Android Studio 3.6 Development Essentials

Java Edition

Android Studio 3.6 Development Essentials – Java Edition

ISBN-13: 978-1-951442-15-6

© 2020 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 2
1.3 Errata 2
1.4 Download the eBook 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 Making the Android SDK Tools Command-line Accessible 8
2.6.1 Windows 7 8
2.6.2 Windows 8.1 9
2.6.3 Windows 10 9
2.6.4 Linux 10
2.6.5 macOS 10
2.7 Android Studio Memory Management 10
2.8 Updating Android Studio and the SDK 11
2.9 Summary 11

3. Creating an Example Android App in Android Studio 13

3.1 About the Project 13
3.2 Creating a New Android Project 13
3.3 Creating an Activity 14
3.4 Defining the Project and SDK Settings 14
3.5 Modifying the Example Application 15
3.6 Modifying the User Interface 16
3.7 Reviewing the Layout and Resource Files 21
3.8 Adding Interaction 23
3.9 Summary 25

4. Creating an Android Virtual Device (AVD) in Android Studio 27

4.1 About Android Virtual Devices 27
4.2 Creating a New AVD 28
4.3 Starting the Emulator 29
4.4 Running the Application in the AVD 29
4.5 Stopping a Running Application 31
4.6 Supporting Dark Theme 31
4.7 AVD Command-line Creation 33

4.8 Android Virtual Device Configuration Files 34
4.9 Moving and Renaming an Android Virtual Device 34
4.10 Summary 35

5. Using and Configuring the Android Studio AVD Emulator 37

5.1 The Emulator Environment 37
5.2 The Emulator Toolbar Options 37
5.3 Working in Zoom Mode 39
5.4 Resizing the Emulator Window 39
5.5 Extended Control Options 39
5.5.1 Location 39
5.5.2 Displays 39
5.5.3 Cellular 40
5.5.4 Camera 40
5.5.5 Battery 40
5.5.6 Phone 40
5.5.7 Directional Pad 40
5.5.8 Microphone 40
5.5.9 Fingerprint 40
5.5.10 Virtual Sensors 40
5.5.11 Snapshots 40
5.5.12 Record and Playback 40
5.5.13 Google Play 41
5.5.14 Settings 41
5.5.15 Help 41
5.6 Working with Snapshots 41
5.7 Configuring Fingerprint Emulation 42
5.8 Summary 43

6. A Tour of the Android Studio User Interface 45

6.1 The Welcome Screen 45
6.2 The Main Window 46
6.3 The Tool Windows 47
6.4 Android Studio Keyboard Shortcuts 50
6.5 Switcher and Recent Files Navigation 50
6.6 Changing the Android Studio Theme 51
6.7 Summary 51

7. Testing Android Studio Apps on a Physical Android Device 53

7.1 An Overview of the Android Debug Bridge (ADB) 53
7.2 Enabling ADB on Android based Devices 53
7.2.1 macOS ADB Configuration 54
7.2.2 Windows ADB Configuration 55
7.2.3 Linux adb Configuration 56
7.3 Testing the adb Connection 56
7.4 Summary 57

8. The Basics of the Android Studio Code Editor 59

8.1 The Android Studio Editor 59
8.2 Splitting the Editor Window 61
8.3 Code Completion 62

8.4 Statement Completion 63
8.5 Parameter Information 64
8.6 Parameter Name Hints 64
8.7 Code Generation 64
8.8 Code Folding 65
8.9 Quick Documentation Lookup 67
8.10 Code Reformatting 67
8.11 Finding Sample Code 68
8.12 Summary 68

9. An Overview of the Android Architecture 69

9.1 The Android Software Stack 69
9.2 The Linux Kernel 70
9.3 Android Runtime – ART 70
9.4 Android Libraries 70
9.4.1 C/C++ Libraries 71
9.5 Application Framework 71
9.6 Applications 72
9.7 Summary 72

10. The Anatomy of an Android Application 73

10.1 Android Activities 73
10.2 Android Fragments 73
10.3 Android Intents 74
10.4 Broadcast Intents 74
10.5 Broadcast Receivers 74
10.6 Android Services 74
10.7 Content Providers 75
10.8 The Application Manifest 75
10.9 Application Resources 75
10.10 Application Context 75
10.11 Summary 75

11. An Overview of Android View Binding 77

11.1 Find View by ID 77
11.2 View Bindings 77
11.3 Converting the AndroidSample Project 78
11.4 Enabling View Binding 78
11.5 Using View Bindings 78
11.6 Choosing an Option 79
11.7 Summary 80

12. Understanding Android Application and Activity Lifecycles 81

12.1 Android Applications and Resource Management 81
12.2 Android Process States 81
12.2.1 Foreground Process 82
12.2.2 Visible Process 82
12.2.3 Service Process 82
12.2.4 Background Process 82
12.2.5 Empty Process 83
12.3 Inter-Process Dependencies 83

12.4 The Activity Lifecycle 83
12.5 The Activity Stack 83
12.6 Activity States 84
12.7 Configuration Changes 84
12.8 Handling State Change 85
12.9 Summary 85

13. Handling Android Activity State Changes 87

13.1 New vs. Old Lifecycle Techniques 87
13.2 The Activity and Fragment Classes 87
13.3 Dynamic State vs. Persistent State 89
13.4 The Android Lifecycle Methods 89
13.5 Lifetimes 91
13.6 Foldable Devices and Multi-Resume 92
13.7 Disabling Configuration Change Restarts 92
13.8 Lifecycle Method Limitations 92
13.9 Summary 93

14. Android Activity State Changes by Example 95

14.1 Creating the State Change Example Project 95
14.2 Designing the User Interface 96
14.3 Overriding the Activity Lifecycle Methods 96
14.4 Filtering the Logcat Panel 98
14.5 Running the Application 99
14.6 Experimenting with the Activity 100
14.7 Summary 101

15. Saving and Restoring the State of an Android Activity 103

15.1 Saving Dynamic State 103
15.2 Default Saving of User Interface State 103
15.3 The Bundle Class 104
15.4 Saving the State 105
15.5 Restoring the State 106
15.6 Testing the Application 106
15.7 Summary 107

16. Understanding Android Views, View Groups and Layouts 109

16.1 Designing for Different Android Devices 109
16.2 Views and View Groups 109
16.3 Android Layout Managers 109
16.4 The View Hierarchy 111
16.5 Creating User Interfaces 112
16.6 Summary 112

17. A Guide to the Android Studio Layout Editor Tool 113

17.1 Basic vs. Empty Activity Templates 113
17.2 The Android Studio Layout Editor 117
17.3 Design Mode 117
17.4 The Palette 118
17.5 Design Mode and Layout Views 119
17.6 Code Mode 120

17.7 Split Mode 120
17.8 Setting Attributes 121
17.9 Converting Views 123
17.10 Displaying Sample Data 123
17.11 Creating a Custom Device Definition 124
17.12 Changing the Current Device 125
17.13 Multi Preview 125
17.14 Summary 126

18. A Guide to the Android ConstraintLayout 127

18.1 How ConstraintLayout Works 127
18.1.1 Constraints 127
18.1.2 Margins 128
18.1.3 Opposing Constraints 128
18.1.4 Constraint Bias 129
18.1.5 Chains 130
18.1.6 Chain Styles 130
18.2 Baseline Alignment 131
18.3 Working with Guidelines 132
18.4 Configuring Widget Dimensions 132
18.5 Working with Barriers 132
18.6 Ratios 134
18.7 ConstraintLayout Advantages 134
18.8 ConstraintLayout Availability 134
18.9 Summary 135

19. A Guide to using ConstraintLayout in Android Studio 137

19.1 Design and Layout Views 137
19.2 Autoconnect Mode 138
19.3 Inference Mode 139
19.4 Manipulating Constraints Manually 139
19.5 Adding Constraints in the Inspector 141
19.6 Viewing Constraints in the Attributes Window 141
19.7 Deleting Constraints 142
19.8 Adjusting Constraint Bias 143
19.9 Understanding ConstraintLayout Margins 143
19.10 The Importance of Opposing Constraints and Bias 145
19.11 Configuring Widget Dimensions 147
19.12 Adding Guidelines 148
19.13 Adding Barriers 149
19.14 Widget Group Alignment and Distribution 151
19.15 Converting other Layouts to ConstraintLayout 152
19.16 Summary 152

20. Working with ConstraintLayout Chains and Ratios in Android Studio 153

20.1 Creating a Chain 153
20.2 Changing the Chain Style 155
20.3 Spread Inside Chain Style 156
20.4 Packed Chain Style 156
20.5 Packed Chain Style with Bias 156
20.6 Weighted Chain 156

20.7 Working with Ratios 157
20.8 Summary 159

21. An Android Studio Layout Editor ConstraintLayout Tutorial 161

21.1 An Android Studio Layout Editor Tool Example 161
21.2 Creating a New Activity 161
21.3 Preparing the Layout Editor Environment 163
21.4 Adding the Widgets to the User Interface 164
21.5 Adding the Constraints 167
21.6 Testing the Layout 168
21.7 Using the Layout Inspector 169
21.8 Summary 169

22. Manual XML Layout Design in Android Studio 171

22.1 Manually Creating an XML Layout 171
22.2 Manual XML vs. Visual Layout Design 174
22.3 Summary 174

23. Managing Constraints using Constraint Sets 175

23.1 Java Code vs. XML Layout Files 175
23.2 Creating Views 175
23.3 View Attributes 176
23.4 Constraint Sets 176
23.4.1 Establishing Connections 176
23.4.2 Applying Constraints to a Layout 176
23.4.3 Parent Constraint Connections 176
23.4.4 Sizing Constraints 177
23.4.5 Constraint Bias 177
23.4.6 Alignment Constraints 177
23.4.7 Copying and Applying Constraint Sets 177
23.4.8 ConstraintLayout Chains 177
23.4.9 Guidelines 178
23.4.10 Removing Constraints 178
23.4.11 Scaling 178
23.4.12 Rotation 179
23.5 Summary 179

24. An Android ConstraintSet Tutorial 181

24.1 Creating the Example Project in Android Studio 181
24.2 Adding Views to an Activity 181
24.3 Setting View Attributes 182
24.4 Creating View IDs 183
24.5 Configuring the Constraint Set 184
24.6 Adding the EditText View 185
24.7 Converting Density Independent Pixels (dp) to Pixels (px) 186
24.8 Summary 187

25. A Guide to using Apply Changes in Android Studio 189

25.1 Introducing Apply Changes 189
25.2 Understanding Apply Changes Options 189
25.3 Using Apply Changes 190

25.4 Configuring Apply Changes Fallback Settings 191
25.5 An Apply Changes Tutorial 191
25.6 Using Apply Code Changes 191
25.7 Using Apply Changes and Restart Activity 192
25.8 Using Run App 192
25.9 Summary 193

26. An Overview and Example of Android Event Handling 195

26.1 Understanding Android Events 195
26.2 Using the android:onClick Resource 195
26.3 Event Listeners and Callback Methods 196
26.4 An Event Handling Example 196
26.5 Designing the User Interface 197
26.6 The Event Listener and Callback Method 198
26.7 Consuming Events 199
26.8 Summary 201

27. Android Touch and Multi-touch Event Handling 203

27.1 Intercepting Touch Events 203
27.2 The MotionEvent Object 203
27.3 Understanding Touch Actions 204
27.4 Handling Multiple Touches 204
27.5 An Example Multi-Touch Application 204
27.6 Designing the Activity User Interface 205
27.7 Implementing the Touch Event Listener 205
27.8 Running the Example Application 208
27.9 Summary 209

28. Detecting Common Gestures using the Android Gesture Detector Class 211

28.1 Implementing Common Gesture Detection 211
28.2 Creating an Example Gesture Detection Project 212
28.3 Implementing the Listener Class 212
28.4 Creating the GestureDetectorCompat Instance 215
28.5 Implementing the onTouchEvent() Method 215
28.6 Testing the Application 216
28.7 Summary 216

29. Implementing Custom Gesture and Pinch Recognition on Android 217

29.1 The Android Gesture Builder Application 217
29.2 The GestureOverlayView Class 217
29.3 Detecting Gestures 217
29.4 Identifying Specific Gestures 217
29.5 Installing and Running the Gesture Builder Application 218
29.6 Creating a Gestures File 218
29.7 Creating the Example Project 218
29.8 Extracting the Gestures File from the SD Card 219
29.9 Adding the Gestures File to the Project 219
29.10 Designing the User Interface 219
29.11 Loading the Gestures File 220
29.12 Registering the Event Listener 221
29.13 Implementing the onGesturePerformed Method 221

29.14 Testing the Application 222
29.15 Configuring the GestureOverlayView 223
29.16 Intercepting Gestures 223
29.17 Detecting Pinch Gestures 223
29.18 A Pinch Gesture Example Project 224
29.19 Summary 226

30. An Introduction to Android Fragments 227

30.1 What is a Fragment? 227
30.2 Creating a Fragment 227
30.3 Adding a Fragment to an Activity using the Layout XML File 228
30.4 Adding and Managing Fragments in Code 230
30.5 Handling Fragment Events 231
30.6 Implementing Fragment Communication 231
30.7 Summary 233

31. Using Fragments in Android Studio - An Example 235

31.1 About the Example Fragment Application 235
31.2 Creating the Example Project 235
31.3 Creating the First Fragment Layout 235
31.4 Creating the First Fragment Class 237
31.5 Creating the Second Fragment Layout 238
31.6 Adding the Fragments to the Activity 240
31.7 Making the Toolbar Fragment Talk to the Activity 241
31.8 Making the Activity Talk to the Text Fragment 245
31.9 Testing the Application 246
31.10 Summary 246

32. Modern Android App Architecture with Jetpack 247

32.1 What is Android Jetpack? 247
32.2 The "Old" Architecture 247
32.3 Modern Android Architecture 247
32.4 The ViewModel Component 248
32.5 The LiveData Component 248
32.6 ViewModel Saved State 249
32.7 LiveData and Data Binding 250
32.8 Android Lifecycles 250
32.9 Repository Modules 250
32.10 Summary 251

33. An Android Jetpack ViewModel Tutorial 253

33.1 About the Project 253
33.2 Creating the ViewModel Example Project 253
33.3 Reviewing the Project 253
33.3.1 The Main Activity 254
33.3.2 The Content Fragment 254
33.3.3 The ViewModel 255
33.4 Designing the Fragment Layout 255
33.5 Implementing the View Model 256
33.6 Associating the Fragment with the View Model 257
33.7 Modifying the Fragment 257

33.8 Accessing the ViewModel Data 259
33.9 Testing the Project 259
33.10 Summary 260

34. An Android Jetpack LiveData Tutorial 261

34.1 LiveData - A Recap 261
34.2 Adding LiveData to the ViewModel 261
34.3 Implementing the Observer 263
34.4 Summary 265

35. An Overview of Android Jetpack Data Binding 267

35.1 An Overview of Data Binding 267
35.2 The Key Components of Data Binding 267
35.2.1 The Project Build Configuration 267
35.2.2 The Data Binding Layout File 268
35.2.3 The Layout File Data Element 269
35.2.4 The Binding Classes 270
35.2.5 Data Binding Variable Configuration 270
35.2.6 Binding Expressions (One-Way) 271
35.2.7 Binding Expressions (Two-Way) 272
35.2.8 Event and Listener Bindings 272
35.3 Summary 273

36. An Android Jetpack Data Binding Tutorial 275

36.1 Removing the Redundant Code 275
36.2 Enabling Data Binding 277
36.3 Adding the Layout Element 278
36.4 Adding the Data Element to Layout File 279
36.5 Working with the Binding Class 279
36.6 Assigning the ViewModel Instance to the Data Binding Variable 280
36.7 Adding Binding Expressions 281
36.8 Adding the Conversion Method 282
36.9 Adding a Listener Binding 282
36.10 Testing the App 283
36.11 Summary 283

37. An Android ViewModel Saved State Tutorial 285

37.1 Understanding ViewModel State Saving 285
37.2 Implementing ViewModel State Saving 286
37.3 Saving and Restoring State 287
37.4 Adding Saved State Support to the ViewModelDemo Project 287
37.5 Summary 289

38. Working with Android Lifecycle-Aware Components 291

38.1 Lifecycle Awareness 291
38.2 Lifecycle Owners 291
38.3 Lifecycle Observers 292
38.4 Lifecycle States and Events 293
38.5 Summary 294

39. An Android Jetpack Lifecycle Awareness Tutorial 295

39.1 Creating the Example Lifecycle Project 295

39.2 Creating a Lifecycle Observer 295
39.3 Adding the Observer 297
39.4 Testing the Observer 297
39.5 Creating a Lifecycle Owner 298
39.6 Testing the Custom Lifecycle Owner 300
39.7 Summary 300

40. An Overview of the Navigation Architecture Component 301

40.1 Understanding Navigation 301
40.2 Declaring a Navigation Host 302
40.3 The Navigation Graph 304
40.4 Accessing the Navigation Controller 305
40.5 Triggering a Navigation Action 305
40.6 Passing Arguments 306
40.7 Summary 306

41. An Android Jetpack Navigation Component Tutorial 307

41.1 Creating the NavigationDemo Project 307
41.2 Adding Navigation to the Build Configuration 307
41.3 Creating the Navigation Graph Resource File 308
41.4 Declaring a Navigation Host 309
41.5 Adding Navigation Destinations 311
41.6 Designing the Destination Fragment Layouts 313
41.7 Adding an Action to the Navigation Graph 314
41.8 Implement the OnFragmentInteractionListener 315
41.9 Triggering the Action 316
41.10 Passing Data Using Safeargs 317
41.11 Summary 320

42. Creating and Managing Overflow Menus on Android 321

42.1 The Overflow Menu 321
42.2 Creating an Overflow Menu 321
42.3 Displaying an Overflow Menu 322
42.4 Responding to Menu Item Selections 323
42.5 Creating Checkable Item Groups 323
42.6 Menus and the Android Studio Menu Editor 324
42.7 Creating the Example Project 325
42.8 Designing the Menu 325
42.9 Modifying the onOptionsItemSelected() Method 328
42.10 Testing the Application 329
42.11 Summary 329

43. Animating User Interfaces with the Android Transitions Framework 331

43.1 Introducing Android Transitions and Scenes 331
43.2 Using Interpolators with Transitions 332
43.3 Working with Scene Transitions 332
43.4 Custom Transitions and TransitionSets in Code 334
43.5 Custom Transitions and TransitionSets in XML 334
43.6 Working with Interpolators 336
43.7 Creating a Custom Interpolator 337
43.8 Using the beginDelayedTransition Method 338

43.9 Summary 338

44. An Android Transition Tutorial using beginDelayedTransition 339

44.1 Creating the Android Studio TransitionDemo Project 339
44.2 Preparing the Project Files 339
44.3 Implementing beginDelayedTransition Animation 339
44.4 Customizing the Transition 342
44.5 Summary 343

45. Implementing Android Scene Transitions – A Tutorial 345

45.1 An Overview of the Scene Transition Project 345
45.2 Creating the Android Studio SceneTransitions Project 345
45.3 Identifying and Preparing the Root Container 345
45.4 Designing the First Scene 345
45.5 Designing the Second Scene 346
45.6 Entering the First Scene 347
45.7 Loading Scene 2 348
45.8 Implementing the Transitions 349
45.9 Adding the Transition File 349
45.10 Loading and Using the Transition Set 350
45.11 Configuring Additional Transitions 351
45.12 Summary 351

46. Working with the Floating Action Button and Snackbar 353

46.1 The Material Design 353
46.2 The Design Library 353
46.3 The Floating Action Button (FAB) 353
46.4 The Snackbar 354
46.5 Creating the Example Project 355
46.6 Reviewing the Project 355
46.7 Removing Navigation Features 356
46.8 Changing the Floating Action Button 357
46.9 Adding the ListView to the Content Layout 358
46.10 Adding Items to the ListView 358
46.11 Adding an Action to the Snackbar 361
46.12 Summary 362

47. Creating a Tabbed Interface using the TabLayout Component 363

47.1 An Introduction to the ViewPager 363
47.2 An Overview of the TabLayout Component 363
47.3 Creating the TabLayoutDemo Project 364
47.4 Creating the First Fragment 364
47.5 Duplicating the Fragments 365
47.6 Adding the TabLayout and ViewPager 366
47.7 Creating the Pager Adapter 367
47.8 Performing the Initialization Tasks 368
47.9 Testing the Application 371
47.10 Customizing the TabLayout 372
47.11 Displaying Icon Tab Items 373
47.12 Summary 374

48. Working with the RecyclerView and CardView Widgets 375

48.1 An Overview of the RecyclerView 375
48.2 An Overview of the CardView 377
48.3 Summary 378

49. An Android RecyclerView and CardView Tutorial 379

49.1 Creating the CardDemo Project 379
49.2 Modifying the Basic Activity Project 379
49.3 Designing the CardView Layout 380
49.4 Adding the RecyclerView 381
49.5 Creating the RecyclerView Adapter 381
49.6 Adding the Image Files 383
49.7 Initializing the RecyclerView Component 384
49.8 Testing the Application 385
49.9 Responding to Card Selections 385
49.10 Summary 387

50. A Layout Editor Sample Data Tutorial 389

50.1 Adding Sample Data to a Project 389
50.2 Using Custom Sample Data 393
50.3 Summary 395

51. Working with the AppBar and Collapsing Toolbar Layouts 397

51.1 The Anatomy of an AppBar 397
51.2 The Example Project 398
51.3 Coordinating the RecyclerView and Toolbar 398
51.4 Introducing the Collapsing Toolbar Layout 400
51.5 Changing the Title and Scrim Color 403
51.6 Summary 404

52. An Android Studio Master/Detail Flow Tutorial 405

52.1 The Master/Detail Flow 405
52.2 Creating a Master/Detail Flow Activity 406
52.3 The Anatomy of the Master/Detail Flow Template 407
52.4 Modifying the Master/Detail Flow Template 408
52.5 Changing the Content Model 408
52.6 Changing the Detail Pane 410
52.7 Modifying the WebsiteDetailFragment Class 411
52.8 Modifying the WebsiteListActivity Class 412
52.9 Adding Manifest Permissions 413
52.10 Running the Application 413
52.11 Summary 414

53. An Overview of Android Intents 415

53.1 An Overview of Intents 415
53.2 Explicit Intents 415
53.3 Returning Data from an Activity 416
53.4 Implicit Intents 417
53.5 Using Intent Filters 418
53.6 Checking Intent Availability 419
53.7 Summary 419

