

# **Android Studio Flamingo Essentials**

---

Java Edition

Android Studio Flamingo Essentials – Java Edition

ISBN-13: 978-1-951442-70-5

© 2023 Neil Smyth / Payload Media, Inc. All Rights Reserved.

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

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

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

Rev: 1.0

## Table of Contents

<b>1. Introduction .....</b>	<b>1</b>
1.1 Downloading the Code Samples .....	1
1.2 Feedback .....	1
1.3 Errata .....	2
<b>2. Setting up an Android Studio Development Environment .....</b>	<b>3</b>
2.1 System requirements .....	3
2.2 Downloading the Android Studio package .....	3
2.3 Installing Android Studio .....	4
2.3.1 Installation on Windows .....	4
2.3.2 Installation on macOS .....	4
2.3.3 Installation on Linux .....	5
2.4 The Android Studio setup wizard .....	5
2.5 Installing additional Android SDK packages .....	6
2.6 Installing the Android SDK Command-line Tools .....	9
2.6.1 Windows 8.1 .....	10
2.6.2 Windows 10 .....	10
2.6.3 Windows 11 .....	11
2.6.4 Linux .....	11
2.6.5 macOS .....	11
2.7 Android Studio memory management .....	11
2.8 Updating Android Studio and the SDK .....	12
2.9 Summary .....	12
<b>3. Creating an Example Android App in Android Studio .....</b>	<b>13</b>
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 .....	24
3.9 Summary .....	25
<b>4. Creating an Android Virtual Device (AVD) in Android Studio .....</b>	<b>27</b>
4.1 About Android Virtual Devices .....	27
4.2 Starting the Emulator .....	29
4.3 Running the Application in the AVD .....	30
4.4 Running on Multiple Devices .....	31
4.5 Stopping a Running Application .....	32
4.6 Supporting Dark Theme .....	32
4.7 Running the Emulator in a Separate Window .....	33
4.8 Enabling the Device Frame .....	35

4.9 Summary .....	36
<b>5. Using and Configuring the Android Studio AVD Emulator .....</b>	<b>37</b>
5.1 The Emulator Environment .....	37
5.2 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 .....	40
5.5.2 Displays.....	40
5.5.3 Cellular .....	40
5.5.4 Battery.....	40
5.5.5 Camera.....	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 .....	41
5.5.11 Snapshots.....	41
5.5.12 Record and Playback .....	41
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 The Emulator in Tool Window Mode.....	43
5.9 Creating a Resizable Emulator.....	44
5.10 Summary .....	45
<b>6. A Tour of the Android Studio User Interface .....</b>	<b>47</b>
6.1 The Welcome Screen .....	47
6.2 The Main Window .....	48
6.3 The Tool Windows .....	49
6.4 Android Studio Keyboard Shortcuts .....	52
6.5 Switcher and Recent Files Navigation .....	53
6.6 Changing the Android Studio Theme .....	54
6.7 Summary .....	55
<b>7. Testing Android Studio Apps on a Physical Android Device.....</b>	<b>57</b>
7.1 An Overview of the Android Debug Bridge (ADB) .....	57
7.2 Enabling USB Debugging ADB on Android Devices.....	57
7.2.1 macOS ADB Configuration .....	58
7.2.2 Windows ADB Configuration .....	59
7.2.3 Linux adb Configuration.....	60
7.3 Resolving USB Connection Issues .....	60
7.4 Enabling Wireless Debugging on Android Devices .....	61
7.5 Testing the adb Connection .....	63
7.6 Device Mirroring.....	63
7.7 Summary .....	63
<b>8. The Basics of the Android Studio Code Editor.....</b>	<b>65</b>



8.1 The Android Studio Editor.....	65
8.2 Splitting the Editor Window.....	67
8.3 Code Completion.....	68
8.4 Statement Completion.....	69
8.5 Parameter Information.....	70
8.6 Parameter Name Hints.....	70
8.7 Code Generation.....	70
8.8 Code Folding.....	71
8.9 Quick Documentation Lookup.....	72
8.10 Code Reformatting.....	73
8.11 Finding Sample Code.....	74
8.12 Live Templates.....	74
8.13 Summary.....	75
<b>9. An Overview of the Android Architecture .....</b>	<b>77</b>
9.1 The Android Software Stack.....	77
9.2 The Linux Kernel.....	78
9.3 Android Runtime – ART.....	78
9.4 Android Libraries.....	78
9.4.1 C/C++ Libraries.....	79
9.5 Application Framework.....	79
9.6 Applications.....	80
9.7 Summary.....	80
<b>10. The Anatomy of an Android Application .....</b>	<b>81</b>
10.1 Android Activities.....	81
10.2 Android Fragments.....	81
10.3 Android Intents.....	82
10.4 Broadcast Intents.....	82
10.5 Broadcast Receivers.....	82
10.6 Android Services.....	82
10.7 Content Providers.....	83
10.8 The Application Manifest.....	83
10.9 Application Resources.....	83
10.10 Application Context.....	83
10.11 Summary.....	83
<b>11. An Overview of Android View Binding.....</b>	<b>85</b>
11.1 Find View by Id.....	85
11.2 View Binding.....	85
11.3 Converting the AndroidSample project.....	86
11.4 Enabling View Binding.....	86
11.5 Using View Binding.....	86
11.6 Choosing an Option.....	87
11.7 View Binding in the Book Examples.....	87
11.8 Migrating a Project to View Binding.....	88
11.9 Summary.....	88
<b>12. Understanding Android Application and Activity Lifecycles.....</b>	<b>91</b>
12.1 Android Applications and Resource Management.....	91
12.2 Android Process States.....	91

## Table of Contents

12.2.1 Foreground Process .....	92
12.2.2 Visible Process .....	92
12.2.3 Service Process .....	92
12.2.4 Background Process.....	92
12.2.5 Empty Process .....	93
12.3 Inter-Process Dependencies .....	93
12.4 The Activity Lifecycle.....	93
12.5 The Activity Stack.....	93
12.6 Activity States .....	94
12.7 Configuration Changes .....	94
12.8 Handling State Change.....	95
12.9 Summary .....	95
<b>13. Handling Android Activity State Changes.....</b>	<b>97</b>
13.1 New vs. Old Lifecycle Techniques.....	97
13.2 The Activity and Fragment Classes.....	97
13.3 Dynamic State vs. Persistent State.....	99
13.4 The Android Lifecycle Methods .....	100
13.5 Lifetimes .....	101
13.6 Foldable Devices and Multi-Resume .....	102
13.7 Disabling Configuration Change Restarts .....	102
13.8 Lifecycle Method Limitations.....	102
13.9 Summary .....	103
<b>14. Android Activity State Changes by Example .....</b>	<b>105</b>
14.1 Creating the State Change Example Project .....	105
14.2 Designing the User Interface .....	106
14.3 Overriding the Activity Lifecycle Methods .....	107
14.4 Filtering the Logcat Panel.....	109
14.5 Running the Application.....	110
14.6 Experimenting with the Activity.....	111
14.7 Summary .....	112
<b>15. Saving and Restoring the State of an Android Activity .....</b>	<b>113</b>
15.1 Saving Dynamic State .....	113
15.2 Default Saving of User Interface State .....	113
15.3 The Bundle Class .....	114
15.4 Saving the State.....	115
15.5 Restoring the State .....	116
15.6 Testing the Application.....	116
15.7 Summary .....	116
<b>16. Understanding Android Views, View Groups and Layouts .....</b>	<b>119</b>
16.1 Designing for Different Android Devices .....	119
16.2 Views and View Groups .....	119
16.3 Android Layout Managers .....	119
16.4 The View Hierarchy .....	121
16.5 Creating User Interfaces .....	122
16.6 Summary .....	122
<b>17. A Guide to the Android Studio Layout Editor Tool .....</b>	<b>123</b>

