 Preparing the Custom Document Printing Project ... 666
77.3 Creating the Custom Print Adapter ... 667
77.4 Implementing the onLayout() Callback Method ... 668
77.5 Implementing the onWrite() Callback Method ... 671
77.6 Checking a Page is in Range ... 673
77.7 Drawing the Content on the Page Canvas ... 674
77.8 Starting the Print Job ... 676
77.9 Testing the Application ... 677
77.10 Summary ... 677

78. An Introduction to Android App Links ... 679

78.1 An Overview of Android App Links ... 679
78.2 App Link Intent Filters ... 679
78.3 Handling App Link Intents ... 680
78.4 Associating the App with a Website ... 680
78.5 Summary ... 681

79. An Android Studio App Links Tutorial ... 683

79.1 About the Example App ... 683
79.2 The Database Schema ... 683
79.3 Loading and Running the Project ... 683
79.4 Adding the URL Mapping ... 685
79.5 Adding the Intent Filter ... 688
79.6 Adding Intent Handling Code ... 689
79.7 Testing the App ... 691
79.8 Creating the Digital Asset Links File ... 691
79.9 Testing the App Link ... 692
79.10 Summary ... 692

80. An Android Biometric Authentication Tutorial ... 693

80.1 An Overview of Biometric Authentication ... 693
80.2 Creating the Biometric Authentication Project ... 693
80.3 Configuring Device Fingerprint Authentication ... 694
80.4 Adding the Biometric Permission to the Manifest File ... 694
80.5 Designing the User Interface ... 695
80.6 Adding a Toast Convenience Method ... 695
80.7 Checking the Security Settings ... 696
80.8 Configuring the Authentication Callbacks ... 697
80.9 Adding the CancellationSignal ... 698
80.10 Starting the Biometric Prompt ... 698

80.11 Testing the Project.......699
80.12 Summary.......700

81. Creating, Testing, and Uploading an Android App Bundle.......701

81.1 The Release Preparation Process.......701
81.2 Android App Bundles.......701
81.3 Register for a Google Play Developer Console Account.......702
81.4 Configuring the App in the Console.......703
81.5 Enabling Google Play App Signing.......704
81.6 Creating a Keystore File.......704
81.7 Creating the Android App Bundle.......705
81.8 Generating Test APK Files.......707
81.9 Uploading the App Bundle to the Google Play Developer Console.......708
81.10 Exploring the App Bundle.......709
81.11 Managing Testers.......710
81.12 Rolling the App Out for Testing.......710
81.13 Uploading New App Bundle Revisions.......711
81.14 Analyzing the App Bundle File.......712
81.15 Summary.......713

82. An Overview of Android In-App Billing.......715

82.1 Preparing a Project for In-App Purchasing.......715
82.2 Creating In-App Products and Subscriptions.......715
82.3 Billing Client Initialization.......716
82.4 Connecting to the Google Play Billing Library.......717
82.5 Querying Available Products.......718
82.6 Starting the Purchase Process.......718
82.7 Completing the Purchase.......719
82.8 Querying Previous Purchases.......720
82.9 Summary.......721

83. An Android In-App Purchasing Tutorial.......723

83.1 About the In-App Purchasing Example Project.......723
83.2 Creating the InAppPurchase Project.......723
83.3 Adding Libraries to the Project.......723
83.4 Designing the User Interface.......724
83.5 Adding the App to the Google Play Store.......724
83.6 Creating an In-App Product.......725
83.7 Enabling License Testers.......725
83.8 Initializing the Billing Client.......726
83.9 Querying the Product.......728
83.10 Launching the Purchase Flow.......729
83.11 Handling Purchase Updates.......729
83.12 Consuming the Product.......730
83.13 Restoring a Previous Purchase.......731
83.14 Testing the App.......732
83.15 Troubleshooting.......733
83.16 Summary.......734

84. Working with Material Design 3 Theming.......735

84.1 Material Design 2 vs. Material Design 3.......735

84.2 Understanding Material Design Theming 735
84.3 Material Design 3 Theming 735
84.4 Building a Custom Theme 737
84.5 Summary 738

85. A Material Design 3 Theming and Dynamic Color Tutorial 739

85.1 Creating the ThemeDemo Project 739
85.3 Designing the User Interface 739
85.4 Building a New Theme 741
85.5 Adding the Theme to the Project 742
85.6 Enabling Dynamic Color Support 743
85.7 Previewing Dynamic Colors 744
85.8 Summary 745

86. An Overview of Gradle in Android Studio 747

86.1 An Overview of Gradle 747
86.2 Gradle and Android Studio 747
86.2.1 Sensible Defaults 747
86.2.2 Dependencies 747
86.2.3 Build Variants 748
86.2.4 Manifest Entries 748
86.2.5 APK Signing 748
86.2.6 ProGuard Support 748
86.3 The Property and Settings Gradle Build File 748
86.4 The Top-level Gradle Build File 749
86.5 Module Level Gradle Build Files 750
86.6 Configuring Signing Settings in the Build File 752
86.7 Running Gradle Tasks from the Command Line 753
86.8 Summary 754

Index 755

Chapter 1

1. Introduction

Fully updated for Android Studio Giraffe and the new UI, this book aims to teach 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 are also included covering the Android Architecture Components, including view models, lifecycle management, Room database access, 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.ebookfrenzy.com/retail/giraffejava/index.php

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 *feedback@ebookfrenzy.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.ebookfrenzy.com/errata/giraffejava.html

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

Chapter 2

2. Setting up an Android Studio Development Environment

Before any work can begin on developing an Android application, the first step is to configure a computer system to act as the development platform. This involves several steps consisting of installing the Android Studio Integrated Development Environment (IDE), including the Android Software Development Kit (SDK) and the OpenJDK Java development environment.

This chapter will cover the steps necessary to install the requisite components for Android application development on Windows, macOS, and Linux-based systems.

2.1 System requirements

Android application development may be performed on any of the following system types:

- Windows 8/10/11 64-bit
- macOS 10.14 or later running on Intel or Apple silicon
- Chrome OS device with Intel i5 or higher
- Linux systems with version 2.31 or later of the GNU C Library (glibc)
- Minimum of 8GB of RAM
- Approximately 8GB of available disk space
- 1280 x 800 minimum screen resolution

2.2 Downloading the Android Studio package

Most of the work involved in developing applications for Android will be performed using the Android Studio environment. The content and examples in this book were created based on Android Studio Giraffe 2022.3.1 using the Android API 33 SDK (Tiramisu), which, at the time of writing, are the latest stable releases.

Android Studio is, however, subject to frequent updates, so a newer version may have been released since this book was published.

The latest release of Android Studio may be downloaded from the primary download page, which can be found at the following URL:

https://developer.android.com/studio/index.html

If this page provides instructions for downloading a newer version of Android Studio, there may be differences between this book and the software. A web search for "Android Studio Giraffe" should provide the option to download the older version if these differences become a problem. Alternatively, visit the following web page to find Android Studio Giraffe 2022.3.1 in the archives:

https://developer.android.com/studio/archive

2.3 Installing Android Studio

Once downloaded, the exact steps to install Android Studio differ depending on the operating system on which the installation is performed.

2.3.1 Installation on Windows

Locate the downloaded Android Studio installation executable file (named *android-studio-<version>-windows.exe*) in a Windows Explorer window and double-click on it to start the installation process, clicking the *Yes* button in the User Account Control dialog if it appears.

Once the Android Studio setup wizard appears, work through the various screens to configure the installation to meet your requirements in terms of the file system location into which Android Studio should be installed and whether or not it should be made available to other system users. When prompted to select the components to install, ensure that the *Android Studio* and *Android Virtual Device* options are all selected.

Although there are no strict rules on where Android Studio should be installed on the system, the remainder of this book will assume that the installation was performed into *C:\Program Files\Android\Android Studio* and that the Android SDK packages have been installed into the user's *AppData\Local\Android\sdk* sub-folder. Once the options have been configured, click the *Install* button to begin the installation process.

On versions of Windows with a Start menu, the newly installed Android Studio can be launched from the entry added to that menu during the installation. The executable may be pinned to the taskbar for easy access by navigating to the *Android Studio\bin* directory, right-clicking on the *studio64* executable, and selecting the *Pin to Taskbar* menu option (on Windows 11, this option can be found by selecting *Show more options* from the menu).

2.3.2 Installation on macOS

Android Studio for macOS is downloaded as a disk image (.dmg) file. Once the *android-studio-<version>-mac.dmg* file has been downloaded, locate it in a Finder window and double-click on it to open it, as shown in Figure 2-1:

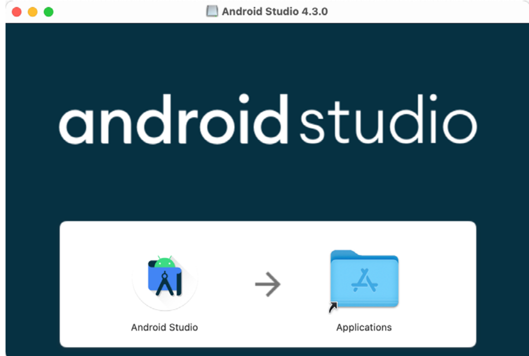


Figure 2-1

To install the package, drag the Android Studio icon and drop it onto the Applications folder. The Android Studio package will then be installed into the Applications folder of the system, a process that will typically take a few seconds to complete.

To launch Android Studio, locate the executable in the Applications folder using a Finder window and double-click on it.

For future, easier access to the tool, drag the Android Studio icon from the Finder window and drop it onto the dock.

2.3.3 Installation on Linux

Having downloaded the Linux Android Studio package, open a terminal window, change directory to the location where Android Studio is to be installed, and execute the following command:

```
tar xvfz /<path to package>/android-studio-<version>-linux.tar.gz
```

Note that the Android Studio bundle will be installed into a subdirectory named *android-studio*. Therefore, assuming that the above command was executed in */home/demo*, the software packages will be unpacked into */home/demo/android-studio*.

To launch Android Studio, open a terminal window, change directory to the *android-studio/bin* sub-directory, and execute the following command:

```
./studio.sh
```

2.4 The Android Studio setup wizard

If you have previously installed an earlier version of Android Studio, the first time this new version is launched, a dialog may appear providing the option to import settings from a previous Android Studio version. If you have settings from a previous version and would like to import them into the latest installation, select the appropriate option and location. Alternatively, indicate that you do not need to import any previous settings and click the OK button to proceed.

If you are installing Android Studio for the first time, the initial dialog that appears once the setup process starts may resemble that shown in Figure 2-2 below:

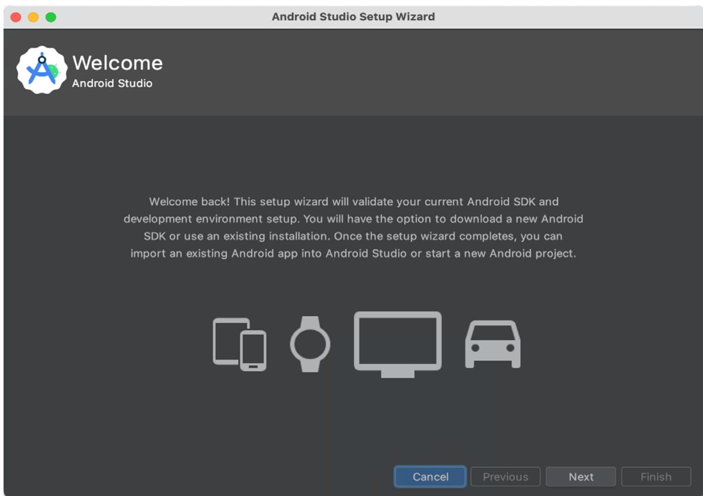


Figure 2-2

If this dialog appears, click the Next button to display the Install Type screen (Figure 2-3). On this screen, select the Standard installation option before clicking Next.

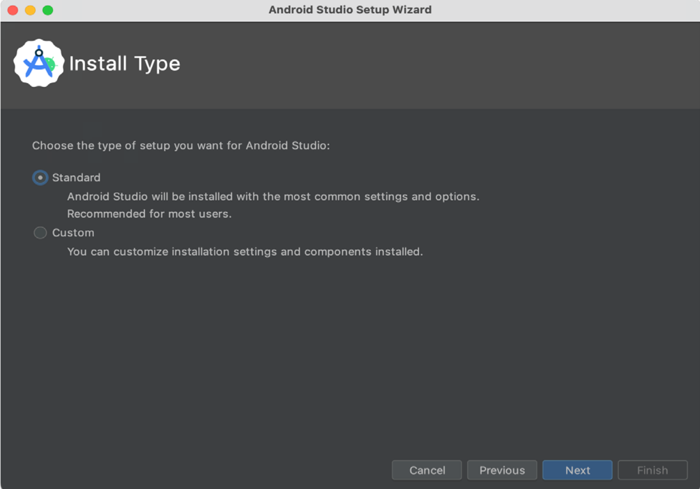


Figure 2-3

On the Select UI Theme screen, select either the Darcula or Light theme based on your preferences. After making a choice, click Next, and review the options in the Verify Settings screen before proceeding to the License Agreement screen. Select each license category and enable the Accept checkbox. Finally, click the Finish button to initiate the installation.

After these initial setup steps have been taken, click the Finish button to display the Welcome to Android Studio screen using your chosen UI theme:

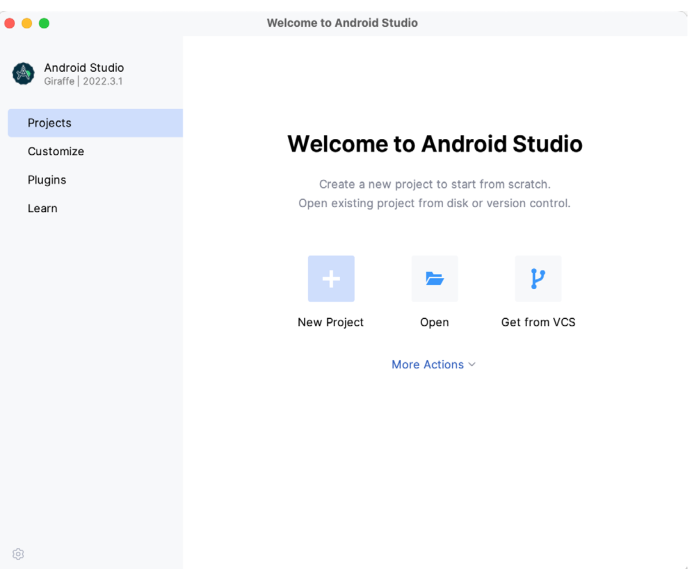


Figure 2-4

2.5 Installing additional Android SDK packages

The steps performed so far have installed the Android Studio IDE and the current set of default Android SDK packages. Before proceeding, it is worth taking some time to verify which packages are installed and to install any missing or updated packages.

This task can be performed by clicking on the *More Actions* link within the welcome dialog and selecting the *SDK Manager* option from the drop-down menu. Once invoked, the *Android SDK* screen of the Settings dialog will appear as shown in Figure 2-5:

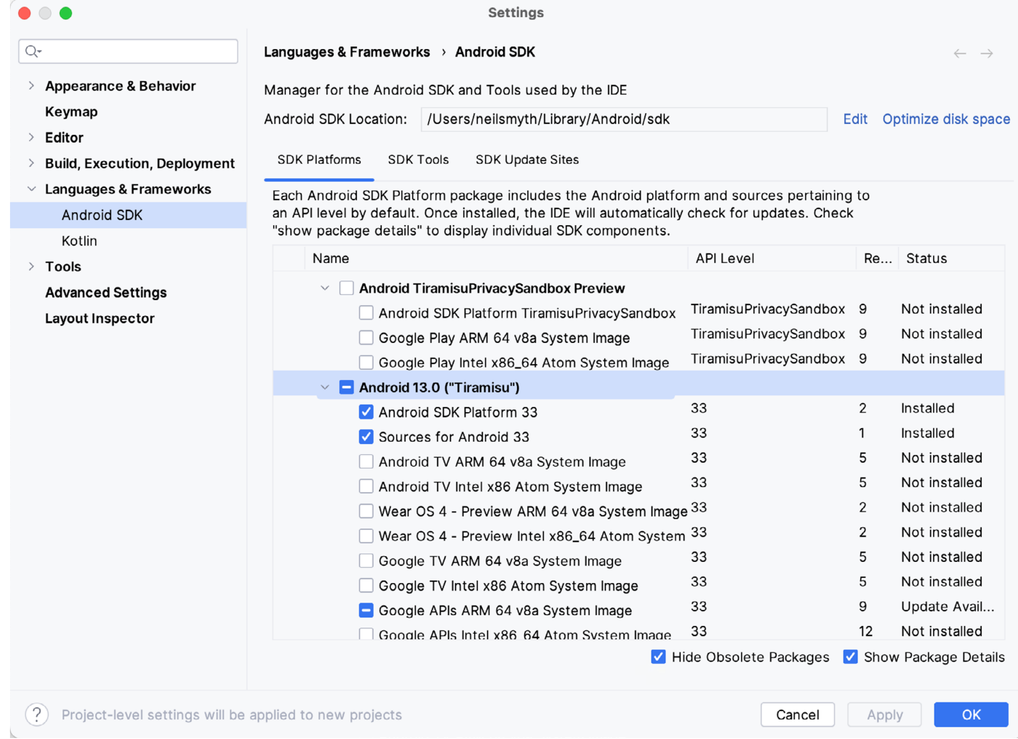


Figure 2-5

Google pairs each release of Android Studio with a maximum supported Application Programming Interface (API) level of the Android SDK. In the case of Android Studio Giraffe, this is Android Tiramisu (API Level 33). This information can be confirmed using the following link:

https://developer.android.com/studio/releases#api-level-support

Immediately after installing Android Studio for the first time, it is likely that only the latest supported version of the Android SDK has been installed. To install older versions of the Android SDK, select the checkboxes corresponding to the versions and click the *Apply* button. The rest of this book assumes that the Android Tiramisu (API Level 33) SDK is installed.

Most of the examples in this book will support older versions of Android as far back as Android 8.0 (Oreo). This ensures that the apps run on a wide range of Android devices. Within the list of SDK versions, enable the checkbox next to Android 8.0 (Oreo) and click the Apply button. Click the OK button to install the SDK in the resulting confirmation dialog. Subsequent dialogs will seek the acceptance of licenses and terms before performing the installation. Click Finish once the installation is complete.

It is also possible that updates will be listed as being available for the latest SDK. To access detailed information about the packages that are ready to be updated, enable the *Show Package Details* option located in the lower right-hand corner of the screen. This will display information similar to that shown in Figure 2-6:

Name	API Level	Revision	Status
☐ Android TV ARM 64 v8a System Image	33	5	Not installed
☐ Android TV Intel x86 Atom System Image	33	5	Not installed
☐ Google TV ARM 64 v8a System Image	33	5	Not installed
☐ Google TV Intel x86 Atom System Image	33	5	Not installed
⊟ Google APIs ARM 64 v8a System Image	33	8	Update Available: 9
☐ Google APIs Intel x86 Atom_64 System Image	33	9	Not installed
☑ Google Play ARM 64 v8a System Image	33	7	Installed

Figure 2-6

The above figure highlights the availability of an update. To install the updates, enable the checkbox to the left of the item name and click the *Apply* button.

In addition to the Android SDK packages, several tools are also installed for building Android applications. To view the currently installed packages and check for updates, remain within the SDK settings screen and select the SDK Tools tab as shown in Figure 2-7:

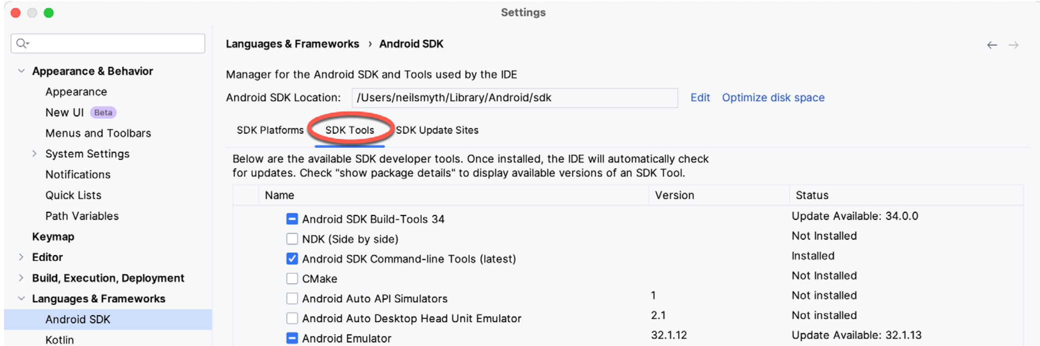


Figure 2-7

Within the Android SDK Tools screen, make sure that the following packages are listed as *Installed* in the Status column:

- Android SDK Build-tools
- Android Emulator
- Android SDK Platform-tools
- Google Play Services
- Intel x86 Emulator Accelerator (HAXM installer)*
- Google USB Driver (Windows only)
- Layout Inspector image server for API 31 and 34

*Note that the Intel x86 Emulator Accelerator (HAXM installer) cannot be installed on Apple silicon-based Macs.

If any of the above packages are listed as *Not Installed* or requiring an update, select the checkboxes next to those packages and click the *Apply* button to initiate the installation process. If the HAXM emulator settings dialog appears, select the recommended memory allocation:

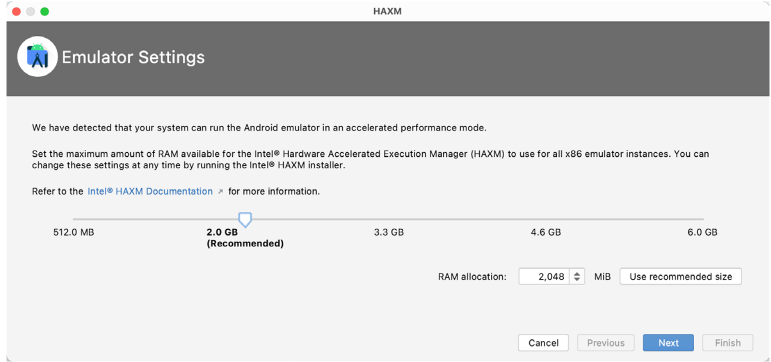


Figure 2-8

Once the installation is complete, review the package list and ensure that the selected packages are listed as *Installed* in the *Status* column. If any are listed as *Not installed,* make sure they are selected and click the *Apply* button again.

2.6 Installing the Android SDK Command-line Tools

Android Studio includes tools that allow some tasks to be performed from your operating system command line. To install these tools on your system, open the SDK Manager, select the SDK Tools tab, and locate the *Android SDK Command-line Tools (latest)* package as shown in Figure 2-9:

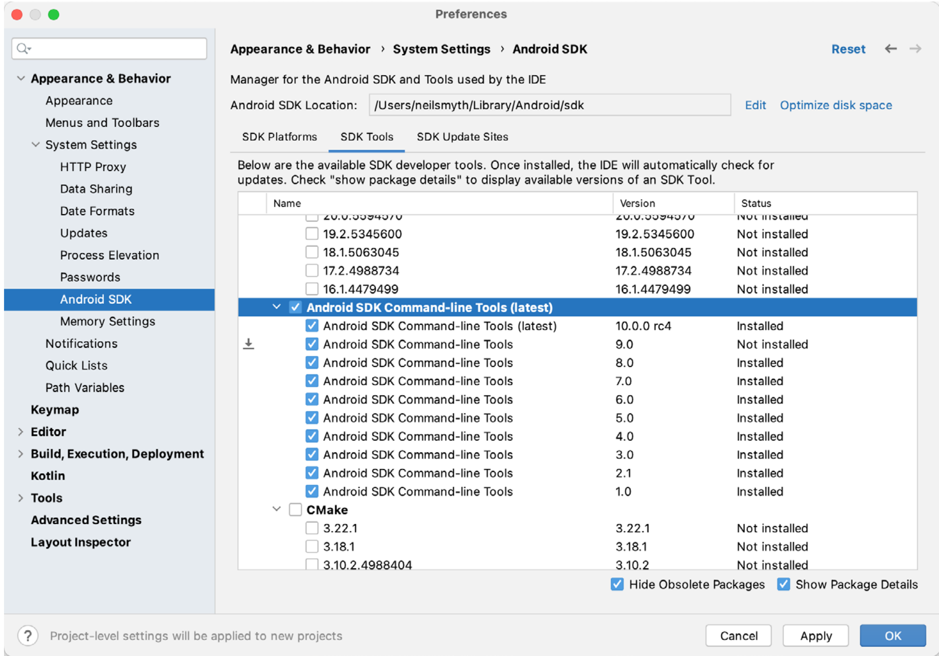


Figure 2-9

If the command-line tools package is not already installed, enable it and click Apply, followed by OK to complete the installation. When the installation completes, click Finish and close the SDK Manager dialog.

For the operating system on which you are developing to be able to find these tools, it will be necessary to add them to the system's *PATH* environment variable.

Regardless of your operating system, you will need to configure the PATH environment variable to include the following paths (where *<path_to_android_sdk_installation>* represents the file system location into which you installed the Android SDK):

```
<path_to_android_sdk_installation>/sdk/cmdline-tools/latest/bin
<path_to_android_sdk_installation>/sdk/platform-tools
```

You can identify the location of the SDK on your system by launching the SDK Manager and referring to the *Android SDK Location:* field located at the top of the settings panel, as highlighted in Figure 2-10:

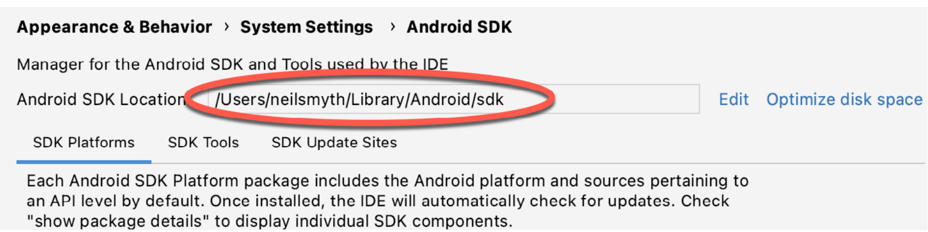


Figure 2-10

Once the location of the SDK has been identified, the steps to add this to the PATH variable are operating system dependent:

2.6.1 Windows 8.1

1. On the start screen, move the mouse to the bottom right-hand corner of the screen and select Search from the resulting menu. In the search box, enter Control Panel. When the Control Panel icon appears in the results area, click on it to launch the tool on the desktop.

2. Within the Control Panel, use the Category menu to change the display to Large Icons. From the list of icons, select the one labeled System.

3. In the Environment Variables dialog, locate the Path variable in the System variables list, select it, and click the *Edit…* button. Using the *New* button in the edit dialog, add two new entries to the path. For example, assuming the Android SDK was installed into *C:\Users\demo\AppData\Local\Android\Sdk*, the following entries would need to be added:

```
C:\Users\demo\AppData\Local\Android\Sdk\cmdline-tools\latest\bin
C:\Users\demo\AppData\Local\Android\Sdk\platform-tools
```

4. Click OK in each dialog box and close the system properties control panel.

Open a command prompt window by pressing Windows + R on the keyboard and entering *cmd* into the Run dialog. Within the Command Prompt window, enter:

```
echo %Path%
```

The returned path variable value should include the paths to the Android SDK platform tools folders. Verify that the *platform-tools* value is correct by attempting to run the *adb* tool as follows:

```
adb
```

The tool should output a list of command-line options when executed.

Similarly, check the *tools* path setting by attempting to run the AVD Manager command-line tool (don't worry if the avdmanager tool reports a problem with Java - this will be addressed later):

```
avdmanager
```

If a message similar to the following message appears for one or both of the commands, it is most likely that an

incorrect path was appended to the Path environment variable:

```
'adb' is not recognized as an internal or external command,
operable program or batch file.
```

2.6.2 Windows 10

Right-click on the Start menu, select Settings from the resulting menu and enter "Edit the system environment variables" into the *Find a setting* text field. In the System Properties dialog, click the *Environment Variables...* button. Follow the steps outlined for Windows 8.1 starting from step 3.

2.6.3 Windows 11

Right-click on the Start icon located in the taskbar and select Settings from the resulting menu. When the Settings dialog appears, scroll down the list of categories and select the "About" option. In the About screen, select *Advanced system settings* from the Related links section. When the System Properties window appears, click the *Environment Variables...* button. Follow the steps outlined for Windows 8.1 starting from step 3.

2.6.4 Linux

This configuration can be achieved on Linux by adding a command to the *.bashrc* file in your home directory (specifics may differ depending on the particular Linux distribution in use). Assuming that the Android SDK bundle package was installed into */home/demo/Android/sdk*, the export line in the *.bashrc* file would read as follows:

```
export PATH=/home/demo/Android/sdk/platform-tools:/home/demo/Android/sdk/cmdline-tools/latest/bin:/home/demo/android-studio/bin:$PATH
```

Note also that the above command adds the *android-studio/bin* directory to the PATH variable. This will enable the *studio.sh* script to be executed regardless of the current directory within a terminal window.

2.6.5 macOS

Several techniques may be employed to modify the $PATH environment variable on macOS. Arguably the cleanest method is to add a new file in the */etc/paths.d* directory containing the paths to be added to $PATH. Assuming an Android SDK installation location of */Users/demo/Library/Android/sdk*, the path may be configured by creating a new file named *android-sdk* in the */etc/paths.d* directory containing the following lines:

```
/Users/demo/Library/Android/sdk/cmdline-tools/latest/bin
/Users/demo/Library/Android/sdk/platform-tools
```

Note that since this is a system directory, it will be necessary to use the *sudo* command when creating the file. For example:

```
sudo vi /etc/paths.d/android-sdk
```

2.7 Android Studio memory management

Android Studio is a large and complex software application with many background processes. Although Android Studio has been criticized in the past for providing less than optimal performance, Google has made significant performance improvements in recent releases and continues to do so with each new version. These improvements include allowing the user to configure the amount of memory used by both the Android Studio IDE and the background processes used to build and run apps. This allows the software to take advantage of systems with larger amounts of RAM.

If you are running Android Studio on a system with sufficient unused RAM to increase these values (this feature is only available on 64-bit systems with 5GB or more of RAM) and find that Android Studio performance appears to be degraded, it may be worth experimenting with these memory settings. Android Studio may also notify you that performance can be increased via a dialog similar to the one shown below:

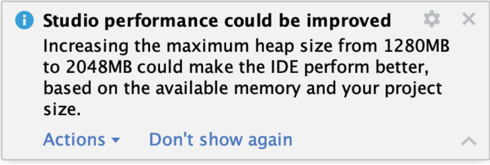


Figure 2-11

To view and modify the current memory configuration, select the *File -> Settings...* main menu option (*Android Studio -> Settings...* on macOS) and, in the resulting dialog, select *Appearance & Behavior* followed by the *Memory Settings* option listed under *System Settings* in the left-hand navigation panel, as illustrated in Figure 2-12 below:

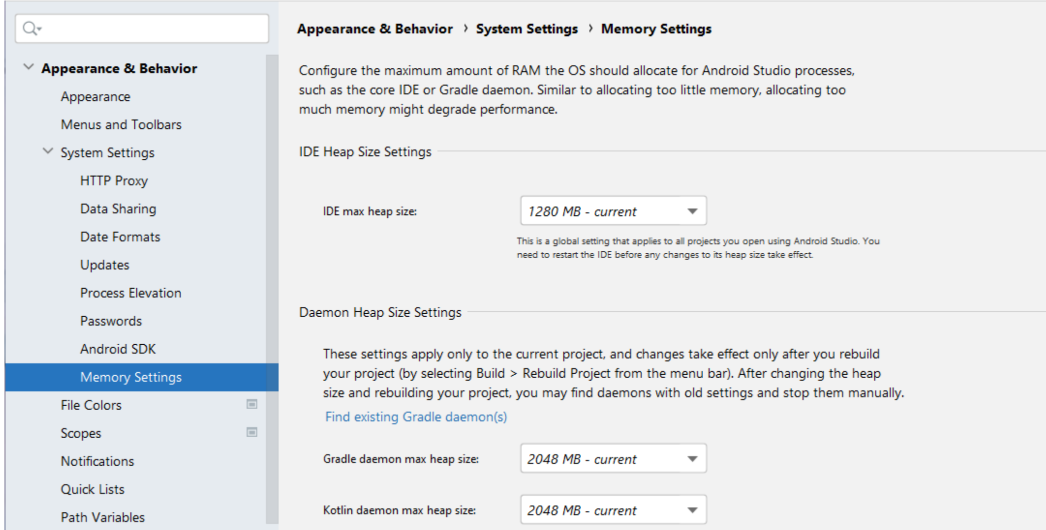


Figure 2-12

When changing the memory allocation, be sure not to allocate more memory than necessary or than your system can spare without slowing down other processes.

The IDE heap size setting adjusts the memory allocated to Android Studio and applies regardless of the currently loaded project. On the other hand, when a project is built and run from within Android Studio, several background processes (referred to as daemons) perform the task of compiling and running the app. When compiling and running large and complex projects, build time could be improved by adjusting the daemon heap settings. Unlike the IDE heap settings, these daemon settings apply only to the current project and can only be accessed when a project is open in Android Studio. To display the SDK Manager from within an open project, select the *Tools -> SDK Manager...* menu option from the main menu.

2.8 Updating Android Studio and the SDK

From time to time, new versions of Android Studio and the Android SDK are released. New versions of the SDK are installed using the Android SDK Manager. Android Studio will typically notify you when an update is ready to be installed.

To manually check for Android Studio updates, use the Help -> Check for Updates... menu option from the Android Studio main window (Android Studio -> Check for Updates... on macOS).

2.9 Summary

Before beginning the development of Android-based applications, the first step is to set up a suitable development environment. This consists of the Android SDKs and Android Studio IDE (which also includes the OpenJDK development environment). This chapter covers the steps necessary to install these packages on Windows, macOS, and Linux.

3. Creating an Example Android App in Android Studio

The preceding chapters of this book have covered the steps necessary 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:

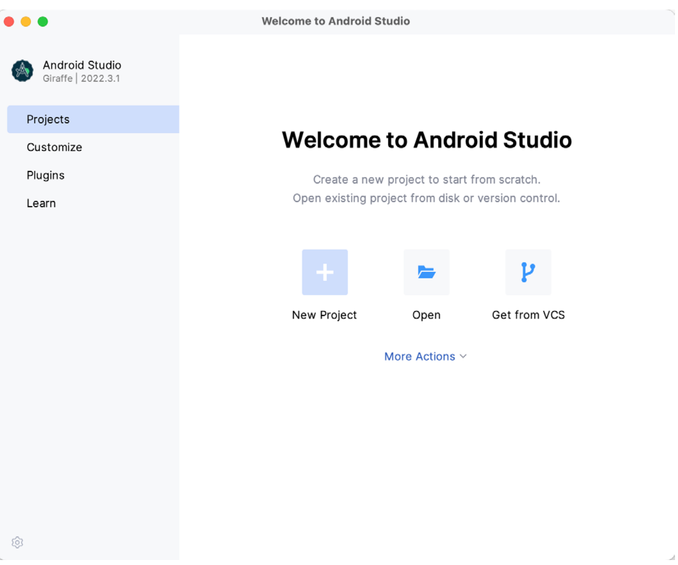


Figure 3-1

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.

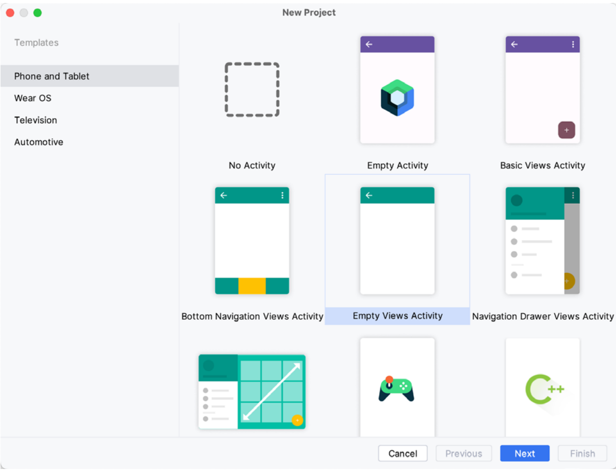


Figure 3-2

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:

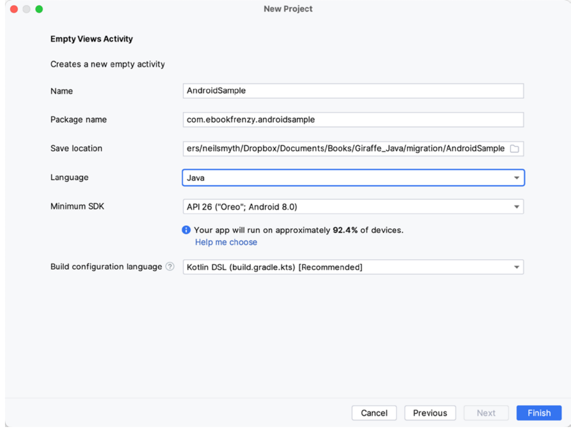


Figure 3-3

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 Enabling the New Android Studio UI

Android Studio is transitioning to a new, modern user interface that is not enabled by default in the Giraffe version. If your installation of Android Studio resembles Figure 3-4 below, then you will need to enable the new UI before proceeding:

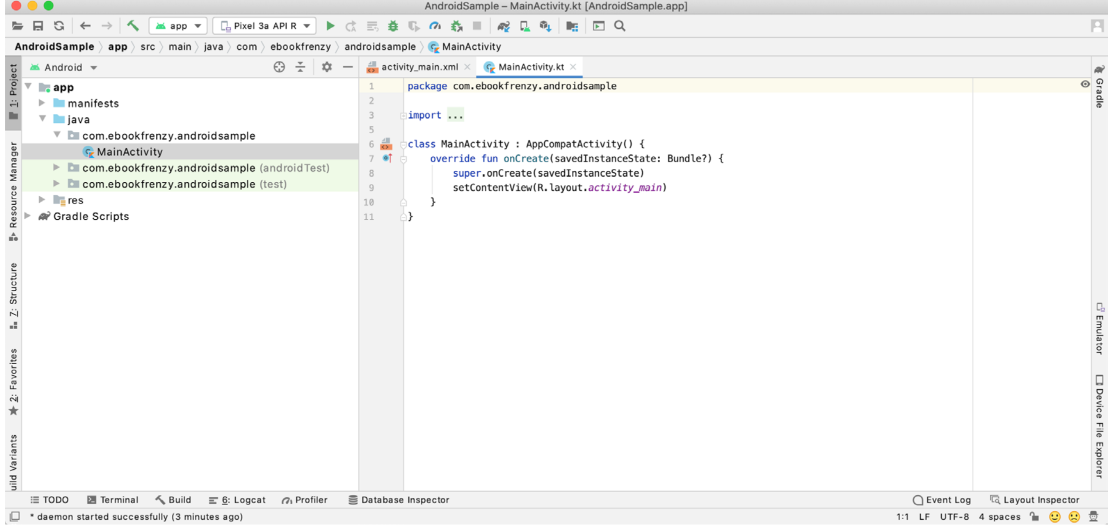


Figure 3-4

Enable the new UI by selecting the *File -> Settings...* menu option (*Android Studio -> Settings...* on macOS) and selecting the New UI option under Appearance and Behavior in the left-hand panel. From the main panel, turn on the *Enable new UI* checkbox before clicking Apply, followed by OK to commit the change:

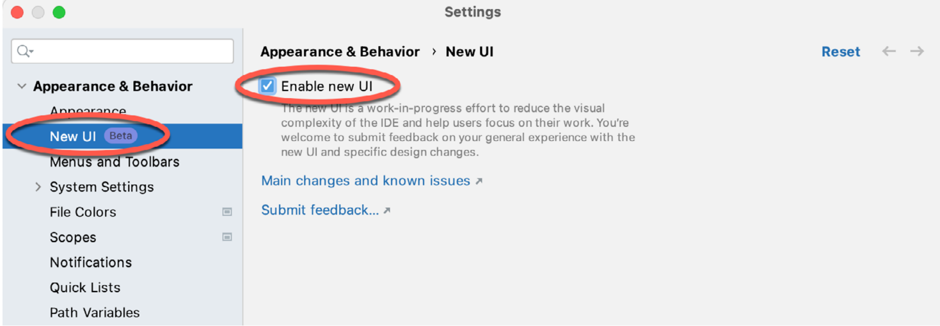


Figure 3-5

When prompted, restart Android Studio to activate the new user interface.