54. Android Explicit Intents – A Worked Example 421

54.1 Creating the Explicit Intent Example Application 421
54.2 Designing the User Interface Layout for MainActivity 421
54.3 Creating the Second Activity Class 422
54.4 Designing the User Interface Layout for ActivityB 423
54.5 Reviewing the Application Manifest File 423
54.6 Creating the Intent 424
54.7 Extracting Intent Data 425
54.8 Launching ActivityB as a Sub-Activity 426
54.9 Returning Data from a Sub-Activity 427
54.10 Testing the Application 427
54.11 Summary 427

55. Android Implicit Intents – A Worked Example 429

55.1 Creating the Android Studio Implicit Intent Example Project 429
55.2 Designing the User Interface 429
55.3 Creating the Implicit Intent 430
55.4 Adding a Second Matching Activity 430
55.5 Adding the Web View to the UI 431
55.6 Obtaining the Intent URL 431
55.7 Modifying the MyWebView Project Manifest File 432
55.8 Installing the MyWebView Package on a Device 434
55.9 Testing the Application 434
55.10 Summary 435

56. Android Broadcast Intents and Broadcast Receivers 437

56.1 An Overview of Broadcast Intents 437
56.2 An Overview of Broadcast Receivers 438
56.3 Obtaining Results from a Broadcast 439
56.4 Sticky Broadcast Intents 439
56.5 The Broadcast Intent Example 440
56.6 Creating the Example Application 440
56.7 Creating and Sending the Broadcast Intent 440
56.8 Creating the Broadcast Receiver 441
56.9 Registering the Broadcast Receiver 442
56.10 Testing the Broadcast Example 443
56.11 Listening for System Broadcasts 443
56.12 Summary 444

57. A Basic Overview of Threads and AsyncTasks 445

57.1 An Overview of Threads 445
57.2 The Application Main Thread 445
57.3 Thread Handlers 445
57.4 A Basic AsyncTask Example 445
57.5 Subclassing AsyncTask 447
57.6 Testing the App 450
57.7 Canceling a Task 450
57.8 Summary 451

58. An Overview of Android Started and Bound Services 453

58.1 Started Services 453
58.2 Intent Service 453

58.3 Bound Service 454
58.4 The Anatomy of a Service 454
58.5 Controlling Destroyed Service Restart Options 455
58.6 Declaring a Service in the Manifest File 455
58.7 Starting a Service Running on System Startup 456
58.8 Summary 456

59. Implementing an Android Started Service – A Worked Example 457

59.1 Creating the Example Project 457
59.2 Creating the Service Class 457
59.3 Adding the Service to the Manifest File 458
59.4 Starting the Service 459
59.5 Testing the IntentService Example 460
59.6 Using the Service Class 460
59.7 Creating the New Service 460
59.8 Modifying the User Interface 462
59.9 Running the Application 463
59.10 Creating an AsyncTask for Service Tasks 463
59.11 Summary 465

60. Android Local Bound Services – A Worked Example 467

60.1 Understanding Bound Services 467
60.2 Bound Service Interaction Options 467
60.3 An Android Studio Local Bound Service Example 467
60.4 Adding a Bound Service to the Project 468
60.5 Implementing the Binder 468
60.6 Binding the Client to the Service 471
60.7 Completing the Example 472
60.8 Testing the Application 473
60.9 Summary 473

61. Android Remote Bound Services – A Worked Example 475

61.1 Client to Remote Service Communication 475
61.2 Creating the Example Application 475
61.3 Designing the User Interface 475
61.4 Implementing the Remote Bound Service 476
61.5 Configuring a Remote Service in the Manifest File 477
61.6 Launching and Binding to the Remote Service 478
61.7 Sending a Message to the Remote Service 479
61.8 Summary 480

62. An Android Notifications Tutorial 481

62.1 An Overview of Notifications 481
62.2 Creating the NotifyDemo Project 483
62.3 Designing the User Interface 483
62.4 Creating the Second Activity 483
62.5 Creating a Notification Channel 484
62.6 Creating and Issuing a Basic Notification 486
62.7 Launching an Activity from a Notification 489
62.8 Adding Actions to a Notification 490
62.9 Bundled Notifications 491

62.10 Summary 493

63. An Android Direct Reply Notification Tutorial 495

63.1 Creating the DirectReply Project 495
63.2 Designing the User Interface 495
63.3 Creating the Notification Channel 496
63.4 Building the RemoteInput Object 497
63.5 Creating the PendingIntent 498
63.6 Creating the Reply Action 498
63.7 Receiving Direct Reply Input 501
63.8 Updating the Notification 502
63.9 Summary 503

64. Foldable Devices and Multi-Window Support 505

64.1 Foldables and Multi-Window Support 505
64.2 Using a Foldable Emulator 506
64.3 Entering Multi-Window Mode 507
64.4 Enabling and using Freeform Support 508
64.5 Checking for Freeform Support 508
64.6 Enabling Multi-Window Support in an App 508
64.7 Specifying Multi-Window Attributes 509
64.8 Detecting Multi-Window Mode in an Activity 510
64.9 Receiving Multi-Window Notifications 510
64.10 Launching an Activity in Multi-Window Mode 511
64.11 Configuring Freeform Activity Size and Position 511
64.12 Summary 512

65. An Overview of Android SQLite Databases 513

65.1 Understanding Database Tables 513
65.2 Introducing Database Schema 513
65.3 Columns and Data Types 513
65.4 Database Rows 514
65.5 Introducing Primary Keys 514
65.6 What is SQLite? 514
65.7 Structured Query Language (SQL) 514
65.8 Trying SQLite on an Android Virtual Device (AVD) 515
65.9 The Android Room Persistence Library 517
65.10 Summary 517

66. The Android Room Persistence Library 519

66.1 Revisiting Modern App Architecture 519
66.2 Key Elements of Room Database Persistence 519
66.2.1 Repository 520
66.2.2 Room Database 520
66.2.3 Data Access Object (DAO) 520
66.2.4 Entities 520
66.2.5 SQLite Database 520
66.3 Understanding Entities 521
66.4 Data Access Objects 524
66.5 The Room Database 525
66.6 The Repository 526

66.7 In-Memory Databases 527
66.8 Summary 527

67. An Android TableLayout and TableRow Tutorial 529

67.1 The TableLayout and TableRow Layout Views 529
67.2 Creating the Room Database Project 530
67.3 Converting to a LinearLayout 530
67.4 Adding the TableLayout to the User Interface 531
67.5 Configuring the TableRows 532
67.6 Adding the Button Bar to the Layout 533
67.7 Adding the RecyclerView 534
67.8 Adjusting the Layout Margins 535
67.9 Summary 535

68. An Android Room Database and Repository Tutorial 537

68.1 About the RoomDemo Project 537
68.2 Modifying the Build Configuration 537
68.3 Building the Entity 537
68.4 Creating the Data Access Object 539
68.5 Adding the Room Database 541
68.6 Adding the Repository 542
68.7 Modifying the ViewModel 546
68.8 Creating the Product Item Layout 547
68.9 Adding the RecyclerView Adapter 547
68.10 Preparing the Main Fragment 549
68.11 Adding the Button Listeners 550
68.12 Adding LiveData Observers 551
68.13 Initializing the RecyclerView 552
68.14 Testing the RoomDemo App 553
68.15 Summary 553

69. Accessing Cloud Storage using the Android Storage Access Framework 555

69.1 The Storage Access Framework 555
69.2 Working with the Storage Access Framework 556
69.3 Filtering Picker File Listings 556
69.4 Handling Intent Results 557
69.5 Reading the Content of a File 558
69.6 Writing Content to a File 558
69.7 Deleting a File 559
69.8 Gaining Persistent Access to a File 559
69.9 Summary 560

70. An Android Storage Access Framework Example 561

70.1 About the Storage Access Framework Example 561
70.2 Creating the Storage Access Framework Example 561
70.3 Designing the User Interface 561
70.4 Declaring Request Codes 562
70.5 Creating a New Storage File 563
70.6 The onActivityResult() Method 564
70.7 Saving to a Storage File 566
70.8 Opening and Reading a Storage File 568

70.9 Testing the Storage Access Application 570
70.10 Summary 571

71. Implementing Video Playback on Android using the VideoView and MediaController Classes 573

71.1 Introducing the Android VideoView Class 573
71.2 Introducing the Android MediaController Class 574
71.3 Creating the Video Playback Example 574
71.4 Designing the VideoPlayer Layout 574
71.5 Downloading the Video File 575
71.6 Configuring the VideoView 576
71.7 Adding the MediaController to the Video View 577
71.8 Setting up the onPreparedListener 578
71.9 Summary 579

72. Android Picture-in-Picture Mode 581

72.1 Picture-in-Picture Features 581
72.2 Enabling Picture-in-Picture Mode 582
72.3 Configuring Picture-in-Picture Parameters 582
72.4 Entering Picture-in-Picture Mode 583
72.5 Detecting Picture-in-Picture Mode Changes 583
72.6 Adding Picture-in-Picture Actions 584
72.7 Summary 584

73. An Android Picture-in-Picture Tutorial 587

73.1 Adding Picture-in-Picture Support to the Manifest 587
73.2 Adding a Picture-in-Picture Button 587
73.3 Entering Picture-in-Picture Mode 587
73.4 Detecting Picture-in-Picture Mode Changes 589
73.5 Adding a Broadcast Receiver 590
73.6 Adding the PiP Action 591
73.7 Testing the Picture-in-Picture Action 594
73.8 Summary 594

74. Making Runtime Permission Requests in Android 595

74.1 Understanding Normal and Dangerous Permissions 595
74.2 Creating the Permissions Example Project 597
74.3 Checking for a Permission 597
74.4 Requesting Permission at Runtime 599
74.5 Providing a Rationale for the Permission Request 600
74.6 Testing the Permissions App 602
74.7 Summary 602

75. Android Audio Recording and Playback using MediaPlayer and MediaRecorder 603

75.1 Playing Audio 603
75.2 Recording Audio and Video using the MediaRecorder Class 604
75.3 About the Example Project 605
75.4 Creating the AudioApp Project 605
75.5 Designing the User Interface 605
75.6 Checking for Microphone Availability 606
75.7 Performing the Activity Initialization 607
75.8 Implementing the recordAudio() Method 608

75.9 Implementing the stopAudio() Method 609
75.10 Implementing the playAudio() method 609
75.11 Configuring and Requesting Permissions 610
75.12 Testing the Application 612
75.13 Summary 613

76. Working with the Google Maps Android API in Android Studio 615

76.1 The Elements of the Google Maps Android API 615
76.2 Creating the Google Maps Project 616
76.3 Obtaining Your Developer Signature 616
76.4 Adding the Apache HTTP Legacy Library Requirement 617
76.5 Testing the Application 617
76.6 Understanding Geocoding and Reverse Geocoding 618
76.7 Adding a Map to an Application 620
76.8 Requesting Current Location Permission 620
76.9 Displaying the User's Current Location 621
76.10 Changing the Map Type 622
76.11 Displaying Map Controls to the User 623
76.12 Handling Map Gesture Interaction 624
76.12.1 Map Zooming Gestures 624
76.12.2 Map Scrolling/Panning Gestures 624
76.12.3 Map Tilt Gestures 625
76.12.4 Map Rotation Gestures 625
76.13 Creating Map Markers 625
76.14 Controlling the Map Camera 626
76.15 Summary 627

77. Printing with the Android Printing Framework 629

77.1 The Android Printing Architecture 629
77.2 The Print Service Plugins 629
77.3 Google Cloud Print 630
77.4 Printing to Google Drive 630
77.5 Save as PDF 631
77.6 Printing from Android Devices 631
77.7 Options for Building Print Support into Android Apps 632
77.7.1 Image Printing 632
77.7.2 Creating and Printing HTML Content 633
77.7.3 Printing a Web Page 634
77.7.4 Printing a Custom Document 635
77.8 Summary 635

78. An Android HTML and Web Content Printing Example 637

78.1 Creating the HTML Printing Example Application 637
78.2 Printing Dynamic HTML Content 637
78.3 Creating the Web Page Printing Example 640
78.4 Removing the Floating Action Button 640
78.5 Removing Navigation Features 640
78.6 Designing the User Interface Layout 641
78.7 Loading the Web Page into the WebView 642
78.8 Adding the Print Menu Option 643
78.9 Summary 645

79. A Guide to Android Custom Document Printing 647

79.1 An Overview of Android Custom Document Printing 647
79.1.1 Custom Print Adapters 647
79.2 Preparing the Custom Document Printing Project 648
79.3 Creating the Custom Print Adapter 649
79.4 Implementing the onLayout() Callback Method 650
79.5 Implementing the onWrite() Callback Method 653
79.6 Checking a Page is in Range 655
79.7 Drawing the Content on the Page Canvas 656
79.8 Starting the Print Job 658
79.9 Testing the Application 659
79.10 Summary 659

80. An Introduction to Android App Links 661

80.1 An Overview of Android App Links 661
80.2 App Link Intent Filters 661
80.3 Handling App Link Intents 662
80.4 Associating the App with a Website 662
80.5 Summary 663

81. An Android Studio App Links Tutorial 665

81.1 About the Example App 665
81.2 The Database Schema 665
81.3 Loading and Running the Project 666
81.4 Adding the URL Mapping 667
81.5 Adding the Intent Filter 670
81.6 Adding Intent Handling Code 670
81.7 Testing the App Link 674
81.8 Associating an App Link with a Web Site 674
81.9 Summary 676

82. A Guide to the Android Studio Profiler 677

82.1 Accessing the Android Profiler 677
82.2 Enabling Advanced Profiling 677
82.3 The Android Profiler Tool Window 678
82.4 The Sessions Panel 679
82.5 The CPU Profiler 680
82.6 Memory Profiler 683
82.7 Network Profiler 684
82.8 Energy Profiler 686
82.9 Summary 686

83. An Android Biometric Authentication Tutorial 687

83.1 An Overview of Biometric Authentication 687
83.2 Creating the Biometric Authentication Project 687
83.3 Configuring Device Fingerprint Authentication 688
83.4 Adding the Biometric Permission to the Manifest File 688
83.5 Designing the User Interface 689
83.6 Adding a Toast Convenience Method 689
83.7 Checking the Security Settings 690
83.8 Configuring the Authentication Callbacks 691

83.9 Adding the CancellationSignal 692
83.10 Starting the Biometric Prompt 693
83.11 Testing the Project 693
83.12 Summary 695

84. Creating, Testing and Uploading an Android App Bundle 697

84.1 The Release Preparation Process 697
84.2 Android App Bundles 697
84.3 Register for a Google Play Developer Console Account 698
84.4 Configuring the App in the Console 699
84.5 Enabling Google Play App Signing 699
84.6 Creating a Keystore File 700
84.7 Creating the Android App Bundle 702
84.8 Generating Test APK Files 703
84.9 Uploading the App Bundle to the Google Play Developer Console 704
84.10 Exploring the App Bundle 704
84.11 Managing Testers 706
84.12 Uploading New App Bundle Revisions 707
84.13 Analyzing the App Bundle File 708
84.14 Enabling Google Play Signing for an Existing App 709
84.15 Summary 710

85. An Overview of Android Dynamic Feature Modules 711

85.1 An Overview of Dynamic Feature Modules 711
85.2 Dynamic Feature Module Architecture 711
85.3 Creating a Dynamic Feature Module 712
85.4 Converting an Existing Module for Dynamic Delivery 714
85.5 Working with Dynamic Feature Modules 717
85.6 Handling Large Dynamic Feature Modules 719
85.7 Summary 720

86. An Android Studio Dynamic Feature Tutorial 721

86.1 Creating the DynamicFeature Project 721
86.2 Adding Dynamic Feature Support to the Project 721
86.3 Designing the Base Activity User Interface 722
86.4 Adding the Dynamic Feature Module 723
86.5 Reviewing the Dynamic Feature Module 724
86.6 Adding the Dynamic Feature Activity 726
86.7 Implementing the launchIntent() Method 728
86.8 Uploading the App Bundle for Testing 729
86.9 Implementing the installFeature() Method 730
86.10 Adding the Update Listener 732
86.11 Handling Large Downloads 735
86.12 Using Deferred Installation 736
86.13 Removing a Dynamic Module 737
86.14 Summary 737

87. An Overview of Gradle in Android Studio 739

87.1 An Overview of Gradle 739
87.2 Gradle and Android Studio 739
87.2.1 Sensible Defaults 739

87.2.2 Dependencies 739
87.2.3 Build Variants 740
87.2.4 Manifest Entries 740
87.2.5 APK Signing 740
87.2.6 ProGuard Support 740
87.3 The Top-level Gradle Build File 740
87.4 Module Level Gradle Build Files 742
87.5 Configuring Signing Settings in the Build File 744
87.6 Running Gradle Tasks from the Command-line 745
87.7 Summary 745
Index 747

Chapter 1

1. Introduction

In 2018 Google introduced Android Jetpack to the developer community. Designed to make it quicker and easier to develop modern and reliable Android apps, Jetpack consists of a set of tools, libraries and architectural guidelines. The main elements of Android Jetpack consist of the Android Studio Integrated Development Environment (IDE), the Android Architecture Components and the Modern App Architecture Guidelines, all of which are covered in this latest edition of Android Studio Development Essentials.

Fully updated for Android Studio 3.6 and Android 10 (Q), the goal of this book is to teach the skills necessary to develop Android based applications using the Java programming language.

Beginning with the basics, this book provides an outline of the steps necessary to set up an Android development and testing environment. An overview of Android Studio is included covering 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, 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 edition of the book also covers printing, transitions, cloud-based file storage 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.

In addition to covering general Android development techniques, the book also includes Google Play specific topics such as implementing maps using the Google Maps Android API, and submitting apps to the Google Play Developer Console.

Other key features of Android Studio 3.6 and Android 10 are also covered in detail including the Layout Editor, the ConstraintLayout and ConstraintSet classes, view binding, constraint chains, barriers and direct reply notifications.

Chapters also cover advanced features of Android Studio such as App Links, Dynamic Delivery, the Android Studio Profiler and Gradle build configuration.

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 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/androidstudio36/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, select the Open an existing Android Studio project 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/androidstudio36.html

In the event that 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.

1.4 Download the eBook

Thank you for purchasing the print edition of this book. If you would like to download the PDF version of this book, please email proof of purchase (for example a receipt, delivery notice or photo of the physical book) to *feedback@ebookfrenzy.com* and we will provide you with a download link for the book in PDF format.

Chapter 2

2. Setting up an Android Studio Development Environment

Before any work can begin on the development of an Android application, the first step is to configure a computer system to act as the development platform. This involves a number of steps consisting of installing the Android Studio Integrated Development Environment (IDE) which also includes the Android Software Development Kit (SDK) and 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 7/8/10 (32-bit or 64-bit though the Android emulator will only run on 64-bit systems)
- macOS 10.10 or later (Intel based systems only)
- ChromeOS device with Intel i5 or higher and minimum 8GB of RAM
- Linux systems with version 2.19 or later of GNU C Library (glibc)
- Minimum of 4GB of RAM (8GB is preferred)
- Approximately 4GB 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 version 3.6 using the Android 10.0 (Q) API 29 SDK which, at the time writing are the current versions.

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 it is important to note that there may be some minor differences between this book and the software. A web search for Android Studio 3.6 should provide the option to download the older version in the event that these differences become a problem. Alternatively, visit the following web page to find Android Studio 3.6 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 being performed.

2.3.1 Installation on Windows

Locate the downloaded Android Studio installation executable file (named *android-studio-ide-<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 users of the system. When prompted to select the components to install, make sure 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 on 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 task bar for easy access by navigating to the *Android Studio\bin* directory, right-clicking on the executable and selecting the *Pin to Taskbar* menu option. Note that the executable is provided in 32-bit (*studio*) and 64-bit (*studio64*) executable versions. If you are running a 32-bit system be sure to use the *studio* executable.