17.1 Basic vs. Empty Views Activity Templates .....	123
17.2 The Android Studio Layout Editor .....	127
17.3 Design Mode.....	127
17.4 The Palette .....	128
17.5 Design Mode and Layout Views.....	129
17.6 Night Mode .....	130
17.7 Code Mode.....	130
17.8 Split Mode .....	131
17.9 Setting Attributes.....	131
17.10 Transforms .....	133
17.11 Tools Visibility Toggles.....	134
17.12 Converting Views.....	135
17.13 Displaying Sample Data .....	136
17.14 Creating a Custom Device Definition .....	137
17.15 Changing the Current Device.....	137
17.16 Layout Validation .....	138
17.17 Summary .....	139
<b>18. A Guide to the Android ConstraintLayout.....</b>	<b>141</b>
18.1 How ConstraintLayout Works.....	141
18.1.1 Constraints.....	141
18.1.2 Margins.....	142
18.1.3 Opposing Constraints.....	142
18.1.4 Constraint Bias .....	143
18.1.5 Chains.....	144
18.1.6 Chain Styles.....	144
18.2 Baseline Alignment.....	145
18.3 Configuring Widget Dimensions.....	145
18.4 Guideline Helper .....	146
18.5 Group Helper .....	146
18.6 Barrier Helper .....	146
18.7 Flow Helper .....	148
18.8 Ratios .....	149
18.9 ConstraintLayout Advantages .....	149
18.10 ConstraintLayout Availability.....	150
18.11 Summary .....	150
<b>19. A Guide to Using ConstraintLayout in Android Studio .....</b>	<b>151</b>
19.1 Design and Layout Views.....	151
19.2 Autoconnect Mode .....	152
19.3 Inference Mode.....	153
19.4 Manipulating Constraints Manually.....	153
19.5 Adding Constraints in the Inspector .....	154
19.6 Viewing Constraints in the Attributes Window.....	155
19.7 Deleting Constraints.....	156
19.8 Adjusting Constraint Bias .....	156
19.9 Understanding ConstraintLayout Margins.....	157
19.10 The Importance of Opposing Constraints and Bias .....	158
19.11 Configuring Widget Dimensions.....	160
19.12 Design Time Tools Positioning .....	161

## Table of Contents

19.13 Adding Guidelines .....	162
19.14 Adding Barriers .....	164
19.15 Adding a Group .....	165
19.16 Working with the Flow Helper .....	166
19.17 Widget Group Alignment and Distribution .....	167
19.18 Converting other Layouts to ConstraintLayout .....	168
19.19 Summary .....	168
<b>20. Working with ConstraintLayout Chains and Ratios in Android Studio .....</b>	<b>169</b>
20.1 Creating a Chain .....	169
20.2 Changing the Chain Style .....	171
20.3 Spread Inside Chain Style .....	171
20.4 Packed Chain Style .....	172
20.5 Packed Chain Style with Bias .....	172
20.6 Weighted Chain .....	172
20.7 Working with Ratios .....	173
20.8 Summary .....	175
<b>21. An Android Studio Layout Editor ConstraintLayout Tutorial .....</b>	<b>177</b>
21.1 An Android Studio Layout Editor Tool Example .....	177
21.2 Preparing the Layout Editor Environment .....	177
21.3 Adding the Widgets to the User Interface .....	178
21.4 Adding the Constraints .....	181
21.5 Testing the Layout .....	182
21.6 Using the Layout Inspector .....	183
21.7 Summary .....	184
<b>22. Manual XML Layout Design in Android Studio .....</b>	<b>185</b>
22.1 Manually Creating an XML Layout .....	185
22.2 Manual XML vs. Visual Layout Design .....	188
22.3 Summary .....	188
<b>23. Managing Constraints using Constraint Sets .....</b>	<b>189</b>
23.1 Java Code vs. XML Layout Files .....	189
23.2 Creating Views .....	189
23.3 View Attributes .....	190
23.4 Constraint Sets .....	190
23.4.1 Establishing Connections .....	190
23.4.2 Applying Constraints to a Layout .....	190
23.4.3 Parent Constraint Connections .....	190
23.4.4 Sizing Constraints .....	191
23.4.5 Constraint Bias .....	191
23.4.6 Alignment Constraints .....	191
23.4.7 Copying and Applying Constraint Sets .....	191
23.4.8 ConstraintLayout Chains .....	191
23.4.9 Guidelines .....	192
23.4.10 Removing Constraints .....	192
23.4.11 Scaling .....	192
23.4.12 Rotation .....	193
23.5 Summary .....	193
<b>24. An Android ConstraintSet Tutorial .....</b>	<b>195</b>

24.1 Creating the Example Project in Android Studio .....	195
24.2 Adding Views to an Activity.....	195
24.3 Setting View Attributes.....	196
24.4 Creating View IDs.....	197
24.5 Configuring the Constraint Set .....	198
24.6 Adding the EditText View .....	199
24.7 Converting Density Independent Pixels (dp) to Pixels (px).....	200
24.8 Summary .....	201
<b>25. A Guide to using Apply Changes in Android Studio.....</b>	<b>203</b>
25.1 Introducing Apply Changes.....	203
25.2 Understanding Apply Changes Options .....	203
25.3 Using Apply Changes.....	204
25.4 Configuring Apply Changes Fallback Settings.....	205
25.5 An Apply Changes Tutorial.....	205
25.6 Using Apply Code Changes .....	205
25.7 Using Apply Changes and Restart Activity.....	206
25.8 Using Run App .....	206
25.9 Summary .....	206
<b>26. An Overview and Example of Android Event Handling .....</b>	<b>207</b>
26.1 Understanding Android Events.....	207
26.2 Using the android:onClick Resource .....	207
26.3 Event Listeners and Callback Methods .....	208
26.4 An Event Handling Example .....	208
26.5 Designing the User Interface .....	209
26.6 The Event Listener and Callback Method.....	209
26.7 Consuming Events .....	211
26.8 Summary .....	212
<b>27. Android Touch and Multi-touch Event Handling .....</b>	<b>213</b>
27.1 Intercepting Touch Events .....	213
27.2 The MotionEvent Object.....	213
27.3 Understanding Touch Actions.....	214
27.4 Handling Multiple Touches .....	214
27.5 An Example Multi-Touch Application .....	214
27.6 Designing the Activity User Interface .....	215
27.7 Implementing the Touch Event Listener.....	215
27.8 Running the Example Application.....	218
27.9 Summary .....	219
<b>28. Detecting Common Gestures Using the Android Gesture Detector Class .....</b>	<b>221</b>
28.1 Implementing Common Gesture Detection.....	221
28.2 Creating an Example Gesture Detection Project .....	222
28.3 Implementing the Listener Class.....	222
28.4 Creating the GestureDetectorCompat Instance.....	224
28.5 Implementing the onTouchEvent() Method.....	225
28.6 Testing the Application.....	225
28.7 Summary .....	226
<b>29. Implementing Custom Gesture and Pinch Recognition on Android .....</b>	<b>227</b>

## Table of Contents

29.1 The Android Gesture Builder Application.....	227
29.2 The GestureOverlayView Class .....	227
29.3 Detecting Gestures.....	227
29.4 Identifying Specific Gestures .....	227
29.5 Installing and Running the Gesture Builder Application .....	228
29.6 Creating a Gestures File .....	228
29.7 Creating the Example Project.....	228
29.8 Extracting the Gestures File from the SD Card .....	229
29.9 Adding the Gestures File to the Project .....	229
29.10 Designing the User Interface .....	229
29.11 Loading the Gestures File .....	230
29.12 Registering the Event Listener.....	231
29.13 Implementing the onGesturePerformed Method.....	231
29.14 Testing the Application.....	232
29.15 Configuring the GestureOverlayView.....	233
29.16 Intercepting Gestures.....	233
29.17 Detecting Pinch Gestures.....	233
29.18 A Pinch Gesture Example Project.....	234
29.19 Summary.....	236
<b>30. An Introduction to Android Fragments .....</b>	<b>237</b>
30.1 What is a Fragment? .....	237
30.2 Creating a Fragment .....	237
30.3 Adding a Fragment to an Activity using the Layout XML File.....	238
30.4 Adding and Managing Fragments in Code .....	240
30.5 Handling Fragment Events .....	241
30.6 Implementing Fragment Communication.....	242
30.7 Summary .....	243
<b>31. Using Fragments in Android Studio - An Example.....</b>	<b>245</b>
31.1 About the Example Fragment Application .....	245
31.2 Creating the Example Project.....	245
31.3 Creating the First Fragment Layout.....	245
31.4 Migrating a Fragment to View Binding .....	247
31.5 Adding the Second Fragment.....	248
31.6 Adding the Fragments to the Activity .....	249
31.7 Making the Toolbar Fragment Talk to the Activity .....	250
31.8 Making the Activity Talk to the Text Fragment .....	253
31.9 Testing the Application.....	254
31.10 Summary.....	255
<b>32. Modern Android App Architecture with Jetpack.....</b>	<b>257</b>
32.1 What is Android Jetpack? .....	257
32.2 The “Old” Architecture.....	257
32.3 Modern Android Architecture.....	257
32.4 The ViewModel Component .....	258
32.5 The LiveData Component.....	258
32.6 ViewModel Saved State.....	259
32.7 LiveData and Data Binding.....	260
32.8 Android Lifecycles .....	260
32.9 Repository Modules.....	260