3.6 Modifying the Example Application

Once Android Studio has restarted, the main window will reappear using the new UI and containing our AndroidSample project as illustrated in Figure 3-6 below:

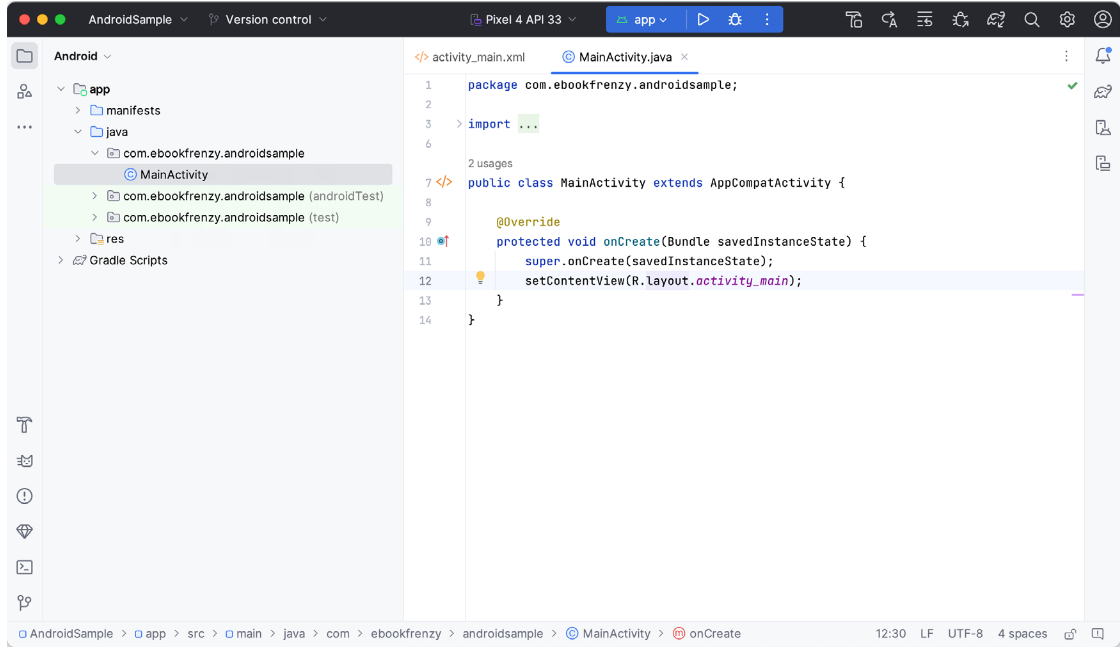


Figure 3-6

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 3-7. If the panel is not currently in Android mode, use the menu to switch mode:

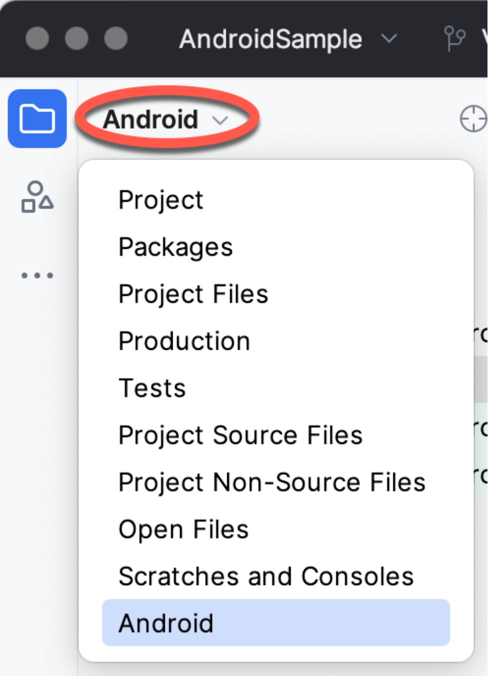


Figure 3-7

3.7 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:

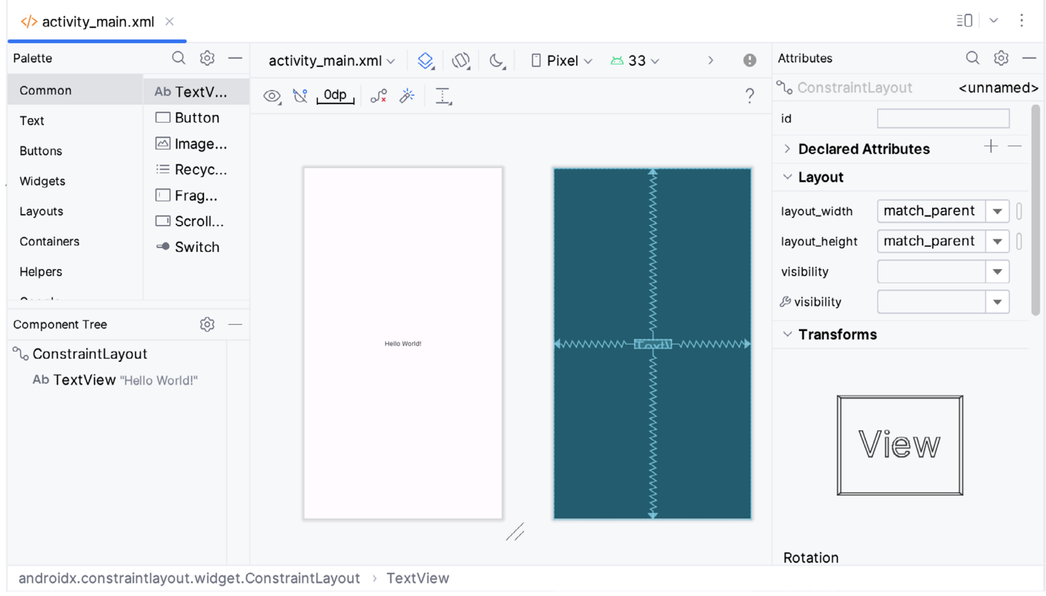


Figure 3-8

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 3-9:

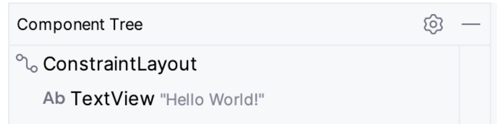


Figure 3-9

As we can see from the component tree hierarchy, the user interface layout consists of a ConstraintLayout parent 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 3-10). If necessary, re-enable Autoconnect mode by clicking on this button.

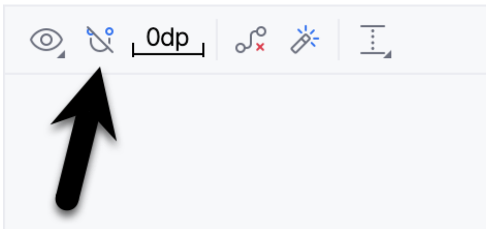


Figure 3-10

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 3-11, for example, the Button view is currently selected within the Buttons category:

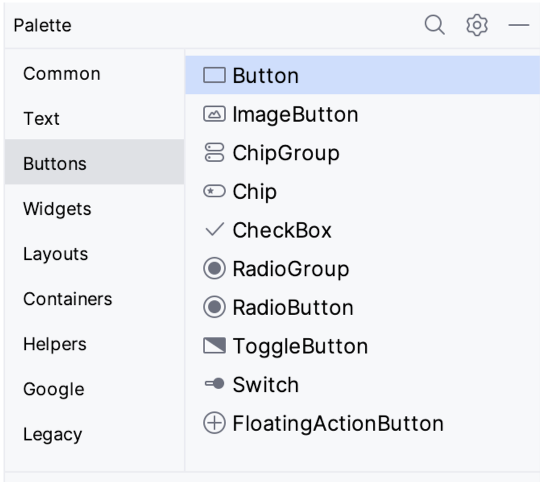


Figure 3-11

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:

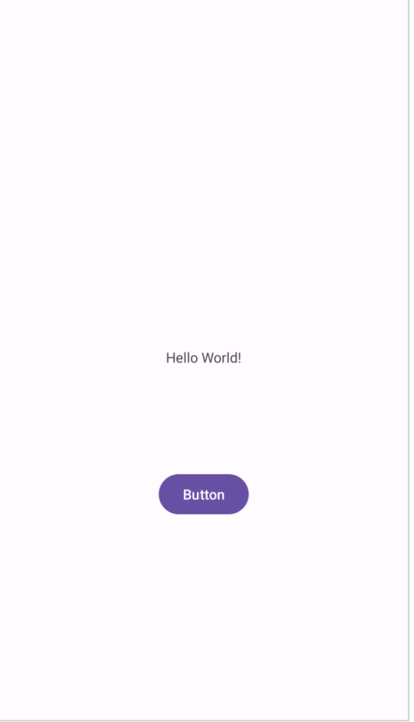


Figure 3-12

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 3-13:

strokeWidth
cornerRadius @null
rippleColor @color/m3_button_rip
text Convert
text
contentDescription

Figure 3-13

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 3-14) to add any missing constraints to the layout:

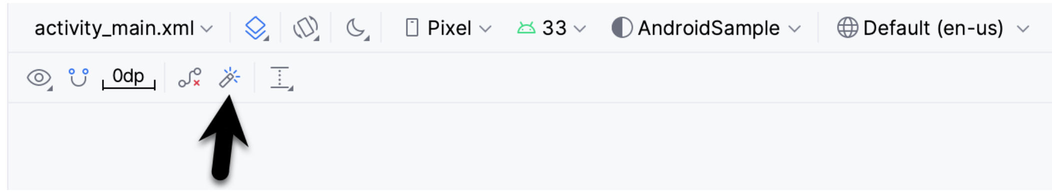


Figure 3-14

It is important to explain the warning button in the top right-hand corner of the Layout Editor tool, as indicated in Figure 3-15. This warning indicates potential problems with the layout. For details on any problems, click on the button:

Figure 3-15

When clicked, the Problems tool window (Figure 3-16) will appear, describing the nature of the problems:

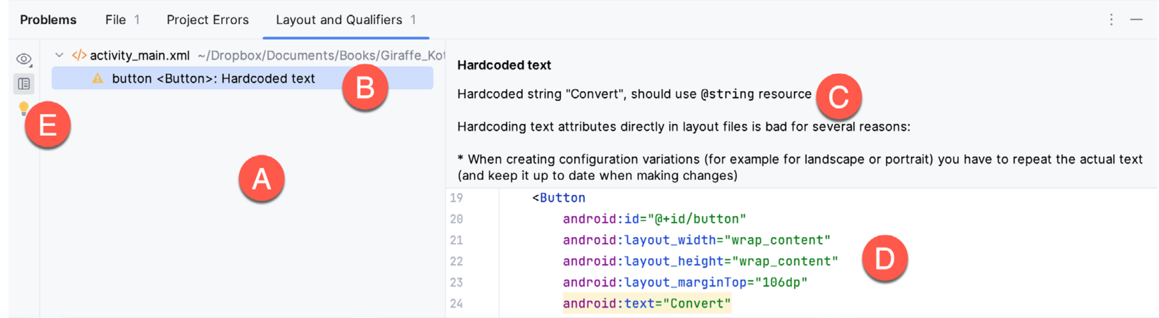


Figure 3-16

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 3-17:

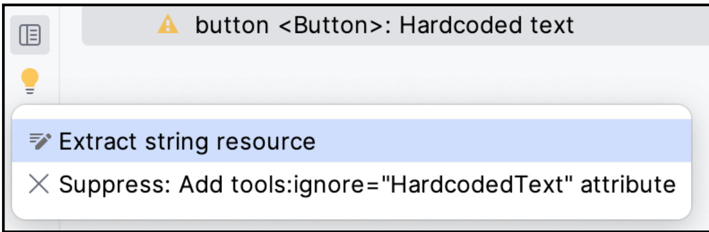


Figure 3-17

After selecting this option, the *Extract Resource* panel (Figure 3-18) 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:

Extract Resource
Resource name: convert_string
Resource value: Convert
Source set: main src/main/res
File name: strings.xml
Create the resource in directories:
values
values-night
Cancel OK

Figure 3-18

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 3-19:

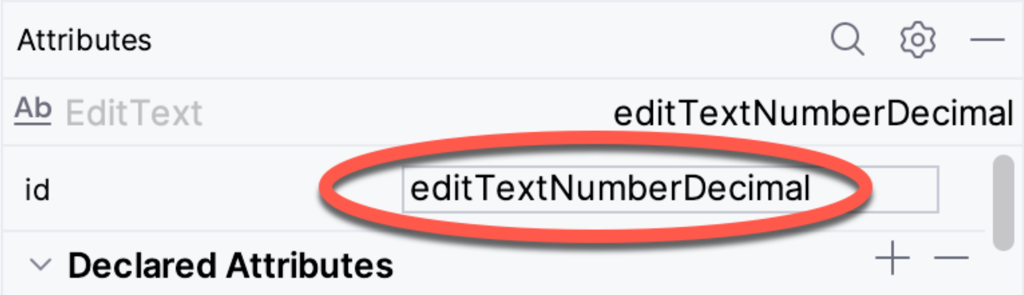


Figure 3-19

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:

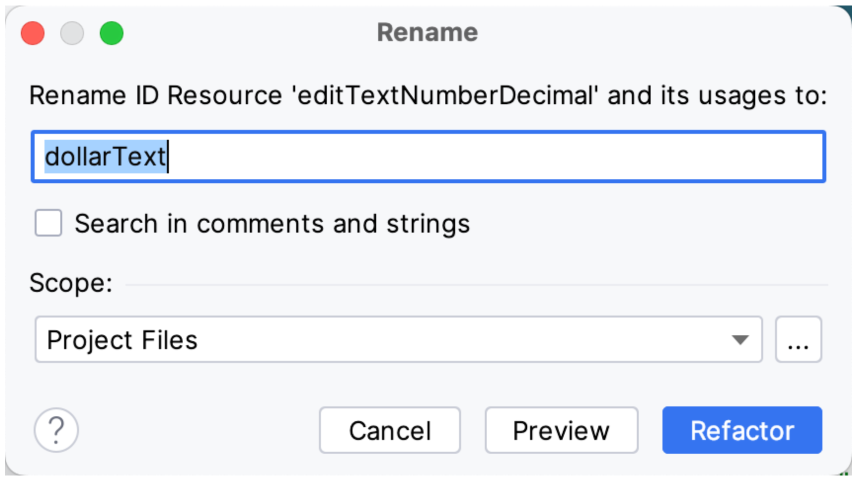


Figure 3-20

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 3-21:

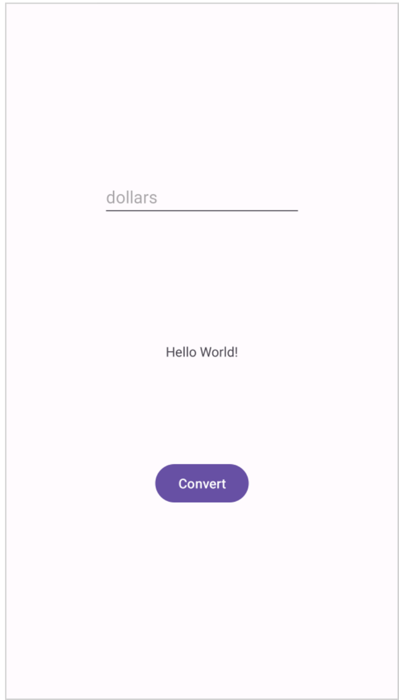


Figure 3-21

3.8 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 is the View Modes menu button marked A in Figure 3-22 below:

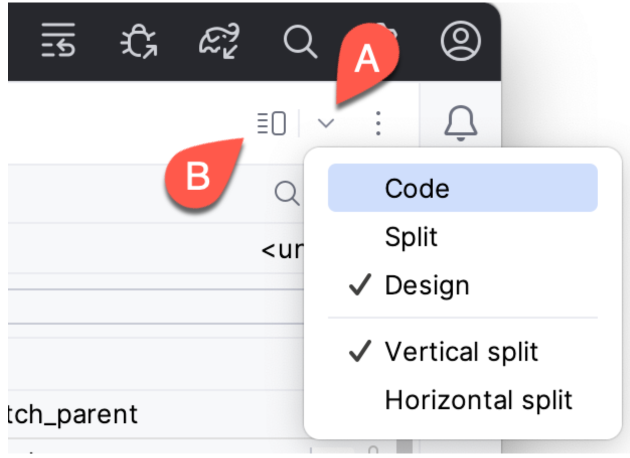


Figure 3-22

By default, the editor will be in *Design* mode, whereby just the visual representation of the layout is displayed.

In *Code* mode, the editor will display the XML for the layout, while in *Split* mode, both the layout and XML are displayed, as shown in Figure 3-23:

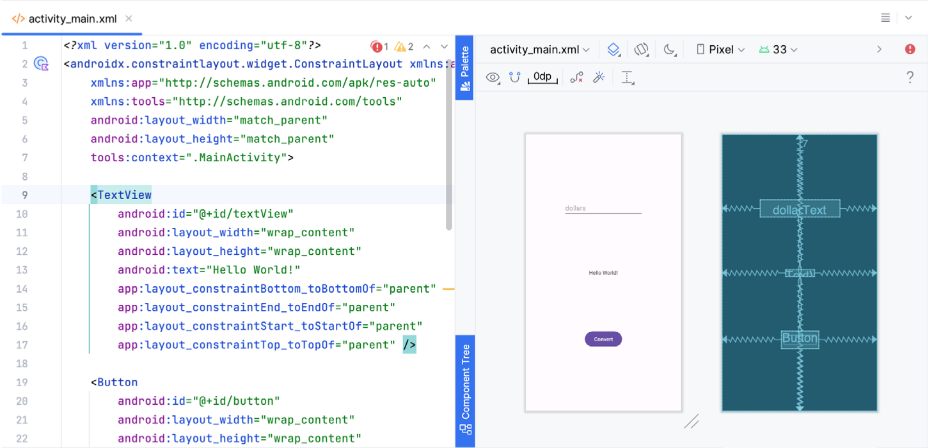


Figure 3-23

The button to the left of the View Modes button (marked B in Figure 3-22 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: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:

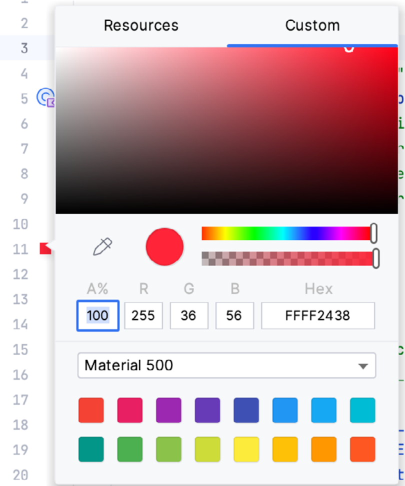


Figure 3-24

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="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:

activity_main.xml | strings.xml | Translations Editor

Show All Keys | Show All Locales

Key	Resource Folder	Untranslatable	Default Value
app_name	app/src/main/res	☐	AndroidSample
convert_string	app/src/main/res	☐	Convert
dollars_hint	app/src/main/res	☐	dollars
no_value_string	app/src/main/res	☐	No Value

XML:
Key:
Default value:
Translation:

Figure 3-25

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

3.9 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:

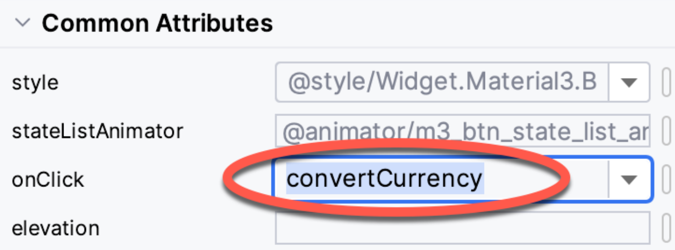


Figure 3-26

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.ebookfrenzy.androidsample;

import androidx.appcompat.app.AppCompatActivity;

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);
        setContentView(R.layout.activity_main);
    }

    public void convertCurrency(View view) {

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

        if (!dollarText.getText().toString().equals("")) {

            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"*.

3.10 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.

Chapter 11

11. An Overview of Android View Binding

An essential part of developing Android apps involves the interaction between the code and the views that make up the user interface layouts. This chapter will look at the options available for gaining access to layout views in code, emphasizing an option known as view binding. Once the basics of view bindings have been covered, the chapter will outline how to convert the AndroidSample project to use this approach.

11.1 Find View by Id

As outlined in the chapter entitled *"The Anatomy of an Android Application"*, all of the resources that make up an application are compiled into a class named *R*. Amongst those resources are those that define layouts. Within the R class is a subclass named *layout*, which contains the layout resources, including the views that make up the user interface. Most apps will need to implement interaction between the code and these views, for example, when reading the value entered into the EditText view or changing the content displayed on a TextView.

Before the introduction of Android Studio 3.6, the most common option for gaining access to a view from within the app code involved writing code to manually find a view based on its id via the *findViewById()* method. For example:

```
TextView exampleView = findViewById(R.id.exampleView);
```

With the reference obtained, the view's properties can then be accessed. For example:

```
exampleView.setText("Hello");
```

While finding views by id is still a viable option, it has some limitations, the most significant disadvantage of *findViewById()* being that it is possible to obtain a reference to a view that has not yet been created within the layout, leading to a null pointer exception when an attempt is made to access the view's properties.

Since Android Studio 3.6, an alternative way of accessing views from the app code has been available in the form of *view binding*.

11.2 View Binding

When view binding is enabled in an app module, Android Studio automatically generates a binding class for each layout file. The layout views can be accessed from within the code using this binding class without using *findViewById()*.

The name of the binding class generated by Android Studio is based on the layout file name converted to so-called "camel case" with the word "Binding" appended to the end. For the *activity_main.xml* file, for example, the binding class will be called ActivityMainBinding.

Android Studio Giraffe is inconsistent in using view bindings within project templates. For example, the Empty Views Activity template used when we created the AndroidSample project does not use view bindings. The Basic Views Activity template, on the other hand, is implemented using view binding. If you use a template that does not use view binding, it is important to know how to add it to your project.

11.3 Converting the AndroidSample project

In the remainder of this chapter, we will practice migrating to view bindings by converting the AndroidSample project to use view binding instead of *findViewById()*.

Begin by launching Android Studio and opening the AndroidSample project created in the chapter entitled *"Creating an Example Android App in Android Studio"*.

11.4 Enabling View Binding

To use view binding, some changes must first be made to the *build.gradle.kts* file for each module in which view binding is needed. In the case of the AndroidSample project, this will require a slight change to the *Gradle Scripts -> build.gradle.kts (Module: app)* file. Load this file into the editor, locate the *android* section and add an entry to enable the *viewBinding* property as follows:

```
plugins {
    id("com.android.application")
}


android {


    buildFeatures {
        viewBinding = true
    }
.
.
```

Once this change has been made, click on the Sync Now link at the top of the editor panel, then use the Build menu to clean and rebuild the project to ensure the binding class is generated. The next step is to use the binding class within the code.

11.5 Using View Binding

The first step in this process is to "inflate" the view binding class to access the root view within the layout. This root view will then be used as the content view for the layout.

The logical place to perform these tasks is within the *onCreate()* method of the activity associated with the layout. A typical *onCreate()* method will read as follows:

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

To switch to using view binding, the view binding class will need to be imported and the class modified as follows. Note that since the layout file is named *activity_main.xml*, we can surmise that the binding class generated by Android Studio will be named ActivityMainBinding. Note that if you used a domain other than *com.example* when creating the project, the import statement below would need to be changed to reflect this:

```
.
.
import android.widget.EditText;
import android.widget.TextView;
```

```
.
.
import com.example.androidsample.databinding.ActivityMainBinding;
.
.
public class MainActivity extends AppCompatActivity {

    private ActivityMainBinding binding;
.
.
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    binding = ActivityMainBinding.inflate(getLayoutInflater());
    View view = binding.getRoot();
    setContentView(view);
}
```

Now that we have a reference to the binding, we can access the views by name as follows:

```
public void convertCurrency(View view) {

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

    if (!binding.dollarText.getText().toString().equals("")) {

        Float dollarValue = Float.valueOf(
                    binding.dollarText.getText().toString());
        Float euroValue = dollarValue * 0.85F;
        binding.textView.setText(String.format(Locale.ENGLISH,"%.2f",
                                euroValue));
    } else {
        binding.textView.setText(R.string.no_value_string);
    }
}
```

Compile and run the app and verify that the currency conversion process works as before.

11.6 Choosing an Option

Notwithstanding their failure to adopt view bindings in the Empty Views Activity project template, Google strongly recommends using view binding wherever possible. Therefore, view binding should be used when developing your own projects.

11.7 View Binding in the Book Examples

Any chapters in this book that rely on a project template that does not implement view binding will first be migrated. Instead of replicating the steps every time a migration needs to be performed, however, these chapters

will refer you back here to refresh your memory (don't worry, after a few chapters, the necessary changes will become second nature). To help with the process, the following section summarizes the migration steps more concisely.

11.8 Migrating a Project to View Binding

The process for converting a project module to use view binding involves the following steps:

1. Edit the module-level Gradle build script file listed in the Project tool window as *Gradle Scripts -> build.gradle.kts (Module :app)* where *<project name>* is the name of the project (for example AndroidSample).

2. Locate the *android* section of the file and add an entry to enable the *viewBinding* property as follows:

```
android {

    buildFeatures {
        viewBinding = true
    }
.
.
```

3. Click on the *Sync Now* link at the top of the editor to resynchronize the project with these new build settings.

4. Edit the *MainActivity.java* file and modify it to read as follows (where *<reverse domain>* represents the domain name used when the project was created and *<project name>* is replaced by the lowercase name of the project, for example, *androidsample*) and *<binding name>* is the name of the binding for the corresponding layout resource file (for example, the binding for *activity_main.xml* is ActivityMainBinding).

```
.
.
import android.view.View;

import com.<reverse domain>.<project name>.databinding.<binding name>;

public class MainActivity extends AppCompatActivity {

    private <binding name> binding;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        binding = <binding name>.inflate(getLayoutInflater());
        View view = binding.getRoot();
        setContentView(view);
    }
```

5. Access views by name as properties of the binding object.

11.9 Summary

Before the introduction of Android Studio 3.6, access to layout views from within the code of an app involved using the *findViewById()* method. An alternative is now available in the form of view bindings. View bindings

consist of classes Android Studio automatically generates for each XML layout file. These classes contain bindings to each view in the corresponding layout, providing a safer option than the *findViewById()* method. However, as of Android Studio Giraffe, view bindings are not enabled by default in some project templates. Additional steps are required to enable and configure support within each project module manually.

Chapter 12

12. Understanding Android Application and Activity Lifecycles

In earlier chapters, we learned that Android applications run within processes and comprise multiple components in the form of activities, services, and broadcast receivers. This chapter aims to expand on this knowledge by looking at the lifecycle of applications and activities within the Android runtime system.

Regardless of the fanfare about how much memory and computing power resides in the mobile devices of today compared to the desktop systems of yesterday, it is important to keep in mind that these devices are still considered to be "resource constrained" by the standards of modern desktop and laptop-based systems, particularly in terms of memory. As such, a key responsibility of the Android system is to ensure that these limited resources are managed effectively and that the operating system and the applications running on it remain responsive to the user at all times. To achieve this, Android is given complete control over the lifecycle and state of the processes in which the applications run and the individual components that comprise those applications.

An important factor in developing Android applications, therefore, is to understand Android's application and activity lifecycle management models of Android, and how an application can react to the state changes likely to be imposed upon it during its execution lifetime.

12.1 Android Applications and Resource Management

The operating system views each running Android application as a separate process. If the system identifies that resources on the device are reaching capacity, it will take steps to terminate processes to free up memory.

When determining which process to terminate to free up memory, the system considers both the *priority* and *state* of all currently running processes, combining these factors to create what is referred to by Google as an *importance hierarchy*. Processes are then terminated, starting with the lowest priority and working up the hierarchy until sufficient resources have been liberated for the system to function.

12.2 Android Process States

Processes host applications, and applications are made up of components. Within an Android system, the current state of a process is defined by the highest-ranking active component within the application it hosts. As outlined in Figure 12-1, a process can be in one of the following five states at any given time:

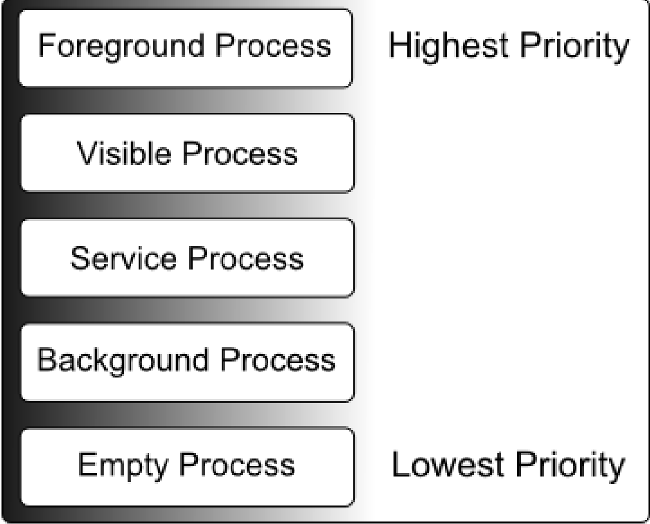


Figure 12-1

12.2.1 Foreground Process

These processes are assigned the highest level of priority. At any one time, there are unlikely to be more than one or two foreground processes active, which are usually the last to be terminated by the system. A process must meet one or more of the following criteria to qualify for foreground status:

- Hosts an activity with which the user is currently interacting.
- Hosts a Service connected to the activity with which the user is interacting.
- Hosts a Service that has indicated, via a call to *startForeground()*, that termination would disrupt the user experience.
- Hosts a Service executing either its *onCreate()*, *onResume()*, or *onStart()* callbacks.
- Hosts a Broadcast Receiver that is currently executing its *onReceive()* method.

12.2.2 Visible Process

A process containing an activity that is visible to the user but is not the activity with which the user is interacting is classified as a "visible process". This is typically the case when an activity in the process is visible to the user, but another activity, such as a partial screen or dialog, is in the foreground. A process is also eligible for visible status if it hosts a Service that is, itself, bound to a visible or foreground activity.

12.2.3 Service Process

Processes that contain a Service that has already been started and is currently executing.

12.2.4 Background Process

A process that contains one or more activities that are not currently visible to the user and does not host a Service that qualifies for *Service Process* status. Processes that fall into this category are at high risk of termination if additional memory needs to be freed for higher-priority processes. Android maintains a dynamic list of background processes, terminating processes in chronological order such that processes that were the least recently in the foreground are killed first.

12.2.5 Empty Process

Empty processes no longer contain active applications and are held in memory, ready to serve as hosts for newly launched applications. This is analogous to keeping the doors open and the engine running on a bus in anticipation of passengers arriving. Such processes are considered the lowest priority and are the first to be killed to free up resources.

12.3 Inter-Process Dependencies

Determining the highest priority process is more complex than outlined in the preceding section because processes can often be interdependent. As such, when determining the priority of a process, the Android system will also consider whether the process is in some way serving another process of higher priority (for example, a service process acting as the content provider for a foreground process). As a basic rule, the Android documentation states that a process can never be ranked lower than another process that it is currently serving.

12.4 The Activity Lifecycle

As we have previously determined, the state of an Android process is primarily determined by the status of the activities and components that make up the application it hosts. It is important to understand, therefore, that these activities also transition through different states during the execution lifetime of an application. The current state of an activity is determined, in part, by its position in something called the Activity Stack.

12.5 The Activity Stack

The runtime system maintains an *Activity Stack* for each application running on an Android device. When an application is launched, the first of the application's activities to be started is placed onto the stack. When a second activity is started, it is placed on the top of the stack, and the previous activity is *pushed* down. The activity at the top of the stack is called the *active (*or *running*) activity. When the active activity exits, it is *popped* off the stack by the runtime and the activity located immediately beneath it in the stack becomes the current active activity. For example, the activity at the top of the stack might exit because the task for which it is responsible has been completed. Alternatively, the user may have selected a "Back" button on the screen to return to the previous activity, causing the current activity to be popped off the stack by the runtime system and destroyed. A visual representation of the Android Activity Stack is illustrated in Figure 12-2.

As shown in the diagram, new activities are pushed onto the top of the stack when they are started. The current active activity is located at the top of the stack until it is either pushed down the stack by a new activity or popped off the stack when it exits or the user navigates to the previous activity. If resources become constrained, the runtime will kill activities, starting with those at the bottom of the stack.

The Activity Stack is what is referred to in programming terminology as a Last-In-First-Out (LIFO) stack in that the last item to be pushed onto the stack is the first to be popped off.

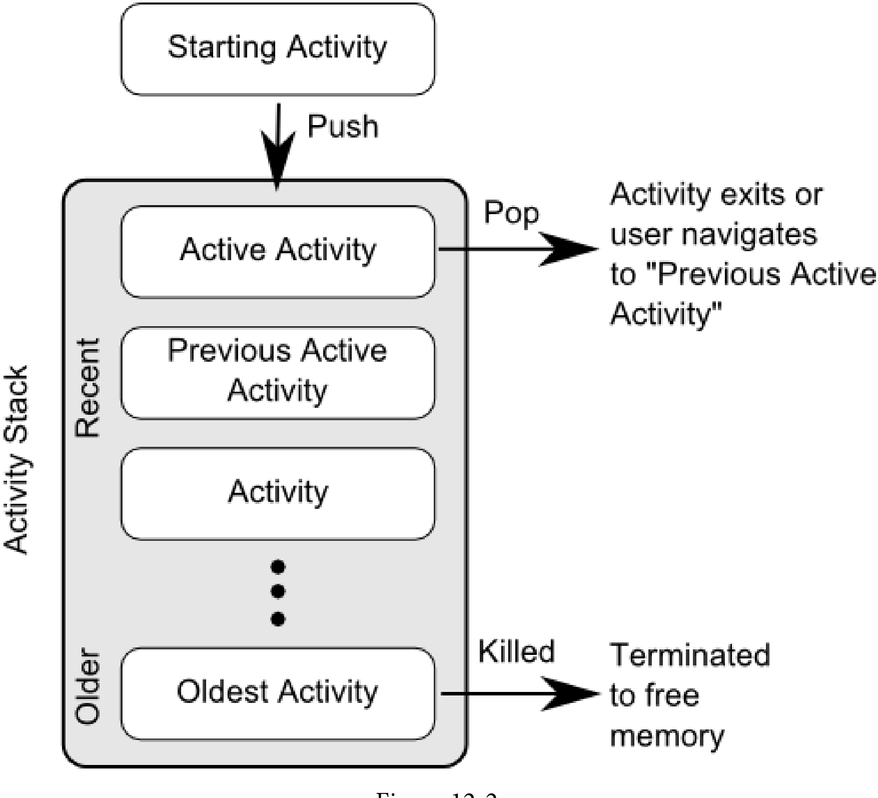


Figure 12-2

12.6 Activity States

An activity can be in one of several states during the course of its execution within an application:

- **Active / Running** – The activity is at the top of the Activity Stack, is the foreground task visible on the device screen, has focus, and is currently interacting with the user. This is the least likely activity to be terminated in the event of a resource shortage.
- **Paused** – The activity is visible to the user but does not currently have focus (typically because the current *active* activity partially obscures this activity). Paused activities are held in memory, remain attached to the window manager, retain all state information, and can quickly be restored to active status when moved to the top of the Activity Stack.
- **Stopped** – The activity is currently not visible to the user (in other words, it is obscured on the device display by other activities). As with paused activities, it retains all state and member information but is at higher risk of termination in low-memory situations.
- **Killed** – The runtime system has terminated the activity to free up memory and is no longer present on the Activity Stack. Such activities must be restarted if required by the application.