2.3.2 Installation on macOS

Android Studio for macOS is downloaded in the form of a disk image (.dmg) file. Once the *android-studio-ide-<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:

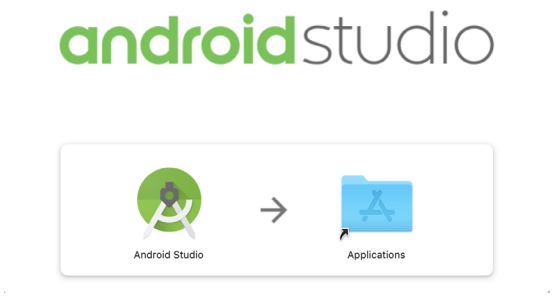

Figure 2-1

To install the package, simply 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 which will typically take a few minutes 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:

```
unzip /<path to package>/android-studio-ide-<version>-linux.zip
```

Note that the Android Studio bundle will be installed into a sub-directory named *android-studio*. Assuming, therefore, 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
```

When running on a 64-bit Linux system, it will be necessary to install some 32-bit support libraries before Android Studio will run. On Ubuntu these libraries can be installed using the following command:

```
sudo apt-get install libc6:i386 libncurses5:i386 libstdc++6:i386 lib32z1 libbz2-1.0:i386
```

On Red Hat and Fedora based 64-bit systems, use the following command:

```
sudo yum install zlib.i686 ncurses-libs.i686 bzip2-libs.i686
```

2.4 The Android Studio Setup Wizard

The first time that Android Studio is launched after being installed, a dialog will 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 on the OK button to proceed.

Next, the setup wizard may appear as shown in Figure 2-2 though this dialog does not appear on all platforms:

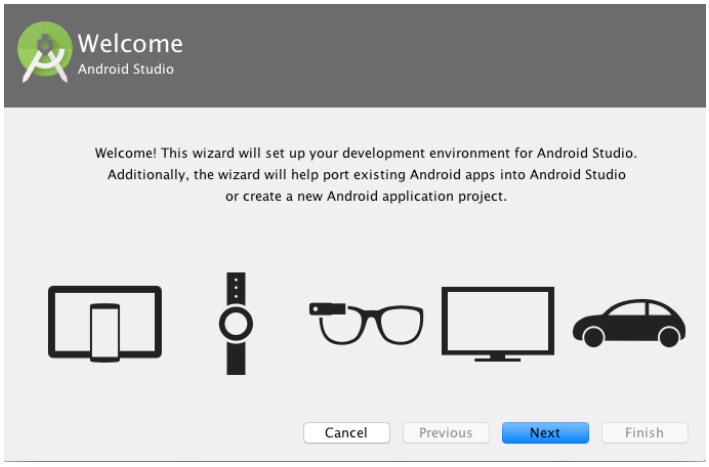

Figure 2-2

If the wizard appears, click on the Next button, choose the Standard installation option and click on Next once again.

Android Studio will proceed to download and configure the latest Android SDK and some additional components

and packages. Once this process has completed, click on the *Finish* button in the *Downloading Components* dialog at which point the Welcome to Android Studio screen should then appear:

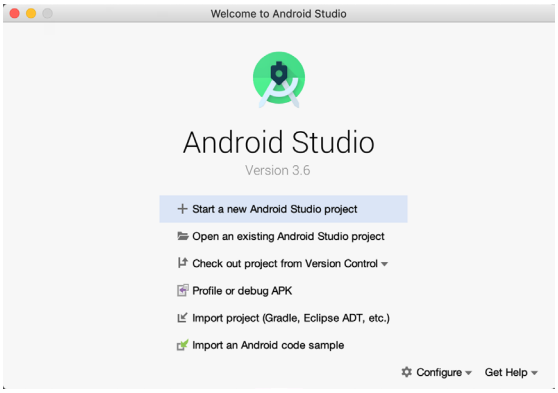

Figure 2-3

2.5 Installing Additional Android SDK Packages

The steps performed so far have installed Java, 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 using the *Android SDK Settings* screen, which may be launched from within the Android Studio tool by selecting the *Configure -> SDK Manager* option from within the Android Studio welcome dialog. Once invoked, the *Android SDK* screen of the default settings dialog will appear as shown in Figure 2-4:

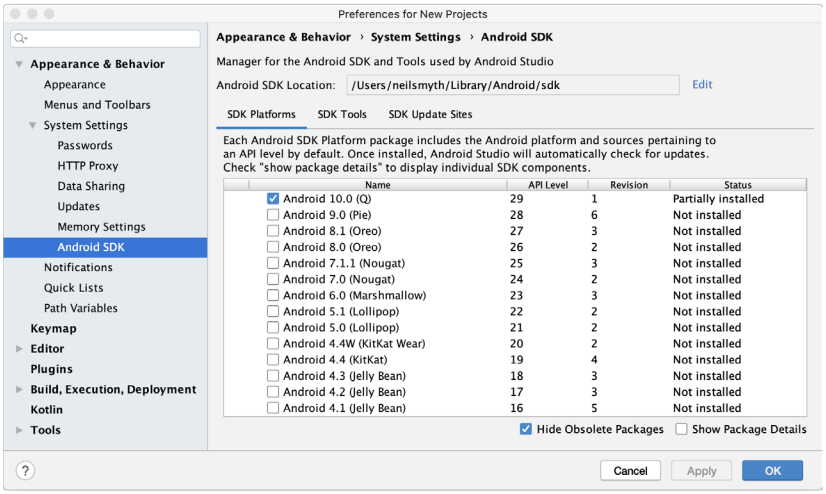

Figure 2-4

Immediately after installing Android Studio for the first time it is likely that only the latest released version of the Android SDK has been installed. To install older versions of the Android SDK simply select the checkboxes corresponding to the versions and click on the *Apply* button.

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 available for update, 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 2-5:

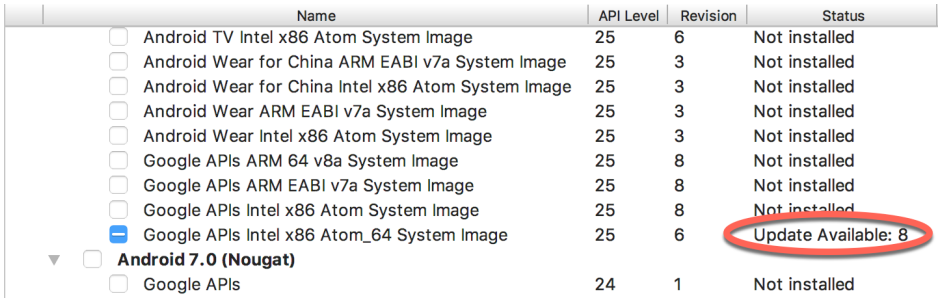

Figure 2-5

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 on the *Apply* button.

In addition to the Android SDK packages, a number of 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 2-6:

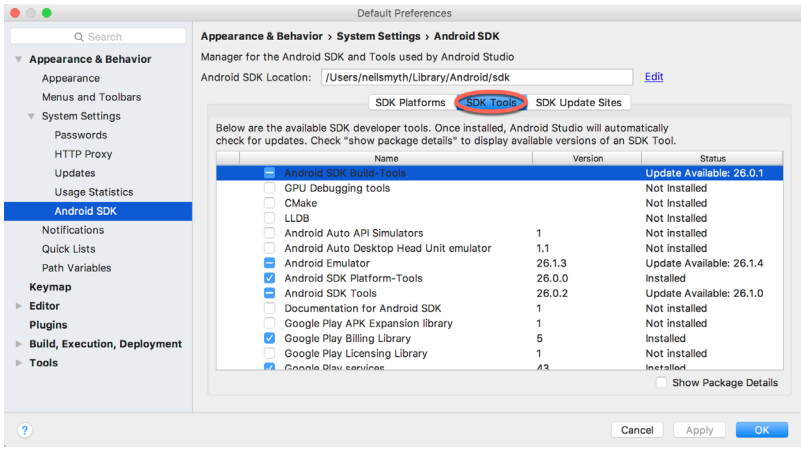

Figure 2-6

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
- Android SDK Tools
- Google Play Services
- Intel x86 Emulator Accelerator (HAXM installer)
- Google USB Driver (Windows only)

In the event that any of the above packages are listed as *Not Installed* or requiring an update, simply select the checkboxes next to those packages and click on the *Apply* button to initiate the installation process.

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

2.6 Making the Android SDK Tools Command-line Accessible

Most of the time, the underlying tools of the Android SDK will be accessed from within the Android Studio environment. That being said, however, there will also be instances where it will be useful to be able to invoke those tools from a command prompt or terminal window. In order 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 operating system, the PATH variable needs to be configured to include the following paths (where *<path_to_android_sdk_installation>* represents the file system location into which the Android SDK was installed):

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

The location of the SDK on your system can be identified 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 2-7:

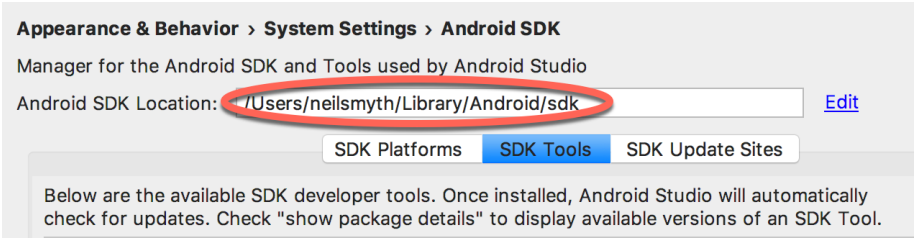

Figure 2-7

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 7

1. Right-click on Computer in the desktop start menu and select Properties from the resulting menu.
2. In the properties panel, select the Advanced System Settings link and, in the resulting dialog, click on the Environment Variables… button.
3. In the Environment Variables dialog, locate the Path variable in the System variables list, select it and click on the *Edit…* button. Using the *New* button in the edit dialog, add three 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\platform-tools
C:\Users\demo\AppData\Local\Android\Sdk\tools
C:\Users\demo\AppData\Local\Android\Sdk\tools\bin
```

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

Once the above steps are complete, verify that the path is correctly set by opening a *Command Prompt* window (*Start -> All Programs -> Accessories -> Command Prompt*) and at the prompt 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 launch the AVD Manager command line tool (don't worry if the avdmanager tool reports a problem with Java - this will be addressed later):

```
avdmanager
```

In the event that 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 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. Follow the steps outlined for Windows 7 starting from step 2 through to step 4.

Open the command prompt window (move the mouse to the bottom right-hand corner of the screen, select the Search option and enter *cmd* into the search box). Select *Command Prompt* from the search results.

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
```

In the event that 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.
```

2.6.3 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 7 starting from step 3.

2.6.4 Linux

On Linux, this configuration can typically be achieved 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/tools:/
home/demo/Android/sdk/tools/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

A number of 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/tools
/Users/demo/Library/Android/sdk/tools/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 that consists of 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. Part of 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:

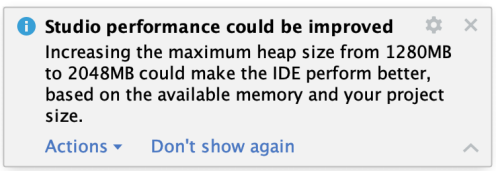

Figure 2-8

To view and modify the current memory configuration, select the *File -> Settings...* (*Android Studio -> Preferences...* on macOS) menu option and, in the resulting dialog, select the *Memory Settings* option listed under *System Settings* in the left-hand navigation panel as illustrated in Figure 2-9 below.

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.

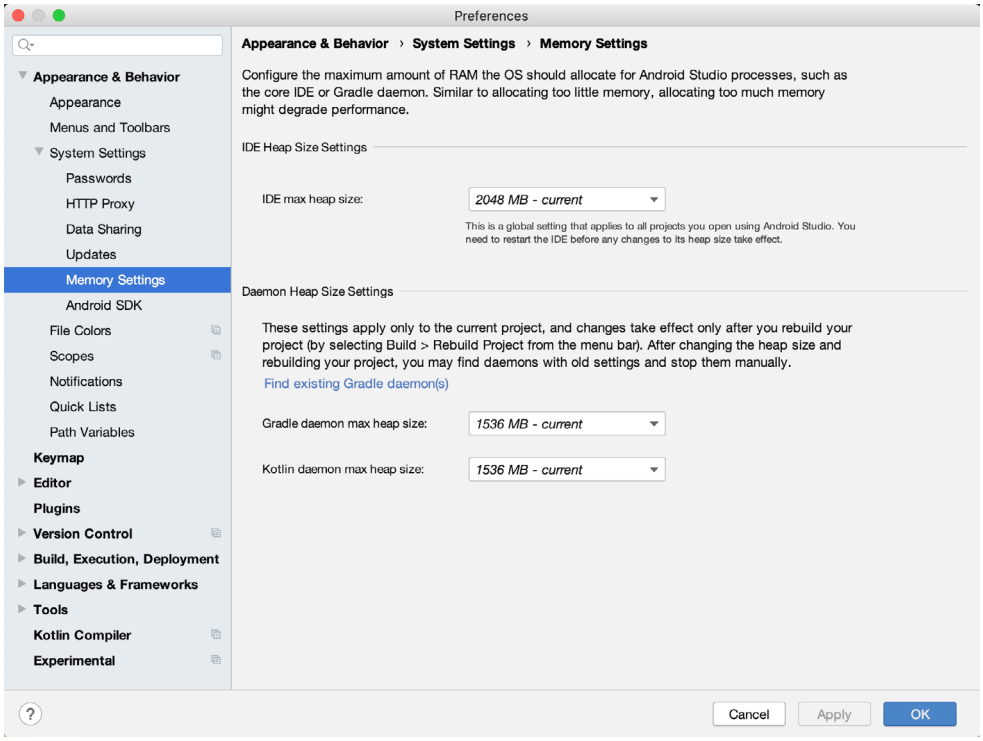

Figure 2-9

The IDE memory setting adjusts the memory allocated to Android Studio and applies regardless of the currently loaded project. When a project is built and run from within Android Studio, on the other hand, a number of 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 may potentially be improved by adjusting the daemon heap settings. Unlike the IDE heap settings, these settings apply only to the current project.

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, click on the *Configure -> Check for Updates* menu option within the Android Studio welcome screen, or use the *Help -> Check for Updates...* (*Android Studio -> Check for Updates...* on macOS) menu option accessible from within the Android Studio main window.

2.9 Summary

Prior to 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). In this chapter, we have covered the steps necessary to install these packages on Windows, macOS and Linux.

Chapter 3

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 the development of Android applications using the Android Studio IDE. Before moving on to slightly more advanced topics, now is a good time to validate that all of the 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 the creation of a simple Android application project using Android Studio. Once the project has been created, a later chapter will explore the use of 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 very simple 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 make use of one of the most basic of Android Studio project templates. This simplicity allows us to introduce some of the key aspects of Android app development without overwhelming the beginner by trying to introduce 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 all of the techniques and code used in this initial example project will be covered in much greater detail in later chapters.

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:

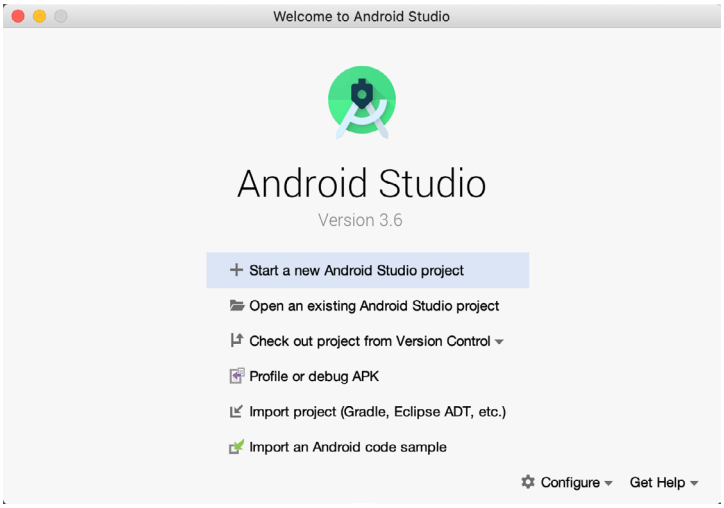

Figure 3-1

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

3.3 Creating an Activity

The first step is to define the type of initial activity that is to be created for the application. Options are available to create projects for Phone and Tablet, Wear OS, TV, Android Audio or Android Things. A range of different activity types is available when developing Android applications, many of which will be covered extensively in later chapters. For the purposes of this example, however, simply select the option to create an *Empty Activity* on the Phone and Tablet screen. The Empty Activity option creates a template user interface consisting of a single TextView object.

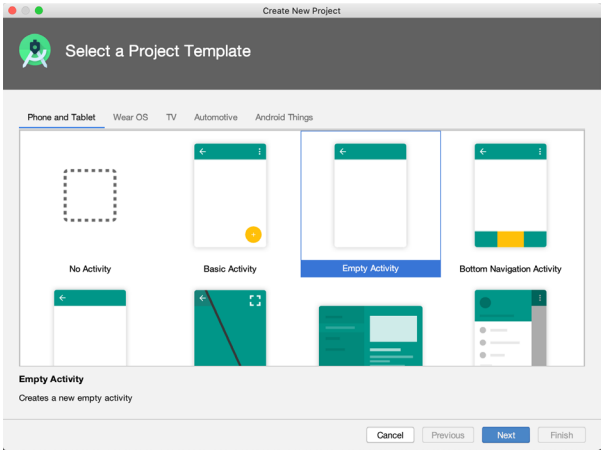

Figure 3-2

With the Empty 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* is used to uniquely identify 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 name of the application. 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 the purposes of 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: Android 8.0 (Oreo). This is the SDK that will be used in most of the

projects created in this book unless a necessary feature is only available in a more recent version.

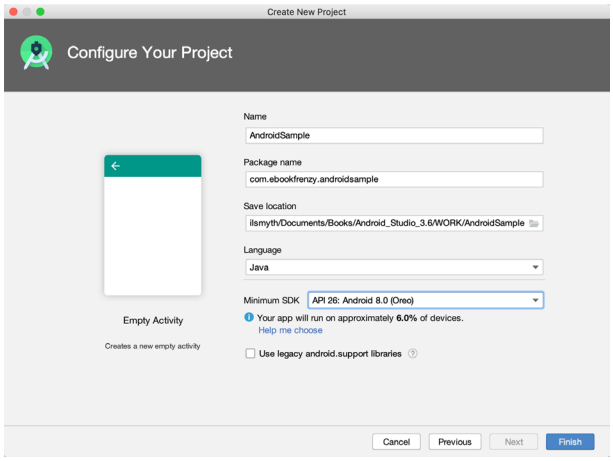

Figure 3-3

Finally, change the *Language* menu to *Java* and click on *Finish* to initiate the project creation process.

3.5 Modifying the Example Application

At this point, Android Studio has created a minimal example application project and opened the main window.

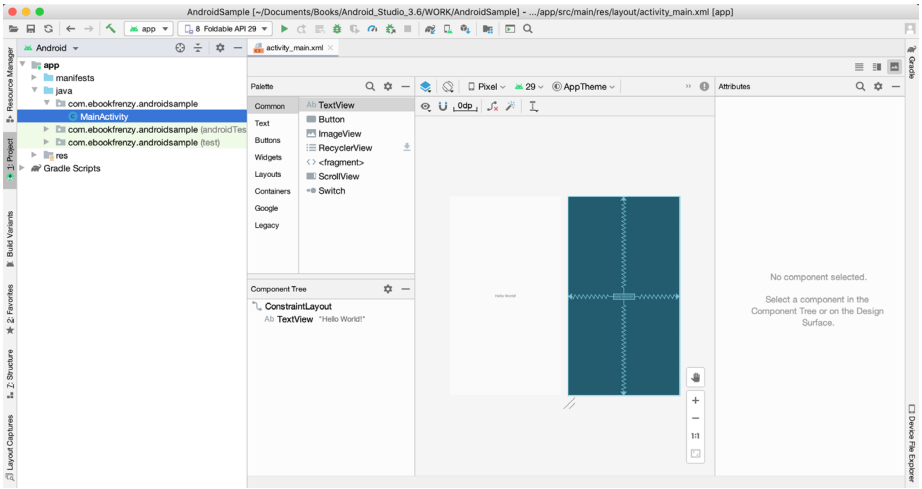

Figure 3-4

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

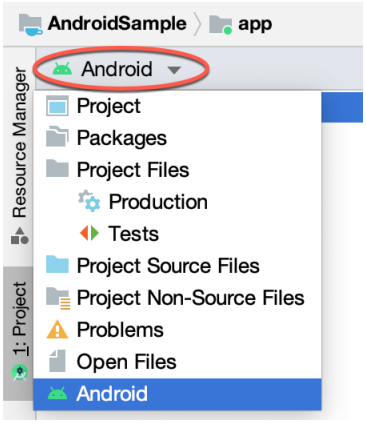

Figure 3-5

3.6 Modifying the User Interface

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

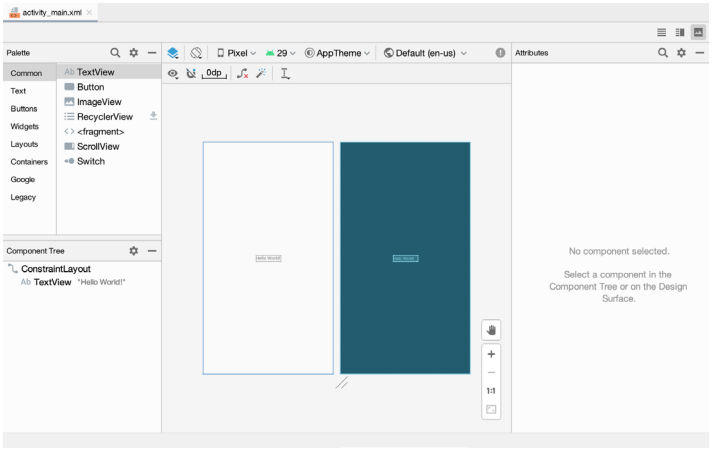

Figure 3-6

In the toolbar across the top of the Layout Editor window is a menu (currently set to *Pixel* in the above figure) which is reflected in the visual representation of the device within the Layout Editor panel. A wide range of other device options are available for selection by clicking on this menu.

To change the orientation of the device representation between landscape and portrait simply use the drop down menu immediately to the left of the device selection menu showing the icon.

As can be seen 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. It should be noted, however, that not all user interface components are obviously visible to the user. One such category consists of *layouts*. Android supports a variety of layouts that provide different levels of control over how visual

user interface components are positioned and managed on the screen. Though it is difficult to tell from looking at the visual representation of the user interface, the current design has been created using a ConstraintLayout. This can be confirmed by reviewing the information in the *Component Tree* panel which, by default, is located in the lower left-hand corner of the Layout Editor panel and is shown in Figure 3-7:

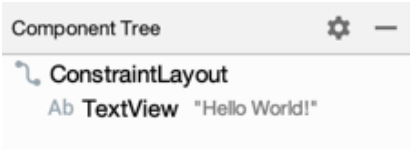

Figure 3-7

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

Before proceeding, also 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 make sure 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 magnet icon. When disabled the magnet appears with a diagonal line through it (Figure 3-8). If necessary, re-enable Autoconnect mode by clicking on this button.

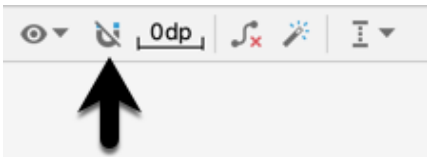

Figure 3-8

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

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

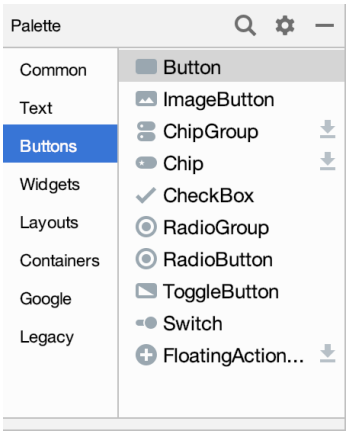

Figure 3-9

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

Figure 3-10

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

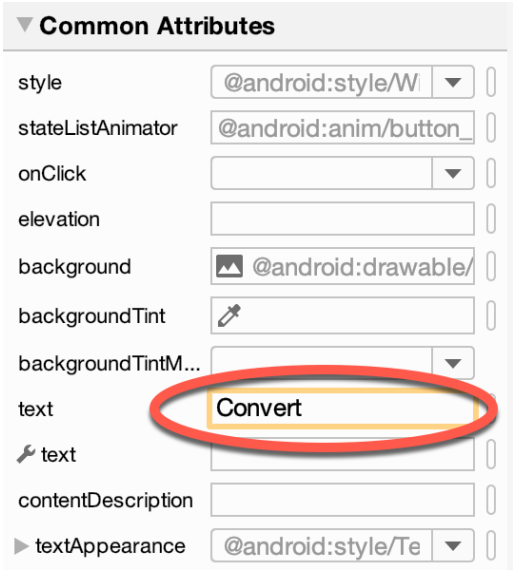

Figure 3-11

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

Just in case the Autoconnect system failed to set all of the layout connections, click on the Infer constraints

button (Figure 3-12) to add any missing constraints to the layout:

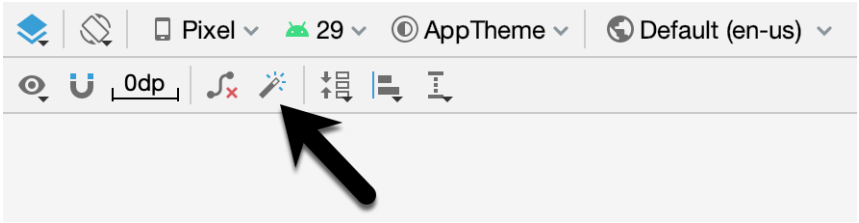

Figure 3-12

At this point it is important to explain the warning button located in the top right-hand corner of the Layout Editor tool as indicated in Figure 3-13. Obviously, this is indicating potential problems with the layout. For details on any problems, click on the button:

Figure 3-13

When clicked, a panel (Figure 3-14) will appear describing the nature of the problems and offering some possible corrective measures:

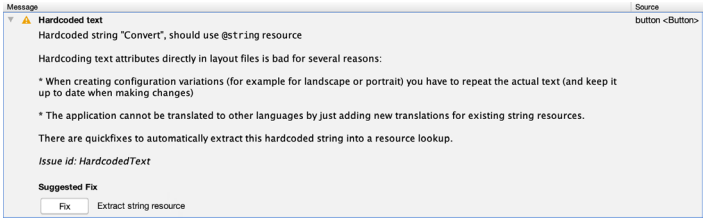

Figure 3-14

Currently, the only warning listed reads as follows:

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

This I18N message is informing us that a potential issue exists with regard to 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 is reminding us that when developing Android applications, attributes and values such as text strings should be stored in the form of *resources* wherever possible. 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".