32.10 Summary .....	261
<b>33. An Android ViewModel Tutorial .....</b>	<b>263</b>
33.1 About the Project .....	263
33.2 Creating the ViewModel Example Project.....	263
33.3 Removing Unwanted Project Elements.....	263
33.4 Designing the Fragment Layout.....	264
33.5 Implementing the View Model.....	265
33.6 Associating the Fragment with the View Model.....	266
33.7 Modifying the Fragment .....	267
33.8 Accessing the ViewModel Data .....	268
33.9 Testing the Project.....	268
33.10 Summary .....	269
<b>34. An Android Jetpack LiveData Tutorial .....</b>	<b>271</b>
34.1 LiveData - A Recap .....	271
34.2 Adding LiveData to the ViewModel.....	271
34.3 Implementing the Observer.....	273
34.4 Summary .....	275
<b>35. An Overview of Android Jetpack Data Binding .....</b>	<b>277</b>
35.1 An Overview of Data Binding.....	277
35.2 The Key Components of Data Binding .....	277
35.2.1 The Project Build Configuration .....	277
35.2.2 The Data Binding Layout File.....	278
35.2.3 The Layout File Data Element .....	279
35.2.4 The Binding Classes .....	280
35.2.5 Data Binding Variable Configuration.....	280
35.2.6 Binding Expressions (One-Way).....	281
35.2.7 Binding Expressions (Two-Way).....	282
35.2.8 Event and Listener Bindings.....	282
35.3 Summary .....	283
<b>36. An Android Jetpack Data Binding Tutorial.....</b>	<b>285</b>
36.1 Removing the Redundant Code .....	285
36.2 Enabling Data Binding .....	286
36.3 Adding the Layout Element .....	287
36.4 Adding the Data Element to Layout File.....	288
36.5 Working with the Binding Class .....	289
36.6 Assigning the ViewModel Instance to the Data Binding Variable .....	290
36.7 Adding Binding Expressions .....	290
36.8 Adding the Conversion Method .....	291
36.9 Adding a Listener Binding.....	291
36.10 Testing the App.....	292
36.11 Summary .....	292
<b>37. An Android ViewModel Saved State Tutorial.....</b>	<b>293</b>
37.1 Understanding ViewModel State Saving.....	293
37.2 Implementing ViewModel State Saving.....	293
37.3 Saving and Restoring State.....	295
37.4 Adding Saved State Support to the ViewModelDemo Project.....	295

37.5 Summary .....	297
<b>38. Working with Android Lifecycle-Aware Components .....</b>	<b>299</b>
38.1 Lifecycle Awareness .....	299
38.2 Lifecycle Owners .....	299
38.3 Lifecycle Observers .....	300
38.4 Lifecycle States and Events .....	301
38.5 Summary .....	302
<b>39. An Android Jetpack Lifecycle Awareness Tutorial .....</b>	<b>303</b>
39.1 Creating the Example Lifecycle Project .....	303
39.2 Creating a Lifecycle Observer .....	303
39.3 Adding the Observer .....	305
39.4 Testing the Observer .....	305
39.5 Creating a Lifecycle Owner .....	305
39.6 Testing the Custom Lifecycle Owner .....	307
39.7 Summary .....	308
<b>40. An Overview of the Navigation Architecture Component .....</b>	<b>309</b>
40.1 Understanding Navigation .....	309
40.2 Declaring a Navigation Host .....	310
40.3 The Navigation Graph .....	312
40.4 Accessing the Navigation Controller .....	313
40.5 Triggering a Navigation Action .....	313
40.6 Passing Arguments .....	314
40.7 Summary .....	314
<b>41. An Android Jetpack Navigation Component Tutorial .....</b>	<b>315</b>
41.1 Creating the NavigationDemo Project .....	315
41.2 Adding Navigation to the Build Configuration .....	315
41.3 Creating the Navigation Graph Resource File .....	316
41.4 Declaring a Navigation Host .....	317
41.5 Adding Navigation Destinations .....	318
41.6 Designing the Destination Fragment Layouts .....	320
41.7 Adding an Action to the Navigation Graph .....	321
41.8 Implement the OnFragmentInteractionListener .....	323
41.9 Adding View Binding Support to the Destination Fragments .....	324
41.10 Triggering the Action .....	324
41.11 Passing Data Using Safeargs .....	325
41.12 Summary .....	328
<b>42. An Introduction to MotionLayout .....</b>	<b>329</b>
42.1 An Overview of MotionLayout .....	329
42.2 MotionLayout .....	329
42.3 MotionScene .....	329
42.4 Configuring ConstraintSets .....	330
42.5 Custom Attributes .....	331
42.6 Triggering an Animation .....	332
42.7 Arc Motion .....	334
42.8 Keyframes .....	334
42.8.1 Attribute Keyframes .....	334



42.8.2 Position Keyframes .....	335
42.9 Time Linearity .....	338
42.10 KeyTrigger .....	338
42.11 Cycle and Time Cycle Keyframes .....	339
42.12 Starting an Animation from Code .....	339
42.13 Summary .....	340
<b>43. An Android MotionLayout Editor .....</b>	<b>341</b>
43.1 Creating the MotionLayoutDemo Project .....	341
43.2 ConstraintLayout to MotionLayout Conversion .....	341
43.3 Configuring Start and End Constraints .....	343
43.4 Previewing the MotionLayout Animation .....	345
43.5 Adding an OnClick Gesture .....	346
43.6 Adding an Attribute Keyframe to the Transition .....	347
43.7 Adding a CustomAttribute to a Transition .....	350
43.8 Adding Position Keyframes .....	351
43.9 Summary .....	354
<b>44. A MotionLayout KeyCycle Tutorial .....</b>	<b>355</b>
44.1 An Overview of Cycle Keyframes .....	355
44.2 Using the Cycle Editor .....	359
44.3 Creating the KeyCycleDemo Project .....	360
44.4 Configuring the Start and End Constraints .....	360
44.5 Creating the Cycles .....	362
44.6 Previewing the Animation .....	364
44.7 Adding the KeyFrameSet to the MotionScene .....	364
44.8 Summary .....	366
<b>45. Working with the Floating Action Button and Snackbar .....</b>	<b>367</b>
45.1 The Material Design .....	367
45.2 The Design Library .....	367
45.3 The Floating Action Button (FAB) .....	367
45.4 The Snackbar .....	368
45.5 Creating the Example Project .....	369
45.6 Reviewing the Project .....	369
45.7 Removing Navigation Features .....	370
45.8 Changing the Floating Action Button .....	371
45.9 Adding an Action to the Snackbar .....	372
45.10 Summary .....	372
<b>46. Creating a Tabbed Interface using the TabLayout Component .....</b>	<b>375</b>
46.1 An Introduction to the ViewPager2 .....	375
46.2 An Overview of the TabLayout Component .....	375
46.3 Creating the TabLayoutDemo Project .....	376
46.4 Creating the First Fragment .....	377
46.5 Duplicating the Fragments .....	378
46.6 Adding the TabLayout and ViewPager2 .....	379
46.7 Performing the Initialization Tasks .....	381
46.8 Testing the Application .....	383
46.9 Customizing the TabLayout .....	383
46.10 Summary .....	385

<b>47. Working with the RecyclerView and CardView Widgets .....</b>	<b>387</b>
47.1 An Overview of the RecyclerView .....	387
47.2 An Overview of the CardView .....	389
47.3 Summary .....	390
<b>48. An Android RecyclerView and CardView Tutorial .....</b>	<b>391</b>
48.1 Creating the CardDemo Project.....	391
48.2 Modifying the Basic Views Activity Project .....	391
48.3 Designing the CardView Layout.....	392
48.4 Adding the RecyclerView.....	393
48.5 Adding the Image Files.....	393
48.6 Creating the RecyclerView Adapter.....	394
48.7 Initializing the RecyclerView Component.....	396
48.8 Testing the Application.....	397
48.9 Responding to Card Selections.....	397
48.10 Summary.....	399
<b>49. A Layout Editor Sample Data Tutorial .....</b>	<b>401</b>
49.1 Adding Sample Data to a Project .....	401
49.2 Using Custom Sample Data .....	405
49.3 Summary .....	408
<b>50. Working with the AppBar and Collapsing Toolbar Layouts .....</b>	<b>409</b>
50.1 The Anatomy of an AppBar .....	409
50.2 The Example Project .....	410
50.3 Coordinating the RecyclerView and Toolbar .....	410
50.4 Introducing the Collapsing Toolbar Layout .....	412
50.5 Changing the Title and Scrim Color .....	415
50.6 Summary .....	416
<b>51. An Android Studio Primary/Detail Flow Tutorial .....</b>	<b>417</b>
51.1 The Primary/Detail Flow.....	417
51.2 Creating a Primary/Detail Flow Activity .....	418
51.3 Adding the Primary/Detail Flow Activity.....	418
51.4 Modifying the Primary/Detail Flow Template.....	419
51.5 Changing the Content Model.....	419
51.6 Changing the Detail Pane .....	421
51.7 Modifying the ItemDetailFragment Class .....	422
51.8 Modifying the ItemListFragment Class.....	423
51.9 Adding Manifest Permissions.....	424
51.10 Running the Application.....	424
51.11 Summary.....	425
<b>52. An Overview of Android Services.....</b>	<b>427</b>
52.1 Intent Service .....	427
52.2 Bound Service.....	427
52.3 The Anatomy of a Service .....	428
52.4 Controlling Destroyed Service Restart Options.....	428
52.5 Declaring a Service in the Manifest File.....	428
52.6 Starting a Service Running on System Startup.....	430
52.7 Summary .....	430