12.7 Configuration Changes

So far in this chapter, we have looked at two causes for the change in the state of an Android activity, namely the movement of an activity between the foreground and background and the termination of an activity by the runtime system to free up memory. In fact, there is a third scenario in which the state of an activity can dramatically change, which involves a change to the device configuration.

By default, any configuration change that impacts the appearance of an activity (such as rotating the orientation of the device between portrait and landscape or changing a system font setting) will cause the activity to be destroyed and recreated. The reasoning behind this is that such changes affect resources such as the layout of the user interface, and destroying and recreating impacted activities is the quickest way for an activity to respond to the configuration change. It is, however, possible to configure an activity so that the system does not restart it in response to specific configuration changes.

12.8 Handling State Change

It should be clear from this chapter that an application and, by definition, the components contained therein will transition through many states during its lifespan. Of particular importance is the fact that these state changes (up to and including complete termination) are imposed upon the application by the Android runtime subject to the user's actions and the availability of resources on the device.

In practice, however, these state changes are not imposed entirely without notice, and an application will, in most circumstances, be notified by the runtime system of the changes and given the opportunity to react accordingly. This will typically involve saving or restoring both internal data structures and user interface state, thereby allowing the user to switch seamlessly between applications and providing at least the appearance of multiple concurrently running applications.

Android provides two ways to handle the changes to the lifecycle states of the objects within an app. One approach involves responding to state change method calls from the operating system and is covered in detail in the next chapter entitled *"Handling Android Activity State Changes"*.

A new approach that Google recommends involves the lifecycle classes included with the Jetpack Android Architecture components, introduced in *"Modern Android App Architecture with Jetpack"* and explained in more detail in the chapter entitled *"Working with Android Lifecycle-Aware Components"*.

12.9 Summary

Mobile devices are typically considered to be resource constrained, particularly in terms of onboard memory capacity. Consequently, a prime responsibility of the Android operating system is to ensure that applications, and the operating system in general, remain responsive to the user.

Applications are hosted on Android within processes. Each application, in turn, comprises components in the form of activities and Services.

The Android runtime system has the power to terminate both processes and individual activities to free up memory. Process state is considered by the runtime system when deciding whether a process is a suitable candidate for termination. The state of a process largely depends upon the status of the activities hosted by that process.

The key message of this chapter is that an application moves through various states during its execution lifespan and has very little control over its destiny within the Android runtime environment. Those processes and activities not directly interacting with the user run a higher risk of termination by the runtime system. An essential element of Android application development, therefore, involves the ability of an application to respond to state change notifications from the operating system.

Chapter 16

16. Understanding Android Views, View Groups and Layouts

With the possible exception of listening to streaming audio, a user's interaction with an Android device is primarily visual and tactile. All of this interaction occurs through the user interfaces of the applications installed on the device, including both the built-in applications and any third-party applications installed by the user. Therefore, it should come as no surprise that a critical element of developing Android applications involves designing and creating user interfaces.

This chapter covers the Android user interface structure, including an overview of the elements that can be combined to make up a user interface: Views, View Groups, and Layouts.

16.1 Designing for Different Android Devices

The term "Android device" covers many tablet and smartphone products with different screen sizes and resolutions. As a result, application user interfaces must now be carefully designed to ensure correct presentation on as wide a range of display sizes as possible. A key part of this is ensuring that the user interface layouts resize correctly when run on different devices. This can largely be achieved through careful planning and using the layout managers outlined in this chapter.

It is also essential to remember that most Android-based smartphones and tablets can be held by the user in both portrait and landscape orientations. A well-designed user interface should be able to adapt to such changes and make sensible layout adjustments to utilize the available screen space in each orientation.

16.2 Views and View Groups

Every item in a user interface is a subclass of the Android *View* class (to be precise *android.view.View*). The Android SDK provides a set of pre-built views that can be used to construct a user interface. Typical examples include standard items such as the Button, CheckBox, ProgressBar, and TextView classes. Such views are also referred to as *widgets* or *components*. For requirements not met by the widgets supplied with the SDK, new views may be created by subclassing and extending an existing class or creating an entirely new component by building directly on top of the View class.

A view can also comprise multiple other views (otherwise known as a *composite view*). Such views are subclassed from the Android *ViewGroup* class (*android.view.ViewGroup*), which is itself a subclass of *View*. An example of such a view is the RadioGroup, which is intended to contain multiple RadioButton objects such that only one can be in the "on" position at any one time. Regarding structure, composite views consist of a single parent view (derived from the ViewGroup class and otherwise known as a *container view* or *root element)* capable of containing other views (known as *child views*).

Another category of ViewGroup-based container view is that of the layout manager.

16.3 Android Layout Managers

In addition to the widget style views discussed in the previous section, the SDK also includes a set of views referred to as *layouts*. Layouts are container views (and, therefore, subclassed from ViewGroup) designed to control how child views are positioned on the screen.

The Android SDK includes the following layout views that may be used within an Android user interface design:

- **ConstraintLayout** – Introduced in Android 7, this layout manager is recommended for most layout requirements. ConstraintLayout allows the positioning and behavior of the views in a layout to be defined by simple constraint settings assigned to each child view. The flexibility of this layout allows complex layouts to be quickly and easily created without the necessity to nest other layout types inside each other, resulting in improved layout performance. ConstraintLayout is also tightly integrated into the Android Studio Layout Editor tool. Unless otherwise stated, this is the layout of choice for most of examples in this book.

- **LinearLayout** – Positions child views in a single row or column depending on the orientation selected. A *weight* value can be set on each child to specify how much of the layout space that child should occupy relative to other children.

- **TableLayout** – Arranges child views into a grid format of rows and columns. Each row within a table is represented by a *TableRow* object child, which, in turn, contains a view object for each cell.

- **FrameLayout** – The purpose of the FrameLayout is to allocate an area of the screen, typically to display a single view. If multiple child views are added, they will, by default, appear on top of each other and be positioned in the top left-hand corner of the layout area. Alternate positioning of individual child views can be achieved by setting gravity values on each child. For example, setting a *center_vertical* gravity value on a child will cause it to be positioned in the vertical center of the containing FrameLayout view.

- **RelativeLayout** – The RelativeLayout allows child views to be positioned relative to each other and the containing layout view through the specification of alignments and margins on child views. For example, child *View A* may be configured to be positioned in the vertical and horizontal center of the containing RelativeLayout view. *View B*, on the other hand, might also be configured to be centered horizontally within the layout view but positioned 30 pixels above the top edge of *View A*, thereby making the vertical position *relative* to that of *View A*. The RelativeLayout manager can be helpful when designing a user interface that must work on various screen sizes and orientations.

- **AbsoluteLayout** – Allows child views to be positioned at specific X and Y coordinates within the containing layout view. Using this layout is discouraged since it lacks the flexibility to respond to screen size and orientation changes.

- **GridLayout** – A GridLayout instance is divided by invisible lines that form a grid containing rows and columns of cells. Child views are then placed in cells and may be configured to cover multiple cells horizontally and vertically, allowing a wide range of layout options to be quickly and easily implemented. Gaps between components in a GridLayout may be implemented by placing a special type of view called a *Space* view into adjacent cells or setting margin parameters.

- **CoordinatorLayout** – Introduced as part of the Android Design Support Library with Android 5.0, the CoordinatorLayout is designed specifically for coordinating the appearance and behavior of the app bar across the top of an application screen with other view elements. When creating a new activity using the Basic Views Activity template, the parent view in the main layout will be implemented using a CoordinatorLayout instance. This layout manager will be covered in greater detail, starting with the chapter *"Working with the Floating Action Button and Snackbar"*.

When considering layouts in the user interface for an Android application, it is worth keeping in mind that, as outlined in the next section, these can be nested within each other to create a user interface design of just about any necessary level of complexity.

16.4 The View Hierarchy

Each view in a user interface represents a rectangular area of the display. A view is responsible for what is drawn in that rectangle and responding to events within that part of the screen (such as a touch event).

A user interface screen is comprised of a view hierarchy with a *root view* positioned at the top of the tree and child views positioned on branches below. The child of a container view appears on top of its parent view and is constrained to appear within the bounds of the parent view's display area. Consider, for example, the user interface illustrated in Figure 16-1:

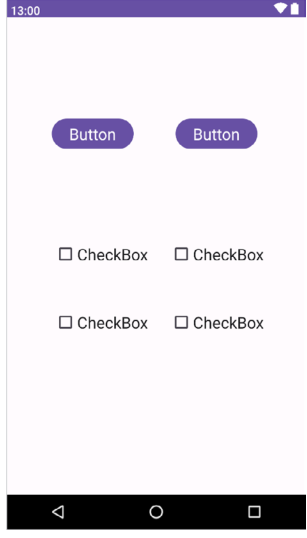


Figure 16-1

In addition to the visible button and checkbox views, the user interface actually includes a number of layout views that control how the visible views are positioned. Figure 16-2 shows an alternative view of the user interface, this time highlighting the presence of the layout views in relation to the child views:

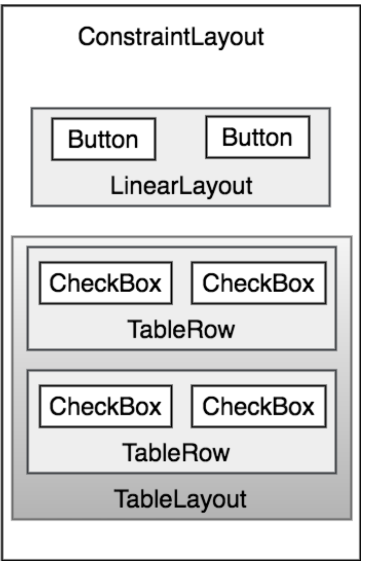


Figure 16-2

As was previously discussed, user interfaces are constructed in the form of a view hierarchy with a root view at the top. This being the case, we can also visualize the above user interface example in the form of the view tree illustrated in Figure 16-3:

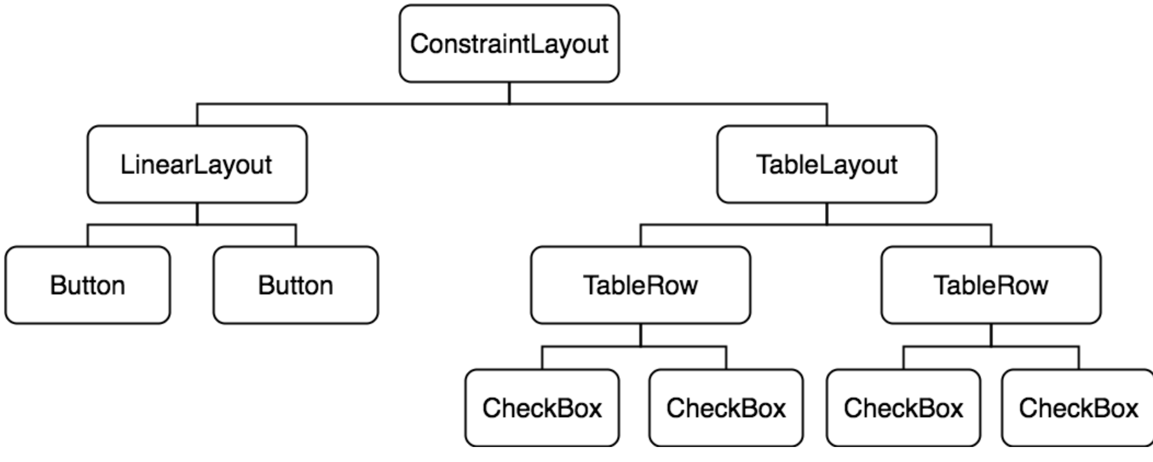


Figure 16-3

The view hierarchy diagram gives probably the clearest overview of the relationship between the various views that make up the user interface shown in Figure 16-1. When a user interface is displayed to the user, the Android runtime walks the view hierarchy, starting at the root view and working down the tree as it renders each view.

16.5 Creating User Interfaces

With a clearer understanding of the concepts of views, layouts and the view hierarchy, the following few chapters will focus on the steps involved in creating user interfaces for Android activities. In fact, there are three different approaches to user interface design: using the Android Studio Layout Editor tool, handwriting XML layout resource files or writing Java code, each of which will be covered.

16.6 Summary

Each element within a user interface screen of an Android application is a view that is ultimately subclassed from the *android.view.View* class. Each view represents a rectangular area of the device display and is responsible both for what appears in that rectangle and for handling events that take place within the view's bounds. Multiple views may be combined to create a single *composite view*. The views within a composite view are children of a *container view* which is generally a subclass of *android.view.ViewGroup* (which is itself a subclass of *android.view.View*). A user interface is comprised of views constructed in the form of a view hierarchy.

The Android SDK includes a range of pre-built views that can be used to create a user interface. These include basic components such as text fields and buttons, in addition to a range of layout managers that can be used to control the positioning of child views. If the supplied views do not meet a specific requirement, custom views may be created, either by extending or combining existing views, or by subclassing *android.view.View* and creating an entirely new class of view.

User interfaces may be created using the Android Studio Layout Editor tool, handwriting XML layout resource files or by writing Java code. Each of these approaches will be covered in the chapters that follow.

Chapter 17

17. A Guide to the Android Studio Layout Editor Tool

It is challenging to think of an Android application concept that does not require some form of user interface. Most Android devices come equipped with a touch screen and keyboard (either virtual or physical), and taps and swipes are the primary interaction between the user and the application. Invariably these interactions take place through the application's user interface.

A well-designed and implemented user interface, an essential factor in creating a successful and popular Android application, can vary from simple to highly complex, depending on the design requirements of the individual application. Regardless of the level of complexity, the Android Studio Layout Editor tool significantly simplifies the task of designing and implementing Android user interfaces.

17.1 Basic vs. Empty Views Activity Templates

As outlined in the chapter entitled *"The Anatomy of an Android Application"*, Android applications comprise one or more activities. An activity is a standalone module of application functionality that usually correlates directly to a single user interface screen. As such, when working with the Android Studio Layout Editor, we are invariably work on the layout for an activity.

When creating a new Android Studio project, several templates are available to be used as the starting point for the user interface of the main activity. The most basic templates are the Basic Views Activity and Empty Views Activity templates. Although these seem similar at first glance, there are considerable differences between the two options. To see these differences within the layout editor, use the View Options menu to enable Show System UI, as shown in Figure 17-1 below:

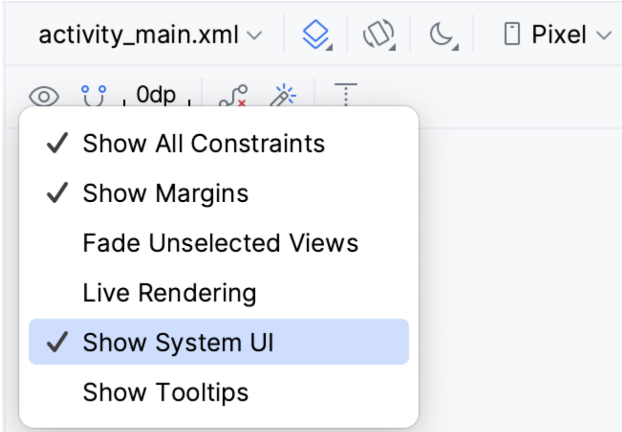


Figure 17-1

The Empty Views Activity template creates a single layout file consisting of a ConstraintLayout manager instance containing a TextView object, as shown in Figure 17-2:

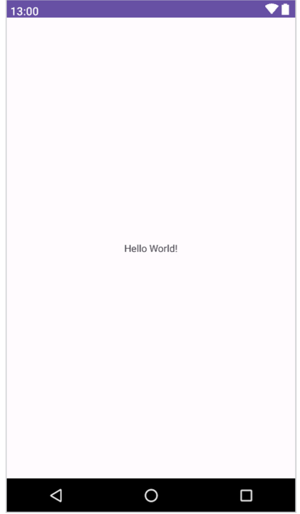


Figure 17-2

The Basic Views Activity, on the other hand, consists of multiple layout files. The top-level layout file has a CoordinatorLayout as the root view, a configurable app bar (which contains a toolbar) that appears across the top of the device screen (marked A in Figure 17-3), and a floating action button (the email button marked B). In addition to these items, the *activity_main.xml* layout file contains a reference to a second file named *content_main.xml* containing the content layout (marked C):

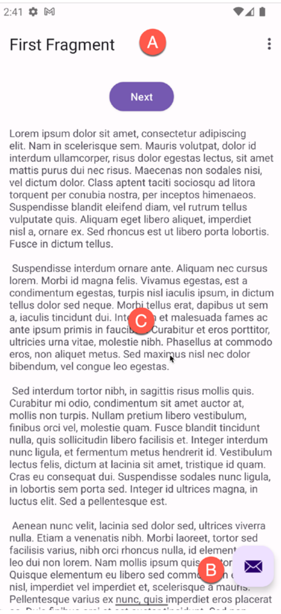


Figure 17-3

The Basic Views Activity contains layouts for two screens containing a button and a text view. This template aims to demonstrate how to implement navigation between multiple screens within an app. If an unmodified app using the Basic Views Activity template were to be run, the first of these two screens would appear (marked A in Figure 17-4). Pressing the Next button would navigate to the second screen (B), which, in turn, contains a button to return to the first screen:

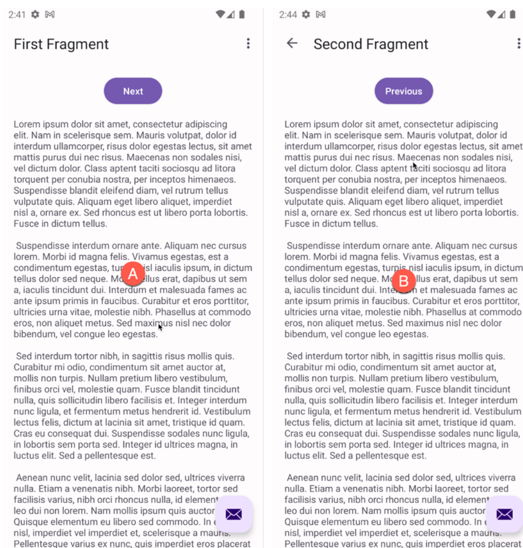


Figure 17-4

This app behavior uses of two Android features referred to as *fragments* and *navigation*, which will be covered starting with the chapters entitled *"An Introduction to Android Fragments"* and *"An Overview of the Navigation Architecture Component"* respectively.

The *content_main.xml* file contains a special fragment, known as a Navigation Host Fragment which allows different content to be switched in and out of view depending on the settings configured in the *res -> layout -> nav_graph.xml* file. In the case of the Basic Views Activity template, the *nav_graph.xml* file is configured to switch between the user interface layouts defined in the *fragment_first.xml* and *fragment_second.xml* files based on the Next and Previous button selections made by the user.

The Empty Views Activity template is helpful if you need neither a floating action button nor a menu in your activity and do not need the special app bar behavior provided by the CoordinatorLayout, such as options to make the app bar and toolbar collapse from view during certain scrolling operations (a topic covered in the chapter entitled "*Working with the AppBar and Collapsing Toolbar Layouts*"). However, the Basic Views Activity is helpful because it provides these elements by default. In fact, it is often quicker to create a new activity using the Basic Views Activity template and delete the elements you do not require than to use the Empty Views Activity template and manually implement behavior such as collapsing toolbars, a menu, or a floating action button.

Since not all of the examples in this book require the features of the Basic Views Activity template, however, most of the examples in this chapter will use the Empty Views Activity template unless the example requires one or other of the features provided by the Basic Views Activity template.

For future reference, if you need a menu but not a floating action button, use the Basic Views Activity and follow these steps to delete the floating action button:

1. Double-click on the main *activity_main.xml* layout file in the Project tool window under *app -> res -> layout* to load it into the Layout Editor. With the layout loaded into the Layout Editor tool, select the floating action button and tap the keyboard *Delete* key to remove the object from the layout.

2. Locate and edit the Java code for the activity (located under *app -> java -> <package name> -> <activity class name>* and remove the floating action button code from the onCreate method as follows:

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

    binding = ActivityMainBinding.inflate(getLayoutInflater());
    setContentView(binding.getRoot());

    setSupportActionBar(binding.toolbar);

    NavController navController = Navigation.findNavController(this, R.id.nav_
host_fragment_content_main);
    appBarConfiguration = new AppBarConfiguration.Builder(navController.
getGraph()).build();
    NavigationUI.setupActionBarWithNavController(this, navController,
appBarConfiguration);

    binding.fab.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            Snackbar.make(view, "Replace with your own action", Snackbar.LENGTH_
LONG)
                    .setAction("Action", null).show();
        }
    });
}
```

If you need a floating action button but no menu, use the Basic Views Activity template and follow these steps:

1. Edit the main activity class file and delete the *onCreateOptionsMenu* and *onOptionsItemSelected* methods.
2. Select the *res -> menu* item in the Project tool window and tap the keyboard *Delete* key to remove the folder and corresponding menu resource files from the project.

If you need to use the Basic Views Activity template but need neither the navigation features nor the second content fragment, follow these steps:

1. Within the Project tool window, navigate to and double-click on the *app -> res -> navigation -> nav_graph.xml* file to load it into the navigation editor.
2. Within the editor, select the SecondFragment entry in the graph panel and tap the keyboard delete key to remove it from the graph.
3. Locate and delete the *SecondFragment.java* (*app -> java -> <package name> -> SecondFragment*) and *fragment_second.xml* (*app -> res -> layout -> fragment_second.xml*) files.
4. The final task is to remove some code from the FirstFragment class so that the Button view no longer navigates to the now non-existent second fragment when clicked. Locate the *FirstFragment.java* file, double-click on it to load it into the editor, and remove the code from the *onViewCreated()* method so that it reads as follows:

```
public void onViewCreated(@NonNull View view, Bundle savedInstanceState) {
    super.onViewCreated(view, savedInstanceState);
```

```
    binding.buttonFirst.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            NavHostFragment.findNavController(FirstFragment.this)
                    .navigate(R.id.action_FirstFragment_to_SecondFragment);
        }
    });
}
```

17.2 The Android Studio Layout Editor

As demonstrated in previous chapters, the Layout Editor tool provides a "what you see is what you get" (WYSIWYG) environment in which views can be selected from a palette and then placed onto a canvas representing the display of an Android device. Once a view has been placed on the canvas, it can be moved, deleted, and resized (subject to the constraints of the parent view). Moreover, various properties relating to the selected view may be modified using the Attributes tool window.

Under the surface, the Layout Editor tool constructs an XML resource file containing the definition of the user interface that is being designed. As such, the Layout Editor tool operates in three distinct modes: Design, Code, and Split.

17.3 Design Mode

In design mode, the user interface can be visually manipulated by directly working with the view palette and the graphical representation of the layout. Figure 17-5 highlights the key areas of the Android Studio Layout Editor tool in design mode:

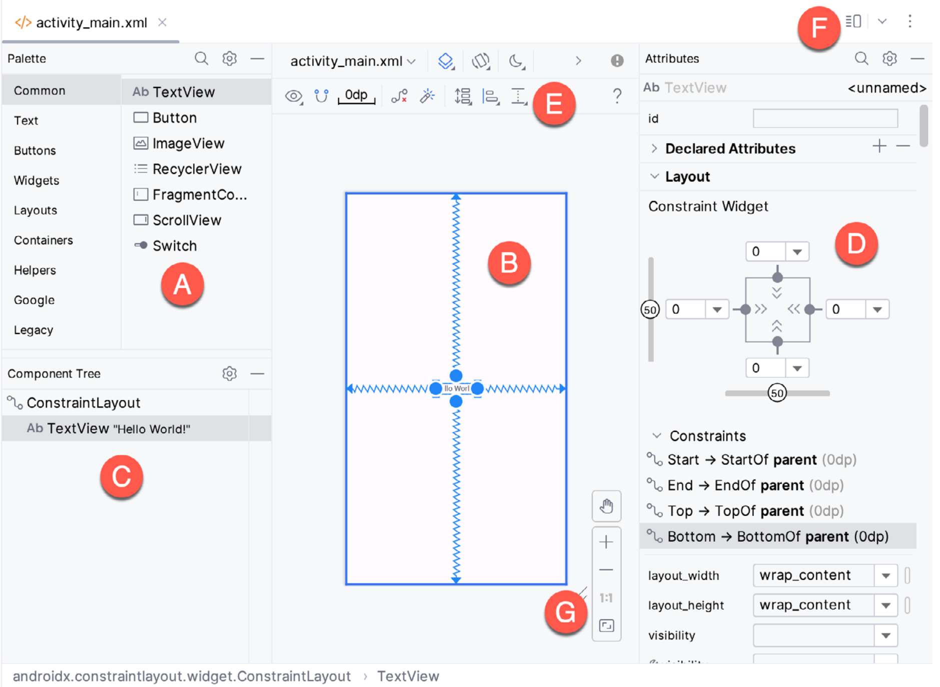


Figure 17-5

A – Palette – The palette provides access to the range of view components the Android SDK provides. These are grouped into categories for easy navigation. Items may be added to the layout by dragging a view component from the palette and dropping it at the desired position on the layout.

B – Device Screen – The device screen provides a visual "what you see is what you get" representation of the user interface layout as it is being designed. This layout allows direct design manipulation by allowing views to be selected, deleted, moved, and resized. The device model represented by the layout can be changed anytime using a menu in the toolbar.

C – Component Tree – As outlined in the previous chapter (*"Understanding Android Views, View Groups and Layouts"*), user interfaces are constructed using a hierarchical structure. The component tree provides a visual overview of the hierarchy of the user interface design. Selecting an element from the component tree will cause the corresponding view in the layout to be selected. Similarly, selecting a view from the device screen layout will select that view in the component tree hierarchy.

D – Attributes – All of the component views listed in the palette have associated with them a set of attributes that can be used to adjust the behavior and appearance of that view. The Layout Editor's attributes panel provides access to the attributes of the currently selected view in the layout allowing changes to be made.

E – Toolbar – The Layout Editor toolbar provides quick access to a wide range of options, including, amongst other options, the ability to zoom in and out of the device screen layout, change the device model currently displayed, rotate the layout between portrait and landscape and switch to a different Android SDK API level. The toolbar also has a set of context-sensitive buttons which will appear when relevant view types are selected in the device screen layout.

F – Mode Switching Controls – These three buttons provide a way to switch back and forth between the Layout Editor tool's Design, Code, and Split modes.

G - Zoom and Pan Controls - This control panel allows you to zoom in and out of the design canvas, grab the canvas, and pan around to find obscured areas when zoomed in.

17.4 The Palette

The Layout Editor palette is organized into two panels designed to make it easy to locate and preview view components for addition to a layout design. The category panel (marked A in Figure 17-6) lists the different categories of view components supported by the Android SDK. When a category is selected from the list, the second panel (B) updates to display a list of the components that fall into that category:

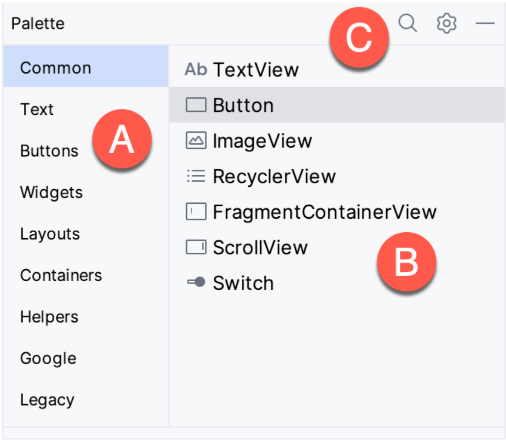


Figure 17-6

To add a component from the palette onto the layout canvas, select the item from the component list or the preview panel, drag it to the desired location on the canvas, and drop it into place.

A search for a specific component within the selected category may be initiated by clicking the search button (marked C in Figure 17-6 above) in the palette toolbar and typing in the component name. As characters are typed, matching results will appear in the component list panel. If you are unsure of the component's category, select the All Results category before or during the search operation.

17.5 Design Mode and Layout Views

By default, the layout editor will appear in Design mode, as shown in Figure 17-5 above. This mode provides a visual representation of the user interface. Design mode can be selected at any time by clicking on the View Modes button, as shown in Figure 17-7:

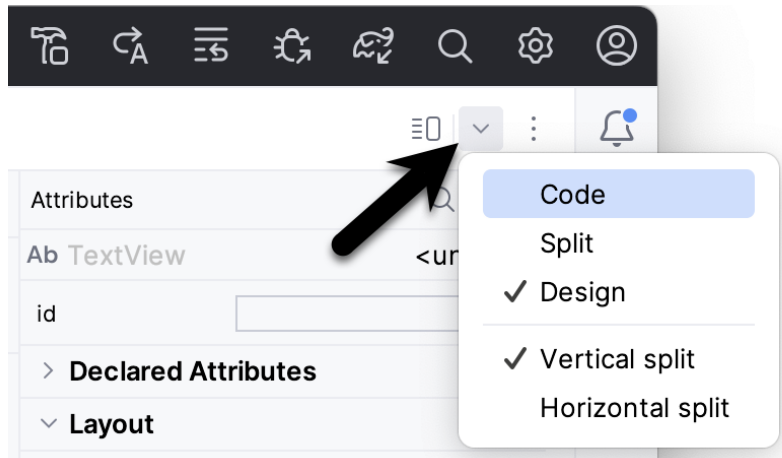


Figure 17-7

When the Layout Editor tool is in Design mode, the layout can be viewed in two ways. The view shown in Figure 17-5 above is the Design view and shows the layout and widgets as they will appear in the running app. A second mode, the Blueprint view, can be shown instead of or concurrently with the Design view. The toolbar menu in Figure 17-8 provides options to display the Design, Blueprint, or both views. Settings are also available to adjust for color blindness. A fifth option, *Force Refresh Layout*, causes the layout to rebuild and redraw. This can be useful when the layout enters an unexpected state or is not accurately reflecting the current design settings:

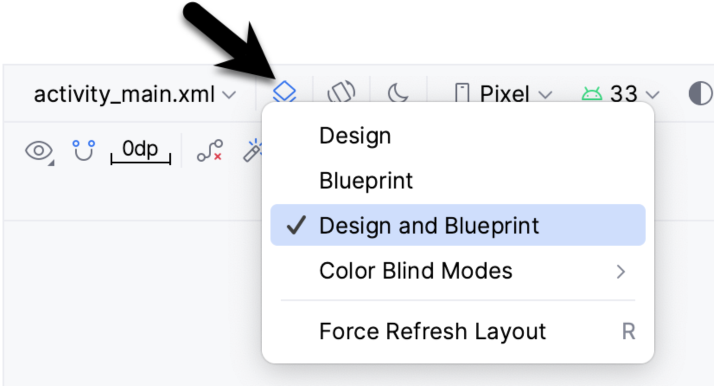


Figure 17-8

17.17 Summary

A key part of developing Android applications involves the creation of the user interface. This is performed within the Android Studio environment using the Layout Editor tool, which operates in three modes. In Design mode, view components are selected from a palette, positioned on a layout representing an Android device screen, and configured using a list of attributes. The underlying XML representing the user interface layout can be directly edited in Code mode. Split mode, on the other hand, allows the layout to be created and modified both visually and via direct XML editing. These modes combine to provide an extensive and intuitive user interface design environment.

The layout validation panel allows user interface layouts to be quickly previewed on various device screen sizes.

Chapter 25

25. A Guide to Using Apply Changes in Android Studio

Now that some of the basic concepts of Android development using Android Studio have been covered, this is a good time to introduce the Android Studio Apply Changes feature. As all experienced developers know, every second spent waiting for an app to compile and run is better spent writing and refining code.

25.1 Introducing Apply Changes

In early versions of Android Studio, each time a change to a project needed to be tested, Android Studio would recompile the code, convert it to Dex format, generate the APK package file, and install it on the device or emulator. Having performed these steps, the app would finally be launched and ready for testing. Even on a fast development system, this process takes considerable time to complete. It is not uncommon for it to take a minute or more for this process to complete for a large application.

Apply Changes, in contrast, allows many code and resource changes within a project to be reflected nearly instantaneously within the app while it is already running on a device or emulator session.

Consider, for example, an app being developed in Android Studio which has already been launched on a device or emulator. If changes are made to resource settings or the code within a method, Apply Changes will push the updated code and resources to the running app and dynamically "swap" the changes. The changes are then reflected in the running app without the need to build, deploy and relaunch the entire app. This often allows changes to be tested in a fraction of the time without Apply Changes.

25.2 Understanding Apply Changes Options

Android Studio provides three options for applying changes to a running app in the form of *Run App*, *Apply Changes and Restart Activity* and *Apply Code Changes*. These options can be summarized as follows:

- **Run App** - Stops the currently running app and restarts it. If no changes have been made to the project since it was last launched, this option will restart the app. If, on the other hand, changes have been made to the project, Android Studio will rebuild and re-install the app onto the device or emulator before launching it.

- **Apply Code Changes** - This option can be used when the only changes made to a project involve modifications to the body of existing methods or when a new class or method has been added. When selected, the changes will be applied to the running app without needing to restart the app or the currently running activity. This mode cannot, however, be used when changes have been made to any project resources, such as a layout file. Other restrictions include removing methods, changing a method signature, renaming classes, and other structural code changes. It is also impossible to use this option when changes have been made to the project manifest.

- **Apply Changes and Restart Activity** - When selected, this mode will dynamically apply any code or resource changes made within the project and restart the activity without re-installing or restarting the app. Unlike the Apply Code changes option, this can be used when changes have been made to the code and resources of the project. However, the same restrictions involving some structural code changes and manifest modifications apply.

25.3 Using Apply Changes

When a project has been loaded into Android Studio but is not yet running on a device or emulator, it can be launched as usual using either the run (marked A in Figure 25-1) or debug (B) button located in the toolbar:

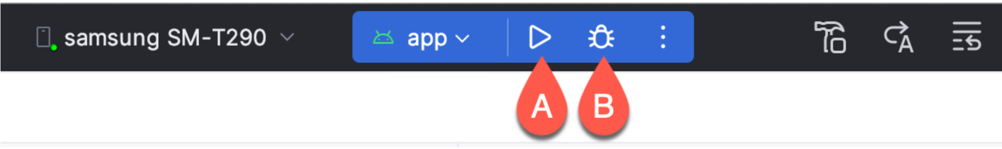


Figure 25-1

After the app has launched and is running, a stop button (marked A in Figure 25-2) will appear, and the *Apply Changes and Restart Activity* (B) and *Apply Code Changes* (C) buttons will be enabled:

Figure 25-2

If the changes cannot be applied when one of the Apply Changes buttons is selected, Android Studio will display a message indicating the failure and an explanation. Figure 25-3, for example, shows the message displayed by Android Studio when the *Apply Code Changes* option is selected after a change has been made to a resource file:

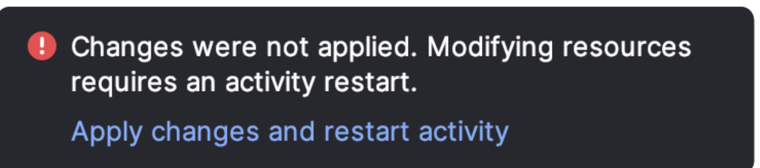


Figure 25-3

In this situation, the solution is to use the *Apply Changes and Restart Activity* option (for which a link is provided). Similarly, the following message will appear when an attempt to apply changes that involve the removal of a method is made:

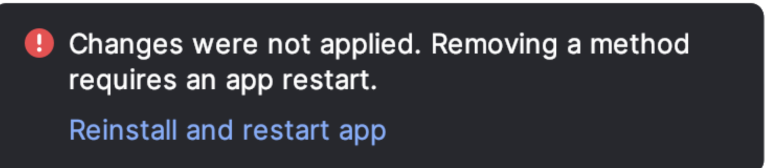


Figure 25-4

In this case, the only option is to click on the *Run App* button to re-install and restart the app. As an alternative to manually selecting the correct option, Android Studio may be configured to automatically fall back to performing a Run App operation.

25.4 Configuring Apply Changes Fallback Settings

The Apply Changes fallback settings are located in the Android Studio Settings dialog. Within the Settings dialog, select the *Build, Execution, Deployment* entry in the left-hand panel, followed by *Deployment*, as shown in Figure 25-5:

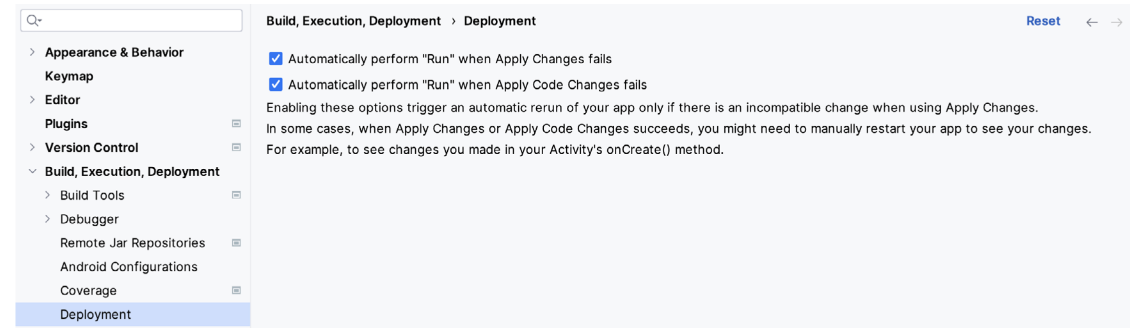


Figure 25-5

Once the required options have been enabled, click on Apply, followed by the OK button to commit the changes and dismiss the dialog. After these defaults have been enabled, Android Studio will automatically re-install and restart the app when necessary.

25.5 An Apply Changes Tutorial

Launch Android Studio, select the New Project option from the welcome screen, and choose the Basic Views Activity template within the resulting new project dialog before clicking the Next button.

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

25.6 Using Apply Code Changes

Begin by clicking the run button and selecting an emulator or physical device as the run target. After clicking the run button, track the time before the example app appears on the device or emulator.

Once running, click on the action button (the button displaying an envelope icon in the screen's lower right-hand corner). Note that a Snackbar instance appears, displaying text which reads "Replace with your own action", as shown in Figure 25-6:

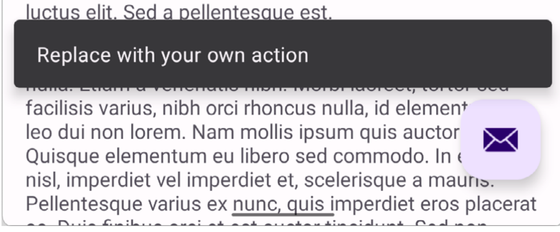


Figure 25-6

Once the app is running, the Apply Changes buttons should have been enabled, indicating that certain project changes can be applied without reinstalling and restarting the app. To see this in action, edit the *MainActivity.java* file, locate the *onCreate* method, and modify the action code so that a different message is displayed when the action button is selected:

```
binding.fab.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        Snackbar.make(view, "Apply Changes is Amazing!", Snackbar.LENGTH_LONG)
                .setAction("Action", null).show();
    }
});
```

With the code change implemented, click the *Apply Code Changes* button and note that a message appears within a few seconds indicating the app has been updated. Tap the action button and note that the new message is now displayed in the Snackbar.

25.7 Using Apply Changes and Restart Activity

Any resource change will require the use of the *Apply Changes and Restart Activity* option. Within Android Studio, select the *app -> res -> layout -> fragment_first.xml* layout file. With the Layout Editor tool in Design mode, select the default TextView component and change the text property in the attributes tool window to "Hello Android".

Ensure that the fallback options outlined in *"Configuring Apply Changes Fallback Settings"* above are turned off before clicking on the *Apply Code Changes* button. Note that the request fails because this change involves project resources. Click on the *Apply Changes and Restart Activity* button and verify that the activity restarts and displays the new text on the TextView widget.

25.8 Using Run App

As previously described, removing a method requires the complete re-installation and restart of the running app. To experience this, edit the *MainActivity.java* file and add a new method after the *onCreate* method as follows:

```
public void demoMethod() {
}
```

Use the *Apply Code Changes* button and confirm that the changes are applied without re-installing the app.

Next, delete the new method and verify that clicking on either of the two Apply Changes buttons will result in the request failing. The only way to run the app after such a change is to click the Run App button.

25.9 Summary

Apply Changes is a feature of Android Studio designed to significantly accelerate the code, build and run cycle performed when developing an app. The Apply Changes feature can push updates to the running application, in many cases, without reinstalling or restarting the app. Apply Changes provides several different levels of support depending on the nature of the modification being applied to the project.

Chapter 26

26. 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.

26.1 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.

26.2 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.

26.3 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.

26.4 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.

26.5 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 26-1.

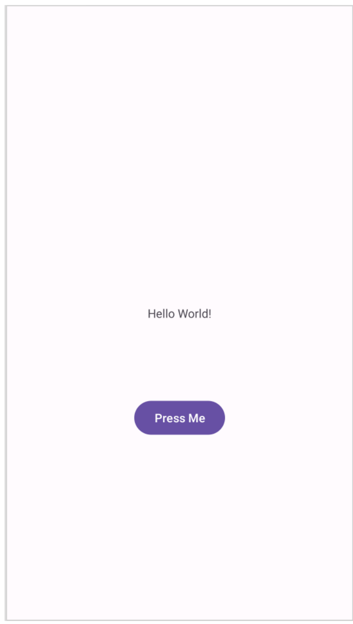


Figure 26-1

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.

26.6 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 -> 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);
        binding = ActivityMainBinding.inflate(getLayoutInflater());
        View view = binding.getRoot();
        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) {
       super.onCreate(savedInstanceState);
        binding = ActivityMainBinding.inflate(getLayoutInflater());
        View view = binding.getRoot();
        setContentView(view);
```

```
        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.

26.7 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.

26.8 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 27

27. Android Touch and Multi-touch Event Handling

Most Android-based devices use a touch screen as the primary interface between the user and the device. The previous chapter introduced how a touch on the screen translates into an action within a running Android application. There is, however, much more to touch event handling than responding to a single finger tap on a view object. Most Android devices can, for example, detect more than one touch at a time. Nor are touches limited to a single point on the device display. Touches can be dynamic as the user slides one or more contact points across the screen's surface.

An application can also interpret touches as a gesture. Consider, for example, that a horizontal swipe is typically used to turn the page of an eBook or how a pinching motion can zoom in and out of an image displayed on the screen.

An application can also interpret touches as a *gesture*. Consider, for example, that a horizontal swipe is typically used to turn the page of an eBook or how a pinching motion can zoom in and out of an image displayed on the screen.

This chapter will explain the handling of touches that involve motion and explore the concept of intercepting multiple concurrent touches. The topic of identifying distinct gestures will be covered in the next chapter.

27.1 Intercepting Touch Events

A view object can intercept touch events by registering an onTouchListener event listener and implementing the corresponding onTouch() callback method. The following code, for example, ensures that any touches on a ConstraintLayout view instance named *myLayout* result in a call to the *onTouch()* method:

```
binding.myLayout.setOnTouchListener(
        new ConstraintLayout.OnTouchListener() {
                public boolean onTouch(View v, MotionEvent m) {
                        // Perform tasks here
                        return true;
                }
        }
);
```

As indicated in the code example, the *onTouch()* callback is required to return a Boolean value indicating to the Android runtime system whether or not the event should be passed on to other event listeners registered on the same view or discarded. The method is passed both a reference to the view on which the event was triggered and an object of type *MotionEvent*.

27.2 The MotionEvent Object

The MotionEvent object passed through to the *onTouch()* callback method is the key to obtaining information about the event. Information within the object includes the location of the touch within the view and the type of action performed. The MotionEvent object is also the key to handling multiple touches.

27.3 Understanding Touch Actions

An important aspect of touch event handling involves identifying the type of action the user performed. The type of action associated with an event can be obtained by making a call to the *getActionMasked()* method of the MotionEvent object, which was passed through to the *onTouch()* callback method. When the first touch on a view occurs, the MotionEvent object will contain an action type of ACTION_DOWN together with the coordinates of the touch. When that touch is lifted from the screen, an ACTION_UP event is generated. Any motion of the touch between the ACTION_DOWN and ACTION_UP events will be represented by ACTION_MOVE events.

When more than one touch is performed simultaneously on a view, the touches are referred to as *pointers*. In a multi-touch scenario, pointers begin and end with event actions of type ACTION_POINTER_DOWN and ACTION_POINTER_UP, respectively. To identify the index of the pointer that triggered the event, the *getActionIndex()* callback method of the MotionEvent object must be called.

27.4 Handling Multiple Touches

The chapter entitled *"An Overview and Example of Android Event Handling"* began exploring event handling within the narrow context of a single-touch event. In practice, most Android devices can respond to multiple consecutive touches (though it is important to note that the number of simultaneous touches that can be detected varies depending on the device).

As previously discussed, each touch in a multi-touch situation is considered by the Android framework to be a *pointer*. Each pointer, in turn, is referenced by an *index* value and assigned an *ID*. The current number of pointers can be obtained via a call to the *getPointerCount()* method of the current MotionEvent object. The ID for a pointer at a particular index in the list of current pointers may be obtained via a call to the MotionEvent *getPointerId()* method. For example, the following code excerpt obtains a count of pointers and the ID of the pointer at index 0:

```
public boolean onTouch(View v, MotionEvent m) {
        int pointerCount = m.getPointerCount();
        int pointerId = m.getPointerId(0);
        return true;
}
```

Note that the pointer count will always be greater than or equal to 1 when the *onTouch* listener is triggered (since at least one touch must have occurred for the callback to be triggered).

A touch on a view, particularly one involving motion across the screen, will generate a stream of events before the point of contact with the screen is lifted. An application will likely need to track individual touches over multiple touch events. While the ID of a specific touch gesture will not change from one event to the next, it is important to remember that the index value will change as other touch events come and go. When working with a touch gesture over multiple events, the ID value must be used as the touch reference to ensure the same touch is being tracked. When calling methods that require an index value, this should be obtained by converting the ID for a touch to the corresponding index value via a call to the *findPointerIndex()* method of the *MotionEvent* object.

27.5 An Example Multi-Touch Application

The example application created in the remainder of this chapter will track up to two touch gestures as they move across a layout view. As the events for each touch are triggered, the coordinates, index, and ID for each touch will be displayed on the screen.

Select the *New Project* option from the welcome screen and, within the resulting new project dialog, choose the

Empty Views Activity template before clicking on the Next button.

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

Adapt the project to use view binding as outlined in section *11.8 Migrating a Project to View Binding*.

27.6 Designing the Activity User Interface

The user interface for the application's sole activity is to consist of a ConstraintLayout view containing two TextView objects. Within the Project tool window, navigate to *app -> res -> layout* and double-click on the *activity_main.xml* layout resource file to load it into the Android Studio Layout Editor tool.

Select and delete the default "Hello World!" TextView widget and then, with autoconnect enabled, drag and drop a new TextView widget so that it is centered horizontally and positioned at the 16dp margin line on the top edge of the layout:

Figure 27-1

Drag a second TextView widget and position and constrain it so that a 32dp margin distances it from the bottom of the first widget:

Figure 27-2

Using the Attributes tool window, change the IDs for the TextView widgets to *textView1* and *textView2*, respectively. Change the text displayed on the widgets to read "Touch One Status" and "Touch Two Status" and extract the strings to resources using the warning button in the top right-hand corner of the Layout Editor.

27.7 Implementing the Touch Event Listener

To receive touch event notification, it will be necessary to register a touch listener on the layout view within the *onCreate()* method of the *MainActivity* activity class. Select the *MainActivity.java* tab from the Android Studio editor panel to display the source code. Within the *onCreate()* method, add code to register the touch listener and implement code which, in this case, is going to call a second method named *handleTouch()* to which is passed the MotionEvent object:

```
package com.ebookfrenzy.motionevent;

import androidx.appcompat.app.AppCompatActivity;
import androidx.constraintlayout.widget.ConstraintLayout;

import android.os.Bundle;
import android.view.MotionEvent;
import android.view.View;
```

```
import com.ebookfrenzy.motionevent.databinding.ActivityMainBinding;

public class MainActivity extends AppCompatActivity {

    private ActivityMainBinding binding;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        binding = ActivityMainBinding.inflate(getLayoutInflater());
        View view = binding.getRoot();
        setContentView(view);

        binding.activityMain.setOnTouchListener(
                new ConstraintLayout.OnTouchListener() {
                    public boolean onTouch(View v, MotionEvent m) {
                        handleTouch(m);
                        return true;
                    }
                }
        );
    }
```

When we designed the user interface, the parent ConstraintLayout was not assigned an ID that would allow us to access it via the view binding mechanism. Since this layout component is the topmost component in the UI layout hierarchy, we have been able to reference it using the *root* binding property in the code above.

Before testing the application, the final task is to implement the *handleTouch()* method called by the listener. The code for this method reads as follows:

```
void handleTouch(MotionEvent m) {

    int pointerCount = m.getPointerCount();

    for (int i = 0; i < pointerCount; i++)
    {
        int x = (int) m.getX(i);
        int y = (int) m.getY(i);
        int id = m.getPointerId(i);
        int action = m.getActionMasked();
        int actionIndex = m.getActionIndex();
        String actionString;

        switch (action)
        {
            case MotionEvent.ACTION_DOWN:
                actionString = "DOWN";
                break;
```

```
            case MotionEvent.ACTION_UP:
                actionString = "UP";
                break;
            case MotionEvent.ACTION_POINTER_DOWN:
                actionString = "PNTR DOWN";
                break;
            case MotionEvent.ACTION_POINTER_UP:
                actionString = "PNTR UP";
                break;
            case MotionEvent.ACTION_MOVE:
                actionString = "MOVE";
                break;
            default:
                actionString = "";
        }

        String touchStatus = "Action: " + actionString + " Index: " + actionIndex
+ " ID: " + id + " X: " + x + " Y: " + y;

        if (id == 0)
            binding.textView1.setText(touchStatus);
        else
            binding.textView2.setText(touchStatus);
    }
}
```

Before compiling and running the application, it is worth taking the time to walk through this code systematically to highlight the tasks performed.

The code begins by obtaining references to the two TextView objects in the user interface and identifying how many pointers are currently active on the view:

```
TextView textView1 = findViewById(R.id.textView1);
TextView textView2 = findViewById(R.id.textView2);

int pointerCount = m.getPointerCount();
```

Next, the *pointerCount* variable initiates a for loop, which performs tasks for each active pointer. The first few lines of the loop obtain the X and Y coordinates of the touch together with the corresponding event ID, action type, and action index. Lastly, a string variable is declared:

```
for (int i = 0; i < pointerCount; i++)
{
    int x = (int) m.getX(i);
    int y = (int) m.getY(i);
    int id = m.getPointerId(i);
    int action = m.getActionMasked();
    int actionIndex = m.getActionIndex();
    String actionString;
```

Since action types equate to integer values, a *switch* statement is used to convert the action type to a more meaningful string value, which is stored in the previously declared *actionString* variable:

```
switch (action)
{
    case MotionEvent.ACTION_DOWN:
        actionString = "DOWN";
        break;
    case MotionEvent.ACTION_UP:
        actionString = "UP";
        break;
    case MotionEvent.ACTION_POINTER_DOWN:
        actionString = "PNTR DOWN";
        break;
    case MotionEvent.ACTION_POINTER_UP:
        actionString = "PNTR UP";
        break;
    case MotionEvent.ACTION_MOVE:
        actionString = "MOVE";
        break;
    default:
        actionString = "";
}
```

Finally, the string message is constructed using the *actionString* value, the action index, touch ID, and X and Y coordinates. The ID value is then used to decide whether the string should be displayed on the first or second TextView object:

```
String touchStatus = "Action: " + actionString + " Index: "
         + actionIndex + " ID: " + id + " X: " + x + " Y: " + y;

if (id == 0)
     binding.textView1.setText(touchStatus);
else
     binding.textView2.setText(touchStatus);
```

27.8 Running the Example Application

Compile and run the application and, once launched, experiment with single and multiple touches on the screen and note that the text views update to reflect the events as illustrated in Figure 27-3. When running on an emulator, multiple touches may be simulated by holding down the Ctrl (Cmd on macOS) key while clicking the mouse button (note that simulating multiple touches may not work if the emulator is running in a tool window):

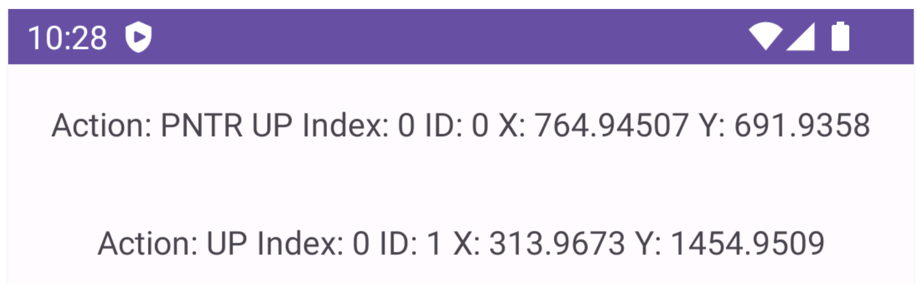


Figure 27-3

27.9 Summary

Activities receive notifications of touch events by registering an onTouchListener event listener and implementing the onTouch() callback method, which, in turn, is passed a MotionEvent object when called by the Android runtime. This object contains information about the touch, such as the type of touch event, the coordinates of the touch, and a count of the number of touches currently in contact with the view.

When multiple touches are involved, each point of contact is referred to as a pointer, with each assigned an index and an ID. While the index of a touch can change from one event to another, the ID will remain unchanged until the touch ends.

This chapter has worked through creating an example Android application designed to display the coordinates and action type of up to two simultaneous touches on a device display.

Having covered touches in general, the next chapter (entitled *"Detecting Common Gestures Using the Android Gesture Detector Class"*) will look further at touchscreen event handling through gesture recognition.

28. Detecting Common Gestures Using the Android Gesture Detector Class

The term "gesture" defines a contiguous sequence of interactions between the touch screen and the user. A typical gesture begins at the point that the screen is first touched and ends when the last finger or pointing device leaves the display surface. When correctly harnessed, gestures can be implemented to communicate between the user and the application. Swiping motions to turn the pages of an eBook or a pinching movement involving two touches to zoom in or out of an image are prime examples of how gestures can interact with an application.

The Android SDK provides mechanisms for the detection of both common and custom gestures within an application. Common gestures involve interactions such as a tap, double tap, long press, or a swiping motion in either a horizontal or a vertical direction (referred to in Android nomenclature as a *fling*).

This chapter explores using the Android GestureDetector class to detect common gestures performed on the display of an Android device. The next chapter, *"Implementing Custom Gesture and Pinch Recognition on Android"*, will cover detecting more complex, custom gestures such as circular motions and pinches.

28.1 Implementing Common Gesture Detection

When a user interacts with the display of an Android device, the *onTouchEvent()* method of the currently active application is called by the system and passed MotionEvent objects containing data about the user's contact with the screen. This data can be interpreted to identify if the motion on the screen matches a common gesture such as a tap or a swipe. This can be achieved with minimal programming effort by using the Android GestureDetectorCompat class. This class is designed to receive motion event information from the application and trigger method calls based on the type of common gesture, if any, detected.

The basic steps in detecting common gestures are as follows:

1. Declaration of a class which implements the GestureDetector.OnGestureListener interface including the required *onFling()*, *onDown()*, *onScroll()*, *onShowPress()*, *onSingleTapUp()* and *onLongPress()* callback methods. Note that this can be either an entirely new or an enclosing activity class. If double-tap gesture detection is required, the class must also implement the GestureDetector.OnDoubleTapListener interface and include the corresponding *onDoubleTap()* method.

2. Creation of an instance of the Android GestureDetectorCompat class, passing through an instance of the class created in step 1 as an argument.

3. An optional call to the *setOnDoubleTapListener()* method of the GestureDetectorCompat instance to enable double tap detection if required.

4. Implementation of the *onTouchEvent()* callback method on the enclosing activity, which, in turn, must call the *onTouchEvent()* method of the GestureDetectorCompat instance, passing through the current motion event object as an argument to the method.

Once implemented, the result is a set of methods within the application code that will be called when a gesture of a particular type is detected. The code within these methods can then be implemented to perform any tasks that need to be performed in response to the corresponding gesture.

In the remainder of this chapter, we will work through creating an example project intended to put the above steps into practice.

28.2 Creating an Example Gesture Detection Project

This project aims to detect the full range of common gestures currently supported by the GestureDetectorCompat class and to display status information to the user indicating the type of gesture that has been detected.

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

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

Adapt the project to use view binding as outlined in section *11.8 Migrating a Project to View Binding*.

Once the new project has been created, navigate to the *app -> res -> layout -> activity_main.xml* file in the Project tool window and double-click on it to load it into the Layout Editor tool.

Within the Layout Editor tool, select the "Hello, World!" TextView component and, in the Attributes tool window, enter *gestureStatusText* as the ID. Finally, set the textSize to 20sp and enable the bold textStyle:

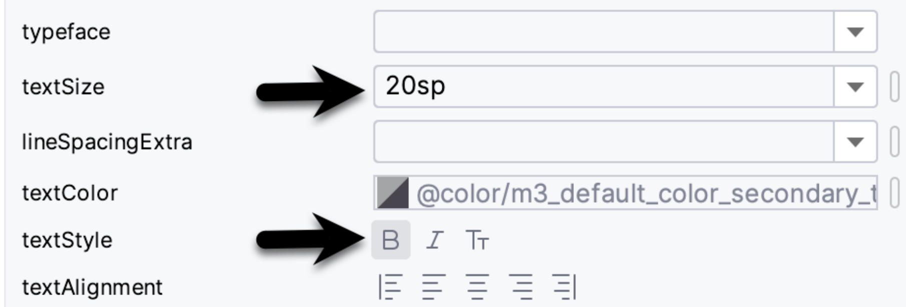


Figure 28-1

28.3 Implementing the Listener Class

As previously outlined, it is necessary to create a class that implements the GestureDetector.OnGestureListener interface and, if double tap detection is required, the GestureDetector.OnDoubleTapListener interface. While this can be an entirely new class, it is also perfectly valid to implement this within the current activity class. Therefore, we will modify the MainActivity class to implement these listener interfaces for this example. Edit the *MainActivity.java* file so that it reads as follows:

```
package com.ebookfrenzy.commongestures;

import android.view.GestureDetector;
.
.
public class MainActivity extends AppCompatActivity
        implements GestureDetector.OnGestureListener,
```

```
        GestureDetector.OnDoubleTapListener
{
.
.
}
```

Declaring that the class implements the listener interfaces mandates that the corresponding methods also be implemented in the class:

```
package com.ebookfrenzy.commongestures;
.
.
import android.view.MotionEvent;

public class MainActivity extends AppCompatActivity
        implements GestureDetector.OnGestureListener,
        GestureDetector.OnDoubleTapListener {
.
.
    @Override
    public boolean onDown(MotionEvent event) {
        binding.gestureStatusText.setText ("onDown");
        return true;
    }

    @Override
    public boolean onFling(MotionEvent event1, MotionEvent event2,
                           float velocityX, float velocityY) {
        binding.gestureStatusText.setText("onFling");
        return true;
    }

    @Override
    public void onLongPress(MotionEvent event) {
        binding.gestureStatusText.setText("onLongPress");
    }

    @Override
    public boolean onScroll(MotionEvent e1, MotionEvent e2,
                            float distanceX, float distanceY) {
        binding.gestureStatusText.setText("onScroll");
        return true;
    }

    @Override
    public void onShowPress(MotionEvent event) {
        binding.gestureStatusText.setText("onShowPress");
```

```
    }

    @Override
    public boolean onSingleTapUp(MotionEvent event) {
        binding.gestureStatusText.setText("onSingleTapUp");
        return true;
    }

    @Override
    public boolean onDoubleTap(MotionEvent event) {
        binding.gestureStatusText.setText("onDoubleTap");
        return true;
    }

    @Override
    public boolean onDoubleTapEvent(MotionEvent event) {
        binding.gestureStatusText.setText("onDoubleTapEvent");
        return true;
    }

    @Override
    public boolean onSingleTapConfirmed(MotionEvent event) {
        binding.gestureStatusText.setText("onSingleTapConfirmed");
        return true;
    }
.
.
.
}
```

Note that many of these methods return *true*. This indicates to the Android Framework that the method has consumed the event and does not need to be passed to the next event handler in the stack.

28.4 Creating the GestureDetectorCompat Instance

With the activity class now updated to implement the listener interfaces, the next step is to create an instance of the GestureDetectorCompat class. Since this only needs to be performed once at the point that the activity is created, the best place for this code is in the *onCreate()* method. Since we also want to detect double taps, the code also needs to call the *setOnDoubleTapListener()* method of the GestureDetectorCompat instance:

```
package com.ebookfrenzy.commongestures;
.
.
import androidx.core.view.GestureDetectorCompat;

public class MainActivity extends AppCompatActivity
        implements GestureDetector.OnGestureListener,
        GestureDetector.OnDoubleTapListener {
```

```
    private ActivityMainBinding binding;
    private GestureDetectorCompat gDetector;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        binding = ActivityMainBinding.inflate(getLayoutInflater());
        View view = binding.getRoot();
        setContentView(view);

        this.gDetector = new GestureDetectorCompat(this,this);
        gDetector.setOnDoubleTapListener(this);
    }
.
.
}
```

28.5 Implementing the onTouchEvent() Method

If the application were to be compiled and run at this point, nothing would happen if gestures were performed on the device display. This is because no code has been added to intercept touch events and to pass them through to the GestureDetectorCompat instance. To achieve this, it is necessary to override the *onTouchEvent()* method within the activity class and implement it such that it calls the *onTouchEvent()* method of the GestureDetectorCompat instance. Remaining in the *MainActivity.java* file, therefore, implement this method so that it reads as follows:

```
@Override
public boolean onTouchEvent(MotionEvent event) {
        this.gDetector.onTouchEvent(event);
        // Be sure to call the superclass implementation
        return super.onTouchEvent(event);
}
```

28.6 Testing the Application

Compile and run the application on either a physical Android device or an AVD emulator. Once launched, experiment with swipes, presses, scrolling motions, and double and single taps. Note that the text view updates to reflect the events as illustrated in Figure 28-2:

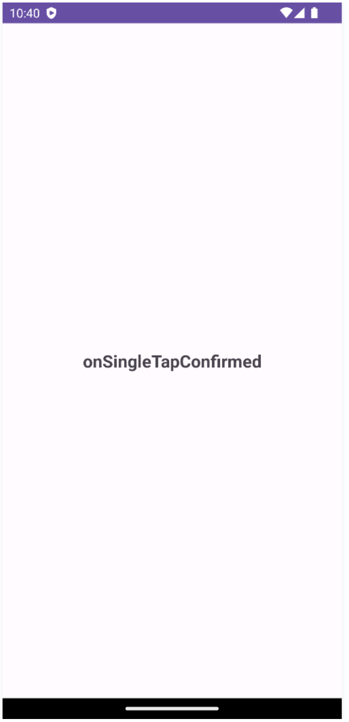


Figure 28-2

28.7 Summary

Any physical contact between the user and the touchscreen display of a device can be considered a "gesture". Lacking the physical keyboard and mouse pointer of a traditional computer system, gestures are widely used as a method of interaction between the user and the application. While a gesture can comprise just about any sequence of motions, there is a widely used set of gestures with which users of touchscreen devices have become familiar. Some of these so-called "common gestures" can be easily detected within an application by using the Android Gesture Detector classes. In this chapter, the use of this technique has been outlined both in theory and through the implementation of an example project.

Having covered common gestures in this chapter, the next chapter will look at detecting a wider range of gesture types, including the ability to design and detect your own gestures.

Chapter 32

32. Modern Android App Architecture with Jetpack

For many years, Google did not recommend a specific approach to building Android apps other than to provide tools and development kits while letting developers decide what worked best for a particular project or individual programming style. That changed in 2017 with the introduction of the Android Architecture Components, which, in turn, became part of Android Jetpack when it was released in 2018.

This chapter provides an overview of the concepts of Jetpack, Android app architecture recommendations, and some key architecture components. Once the basics have been covered, these topics will be covered in more detail and demonstrated through practical examples in later chapters.

32.1 What is Android Jetpack?

Android Jetpack consists of Android Studio, the Android Architecture Components, the Android Support Library, and a set of guidelines recommending how an Android App should be structured. The Android Architecture Components are designed to make it quicker and easier to perform common tasks when developing Android apps while also conforming to the key principle of the architectural guidelines.

While all Android Architecture Components will be covered in this book, this chapter will focus on the key architectural guidelines and the ViewModel, LiveData, and Lifecycle components while introducing Data Binding and Repositories.

Before moving on, it is important to understand that the Jetpack approach to app development is optional. While highlighting some of the shortcomings of other techniques that have gained popularity over the years, Google stopped short of completely condemning those approaches to app development. Google is taking the position that while there is no right or wrong way to develop an app, there is a recommended way.

32.2 The "Old" Architecture

In the chapter entitled *"Creating an Example Android App in Android Studio*", an Android project was created consisting of a single activity that contained all of the code for presenting and managing the user interface together with the back-end logic of the app. Until the introduction of Jetpack, the most common architecture followed this paradigm with apps consisting of multiple activities (one for each screen within the app), with each activity class to some degree mixing user interface and back-end code.

This approach led to a range of problems related to the lifecycle of an app (for example, an activity is destroyed and recreated each time the user rotates the device leading to the loss of any app data that had not been saved to some form of persistent storage) as well as issues such as inefficient navigation involving launching a new activity for each app screen accessed by the user.

32.3 Modern Android Architecture

At the most basic level, Google now advocates single-activity apps where different screens are loaded as content within the same activity.

Modern architecture guidelines also recommend separating different areas of responsibility within an app into entirely separate modules (a concept referred to as "separation of concerns"). One of the keys to this approach

is the ViewModel component.

32.4 The ViewModel Component

The purpose of ViewModel is to separate the user interface-related data model and logic of an app from the code responsible for displaying and managing the user interface and interacting with the operating system. When designed this way, an app will consist of one or more UI Controllers, such as an activity, together with ViewModel instances responsible for handling the data those controllers need.

The ViewModel only knows about the data model and corresponding logic. It knows nothing about the user interface and does not attempt to directly access or respond to events relating to views within the user interface. When a UI controller needs data to display, it asks the ViewModel to provide it. Similarly, when the user enters data into a view within the user interface, the UI controller passes it to the ViewModel for handling.

This separation of responsibility addresses the issues relating to the lifecycle of UI controllers. Regardless of how often the UI controller is recreated during the lifecycle of an app, the ViewModel instances remain in memory, thereby maintaining data consistency. For example, a ViewModel used by an activity will remain in memory until the activity finishes, which, in the single activity app, is not until the app exits.

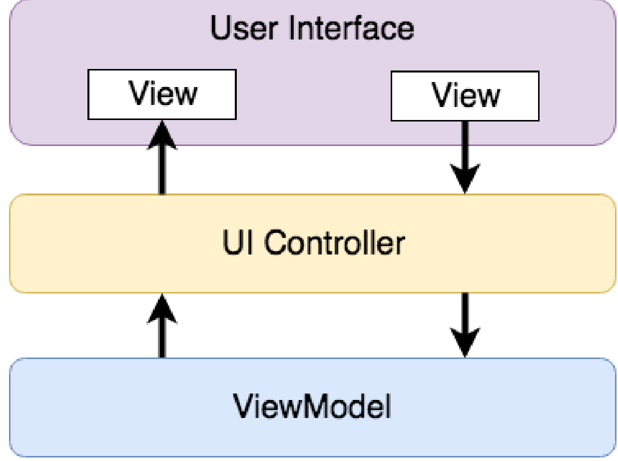


Figure 32-1

32.5 The LiveData Component

Consider an app that displays real-time data, such as the current price of a financial stock. The app could use a stock price web service to continuously update the data model within the ViewModel with the latest information. This real-time data is of use only if it is displayed to the user promptly. There are only two ways that the UI controller can ensure that the latest data is displayed in the user interface. One option is for the controller to continuously check with the ViewModel to determine if the data has changed since it was last displayed. However, the problem with this approach is that it could be more efficient. To maintain the real-time nature of the data feed, the UI controller would have to run on a loop, continuously checking for the data to change.

A better solution would be for the UI controller to receive a notification when a specific data item within a ViewModel changes. This is made possible by using the LiveData component. LiveData is a data holder that allows a value to become *observable*. In basic terms, an observable object can notify other objects when changes to its data occur, thereby solving the problem of ensuring that the user interface always matches the data within the ViewModel.

This means, for example, that a UI controller interested in a ViewModel value can set up an observer, which will, in turn, be notified when that value changes. In our hypothetical application, for example, the stock price would

be wrapped in a LiveData object within the ViewModel, and the UI controller would assign an observer to the value, declaring a method to be called when the value changes. When triggered by data change, this method will read the updated value from the ViewModel and use it to update the user interface.

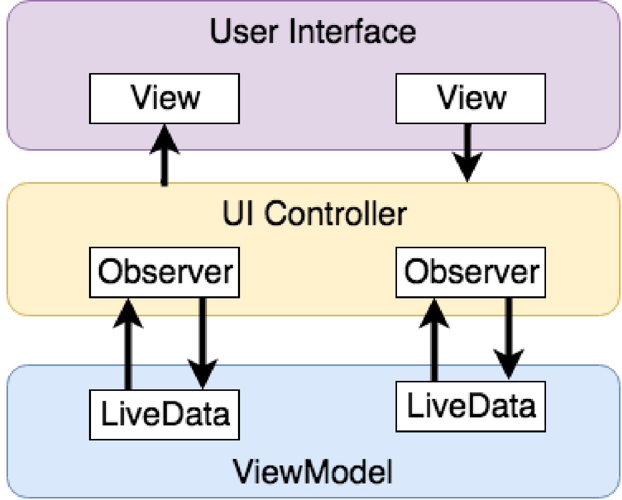


Figure 32-2

A LiveData instance may also be declared as mutable, allowing the observing entity to update the underlying value held within the LiveData object. The user might, for example, enter a value in the user interface that needs to overwrite the value stored in the ViewModel.

Another of the key advantages of using LiveData is that it is aware of the *lifecycle state* of its observers. If, for example, an activity contains a LiveData observer, the corresponding LiveData object will know when the activity's lifecycle state changes and respond accordingly. If the activity is paused (perhaps the app is put into the background), the LiveData object will stop sending events to the observer. Suppose the activity has just started or resumes after being paused. In that case, the LiveData object will send a LiveData event to the observer so that the activity has the most up-to-date value. Similarly, the LiveData instance will know when the activity is destroyed and remove the observer to free up resources.

So far, we've only talked about UI controllers using observers. In practice, however, an observer can be used within any object that conforms to the Jetpack approach to lifecycle management.

32.6 ViewModel Saved State

Android allows the user to place an active app in the background and return to it after performing other tasks on the device (including running other apps). When a device runs low on resources, the operating system will rectify this by terminating background app processes, starting with the least recently used app. However, when the user returns to the terminated background app, it should appear in the same state as when it was placed in the background, regardless of whether it was terminated. In terms of the data associated with a ViewModel, this can be implemented using the ViewModel Saved State module. This module allows values to be stored in the app's *saved state* and restored in case of system-initiated process termination. This topic will be covered later in the *"An Android ViewModel Saved State Tutorial"* chapter.

32.7 LiveData and Data Binding

Android Jetpack includes the Data Binding Library, which allows data in a ViewModel to be mapped directly to specific views within the XML user interface layout file. In the AndroidSample project created earlier, code had to be written to obtain references to the EditText and TextView views and to set and get the text properties to

reflect data changes. Data binding allows the LiveData value stored in the ViewModel to be referenced directly within the XML layout file avoiding the need to write code to keep the layout views updated.

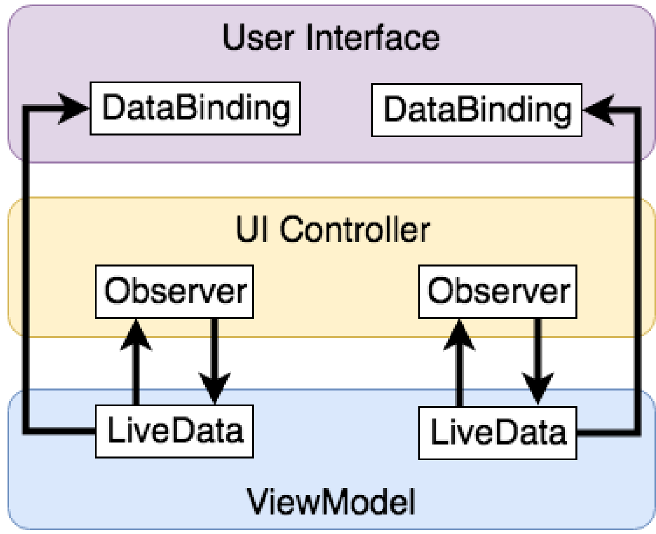


Figure 32-3

Data binding will be covered in greater detail, starting with the chapter *"An Overview of Android Jetpack Data Binding"*.

32.8 Android Lifecycles

The duration from when an Android component is created to the point that it is destroyed is called the *lifecycle*. During this lifecycle, the component will change between different lifecycle states, usually under the operating system's control and in response to user actions. An activity, for example, will begin in the *initialized* state before transitioning to the *created* state. Once the activity runs, it will switch to the *started* state, from which it will cycle through various states, including *created*, *started*, *resumed*, and *destroyed*.

Many Android Framework classes and components allow other objects to access their current state. *Lifecycle observers* may also be used so that an object receives a notification when the lifecycle state of another object changes. The ViewModel component uses this technique behind the scenes to identify when an observer has restarted or been destroyed. This functionality is not limited to Android framework and architecture components. It may also be built into any other classes using a set of lifecycle components included with the architecture components.

Objects that can detect and react to lifecycle state changes in other objects are said to be *lifecycle-aware*. In contrast, objects that provide access to their lifecycle state are called *lifecycle owners*. The chapter entitled *"Working with Android Lifecycle-Aware Components"* will cover Lifecycles in greater detail.

32.9 Repository Modules

If a ViewModel obtains data from one or more external sources (such as databases or web services, it is important to separate the code involved in handling those data sources from the ViewModel class. Failure to do this would, after all, violate the separation of concerns guidelines. To avoid mixing this functionality with the ViewModel, Google's architecture guidelines recommend placing this code in a separate *Repository* module.

A repository is not an Android architecture component but a Java class created by the app developer that is responsible for interfacing with the various data sources. The class then provides an interface to the ViewModel, allowing that data to be stored in the model.

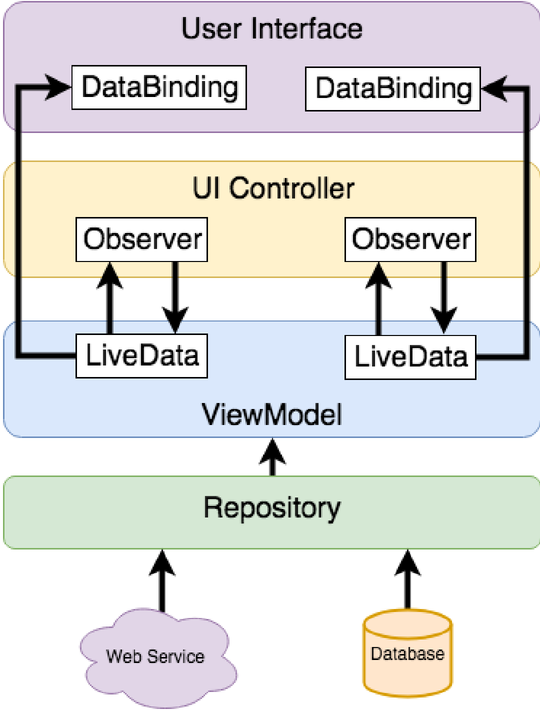


Figure 32-4

32.10 Summary

Until recently, Google has tended not to recommend any particular approach to structuring an Android app. That has now changed with the introduction of Android Jetpack, consisting of tools, components, libraries, and architecture guidelines. Google now recommends that an app project be divided into separate modules, each responsible for a particular area of functionality, otherwise known as "separation of concerns".

In particular, the guidelines recommend separating the view data model of an app from the code responsible for handling the user interface. In addition, the code responsible for gathering data from data sources such as web services or databases should be built into a separate repository module instead of being bundled with the view model.

Android Jetpack includes the Android Architecture Components, designed to make developing apps that conform to the recommended guidelines easier. This chapter has introduced the ViewModel, LiveData, and Lifecycle components. These will be covered in more detail, starting with the next chapter. Other architecture components not mentioned in this chapter will be covered later in the book.

Chapter 33

33. An Android ViewModel Tutorial

The previous chapter introduced the fundamental concepts of Android Jetpack and outlined the basics of modern Android app architecture. Jetpack defines a set of recommendations describing how an Android app project should be structured while providing a set of libraries and components that make it easier to conform to these guidelines to develop reliable apps with less coding and fewer errors.

To help reinforce and clarify the information provided in the previous chapter, this chapter will step through creating an example app project that uses the ViewModel component. The next chapter will further enhance this example by including LiveData and data binding support.

33.1 About the Project

In the chapter entitled *"Creating an Example Android App in Android Studio"*, a project named AndroidSample was created in which all of the code for the app was bundled into the main Activity class file. In the following chapter, an AVD emulator was created and used to run the app. While the app was running, we experienced first-hand the problems that occur when developing apps in this way when the data displayed on a TextView widget was lost during a device rotation.

This chapter will implement the same currency converter app, using the ViewModel component and following the Google app architecture guidelines to avoid Activity lifecycle complications.

33.2 Creating the ViewModel Example Project

When the AndroidSample project was created, the Empty Views Activity template was chosen as the basis for the project. However, the Basic Views Template template will be used for this project.

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

Enter *ViewModelDemo* into the Name field and specify *com.ebookfrenzy.viewmodeldemo* 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.

33.3 Removing Unwanted Project Elements

As outlined in the *"A Guide to the Android Studio Layout Editor Tool"*, the Basic Views Activity template includes features not required by all projects. Before adding the ViewModel to the project, we first need to remove the navigation features, the second content fragment, and the floating action button as follows:

1. Double-click on the *activity_main.xml* layout file in the Project tool window, select the floating action button, and tap the keyboard delete key to remove the object from the layout.

2. Edit the *MainActivity.java* file and remove the floating action button code from the onCreate method as follows:

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