Click on the *Fix* button in the Issue Explanation panel to display the *Extract Resource* panel (Figure 3-15).

Within this panel, change the resource name field to *convert_string* and leave the resource value set to *Convert* before clicking on the OK button.

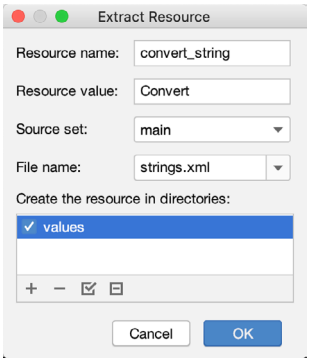

Figure 3-15

It is also worth noting that the string could also have been assigned to a resource when it was entered into the Attributes panel. This involves clicking on the narrow button to the right of the property field in the Attributes panel and selecting the *Add new resource -> New String Value…* menu option from the resulting Resources dialog. In practice, however, it is often quicker to simply set values directly into the Attributes panel fields for any widgets in the layout, then work sequentially through the list in the warnings dialog to extract any necessary resources when the layout is complete.

The next widget to be added is an EditText widget into which the user will enter the dollar amount to be converted. From the widget palette, 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 3-16:

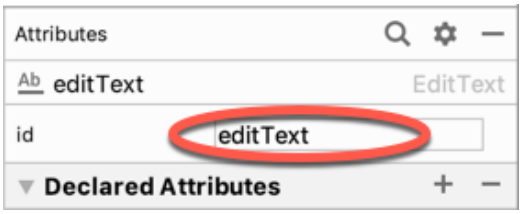

Figure 3-16

Change the id to *dollarText* before proceeding.

Add any missing layout constraints by clicking on the *Infer constraints* button. At this point the layout should resemble that shown in Figure 3-17:

Figure 3-17

3.7 Reviewing the Layout and Resource Files

Before moving on to the next step, we are going to look at some of the internal aspects of user interface design and resource handling. In the previous section, we made some changes to 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 in order to make user interface changes and, in some instances, this may actually be quicker than using the Layout Editor tool. In the top right-hand corner of the Layout Editor panel are three buttons as highlighted in Figure 3-18 below:

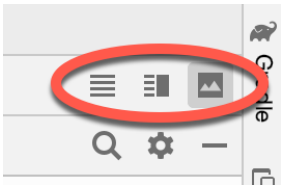

Figure 3-18

By default, the editor will be in *Design* mode whereby just the visual representation of the layout is displayed. The left-most button will switch to *Code* mode to display the XML for the layout, while the middle button enters *Split* mode where both the layout and XML are displayed, as shown in Figure 3-19:

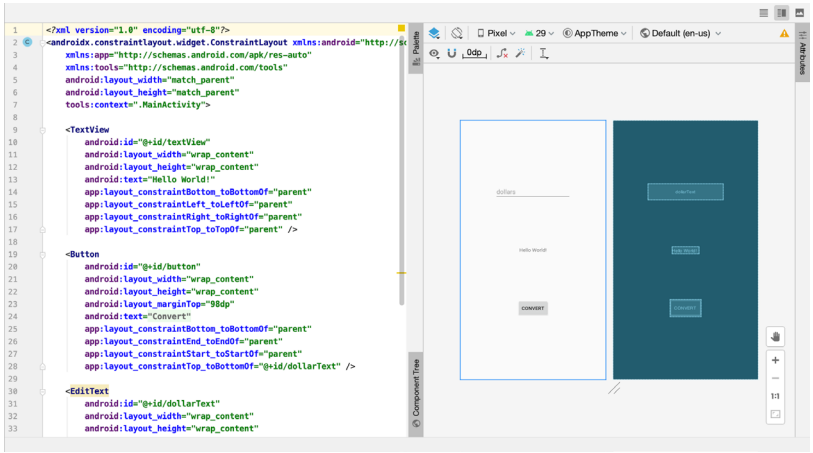

Figure 3-19

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 varying in complexity and content, 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 color of the layout changes in real-time to match the new setting in the XML file. Note also that a small red square appears in the left-hand margin (also referred to as the *gutter*) of the XML editor 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.

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>
```

As a demonstration of 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 back 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. To open this editor, right-click on the *app -> res -> values -> strings.xml* file and select the *Open Translations Editor* menu option. This will display the Translation Editor in the main panel of the Android Studio window:

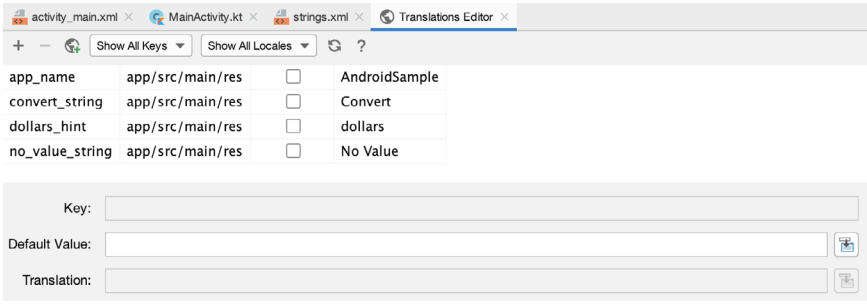

Figure 3-20

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

3.8 Adding Interaction

The final step in this example project is to make the app interactive so that when the user enters a dollar value into the EditText field and clicks the convert button the converted euro value appears on the TextView. This involves the implementation of some event handling on the Button widget. Specifically, the Button needs to be configured so that a method in the app code is called when an *onClick* event is triggered. Event handling can be implemented in a number of different ways and is covered in detail 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:

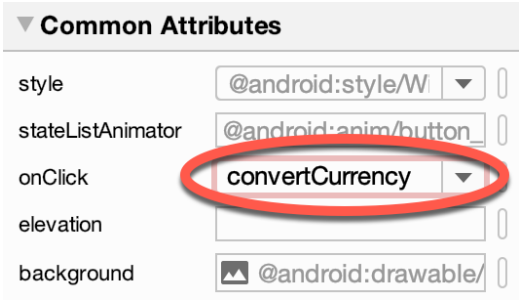

Figure 3-21

Note that the text field for the onClick property is now highlighted with a red border to warn us that the button has been configured to call a method which does not yet exist. To address this, 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;

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.valueOf(dollarText.getText().toString());
            Float euroValue = dollarValue * 0.85F;
            textView.setText(euroValue.toString());
        } 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.

3.9 Summary

While not excessively complex, a number of steps are involved in setting up an Android development environment. Having performed those steps, it is worth working through a simple example to make sure the environment is correctly installed and configured. In this chapter, we have created a simple 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 in the case of string values, and briefly touched on the topic of layouts. Next we looked at the underlying XML that is used to store the user interface designs of Android applications.

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

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

Chapter 4

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

In the course of developing Android apps in Android Studio it will be necessary to compile and run an application multiple times. An Android application may be tested by installing and running it either on a physical device or in an *Android Virtual Device (AVD)* emulator environment. Before an AVD can be used, it must first be created and configured to match the specifications of a particular device model. The goal of this chapter, therefore, is to work through the steps involved in creating such a virtual device using the Pixel 3 phone as a reference example.

4.1 About Android Virtual Devices

AVDs are essentially emulators that allow Android applications to be tested without the necessity to install the application on a physical Android based device. An AVD may be configured to emulate a variety of hardware features including options such as screen size, memory capacity and the presence or otherwise of features such as a camera, GPS navigation support or an accelerometer. As part of the standard Android Studio installation, a number of emulator templates are installed allowing AVDs to be configured for a range of different devices. Custom configurations may be created to match any physical Android device by specifying properties such as processor type, memory capacity and the size and pixel density of the screen.

When launched, an AVD will appear as a window containing an emulated Android device environment. Figure 4-1, for example, shows an AVD session configured to emulate the Google Pixel 3 model.

New AVDs are created and managed using the Android Virtual Device Manager, which may be used either in command-line mode or with a more user-friendly graphical user interface.

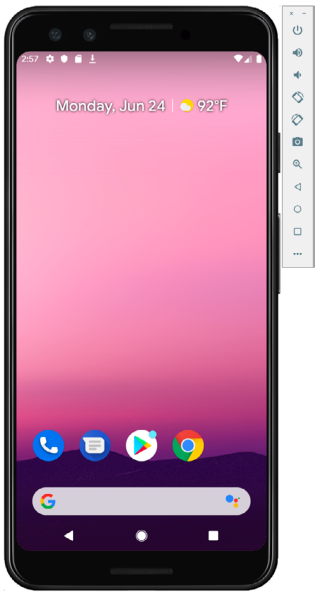

Figure 4-1

4.2 Creating a New AVD

In order to test the behavior of an application in the absence of a physical device, it will be necessary to create an AVD for a specific Android device configuration.

To create a new AVD, the first step is to launch the AVD Manager. This can be achieved from within the Android Studio environment by selecting the *Tools -> AVD Manager* menu option from within the main window.

Once launched, the tool will appear as outlined in Figure 4-2 if existing AVD instances have been created:

Figure 4-2

To add an additional AVD, begin by clicking on the *Create Virtual Device* button in order to invoke the *Virtual Device Configuration* dialog:

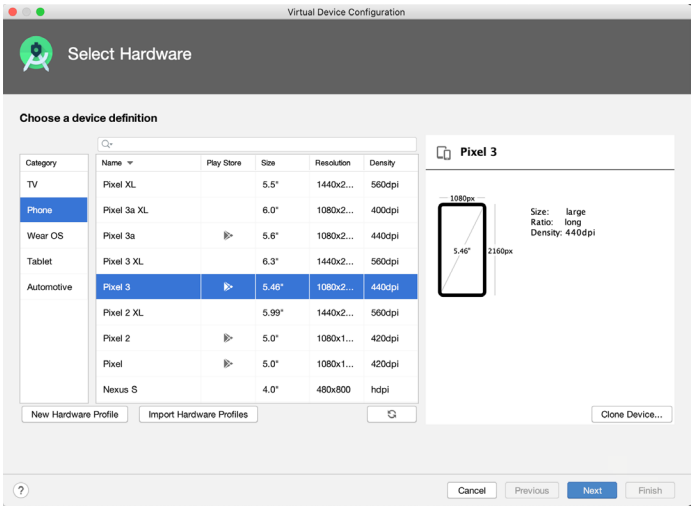

Figure 4-3

Within the dialog, perform the following steps to create a Pixel 3 compatible emulator:

1. From the *Category* panel, select the *Phone* option to display the list of available Android phone AVD templates.

2. Select the *Pixel 3* device option and click *Next*.

3. On the System Image screen, select the latest version of Android for the *x86* ABI. Note that if the system image has not yet been installed a *Download* link will be provided next to the Release Name. Click this link to download and install the system image before selecting it. If the image you need is not listed, click on the *x86 images* and *Other images* tabs to view alternative lists.

4. Click *Next* to proceed and enter a descriptive name (for example *Pixel 3 API 29*) into the name field or simply accept the default name.

5. Click *Finish* to create the AVD.

6. With the AVD created, the AVD Manager may now be closed. If future modifications to the AVD are necessary, simply re-open the AVD Manager, select the AVD from the list and click on the pencil icon in the *Actions* column of the device row in the AVD Manager.

4.3 Starting the Emulator

To perform a test run of the newly created AVD emulator, simply select the emulator from the AVD Manager and click on the launch button (the green triangle in the Actions column). The emulator will appear in a new window and begin the startup process. The amount of time it takes for the emulator to start will depend on the configuration of both the AVD and the system on which it is running.

Although the emulator probably defaulted to appearing in portrait orientation, this and other default options can be changed. Within the AVD Manager, select the new Pixel 3 entry and click on the pencil icon in the *Actions* column of the device row. In the configuration screen locate the *Startup and orientation* section and change the orientation setting. Exit and restart the emulator session to see this change take effect. More details on the emulator are covered in the next chapter (*"Using and Configuring the Android Studio AVD Emulator"*).

To save time in the next section of this chapter, leave the emulator running before proceeding.

4.4 Running the Application in the AVD

With an AVD emulator configured, the example AndroidSample application created in the earlier chapter now can be compiled and run. With the AndroidSample project loaded into Android Studio, make sure that the newly created Pixel 3 AVD is displayed in the device menu (marked A in Figure 4-4 below), then either click on the run button represented by a green triangle (B), select the *Run -> Run 'app'* menu option or use the Ctrl-R keyboard shortcut:

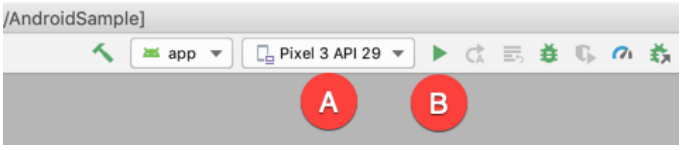

Figure 4-4

The device menu (A) may be used to select a different AVD instance or physical device as the run target, and also to run the app on multiple devices. The menu also provides access to the AVD Manager and device connection trouble shooting options:

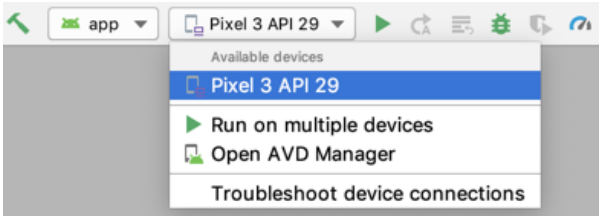

Figure 4-5

Once the application is installed and running, the user interface for the first fragment will appear within the emulator:

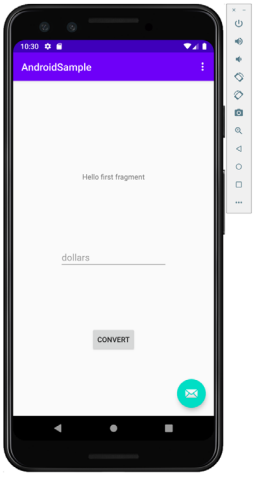

Figure 4-6

In the event that the activity does not automatically launch, check to see if the launch icon has appeared among the apps on the emulator. If it has, simply click on it to launch the application. Once the run process begins, the Run tool window will become available. The Run tool window will display diagnostic information as the application package is installed and launched. Figure 4-7 shows the Run tool window output from a successful application launch:

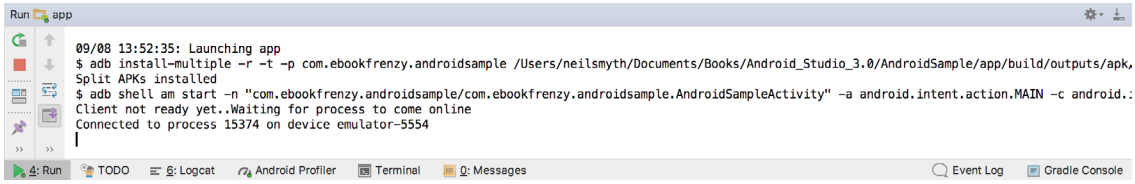

Figure 4-7

If problems are encountered during the launch process, the Run tool window will provide information that will hopefully help to isolate the cause of the problem.

Assuming that the application loads into the emulator and runs as expected, we have safely verified that the Android development environment is correctly installed and configured.

4.5 Stopping a Running Application

To stop a running application, simply click on the stop button located in the main toolbar as shown in Figure 4-8:

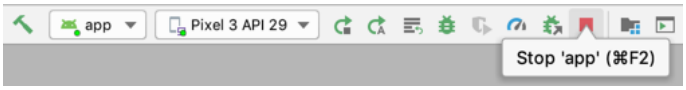

Figure 4-8

An app may also be terminated using the Run tool window. Begin by displaying the *Run* tool window using the window bar button that becomes available when the app is running. Once the Run tool window appears, click the stop button highlighted in Figure 4-9 below:

Figure 4-9

4.6 Supporting Dark Theme

Android 10 introduced the much awaited dark theme, support for which is not enabled by default in Android Studio app projects. To test dark theme in the AVD emulator, open the Settings app within the running Android instance in the emulator. There are a number of different ways to access the settings app. The quickest is to display the home screen and then click and drag upwards from the bottom of the screen (just below the search bar). This will display all of the apps installed on the device, one of which will be the Settings app.

Within the Settings app, choose the *Display* category and enable the *Dark Theme* option as shown in Figure 4-10 so that the screen background turns black:

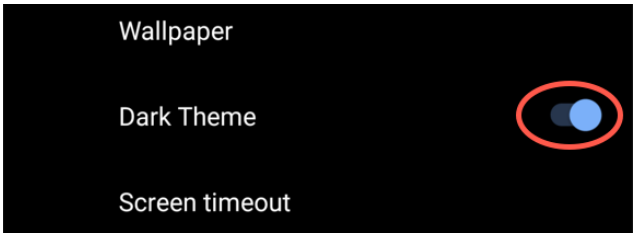

Figure 4-10

With dark theme enabled, run the AndroidSample app and note that it appears as before and does not conform to the dark theme.

In order for an app to adopt dark theme, it must be derived from the Android DayNight theme. By default, new projects use the Light.DarkActionBar theme. To change this setting, navigate to the *res -> values -> styles.xml* file in the Project window as shown in Figure 4-11 and double-click on it to load it into the editor:

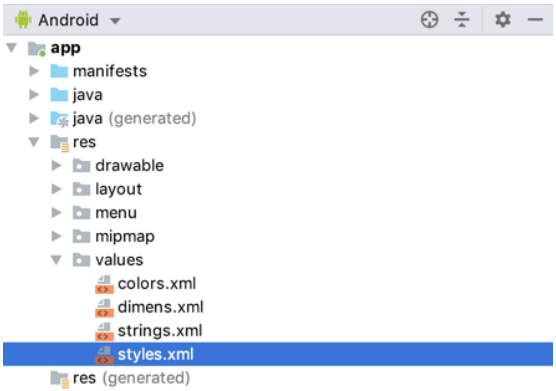

Figure 4-11

Once loaded, edit the AppTheme style entry so that it reads as follows:

```
<resources>

    <!-- Base application theme. -->
    <style name="AppTheme" parent="Theme.AppCompat.DayNight">
        <!-- Customize your theme here. -->
.
.
```

After making the change, re-run the app on the emulator and note that it now conforms to the dark theme as shown in Figure 4-12:

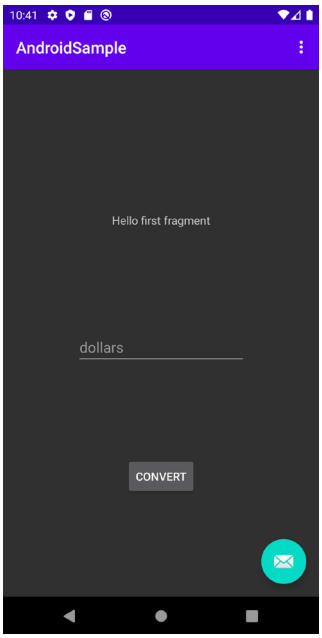

Figure 4-12

Open the Settings app, turn off dark theme and return to the AndroidSample app. The app should have

automatically switched back to light mode.

4.7 AVD Command-line Creation

As previously discussed, in addition to the graphical user interface it is also possible to create a new AVD directly from the command-line. This is achieved using the *avdmanager* tool in conjunction with some command-line options. Once initiated, the tool will prompt for additional information before creating the new AVD.

The avdmanager tool requires access to the Java Runtime Environment (JRE) in order to run. If, when attempting run avdmanager, an error message appears indicating that the 'java' command cannot be found, the command prompt or terminal window within which you are running the command can be configured to use the OpenJDK environment bundled with Android Studio. Begin by identifying the location of the OpenJDK JRE as follows:

1. Launch Android Studio and open the AndroidSample project created earlier in the book.
2. Select the *File -> Project Structure...* menu option.
3. Copy the path contained within the *JDK location* field of the Project Structure dialog. This represents the location of the JRE bundled with Android Studio.

On Windows, execute the following command within the command prompt window from which avdmanager is to be run (where <*path to jre*> is replaced by the path copied from the Project Structure dialog above):

```
set JAVA_HOME=<path to jre>
```

On macOS or Linux, execute the following command:

```
export JAVA_HOME="<path to jre>"
```

If you expect to use the avdmanager tool frequently, follow the environment variable steps for your operating system outlined in the chapter entitled *"Setting up an Android Studio Development Environment"* to configure JAVA_HOME on a system-wide basis.

Assuming that the system has been configured such that the Android SDK *tools* directory is included in the PATH environment variable, a list of available targets for the new AVD may be obtained by issuing the following command in a terminal or command window:

```
avdmanager list targets
```

The resulting output from the above command will contain a list of Android SDK versions that are available on the system. For example:

```
Available Android targets:
----------
id: 1 or "android-29"
     Name: Android API 29
     Type: Platform
     API level: 29
     Revision: 1
----------
id: 2 or "android-26"
     Name: Android API 26
     Type: Platform
     API level: 26
     Revision: 1
```

The avdmanager tool also allows new AVD instances to be created from the command line. For example, to

create a new AVD named *myAVD* using the target ID for the Android API level 29 device using the x86 ABI, the following command may be used:

```
avdmanager create avd -n myAVD -k "system-images;android-29;google_apis_
playstore;x86"
```

The android tool will create the new AVD to the specifications required for a basic Android 8 device, also providing the option to create a custom configuration to match the specification of a specific device if required. Once a new AVD has been created from the command line, it may not show up in the Android Device Manager tool until the *Refresh* button is clicked.

In addition to the creation of new AVDs, a number of other tasks may be performed from the command line. For example, a list of currently available AVDs may be obtained using the *list avd* command line arguments:

```
avdmanager list avd


Available Android Virtual Devices:
    Name: Pixel_XL_API_28_No_Play
  Device: pixel_xl (Google)
    Path: /Users/neilsmyth/.android/avd/Pixel_XL_API_28_No_Play.avd
  Target: Google APIs (Google Inc.)
          Based on: Android API 28 Tag/ABI: google_apis/x86
    Skin: pixel_xl_silver
  Sdcard: 512M
```

Similarly, to delete an existing AVD, simply use the *delete* option as follows:

```
avdmanager delete avd –n <avd name>
```

4.8 Android Virtual Device Configuration Files

By default, the files associated with an AVD are stored in the *.android/avd* sub-directory of the user's home directory, the structure of which is as follows (where <*avd name*> is replaced by the name assigned to the AVD):

```
<avd name>.avd/config.ini
<avd name>.avd/userdata.img
<avd name>.ini
```

The *config.ini* file contains the device configuration settings such as display dimensions and memory specified during the AVD creation process. These settings may be changed directly within the configuration file and will be adopted by the AVD when it is next invoked.

The <*avd name*>*.ini* file contains a reference to the target Android SDK and the path to the AVD files. Note that a change to the *image.sysdir* value in the *config.ini* file will also need to be reflected in the *target* value of this file.

4.9 Moving and Renaming an Android Virtual Device

The current name or the location of the AVD files may be altered from the command line using the *avdmanager* tool's *move avd* argument. For example, to rename an AVD named Nexus9 to Nexus9B, the following command may be executed:

```
avdmanager move avd -n Nexus9 -r Nexus9B
```

To physically relocate the files associated with the AVD, the following command syntax should be used:

```
avdmanager move avd -n <avd name> -p <path to new location>
```

For example, to move an AVD from its current file system location to /tmp/Nexus9Test:

```
avdmanager move avd -n Nexus9 -p /tmp/Nexus9Test
```

Note that the destination directory must not already exist prior to executing the command to move an AVD.

4.10 Summary

A typical application development process follows a cycle of coding, compiling and running in a test environment. Android applications may be tested on either a physical Android device or using an Android Virtual Device (AVD) emulator. AVDs are created and managed using the Android AVD Manager tool which may be used either as a command line tool or using a graphical user interface. When creating an AVD to simulate a specific Android device model it is important that the virtual device be configured with a hardware specification that matches that of the physical device.

Chapter 5

5. Using and Configuring the Android Studio AVD Emulator

The Android Virtual Device (AVD) emulator environment bundled with Android Studio 1.x was an uncharacteristically weak point in an otherwise reputable application development environment. Regarded by many developers as slow, inflexible and unreliable, the emulator was long overdue for an overhaul. Fortunately, Android Studio 2 introduced an enhanced emulator environment providing significant improvements in terms of configuration flexibility and overall performance and further enhancements have been made in subsequent releases.

Before the next chapter explores testing on physical Android devices, this chapter will take some time to provide an overview of the Android Studio AVD emulator and highlight many of the configuration features that are available to customize the environment.

5.1 The Emulator Environment

When launched, the emulator displays an initial splash screen during the loading process. Once loaded, the main emulator window appears containing a representation of the chosen device type (in the case of Figure 5-1 this is a Nexus 5X device):

Figure 5-1

Positioned along the right-hand edge of the window is the toolbar providing quick access to the emulator controls and configuration options.

5.2 The Emulator Toolbar Options

The emulator toolbar (Figure 5-2) provides access to a range of options relating to the appearance and behavior of the emulator environment.

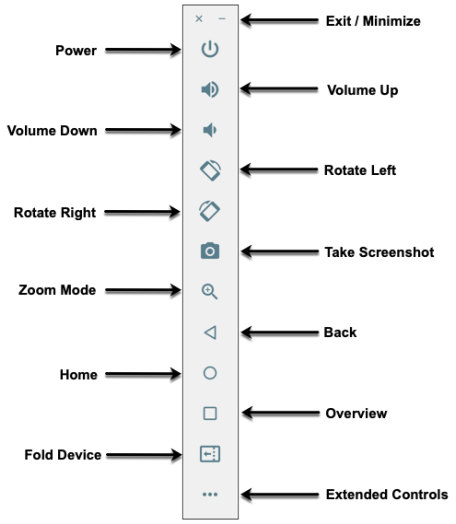

Figure 5-2

Each button in the toolbar has associated with it a keyboard accelerator which can be identified either by hovering the mouse pointer over the button and waiting for the tooltip to appear, or via the help option of the extended controls panel.

Though many of the options contained within the toolbar are self-explanatory, each option will be covered for the sake of completeness:

- **Exit / Minimize** – The uppermost 'x' button in the toolbar exits the emulator session when selected while the '-' option minimizes the entire window.
- **Power** – The Power button simulates the hardware power button on a physical Android device. Clicking and releasing this button will lock the device and turn off the screen. Clicking and holding this button will initiate the device "Power off" request sequence.
- **Volume Up / Down** – Two buttons that control the audio volume of playback within the simulator environment.
- **Rotate Left/Right** – Rotates the emulated device between portrait and landscape orientations.
- **Take Screenshot** – Takes a screenshot of the content currently displayed on the device screen. The captured image is stored at the location specified in the Settings screen of the extended controls panel as outlined later in this chapter.
- **Zoom Mode** – This button toggles in and out of zoom mode, details of which will be covered later in this chapter.
- **Back** – Simulates selection of the standard Android "Back" button. As with the Home and Overview buttons outlined below, the same results can be achieved by selecting the actual buttons on the emulator screen.
- **Home** – Simulates selection of the standard Android "Home" button.
- **Overview** – Simulates selection of the standard Android "Overview" button which displays the currently running apps on the device.
- **Fold Device** – Simulates the folding and unfolding of a foldable device. This option is only available if the emulator is running a foldable device system image.

- **Extended Controls** – Displays the extended controls panel, allowing for the configuration of options such as simulated location and telephony activity, battery strength, cellular network type and fingerprint identification.

5.3 Working in Zoom Mode

The zoom button located in the emulator toolbar switches in and out of zoom mode. When zoom mode is active the toolbar button is depressed and the mouse pointer appears as a magnifying glass when hovering over the device screen. Clicking the left mouse button will cause the display to zoom in relative to the selected point on the screen, with repeated clicking increasing the zoom level. Conversely, clicking the right mouse button decreases the zoom level. Toggling the zoom button off reverts the display to the default size.

Clicking and dragging while in zoom mode will define a rectangular area into which the view will zoom when the mouse button is released.

While in zoom mode the visible area of the screen may be panned using the horizontal and vertical scrollbars located within the emulator window.

5.4 Resizing the Emulator Window

The size of the emulator window (and the corresponding representation of the device) can be changed at any time by clicking and dragging on any of the corners or sides of the window.

5.5 Extended Control Options

The extended controls toolbar button displays the panel illustrated in Figure 5-3. By default, the location settings will be displayed. Selecting a different category from the left-hand panel will display the corresponding group of controls:

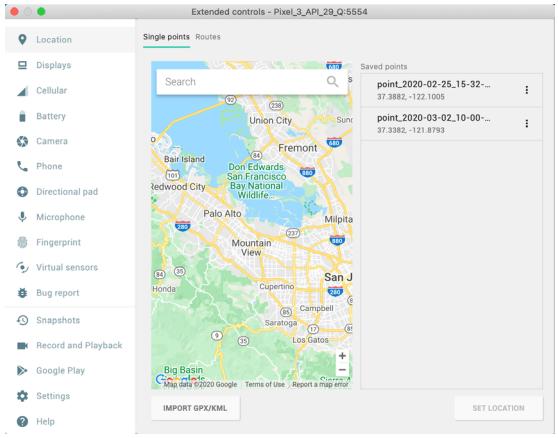

Figure 5-3

5.5.1 Location

The location controls allow simulated location information to be sent to the emulator in the form of decimal or sexigesimal coordinates. Location information can take the form of a single location, or a sequence of points representing movement of the device, the latter being provided via a file in either GPS Exchange (GPX) or Keyhole Markup Language (KML) format. Alternatively, the integrated Google Maps panel may be used to visually select single points or travel routes.

5.5.2 Displays

In addition to the main display shown within the emulator screen, the Displays option allows additional displays to be added running within the same Android instance. This can be useful for testing apps for dual screen

devices such as the Microsoft Surface Duo. These additional screens can be configured to be any required size and appear within the same emulator window as the main screen.

5.5.3 Cellular

The type of cellular connection being simulated can be changed within the cellular settings screen. Options are available to simulate different network types (CSM, EDGE, HSDPA etc) in addition to a range of voice and data scenarios such as roaming and denied access.

5.5.4 Camera

The emulator simulates a 3D scene when the camera is active. This takes the form of the interior of a virtual building through which you can navigate by holding down the Option key (Alt on Windows) while using the mouse pointer and keyboard keys when recording video or before taking a photo within the emulator. This extended configuration option allows different images to be uploaded for display within the virtual environment.

5.5.5 Battery

A variety of battery state and charging conditions can be simulated on this panel of the extended controls screen, including battery charge level, battery health and whether the AC charger is currently connected.

5.5.6 Phone

The phone extended controls provide two very simple but useful simulations within the emulator. The first option allows for the simulation of an incoming call from a designated phone number. This can be of particular use when testing the way in which an app handles high level interrupts of this nature.

The second option allows the receipt of text messages to be simulated within the emulator session. As in the real world, these messages appear within the Message app and trigger the standard notifications within the emulator.

5.5.7 Directional Pad

A directional pad (D-Pad) is an additional set of controls either built into an Android device or connected externally (such as a game controller) that provides directional controls (left, right, up, down). The directional pad settings allow D-Pad interaction to be simulated within the emulator.

5.5.8 Microphone

The microphone settings allow the microphone to be enabled and virtual headset and microphone connections to be simulated. A button is also provided to launch the Voice Assistant on the emulator.

5.5.9 Fingerprint

Many Android devices are now supplied with built-in fingerprint detection hardware. The AVD emulator makes it possible to test fingerprint authentication without the need to test apps on a physical device containing a fingerprint sensor. Details on how to configure fingerprint testing within the emulator will be covered in detail later in this chapter.

5.5.10 Virtual Sensors

The virtual sensors option allows the accelerometer and magnetometer to be simulated to emulate the effects of the physical motion of a device such as rotation, movement and tilting through yaw, pitch and roll settings.

5.5.11 Snapshots

Snapshots contain the state of the currently running AVD session to be saved and rapidly restored making it easy to return the emulator to an exact state. Snapshots are covered in detail later in this chapter.

5.5.12 Record and Playback

Allows the emulator screen and audio to be recorded and saved in either WebM or animated GIF format.

5.5.13 Google Play

If the emulator is running a version of Android with Google Play Services installed, this option displays the current Google Play version and provides the option to update the emulator to the latest version.

5.5.14 Settings

The settings panel provides a small group of configuration options. Use this panel to choose a darker theme for the toolbar and extended controls panel, specify a file system location into which screenshots are to be saved, configure OpenGL support levels, and to configure the emulator window to appear on top of other windows on the desktop.

5.5.15 Help

The Help screen contains three sub-panels containing a list of keyboard shortcuts, links to access the emulator online documentation, file bugs and send feedback, and emulator version information.

5.6 Working with Snapshots

When an emulator starts for the very first time it performs a *cold boot* much like a physical Android device when it is powered on. This cold boot process can take some time to complete as the operating system loads and all the background processes are started. To avoid the necessity of going through this process every time the emulator is started, the system is configured to automatically save a snapshot (referred to as a *quick-boot snapshot*) of the emulator's current state each time it exits. The next time the emulator is launched, the quick-boot snapshot is loaded into memory and execution resumes from where it left off previously, allowing the emulator to restart in a fraction of the time needed for a cold boot to complete.

The Snapshots screen of the extended controls panel can be used to store additional snapshots at any point during the execution of the emulator. This saves the exact state of the entire emulator allowing the emulator to be restored to the exact point in time that the snapshot was taken. From within the screen, snapshots can be taken using the *Take Snapshot* button (marked A in Figure 5-4). To restore an existing snapshot, select it from the list (B) and click the run button (C) located at the bottom of the screen. Options are also provided to edit (D) the snapshot name and description and to delete (E) the currently selected snapshot:

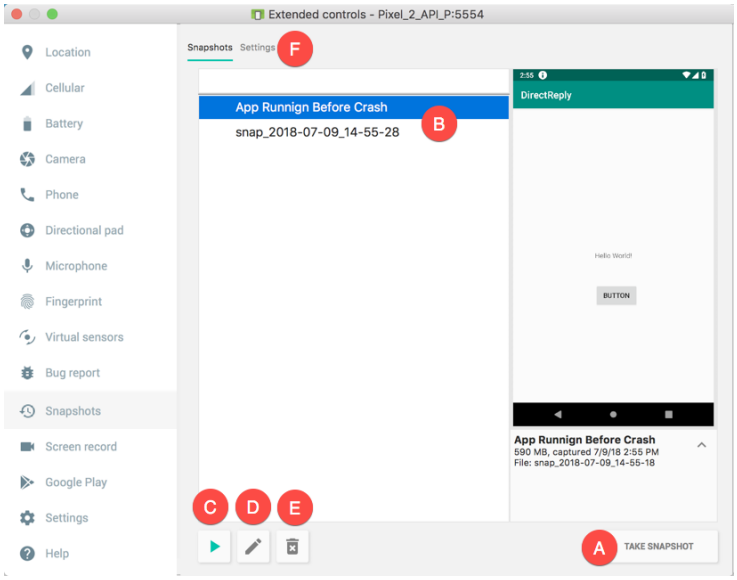

Figure 5-4

The Settings option (F) provides the option to configure the automatic saving of quick-boot snapshots (by default the emulator will ask whether to save the quick boot snapshot each time the emulator exits) and to reload the most recent snapshot. To force an emulator session to perform a cold boot instead of using a previous quick-boot snapshot, open the AVD Manager (*Tools -> AVD Manager*), click on the down arrow in the actions column for the emulator and select the Cold Boot Now menu option.

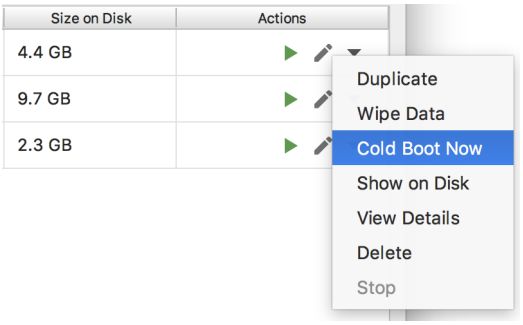

Figure 5-5

5.7 Configuring Fingerprint Emulation

The emulator allows up to 10 simulated fingerprints to be configured and used to test fingerprint authentication within Android apps. To configure simulated fingerprints begin by launching the emulator, opening the Settings app and selecting the *Security & Location* option.

Within the Security settings screen, select the *Use fingerprint* option. On the resulting information screen click on the *Next* button to proceed to the Fingerprint setup screen. Before fingerprint security can be enabled a backup screen unlocking method (such as a PIN number) must be configured. Click on the *Fingerprint + PIN* button and, when prompted, choose not to require the PIN on device startup. Enter and confirm a suitable PIN number and complete the PIN entry process by accepting the default notifications option.

Proceed through the remaining screens until the Settings app requests a fingerprint on the sensor. At this point display the extended controls dialog, select the *Fingerprint* category in the left-hand panel and make sure that *Finger 1* is selected in the main settings panel:

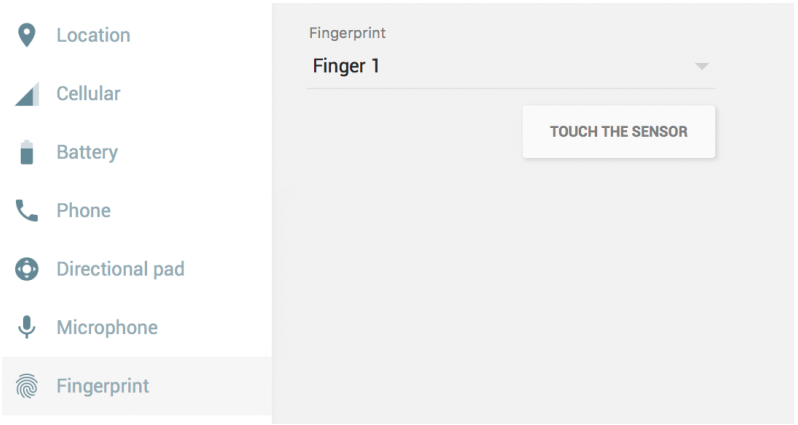

Figure 5-6

Click on the *Touch the Sensor* button to simulate Finger 1 touching the fingerprint sensor. The emulator will report the successful addition of the fingerprint:

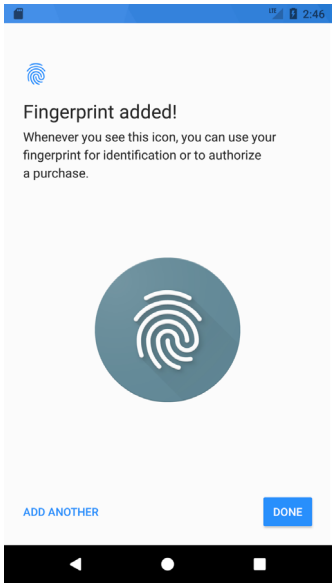

Figure 5-7

To add additional fingerprints click on the *Add Another* button and select another finger from the extended controls panel menu before clicking on the *Touch the Sensor* button once again. The topic of building fingerprint authentication into an Android app is covered in detail in the chapter entitled *"An Android Biometric Authentication Tutorial"*.

5.8 Summary

Android Studio 3.6 contains a new and improved Android Virtual Device emulator environment designed to make it easier to test applications without the need to run on a physical Android device. This chapter has provided a brief tour of the emulator and highlighted key features that are available to configure and customize the environment to simulate different testing conditions

Chapter 6

6. A Tour of the Android Studio User Interface

While it is tempting to plunge into running the example application created in the previous chapter, doing so involves using aspects of the Android Studio user interface which are best described in advance.

Android Studio is a powerful and feature rich development environment that is, to a large extent, intuitive to use. That being said, taking the time now to gain familiarity with the layout and organization of the Android Studio user interface will considerably shorten the learning curve in later chapters of the book. With this in mind, this chapter will provide an initial overview of the various areas and components that make up the Android Studio environment.

6.1 The Welcome Screen

The welcome screen (Figure 6-1) is displayed any time that Android Studio is running with no projects currently open (open projects can be closed at any time by selecting the *File -> Close Project* menu option). If Android Studio was previously exited while a project was still open, the tool will by-pass the welcome screen next time it is launched, automatically opening the previously active project.

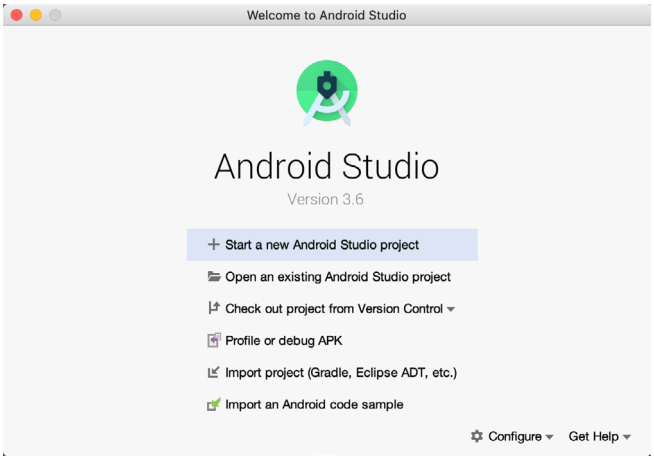

Figure 6-1

In addition to a list of recent projects, the Quick Start menu provides a range of options for performing tasks such as opening, creating and importing projects along with access to projects currently under version control. In addition, the *Configure* menu at the bottom of the window provides access to the SDK Manager along with a vast array of settings and configuration options. A review of these options will quickly reveal that there is almost no aspect of Android Studio that cannot be configured and tailored to your specific needs.

The Configure menu also includes an option to check if updates to Android Studio are available for download.

6.2 The Main Window

When a new project is created, or an existing one opened, the Android Studio *main window* will appear. When multiple projects are open simultaneously, each will be assigned its own main window. The precise configuration of the window will vary depending on which tools and panels were displayed the last time the project was open, but will typically resemble that of Figure 6-2.

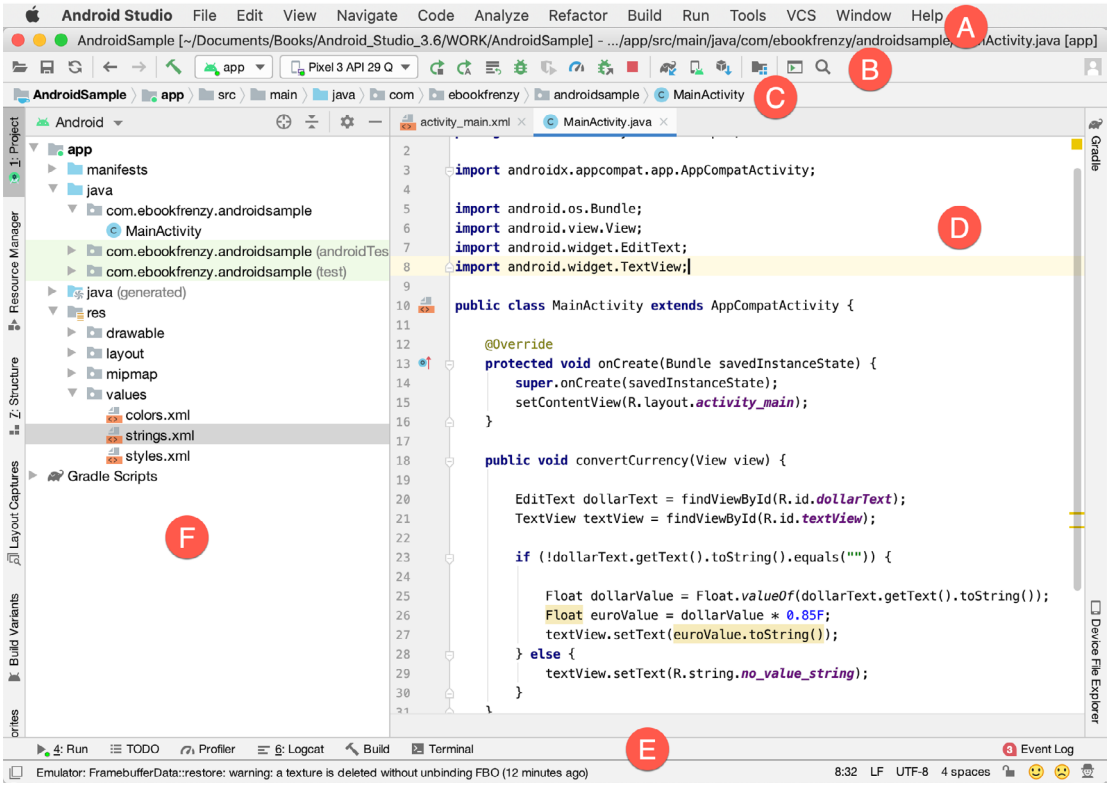

Figure 6-2

The various elements of the main window can be summarized as follows:

A – Menu Bar – Contains a range of menus for performing tasks within the Android Studio environment.

B – Toolbar – A selection of shortcuts to frequently performed actions. The toolbar buttons provide quicker access to a select group of menu bar actions. The toolbar can be customized by right-clicking on the bar and selecting the *Customize Menus and Toolbars…* menu option. If the toolbar is not visible, it can be displayed using the *View -> Appearance -> Toolbar* menu option.

C – Navigation Bar – The navigation bar provides a convenient way to move around the files and folders that make up the project. Clicking on an element in the navigation bar will drop down a menu listing the subfolders and files at that location ready for selection. This provides an alternative to the Project tool window. Hide and display this bar using the *View -> Appearance -> Navigation Bar* menu option.

D – Editor Window – The editor window displays the content of the file on which the developer is currently working. What gets displayed in this location, however, is subject to context. When editing code, for example, the code editor will appear. When working on a user interface layout file, on the other hand, the user interface Layout Editor tool will appear. When multiple files are open, each file is represented by a tab located along the

top edge of the editor as shown in Figure 6-3.

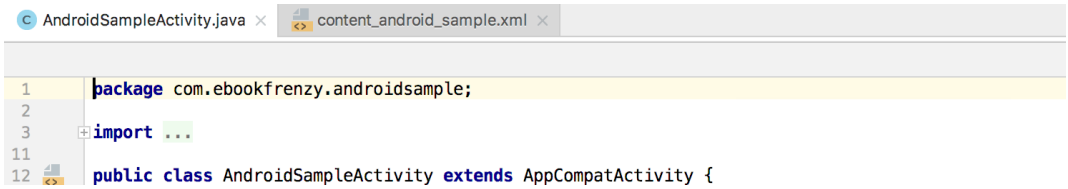

Figure 6-3

E – Status Bar – The status bar displays informational messages about the project and the activities of Android Studio together with the tools menu button located in the far left corner. Hovering over items in the status bar will provide a description of that field. Many fields are interactive, allowing the user to click to perform tasks or obtain more detailed status information.

F – Project Tool Window – The project tool window provides a hierarchical overview of the project file structure allowing navigation to specific files and folders to be performed. The toolbar can be used to display the project in a number of different ways. The default setting is the *Android* view which is the mode primarily used in the remainder of this book.

The project tool window is just one of a number of tool windows available within the Android Studio environment.

6.3 The Tool Windows

In addition to the project view tool window, Android Studio also includes a number of other windows which, when enabled, are displayed along the bottom and sides of the main window. The tool window quick access menu can be accessed by hovering the mouse pointer over the button located in the far left-hand corner of the status bar (Figure 6-4) without clicking the mouse button.

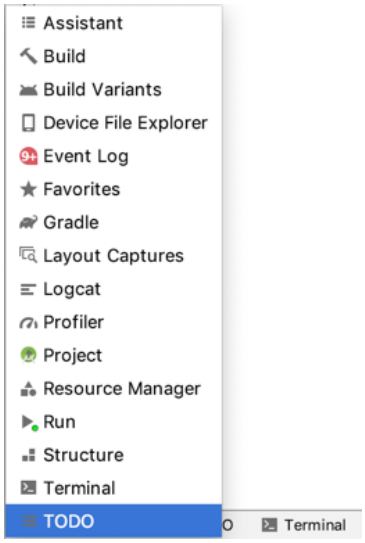

Figure 6-4

Selecting an item from the quick access menu will cause the corresponding tool window to appear within the main window.

Alternatively, a set of *tool window bars* can be displayed by clicking on the quick access menu icon in the status bar. These bars appear along the left, right and bottom edges of the main window (as indicated by the arrows in

Figure 6-5) and contain buttons for showing and hiding each of the tool windows. When the tool window bars are displayed, a second click on the button in the status bar will hide them.

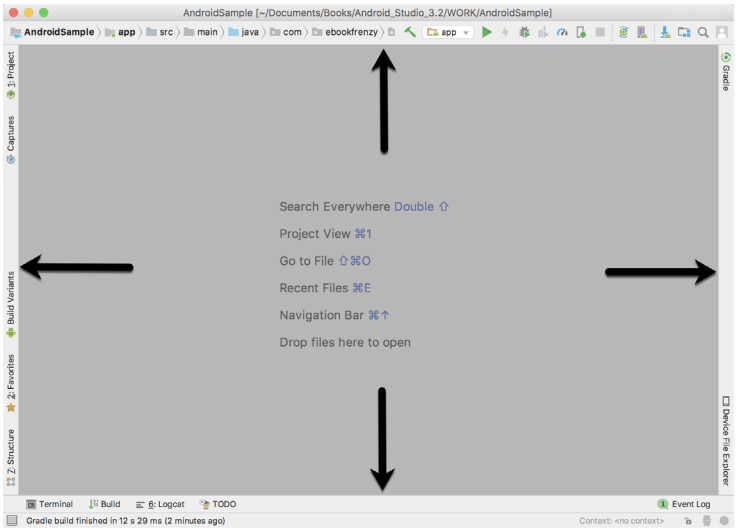

Figure 6-5

Clicking on a button will display the corresponding tool window while a second click will hide the window. Buttons prefixed with a number (for example 1: Project) indicate that the tool window may also be displayed by pressing the Alt key on the keyboard (or the Command key for macOS) together with the corresponding number.

The location of a button in a tool window bar indicates the side of the window against which the window will appear when displayed. These positions can be changed by clicking and dragging the buttons to different locations in other window tool bars.

Each tool window has its own toolbar along the top edge. The buttons within these toolbars vary from one tool to the next, though all tool windows contain a settings option, represented by the cog icon, which allows various aspects of the window to be changed. Figure 6-6 shows the settings menu for the project view tool window. Options are available, for example, to undock a window and to allow it to float outside of the boundaries of the Android Studio main window and to move and resize the tool panel.

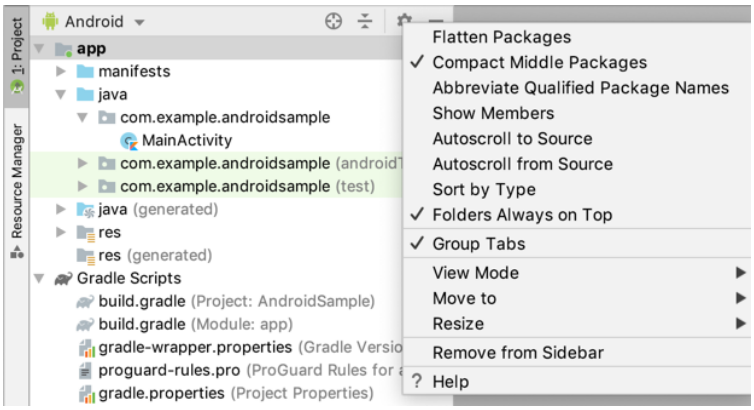

Figure 6-6

All of the windows also include a far right button on the toolbar providing an additional way to hide the tool window from view. A search of the items within a tool window can be performed simply by giving that window focus by clicking in it and then typing the search term (for example the name of a file in the Project tool window). A search box will appear in the window's tool bar and items matching the search highlighted.

Android Studio offers a wide range of tool windows, the most commonly used of which are as follows:

Project – The project view provides an overview of the file structure that makes up the project allowing for quick navigation between files. Generally, double-clicking on a file in the project view will cause that file to be loaded into the appropriate editing tool.

Structure – The structure tool provides a high level view of the structure of the source file currently displayed in the editor. This information includes a list of items such as classes, methods and variables in the file. Selecting an item from the structure list will take you to that location in the source file in the editor window.

Layout Captures – Provides access to all of the layout hierarchy snapshots previously captured using the Layout Inspector tool (*Tools -> Layout Inspector*).

Favorites – A variety of project items can be added to the favorites list. Right-clicking on a file in the project view, for example, provides access to an *Add to Favorites* menu option. Similarly, a method in a source file can be added as a favorite by right-clicking on it in the Structure tool window. Anything added to a Favorites list can be accessed through this Favorites tool window.

Build Variants – The build variants tool window provides a quick way to configure different build targets for the current application project (for example different builds for debugging and release versions of the application, or multiple builds to target different device categories).

TODO – As the name suggests, this tool provides a place to review items that have yet to be completed on the project. Android Studio compiles this list by scanning the source files that make up the project to look for comments that match specified TODO patterns. These patterns can be reviewed and changed by selecting the *File -> Settings…* menu option (*Android Studio -> Preferences…* on macOS) and navigating to the *TODO* page listed under *Editor*.

Logcat – The Logcat tool window provides access to the monitoring log output from a running application in addition to options for taking screenshots and videos of the application and stopping and restarting a process.

Terminal – Provides access to a terminal window on the system on which Android Studio is running. On Windows systems this is the Command Prompt interface, while on Linux and macOS systems this takes the form of a Terminal prompt.

Build - The build tool windows displays information about the build process while a project is being compiled and packaged and displays details of any errors encountered.

Run – The run tool window becomes available when an application is currently running and provides a view of the results of the run together with options to stop or restart a running process. If an application is failing to install and run on a device or emulator, this window will typically provide diagnostic information relating to the problem.

Event Log – The event log window displays messages relating to events and activities performed within Android Studio. The successful build of a project, for example, or the fact that an application is now running will be reported within this tool window.

Gradle – The Gradle tool window provides a view onto the Gradle tasks that make up the project build configuration. The window lists the tasks that are involved in compiling the various elements of the project into

an executable application. Right-click on a top level Gradle task and select the *Open Gradle Config* menu option to load the Gradle build file for the current project into the editor. Gradle will be covered in greater detail later in this book.

Profiler – The Android Profiler tool window provides realtime monitoring and analysis tools for identifying performance issues within running apps, including CPU, memory and network usage. This option becomes available when an app is currently running.

Device File Explorer – The Device File Explorer tool window provides direct access to the filesystem of the currently connected Android device or emulator allowing the filesystem to be browsed and files copied to the local filesystem.

Resource Manager - A tool for adding and managing resources and assets such as images, colors and layout files contained with the project.

6.4 Android Studio Keyboard Shortcuts

Android Studio includes an abundance of keyboard shortcuts designed to save time when performing common tasks. A full keyboard shortcut keymap listing can be viewed and printed from within the Android Studio project window by selecting the *Help -> Keymap Reference* menu option.

6.5 Switcher and Recent Files Navigation

Another useful mechanism for navigating within the Android Studio main window involves the use of the *Switcher*. Accessed via the *Ctrl-Tab* keyboard shortcut, the switcher appears as a panel listing both the tool windows and currently open files (Figure 6-7).

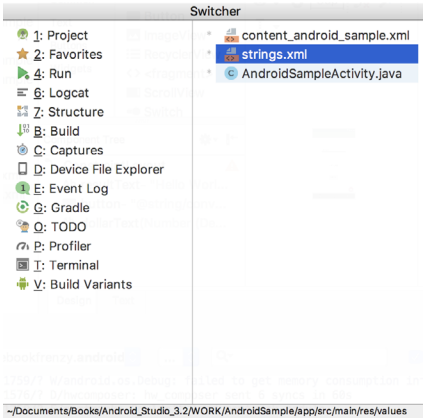

Figure 6-7

Once displayed, the switcher will remain visible for as long as the Ctrl key remains depressed. Repeatedly tapping the Tab key while holding down the Ctrl key will cycle through the various selection options, while releasing the Ctrl key causes the currently highlighted item to be selected and displayed within the main window.

In addition to the switcher, navigation to recently opened files is provided by the Recent Files panel (Figure 6-8). This can be accessed using the Ctrl-E keyboard shortcut (Cmd-E on macOS). Once displayed, either the mouse pointer can be used to select an option or, alternatively, the keyboard arrow keys used to scroll through the file name and tool window options. Pressing the Enter key will select the currently highlighted item.

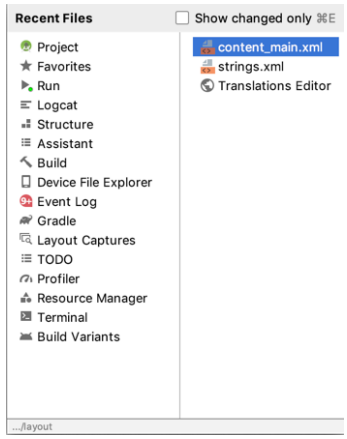

Figure 6-8

6.6 Changing the Android Studio Theme

The overall theme of the Android Studio environment may be changed either from the welcome screen using the *Configure* -> *Settings* option, or via the *File* -> *Settings...* menu option (*Android Studio* -> *Preferences...* on macOS) of the main window.

Once the settings dialog is displayed, select the *Appearance* option in the left-hand panel and then change the setting of the *Theme* menu before clicking on the *Apply* button. The themes available will depend on the platform but usually include options such as Light, IntelliJ, Windows, High Contrast and Darcula. Figure 6-9 shows an example of the main window with the Darcula theme selected:

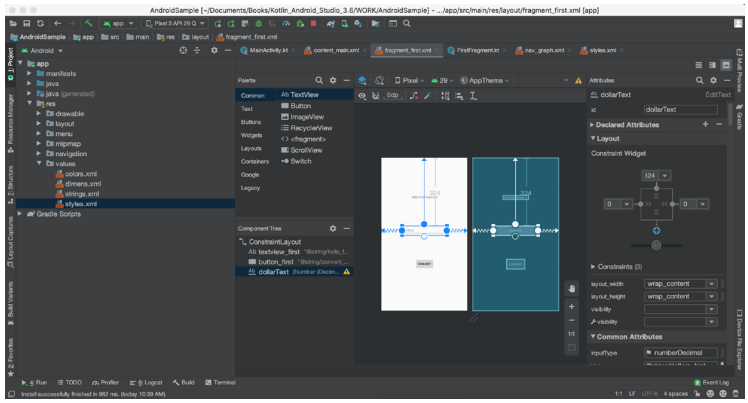

Figure 6-9

6.7 Summary

The primary elements of the Android Studio environment consist of the welcome screen and main window. Each open project is assigned its own main window which, in turn, consists of a menu bar, toolbar, editing and design area, status bar and a collection of tool windows. Tool windows appear on the sides and bottom edges of the main window and can be accessed either using the quick access menu located in the status bar, or via the optional tool window bars.

There are very few actions within Android Studio which cannot be triggered via a keyboard shortcut. A keymap of default keyboard shortcuts can be accessed at any time from within the Android Studio main window.

Chapter 7

7. Testing Android Studio Apps on a Physical Android Device

While much can be achieved by testing applications using an Android Virtual Device (AVD), there is no substitute for performing real world application testing on a physical Android device and there are a number of Android features that are only available on physical Android devices.

Communication with both AVD instances and connected Android devices is handled by the *Android Debug Bridge (ADB)*. In this chapter we will work through the steps to configure the adb environment to enable application testing on a physical Android device with macOS, Windows and Linux based systems.

7.1 An Overview of the Android Debug Bridge (ADB)

The primary purpose of the ADB is to facilitate interaction between a development system, in this case Android Studio, and both AVD emulators and physical Android devices for the purposes of running and debugging applications.

The ADB consists of a client, a server process running in the background on the development system and a daemon background process running in either AVDs or real Android devices such as phones and tablets.

The ADB client can take a variety of forms. For example, a client is provided in the form of a command-line tool named *adb* located in the Android SDK *platform-tools* sub-directory. Similarly, Android Studio also has a built-in client.

A variety of tasks may be performed using the *adb* command-line tool. For example, a listing of currently active virtual or physical devices may be obtained using the *devices* command-line argument. The following command output indicates the presence of an AVD on the system but no physical devices:

```
$ adb devices
List of devices attached
emulator-5554   device
```

7.2 Enabling ADB on Android based Devices

Before ADB can connect to an Android device, that device must first be configured to allow the connection. On phone and tablet devices running Android 6.0 or later, the steps to achieve this are as follows:

1. Open the Settings app on the device and select the *About tablet* or *About phone* option (on newer versions of Android this can be found on the *System* page of the Settings app).

2. On the *About* screen, scroll down to the *Build number* field (Figure 7-1) and tap on it seven times until a message appears indicating that developer mode has been enabled. If the build number is not displayed, unfold the Advanced section of the list.

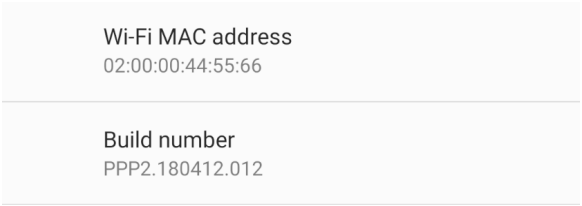

Figure 7-1

3. Return to the main Settings screen and note the appearance of a new option titled Developer options. Select this option and locate the setting on the developer screen entitled USB debugging. Enable the switch next to this item as illustrated in Figure 7-2:

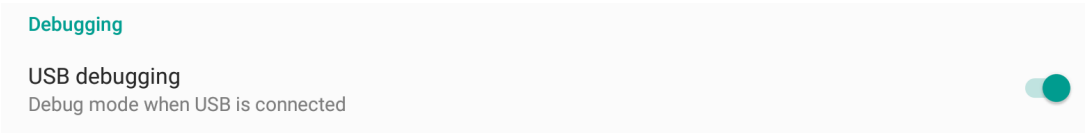

Figure 7-2

4. Swipe downward from the top of the screen to display the notifications panel (Figure 7-3) and note that the device is currently connected for debugging.

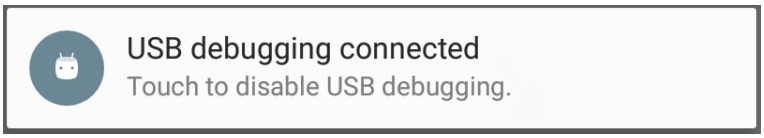

Figure 7-3

At this point, the device is now configured to accept debugging connections from adb on the development system. All that remains is to configure the development system to detect the device when it is attached. While this is a relatively straightforward process, the steps involved differ depending on whether the development system is running Windows, macOS or Linux. Note that the following steps assume that the Android SDK *platform-tools* directory is included in the operating system PATH environment variable as described in the chapter entitled *"Setting up an Android Studio Development Environment"*.

7.2.1 macOS ADB Configuration

In order to configure the ADB environment on a macOS system, connect the device to the computer system using a USB cable, open a terminal window and execute the following command to restart the adb server:

```
$ adb kill-server
$ adb start-server
* daemon not running. starting it now on port 5037 *
* daemon started successfully *
```

Once the server is successfully running, execute the following command to verify that the device has been detected:

```
$ adb devices
List of devices attached
74CE000600000001        offline
```

If the device is listed as *offline*, go to the Android device and check for the presence of the dialog shown in Figure

7-4 seeking permission to *Allow USB debugging*. Enable the checkbox next to the option that reads *Always allow from this computer*, before clicking on *OK*. Repeating the *adb devices* command should now list the device as being available:

```
List of devices attached
015d41d4454bf80c        device
```

In the event that the device is not listed, try logging out and then back in to the macOS desktop and, if the problem persists, rebooting the system.

7.2.2 Windows ADB Configuration

The first step in configuring a Windows based development system to connect to an Android device using ADB is to install the appropriate USB drivers on the system. The USB drivers to install will depend on the model of Android Device. If you have a Google Nexus device, then it will be necessary to install and configure the Google USB Driver package on your Windows system. Detailed steps to achieve this are outlined on the following web page:

https://developer.android.com/sdk/win-usb.html

For Android devices not supported by the Google USB driver, it will be necessary to download the drivers provided by the device manufacturer. A listing of drivers together with download and installation information can be obtained online at:

https://developer.android.com/tools/extras/oem-usb.html

With the drivers installed and the device now being recognized as the correct device type, open a Command Prompt window and execute the following command:

```
adb devices
```

This command should output information about the connected device similar to the following:

```
List of devices attached
HT4CTJT01906        offline
```

If the device is listed as *offline* or *unauthorized*, go to the device display and check for the dialog shown in Figure 7-4 seeking permission to *Allow USB debugging*.

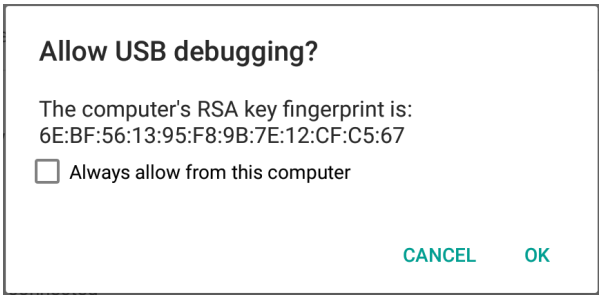

Figure 7-4

Enable the checkbox next to the option that reads *Always allow from this computer*, before clicking on *OK*. Repeating the *adb devices* command should now list the device as being ready:

```
List of devices attached
HT4CTJT01906    device
```

In the event that the device is not listed, execute the following commands to restart the ADB server:

```
adb kill-server
adb start-server
```

If the device is still not listed, try executing the following command:

```
android update adb
```

Note that it may also be necessary to reboot the system.

7.2.3 Linux adb Configuration

For the purposes of this chapter, we will once again use Ubuntu Linux as a reference example in terms of configuring adb on Linux to connect to a physical Android device for application testing.

Physical device testing on Ubuntu Linux requires the installation of a package named *android-tools-adb* which, in turn, requires that the Android Studio user be a member of the *plugdev* group. This is the default for user accounts on most Ubuntu versions and can be verified by running the *id* command. If the plugdev group is not listed, run the following command to add your account to the group:

```
sudo usermod -aG plugdev $LOGNAME
```

After the group membership requirement has been met, the *android-tools-adb* package can be installed by executing the following command:

```
sudo apt-get install android-tools-adb
```

Once the above changes have been made, reboot the Ubuntu system. Once the system has restarted, open a Terminal window, start the adb server and check the list of attached devices:

```
$ adb start-server
* daemon not running. starting it now on port 5037 *
* daemon started successfully *
$ adb devices
List of devices attached
015d41d4454bf80c        offline
```

If the device is listed as *offline* or *unauthorized*, go to the Android device and check for the dialog shown in Figure 7-4 seeking permission to *Allow USB debugging*.

7.3 Testing the adb Connection

Assuming that the adb configuration has been successful on your chosen development platform, the next step is to try running the test application created in the chapter entitled *"Creating an Example Android App in Android Studio"* on the device. Launch Android Studio, open the AndroidSample project and verify that the device appears in the device selection menu as highlighted in Figure 7-5:

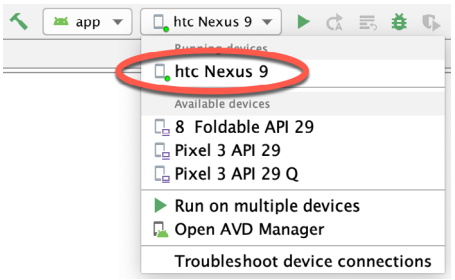

Figure 7-5

Note that this menu also includes the option to test the app on multiple devices and emulators simultaneously.

When selected, this option displays the dialog shown in Figure 7-6 where multiple deployment targets may be selected.

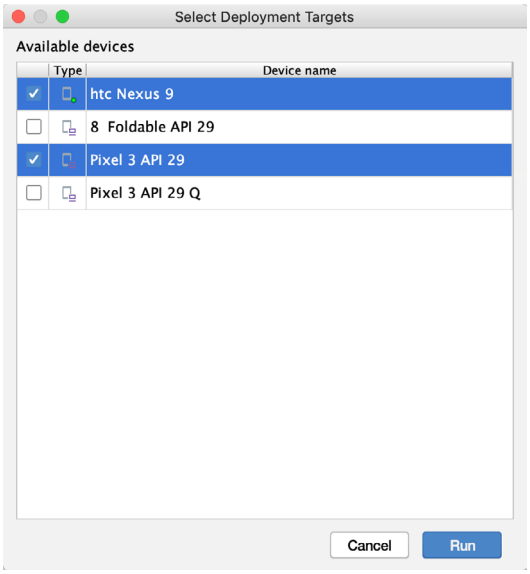

Figure 7-6

7.4 Summary

While the Android Virtual Device emulator provides an excellent testing environment, it is important to keep in mind that there is no real substitute for making sure an application functions correctly on a physical Android device. This, after all, is where the application will be used in the real world.

By default, however, the Android Studio environment is not configured to detect Android devices as a target testing device. It is necessary, therefore, to perform some steps in order to be able to load applications directly onto an Android device from within the Android Studio development environment. The exact steps to achieve this goal differ depending on the development platform being used. In this chapter, we have covered those steps for Linux, macOS and Windows based platforms.

Chapter 8

8. The Basics of the Android Studio Code Editor

Developing applications for Android involves a considerable amount of programming work which, by definition, involves typing, reviewing and modifying lines of code. It should come as no surprise that the majority of a developer's time spent using Android Studio will typically involve editing code within the editor window.

The modern code editor needs to go far beyond the original basics of typing, deleting, cutting and pasting. Today the usefulness of a code editor is generally gauged by factors such as the amount by which it reduces the typing required by the programmer, ease of navigation through large source code files and the editor's ability to detect and highlight programming errors in real-time as the code is being written. As will become evident in this chapter, these are just a few of the areas in which the Android Studio editor excels.

While not an exhaustive overview of the features of the Android Studio editor, this chapter aims to provide a guide to the key features of the tool. Experienced programmers will find that some of these features are common to most code editors available today, while a number are unique to this particular editing environment.

8.1 The Android Studio Editor

The Android Studio editor appears in the center of the main window when a Java, Kotlin, XML or other text based file is selected for editing. Figure 8-1, for example, shows a typical editor session with a Java source code file loaded:

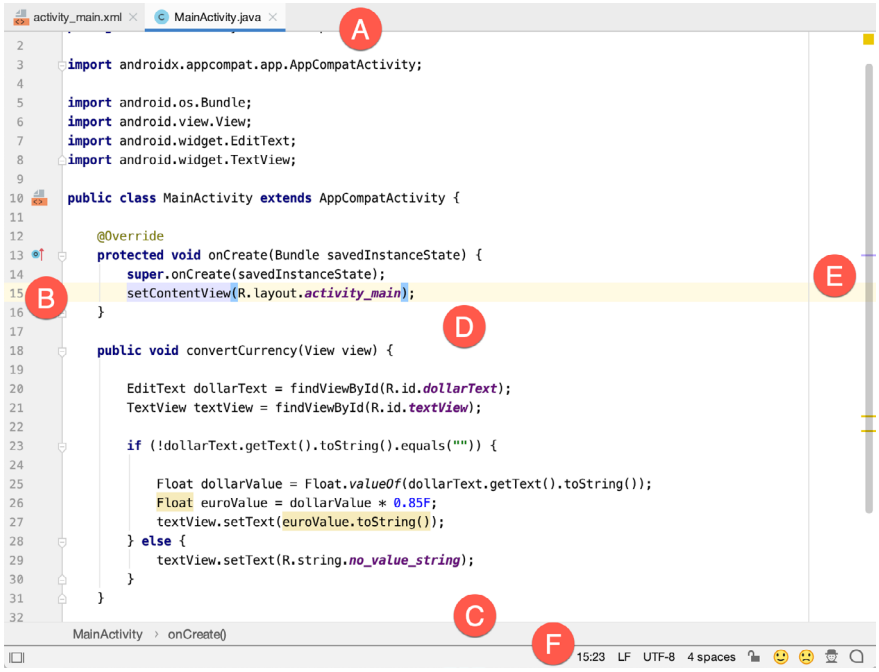

Figure 8-1

The elements that comprise the editor window can be summarized as follows:

A – Document Tabs – Android Studio is capable of holding multiple files open for editing at any one time. As each file is opened, it is assigned a document tab displaying the file name in the tab bar located along the top edge of the editor window. A small dropdown menu will appear in the far right-hand corner of the tab bar when there is insufficient room to display all of the tabs. Clicking on this menu will drop down a list of additional open files. A wavy red line underneath a file name in a tab indicates that the code in the file contains one or more errors that need to be addressed before the project can be compiled and run.

Switching between files is simply a matter of clicking on the corresponding tab or using the *Alt-Left* and *Alt-Right* keyboard shortcuts. Navigation between files may also be performed using the Switcher mechanism (accessible via the *Ctrl-Tab* keyboard shortcut).

To detach an editor panel from the Android Studio main window so that it appears in a separate window, click on the tab and drag it to an area on the desktop outside of the main window. To return the editor to the main window, click on the file tab in the separated editor window and drag and drop it onto the original editor tab bar in the main window.

B – The Editor Gutter Area - The gutter area is used by the editor to display informational icons and controls. Some typical items, among others, which appear in this gutter area are debugging breakpoint markers, controls to fold and unfold blocks of code, bookmarks, change markers and line numbers. Line numbers are switched on by default but may be disabled by right-clicking in the gutter and selecting the *Show Line Numbers* menu option.

C – Code Structure Location - This bar at the bottom of the editor displays the current position of the cursor as it relates to the overall structure of the code. In the following figure, for example, the bar indicates that a setOnClickListener method call is currently being edited, and that this line of code is contained within the onCreate method of the MainActivity class.

Figure 8-2

Selecting an element within the bar will move the cursor to the corresponding location within the code file. For example, selecting the *onCreate()* entry will move the cursor to the top of the onCreate method within the source code.

D – The Editor Area – This is the main area where the code is displayed, entered and edited by the user. Later sections of this chapter will cover the key features of the editing area in detail.

E – The Validation and Marker Sidebar – Android Studio incorporates a feature referred to as "on-the-fly code analysis". What this essentially means is that as you are typing code, the editor is analyzing the code to check for warnings and syntax errors. The indicator at the top of the validation sidebar will change from a green check mark (no warnings or errors detected) to a yellow square (warnings detected) or red alert icon (errors have been detected). Clicking on this indicator will display a popup containing a summary of the issues found with the code in the editor as illustrated in Figure 8-3:

Figure 8-3

The sidebar also displays markers at the locations where issues have been detected using the same color coding. Hovering the mouse pointer over a marker when the line of code is visible in the editor area will display a popup containing a description of the issue (Figure 8-4):

Figure 8-4

Hovering the mouse pointer over a marker for a line of code which is currently scrolled out of the viewing area of the editor will display a "lens" overlay containing the block of code where the problem is located (Figure 8-5) allowing it to be viewed without the necessity to scroll to that location in the editor:

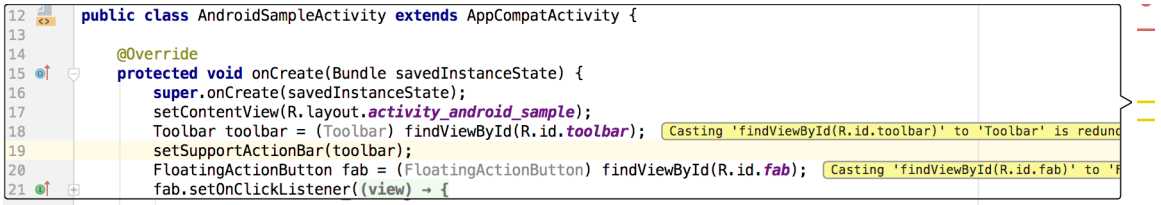

Figure 8-5

It is also worth noting that the lens overlay is not limited to warnings and errors in the sidebar. Hovering over any part of the sidebar will result in a lens appearing containing the code present at that location within the source file.

F – The Status Bar – Though the status bar is actually part of the main window, as opposed to the editor, it does contain some information about the currently active editing session. This information includes the current position of the cursor in terms of lines and characters and the encoding format of the file (UTF-8, ASCII etc.). Clicking on these values in the status bar allows the corresponding setting to be changed. Clicking on the line number, for example, displays the *Go to Line* dialog.

Having provided an overview of the elements that comprise the Android Studio editor, the remainder of this chapter will explore the key features of the editing environment in more detail.

8.2 Splitting the Editor Window

By default, the editor will display a single panel showing the content of the currently selected file. A particularly useful feature when working simultaneously with multiple source code files is the ability to split the editor into

multiple panes. To split the editor, right-click on a file tab within the editor window and select either the *Split Vertically* or *Split Horizontally* menu option. Figure 8-6, for example, shows the splitter in action with the editor split into three panels:

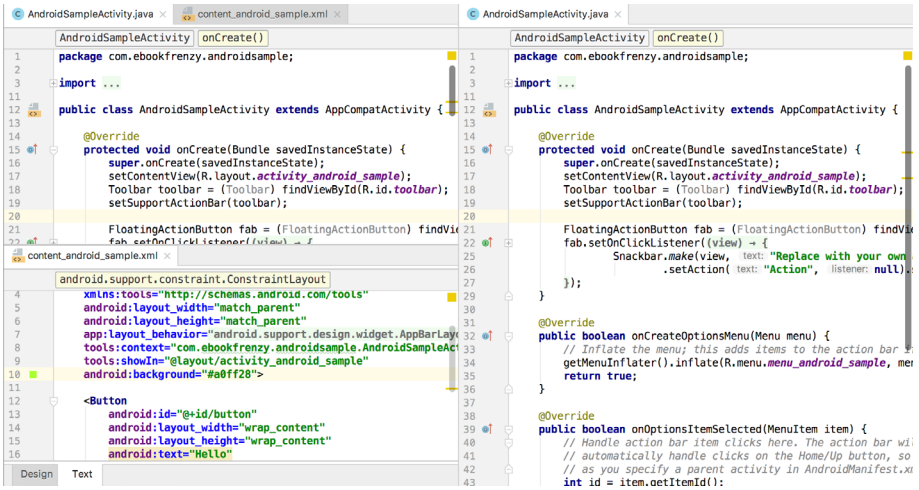

Figure 8-6

The orientation of a split panel may be changed at any time by right-clicking on the corresponding tab and selecting the *Change Splitter Orientation* menu option. Repeat these steps to unsplit a single panel, this time selecting the *Unsplit* option from the menu. All of the split panels may be removed by right-clicking on any tab and selecting the *Unsplit All* menu option.

Window splitting may be used to display different files, or to provide multiple windows onto the same file, allowing different areas of the same file to be viewed and edited concurrently.

8.3 Code Completion

The Android Studio editor has a considerable amount of built-in knowledge of Java programming syntax and the classes and methods that make up the Android SDK, as well as knowledge of your own code base. As code is typed, the editor scans what is being typed and, where appropriate, makes suggestions with regard to what might be needed to complete a statement or reference. When a completion suggestion is detected by the editor, a panel will appear containing a list of suggestions. In Figure 8-7, for example, the editor is suggesting possibilities for the beginning of a String declaration:

Figure 8-7

If none of the auto completion suggestions are correct, simply keep typing and the editor will continue to refine the suggestions where appropriate. To accept the top most suggestion, simply press the Enter or Tab key on the

keyboard. To select a different suggestion, use the arrow keys to move up and down the list, once again using the Enter or Tab key to select the highlighted item.

Completion suggestions can be manually invoked using the *Ctrl-Space* keyboard sequence. This can be useful when changing a word or declaration in the editor. When the cursor is positioned over a word in the editor, that word will automatically highlight. Pressing *Ctrl-Space* will display a list of alternate suggestions. To replace the current word with the currently highlighted item in the suggestion list, simply press the Tab key.

In addition to the real-time auto completion feature, the Android Studio editor also offers a system referred to as *Smart Completion*. Smart completion is invoked using the *Shift-Ctrl-Space* keyboard sequence and, when selected, will provide more detailed suggestions based on the current context of the code. Pressing the *Shift-Ctrl-Space* shortcut sequence a second time will provide more suggestions from a wider range of possibilities.

Code completion can be a matter of personal preference for many programmers. In recognition of this fact, Android Studio provides a high level of control over the auto completion settings. These can be viewed and modified by selecting the *File -> Settings...* menu option (or *Android Studio -> Preferences...* on macOS) and choosing *Editor -> General -> Code Completion* from the settings panel as shown in Figure 8-8:

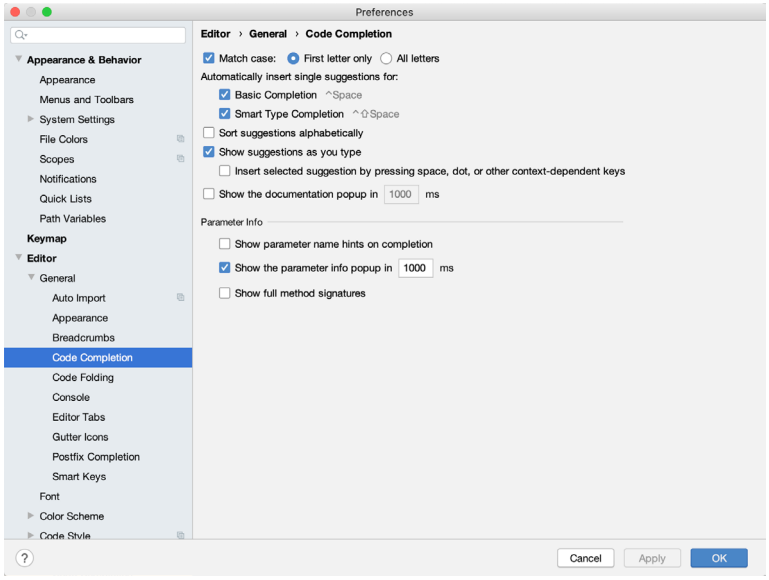

Figure 8-8

8.4 Statement Completion

Another form of auto completion provided by the Android Studio editor is statement completion. This can be used to automatically fill out the parentheses and braces for items such as methods and loop statements. Statement completion is invoked using the *Shift-Ctrl-Enter* (*Shift-Cmd-Enter* on macOS) keyboard sequence. Consider for example the following code:

```
myMethod()
```

Having typed this code into the editor, triggering statement completion will cause the editor to automatically add the braces to the method:

```
myMethod() {


}
```

8.5 Parameter Information

It is also possible to ask the editor to provide information about the argument parameters accepted by a method. With the cursor positioned between the brackets of a method call, the *Ctrl-P* (*Cmd-P* on macOS) keyboard sequence will display the parameters known to be accepted by that method, with the most likely suggestion highlighted in bold:

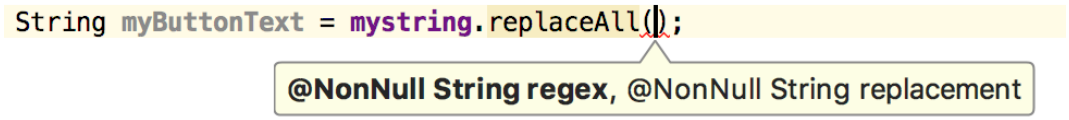

Figure 8-9

8.6 Parameter Name Hints

The code editor may be configured to display parameter name hints within method calls. Figure 8-10, for example, highlights the parameter name hints within the calls to the *make()* and *setAction()* methods of the *Snackbar* class:

```
FloatingActionButton fab = (FloatingActionButton) findViewById(R.id.fab);
fab.setOnClickListener((view) → {
        Snackbar.make(view, text: "Replace with your own action", Snackbar.LENGTH_LONG)
                .setAction( text: "Action",  listener: null).show();
});
```

Figure 8-10

The settings for this mode may be configured by selecting the *File -> Settings* (*Android Studio -> Preferences* on macOS) menu option followed by *Editor -> Appearance* in the left-hand panel. On the Appearance screen, enable or disable the *Show parameter name hints* option. To adjust the hint settings, click on the *Configure...* button, select the programming language and make any necessary adjustments.

8.7 Code Generation

In addition to completing code as it is typed the editor can, under certain conditions, also generate code for you. The list of available code generation options shown in Figure 8-11 can be accessed using the *Alt-Insert* (*Cmd-N* on macOS) keyboard shortcut when the cursor is at the location in the file where the code is to be generated.

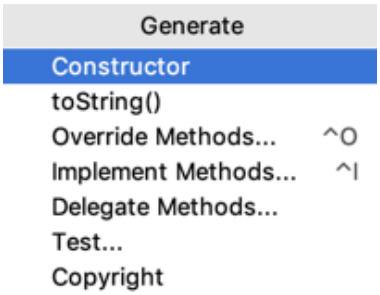

Figure 8-11

For the purposes of an example, consider a situation where we want to be notified when an Activity in our project is about to be destroyed by the operating system. As will be outlined in a later chapter of this book, this can be achieved by overriding the *onStop()* lifecycle method of the Activity superclass. To have Android Studio

generate a stub method for this, simply select the *Override Methods…* option from the code generation list and select the *onStop()* method from the resulting list of available methods:

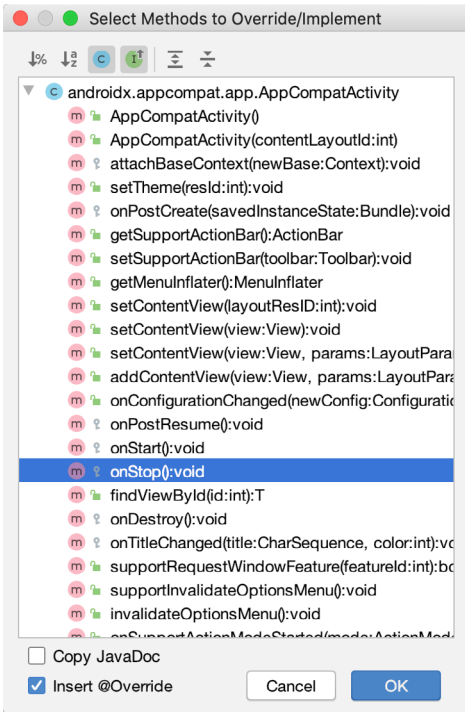

Figure 8-12

Having selected the method to override, clicking on *OK* will generate the stub method at the current cursor location in the Java source file as follows:

```
@Override
protected void onStop() {
    super.onStop();
}
```

8.8 Code Folding

Once a source code file reaches a certain size, even the most carefully formatted and well organized code can become overwhelming and difficult to navigate. Android Studio takes the view that it is not always necessary to have the content of every code block visible at all times. Code navigation can be made easier through the use of the *code folding* feature of the Android Studio editor. Code folding is controlled using markers appearing in the editor gutter at the beginning and end of each block of code in a source file. Figure 8-13, for example, highlights the start and end markers for a method declaration which is not currently folded:

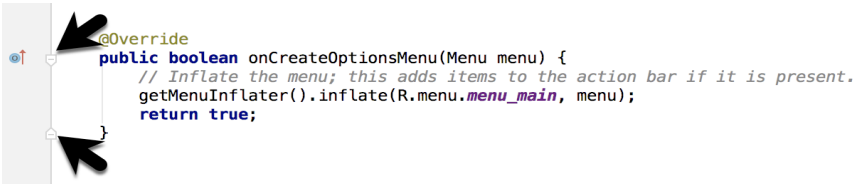

Figure 8-13

Clicking on either of these markers will fold the statement such that only the signature line is visible as shown in Figure 8-14:

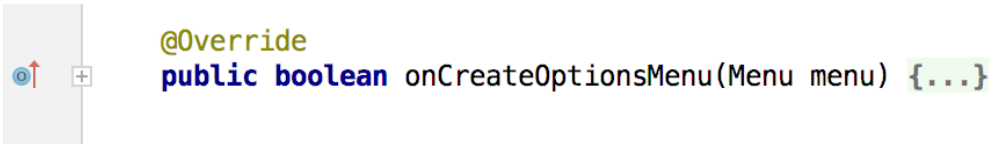

Figure 8-14

To unfold a collapsed section of code, simply click on the '+' marker in the editor gutter. To see the hidden code without unfolding it, hover the mouse pointer over the "{…}" indicator as shown in Figure 8-15. The editor will then display the lens overlay containing the folded code block:

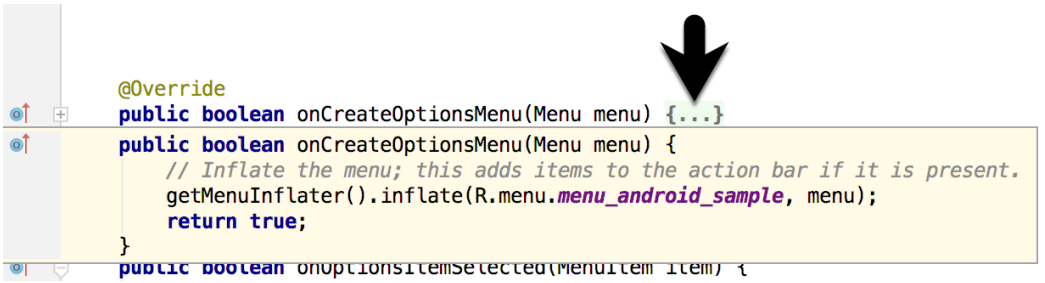

Figure 8-15

All of the code blocks in a file may be folded or unfolded using the *Ctrl-Shift-Plus* and *Ctrl-Shift-Minus* keyboard sequences.

By default, the Android Studio editor will automatically fold some code when a source file is opened. To configure the conditions under which this happens, select *File -> Settings…* (*Android Studio -> Preferences…* on macOS) and choose the *Editor -> General -> Code Folding* entry in the resulting settings panel (Figure 8-16):

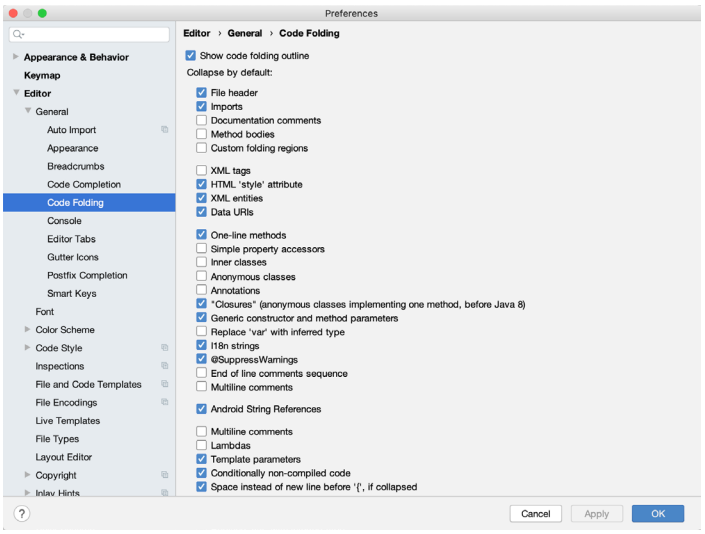

Figure 8-16

8.9 Quick Documentation Lookup

Context sensitive Java and Android documentation can be accessed by placing the cursor over the declaration for which documentation is required and pressing the *Ctrl-Q* keyboard shortcut (*Ctrl-J* on macOS). This will display a popup containing the relevant reference documentation for the item. Figure 8-17, for example, shows the documentation for the Android Snackbar class.

Figure 8-17

Once displayed, the documentation popup can be moved around the screen as needed. Clicking on the push pin icon located in the right-hand corner of the popup title bar will ensure that the popup remains visible once focus moves back to the editor, leaving the documentation visible as a reference while typing code.

8.10 Code Reformatting

In general, the Android Studio editor will automatically format code in terms of indenting, spacing and nesting of statements and code blocks as they are added. In situations where lines of code need to be reformatted (a common occurrence, for example, when cutting and pasting sample code from a web site), the editor provides a source code reformatting feature which, when selected, will automatically reformat code to match the prevailing code style.

To reformat source code, press the *Ctrl-Alt-L* (*Cmd-Opt-L* on macOS) keyboard shortcut sequence. To display the *Reformat Code* dialog (Figure 8-18) use the *Ctrl-Alt-Shift-L* (*Cmd-Opt-Shift-L* on macOS). This dialog provides the option to reformat only the currently selected code, the entire source file currently active in the editor or only code that has changed as the result of a source code control update.

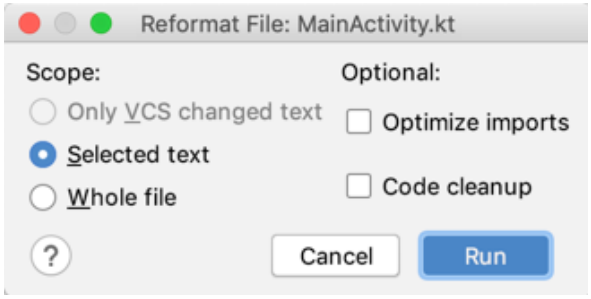

Figure 8-18

The full range of code style preferences can be changed from within the project settings dialog. Select the *File -> Settings* menu option (*Android Studio -> Preferences...* on macOS) and choose *Code Style* in the left-hand panel to access a list of supported programming and markup languages. Selecting a language will provide access to a vast array of formatting style options, all of which may be modified from the Android Studio default to match your preferred code style. To configure the settings for the *Rearrange code* option in the above dialog, for example, unfold the *Code Style* section, select Java and, from the Java settings, select the *Arrangement* tab.

8.11 Finding Sample Code

The Android Studio editor provides a way to access sample code relating to the currently highlighted entry within the code listing. This feature can be useful for learning how a particular Android class or method is used. To find sample code, highlight a method or class name in the editor, right-click on it and select the *Find Sample Code* menu option. The Find Sample Code panel (Figure 8-19) will appear beneath the editor with a list of matching samples. Selecting a sample from the list will load the corresponding code into the right-hand panel:

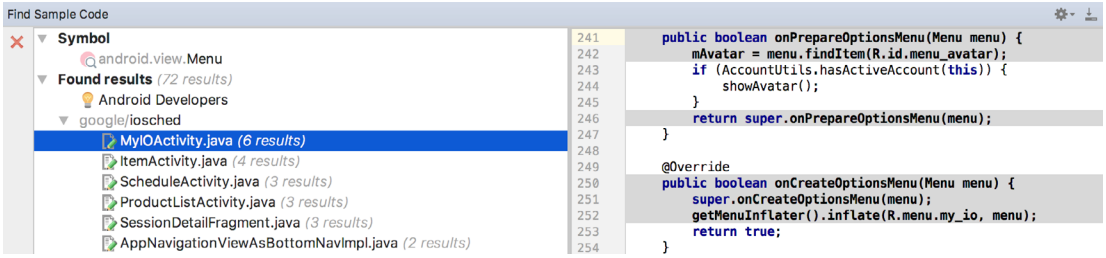

Figure 8-19

8.12 Summary

The Android Studio editor goes to great length to reduce the amount of typing needed to write code and to make that code easier to read and navigate. In this chapter we have covered a number of the key editor features including code completion, code generation, editor window splitting, code folding, reformatting and documentation lookup.

Chapter 9

9. An Overview of the Android Architecture

So far in this book, steps have been taken to set up an environment suitable for the development of Android applications using Android Studio. An initial step has also been taken into the process of application development through the creation of a simple Android Studio application project.

Before delving further into the practical matters of Android application development, however, it is important to gain an understanding of some of the more abstract concepts of both the Android SDK and Android development in general. Gaining a clear understanding of these concepts now will provide a sound foundation on which to build further knowledge.

Starting with an overview of the Android architecture in this chapter, and continuing in the next few chapters of this book, the goal is to provide a detailed overview of the fundamentals of Android development.

9.1 The Android Software Stack

Android is structured in the form of a software stack comprising applications, an operating system, run-time environment, middleware, services and libraries. This architecture can, perhaps, best be represented visually as outlined in Figure 9-1. Each layer of the stack, and the corresponding elements within each layer, are tightly integrated and carefully tuned to provide the optimal application development and execution environment for mobile devices. The remainder of this chapter will work through the different layers of the Android stack, starting at the bottom with the Linux Kernel.

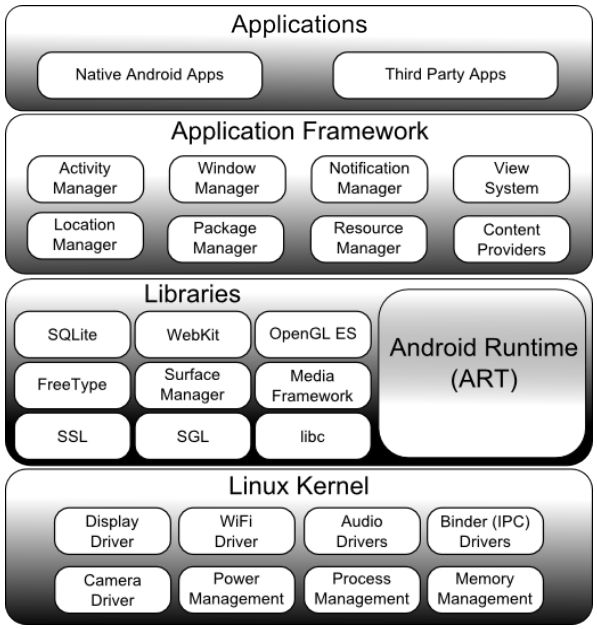

Figure 9-1

9.2 The Linux Kernel

Positioned at the bottom of the Android software stack, the Linux Kernel provides a level of abstraction between the device hardware and the upper layers of the Android software stack. Based on Linux version 2.6, the kernel provides preemptive multitasking, low-level core system services such as memory, process and power management in addition to providing a network stack and device drivers for hardware such as the device display, Wi-Fi and audio.

The original Linux kernel was developed in 1991 by Linus Torvalds and was combined with a set of tools, utilities and compilers developed by Richard Stallman at the Free Software Foundation to create a full operating system referred to as GNU/Linux. Various Linux distributions have been derived from these basic underpinnings such as Ubuntu and Red Hat Enterprise Linux.

It is important to note, however, that Android uses only the Linux kernel. That said, it is worth noting that the Linux kernel was originally developed for use in traditional computers in the form of desktops and servers. In fact, Linux is now most widely deployed in mission critical enterprise server environments. It is a testament to both the power of today's mobile devices and the efficiency and performance of the Linux kernel that we find this software at the heart of the Android software stack.

9.3 Android Runtime – ART

When an Android app is built within Android Studio it is compiled into an intermediate bytecode format (referred to as DEX format). When the application is subsequently loaded onto the device, the Android Runtime (ART) uses a process referred to as Ahead-of-Time (AOT) compilation to translate the bytecode down to the native instructions required by the device processor. This format is known as Executable and Linkable Format (ELF).

Each time the application is subsequently launched, the ELF executable version is run, resulting in faster application performance and improved battery life.

This contrasts with the Just-in-Time (JIT) compilation approach used in older Android implementations whereby the bytecode was translated within a virtual machine (VM) each time the application was launched.

9.4 Android Libraries

In addition to a set of standard Java development libraries (providing support for such general purpose tasks as string handling, networking and file manipulation), the Android development environment also includes the Android Libraries. These are a set of Java-based libraries that are specific to Android development. Examples of libraries in this category include the application framework libraries in addition to those that facilitate user interface building, graphics drawing and database access.

A summary of some key core Android libraries available to the Android developer is as follows:

- **android.app** – Provides access to the application model and is the cornerstone of all Android applications.
- **android.content** – Facilitates content access, publishing and messaging between applications and application components.
- **android.database** – Used to access data published by content providers and includes SQLite database management classes.
- **android.graphics** – A low-level 2D graphics drawing API including colors, points, filters, rectangles and canvases.
- **android.hardware** – Presents an API providing access to hardware such as the accelerometer and light sensor.

- **android.opengl** – A Java interface to the OpenGL ES 3D graphics rendering API.
- **android.os** – Provides applications with access to standard operating system services including messages, system services and inter-process communication.
- **android.media** – Provides classes to enable playback of audio and video.
- **android.net** – A set of APIs providing access to the network stack. Includes *android.net.wifi*, which provides access to the device's wireless stack.
- **android.print** – Includes a set of classes that enable content to be sent to configured printers from within Android applications.
- **android.provider** – A set of convenience classes that provide access to standard Android content provider databases such as those maintained by the calendar and contact applications.
- **android.text** – Used to render and manipulate text on a device display.
- **android.util** – A set of utility classes for performing tasks such as string and number conversion, XML handling and date and time manipulation.
- **android.view** – The fundamental building blocks of application user interfaces.
- **android.widget** - A rich collection of pre-built user interface components such as buttons, labels, list views, layout managers, radio buttons etc.
- **android.webkit** – A set of classes intended to allow web-browsing capabilities to be built into applications.

Having covered the Java-based libraries in the Android runtime, it is now time to turn our attention to the C/C++ based libraries contained in this layer of the Android software stack.

9.4.1 C/C++ Libraries

The Android runtime core libraries outlined in the preceding section are Java-based and provide the primary APIs for developers writing Android applications. It is important to note, however, that the core libraries do not perform much of the actual work and are, in fact, essentially Java "wrappers" around a set of C/C++ based libraries. When making calls, for example, to the *android.opengl* library to draw 3D graphics on the device display, the library actually ultimately makes calls to the *OpenGL ES* C++ library which, in turn, works with the underlying Linux kernel to perform the drawing tasks.

C/C++ libraries are included to fulfill a wide and diverse range of functions including 2D and 3D graphics drawing, Secure Sockets Layer (SSL) communication, SQLite database management, audio and video playback, bitmap and vector font rendering, display subsystem and graphic layer management and an implementation of the standard C system library (libc).

In practice, the typical Android application developer will access these libraries solely through the Java based Android core library APIs. In the event that direct access to these libraries is needed, this can be achieved using the Android Native Development Kit (NDK), the purpose of which is to call the native methods of non-Java or Kotlin programming languages (such as C and C++) from within Java code using the Java Native Interface (JNI).

9.5 Application Framework

The Application Framework is a set of services that collectively form the environment in which Android applications run and are managed. This framework implements the concept that Android applications are constructed from reusable, interchangeable and replaceable components. This concept is taken a step further in that an application is also able to *publish* its capabilities along with any corresponding data so that they can be

found and reused by other applications.

The Android framework includes the following key services:

- **Activity Manager** – Controls all aspects of the application lifecycle and activity stack.
- **Content Providers** – Allows applications to publish and share data with other applications.
- **Resource Manager** – Provides access to non-code embedded resources such as strings, color settings and user interface layouts.
- **Notifications Manager** – Allows applications to display alerts and notifications to the user.
- **View System** – An extensible set of views used to create application user interfaces.
- **Package Manager** – The system by which applications are able to find out information about other applications currently installed on the device.
- **Telephony Manager** – Provides information to the application about the telephony services available on the device such as status and subscriber information.
- **Location Manager** – Provides access to the location services allowing an application to receive updates about location changes.

9.6 Applications

Located at the top of the Android software stack are the applications. These comprise both the native applications provided with the particular Android implementation (for example web browser and email applications) and the third party applications installed by the user after purchasing the device.

9.7 Summary

A good Android development knowledge foundation requires an understanding of the overall architecture of Android. Android is implemented in the form of a software stack architecture consisting of a Linux kernel, a runtime environment and corresponding libraries, an application framework and a set of applications. Applications are predominantly written in Java or Kotlin and compiled down to bytecode format within the Android Studio build environment. When the application is subsequently installed on a device, this bytecode is compiled down by the Android Runtime (ART) to the native format used by the CPU. The key goals of the Android architecture are performance and efficiency, both in application execution and in the implementation of reuse in application design.

Chapter 10

10. The Anatomy of an Android Application

Regardless of your prior programming experiences, be it Windows, macOS, Linux or even iOS based, the chances are good that Android development is quite unlike anything you have encountered before.

The objective of this chapter, therefore, is to provide an understanding of the high-level concepts behind the architecture of Android applications. In doing so, we will explore in detail both the various components that can be used to construct an application and the mechanisms that allow these to work together to create a cohesive application.

10.1 Android Activities

Those familiar with object-oriented programming languages such as Java, Kotlin, C++ or C# will be familiar with the concept of encapsulating elements of application functionality into classes that are then instantiated as objects and manipulated to create an application. Since Android applications are written in Java and Kotlin, this is still very much the case. Android, however, also takes the concept of re-usable components to a higher level.

Android applications are created by bringing together one or more components known as *Activities*. An activity is a single, standalone module of application functionality that usually correlates directly to a single user interface screen and its corresponding functionality. An appointments application might, for example, have an activity screen that displays appointments set up for the current day. The application might also utilize a second activity consisting of a screen where new appointments may be entered by the user.

Activities are intended as fully reusable and interchangeable building blocks that can be shared amongst different applications. An existing email application, for example, might contain an activity specifically for composing and sending an email message. A developer might be writing an application that also has a requirement to send an email message. Rather than develop an email composition activity specifically for the new application, the developer can simply use the activity from the existing email application.

Activities are created as subclasses of the Android *Activity* class and must be implemented so as to be entirely independent of other activities in the application. In other words, a shared activity cannot rely on being called at a known point in a program flow (since other applications may make use of the activity in unanticipated ways) and one activity cannot directly call methods or access instance data of another activity. This, instead, is achieved using *Intents* and *Content Providers*.

By default, an activity cannot return results to the activity from which it was invoked. If this functionality is required, the activity must be specifically started as a *sub-activity* of the originating activity.

10.2 Android Fragments

An activity, as described above, typically represents a single user interface screen within an app. One option is to construct the activity using a single user interface layout and one corresponding activity class file. A better alternative, however, is to break the activity into different sections. Each of these sections is referred to as a fragment, each of which consists of part of the user interface layout and a matching class file (declared as a subclass of the Android *Fragment* class). In this scenario, an activity simply becomes a container into which one or more fragments are embedded.

In fact, fragments provide an efficient alternative to having each user interface screen represented by a separate activity. Instead, an app can consist of a single activity that switches between different fragments, each representing a different app screen.

10.3 Android Intents

Intents are the mechanism by which one activity is able to launch another and implement the flow through the activities that make up an application. Intents consist of a description of the operation to be performed and, optionally, the data on which it is to be performed.

Intents can be *explicit*, in that they request the launch of a specific activity by referencing the activity by class name, or *implicit* by stating either the type of action to be performed or providing data of a specific type on which the action is to be performed. In the case of implicit intents, the Android runtime will select the activity to launch that most closely matches the criteria specified by the Intent using a process referred to as *Intent Resolution*.

10.4 Broadcast Intents

Another type of Intent, the *Broadcast Intent*, is a system wide intent that is sent out to all applications that have registered an "interested" *Broadcast Receiver*. The Android system, for example, will typically send out Broadcast Intents to indicate changes in device status such as the completion of system start up, connection of an external power source to the device or the screen being turned on or off.

A Broadcast Intent can be *normal* (asynchronous) in that it is sent to all interested Broadcast Receivers at more or less the same time, or *ordered* in that it is sent to one receiver at a time where it can be processed and then either aborted or allowed to be passed to the next Broadcast Receiver.

10.5 Broadcast Receivers

Broadcast Receivers are the mechanism by which applications are able to respond to Broadcast Intents. A Broadcast Receiver must be registered by an application and configured with an *Intent Filter* to indicate the types of broadcast in which it is interested. When a matching intent is broadcast, the receiver will be invoked by the Android runtime regardless of whether the application that registered the receiver is currently running. The receiver then has 5 seconds in which to complete any tasks required of it (such as launching a Service, making data updates or issuing a notification to the user) before returning. Broadcast Receivers operate in the background and do not have a user interface.

10.6 Android Services

Android Services are processes that run in the background and do not have a user interface. They can be started and subsequently managed from activities, Broadcast Receivers or other Services. Android Services are ideal for situations where an application needs to continue performing tasks but does not necessarily need a user interface to be visible to the user. Although Services lack a user interface, they can still notify the user of events using notifications and *toasts* (small notification messages that appear on the screen without interrupting the currently visible activity) and are also able to issue Intents.

Services are given a higher priority by the Android runtime than many other processes and will only be terminated as a last resort by the system in order to free up resources. In the event that the runtime does need to kill a Service, however, it will be automatically restarted as soon as adequate resources once again become available. A Service can reduce the risk of termination by declaring itself as needing to run in the *foreground*. This is achieved by making a call to *startForeground()*. This is only recommended for situations where termination would be detrimental to the user experience (for example, if the user is listening to audio being streamed by the Service).

Example situations where a Service might be a practical solution include, as previously mentioned, the streaming

of audio that should continue when the application is no longer active, or a stock market tracking application that needs to notify the user when a share hits a specified price.

10.7 Content Providers

Content Providers implement a mechanism for the sharing of data between applications. Any application can provide other applications with access to its underlying data through the implementation of a Content Provider including the ability to add, remove and query the data (subject to permissions). Access to the data is provided via a Universal Resource Identifier (URI) defined by the Content Provider. Data can be shared in the form of a file or an entire SQLite database.

The native Android applications include a number of standard Content Providers allowing applications to access data such as contacts and media files. The Content Providers currently available on an Android system may be located using a *Content Resolver*.

10.8 The Application Manifest

The glue that pulls together the various elements that comprise an application is the Application Manifest file. It is within this XML based file that the application outlines the activities, services, broadcast receivers, data providers and permissions that make up the complete application.

10.9 Application Resources

In addition to the manifest file and the Dex files that contain the byte code, an Android application package will also typically contain a collection of *resource files*. These files contain resources such as the strings, images, fonts and colors that appear in the user interface together with the XML representation of the user interface layouts. By default, these files are stored in the */res* sub-directory of the application project's hierarchy.

10.10 Application Context

When an application is compiled, a class named *R* is created that contains references to the application resources. The application manifest file and these resources combine to create what is known as the *Application Context*. This context, represented by the Android *Context* class, may be used in the application code to gain access to the application resources at runtime. In addition, a wide range of methods may be called on an application's context to gather information and make changes to the application's environment at runtime.

10.11 Summary

A number of different elements can be brought together in order to create an Android application. In this chapter, we have provided a high-level overview of Activities, Fragments, Services, Intents and Broadcast Receivers together with an overview of the manifest file and application resources.

Maximum reuse and interoperability are promoted through the creation of individual, standalone modules of functionality in the form of activities and intents, while data sharing between applications is achieved by the implementation of content providers.

While activities are focused on areas where the user interacts with the application (an activity essentially equating to a single user interface screen and often made up of one or more fragments), background processing is typically handled by Services and Broadcast Receivers.

The components that make up the application are outlined for the Android runtime system in a manifest file which, combined with the application's resources, represents the application's context.

Much has been covered in this chapter that is most likely new to the average developer. Rest assured, however, that extensive exploration and practical use of these concepts will be made in subsequent chapters to ensure a solid knowledge foundation on which to build your own applications.