<b>53. An Overview of Android Intents .....</b>	<b>431</b>
53.1 An Overview of Intents .....	431
53.2 Explicit Intents.....	431
53.3 Returning Data from an Activity .....	432
53.4 Implicit Intents .....	433
53.5 Using Intent Filters.....	434
53.6 Automatic Link Verification .....	435
53.7 Manually Enabling Links .....	437
53.8 Checking Intent Availability .....	438
53.9 Summary .....	439
<b>54. Android Explicit Intents – A Worked Example .....</b>	<b>441</b>
54.1 Creating the Explicit Intent Example Application .....	441
54.2 Designing the User Interface Layout for MainActivity .....	441
54.3 Creating the Second Activity Class.....	442
54.4 Designing the User Interface Layout for SecondActivity .....	443
54.5 Reviewing the Application Manifest File .....	443
54.6 Creating the Intent .....	444
54.7 Extracting Intent Data .....	445
54.8 Launching SecondActivity as a Sub-Activity.....	446
54.9 Returning Data from a Sub-Activity.....	447
54.10 Testing the Application.....	447
54.11 Summary .....	447
<b>55. Android Implicit Intents – A Worked Example .....</b>	<b>449</b>
55.1 Creating the Android Studio Implicit Intent Example Project .....	449
55.2 Designing the User Interface .....	449
55.3 Creating the Implicit Intent .....	450
55.4 Adding a Second Matching Activity .....	451
55.5 Adding the Web View to the UI.....	451
55.6 Obtaining the Intent URL .....	452
55.7 Modifying the MyWebView Project Manifest File .....	453
55.8 Installing the MyWebView Package on a Device .....	454
55.9 Testing the Application.....	455
55.10 Manually Enabling the Link .....	455
55.11 Automatic Link Verification .....	457
55.12 Summary .....	459
<b>56. Android Broadcast Intents and Broadcast Receivers .....</b>	<b>461</b>
56.1 An Overview of Broadcast Intents .....	461
56.2 An Overview of Broadcast Receivers .....	462
56.3 Obtaining Results from a Broadcast .....	463
56.4 Sticky Broadcast Intents .....	463
56.5 The Broadcast Intent Example.....	464
56.6 Creating the Example Application .....	464
56.7 Creating and Sending the Broadcast Intent .....	464
56.8 Creating the Broadcast Receiver .....	465
56.9 Registering the Broadcast Receiver.....	466
56.10 Testing the Broadcast Example .....	467
56.11 Listening for System Broadcasts.....	467

56.12 Summary .....	468
<b>57. Android Local Bound Services – A Worked Example .....</b>	<b>469</b>
57.1 Understanding Bound Services .....	469
57.2 Bound Service Interaction Options .....	469
57.3 A Local Bound Service Example .....	469
57.4 Adding a Bound Service to the Project .....	470
57.5 Implementing the Binder .....	470
57.6 Binding the Client to the Service .....	473
57.7 Completing the Example .....	474
57.8 Testing the Application .....	475
57.9 Summary .....	475
<b>58. Android Remote Bound Services – A Worked Example .....</b>	<b>477</b>
58.1 Client to Remote Service Communication .....	477
58.2 Creating the Example Application .....	477
58.3 Designing the User Interface .....	477
58.4 Implementing the Remote Bound Service .....	478
58.5 Configuring a Remote Service in the Manifest File .....	479
58.6 Launching and Binding to the Remote Service .....	480
58.7 Sending a Message to the Remote Service .....	481
58.8 Summary .....	482
<b>59. A Basic Overview of Java Threads, Handlers and Executors .....</b>	<b>483</b>
59.1 The Application Main Thread .....	483
59.2 Thread Handlers .....	483
59.3 A Threading Example .....	483
59.4 Building the App .....	484
59.5 Creating a New Thread .....	485
59.6 Implementing a Thread Handler .....	486
59.7 Passing a Message to the Handler .....	488
59.8 Java Executor Concurrency .....	488
59.9 Working with Runnable Tasks .....	489
59.10 Shutting down an Executor Service .....	490
59.11 Working with Callable Tasks and Futures .....	490
59.12 Handling a Future Result .....	492
59.13 Scheduling Tasks .....	493
59.14 Summary .....	494
<b>60. Making Runtime Permission Requests in Android .....</b>	<b>495</b>
60.1 Understanding Normal and Dangerous Permissions .....	495
60.2 Creating the Permissions Example Project .....	497
60.3 Checking for a Permission .....	497
60.4 Requesting Permission at Runtime .....	499
60.5 Providing a Rationale for the Permission Request .....	500
60.6 Testing the Permissions App .....	502
60.7 Summary .....	502
<b>61. An Android Notifications Tutorial .....</b>	<b>503</b>
61.1 An Overview of Notifications .....	503
61.2 Creating the NotifyDemo Project .....	505

61.3 Designing the User Interface .....	505
61.4 Creating the Second Activity .....	505
61.5 Creating a Notification Channel .....	506
61.6 Requesting Notification Permission .....	507
61.7 Creating and Issuing a Notification .....	510
61.8 Launching an Activity from a Notification.....	512
61.9 Adding Actions to a Notification .....	514
61.10 Bundled Notifications.....	515
61.11 Summary .....	517
<b>62. An Android Direct Reply Notification Tutorial .....</b>	<b>519</b>
62.1 Creating the DirectReply Project .....	519
62.2 Designing the User Interface .....	519
62.3 Requesting Notification Permission .....	520
62.4 Creating the Notification Channel.....	521
62.5 Building the RemoteInput Object.....	522
62.6 Creating the PendingIntent.....	523
62.7 Creating the Reply Action.....	524
62.8 Receiving Direct Reply Input.....	526
62.9 Updating the Notification .....	527
62.10 Summary .....	529
<b>63. Foldable Devices and Multi-Window Support .....</b>	<b>531</b>
63.1 Foldables and Multi-Window Support.....	531
63.2 Using a Foldable Emulator.....	532
63.3 Entering Multi-Window Mode .....	533
63.4 Enabling and using Freeform Support .....	534
63.5 Checking for Freeform Support .....	534
63.6 Enabling Multi-Window Support in an App .....	534
63.7 Specifying Multi-Window Attributes .....	535
63.8 Detecting Multi-Window Mode in an Activity.....	536
63.9 Receiving Multi-Window Notifications .....	536
63.10 Launching an Activity in Multi-Window Mode .....	537
63.11 Configuring Freeform Activity Size and Position.....	537
63.12 Summary .....	538
<b>64. An Overview of Android SQLite Databases .....</b>	<b>539</b>
64.1 Understanding Database Tables .....	539
64.2 Introducing Database Schema .....	539
64.3 Columns and Data Types .....	539
64.4 Database Rows .....	540
64.5 Introducing Primary Keys .....	540
64.6 What is SQLite? .....	540
64.7 Structured Query Language (SQL).....	540
64.8 Trying SQLite on an Android Virtual Device (AVD) .....	541
64.9 The Android Room Persistence Library.....	543
64.10 Summary .....	543
<b>65. The Android Room Persistence Library .....</b>	<b>545</b>
65.1 Revisiting Modern App Architecture .....	545
65.2 Key Elements of Room Database Persistence.....	545