```
    binding.fab.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            Snackbar.make(view, "Replace with your own action", Snackbar.LENGTH_LONG)
                    .setAnchorView(R.id.fab)
                    .setAction("Action", null).show();
        }
    });
}
```

3. Within the Project tool window, navigate to and double-click on the *app -> res -> navigation -> nav_graph.xml* file to load it into the navigation editor.

4. Within the editor, select the SecondFragment entry in the graph panel and tap the keyboard delete key to remove it from the graph.

5. Locate and delete the *SecondFragment.java* and *fragment_second.xml* files.

6. The final task is to remove some code from the FirstFragment class so that the Button view no longer navigates to the now non-existent second fragment when clicked. Edit the *FirstFragment.java* file and remove the code from the *onViewCreated()* method so that it reads as follows:

```
public void onViewCreated(@NonNull View view, Bundle savedInstanceState) {
    super.onViewCreated(view, savedInstanceState);

    binding.buttonFirst.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            NavHostFragment.findNavController(FirstFragment.this)
                    .navigate(R.id.action_FirstFragment_to_SecondFragment);
        }
    });
}
```

33.4 Designing the Fragment Layout

The next step is to design the layout of the fragment. First, locate the *fragment_first.xml* file in the Project tool window and double-click on it to load it into the layout editor. Once the layout has loaded, select and delete the existing Button, TextView, and ConstraintLayout components. Next, right-click on the NestedScrollView instance in the Component Tree panel and select the *Convert NestedScrollView to ConstraintLayout* menu option as shown in Figure 33-1, and accept the default settings in the resulting dialog:

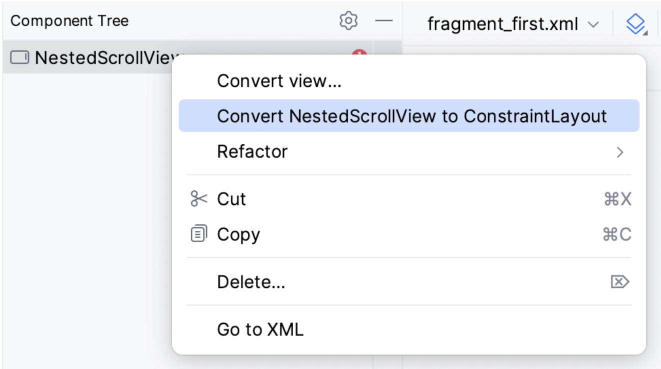


Figure 33-1

Select the converted ConstraintLayout component and use the Attributes tool window to change the id to *constraintLayout*.

Add a new TextView, position it in the center of the layout, and change the id to *resultText*. Next, drag a Number (Decimal) view from the palette and position it above the existing TextView. With the view selected in the layout, refer to the Attributes tool window and change the id to *dollarText*.

Drag a Button widget onto the layout to position it below the TextView, and change the text attribute to read "Convert". With the button still selected, change the id property to *convertButton*. At this point, the layout should resemble that illustrated in Figure 33-2 (note that the three views have been constrained using a vertical chain):

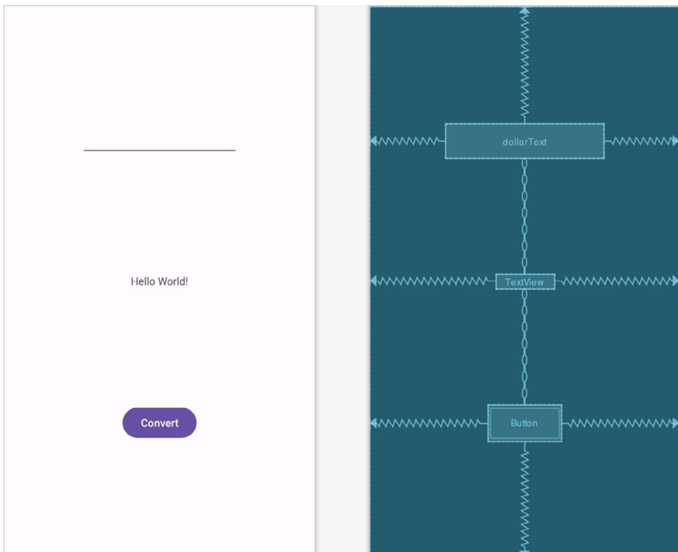


Figure 33-2

Finally, click on the warning icon in the top right-hand corner of the layout editor and convert the hard-coded strings to resources.

33.5 Implementing the View Model

With the user interface layout completed, the data model for the app needs to be created within the view model. Begin by locating the *com.ebookfrenzy.viewmodeldemo* entry in the Project tool window, right-clicking on it, and selecting the *New -> Java Class* menu option. Name the new class MainViewModel and press the keyboard enter

key. Edit the new class file so that it reads as follows:

```
package com.ebookfrenzy.viewmodeldemo.ui.main;

import androidx.lifecycle.ViewModel;

public class  MainViewModel extends ViewModel {

    private static final Float rate = 0.74F;
    private String dollarText = "";
    private Float result = 0F;

    public void setAmount(String value) {
        this.dollarText = value;
        result = Float.parseFloat(dollarText)*rate;
    }

    public Float getResult()
    {
        return result;
    }
}
```

The class declares variables to store the current dollar string value and the converted amount together with getter and setter methods to provide access to those data values. When called, the *setAmount()* method takes the current dollar amount as an argument and stores it in the local *dollarText* variable. The dollar string value is converted to a floating point number, multiplied by a fictitious exchange rate, and the resulting euro value is stored in the *result* variable. The *getResult()* method, on the other hand, returns the current value assigned to the *result* variable.

33.6 Associating the Fragment with the View Model

There needs to be some way for the fragment to obtain a reference to the ViewModel to access the model and observe data changes. A Fragment or Activity maintains references to the ViewModels on which it relies for data using an instance of the ViewModelProvider class.

A ViewModelProvider instance is created using the ViewModelProvider class from within the Fragment. When called, the class initializer is passed a reference to the current Fragment or Activity and returns a ViewModelProvider instance as follows:

```
ViewModelProvider viewModelProvider = new ViewModelProvider(this);
```

Once the ViewModelProvider instance has been created, an index value can be used to request a specific ViewModel class. The provider will then either create a new instance of that ViewModel class or return an existing instance, for example:

```
ViewModel viewModel = viewModelProvider.get(MainViewModel.class);
```

Edit the *FirstFragment.java* file and override the *onCreate()* method to set up the ViewModelProvider:

```
.
.
import androidx.lifecycle.ViewModelProvider;
```

```
import androidx.annotation.Nullable;
.
.
public class FirstFragment extends Fragment {

    private MainViewModel viewModel;
.
.
   @Override
    public void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        viewModel = new ViewModelProvider(this).get(MainViewModel.class);
    }
.
.
```

With access to the model view, code can now be added to the Fragment to begin working with the data model.

33.7 Modifying the Fragment

The fragment class needs to be updated to react to button clicks and interact with the data values stored in the ViewModel. The class will also need references to the three views in the user interface layout to react to button clicks, extract the current dollar value, and display the converted currency amount.

In the chapter entitled *"Creating an Example Android App in Android Studio"*, the onClick property of the Button widget was used to designate the method to be called when the user clicks the button. Unfortunately, this property can only call methods on an Activity and cannot be used to call a method in a Fragment. To overcome this limitation, we must add some code to the Fragment class to set up an onClick listener on the button. This can be achieved in the *onViewCreated()* lifecycle method in the *FirstFragment.java* file as outlined below:

```
.
.
public class MainFragment extends Fragment {

    private MainViewModel viewModel;
.
.
    @Override
    public void onViewCreated(@NonNull View view, @Nullable Bundle 
savedInstanceState) {
        super.onViewCreated(view, savedInstanceState);

        binding.convertButton.setOnClickListener(v -> {

        });
    }
.
.
}
```

With the listener added, any code placed within the *onClick()* method will be called whenever the user clicks the button.

33.8 Accessing the ViewModel Data

When the button is clicked, the *onClick()* method needs to read the current value from the EditText view, confirm that the field is not empty, and then call the *setAmount()* method of the ViewModel instance. The method will then need to call the ViewModel's *getResult()* method and display the converted value on the TextView widget.

Since LiveData has yet to be used in the project, it will also be necessary to get the latest result value from the ViewModel each time the Fragment is created.

Remaining in the *FirstFragment.java* file, implement these requirements as follows in the *onViewCreated()* method:

```
.
.
import java.util.Locale;
.
.
@Override
public void onViewCreated(@NonNull View view, @Nullable Bundle
savedInstanceState) {
    super.onViewCreated(view, savedInstanceState);

    binding.resultText.setText(String.format(Locale.ENGLISH,"%.2f",
            viewModel.getResult()));

        binding.convertButton.setOnClickListener(v -> {
            if (!binding.dollarText.getText().toString().equals("")) {
                viewModel.setAmount(String.format(Locale.ENGLISH,"%s",
                        binding.dollarText.getText()));
                binding.resultText.setText(String.format(Locale.ENGLISH,"%.2f",
                        viewModel.getResult()));
            } else {
                binding.resultText.setText("No Value");
            }
        });
}
```

33.9 Testing the Project

With this project development phase completed, build and run the app on the simulator or a physical device, enter a dollar value, and click the Convert button. The converted amount should appear on the TextView, indicating that the UI controller and ViewModel re-structuring is working as expected.

When the original AndroidSample app was run, rotating the device caused the value displayed on the *resultText* TextView widget to be lost. Repeat this test now with the ViewModelDemo app and note that the current euro value is retained after the rotation. This is because the ViewModel remained in memory as the Fragment was destroyed and recreated, and code was added to the *onViewCreated()* method to update the TextView with the result data value from the ViewModel each time the Fragment re-started.

While this is an improvement on the original AndroidSample app, much more can be done to simplify the project by using LiveData and data binding, both of which are the topics of the next chapters.

33.10 Summary

In this chapter, we revisited the AndroidSample project created earlier in the book and created a new version of the project structured to comply with the Android Jetpack architectural guidelines. The example project also demonstrated the use of ViewModels to separate data handling from user interface-related code. Finally, the chapter showed how the ViewModel approach avoids problems handling Fragment and Activity lifecycles.

Chapter 35

35. An Overview of Android Jetpack Data Binding

In the chapter entitled *"Modern Android App Architecture with Jetpack"*, we introduced the concept of Android Data Binding. We explained how it is used to directly connect the views in a user interface layout to the methods and data located in other objects within an app without the need to write code. This chapter will provide more details on data binding, emphasizing how data binding is implemented within an Android Studio project. The tutorial in the next chapter (*"An Android Jetpack Data Binding Tutorial"*) will provide a practical example of data binding in action.

35.1 An Overview of Data Binding

The Android Jetpack Data Binding Library provides data binding support, primarily providing a simple way to connect the views in a user interface layout to the data stored within the app's code (typically within ViewModel instances). Data binding also provides a convenient way to map user interface controls, such as Button widgets, to event and listener methods within other objects, such as UI controllers and ViewModel instances.

Data binding becomes particularly powerful when used in conjunction with the LiveData component. Consider, for example, an EditText view bound to a LiveData variable within a ViewModel using data binding. When connected in this way, any changes to the data value in the ViewModel will automatically appear within the EditText view, and when using two-way binding, any data typed into the EditText will automatically be used to update the LiveData value. Perhaps most impressive is that this can be achieved with no code beyond that necessary to initially set up the binding.

Connecting an interactive view, such as a Button widget, to a method within a UI controller traditionally required that the developer write code to implement a listener method to be called when the button is clicked. Data binding makes this as simple as referencing the method to be called within the Button element in the layout XML file.

35.2 The Key Components of Data Binding

An Android Studio project is not configured for data binding support by default. Several elements must be combined before an app can begin using data binding. These involve the project build configuration, the layout XML file, data binding classes, and the use of the data binding expression language. While this may appear overwhelming at first, when taken separately, these are quite simple steps that, once completed, are more than worthwhile in terms of saved coding effort. Each element will be covered in detail in the remainder of this chapter. Once these basics have been covered, the next chapter will work through a detailed tutorial demonstrating these steps.

35.2.1 The Project Build Configuration

Before a project can use data binding, it must be configured to use the Android Data Binding Library and to enable support for data binding classes and the binding syntax. Fortunately, this can be achieved with just a few lines added to the module level *build.gradle.kts* file (the one listed as *build.gradle.kts (Module: app)* under *Gradle Scripts* in the Project tool window). The following lists a partial build file with data binding enabled:

.

```
.
android {

    buildFeatures {
        dataBinding = true
    }
.
.
```

35.2.2 The Data Binding Layout File

As we have seen in previous chapters, the user interfaces for an app are typically contained within an XML layout file. Before the views contained within one of these layout files can take advantage of data binding, the layout file must be converted to a *data binding layout file*.

As outlined earlier in the book, XML layout files define the hierarchy of components in the layout, starting with a top-level or *root view*. Invariably, this root view takes the form of a layout container such as a ConstraintLayout, FrameLayout, or LinearLayout instance, as is the case in the *fragment_main.xml* file for the ViewModelDemo project:

```
<?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=".ui.main.MainFragment">
.
.
</androidx.constraintlayout.widget.ConstraintLayout>
```

To use data binding, the layout hierarchy must have a *layout* component as the root view, which, in turn, becomes the parent of the current root view.

In the case of the above example, this would require that the following changes be made to the existing layout file:

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

<layout xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    xmlns:android="http://schemas.android.com/apk/res/android">

        <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=".ui.main.MainFragment">
.
.
        </androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```

35.2.3 The Layout File Data Element

The data binding layout file needs some way to declare the classes within the project to which the views in the layout are to be bound (for example, a ViewModel or UI controller). Having declared these classes, the layout file will need a variable name to reference those instances within binding expressions.

This is achieved using the *data* element, an example of which is shown below:

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

<layout xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    xmlns:android="http://schemas.android.com/apk/res/android">

    <data>
        <variable
            name="myViewModel"
            type="com.ebookfrenzy.myapp.ui.main.MainViewModel" />
    </data>

    <androidx.constraintlayout.widget.ConstraintLayout
        android:id="@+id/main"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        tools:context=".ui.main.MainFragment">
.
.
</layout>
```

The above data element declares a new variable named *myViewModel* of type MainViewModel (note that it is necessary to declare the full package name of the MyViewModel class when declaring the variable).

The data element can import other classes that may then be referenced within binding expressions elsewhere in the layout file. For example, if you have a class containing a method that needs to be called on a value before it is displayed to the user, the class could be imported as follows:

```
<data>
        <import type="com.ebookfrenzy.MyFormattingTools" />
        <variable
            name="viewModel"
            type="com.ebookfrenzy.myapp.ui.main.MainViewModel" />
    </data>
```

35.2.4 The Binding Classes

For each class referenced in the *data* element within the binding layout file, Android Studio will automatically generate a corresponding *binding class*. This subclass of the Android ViewDataBinding class will be named based on the layout filename using word capitalization and the *Binding* suffix. Therefore, the binding class for a layout file named *fragment_main.xml* file will be named *FragmentMainBinding*. The binding class contains the bindings specified within the layout file and maps them to the variables and methods within the bound objects.

Although the binding class is generated automatically, code must be written to create an instance of the class based on the corresponding data binding layout file. Fortunately, this can be achieved by making use of the DataBindingUtil class.

The initialization code for an Activity or Fragment will typically set the content view or "inflate" the user interface layout file. This means that the code opens the layout file, parses the XML, and creates and configures all of the view objects in memory. In the case of an existing Activity class, the code to achieve this can be found in the *onCreate()* method and will read as follows:

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

In the case of a Fragment, this takes place in the *onCreateView()* method:

```
return inflater.inflate(R.layout.fragment_main, container, false);
```

All that is needed to create the binding class instances within an Activity class is to modify this initialization code as follows:

```
ActivityMainBinding binding;

binding = DataBindingUtil.setContentView(this, R.layout.activity_main, false);
```

In the case of a Fragment, the code would read as follows:

```
FragmentMainBinding binding;

binding = DataBindingUtil.inflate(
        inflater, R.layout.fragment_main, container, false);

binding.setLifecycleOwner(this);
View view = binding.getRoot();
return view;
```

35.2.5 Data Binding Variable Configuration

As outlined above, the data binding layout file contains the *data* element, which contains *variable* elements consisting of variable names and the class types to which the bindings are to be established. For example:

```
<data>
    <variable
        name="viewModel"
        type="com.ebookfrenzy.viewmodeldemo.ui.main.MainViewModel" />
    <variable
            name="uiController"
            type="com.ebookfrenzy.viewmodeldemo_databinding.ui.main.MainFragment"
/>
</data>
```

In the above example, the first variable knows that it will be binding to an instance of a ViewModel class of type MainViewModel but has yet to be connected to an actual MainViewModel object instance. This requires the additional step of assigning the MainViewModel instance used within the app to the variable declared in the layout file. This is performed via a call to the *setVariable()* method of the data binding instance, a reference to which was obtained in the previous chapter:

```
MainViewModel mViewModel = new ViewModelProvider(this).get(MainViewModel.class);
binding.setVariable(viewModel, mViewModel);
```

The second variable in the above data element references a UI controller class in the form of a Fragment named MainFragment. In this situation, the code within a UI controller (be it an Activity or Fragment) would need to assign itself to the variable as follows:

```
binding.setVariable(uiController, this);
```

35.2.6 Binding Expressions (One-Way)

Binding expressions define how a particular view interacts with bound objects. For example, a binding expression on a Button might declare which method on an object is called in response to a click. Alternatively, a binding expression might define which data value stored in a ViewModel is to appear within a TextView and how it is to be presented and formatted.

Binding expressions use a declarative language that allows logic and access to other classes and methods to decide how bound data is used. Expressions can, for example, include mathematical expressions, method calls, string concatenations, access to array elements, and comparison operations. In addition, all standard Java language libraries are imported by default, so many things that can be achieved in Java can also be performed in a binding expression. As already discussed, the data element may also be used to import custom classes to add more capability to expressions.

A binding expression begins with an @ symbol followed by the expression enclosed in curly braces ({}).

Consider, for example, a ViewModel instance containing a variable named *result*. Assume that this class has been assigned to a variable named *viewModel* within the data binding layout file and needs to be bound to a TextView object so that the view always displays the latest result value. If this value were stored as a String object, this would be declared within the layout file as follows:

```
<TextView
    android:id="@+id/resultText"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@{viewModel.result}"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
```

In the above XML, the *text* property is set to the value stored in the *result* LiveData property of the viewModel object.

Consider, however, that the result is stored within the model as a Float value instead of a String. That being the case, the above expression would cause a compilation error. Clearly, the Float value must be converted to a string before the TextView can display it. To resolve issues such as this, the binding expression can include the necessary steps to complete the conversion using the standard Java language classes:

```
android:text="@{String.valueOf(viewModel.result)}"
```

When running the app after making this change, it is important to be aware that the following warning may appear in the Android Studio console:

```
warning: myViewModel.result.getValue() is a boxed field but needs to be un-boxed
to execute String.valueOf(viewModel.result.getValue()).
```

Values in Java can take the form of primitive values such as the *boolean* type (referred to as being *unboxed*) or wrapped in a Java object such as the *Boolean* type and accessed via reference to that object (i.e., *boxed*). The unboxing process involves unwrapping the primitive value from the object.

To avoid this message, wrap the offending operation in a *safeUnbox()* call as follows:

```
android:text="@{String.valueOf(safeUnbox(myViewModel.result))}"
```

String concatenation may also be used. For example, to include the word "dollars" after the result string value, the following expression would be used:

```
android:text='@{String.valueOf(safeUnbox(myViewModel.result)) + " dollars"}'
```

Note that since the appended result string is wrapped in double quotes, the expression is now encapsulated with single quotes to avoid syntax errors.

The expression syntax also allows ternary statements to be declared. In the following expression, the view will display different text depending on whether or not the result value is greater than 10.

```
@{myViewModel.result > 10 ? "Out of range" : "In range"}
```

Expressions may also be constructed to access specific elements in a data array:

```
@{myViewModel.resultsArray[3]}
```

35.2.7 Binding Expressions (Two-Way)

The type of expression covered so far is called *one-way binding*. In other words, the layout is constantly updated as the corresponding value changes, but changes to the value from within the layout do not update the stored value.

A *two-way binding*, on the other hand, allows the data model to be updated in response to changes in the layout. An EditText view, for example, could be configured with a two-way binding so that when the user enters a different value, that value is used to update the corresponding data model value. When declaring a two-way expression, the syntax is similar to a one-way expression except that it begins with @=. For example:

```
android:text="@={myViewModel.result}"
```

35.2.8 Event and Listener Bindings

Binding expressions may also trigger method calls in response to events on a view. A Button view, for example, can be configured to call a method when clicked. In the chapter entitled *"Creating an Example Android App in Android Studio"*, for example, the onClick property of a button was configured to call a method within the app's main activity named *convertCurrency()*. Within the XML file, this was represented as follows:

```
android:onClick="convertCurrency"
```

The *convertCurrency()* method was declared along the following lines:

```
public void convertCurrency(View view) {
.
.
}
```

Note that this type of method call is always passed a reference to the view on which the event occurred. The same effect can be achieved in data binding using the following expression (assuming the layout has been bound to a

class with a variable name of *uiController*):

```
android:onClick="@{uiController::convertCurrency}"
```

Another option, and one which provides the ability to pass parameters to the method, is referred to as a *listener binding*. The following expression uses this approach to call a method on the same viewModel instance with no parameters:

```
android:onClick='@{() -> myViewModel.methodOne()}'
```

The following expression calls a method that expects three parameters:

```
android:onClick='@{() -> myViewModel.methodTwo(viewModel.result, 10, "A String")}'
```

Binding expressions provide a rich and flexible language to bind user interface views to data and methods in other objects. This chapter has only covered the most common use cases. To learn more about binding expressions, review the Android documentation online at:

```
https://developer.android.com/topic/libraries/data-binding/expressions
```

35.3 Summary

Android data bindings provide a system for creating connections between the views in a user interface layout and the data and methods of other objects within the app architecture without writing code. Once some initial configuration steps have been performed, data binding involves using binding expressions within the view elements of the layout file. These binding expressions can be either one-way or two-way and may also be used to bind methods to be called in response to events such as button clicks within the user interface.

Chapter 47

47. Working with the RecyclerView and CardView Widgets

The RecyclerView and CardView widgets work together to provide scrollable lists of information to the user in which the information is presented as individual cards. Details of both classes will be covered in this chapter before working through the design and implementation of an example project.

47.1 An Overview of the RecyclerView

Much like the ListView class outlined in the chapter entitled *"Working with the Floating Action Button and Snackbar"*, the RecyclerView's purpose is to allow information to be presented to the user as a scrollable list. The RecyclerView, however, provides several advantages over the ListView. In particular, the RecyclerView is significantly more efficient in managing the views that make up a list, reusing existing views that makeup list items as they scroll off the screen instead of creating new ones (hence the name "recycler"). This increases the performance and reduces the resources a list uses, a feature of particular benefit when presenting large amounts of data to the user.

Unlike the ListView, the RecyclerView also provides a choice of three built-in layout managers to control how the list items are presented to the user:

- **LinearLayoutManager** – The list items are presented as horizontal or vertical scrolling lists.

Figure 47-1

- **GridLayoutManager** – The list items are presented in grid format. This manager is best used when the list items are of uniform size.

Figure 47-2

- **StaggeredGridLayoutManager** - The list items are presented in a staggered grid format. This manager is best

used when the list items are of different sizes.

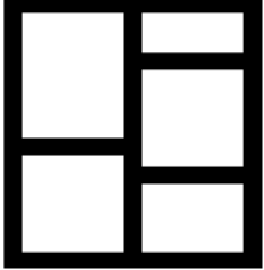


Figure 47-3

For situations where none of the three built-in managers provide the necessary layout, custom layout managers may be implemented by subclassing the RecyclerView.LayoutManager class.

Each list item displayed in a RecyclerView is created as an instance of the ViewHolder class. The ViewHolder instance contains everything necessary for the RecyclerView to display the list item, including the information to be displayed and the view layout used to display the item.

As with the ListView, the RecyclerView depends on an adapter to act as the intermediary between the RecyclerView instance and the data to be displayed to the user. The adapter is created as a subclass of the RecyclerView.Adapter class and must, at a minimum, implement the following methods, which will be called at various points by the RecyclerView object to which the adapter is assigned:

- **getItemCount()** – This method must return a count of the number of items to be displayed in the list.
- **onCreateViewHolder()** – This method creates and returns a ViewHolder object initialized with the view that is to be used to display the data. This view is typically created by inflating the XML layout file.
- **onBindViewHolder()** – This method is passed the ViewHolder object created by the *onCreateViewHolder()* method together with an integer value indicating the list item that is about to be displayed. Contained within the ViewHolder object is the layout assigned by the *onCreateViewHolder()* method. The *onBindViewHolder()* method is responsible for populating the views in the layout with the text and graphics corresponding to the specified item and returning the object to the RecyclerView, where it will be presented to the user.

Adding a RecyclerView to a layout is a matter of adding the appropriate element to the XML content layout file of the activity in which it is to appear. For example:

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

    <androidx.recyclerview.widget.RecyclerView
        android:id="@+id/recycler_view"
```

```
        android:layout_width="0dp"
        android:layout_height="0dp"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        tools:listItem="@layout/card_layout" />

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

The RecyclerView has been embedded into the CoordinatorLayout of a main activity layout file along with the AppBar and Toolbar in the above example. This provides some additional features, such as configuring the Toolbar and AppBar to scroll off the screen when the user scrolls up within the RecyclerView (a topic covered in more detail in the chapter entitled *"Working with the AppBar and Collapsing Toolbar Layouts"*).

47.2 An Overview of the CardView

The CardView class is a user interface view that allows information to be presented in groups using a card metaphor. Cards are usually presented in lists using a RecyclerView instance and may be configured to appear with shadow effects and rounded corners. Figure 47-4, for example, shows three CardView instances configured to display a layout consisting of an ImageView and two TextViews:

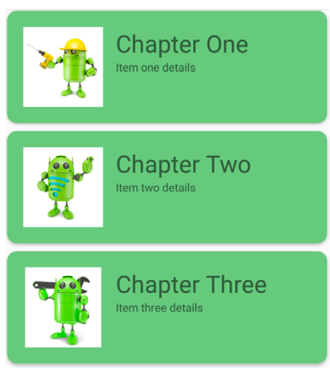


Figure 47-4

The user interface layout to be presented with a CardView instance is defined within an XML layout resource file and loaded into the CardView at runtime. The CardView layout can contain a layout of any complexity using the standard layout managers such as RelativeLayout and LinearLayout. The following XML layout file represents a card view layout consisting of a RelativeLayout and a single ImageView. The card is configured to be elevated to create a shadowing effect and to appear with rounded corners:

```
<?xml version="1.0" encoding="utf-8"?>
    <androidx.cardview.widget.CardView
        xmlns:card_view="http://schemas.android.com/apk/res-auto"
        xmlns:android="http://schemas.android.com/apk/res/android"
        android:id="@+id/card_view"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_margin="5dp"
```

```
        card_view:cardCornerRadius="12dp"
        card_view:cardElevation="3dp"
        card_view:contentPadding="4dp">

        <RelativeLayout
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:padding="16dp" >

            <ImageView
                android:layout_width="100dp"
                android:layout_height="100dp"
                android:id="@+id/item_image"
                android:layout_alignParentLeft="true"
                android:layout_alignParentTop="true"
                android:layout_marginRight="16dp" />
        </RelativeLayout>
</androidx.cardview.widget.CardView>
```

When combined with the RecyclerView to create a scrollable list of cards, the *onCreateViewHolder()* method of the recycler view inflates the layout resource file for the card, assigns it to the ViewHolder instance and returns it to the RecyclerView instance.

47.3 Summary

This chapter has introduced the Android RecyclerView and CardView components. The RecyclerView provides a resource-efficient way to display scrollable lists of views within an Android app. The CardView is useful when presenting groups of data (such as a list of names and addresses) in the form of cards. As previously outlined and demonstrated in the tutorial contained in the next chapter, RecyclerView and CardView are particularly useful when combined.

Chapter 48

48. An Android RecyclerView and CardView Tutorial

This chapter will create an example project that uses both the CardView and RecyclerView components to create a scrollable list of cards. The completed app will display a list of cards containing images and text. In addition to displaying the list of cards, the project will be implemented such that selecting a card causes messages to be displayed to the user indicating which card was tapped.

48.1 Creating the CardDemo Project

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

Enter *CardDemo* into the Name field and specify *com.ebookfrenzy.carddemo* 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.

48.2 Modifying the Basic Views Activity Project

Since the Basic Views Activity was selected, the layout includes a floating action button which is not required for this project. Load the *activity_main.xml* layout file into the Layout Editor tool, select the floating action button, and tap the keyboard delete key to remove the object from the layout. Edit the *MainActivity.java* file and remove the floating action button and navigation controller code from the onCreate method as follows:

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

    binding = ActivityMainBinding.inflate(getLayoutInflater());
    setContentView(binding.getRoot());

    setSupportActionBar(binding.toolbar);

    NavController navController =
         Navigation.findNavController(this, R.id.nav_host_fragment_content_main);
    appBarConfiguration =
         new AppBarConfiguration.Builder(navController.getGraph()).build();
    NavigationUI.setupActionBarWithNavController(this, navController,
         appBarConfiguration);

    binding.fab.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            Snackbar.make(view, "Replace with your own action", Snackbar.LENGTH_
```

```
LONG)
                        .setAction("Action", null).show();
          }
      });
}
```

Also, remove the *onSupportNavigateUp()* method, then open the *content_main.xml* file and delete the *nav_host_fragment_content_main* object from the layout so that only the ConstraintLayout parent remains.

48.3 Designing the CardView Layout

The layout of the views contained within the cards will be defined within a separate XML layout file. Within the Project tool window, right-click on the *app -> res -> layout* entry and select the *New -> Layout Resource File* menu option. In the New Resource Dialog, enter *card_layout* into the *File name:* field and *androidx.cardview.widget.CardView* into the root element field before clicking on the *OK* button.

Load the *card_layout.xml* file into the Layout Editor tool, switch to Code mode, and modify the layout so that it reads as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.cardview.widget.CardView
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:id="@+id/card_view"
    android:layout_margin="5dp"
    app:cardBackgroundColor="#80B3EF"
    app:cardCornerRadius="12dp"
    app:cardElevation="3dp"
    app:contentPadding="4dp" >

    <androidx.constraintlayout.widget.ConstraintLayout
        android:id="@+id/relativeLayout"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:padding="16dp">

        <ImageView
            android:id="@+id/itemImage"
            android:layout_width="100dp"
            android:layout_height="100dp"
            app:layout_constraintLeft_toLeftOf="parent"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintTop_toTopOf="parent" />

        <TextView
            android:id="@+id/itemTitle"
            android:layout_width="236dp"
```

```
            android:layout_height="39dp"
            android:layout_marginStart="16dp"
            android:textSize="30sp"
            app:layout_constraintLeft_toRightOf="@+id/itemImage"
            app:layout_constraintStart_toEndOf="@+id/itemImage"
            app:layout_constraintTop_toTopOf="parent" />

        <TextView
            android:id="@+id/itemDetail"
            android:layout_width="236dp"
            android:layout_height="16dp"
            android:layout_marginStart="16dp"
            android:layout_marginTop="8dp"
            app:layout_constraintLeft_toRightOf="@+id/itemImage"
            app:layout_constraintStart_toEndOf="@+id/itemImage"
            app:layout_constraintTop_toBottomOf="@+id/itemTitle" />
    </androidx.constraintlayout.widget.ConstraintLayout>
</androidx.cardview.widget.CardView>
```

48.4 Adding the RecyclerView

Select the *content_main.xml* layout file and drag a RecyclerView object from the *Containers* section of the palette onto the layout so that it is positioned in the center of the screen, where it should automatically resize to fill the entire screen. Use the *Infer constraints* toolbar button to add any missing layout constraints to the view. Using the Attributes tool window, change the ID of the RecyclerView instance to *recyclerView* and the layout_width and layout_height properties to *match_constraint*.

48.5 Adding the Image Files

In addition to the two TextViews, the card layout contains an ImageView on which the Recycler adapter has been configured to display images. Before the project can be tested, these images must be added. The images that will be used for the project are named *android_image_<n>.jpg* and can be found in the *project_icons* folder of the sample code download available from the following URL:

https://www.ebookfrenzy.com/retail/giraffejava/index.php

Locate these images in the file system navigator for your operating system and select and copy the eight images. Right click on the *app -> res -> drawable* entry in the Project tool window and select Paste to add the files to the folder:

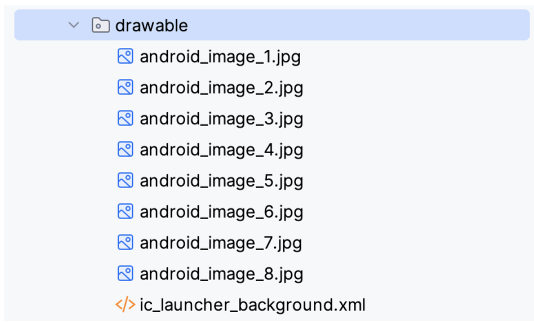


Figure 48-1

48.6 Creating the RecyclerView Adapter

As outlined in the previous chapter, the RecyclerView needs to have an adapter to handle the creation of the list items. Add this new class to the project by right-clicking on the *app -> java -> com.ebookfrenzy.carddemo* entry in the Project tool window and selecting the *New -> Java Class* menu option. In the new class dialog, enter *RecyclerAdapter* into the *Name* field and select *Class* from the list before tapping the Return keyboard key to create the new Java class file.

Edit the new *RecyclerAdapter.java* file to add some import directives and to declare that the class now extends *RecyclerView.Adapter*. Rather than create a separate class to provide the data to be displayed, some basic arrays will also be added to the adapter to act as the data for the app:

```
package com.ebookfrenzy.carddemo;

import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.ImageView;
import android.widget.TextView;

import androidx.annotation.NonNull;
import androidx.recyclerview.widget.RecyclerView;

public class RecyclerAdapter extends RecyclerView.Adapter<RecyclerAdapter.
ViewHolder> {

    final private String[] titles = {"Chapter One",
            "Chapter Two",
            "Chapter Three",
            "Chapter Four",
            "Chapter Five",
            "Chapter Six",
            "Chapter Seven",
            "Chapter Eight"};

    final private String[] details = {"Item one details",
            "Item two details", "Item three details",
            "Item four details", "Item five details",
            "Item six details", "Item seven details",
            "Item eight details"};

   final private int[] images = { R.drawable.android_image_1,
                            R.drawable.android_image_2,
                            R.drawable.android_image_3,
                            R.drawable.android_image_4,
                            R.drawable.android_image_5,
                            R.drawable.android_image_6,
                            R.drawable.android_image_7,
```

```
                                        R.drawable.android_image_8 };
}
```

Within the RecyclerAdapter class, we now need our own implementation of the ViewHolder class configured to reference the view elements in the *card_layout.xml* file. Remaining within the *RecyclerAdapter.java*, file implement this class as follows:

```
public class RecyclerAdapter extends RecyclerView.Adapter<RecyclerAdapter.
ViewHolder> {
.
.
   static class ViewHolder extends RecyclerView.ViewHolder {

        ImageView itemImage;
        TextView itemTitle;
        TextView itemDetail;

        ViewHolder(View itemView) {
            super(itemView);
            itemImage = itemView.findViewById(R.id.itemImage);
            itemTitle = itemView.findViewById(R.id.itemTitle);
            itemDetail = itemView.findViewById(R.id.itemDetail);
        }
    }
}
```

The ViewHolder class contains an ImageView and two TextView variables together with a constructor method that initializes those variables with references to the three view items in the *card_layout.xml* file.