## Table of Contents

65.2.1 Repository .....	546
65.2.2 Room Database .....	546
65.2.3 Data Access Object (DAO) .....	546
65.2.4 Entities .....	546
65.2.5 SQLite Database .....	546
65.3 Understanding Entities.....	547
65.4 Data Access Objects.....	550
65.5 The Room Database.....	551
65.6 The Repository.....	552
65.7 In-Memory Databases .....	553
65.8 Database Inspector.....	553
65.9 Summary .....	553
<b>66. An Android TableLayout and TableRow Tutorial .....</b>	<b>555</b>
66.1 The TableLayout and TableRow Layout Views.....	555
66.2 Creating the Room Database Project .....	556
66.3 Converting to a LinearLayout.....	556
66.4 Adding the TableLayout to the User Interface.....	557
66.5 Configuring the TableRows .....	558
66.6 Adding the Button Bar to the Layout .....	559
66.7 Adding the RecyclerView.....	560
66.8 Adjusting the Layout Margins .....	561
66.9 Summary .....	561
<b>67. An Android Room Database and Repository Tutorial.....</b>	<b>563</b>
67.1 About the RoomDemo Project.....	563
67.2 Modifying the Build Configuration.....	563
67.3 Building the Entity .....	563
67.4 Creating the Data Access Object.....	565
67.5 Adding the Room Database.....	566
67.6 Adding the Repository .....	567
67.7 Adding the ViewModel .....	570
67.8 Creating the Product Item Layout .....	571
67.9 Adding the RecyclerView Adapter.....	571
67.10 Preparing the Main Activity .....	573
67.11 Adding the Button Listeners.....	574
67.12 Adding LiveData Observers .....	575
67.13 Initializing the RecyclerView.....	575
67.14 Testing the RoomDemo App.....	576
67.15 Using the Database Inspector.....	576
67.16 Summary .....	577
<b>68. Accessing Cloud Storage using the Android Storage Access Framework.....</b>	<b>579</b>
68.1 The Storage Access Framework.....	579
68.2 Working with the Storage Access Framework.....	580
68.3 Filtering Picker File Listings .....	580
68.4 Handling Intent Results.....	581
68.5 Reading the Content of a File .....	581
68.6 Writing Content to a File .....	582
68.7 Deleting a File.....	583
68.8 Gaining Persistent Access to a File.....	583

68.9 Summary .....	583
<b>69. An Android Storage Access Framework Example .....</b>	<b>585</b>
69.1 About the Storage Access Framework Example.....	585
69.2 Creating the Storage Access Framework Example.....	585
69.3 Designing the User Interface .....	585
69.4 Adding the Activity Launchers.....	586
69.5 Creating a New Storage File.....	588
69.6 Saving to a Storage File.....	588
69.7 Opening and Reading a Storage File .....	590
69.8 Testing the Storage Access Application .....	591
69.9 Summary .....	592
<b>70. Video Playback on Android using the VideoView and MediaController Classes.....</b>	<b>593</b>
70.1 Introducing the Android VideoView Class .....	593
70.2 Introducing the Android MediaController Class .....	594
70.3 Creating the Video Playback Example .....	594
70.4 Designing the VideoPlayer Layout .....	594
70.5 Downloading the Video File.....	595
70.6 Configuring the VideoView.....	595
70.7 Adding the MediaController to the Video View.....	597
70.8 Setting up the onPreparedListener .....	597
70.9 Summary .....	598
<b>71. Android Picture-in-Picture Mode.....</b>	<b>599</b>
71.1 Picture-in-Picture Features.....	599
71.2 Enabling Picture-in-Picture Mode.....	600
71.3 Configuring Picture-in-Picture Parameters .....	600
71.4 Entering Picture-in-Picture Mode .....	601
71.5 Detecting Picture-in-Picture Mode Changes .....	601
71.6 Adding Picture-in-Picture Actions.....	602
71.7 Summary .....	602
<b>72. An Android Picture-in-Picture Tutorial.....</b>	<b>605</b>
72.1 Adding Picture-in-Picture Support to the Manifest.....	605
72.2 Adding a Picture-in-Picture Button .....	605
72.3 Entering Picture-in-Picture Mode .....	606
72.4 Detecting Picture-in-Picture Mode Changes .....	607
72.5 Adding a Broadcast Receiver .....	608
72.6 Adding the PiP Action.....	609
72.7 Testing the Picture-in-Picture Action .....	612
72.8 Summary .....	612
<b>73. Android Audio Recording and Playback using MediaPlayer and MediaRecorder .....</b>	<b>613</b>
73.1 Playing Audio .....	613
73.2 Recording Audio and Video using the MediaRecorder Class.....	614
73.3 About the Example Project .....	615
73.4 Creating the AudioApp Project.....	615
73.5 Designing the User Interface .....	615
73.6 Checking for Microphone Availability.....	616
73.7 Initializing the Activity.....	617

## Table of Contents

73.8 Implementing the recordAudio() Method.....	618
73.9 Implementing the stopAudio() Method.....	618
73.10 Implementing the playAudio() method.....	619
73.11 Configuring and Requesting Permissions .....	619
73.12 Testing the Application.....	621
73.13 Summary .....	622
<b>74. Working with the Google Maps Android API in Android Studio .....</b>	<b>623</b>
74.1 The Elements of the Google Maps Android API .....	623
74.2 Creating the Google Maps Project.....	624
74.3 Creating a Google Cloud Billing Account .....	624
74.4 Creating a New Google Cloud Project .....	625
74.5 Enabling the Google Maps SDK.....	626
74.6 Generating a Google Maps API Key.....	627
74.7 Adding the API Key to the Android Studio Project .....	628
74.8 Testing the Application.....	629
74.9 Understanding Geocoding and Reverse Geocoding .....	629
74.10 Adding a Map to an Application.....	631
74.11 Requesting Current Location Permission.....	631
74.12 Displaying the User's Current Location .....	632
74.13 Changing the Map Type .....	634
74.14 Displaying Map Controls to the User.....	635
74.15 Handling Map Gesture Interaction.....	635
74.15.1 Map Zooming Gestures.....	636
74.15.2 Map Scrolling/Panning Gestures .....	636
74.15.3 Map Tilt Gestures.....	636
74.15.4 Map Rotation Gestures.....	636
74.16 Creating Map Markers.....	637
74.17 Controlling the Map Camera .....	637
74.18 Summary .....	639
<b>75. Printing with the Android Printing Framework .....</b>	<b>641</b>
75.1 The Android Printing Architecture .....	641
75.2 The Print Service Plugins .....	641
75.3 Google Cloud Print.....	642
75.4 Printing to Google Drive.....	642
75.5 Save as PDF .....	643
75.6 Printing from Android Devices .....	643
75.7 Options for Building Print Support into Android Apps.....	644
75.7.1 Image Printing .....	644
75.7.2 Creating and Printing HTML Content .....	645
75.7.3 Printing a Web Page.....	646
75.7.4 Printing a Custom Document .....	647
75.8 Summary .....	647
<b>76. An Android HTML and Web Content Printing Example .....</b>	<b>649</b>
76.1 Creating the HTML Printing Example Application .....	649
76.2 Printing Dynamic HTML Content .....	649
76.3 Creating the Web Page Printing Example.....	652
76.4 Removing the Floating Action Button .....	652
76.5 Removing Navigation Features.....	652



76.6 Designing the User Interface Layout .....	654
76.7 Accessing the WebView from the Main Activity .....	654
76.8 Loading the Web Page into the WebView .....	655
76.9 Adding the Print Menu Option .....	656
76.10 Summary .....	657
<b>77. A Guide to Android Custom Document Printing .....</b>	<b>659</b>
77.1 An Overview of Android Custom Document Printing .....	659
77.1.1 Custom Print Adapters .....	659
77.2 Preparing the Custom Document Printing Project .....	660
77.3 Creating the Custom Print Adapter .....	661
77.4 Implementing the onLayout() Callback Method .....	662
77.5 Implementing the onWrite() Callback Method .....	665
77.6 Checking a Page is in Range .....	667
77.7 Drawing the Content on the Page Canvas .....	668
77.8 Starting the Print Job .....	670
77.9 Testing the Application .....	671
77.10 Summary .....	671
<b>78. An Introduction to Android App Links .....</b>	<b>673</b>
78.1 An Overview of Android App Links .....	673
78.2 App Link Intent Filters .....	673
78.3 Handling App Link Intents .....	674
78.4 Associating the App with a Website .....	674
78.5 Summary .....	675
<b>79. An Android Studio App Links Tutorial .....</b>	<b>677</b>
79.1 About the Example App .....	677
79.2 The Database Schema .....	677
79.3 Loading and Running the Project .....	678
79.4 Adding the URL Mapping .....	679
79.5 Adding the Intent Filter .....	682
79.6 Adding Intent Handling Code .....	682
79.7 Testing the App .....	685
79.8 Creating the Digital Asset Links File .....	685
79.9 Testing the App Link .....	686
79.10 Summary .....	686
<b>80. An Android Biometric Authentication Tutorial .....</b>	<b>687</b>
80.1 An Overview of Biometric Authentication .....	687
80.2 Creating the Biometric Authentication Project .....	687
80.3 Configuring Device Fingerprint Authentication .....	688
80.4 Adding the Biometric Permission to the Manifest File .....	688
80.5 Designing the User Interface .....	689
80.6 Adding a Toast Convenience Method .....	689
80.7 Checking the Security Settings .....	690
80.8 Configuring the Authentication Callbacks .....	691
80.9 Adding the CancellationSignal .....	692
80.10 Starting the Biometric Prompt .....	692
80.11 Testing the Project .....	693
80.12 Summary .....	694

<b>81. Creating, Testing and Uploading an Android App Bundle .....</b>	<b>695</b>
81.1 The Release Preparation Process .....	695
81.2 Android App Bundles .....	695
81.3 Register for a Google Play Developer Console Account .....	696
81.4 Configuring the App in the Console .....	697
81.5 Enabling Google Play App Signing .....	698
81.6 Creating a Keystore File .....	698
81.7 Creating the Android App Bundle .....	700
81.8 Generating Test APK Files .....	701
81.9 Uploading the App Bundle to the Google Play Developer Console .....	702
81.10 Exploring the App Bundle .....	703
81.11 Managing Testers .....	704
81.12 Rolling the App Out for Testing .....	704
81.13 Uploading New App Bundle Revisions .....	705
81.14 Analyzing the App Bundle File .....	706
81.15 Summary .....	706
<b>82. An Overview of Android In-App Billing .....</b>	<b>709</b>
82.1 Preparing a Project for In-App Purchasing .....	709
82.2 Creating In-App Products and Subscriptions .....	709
82.3 Billing Client Initialization .....	710
82.4 Connecting to the Google Play Billing Library .....	711
82.5 Querying Available Products .....	712
82.6 Starting the Purchase Process .....	712
82.7 Completing the Purchase .....	713
82.8 Querying Previous Purchases .....	714
82.9 Summary .....	715
<b>83. An Android In-App Purchasing Tutorial .....</b>	<b>717</b>
83.1 About the In-App Purchasing Example Project .....	717
83.2 Creating the InAppPurchase Project .....	717
83.3 Adding Libraries to the Project .....	717
83.4 Designing the User Interface .....	718
83.5 Adding the App to the Google Play Store .....	718
83.6 Creating an In-App Product .....	719
83.7 Enabling License Testers .....	719
83.8 Initializing the Billing Client .....	720
83.9 Querying the Product .....	722
83.10 Launching the Purchase Flow .....	723
83.11 Handling Purchase Updates .....	723
83.12 Consuming the Product .....	724
83.13 Restoring a Previous Purchase .....	725
83.14 Testing the App .....	726
83.15 Troubleshooting .....	727
83.16 Summary .....	728
<b>84. An Overview of Android Dynamic Feature Modules .....</b>	<b>729</b>
84.1 An Overview of Dynamic Feature Modules .....	729
84.2 Dynamic Feature Module Architecture .....	729
84.3 Creating a Dynamic Feature Module .....	730

84.4 Converting an Existing Module for Dynamic Delivery.....	732
84.5 Working with Dynamic Feature Modules.....	735
84.6 Handling Large Dynamic Feature Modules .....	737
84.7 Summary .....	738
<b>85. An Android Studio Dynamic Feature Tutorial.....</b>	<b>739</b>
85.1 Creating the DynamicFeature Project.....	739
85.2 Adding Dynamic Feature Support to the Project .....	739
85.3 Designing the Base Activity User Interface .....	740
85.4 Adding the Dynamic Feature Module.....	741
85.5 Reviewing the Dynamic Feature Module.....	742
85.6 Adding the Dynamic Feature Activity.....	743
85.7 Implementing the <code>launchIntent()</code> Method.....	746
85.8 Uploading the App Bundle for Testing.....	747
85.9 Implementing the <code>installFeature()</code> Method .....	748
85.10 Adding the Update Listener.....	750
85.11 Using Deferred Installation .....	753
85.12 Removing a Dynamic Module .....	753
85.13 Summary .....	754
<b>86. Working with Material Design 3 Theming .....</b>	<b>755</b>
86.1 Material Design 2 vs Material Design 3 .....	755
86.2 Understanding Material Design Theming.....	755
86.3 Material Design 3 Theming .....	755
86.4 Building a Custom Theme.....	757
86.5 Summary .....	758
<b>87. A Material Design 3 Theming and Dynamic Color Tutorial.....</b>	<b>759</b>
87.1 Creating the ThemeDemo Project .....	759
87.2 Designing the User Interface .....	759
87.3 Building a New Theme .....	761
87.4 Adding the Theme to the Project.....	762
87.5 Enabling Dynamic Color Support .....	763
87.6 Previewing Dynamic Colors.....	764
87.7 Summary .....	765
<b>88. An Overview of Gradle in Android Studio.....</b>	<b>767</b>
88.1 An Overview of Gradle .....	767
88.2 Gradle and Android Studio .....	767
88.2.1 Sensible Defaults .....	767
88.2.2 Dependencies.....	767
88.2.3 Build Variants .....	768
88.2.4 Manifest Entries .....	768
88.2.5 APK Signing.....	768
88.2.6 ProGuard Support.....	768
88.3 The Property and Settings Gradle Build File.....	768
88.4 The Top-level Gradle Build File.....	769
88.5 Module Level Gradle Build Files.....	770
88.6 Configuring Signing Settings in the Build File.....	772
88.7 Running Gradle Tasks from the Command-line .....	773
88.8 Summary .....	774

**Index** ..... 775

## 1. Introduction

Fully updated for Android Studio Flamingo, this book aims to teach you how to develop Android-based applications using the Java programming language.

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

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

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

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

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

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

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

### 1.1 Downloading the Code Samples

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

<https://www.ebookfrenzy.com/retail/flamingojava/index.php>

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

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

### 1.2 Feedback

We want you to be satisfied with your purchase of this book. If you find any errors in the book, or have any comments, questions or concerns please contact us at [feedback@ebookfrenzy.com](mailto: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/flamingojava.html>*

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

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

### 2.2 Downloading the Android Studio package

Most of the work involved in developing applications for Android will be performed using the Android Studio environment. The content and examples in this book were created based on Android Studio Flamingo 2022.2.1 using the Android API 33 SDK (Tiramisu) which, at the time of writing, are the latest 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 Flamingo” should provide the option to download the older version if these differences become a problem. Alternatively, visit the following web page to find Android Studio Flamingo 2022.2.1 in the archives:

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

## 2.3 Installing Android Studio

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

### 2.3.1 Installation on Windows

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

Once the Android Studio setup wizard appears, work through the various screens to configure the installation to meet your requirements in terms of the file system location into which Android Studio should be installed and whether or not it should be made available to other 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 taskbar for easy access by navigating to the *Android Studio\bin* directory, right-clicking on the *studio64* executable, and selecting the *Pin to Taskbar* menu option (on Windows 11 this option can be found by selecting *Show more options* from the menu).

### 2.3.2 Installation on macOS

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

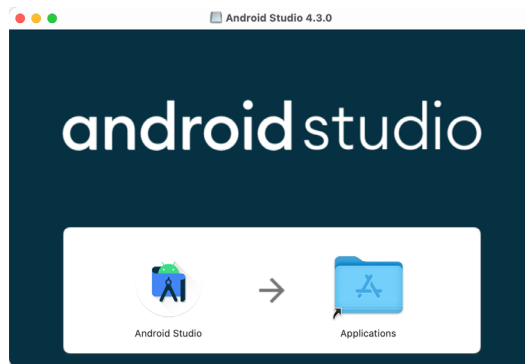


Figure 2-1

To install the package, 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 that will typically take a few seconds to complete.