The next item to be added to the *RecyclerAdapter.java* file is the implementation of the *onCreateViewHolder()* method:

```
@NonNull
@Override
public ViewHolder onCreateViewHolder(ViewGroup viewGroup, int i) {
    View v = LayoutInflater.from(viewGroup.getContext())
                .inflate(R.layout.card_layout, viewGroup, false);
    return new ViewHolder(v);
}
```

This method will be called by the RecyclerView to obtain a ViewHolder object. It inflates the view hierarchy *card_layout.xml* file and creates an instance of our ViewHolder class initialized with the view hierarchy before returning it to the RecyclerView.

The purpose of the *onBindViewHolder()* method is to populate the view hierarchy within the ViewHolder object with the data to be displayed. It is passed the ViewHolder object and an integer value indicating the list item that is to be displayed. This method should now be added, using the item number as an index into the data arrays. This data is then displayed on the layout views using the references created in the constructor method of the ViewHolder class:

```
@Override
public void onBindViewHolder(ViewHolder viewHolder, int i) {
```

```
    viewHolder.itemTitle.setText(titles[i]);
    viewHolder.itemDetail.setText(details[i]);
    viewHolder.itemImage.setImageResource(images[i]);
}
```

The final requirement for the adapter class is an implementation of the *getItem()* method which, in this case, returns the number of items in the *titles* array:

```
@Override
public int getItemCount() {
    return titles.length;
}
```

48.7 Initializing the RecyclerView Component

At this point, the project consists of a RecyclerView instance, an XML layout file for the CardView instances and an adapter for the RecyclerView. The last step before testing the progress so far is to initialize the RecyclerView with a layout manager, create an instance of the adapter and assign that instance to the RecyclerView object. For the purposes of this example, the RecyclerView will be configured to use the LinearLayoutManager layout option.

There is a slight complication here because we need to be able to use view binding to access the recyclerView component from within the MainActivity class. The problem is that recyclerView is contained within the *content_main.xml* layout file which is, in turn, included in the *activity_main.xml* file. To be able to reach down into the *content_main.xml* file, we need to assign it an id at the point that it is included. To do this, edit the *activity_main.xml* file and modify the *include* element so that it reads as follows:

```
.
.
   <include
        android:id="@+id/contentMain"
        layout="@layout/content_main" />
.
.
```

With an id assigned to the included file, the recyclerView component can be accessed using the following binding:

```
binding.contentMain.recyclerView
```

Now edit the *MainActivity.java* file and modify the *onCreate()* method to implement the initialization code:

```
package com.ebookfrenzy.carddemo;
.
.
import androidx.recyclerview.widget.LinearLayoutManager;
import androidx.recyclerview.widget.RecyclerView;
.
.
public class MainActivity extends AppCompatActivity {

    private RecyclerView recyclerView;
    private RecyclerView.LayoutManager layoutManager;
```

```
    private RecyclerView.Adapter adapter;
.
.
    @Override
    protected void onCreate(Bundle savedInstanceState) {
.
.
        setSupportActionBar(toolbar);

        layoutManager = new LinearLayoutManager(this);
        binding.contentMain.recyclerView.setLayoutManager(layoutManager);

        adapter = new RecyclerAdapter();
        binding.contentMain.recyclerView.setAdapter(adapter);
    }
.
.
}
```

48.8 Testing the Application

Compile and run the app on a physical device or emulator session and scroll through the different card items in the list:

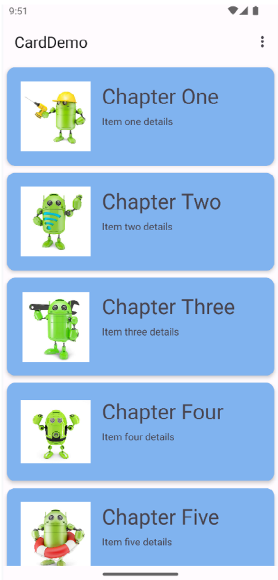


Figure 48-2

When building the project, you may encounter an error that reads in part:

```
Duplicate class kotlin.collections.jdk8.CollectionsJDK8Kt found in modules
kotlin-stdlib
```

This error is caused by a bug in the Android Studio build toolchain and can be resolved by making the following changes to the *build.gradle.kts (Module: app)* file:

```
dependencies {
.
.
    implementation(platform("org.jetbrains.kotlin:kotlin-bom:1.8.0"))
.
.
}
```

48.9 Responding to Card Selections

The last phase of this project is to make the cards in the list selectable so that clicking on a card triggers an event within the app. For this example, the cards will be configured to present a message on the display when tapped by the user. To respond to clicks, the ViewHolder class needs to be modified to assign an onClickListener on each item view. Edit the *RecyclerAdapter.java* file and modify the ViewHolder class declaration so that it reads as follows:

```
.
.
import com.google.android.material.snackbar.Snackbar;
.
.
class ViewHolder extends RecyclerView.ViewHolder{

    ImageView itemImage;
    TextView itemTitle;
    TextView itemDetail;

    ViewHolder(View itemView) {
        super(itemView);
        itemImage = itemView.findViewById(R.id.item_image);
        itemTitle = itemView.findViewById(R.id.item_title);
        itemDetail = itemView.findViewById(R.id.item_detail);

        itemView.setOnClickListener(new View.OnClickListener() {
            @Override public void onClick(View v) {

            }
        });
    }
}
```

Within the body of the onClick handler, code can now be added to display a message indicating that the card has been clicked. Given that the actions performed as a result of a click will likely depend on which card was tapped, it is also important to identify the selected card. This information can be obtained via a call to the *getAdapterPosition()* method of the *RecyclerView.ViewHolder* class. Remaining within the *RecyclerAdapter.java* file, add code to the *onClick* handler so it reads as follows:

```
@override
public void onClick(View v) {
```

```
    int position = getAdapterPosition();

    Snackbar.make(v, "Click detected on item " + (position + 1),
            Snackbar.LENGTH_LONG)
            .setAction("Action", null).show();
        }
});
```

The last task is to enable the material design ripple effect that appears when items are tapped within Android applications. This involves the addition of some properties to the declaration of the CardView instance in the *card_layout.xml* file as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.cardview.widget.CardView
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:card_view="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:id="@+id/card_view"
    android:layout_margin="5dp"
    app:cardBackgroundColor="#80B3EF"
    app:cardCornerRadius="12dp"
    app:cardElevation="3dp"
    app:contentPadding="4dp"
    android:foreground="?selectableItemBackground"
    android:clickable="true" >
```

Run the app once again and verify that tapping a card in the list triggers both the standard ripple effect at the point of contact and the appearance of a Snackbar reporting the number of the selected item.

48.10 Summary

This chapter has worked through the steps involved in combining the CardView and RecyclerView components to display a scrollable list of card-based items. The example also covered the detection of clicks on list items, including the identification of the selected item and the enabling of the ripple effect visual feedback on the tapped CardView instance.

Chapter 52

52. An Overview of Android Services

The Android Service class is designed to allow applications to initiate and perform background tasks. Unlike broadcast receivers, which are intended to perform a task quickly and then exit, services are designed to perform tasks that take a long time to complete (such as downloading a file over an internet connection or streaming music to the user) but do not require a user interface.

This chapter will provide an overview of the services available, including *bound* and *intent services*. Once these basics have been covered, subsequent chapters will work through some examples of services in action.

52.1 Intent Service

As previously outlined, services run by default within the same main thread as the component from which they are launched. As such, any CPU-intensive tasks that need to be performed by the service should occur within a new thread, thereby avoiding impacting the performance of the calling application.

The *JobIntentService* class is a convenience class (subclassed from the Service class) that sets up a worker thread for handling background tasks and handles each request asynchronously. Once the service has handled all queued requests, it exits. All that is required when using the JobIntentService class is to implement the *onHandleWork()* method, containing the code to be executed for each request.

For services that do not require synchronous processing of requests, JobIntentService is the recommended option. However, services requiring synchronous handling of requests will need to subclass from the Service class and manually implement and manage threading to handle any CPU-intensive tasks efficiently.

52.2 Bound Service

A bound service allows a launching component to interact with and receive results from the service. This interaction can also occur across process boundaries through the implementation of interprocess communication (IPC). An activity might, for example, start a service to handle audio playback. The activity will, in all probability, include a user interface providing controls to the user to pause playback or skip to the next track. Similarly, the service will likely need to communicate information to the calling activity to indicate that the current audio track has ended and provide details of the next track that is about to start playing.

A component (referred to in this context as a *client*) starts and *binds* to a bound service via a call to the *bindService()* method. Also, multiple components may bind to a service simultaneously. When a client no longer requires the service binding, a call should be made to the *unbindService()* method. When the last bound client unbinds from a service, the Android runtime system will terminate the service. It is important to remember that a bound service may also be started via a call to *startService()*. Once started, components may then bind to it via *bindService()* calls. When a bound service is launched via a call to *startService()*, it will continue to run even after the last client unbinds from it.

A bound service must include an implementation of the *onBind()* method, which is called both when the service is initially created and when other clients subsequently bind to the running service. The purpose of this method is to return to binding clients an object of type *IBinder* containing the information needed by the client to communicate with the service.

When implementing the communication between a client and a bound service, the recommended technique depends on whether the client and service reside in the same or different processes and whether or not the service

is private to the client. Local communication can be achieved by extending the Binder class and returning an instance from the *onBind()* method. Interprocess communication, on the other hand, requires Messenger and Handler implementation. Details of both of these approaches will be covered in later chapters.

52.3 The Anatomy of a Service

As has already been mentioned, a service must be created as a subclass of the Android Service class (more specifically, *android.app.Service*) or a sub-class thereof (such as *android.app.IntentService*). As part of the subclassing procedure, one or more of the following superclass callback methods must be overridden, depending on the exact nature of the service being created:

- **onStartCommand()** – This method is called when another component starts the service via a call to the *startService()* method. This method does not need to be implemented for bound services.
- **onBind()** – Called when a component binds to the service via a call to the *bindService()* method. When implementing a bound service, this method must return an *IBinder* object facilitating communication with the client.
- **onCreate()** – Intended as a location to perform initialization tasks, this method is called immediately before the call to either *onStartCommand()* or the *first* call to the *onBind()* method.
- **onDestroy()** – Called when the service is being destroyed.
- **onHandleWork()** – Applies only to JobIntentService subclasses. This method is called to handle the processing for the service. It is executed in a separate thread from the main application.

Note that the IntentService class includes its own implementations of the *onStartCommand()* and *onBind()* callback methods, so these do not need to be implemented in subclasses.

52.4 Controlling Destroyed Service Restart Options

The *onStartCommand()* callback method is required to return an integer value to define what should happen with regard to the service if the Android runtime system destroys it. Possible return values for these methods are as follows:

- **START_NOT_STICKY** – Indicates to the system that the service should not be restarted if it is destroyed unless there are pending intents awaiting delivery.
- **START_STICKY** – Indicates that the service should be restarted as soon as possible after it has been destroyed if the destruction occurred after the *onStartCommand()* method returned. If no pending intents are waiting to be delivered, the *onStartCommand()* callback method is called with a NULL intent value. The intent being processed when the service was destroyed is discarded.
- **START_REDELIVER_INTENT** – Indicates that if the service was destroyed after returning from the *onStartCommand()* callback method, the service should be restarted with the current intent redelivered to the *onStartCommand()* method followed by any pending intents.

52.5 Declaring a Service in the Manifest File

For a service to be usable, it must first be declared within a manifest file. This involves embedding an appropriately configured *<service>* element into an existing *<application>* entry. At a minimum, the <service> element must contain a property declaring the class name of the service, as illustrated in the following XML fragment:

```
.
.
    <application
```

```
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name" >
        <activity
            android:label="@string/app_name"
            android:name=".MainActivity" >
            <intent-filter>
              <action android:name="android.intent.action.MAIN" />
              <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <service android:name=".MyService>
           </service>
    </application>
</manifest>
```

By default, services are declared public in that they can be accessed by components outside the application package in which they reside. To make a service private, the *android:exported* property must be declared as *false* within the <service> element of the manifest file. For example:

```
<service android:name="MyService"
              android:exported="false">
</service>
```

When working with JobIntentService, the manifest Service declaration must also request the BIND_JOB_SERVICE permission as follows:

```
<service
    android:name=".MyJobIntentService"
    android:permission="android.permission.BIND_JOB_SERVICE" />
```

As previously discussed, services run within the same process as the calling component by default. To force a service to run within its own process, add an *android:process* property to the <service> element, declaring a name for the process prefixed with a colon (:):

```
<service android:name=".MyService"
android:exported="false"
android:process=":myprocess">
</service>
```

The colon prefix indicates that the new process is private to the local application. If the process name begins with a lowercase letter instead of a colon, however, the process will be global and available for use by other components.

Finally, using the same intent filter mechanisms outlined for activities, a service may also advertise capabilities to other applications running on the device. For more details on intent filters, refer to the chapter *"An Overview of Android Intents"*.

52.6 Starting a Service Running on System Startup

Given the background nature of services, it is not uncommon for a service to need to be started when an Android-based system first boots up. This can be achieved by creating a broadcast receiver with an intent filter configured to listen for the system *android.intent.action.BOOT_COMPLETED* intent. When such an intent is detected, the broadcast receiver would invoke the necessary service and then return. Note that, to function, such

a broadcast receiver must request the *android.permission.RECEIVE_BOOT_COMPLETED* permission.

52.7 Summary

Android services are a powerful mechanism that allows applications to perform tasks in the background. A service, once launched, will continue to run regardless of whether the calling application is the foreground task or not and even if the component that initiated the service is destroyed.

Services are subclassed from the Android Service class. Bound services provide a communication interface to other client components and generally run until the last client unbinds from the service.

By default, services run locally within the same process and main thread as the calling application. A new thread should, therefore, be created within the service to handle CPU-intensive tasks. Remote services may be started within a separate process by making a minor configuration change to the corresponding <service> entry in the application manifest file.

The IntentService class (a subclass of the Android Service class) provides a convenient mechanism for handling asynchronous service requests within a separate worker thread.

Chapter 53

53. An Overview of Android Intents

By this stage of the book, it should be clear that Android applications comprise one or more activities, among other things. However, an area that has yet to be covered in extensive detail is the mechanism by which one activity can trigger the launch of another activity. As outlined briefly in the chapter entitled *"The Anatomy of an Android Application"*, this is achieved primarily using *Intents*.

Before working through some Android Studio-based example implementations of intents in the following chapters, this chapter aims to provide an overview of intents in the form of *explicit intents* and *implicit intents*, together with an introduction to *intent filters*.

53.1 An Overview of Intents

Intents (*android.content.Intent*) are the messaging system by which one activity can launch another activity. An activity can, for example, issue an intent to request the launch of another activity contained within the same application. Intents also go beyond this concept by allowing an activity to request the services of any other appropriately registered activity on the device for which permissions are configured. Consider, for example, an activity contained within an application that requires a web page to be loaded and displayed to the user. Rather than the application having to contain a second activity to perform this task, the code can send an intent to the Android runtime requesting the services of any activity that has registered the ability to display a web page. The runtime system will match the request to available activities on the device and either launch the activity that matches or, in the event of multiple matches, allow the user to decide which activity to use.

Intents also allow data transfer from the sending to the receiving activity. In the previously outlined scenario, for example, the sending activity would need to send the URL of the web page to be displayed to the second activity. Similarly, the receiving activity may be configured to return data to the sending activity when the required tasks are completed.

Though not covered until later chapters, it is also worth highlighting that, in addition to launching activities, intents are also used to launch and communicate with services and broadcast receivers.

Intents are categorized as either *explicit* or *implicit.*

53.2 Explicit Intents

An *explicit intent* requests the launch of a specific activity by referencing the target activity's component name (which is the class name). This approach is most common when launching an activity residing in the same application as the sending activity (since the class name is known to the developer).

An explicit intent is issued by creating an instance of the Intent class, passing through the activity context and the component name of the activity to be launched. A call is then made to the *startActivity()* method, passing the intent object as an argument. For example, the following code fragment issues an intent for the activity with the class name ActivityB to be launched:

```
Intent i = new Intent(this, ActivityB.class);
startActivity(i);
```

Data may be transmitted to the receiving activity by adding it to the intent object before it is started via calls to the *putExtra()* method of the intent object. Data must be added in the form of key-value pairs. The following code extends the previous example to add String and integer values with the keys "myString" and "myInt"

respectively, to the intent:

```
Intent i = new Intent(this, ActivityB.class);
i.putExtra("myString", "This is a message for ActivityB");
i.putExtra("myInt", 100);

startActivity(i);
```

The target activity receives the data as part of a Bundle object which can be obtained via a call to *getIntent().getExtras()*. The *getIntent()* method of the Activity class returns the intent that started the activity, while the *getExtras()* method (of the Intent class) returns a Bundle object containing the data. For example, to extract the data values passed to ActivityB:

```
Bundle extras = getIntent().getExtras();
if (extras != null) {
    String myString = extras.getString("myString");
    int myInt = extras.getInt("myInt");
}
```

When using intents to launch other activities within the same application, those activities must be listed in the application manifest file. The following *AndroidManifest.xml* contents are correctly configured for an application containing activities named ActivityA and ActivityB:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.ebookfrenzy.intent1.intent1" >

    <application
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name" >
        <activity
            android:label="@string/app_name"
            android:name="com.ebookfrenzy.intent1.intent1.ActivityA" >
            <intent-filter>
              <action android:name="android.intent.action.MAIN" />
              <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity
            android:name="ActivityB"
            android:label="ActivityB" >
        </activity>
    </application>
</manifest>
```

53.3 Returning Data from an Activity

As the example in the previous section stands, while data is transferred to ActivityB, there is no way for data to be returned to the first activity (which we will call ActivityA). This can, however, be achieved by launching ActivityB as a *sub-activity* of ActivityA. An activity is started as a sub-activity by creating an ActivityResultLauncher instance. An ActivityResultLauncher instance is created by a call to the *registerForActivityResult()* method and is

passed a callback handler in the form of a lambda. This handler will be called and passed return data when the sub-activity returns. Once an ActivityResultLauncher instance has been created, it can be called with an intent parameter to launch the sub-activity. The code to create an ActivityResultLauncher instance typically reads as follows:

```
ActivityResultLauncher<Intent> startForResult = registerForActivityResult(
        new ActivityResultContracts.StartActivityForResult(),
        new ActivityResultCallback<ActivityResult>() {
            @Override
            public void onActivityResult(ActivityResult result) {
                if (result.getResultCode() == Activity.RESULT_OK) {
                    Intent data = result.getData();
                    // Code to handle returned data
                }
            }
        });
```

Once the launcher is ready, it can be called and passed the intent to be launched as follows:

```
Intent i = new Intent(this, ActivityB.class);
.
.
startForResult.launch(i);
```

To return data to the parent activity, the sub-activity must implement the *finish()* method, the purpose of which is to create a new intent object containing the data to be returned and then call the *setResult()* method of the enclosing activity, passing through a *result code* and the intent containing the return data. The result code is typically *RESULT_OK*, or *RESULT_CANCELED*, but it may also be a custom value subject to the developer's requirements. If a sub-activity crashes, the parent activity will receive a *RESULT_CANCELED* result code.

The following code, for example, illustrates the code for a typical sub-activity *finish()* method:

```
public void finish() {
       Intent data = new Intent();

       data.putExtra("returnString1", "Message to parent activity");
       setResult(RESULT_OK, data);
       super.finish();
}
```

53.4 Implicit Intents

Unlike explicit intents, which reference the class name of the activity to be launched, implicit intents identify the activity to be launched by specifying the action to be performed and the type of data to be handled by the receiving activity. For example, an action type of ACTION_VIEW accompanied by the URL of a web page in the form of a URI object will instruct the Android system to search for and, subsequently, launch a web browser-capable activity. The following implicit intent will, when executed on an Android device, result in the designated web page appearing in a web browser activity:

```
Intent intent = new Intent(Intent.ACTION_VIEW,
        Uri.parse("https://www.ebookfrenzy.com"));

startActivity(intent);
```

When an activity issues the above implicit intent, the Android system will search for activities on the device that have registered the ability to handle ACTION_VIEW requests on HTTP scheme data using a process referred to as *intent resolution*. Before the system launches an activity using an implicit intent, the user must either verify or enable that activity. If neither of these conditions has been met, the activity will not be launched by the intent. Before exploring these two options, we first need to talk about intent filters.

53.5 Using Intent Filters

Intent filters are the mechanism by which activities "advertise" supported actions and data handling capabilities to the Android intent resolution process. These declarations also include the settings required to perform the link verification process. The following *AndroidManifest.xml* file illustrates a configuration for an activity named *WebActivity* within an app named *MyWebView* with an appropriately configured intent filter:

```
<?xml version="1.0" encoding="utf-8"?>
.
.
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.MyWebView">
        <activity
            android:name="WebActivity"
            android:exported="true">
            <intent-filter android:autoVerify="true">
                <action android:name="android.intent.action.VIEW" />
                <category android:name="android.intent.category.BROWSABLE" />
                <category android:name="android.intent.category.DEFAULT" />
                <data android:scheme="https" />
                <data android:host="www.ebookfrenzy.com"/>
            </intent-filter>
        </activity>
    </application>
</manifest>
```

This manifest file configures the WebActivity activity to be launched in response to an implicit intent from another activity when the intent contains the *https://www.ebookfrenzy.com* URL. The following code, for example, would launch the WebActivity activity (assuming that the MyWebView app has been verified or enabled by the user as a support link):

```
Intent intent = new Intent(Intent.ACTION_VIEW,
        Uri.parse("https://www.ebookfrenzy.com"));

startActivity(intent);
```

53.6 Automatic Link Verification

Using a web link to launch an activity on an Android device is considered a potential security hazard. To minimize this risk, the link used to launch an intent must either be automatically verified or manually added as a

supported link on the device by the user. To enable automatic verification, the corresponding intent declaration in the target activity must set autoVerify to true as follows:

```
<intent-filter android:autoVerify="true">
.
.
</intent-filter>
```

Next, the link URL must be associated with the website on which the app link is based. This is achieved by creating a Digital Assets Link file named *assetlinks.json* and installing it within the website's *.well-known* folder.

A digital asset link file comprises a *relation* statement granting permission for a target app to be launched using the website's link URLs and a target statement declaring the companion app package name and SHA-256 certificate fingerprint for that project. A typical asset link file might, for example, read as follows:

```
[{
  "relation": ["delegate_permission/common.handle_all_urls"],
  "target": {
    "namespace": "android_app",
    "package_name": "com.ebookfrenzy.mywebview",
    "sha256_cert_fingerprints":
    ["<your certificate fingerprint here>"]
  }
}]
```

Note that you can either create this file manually or generate it using the online tool available at the following URL:

https://developers.google.com/digital-asset-links/tools/generator

When working with Android, the namespace value is always set to "android_app", while the package name corresponds to the app package to be launched by the intent. Finally, the certificate fingerprint is the hash code used to build the app. When you are testing an app, this will be the debug certificate contained within the *debug.keystore* file. On Windows systems, Android Studio stores this file at the following location:

```
\Users\<your user name>\.android\debug.keystore
```

On macOS and Linux systems, the file can be found at:

```
$HOME/.android/debug.keystore
```

Once you have located the file, the SHA 256 fingerprint can be obtained by running the following command in a terminal or command prompt window:

```
keytool -list -v -keystore <path to debug.keystore file here>
```

When prompted for a password, enter "android" after which output will appear, including the SHA 256 fingerprint:

```
Certificate fingerprints:
        SHA1: 11:E8:66:11:B6:94:3D:AA:7E:50:63:99:77:B8:6A:90:FF:B6:9C:6D
        SHA256: 7F:EE:E3:C8:38:41:C3:EA:11:56:83:94:2A:4C:D2:EA:A0:69:F8:96:D1:17
:77:02:46:EC:AD:6E:3C:64:A9:29
```

When you are ready to build your app's release version, you must ensure you add the release SHA 256 fingerprint to the asset file. Details on generating release keystore files are covered in the chapter entitled *"Creating, Testing, and Uploading an Android App Bundle"*. Once you have a release keystore file, run the above keytool command

to access the fingerprint.

Once you have placed the digital asset file in the correct location on the website, install the app on a device or emulator and wait 30 seconds for the link to be verified. To check the verification status, run the following at a command or terminal prompt:

```
adb shell pm get-app-links --user cur com.example.mywebview
```

The resulting output should include confirmation that the link has been verified:

```
com.example.mywebview:
    ID: 0e399bca-bf58-4cfc-8c7b-d1a6c3b065ec
    Signatures: [7F:EE:E3:C8:38:41:C3:EA:11:56:83:94:2A:4C:D2:EA:A0:69:F8:96:D1:1
7:77:02:46:EC:AD:6E:3C:64:A9:29]
    Domain verification state:
      www.ebookfrenzy.com: verified
    User 0:
      Verification link handling allowed: true
      Selection state:
        Disabled:
          www.ebookfrenzy.com
```

You can also check the status from within the Settings app on the device or emulator using the following steps:

1. Launch the Settings app.
2. Select *Apps* from the main list.
3. Locate and select your app from the list of installed apps.
4. On the settings page for your app, choose the *Open by Default* option.

Choose the Open by Default option on your app's settings page.

Once displayed, the page should indicate that a link has been verified, as shown in Figure 53-1:

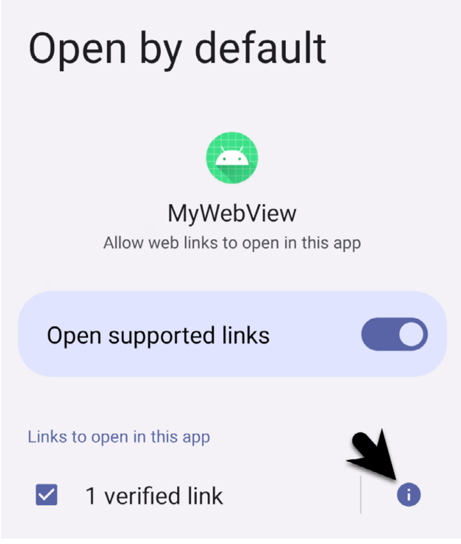


Figure 53-1

To review which links have been verified, tap on the info button indicated by the arrow in the above figure to display the following panel:

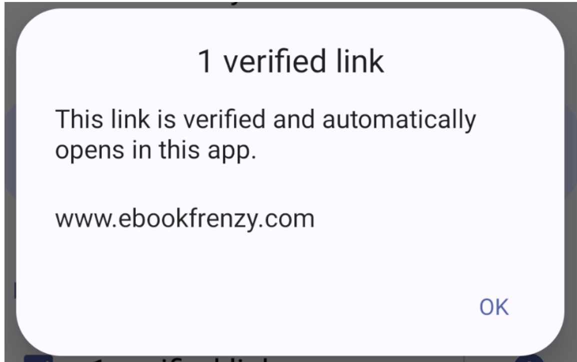


Figure 53-2

The *assetlinks.json* file can contain multiple digital asset links, allowing a single website to be associated with more than one app. If you cannot use auto link verification, add code to your app to prompt the user to enable the link manually.

53.7 Manually Enabling Links

Where it is not possible to auto-verify links using the steps outlined above, the only option is to request that the user manually enable app links. This involves launching the Open by Default screen of the Settings app for the target app where the user can enable the link.

Since the sudden appearance of the Open by Default screen may be confusing to the average user, it is recommended that an explanatory dialog be displayed before launching the Settings app.

To provide the user with the option to enable a link manually, the following code needs to be executed before attempting to launch the intent:

```
.
.
// Code here to display a dialog explaining that the link needs to be enabled
.
.
Intent intent = new Intent(
        Settings.ACTION_APP_OPEN_BY_DEFAULT_SETTINGS,
        Uri.parse("package:com.ebookfrenzy.mywebview"));

startActivity(intent);
```

The above example code will display the Open by Default settings screen for our target MyWebView app, where the user can click on the Add Link button:

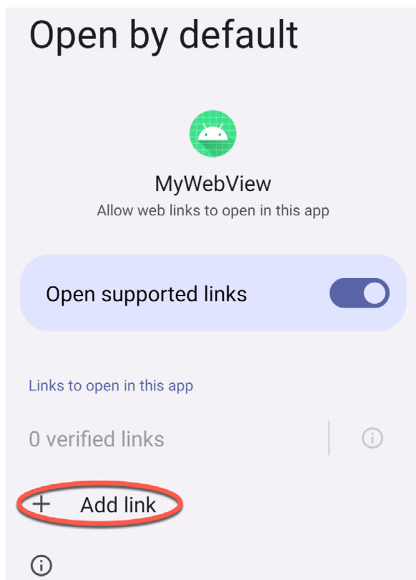


Figure 53-3

Once clicked, a dialog will appear initialized with the link passed in the intent. This can be enabled by setting the checkbox as shown in Figure 53-4:

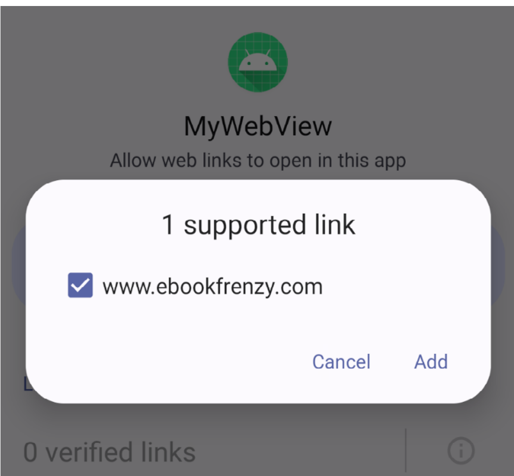


Figure 53-4

53.8 Checking Intent Availability

It is generally unwise to assume that an activity will be available for a particular intent, especially since the absence of a matching action typically results in the application crashing. Fortunately, it is possible to identify the availability of an activity for a specific intent before it is sent to the runtime system. The following method can be used to identify the availability of an activity for a specified intent action type:

```
public static boolean isIntentAvailable(Context context, String action) {
    final PackageManager packageManager = context.getPackageManager();
    final Intent intent = new Intent(action);
    List<ResolveInfo> list =
```

```
            packageManager.queryIntentActivities(intent,
                    PackageManager.MATCH_DEFAULT_ONLY);
    return list.size() > 0;
}
```

53.9 Summary

Intents are the messaging mechanism by which one Android activity can launch another. An explicit intent references a specific activity to be launched by referencing the receiving activity by class name. Explicit intents are typically, though not exclusively, used when launching activities within the same application. An implicit intent specifies the action to be performed and the type of data to be handled and lets the Android runtime find a matching activity to launch. Implicit intents are generally used when launching activities that reside in different applications.

When working with implicit intents, security restrictions require the user to automatically verify or manually enable the app containing the intent activity target before launching the intent. Automatic verification involves the placement of a Digital Assets Link file on the website corresponding to the link URL.

An activity can send data to the receiving activity by bundling data into the intent object as key-value pairs. Data can only be returned from an activity if it is started as a sub-activity of the sending activity.

Activities advertise capabilities to the Android intent resolution process by specifying intent filters in the application manifest file. Both sending and receiving activities must also request appropriate permissions to perform tasks such as accessing the device contact database or the internet.

Having covered the theory of intents, the next few chapters will work through creating some examples in Android Studio that put both explicit and implicit intents into action.

Chapter 73

73. Android Audio Recording and Playback using MediaPlayer and MediaRecorder

This chapter will provide an overview of the MediaRecorder class and explain how this class can be used to record audio or video. The use of the MediaPlayer class to play back audio will also be covered. Having covered the basics, an example application will be created to demonstrate these techniques. In addition to looking at audio and video handling, this chapter will also touch on saving files to the SD card.

73.1 Playing Audio

In terms of audio playback, most implementations of Android support AAC LC/LTP, HE-AACv1 (AAC+), HE-AACv2 (enhanced AAC+), AMR-NB, AMR-WB, MP3, MIDI, Ogg Vorbis, and PCM/WAVE formats.

Audio playback can be performed using either the MediaPlayer or the AudioTrack classes. AudioTrack is a more advanced option that uses streaming audio buffers and provides greater control over the audio. The MediaPlayer class, on the other hand, provides an easier programming interface for implementing audio playback and will meet the needs of most audio requirements.

The MediaPlayer class has associated with it a range of methods that can be called by an application to perform certain tasks. A subset of some of the key methods of this class is as follows:

- **create()** – Called to create a new instance of the class, passing through the Uri of the audio to be played.
- **setDataSource()** – Sets the source from which the audio is to play.
- **prepare()** – Instructs the player to prepare to begin playback.
- **start()** – Starts the playback.
- **pause()** – Pauses the playback. Playback may be resumed via a call to the *resume()* method.
- **stop()** – Stops playback.
- **setVolume()** – Takes two floating-point arguments specifying the playback volume for the left and right channels.
- **resume()** – Resumes a previously paused playback session.
- **reset()** – Resets the state of the media player instance. Essentially sets the instance back to the uninitialized state. At a minimum, a reset player will need to have the data source set again, and the *prepare()* method called.
- **release()** – To be called when the player instance is no longer needed. This method ensures that any resources held by the player are released.

In a typical implementation, an application will instantiate an instance of the MediaPlayer class, set the source

of the audio to be played, and then call *prepare()* followed by *start()*. For example:

```
MediaPlayer mediaPlayer = new MediaPlayer();

mediaPlayer.setDataSource("https://www.yourcompany.com/myaudio.mp3");
mediaPlayer.prepare();
mediaPlayer.start();
```

73.2 Recording Audio and Video using the MediaRecorder Class

As with audio playback, recording can be performed using several different techniques. One option is to use the MediaRecorder class, which, as with the MediaPlayer class, provides several methods that are used to record audio:

- **setAudioSource()** – Specifies the audio source to be recorded (typically, this will be MediaRecorder.AudioSource.MIC for the device microphone).
- **setVideoSource()** – Specifies the source of the video to be recorded (for example MediaRecorder.VideoSource.CAMERA).
- **setOutputFormat()** – Specifies the format into which the recorded audio or video is to be stored (for example MediaRecorder.OutputFormat.AAC_ADTS).
- **setAudioEncoder()** – Specifies the audio encoder for the recorded audio (for example MediaRecorder.AudioEncoder.AAC).
- **setOutputFile()** – Configures the path to the file into which the recorded audio or video will be stored.
- **prepare()** – Prepares the MediaRecorder instance to begin recording.
- **start()** - Begins the recording process.
- **stop()** – Stops the recording process. Once a recorder has been stopped, it must be completely reconfigured and prepared before restarting.
- **reset()** – Resets the recorder. The instance will need to be completely reconfigured and prepared before being restarted.
- **release()** – Should be called when the recorder instance is no longer needed. This method ensures that all resources held by the instance are released.

A typical implementation using this class will set the source, output, encoding format, and output file. Calls will then be made to the *prepare()* and *start()* methods. The *stop()* method will then be called when the recording ends, followed by the *reset()* method. When the application no longer needs the recorder instance, a call to the *release()* method is recommended:

```
MediaRecorder mediaRecorder = new MediaRecorder();

mediaRecorder.setAudioSource(MediaRecorder.AudioSource.MIC);
mediaRecorder.setOutputFormat(MediaRecorder.OutputFormat.AAC_ADTS);
mediaRecorder.setAudioEncoder(MediaRecorder.AudioEncoder.AAC);
mediaRecorder.setOutputFile(audioFilePath);

mediaRecorder.prepare();
mediaRecorder.start();
```

```
.
.
mediaRecorder.stop();
mediaRecorder.reset();
mediaRecorder.release();
```

To record audio, the manifest file for the application must include the android.permission.RECORD_AUDIO permission:

```
<uses-permission android:name="android.permission.RECORD_AUDIO" />
```

As outlined in the chapter entitled *"Making Runtime Permission Requests in Android"*, access to the microphone falls into the category of dangerous permissions. To support Android 6, therefore, a specific request for microphone access must also be made when the application launches, the steps for which will be covered later in this chapter.

73.3 About the Example Project

The remainder of this chapter will create an example application to demonstrate the use of the MediaPlayer and MediaRecorder classes to implement the recording and playback of audio on an Android device.

When developing applications that use specific hardware features, the microphone being a case in point, it is important to check the feature's availability before attempting to access it in the application code. The application created in this chapter will, therefore, also include code to detect the presence of a microphone on the device.

Once completed, this application will provide a straightforward interface allowing the user to record and play audio. The recorded audio will be stored within an audio file on the device. That being the case, this tutorial will also briefly explore the mechanism for using SD Card storage.

73.4 Creating the AudioApp Project

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

Enter *AudioApp* into the Name field and specify *com.ebookfrenzy.audioapp* as the package name. Before clicking on the Finish button, change the Minimum API level setting to API 31: Android 12.0 and the Language menu to Java. Add view binding support to the project using the steps outlined in section *11.8 Migrating a Project to View Binding*.

73.5 Designing the User Interface

Once the new project has been created, select the *activity_main.xml* file from the Project tool window, and with the Layout Editor tool in Design mode, select the "Hello World!" TextView and delete it from the layout.

Drag and drop three Button views onto the layout. The positioning of the buttons is not paramount to this example, though Figure 73-1 shows a suggested layout using a vertical chain.

Configure the buttons to display string resources that read *Play, Record,* and *Stop* and give them view IDs of *playButton*, *recordButton*, and *stopButton*, respectively.

Select the Play button and, within the Attributes panel, configure the *onClick* property to call a method named *playAudio* when selected by the user. Repeat these steps to configure the remaining buttons to call methods named *recordAudio* and *stopAudio,* respectively.

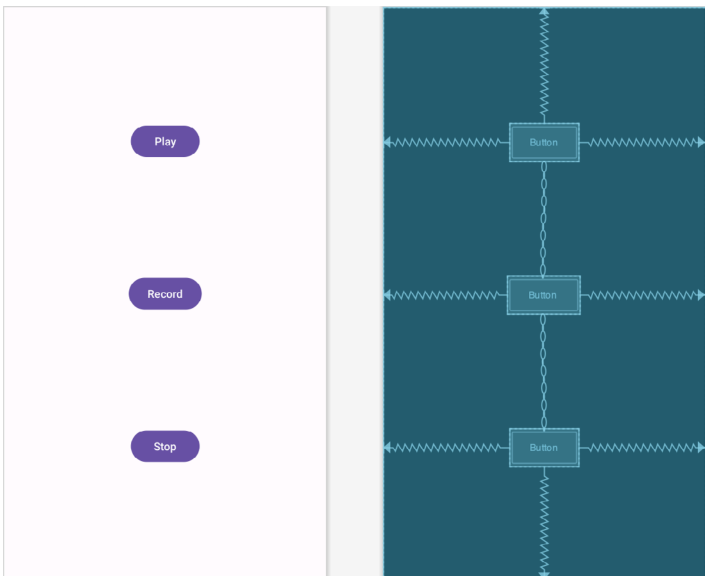


Figure 73-1

73.6 Checking for Microphone Availability

Attempting to record audio on a device without a microphone will cause the Android system to throw an exception. It is vital, therefore, that the code checks for the presence of a microphone before making such an attempt. There are several ways of doing this, including checking for the physical presence of the device. An easier approach that is more likely to work on different Android devices is to ask the Android system if it has a package installed for a particular *feature*. This involves creating an instance of the Android PackageManager class and then calling the object's *hasSystemFeature()* method. *PackageManager.FEATURE_MICROPHONE* is the feature of interest in this case.

For this example, we will create a method named *hasMicrophone()* that may be called upon to check for the presence of a microphone. Within the Project tool window, locate and double-click on the *MainActivity.java* file and modify it to add this method:

```
package com.ebookfrenzy.audioapp;
.
.
import android.content.pm.PackageManager;
.
.
public class MainActivity extends AppCompatActivity {
.
.
    protected boolean hasMicrophone() {
        PackageManager pmanager = this.getPackageManager();
        return pmanager.hasSystemFeature(
                PackageManager.FEATURE_MICROPHONE);
    }
}
```

73.7 Initializing the Activity

The next step is to modify the activity to perform several initialization tasks. Remaining within the *MainActivity.java* file, modify the code as follows:

```
package com.ebookfrenzy.audioapp;

import java.io.File;
import java.io.IOException;

import androidx.annotation.NonNull;
import android.media.MediaRecorder;
import android.os.Environment;
import android.media.MediaPlayer;
.
.
public class MainActivity extends AppCompatActivity {

    private ActivityMainBinding binding;
    private static MediaRecorder mediaRecorder;
    private static MediaPlayer mediaPlayer;

    private static String audioFilePath;
    private boolean isRecording = false;

   @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        binding = ActivityMainBinding.inflate(getLayoutInflater());
        View view = binding.getRoot();
        setContentView(view);
        audioSetup();
    }

    private void audioSetup()
    {
        if (!hasMicrophone())
        {
            binding.stopButton.setEnabled(false);
            binding.playButton.setEnabled(false);
            binding.recordButton.setEnabled(false);
        } else {
            binding.playButton.setEnabled(false);
            binding.stopButton.setEnabled(false);
        }

        File audioFile = new File(this.getFilesDir(), "myaudio.3gp");
```

```
        audioFilePath = audioFile.getAbsolutePath();
    }
.
.
}
```

The added code calls *hasMicrophone()* method to ascertain whether the device includes a microphone. If it does not, all the buttons are disabled; otherwise, only the Stop and Play buttons are disabled.

The next line of code needs a little more explanation:

```
File audioFile = new File(this.getFilesDir(), "myaudio.3gp");
audioFilePath = audioFile.getAbsolutePath();
```

This code creates a new file named *myaudio.3gp* within the app's internal storage to store the audio recording.

73.8 Implementing the recordAudio() Method

The *recordAudio()* method will be called when the user touches the Record button. This method will need to turn the appropriate buttons on and off and configure the MediaRecorder instance with information about the source of the audio, the output format and encoding, and the file's location into which the audio is to be stored. Finally, the *prepare()* and *start()* methods of the MediaRecorder object will need to be called. Combined, these requirements result in the following method implementation in the *MainActivity.java* file:

```
public void recordAudio (View view)
{
   isRecording = true;
   binding.stopButton.setEnabled(true);
   binding.playButton.setEnabled(false);
   binding.recordButton.setEnabled(false);

   try {
     mediaRecorder = new MediaRecorder(this);
     mediaRecorder.setAudioSource(MediaRecorder.AudioSource.MIC);
     mediaRecorder.setOutputFormat(
          MediaRecorder.OutputFormat.THREE_GPP);
     mediaRecorder.setOutputFile(audioFilePath);
     mediaRecorder.setAudioEncoder(MediaRecorder.AudioEncoder.AMR_NB);
     mediaRecorder.prepare();
   } catch (Exception e) {
          e.printStackTrace();
   }
   mediaRecorder.start();
}
```

73.9 Implementing the stopAudio() Method

The *stopAudio()* method enables the Play button, turning off the Stop button, and then stopping and resetting the MediaRecorder instance. The code to achieve this reads as outlined in the following listing and should be added to the *MainActivity.java* file:

```
public void stopAudio (View view)
{
```

```
        binding.stopButton.setEnabled(false);
        binding.playButton.setEnabled(true);

        if (isRecording)
        {
           binding.recordButton.setEnabled(false);
           mediaRecorder.stop();
           mediaRecorder.release();
           mediaRecorder = null;
           isRecording = false;
        } else {
           mediaPlayer.release();
           mediaPlayer = null;
           binding.recordButton.setEnabled(true);
        }
}
```

73.10 Implementing the playAudio() method

The *playAudio()* method will create a new MediaPlayer instance, assign the audio file located on the SD card as the data source and then prepare and start the playback:

```
public void playAudio (View view) throws IOException
{
   binding.playButton.setEnabled(false);
   binding.recordButton.setEnabled(false);
   binding.stopButton.setEnabled(true);

   mediaPlayer = new MediaPlayer();
   mediaPlayer.setDataSource(audioFilePath);
   mediaPlayer.prepare();
   mediaPlayer.start();
}
```

73.11 Configuring and Requesting Permissions

Before testing the application, the appropriate permissions must be requested within the manifest file for the application. Specifically, the application will require permission to access the microphone. Within the Project tool window, locate and double-click on the *AndroidManifest.xml* file to load it into the editor and modify the XML to add the permission tags:

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

    <uses-permission android:name="android.permission.RECORD_AUDIO" />

    <application
.
.
```

The above steps will be adequate to ensure that the user enables microphone access permission when the app is installed on devices running versions of Android predating Android 6.0. Microphone access is categorized in Android as being a dangerous permission because it allows the app to compromise the user's privacy. For the example app to function on Android 6 or later devices, code needs to be added to request permission at app runtime.

Edit the *MainActivity.java* file and begin by adding some additional import directives and a constant to act as request identification codes for the permissions being requested:

```
.
.
import androidx.core.app.ActivityCompat;
import androidx.core.content.ContextCompat;
import android.widget.Toast;
import android.Manifest;
.
.
public class MainActivity extends AppCompatActivity {

    private static final int RECORD_REQUEST_CODE = 101;
.
.
```

Next, a method needs to be added to the class, the purpose of which is to take as arguments the permission to be requested and the corresponding request identification code. Remaining with the *MainActivity.java* class file, implement this method as follows:

```
protected void requestPermission(String permissionType, int requestCode) {
    int permission = ContextCompat.checkSelfPermission(this,
                permissionType);

    if (permission != PackageManager.PERMISSION_GRANTED) {
        ActivityCompat.requestPermissions(this,
                new String[]{permissionType}, requestCode
        );
    }
}
```

Using the steps outlined in the *"Making Runtime Permission Requests in Android"* chapter of this book, the above method verifies that the specified permission has not already been granted before making the request, passing through the identification code as an argument.

When the request has been handled, the *onRequestPermissionsResult()* method will be called on the activity, passing through the identification code and the request results. The next step, therefore, is to implement this method within the *MainActivity.java* file as follows:

```
@Override
public void onRequestPermissionsResult(int requestCode,
                @NonNull String[] permissions, @NonNull int[] grantResults) {
    super.onRequestPermissionsResult(requestCode, permissions, grantResults);
```

```
    if (requestCode == RECORD_REQUEST_CODE) {
        if (grantResults.length == 0
                || grantResults[0] !=
                PackageManager.PERMISSION_GRANTED) {

            binding.recordButton.setEnabled(false);

            Toast.makeText(this,
                    "Record permission required",
                    Toast.LENGTH_LONG).show();
        }
    }
}
```

The above code checks the request identifier code to identify which permission request has returned before checking whether or not the corresponding permission was granted. If permission is denied, a message is displayed to the user indicating that the app will not function and the record button is disabled.

Before testing the app, all that remains is to call the newly added *requestPermission()* method for microphone access when the app launches. Remaining in the *MainActivity.java* file, modify the *audioSetup()* method as follows:

```
private void audioSetup(){
    binding.recordButton = findViewById(R.id.recordButton);
    binding.playButton = findViewById(R.id.playButton);
    binding.stopButton = findViewById(R.id.stopButton);

    if (!hasMicrophone())
    {
        stopButton.setEnabled(false);
        playButton.setEnabled(false);
        recordButton.setEnabled(false);
    } else {
        playButton.setEnabled(false);
        stopButton.setEnabled(false);
    }

    File audioFile = new File(this.getFilesDir(), "myaudio.3gp");
    audioFilePath = audioFile.getAbsolutePath();

    requestPermission(Manifest.permission.RECORD_AUDIO,
            RECORD_REQUEST_CODE);
}
```

73.12 Testing the Application

Compile and run the application on an Android device containing a microphone, allow microphone access, and tap the Record button. After recording, touch Stop followed by Play. At this point, the recorded audio should play back through the device speakers.

73.13 Summary

The Android SDK provides several mechanisms to implement audio recording and playback. This chapter has looked at two of these: the MediaPlayer and MediaRecorder classes. Having covered the theory of using these techniques, this chapter worked through creating an example application designed to record and then play back audio. While working with audio in Android, this chapter also looked at the steps involved in ensuring that the device on which the application is running has a microphone before attempting to record audio.

```
.
.
}
```

This method obtains a reference to the Print Manager service running on the device before creating a new String object to serve as the job name for the print task. Finally, the *print()* method of the Print Manager is called to start the print job, passing through the job name and an instance of our custom print document adapter class.

77.9 Testing the Application

Compile and run the application on an Android device or emulator. When the application has loaded, touch the "Print Document" button to initiate the print job and select a suitable target for the output (the Save to PDF option is useful for avoiding wasting paper and printer ink).

Check the printed output, which should consist of 4 pages, including text and graphics. Figure 77-3, for example, shows the four pages of the document viewed as a PDF file ready to be saved on the device.

Experiment with other print configuration options, such as changing the paper size, orientation, and page settings within the print panel. The printed output should reflect each setting change, indicating that the custom print document adapter functions correctly.

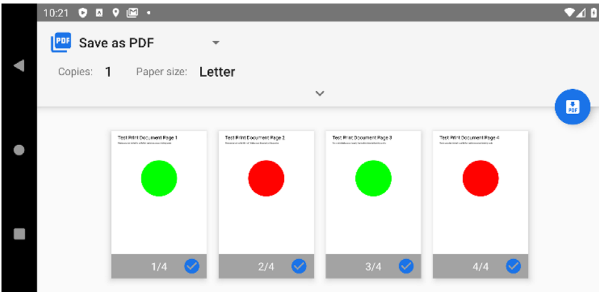


Figure 77-3

77.10 Summary

Although more complex to implement than the Android Printing framework HTML and image printing options, custom document printing provides considerable flexibility in printing complex content within an Android application. Most of the work in implementing custom document printing involves the creation of a custom Print Adapter class that not only draws the content on the document pages but also responds correctly as the user changes print settings, such as the page size and range of pages to be printed.

Chapter 78

78. An Introduction to Android App Links

As technology evolves, the traditional distinction between web and mobile content is beginning to blur. One area where this is particularly true is the growing popularity of progressive web apps, where web apps look and behave much like traditional mobile apps.

Another trend involves making the content within mobile apps discoverable through web searches and via URL links. In the context of Android app development, the App Links feature is designed to make it easier for users to discover and access content stored within an Android app, even if the user does not have the app installed.

78.1 An Overview of Android App Links

An app link is a standard HTTP URL that is an easy way to link directly to a particular place in your app from an external source such as a website or app. App links (also called *deep links*) are used primarily to encourage users to engage with an app and to allow users to share app content.

App link implementation is a multi-step process that involves the addition of intent filters to the project manifest, implementing link handling code within the associated app activities, and the use of digital asset links files to associate app and web-based content.

These steps can be performed manually by making changes within the project or automatically using the Android Studio App Links Assistant.

These steps can be performed manually by making project changes or automatically using the Android Studio App Links Assistant.

The remainder of this chapter will outline app links implementation in terms of the changes that must be made to a project. The next chapter (*"An Android Studio App Links Tutorial"*) will demonstrate the use of the App Links Assistant to achieve the same results.

78.2 App Link Intent Filters

An app link URL needs to be mapped to a specific activity within an app project. This is achieved by adding intent filters to the project's *AndroidManifest.xml* file designed to launch an activity in response to an *android.intent.action.VIEW* action. The intent filters are declared within the element for the activity to be launched and must contain the data outlining the scheme, host, and path of the app link URL. The following manifest fragment, for example, declares an intent filter to launch an activity named MyActivity when an app link matching *http://www.example.com/welcome* is detected:

```
<activity android:name="com.ebookfrenzy.myapp.MyActivity">

    <intent-filter android:autoVerify="true">
        <action android:name="android.intent.action.VIEW" />
        <category android:name="android.intent.category.DEFAULT" />
        <category android:name="android.intent.category.BROWSABLE" />
```

```
        <data
            android:scheme="http"
            android:host="www.example.com"
            android:pathPrefix="/welcome" />
    </intent-filter>
</activity>
```

The order in which ambiguous intent filters are handled can be specified using the *order* property of the intent filter tag as follows:

```
<application>
    <activity android:name=" com.ebookfrenzy.myapp.MyActivity">
        <intent-filter android:autoVerify="true" android:order="1">
.
.
```

The intent filter will cause the app link to launch the correct activity, but code must still be added to the target activity to handle the intent appropriately.

78.3 Handling App Link Intents

In most cases, the launched activity will need to gain access to the app link URL and take specific action based on how the URL is structured. Continuing from the above example, the activity will likely display different content when launched via a URL containing a path of */welcome/newuser* than one with the path set to */welcome/existinguser*.

When the link launches the activity, it is passed an intent object containing data about the action which launched the activity, including a Uri object containing the app link URL. Within the initialization stages of the activity, code can be added to extract this data as follows:

```
Intent appLinkIntent = getIntent();
String appLinkAction = appLinkIntent.getAction();
Uri appLinkData = appLinkIntent.getData();
```

Having obtained the Uri for the app link, the various components that make up the URL path can be used to decide the actions to perform within the activity. In the following code example, the last component of the URL is used to identify whether content should be displayed for a new or existing user:

```
String userType = appLinkData.getLastPathSegment();

if (userType.equals("newuser")) {
    // display new user content
} else {
    // display existing user content
}
```

78.4 Associating the App with a Website

Before an app link will work, an app link URL must be associated with the website on which the app link is based. This is achieved by creating a Digital Asset Links file named *assetlinks.json* and installing it within the website's *.well-known* folder. Note that digital asset linking is only possible for websites that are HTTPS based.

A digital asset links file comprises a *relation* statement granting permission for a target app to be launched using the website's link URLs and a target statement declaring the companion app package name and SHA-256 certificate fingerprint for that project. A typical asset link file might, for example, read as follows:

```
[{
    "relation": ["delegate_permission/common.handle_all_urls"],
    "target" : { "namespace": "android_app",
      "package_name": "<app package name here>",
                 "sha256_cert_fingerprints": ["<app certificate here>"] }
}]
```

The *assetlinks.json* file can contain multiple digital asset links, allowing a single website to be associated with more than one companion app.

78.5 Summary

Android App Links allow app activities to be launched via URL links from external websites and other apps. App links are implemented using intent filters within the project manifest file and intent handling code within the launched activity. Using a Digital Asset Links file, it is also possible to associate the domain name used in an app link with the corresponding website. Once the association has been established, Android no longer needs to ask the user to select the target app when an app link is used.

Chapter 81

81. Creating, Testing, and Uploading an Android App Bundle

Once the development work on an Android application is complete and tested on a wide range of Android devices, the next step is to prepare the application for submission to Google Play. Before submission can take place, however, the application must be packaged for release and signed with a private key. This chapter will work through obtaining a private key, preparing the Android App Bundle for the project, and uploading it to Google Play.

81.1 The Release Preparation Process

Up until this point in the book, we have been building application projects in a mode suitable for testing and debugging. On the other hand, building an application package for release to customers via Google Play requires additional steps. The first requirement is to compile the application in release mode instead of *debug mode*. Secondly, the application must be signed with a private key that uniquely identifies you as the application's developer. Finally, the application must be packaged into an *Android App Bundle*.

While these tasks can be performed outside of the Android Studio environment, the procedures can more easily be performed using the Android Studio build mechanism, as outlined in the remainder of this chapter. First, however, it is important to understand more about Android App Bundles.

81.2 Android App Bundles

When a user installs an app from Google Play, the app is downloaded in the form of an APK file. This file contains everything needed to install and run the app on the user's device. Before the introduction of Android Studio 3.2, the developer would generate one or more APK files using Android Studio and upload them to Google Play. Supporting multiple device types, screen sizes, and locales would require creating and uploading multiple APK files customized for each target device and locale or generating a large *universal APK* containing all of the different configuration resources and platform binaries within a single package.

Creating multiple APK files involved a significant amount of work that had to be repeated each time the app was updated, imposing a considerable time overhead on the app release process.

Creating multiple APK files involved a significant amount of work that had to be repeated each time the app needed to be updated imposing a considerable time overhead to the app release process.

The universal APK option, while less of a burden to the developer, caused an entirely unexpected problem. By analyzing app installation metrics, Google discovered that the larger an installation APK file becomes (resulting in longer download times and increased storage use), the fewer conversions the app receives. The conversion rate is calculated as a percentage of the users who completed the installation of an app after viewing that app on Google Play. Google estimates that the conversion rate for an app drops by 1% for each 6MB increase in APK file size.

Android App Bundles solve these problems by allowing the developer to create a single package from within Android Studio and have custom APK files automatically generated by Google Play for each individual supported configuration (a concept called *Dynamic Delivery*).

An Android App Bundle is a ZIP file containing all the files necessary to build APK files for the devices and locales for which support has been provided within the app project. The project might, for example, include resources and images for different screen sizes. When a user installs the app, Google Play receives information about the device, including the display, processor architecture, and locale. Using this information, the appropriate pre-generated APK files are transferred onto the user's device.

An additional benefit of Dynamic Delivery is the ability to split an app into multiple modules, referred to as *dynamic feature modules,* where each module contains the code and resources for a particular area of functionality within the app. Each dynamic feature module is contained within a separate APK file from the base module and is downloaded to the device only when the user requires that feature. Dynamic Delivery and app bundles also allow for the creation of *instant dynamic feature modules* which can be run instantly on a device without the need to install an entire app.

Although it is still possible to generate APK files from Android Studio, app bundles are now the recommended way to upload apps to Google Play.

81.3 Register for a Google Play Developer Console Account

The first step in the application submission process is to create a Google Play Developer Console account. To do so, navigate to *https://play.google.com/apps/publish/signup/* and follow the instructions to complete the registration process. Note that there is a one-time $25 fee to register. Once an application goes on sale, Google will keep 30% of all revenues associated with the application. After creating the account, the developer console can be accessed at *https://play.google.com/console.*

The next step is to gather together information about the application. To bring your application to market, the following information will be required:

- **Title** – The title of the application.
- **Short Description** - Up to 80 words describing the application.
- **Full Description** – Up to 4000 words describing the application.
- **Screenshots** – Up to 8 screenshots of your application running (a minimum of two is required). Google recommends submitting screenshots of the application running on a 7" or 10" tablet.
- **Language** – The language of the application (the default is US English).
- **Promotional Text** – The text that will be used when your application appears in special promotional features within the Google Play environment.
- **Application Type** – Whether your application is considered a *game* or an *application*.
- **Category** – The category that best describes your application (for example, finance, health and fitness, education, sports, etc.).
- **Locations** – The geographical locations into which you wish your application to be made available for purchase.
- **Contact Details** – Methods by which users may contact you for support relating to the application. Options include web, email, and phone.
- **Pricing & Distribution** – Information about the price of the application and the geographical locations where it is to be marketed and sold.

Having collected the above information, click the *Create app* button within the Google Play Console to begin

the creation process.

81.4 Configuring the App in the Console

When the *Create app* button is first clicked, the app details and declarations screen will appear as shown in Figure 81-1 below:

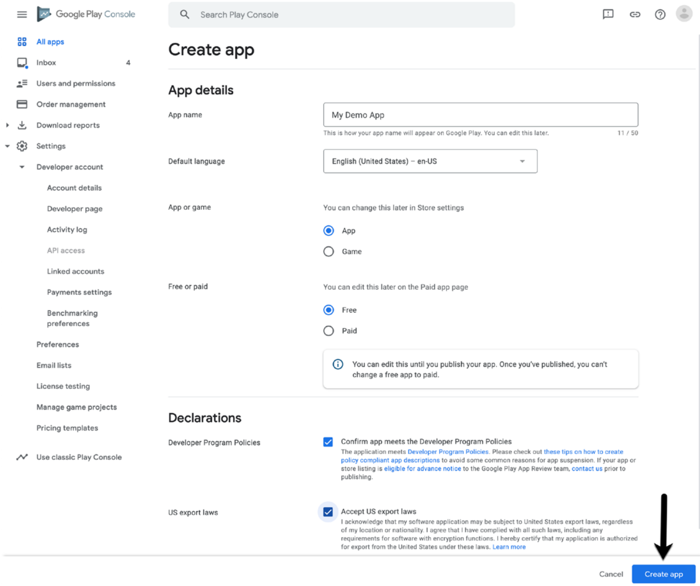


Figure 81-1

Once the app entry has been fully configured, click on the *Create app* button (highlighted in the above figure) to add the app and display the dashboard screen. Within the dashboard, locate the *Initial setup* section and unfold the list of steps to configure the app store listing:

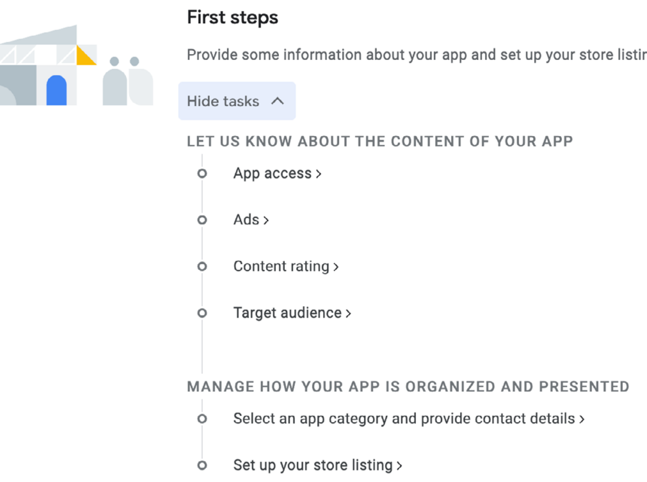


Figure 81-2

Work through the list of links and provide the requested information for your app, making sure to save the changes at each step.

81.5 Enabling Google Play App Signing

Until recently, Google Play uploads were signed with a release app signing key from within Android Studio and then uploaded to the Google Play console. While this option is still available, the recommended way to upload files is to use a process called *Google Play App Signing*. For a newly created app, this involves opting into Google Play App Signing and generating an *upload key* to sign the app bundle file within Android Studio. When the app bundle file generated by Android Studio is uploaded, the Google Play console removes the upload key and signs the file with an app signing key stored securely within the Google Play servers. For existing apps, some additional steps are required to enable Google Play Signing and will be covered at the end of this chapter.

Within the Google Play console, select the newly added app entry from the All Apps screen (accessed via the option located at the top of the left-hand navigation panel), unfold the Setup section (Marked A in Figure 81-3), and select the App Signing option (B).

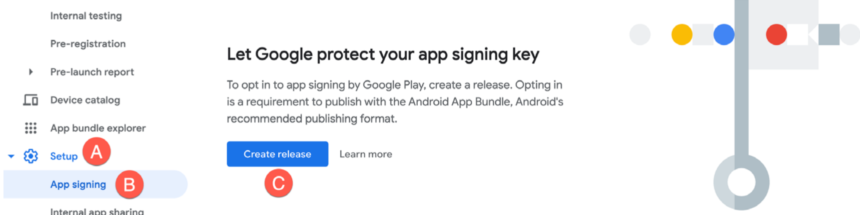


Figure 81-3

Opt into Google Play app signing by clicking on the *Create release* button (C). The console is now ready to create the first release of your app for testing. Before doing so, however, the next step is to generate the *upload* key from within Android Studio. This is performed as part of the process of generating a signed app bundle. Leave the current Google Play Console screen loaded into the browser, as we will be returning to this later in the chapter.

81.6 Creating a Keystore File

To create a keystore file, select the Android Studio *Build -> Generate Signed Bundle / APK…* menu option to display the Generate Signed Bundle or APK Wizard dialog as shown in Figure 81-4:

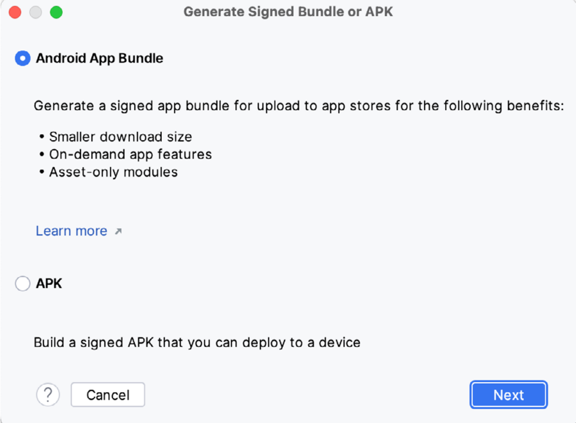


Figure 81-4

Verify that the *Android App Bundle* option is selected before clicking the *Next* button.

If you have an existing release keystore file, click on the *Choose existing…* button on the next screen and navigate to and select the file. If you have not created a keystore file, click the *Create new…* button to display the *New Key Store* dialog (Figure 81-5). Click on the button to the right of the Key store path field and navigate to a suitable location on your file system, enter a name for the keystore file (for example, *release.keystore.jks*) and click the OK button.

The New Key Store dialog is divided into two sections. The top section relates to the keystore file. In this section, enter a strong password to protect the keystore file into both the *Password* and *Confirm* fields. The lower section of the dialog relates to the upload key that will be stored in the key store file.

Figure 81-5

Within the *Key* section of the New Key Store dialog, enter the following details:

- An alias by which the key will be referenced. This can be any sequence of characters, though the system uses only the first eight.
- A suitably strong password to protect the key.
- The number of years for which the key is to be valid (Google recommends a duration in excess of 25 years).

In addition, information must be provided for at least one of the remaining fields (for example, your first and last name or organization name).

Once the information has been entered, click the OK button to create the bundle.

81.7 Creating the Android App Bundle

The next step is instructing Android Studio to build the application app bundle file in release mode and sign it with the newly created private key. At this point, the *Generate Signed Bundle or APK* dialog should still be displayed with the keystore path, passwords, and key alias fields populated with information:

Generate Signed Bundle or APK
Module BiometricDemo.app
Key store path /users/neilsmyth/Desktop/MyKey
Create new... Choose existing...
Key store password •••••••••
Key alias key0
Key password ••••••
Remember passwords
Export encrypted key for enrolling published apps in Google Play App Signing
Encrypted key export path /Users/neilsmyth/Desktop
Cancel Previous Next

Figure 81-6

Ensure that the Export Encrypted Key option is enabled and, assuming the other settings are correct, click on the Next button to proceed to the app bundle generation screen (Figure 81-7). Within this screen, review the *Destination Folder:* setting to verify that the location into which the app bundle file will be generated is acceptable. If another location is preferred, click on the button to the right of the text field and navigate to the desired file system location.

Figure 81-7

Click the *Finish* button and wait for the Gradle system to build the app bundle. Once the build is complete, a dialog will appear providing the option to open the folder containing the app bundle file in an explorer window or to load the file into the APK Analyzer:

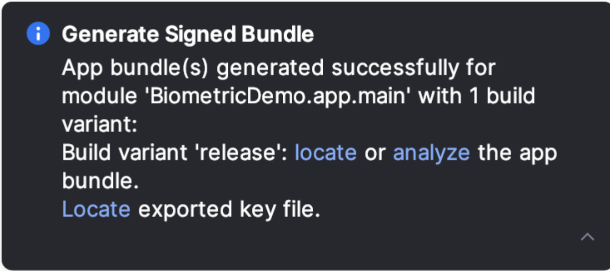


Figure 81-8

At this point, the application is ready to be submitted to Google Play. Click on the locate link to open a filesystem browser window. The file should be named *bundle.aab* and located in the project folder's *app/release* sub-directory unless another location is specified.

The private key generated as part of this process should be used when signing and releasing future applications and, as such, should be kept in a safe place and securely backed up.

81.8 Generating Test APK Files

An optional step at this stage is to generate APK files from the app bundle and install and run them on devices or emulator sessions. Google provides a command-line tool called *bundletool* designed specifically for this purpose which can be downloaded from the following URL:

https://github.com/google/bundletool/releases

At the time of writing, bundletool is provided as a .jar file which can be executed from the command line as follows (noting that the version number may have changed since this book was published):

```
java -jar bundletool-all-0.9.0.jar
```

Running the above command will list all of the options available within the tool. To generate the APK files from the app bundle, the *build-apks* option is used. The files will also need to be signed to generate APK files that can be installed onto a device or emulator. To achieve this, include the *--ks* option specifying the path of the keystore file created earlier in the chapter and the *--ks-key-alias* option specifying the alias provided when the key was generated.

Finally, the *--output* flag must be used to specify the path of the file (called the APK Set) into which the APK files will be generated. This file must not already exist and is required to have a *.apks* filename extension. Bringing these requirements together results in the following command line (allowing for differences in your operating system path structure):

```
java -jar bundletool-all-0.9.0.jar build-apks --bundle=/tmp/MyApps/app/release/
bundle.aab --output=/tmp/MyApks.apks --ks=/MyKeys/release.keystore.jks --ks-key-
alias=MyReleaseKey
```

When this command is executed, a prompt will appear requesting the keystore password before the APK files are generated into the specified APK Set file. The APK Set file is a ZIP file containing all the APK files generated from the app bundle.

To install the appropriate APK files onto a connected device or emulator, use a command similar to the following:

```
java -jar bundletool-all-0.9.0.jar install-apks --apks=/tmp/MyApks.apks
```

This command will instruct the tool to identify the appropriate APK files for the connected device and install them so that the app can be launched and tested.

It is also possible to extract the APK files from the APK Set for the connected device without installing them. The first step in this process is to obtain the specification of the connected device as follows:

```
java -jar bundletool-all-0.9.0.jar get-device-spec --output=/tmp/device.json
```

The above command will generate a JSON file similar to the following:

```
{
  "supportedAbis": ["x86"],
  "supportedLocales": ["en-US"],
  "screenDensity": 420,
  "sdkVersion": 27
```

```
}
```

Next, this specification file is used to extract the matching APK files from the APK Set:

```
java -jar bundletool-all-0.9.0.jar extract-apks --apks=/tmp/MyApks.apks --output-dir=/tmp/nexus5_apks --device-spec=/tmp/device.json
```

When executed, the directory specified via the *--output-dir* flag will contain the correct APK files for the specified device configuration.

The next step in bringing an Android application to market involves submitting it to the Google Play Developer Console o make it available for testing.

81.9 Uploading the App Bundle to the Google Play Developer Console

Return to the Google Play Console and select the *Internal testing* option (marked A in Figure 81-9) located in the *Testing* section of the navigation panel before clicking on the *Create new release* button (B):

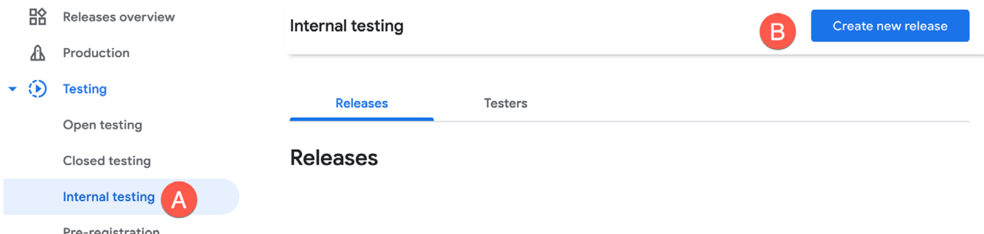


Figure 81-9

On the resulting screen, click on the Continue button (marked A below) to confirm the use of Google Play app signing, then drag and drop the bundle file generated by Android Studio onto the upload drop point (B):

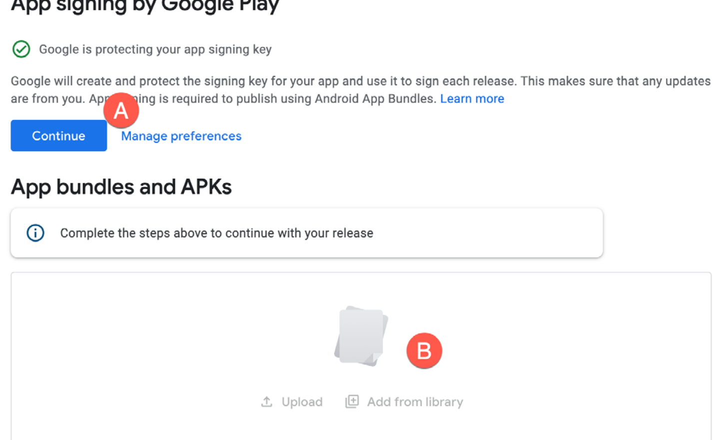


Figure 81-10

When the upload is complete, scroll down the screen and enter the release name and optional release notes. The release name can be any information you need to help you recognize the release, and it is not visible to users.

After the app bundle file is uploaded, Google Play will generate all the necessary APK files ready for testing. Once the APK files have been generated, scroll down to the bottom of the screen and click on the *Save* button. Once the settings have been saved, click on the *Review release* button.

81.10 Exploring the App Bundle

On the review screen, click on the arrow to the right of the uploaded bundle as indicated in Figure 81-11:

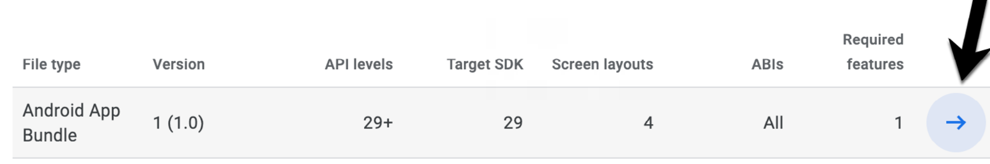


Figure 81-11

In the resulting panel, click on the *Explore bundle* link to load the app bundle explorer. This provides summary information relating to the API levels, screen layouts, and platforms supported by the app bundle:

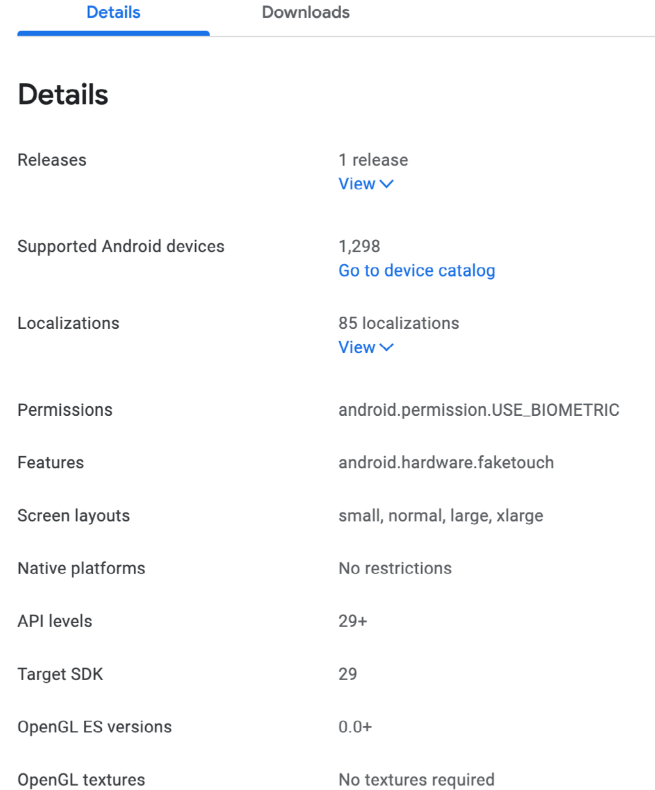


Figure 81-12

Clicking on the *Go to device catalog* link will display the devices that are supported by the APK file:

Device catalog

View and manage the devices that are compatible with your app. Show more

Supported devices | Add filter | Search devices

1,298 devices supported

Manage devices | Download device list

Device	RAM (total memory)	System on Chip	Status
A1 Alpha 20+	3,840 MB (3,726 MB)	Mediatek MT6771T	Supported
AT&T U318AA	1,024 MB (887 MB)	Mediatek MT6739WW	Supported
AT&T U705AA	2,816 MB (2,794 MB)	Mediatek MT6762	Supported

Figure 81-13

Currently, the app is ready for testing but can only be rolled out once some testers have been set up within the console.

81.11 Managing Testers

If the app is still in the Internal, Alpha, or Beta testing phase, a list of authorized testers may be specified by selecting the app from within the Google Play console, clicking on *Internal testing* in the navigation panel, and selecting the *Testers* tab as shown in Figure 81-14:

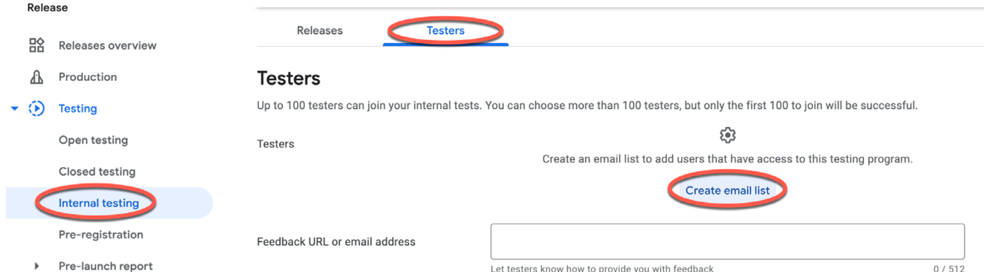


Figure 81-14

To add testers, click on the *Create email list* button, name the list, and specify the test users' email addresses manually or by uploading a CSV file.

The "Join on the web" URL may now be copied from the screen and provided to the test users so that they accept the testing invitation and download the app.

81.12 Rolling the App Out for Testing

Now that an internal release has been created and a list of testers added, the app is ready to be rolled out for testing. Remaining within the *Internal testing* screen, select the Releases tab before clicking on the Edit button for the recently created release:

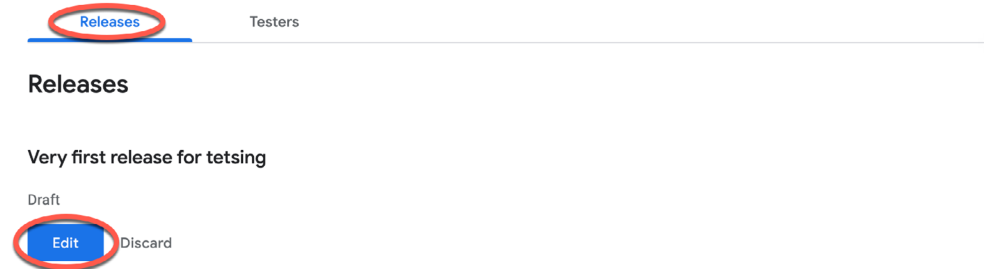


Figure 81-15

On the review screen, scroll to the bottom and click on the *Start rollout to Internal testing* button. After a short delay while the release is processed, the app will be ready to be downloaded and tested by the designated users.

81.13 Uploading New App Bundle Revisions

The first app bundle file uploaded for your application will invariably have a version code of 1. If an attempt is made to upload another bundle file with the same version code number, the console will reject the file with the following error:

```
You need to use a different version code for your APK because you already have
one with version code 1.
```

To resolve this problem, the version code embedded into the bundle file needs to be increased. This is performed in the *module* level *build.gradle.kts* file of the project, shown highlighted in Figure 81-16:

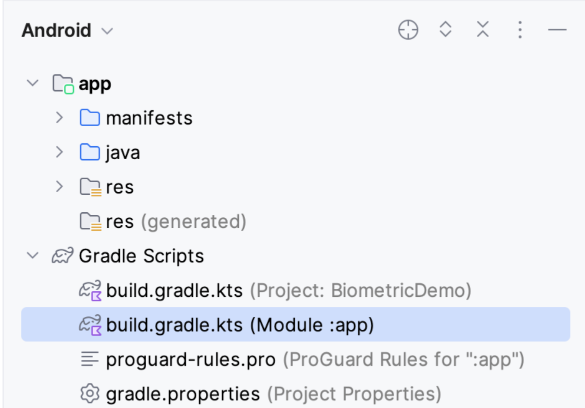


Figure 81-16

This file will typically read as follows:

```
plugins {
    id("com.android.application")
}

android {
    namespace = "com.ebookfrenzy.biometricdemo"
    compileSdk = 33

    defaultConfig {
        applicationId = "com.ebookfrenzy.biometricdemo"
```

```
        minSdk = 29
        targetSdk = 33
        versionCode = 1
        versionName = "1.0"
.
.
}
```

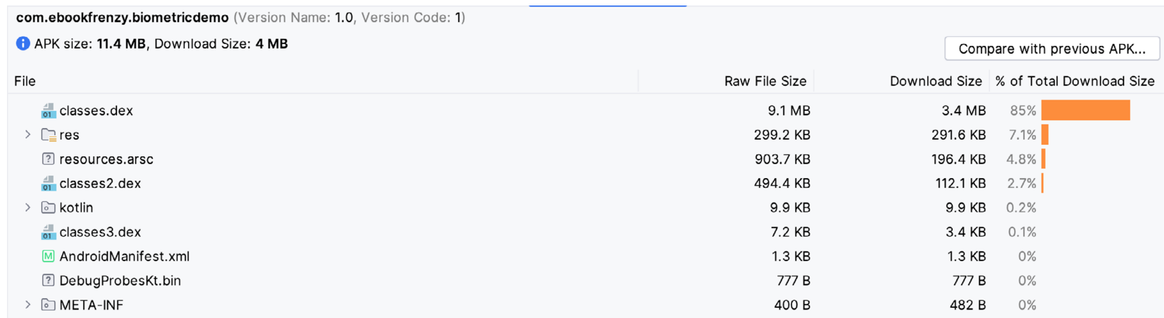
To change the version code, change the number declared next to *versionCode*. To also change the version number displayed to users of your application, change the *versionName* string. For example:

```
versionCode 2
versionName "2.0"
```

After making these changes, rebuild the APK file and perform the upload again.

81.14 Analyzing the App Bundle File

Android Studio provides the ability to analyze the content of an app bundle file. To analyze a bundle file, select the Android Studio *Build -> Analyze APK…* menu option and navigate to and choose the bundle file to be reviewed. Once loaded into the tool, information will be displayed about the raw and download size of the package together with a listing of the file structure of the package as illustrated in Figure 81-17:



com.ebookfrenzy.biometricdemo (Version Name: 1.0, Version Code: 1)
APK size: 11.4 MB, Download Size: 4 MB
Compare with previous APK...

File	Raw File Size	Download Size	% of Total Download Size
classes.dex	9.1 MB	3.4 MB	85%
res	299.2 KB	291.6 KB	7.1%
resources.arsc	903.7 KB	196.4 KB	4.8%
classes2.dex	494.4 KB	112.1 KB	2.7%
kotlin	9.9 KB	9.9 KB	0.2%
classes3.dex	7.2 KB	3.4 KB	0.1%
AndroidManifest.xml	1.3 KB	1.3 KB	0%
DebugProbesKt.bin	777 B	777 B	0%
META-INF	400 B	482 B	0%

Figure 81-17

Selecting the *classes.dex* file will display the class structure of the file in the lower panel. Within this panel, details of the individual classes may be explored down to the level of the methods within a class:

Load Proguard mappings...
This dex file defines 6061 classes with 47393 methods, and references 57837 methods.

Class	Defined Methods	Referenced Methods	Size
androidx	23159	25215	2.7 MB
com	9972	11084	986.5 KB
kotlin	9676	10709	1.1 MB
kotlinx	4501	4733	542.9 KB
android	62	4596	43.2 KB
java		1450	11.8 KB
org	23	45	2.9 KB

Figure 81-18

Similarly, selecting a resource or image file within the file list will display the file content within the lower panel. The size differences between two bundle files may be reviewed by clicking on the *Compare with previous APK…* button and selecting a second bundle file.

81.15 Summary

Once an app project is complete or ready for user testing, it can be uploaded to the Google Play console and published for production, internal, alpha, or beta testing. Before the app can be uploaded, an app entry must be created within the console, including information about the app and screenshots for use within the Play Store. A release Android App Bundle file is generated and signed with an upload key within Android Studio. After the bundle file has been uploaded, Google Play removes the upload key and replaces it with the securely stored app signing key, and the app is ready to be published.

The content of a bundle file can be reviewed at any time by loading it into the Android Studio APK Analyzer tool.

Chapter 82

82. 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.

82.1 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 ("com.android.billingclient:billing:<latest version>")
.
.
}
```

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.

82.2 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 82-1 below:

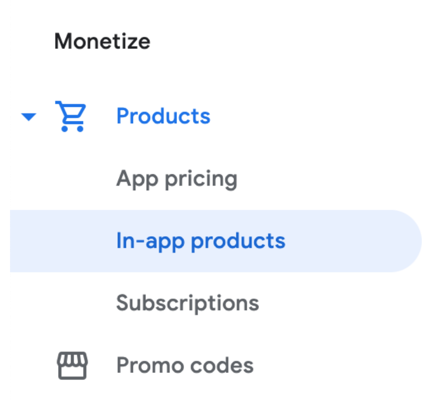


Figure 82-1

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.

82.3 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();
```

82.4 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
    }
});
```

82.5 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.

82.6 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.

82.7 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);
```

82.8 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
    }
});
```

82.9 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> 436
<fragment> 247
<fragment> element 247
<receiver> 470
<service> 436, 480, 487
Code Reformatting 79
.well-known folder 443, 466, 680

A

AbsoluteLayout 126
ACCESS_COARSE_LOCATION permission 504
ACCESS_FINE_LOCATION permission 504
acknowledgePurchase() method 719
ACTION_CREATE_DOCUMENT 596
ACTION_CREATE_INTENT 596
ACTION_DOWN 222
ACTION_MOVE 222
ACTION_OPEN_DOCUMENT intent 588
ACTION_POINTER_DOWN 222
ACTION_POINTER_UP 222
ACTION_UP 222
ACTION_VIEW 461
Active / Running state 100
Activity 87, 103
 adding views in Java code 203
 class 103
 creation 16
 Entire Lifetime 107
 Foreground Lifetime 107
 lifecycle methods 106
 lifecycles 97
 returning data from 440
 state change example 111
 state changes 103
 states 100
 Visible Lifetime 107
Activity Lifecycle 99
Activity Manager 86
ActivityResultLauncher 441
Activity Stack 99
Actual screen pixels 194
adb
 command-line tool 63
 connection testing 69
 device pairing 67
 enabling on Android devices 63
 Linux configuration 66
 list devices 63
 macOS configuration 64
 overview 63
 restart server 64
 testing connection 69
 WiFi debugging 67
 Windows configuration 65
 Wireless debugging 67
 Wireless pairing 67
addCategory() method 469
addMarker() method 644
addView() method 197
ADD_VOICEMAIL permission 504
android
 exported 437
 gestureColor 240
 layout_behavior property 419
 onClick 249
 process 437, 487
 uncertainGestureColor 240
Android
 Activity 87
 architecture 83
 events 215
 intents 88
 onClick Resource 215

runtime 84
SDK Packages 6
android.app 84
Android Architecture Components 265
android.content 84
android.content.Intent 439
android.database 84
Android Debug Bridge. *See* ADB
Android Development
System Requirements 3
Android Devices
designing for different 125
android.graphics 84
android.hardware 84
android.intent.action 475
android.intent.action.BOOT_COMPLETED 437
android.intent.action.MAIN 461
android.intent.category.LAUNCHER 461
Android Libraries 84
android.media 85
Android Monitor tool window 36
Android Native Development Kit 85
android.net 85
android.opengl 84
android.os 85
android.permission.RECORD_AUDIO 623
android.print 85
Android Project
create new 15
android.provider 85
Android SDK Location
identifying 10
Android SDK Manager 8, 10
Android SDK Packages
version requirements 8
Android SDK Tools
command-line access 9
Linux 11
macOS 11
Windows 7 10
Windows 8 10
Android Software Stack 83
Android Storage Access Framework 588
Android Studio
changing theme 61
downloading 3
Editor Window 56
installation 4
Linux installation 5
macOS installation 4
Navigation Bar 55
Project tool window 56
setup wizard 5
Status Bar 56
Toolbar 55
Tool window bars 56
tool windows 56
updating 12
Welcome Screen 53
Windows installation 4
android.text 85
android.util 85
android.view 85
android.view.View 128
android.view.ViewGroup 125, 128
Android Virtual Device. *See* AVD
overview 31
Android Virtual Device Manager 31
android.webkit 85
android.widget 85
AndroidX libraries 748
API Key 635
APK analyzer 712
APK file 705
APK File
analyzing 712
APK Signing 748
APK Wizard dialog 704
App Architecture
modern 265
AppBar
anatomy of 417
appbar_scrolling_view_behavior 419
App Bundles 701

creating 705
overview 701
revisions 711
uploading 708
AppCompatActivity class 104
App Inspector 57
Application
stopping 36
Application Context 89
Application Framework 85
Application Manifest 89
Application Resources 89
App Link
Adding Intent Filter 688
Digital Asset Links file 680, 443
Intent Filter Handling 689
Intent Filters 679
Intent Handling 680
Testing 692
URL Mapping 685
App Links 679
auto verification 442
autoVerify 443
overview 679
Apply Changes 211
Apply Changes and Restart Activity 211
Apply Code Changes 211
fallback settings 213
options 211
Run App 211
tutorial 213
applyToActivitiesIfAvailable() method 743
Architecture Components 265
ART 84
assetlinks.json , 680, 443
Attribute Keyframes 342
Audio
supported formats 621
Audio Playback 621
Audio Recording 621
Autoconnect Mode 159
Automatic Link Verification 442, 465
autoVerify 443, 688
AVD
cold boot 48
command-line creation 31
creation 31
device frame 40
Display mode 51
launch in tool window 40
overview 31
quickboot 48
Resizable 50
running an application 34
Snapshots 47
standalone 37
starting 33
Startup size and orientation 34

B

Background Process 98
Barriers 152
adding 171
constrained views 152
Baseline Alignment 151
beginTransaction() method 248
BillingClient 720
acknowledgePurchase() method 719
consumeAsync() method 719
getPurchaseState() method 719
initialization 716, 726
launchBillingFlow() method 718
queryProductDetailsAsync() method 718
queryPurchasesAsync() method 720
BillingResult 733
getDebugMessage() 733
Binding Expressions 289
one-way 289
two-way 290
BIND_JOB_SERVICE permission 437
bindService() method 435, 477, 482
Biometric Authentication 693
callbacks 697
overview 693

tutorial 693

Biometric Prompt 698

BitmapFactory 589

black activity 16

Blank template 129

Blueprint view 157

BODY_SENSORS permission 504

Bound Service 435, 477

adding to a project 478

Implementing the Binder 478

Interaction options 477

BoundService class 479

Broadcast Intent 469

example 472

overview 88, 469

sending 472

Sticky 471

Broadcast Receiver 469

adding to manifest file 474

creation 473

overview 88, 470

BroadcastReceiver class 470

BroadcastReceiver superclass 473

BufferedReader object 599

Build tool window 58

Build Variants , 58

tool window 58

Bundle class 120

Bundled Notifications 523

C

Calendar permissions 504

CALL_PHONE permission 504

CAMERA permission 504

Camera permissions 504

CameraUpdateFactory class

methods 645

CancellationSignal 698

Canvas class 674

CardView

layout file 397

responding to selection of 406

CardView class 397

CATEGORY_OPENABLE 588

C/C++ Libraries 85

Chain bias 180

chain head 150

chains 150

Chains

creation of 177

Chain style

changing 179

chain styles 150

CheckBox 125

checkSelfPermission() method 508

Circle class 631

Code completion 74

Code Editor

basics 71

Code completion 74

Code Generation 77

Code Reformatting 79

Document Tabs 72

Editing area 72

Gutter Area 72

Live Templates 80

Splitting 74

Statement Completion 76

Status Bar 73

Code Generation 77

code samples

download 1

cold boot 48

CollapsingToolbarLayout

example 420

introduction 420

parallax mode 420

pin mode 420

setting scrim color 423

setting title 423

with image 420

Color class 675

COLOR_MODE_COLOR 650, 670

COLOR_MODE_MONOCHROME 650, 670

Common Gestures 229
detection 229
Component tree 20
Constraint Bias 149
adjusting 163
ConstraintLayout
advantages of 155
Availability 156
Barriers 152
Baseline Alignment 151
chain bias 180
chain head 150
chains 150
chain styles 150
Constraint Bias 149
Constraints 147
conversion to 175
convert to MotionLayout 349
deleting constraints 162
guidelines 169
Guidelines 152
manual constraint manipulation 159
Margins 148, 163
Opposing Constraints 148, 165
overview of 147
Packed chain 151, 180
ratios 155, 181
Spread chain 150
Spread inside 180
Spread inside chain 150
tutorial 185
using in Android Studio 157
Weighted chain 150, 180
Widget Dimensions 151, 167
Widget Group Alignment 173
ConstraintLayout chains
creation of 177
in layout editor 177
ConstraintLayout Chain style
changing 179
Constraints
deleting 162
ConstraintSet
addToHorizontalChain() method 200
addToVerticalChain() method 200
alignment constraints 199
apply to layout 198
applyTo() method 198
centerHorizontally() method 199
centerVertically() method 199
chains 199
clear() method 200
clone() method 199
connect() method 198
connect to parent 198
constraint bias 199
copying constraints 199
create 198
create connection 198
createHorizontalChain() method 199
createVerticalChain() method 199
guidelines 200
removeFromHorizontalChain() method 200
removeFromVerticalChain() method 200
removing constraints 200
rotation 201
scaling 200
setGuidelineBegin() method 200
setGuidelineEnd() method 200
setGuidelinePercent() method 200
setHorizonalBias() method 199
setRotationX() method 201
setRotationY() method 201
setScaleX() method 200
setScaleY() method 200
setTransformPivot() method 201
setTransformPivotX() method 201
setTransformPivotY() method 201
setVerticalBias() method 199
sizing constraints 199
tutorial 203
view IDs 205
ConstraintSet class 197, 198
Constraint Sets 198

ConstraintSets
  configuring 338
consumeAsync() method 719
ConsumeParams 731
ConsumeResponseListener 719
Contacts permissions 504
container view 125
Content Provider 86
  overview 89
Context class 89
CoordinatorLayout 126, 419
createPrintDocumentAdapter() method 665
Custom Attribute 339
Custom Document Printing 653, 665
Custom Gesture
  recognition 235
Custom Print Adapter
  implementation 667
Custom Print Adapters 665
Custom Theme
  building 737
Cycle Editor 367
Cycle Keyframe 347
Cycle Keyframes
  overview 363

D

dangerous permissions
  list of 504
Dark Theme 36
  enable on device 36
Data Access Object (DAO) 554
Data Access Objects (DAO) 558
Database Inspector 561, 584
  live updates 585
  SQL query 585
Database Rows 548
Database Schema 547
Database Tables 547
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 36
Debugging
  enabling on device 63
debug.keystore file 443, 465
DefaultLifecycleObserver 308, 311
deltaRelative 343
Density-independent pixels 193
Density Independent Pixels
  converting to pixels 208
Device Definition
  custom 143
Device File Explorer 58
device frame 40
Device Mirroring 69
  enabling 69
device pairing 67
Digital Asset Links file 680, 443, 443
Direct Reply Input 534
document provider 587
dp 193
Dynamic Colors
  applyToActivitiesIfAvailable() method 743
  enabling in Android 743
Dynamic State 105
  saving 119

E

Empty Process 99
Empty template 129
Emulator
  battery 46
  cellular configuration 46
  configuring fingerprints 48
  directional pad 46
  extended control options 45

Extended controls 45
fingerprint 46
location configuration 46
phone settings 46
Resizable 50
resize 45
rotate 44
Screen Record 47
Snapshots 47
starting 33
take screenshot 44
toolbar 43
toolbar options 43
tool window mode 50
Virtual Sensors 47
zoom 44
enablePendingPurchases() method 719
enabling ADB support 63
ettings.gradle file 748
Event Handling 215
example 216
Event Listener 217
Event Listeners 216
Events
consuming 219
explicit
intent 88
explicit intent 439
Explicit Intent 439
Extended Control
options 45

F

Files
switching between 72
findPointerIndex() method 222
findViewById() 91
Fingerprint
emulation 48
Fingerprint authentication
device configuration 694
permission 694
steps to implement 693
Fingerprint Authentication
overview 693
tutorial 693
FLAG_INCLUDE_STOPPED_PACKAGES 469
flexible space area 417
floating action button 16, 130
changing appearance of 378
margins 376
removing 131
sizes 376
Foldable Devices 108
multi-resume 108
Foldable Emulator 540
Foldables 539
Foreground Process 98
Forward-geocoding 637
Fragment
creation 245
event handling 249
XML file 246
FragmentActivity class 104
Fragment Communication 250
Fragments 245
adding in code 248
duplicating 386
example 253
overview 245
FragmentStateAdapter class 389
FrameLayout 126

G

Geocoder object 638
Geocoding 636
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 504
getAction() method 475
getDebugMessage() 733
getFromLocation() method 638
getId() method 198
getIntent() method 440
getPointerCount() method 222
getPointerId() method 222
getPurchaseState() method 719
getService() method 482
GNU/Linux 84
Google Cloud
  billing account 632
  new project 633
Google Cloud Print 648
Google Drive 588
  printing to 648
GoogleMap 631
  map types 641
GoogleMap.MAP_TYPE_HYBRID 641
GoogleMap.MAP_TYPE_NONE 641
GoogleMap.MAP_TYPE_NORMAL 641
GoogleMap.MAP_TYPE_SATELLITE 641
GoogleMap.MAP_TYPE_TERRAIN 641
Google Maps Android API 631
  Controlling the Map Camera 645
  displaying controls 642
  Map Markers 644
  overview 631
Google Maps SDK 631
  API Key 635
  Credentials 635
  enabling 634
  Maps SDK for Android 635
Google Play App Signing 704
Google Play Console 724
  Creating an in-app product 724
  License Testers 725
Google Play Developer Console 702
Gradle
  APK signing settings 752
  Build Variants 748
  command line tasks 753
  dependencies 747
  Manifest Entries 748
  overview 747
  sensible defaults 747
Gradle Build File
  top level 749
Gradle Build Files
  module level 750
gradle.properties file 748
GridLayout 126
GridLayoutManager 395

H

Handler class 486
HP Print Services Plugin 647
HTML printing 651
HTML Printing
  example 655

I

IBinder 435, 479
IBinder object 477, 487
Image Printing 650
implicit
  intent 88
implicit intent 439
Implicit Intent 441
Implicit Intents
  example 457

importance hierarchy 97
in 193
INAPP 720
In-App Products 715
In-App Purchasing 723
acknowledgePurchase() method 719
BillingClient 716
BillingResult 733
consumeAsync() method 719
ConsumeParams 731
ConsumeResponseListener 719
Consuming purchases 730
enablePendingPurchases() method 719
getPurchaseState() method 719
launchBillingFlow() method 718
Libraries 723
newBuilder() method 716
onBillingServiceDisconnected() callback 728
onBillingServiceDisconnected() method 717
onBillingSetupFinished() listener 728
onProductDetailsResponse() callback 728
Overview 715
ProductDetail 718
ProductDetails 729
products 715
ProductType 720
ProductType.INAPP 720
ProductType.SUBS 720
Purchase Flow 729
PurchaseResponseListener 720
PurchasesUpdatedListener 719
PurchaseUpdatedListener 729
purchase updates 729
queryProductDetailsAsync() 728
queryProductDetailsAsync() method 718
queryPurchasesAsync() 731
queryPurchasesAsync() method 720
runOnUiThread() 729
subscriptions 715
tutorial 723
In-Memory Database 561
Intent 88
explicit 88
implicit 88
Intent Availability
checking for 446
Intent.CATEGORY_OPENABLE 596
Intent Filters 442
App Link 679
Intents 439
ActivityResultLauncher 441
overview 439
registerForActivityResult() 454
Intent Service 435
Intent URL 460

J

Java Native Interface 85
Jetpack 265
overview 265
JobIntentService 435
BIND_JOB_SERVICE permission 437
onHandleWork() method 435

K

KeyAttribute 342
Keyboard Shortcuts 59
KeyCycle 363
Cycle Editor 367
tutorial 363
Keyframe 356
Keyframes 342
KeyFrameSet 372
KeyPosition 343
deltaRelative 343
parentRelative 343
pathRelative 344
Keystore File
creation 704
KeyTimeCycle 363
keytool 443
KeyTrigger 346
Killed state 100

L

launchBillingFlow() method 718

layout_collapseMode

- parallax 422
- pin 422

layout_constraintDimentionRatio 182

layout_constraintHorizontal_bias 180

layout_constraintVertical_bias 180

layout editor

- ConstraintLayout chains 177

Layout Editor 19, 185

- Autoconnect Mode 159
- code mode 136
- Component Tree 134
- design mode 133
- device screen 134
- example project 185
- Inference Mode 159
- palette 134
- properties panel 134
- Sample Data 142
- Setting Properties 138
- toolbar 134
- user interface design 185
- view conversion 141

Layout Editor Tool

- changing orientation 20
- overview 133

Layout Inspector 58

Layout Managers 125

LayoutResultCallback object 670

Layouts 125

layout_scrollFlags

- enterAlwaysCollapsed mode 419
- enterAlways mode 419
- exitUntilCollapsed mode 419
- scroll mode 419

Layout Validation 144

libc 85

License Testers 725

Lifecycle

- awareness 307
- components 268
- owners 307
- states and events 309
- tutorial 311

Lifecycle-Aware Components 307

Lifecycle Methods 106

Lifecycle Observer 311

- creating a 311

Lifecycle Owner

- creating a 313

Lifecycles

- modern 268

LinearLayout 126

LinearLayoutManager 395

LinearLayoutManager layout 404

Linux Kernel 84

list devices 63

LiveData 266, 279

- adding to ViewModel 279
- observer 281
- tutorial 279

Live Templates 80

Local Bound Service 477

- example 477

Location Manager 86

Location permission 504

Logcat

- tool window 57

LogCat

- enabling 115

M

MANAGE_EXTERNAL_STORAGE 505

- adb enabling 505
- testing 505

Manifest File

- permissions 461

Maps 631

MapView 631

- adding to a layout 638

Marker class 631

Master/Detail Flow

creation 426
two pane mode 425
match_parent properties 193
Material design 375
Material Design 2 735
Material Design 2 Theming 735
Material Design 3 735
Material Theme Builder 737
Material You 735
MediaController
adding to VideoView instance 605
MediaController class 602
methods 602
MediaPlayer class 621
methods 621
MediaRecorder class 621
methods 622
recording audio 622
Memory Indicator 73
Messenger object 487
Microphone
checking for availability 624
Microphone permissions 504
mm 193
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 645
multiple devices
testing app on 35
Multiple Touches
handling 222
multi-resume 108
Multi-Touch
example 222
Multi-touch Event Handling 221
Multi-Window
attributes 543
Multi-Window Mode
detecting 544
entering 541
launching activity into 545
Multi-Window Notifications 544
multi-window support 108
Multi-Window Support
enabling 542
My Location Layer 631

N

Navigation 317
adding destinations 326
overview 317
pass data with safeargs 333
passing arguments 322
stack 317
tutorial 323
Navigation Action

triggering 321
Navigation Architecture Component 317
Navigation Component
tutorial 323
Navigation Controller
accessing 321
Navigation Graph 320, 324
adding actions 329
creating a 324
Navigation Host 318
declaring 325
newBuilder() method 716
normal permissions 503
Notification
adding actions 522
Direct Reply Input 534
issuing a basic 518
launch activity from a 520
PendingIntent 530
Reply Action 532
updating direct reply 535
Notifications
bundled 523
overview 511
Notifications Manager 86

O

Observer
implementing a LiveData 281
onAttach() method 250
onBillingServiceDisconnected() callback 728
onBillingServiceDisconnected() method 717
onBillingSetupFinished() listener 728
onBind() method 436, 477
onBindViewHolder() method 403
OnClick 341
onClickListener 216, 217, 220
onClick() method 215
onCreateContextMenuListener 216
onCreate() method 98, 106, 436
onCreateView() method 107
onDestroy() method 106, 436
onDoubleTap() method 229
onDown() method 229
onFling() method 229
onFocusChangeListener 216
OnFragmentInteractionListener
implementation 331
onGesturePerformed() method 235
onHandleWork() method 436
onKeyListener 216
onLayoutFailed() method 670
onLayoutFinished() method 671
onLongClickListener 216
onLongClick() method 219
onLongPress() method 229
onMapReady() method 640
onPageFinished() callback 656
onPause() method 106
onProductDetailsResponse() callback 728
onReceive() method 98, 470, 471, 473
onRequestPermissionsResult() method 507, 628, 516, 528
onRestart() method 106
onRestoreInstanceState() method 107
onResume() method 98, 106
onSaveInstanceState() method 107
onScaleBegin() method 241
onScaleEnd() method 241
onScale() method 241
onScroll() method 229
OnSeekBarChangeListener 260
onServiceConnected() method 477, 481, 488
onServiceDisconnected() method 477, 481, 488
onShowPress() method 229
onSingleTapUp() method 229
onStartCommand() method 436
onStart() method 106
onStop() method 106
onTouchEvent() method 229, 241
onTouchListener 216
onTouch() method 221
onViewCreated() method 107
onViewStatusRestored() method 107
openFileDescriptor() method 588

OpenJDK 3

P

Package Explorer 18
Package Manager 86
PackageManager class 624
PackageManager.FEATURE_MICROPHONE 624
PackageManager.PERMISSION_DENIED 505
PackageManager.PERMISSION_GRANTED 505
Package Name 16
Packed chain 151, 180
PageRange 672, 673
Paint class 675
parentRelative 343
parent view 127
pathRelative 344
Paused state 100
PdfDocument 653
PdfDocument.Page 665, 672
PendingIntent class 530
Permission
  checking for 505
permissions
  normal 503
Persistent State 105
Phone permissions 504
picker 587
Pinch Gesture
  detection 241
  example 241
Pinch Gesture Recognition 235
Position Keyframes 343
POST_NOTIFICATIONS permission 504, 528
PrintAttributes 670
PrintDocumentAdapter 653, 665
Printing
  color 650
  monochrome 650
Printing framework
  architecture 647
Printing Framework 647
Print Job
  starting 676
PrintManager service 657
Problems
  tool window 58
process
  priority 97
  state 97
PROCESS_OUTGOING_CALLS permission 504
Process States 97
ProductDetail 718
ProductDetails 729
ProductType 720
Profiler
  tool window 58
ProgressBar 125
proguard-rules.pro file 752
ProGuard Support 748
Project Name 16
Project tool window 18, 57
pt 193
PurchaseResponseListener 720
PurchasesUpdatedListener 719
PurchaseUpdatedListener 729
putExtra() method 439, 469
px 194

Q

queryProductDetailsAsync() 728
queryProductDetailsAsync() method 718
queryPurchaseHistoryAsync() method 720
queryPurchasesAsync() 731
queryPurchasesAsync() method 720
quickboot snapshot 48
Quick Documentation 79

R

RadioButton 125
ratios 181
READ_CALENDAR permission 504
READ_CALL_LOG permission 504
READ_CONTACTS permission 504
READ_EXTERNAL_STORAGE permission 505

READ_PHONE_STATE permission 504
READ_SMS permission 504
RECEIVE_MMS permission 504
RECEIVE_SMS permission 504
RECEIVE_WAP_PUSH permission 504
Recent Files Navigation 60
RECORD_AUDIO permission 504
Recording Audio
  permission 623
RecyclerView 395
  adding to layout file 396
  GridLayoutManager 395
  initializing 404
  LinearLayoutManager 395
  StaggeredGridLayoutManager 395
RecyclerView Adapter
  creation of 402
RecyclerView.Adapter 396, 402
  getItemCount() method 396
  onBindViewHolder() method 396
  onCreateViewHolder() method 396
RecyclerView.ViewHolder
  getAdapterPosition() method 406
registerForActivityResult() method 440, 454
registerReceiver() method 471
RelativeLayout 126
releasePersistableUriPermission() method 591
Release Preparation 701
Remote Bound Service 485
  client communication 485
  implementation 485
  manifest file declaration 487
RemoteInput.Builder() method 530
RemoteInput Object 530
Remote Service
  launching and binding 488
  sending a message 489
Repository
  tutorial 571
Repository Modules 268
Resizable Emulator 50
Resource
  string creation 23
Resource File 25
Resource Management 97
Resource Manager , 57
result receiver 471
Reverse-geocoding 637
Reverse Geocoding 636
Room
  Data Access Object (DAO) 554
  entities 554, 555
  In-Memory Database 561
  Repository 554
Room Database 554
  tutorial 571
Room Database Persistence 553
Room Persistence Library 550, 553
root element 125
root view 127
Run
  tool window 57
Running Devices
  tool window 69
runOnUiThread() 729

S

safeargs
Sample Data 142, 409
  tutorial 409
Saved State 267, 301
  library dependencies 303
SavedStateHandle 302, 303
  contains() method 303
  keys() method 303
  remove() method 303
Saved State module 301
SavedStateViewModelFactory 302
ScaleGestureDetector class 241
Scale-independent 193
SDK Packages 6
Secure Sockets Layer (SSL) 85
SeekBar 253
sendBroadcast() method 469, 471

sendOrderedBroadcast() method 469, 471
SEND_SMS permission 504
sendStickyBroadcast() method 469
Sensor permissions 504
Service
anatomy 436
launch at system start 437
manifest file entry 436
overview 88
run in separate process 437
ServiceConnection class 488
Service Process 98
Service Restart Options 436
setAudioEncoder() method 622
setAudioSource() method 622
setBackgroundColor() 198
setCompassEnabled() method 642
setContentView() method 197, 203
setId() method 198
setMyLocationButtonEnabled() method 643
setOnClickListener() method 215, 217
setOnDoubleTapListener() method 229, 232
setOutputFile() method 622
setOutputFormat() method 622
setResult() method 441
setRotateGesturesEnabled() method 643
setScrollGesturesEnabled() method 643
setText() method 122
setTiltGesturesEnabled() method 643
settings.gradle.kts file 748
setTransition() 347
setVideoSource() method 622
setZoomControlsEnabled() method 642, 643
SHA-256 certificate fingerprint 443
shouldOverrideUrlLoading() method 656
SimpleOnScaleGestureListener 241
SimpleOnScaleGestureListener class 243
SMS permissions 504
Snackbar 375, 376, 377
Snapshots
emulator 47
sp 193
Spread chain 150
Spread inside 180
Spread inside chain 150
SQL 548
SQLite 547
AVD command-line use 549
Columns and Data Types 547
overview 548
Primary keys 548
StaggeredGridLayoutManager 395
startActivity() method 439
startForeground() method 98
START_NOT_STICKY 436
START_REDELIVER_INTENT 436
START_STICKY 436
State
restoring 122
State Change
handling 101
Statement Completion 76
Status Bar Widgets 73
Memory Indicator 73
Sticky Broadcast Intents 471
Stopped state 100
Storage Access Framework 587
ACTION_CREATE_DOCUMENT 588
ACTION_OPEN_DOCUMENT 588
deleting a file 591
example 593
file creation 596
file filtering 588
file reading 589
file writing 590
intents 588
MIME Types 589
Persistent Access 591
picker 587
Storage permissions 505
StringBuilder object 599
strings.xml file 27
Structure
tool window 58

Structured Query Language 548
Structure tool window 58
SUBS 720
subscriptions 715
SupportMapFragment class 631
Switcher 60
System Broadcasts 475
system requirements 3

T

TabLayout
  adding to layout 387
  app
    tabGravity property 393
    tabMode property 392
  example 384
  fixed mode 392
  getItemCount() method 383
  overview 383
TableLayout 126, 563
TableRow 563
Telephony Manager 86
Templates
  blank vs. empty 129
Terminal
  tool window 58
Theme
  building a custom 737
Theming 735
  tutorial 739
Time Cycle Keyframes 347
TODO
  tool window 59
ToolbarListener 250
tools
  layout 247
Tool window bars 56
Tool windows 56
Touch Actions 222
Touch Event Listener
  implementation 223
Touch Events
  intercepting 221
Touch handling 221

U

UiSettings class 631
unbindService() method 435
unregisterReceiver() method 471
upload key 704
URL Mapping 685
USB connection issues
  resolving 66
USE_BIOMETRIC 694
user interface state 105
USE_SIP permission 504

V

Video Playback 601
VideoView class 601
  methods 601
  supported formats 601
view bindings
  enabling 92
  using 92
View class
  setting properties 204
view conversion 141
ViewGroup 125
View Groups 125
View Hierarchy 127
ViewHolder class 396
  sample implementation 403
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 387

example 384
Views 125
Java creation 197
View System 86
Virtual Device Configuration dialog 32
Virtual Sensors 47
Visible Process 98

W

WebViewClient 651, 656
WebView view 459
Weighted chain 150, 180
Welcome screen 53
Widget Dimensions 151
Widget Group Alignment 173
Widgets palette 186
WiFi debugging 67
Wireless debugging 67
Wireless pairing 67
wrap_content properties 195
WRITE_CALENDAR permission 504
WRITE_CALL_LOG permission 504
WRITE_CONTACTS permission 504
WRITE_EXTERNAL_STORAGE permission 505

X

XML Layout File
manual creation 193
vs. Java Code 197