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

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



### 2.3.3 Installation on Linux

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

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

Note that the Android Studio bundle will be installed into a subdirectory 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 may 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

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

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

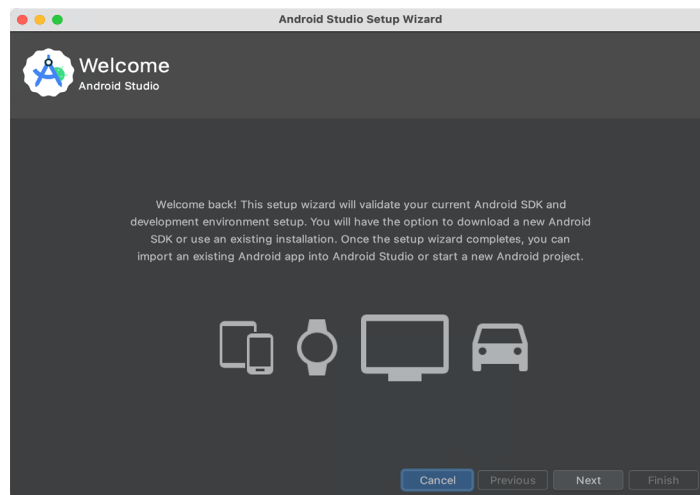


Figure 2-2

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

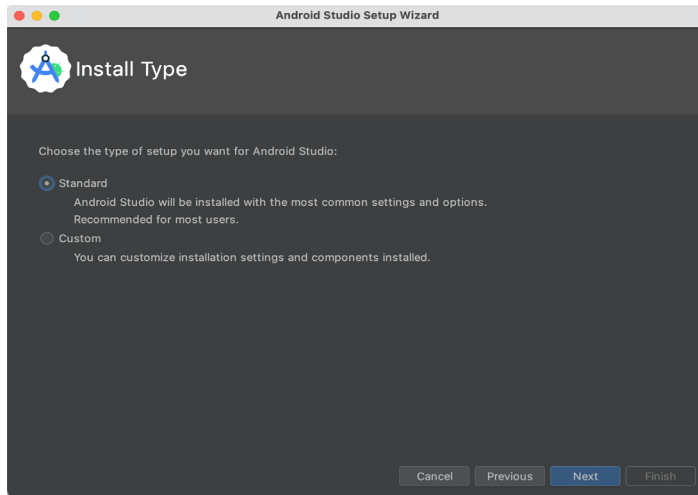


Figure 2-3

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

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

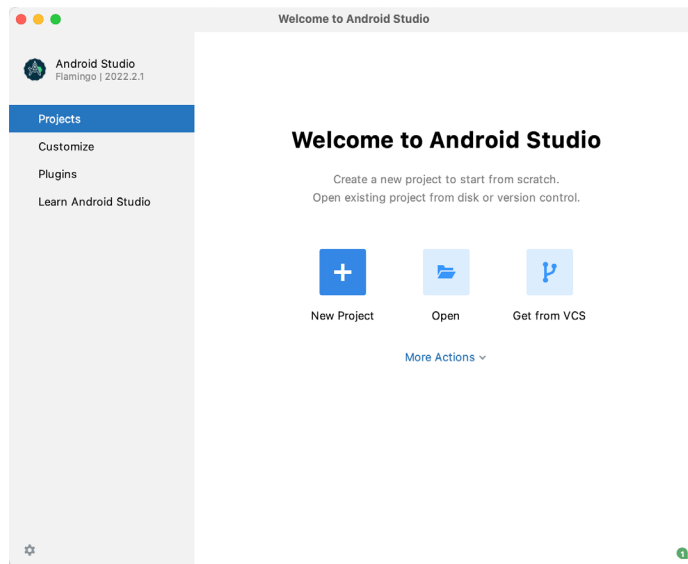


Figure 2-4

## 2.5 Installing additional Android SDK packages

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

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

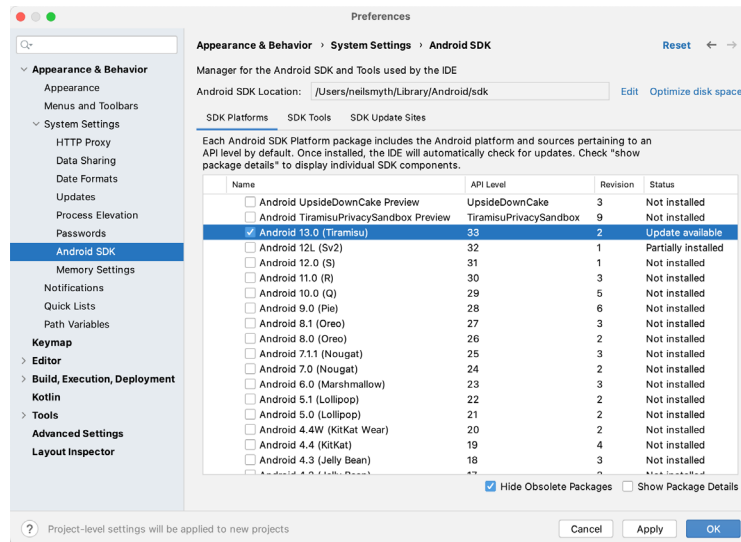


Figure 2-5

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. The rest of this book assumes that the Android Tiramisu (API Level 33) SDK is installed.

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

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

Name	API Level	Revision	Status
<input type="checkbox"/> Android TV ARM 64 v8a System Image	33	5	Not installed
<input type="checkbox"/> Android TV Intel x86 Atom System Image	33	5	Not installed
<input type="checkbox"/> Google TV ARM 64 v8a System Image	33	5	Not installed
<input type="checkbox"/> Google TV Intel x86 Atom System Image	33	5	Not installed
<input checked="" type="checkbox"/> Google APIs ARM 64 v8a System Image	33	8	Update Available: 9
<input type="checkbox"/> Google APIs Intel x86 Atom_64 System Image	33	9	Not installed
<input checked="" type="checkbox"/> Google Play ARM 64 v8a System Image	33	7	Installed

Figure 2-6

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

In addition to the Android SDK packages, several tools are also installed for building Android applications. To

## Setting up an Android Studio Development Environment

view the currently installed packages and check for updates, remain within the SDK settings screen and select the SDK Tools tab as shown in Figure 2-7:

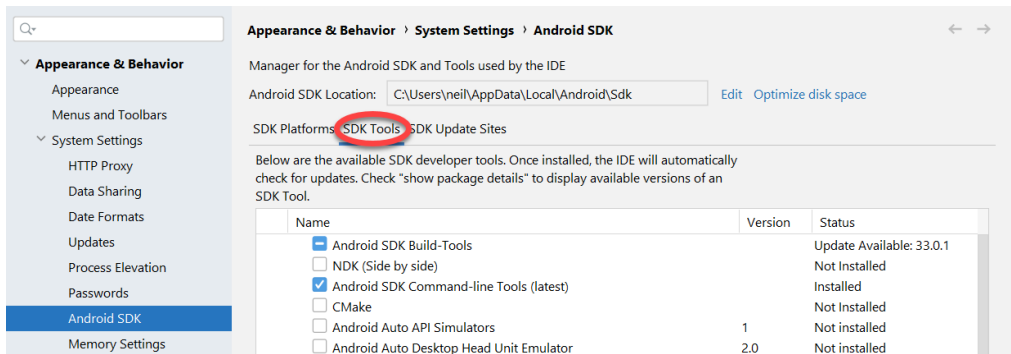


Figure 2-7

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

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

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

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

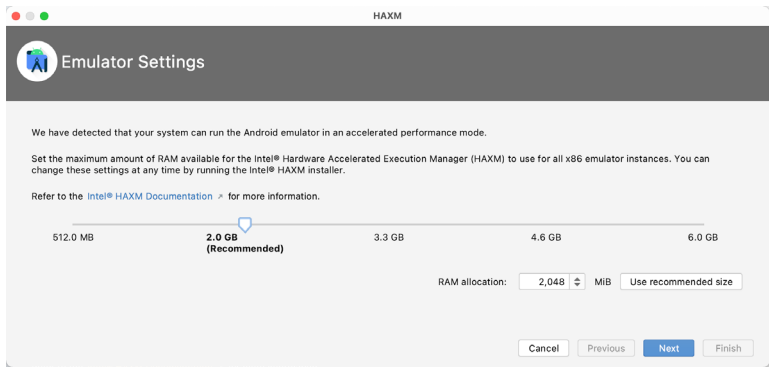


Figure 2-8

Once the installation is complete, review the package list and 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 Installing the Android SDK Command-line Tools

Android Studio includes a set of tools that allow some tasks to be performed from your operating system command line. To install these tools on your system, open the SDK Manager, select the SDK Tools tab and enable the *Show Package Details* option in the bottom left-hand corner of the window. Next, scroll down the list of packages and, when the *Android SDK Command-line Tools (latest)* package comes into view, enable it as shown in Figure 2-9:

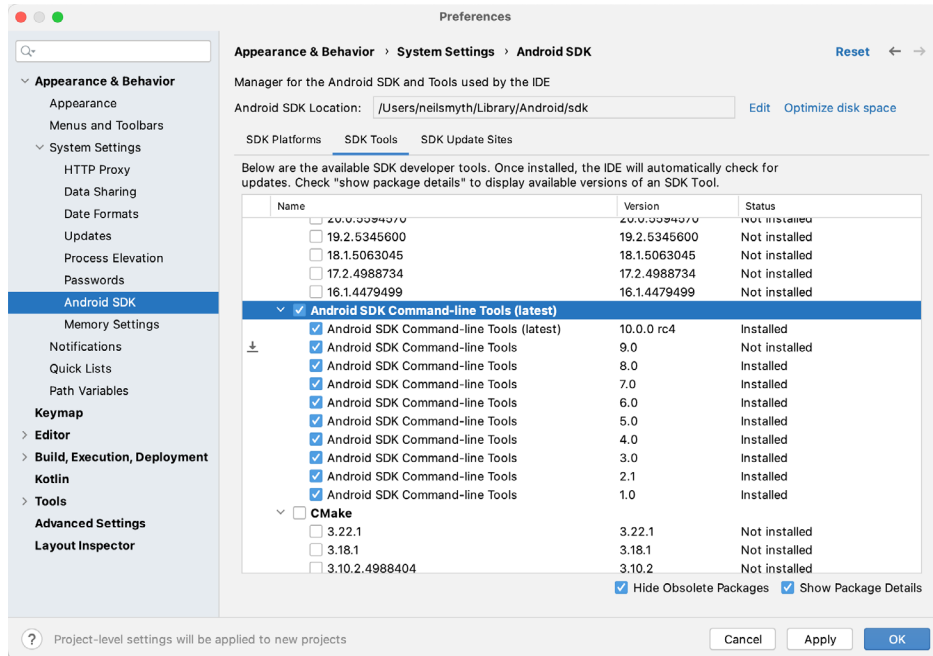


Figure 2-9

After you have selected the command-line tools package, click on *Apply* followed by *OK* to complete the installation. When the installation completes, click *Finish* and close the SDK Manager dialog.

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

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

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

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

## Setting up an Android Studio Development Environment

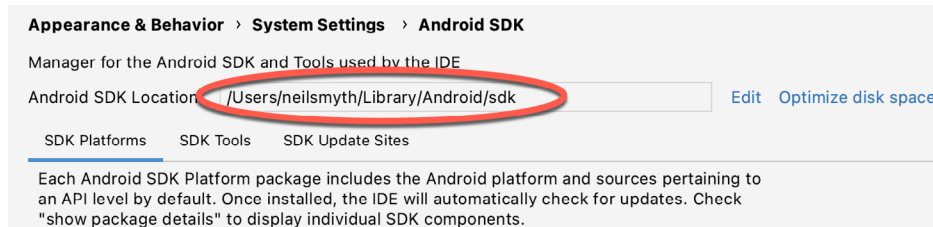


Figure 2-10

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

### 2.6.1 Windows 8.1

1. On the start screen, move the mouse to the bottom right-hand corner of the screen and select Search from the resulting menu. In the search box, enter Control Panel. When the Control Panel icon appears in the results area, click on it to launch the tool on the desktop.
2. Within the Control Panel, use the Category menu to change the display to Large Icons. From the list of icons select the one labeled System.
3. In the Environment Variables dialog, locate the Path variable in the System variables list, select it and click on the *Edit...* button. Using the *New* button in the edit dialog, add two new entries to the path. For example, assuming the Android SDK was installed into C:\Users\demo\AppData\Local\Android\Sdk, the following entries would need to be added:

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

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

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

```
echo %Path%
```

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

```
adb
```

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

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

```
avdmanager
```

If a message similar to the following message appears for one or both of the commands, it is most likely that an incorrect path was appended to the Path environment variable:

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

### 2.6.2 Windows 10

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

## 2.6.3 Windows 11

Right-click on the Start icon located in the taskbar and select Settings from the resulting menu. When the Settings dialog appears, scroll down the list of categories and select the “About” option. In the About screen, select *Advanced system settings* from the Related links section. When the System Properties window appears, click on the *Environment Variables...* button. Follow the steps outlined for Windows 8.1 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/cmdline-tools/latest/bin:/home/demo/android-studio/bin:$PATH
```

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

## 2.6.5 macOS

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

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

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

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

## 2.7 Android Studio memory management

Android Studio is a large and complex software application 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. These improvements include allowing the user to configure the amount of memory used by both the Android Studio IDE and the background processes used to build and run apps. This allows the software to take advantage of systems with larger amounts of RAM.

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

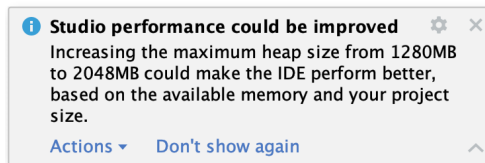


Figure 2-11

To view and modify the current memory configuration, select the *File -> Settings...* (*Android Studio ->*

## Setting up an Android Studio Development Environment

*Preferences...* on macOS) menu option and, in the resulting dialog, select *Appearance & Behavior* followed by the *Memory Settings* option listed under *System Settings* in the left-hand navigation panel, as illustrated in Figure 2-12 below:

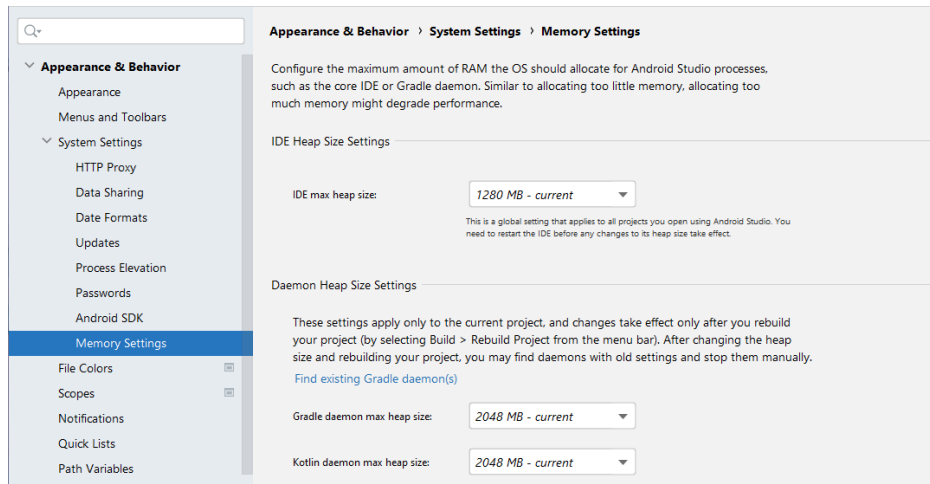


Figure 2-12

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

The IDE heap size setting adjusts the memory allocated to Android Studio and applies regardless of the currently loaded project. On the other hand, when a project is built and run from within Android Studio, 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 could be improved by adjusting the daemon heap settings. Unlike the IDE heap settings, these daemon settings apply only to the current project and can only be accessed when a project is open in Android Studio. To display the SDK Manager from within an open project, select the *Tools -> SDK Manager...* menu option.

## 2.8 Updating Android Studio and the SDK

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

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

## 2.9 Summary

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



## 3. Creating an Example Android App in Android Studio

The preceding chapters of this book have covered the steps necessary to configure an environment suitable for 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 an 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 rudimentary currency conversion calculator (so simple, in fact, that it only converts from dollars to euros and does so using an estimated conversion rate). The project will also 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:

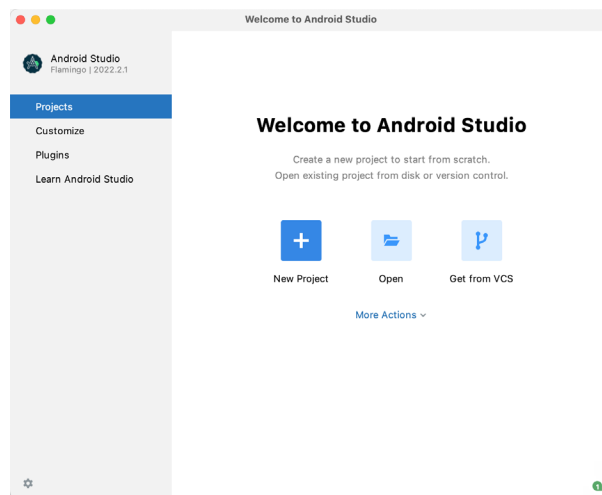


Figure 3-1

Once this window appears, Android Studio is ready for a new project to be created. To create the new project,

## Creating an Example Android App in Android Studio

simply click on the *New 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 *Phone and Tablet* option from the Templates panel followed by the option to create an *Empty Views Activity*. The Empty Views Activity option creates a template user interface consisting of a single TextView object.

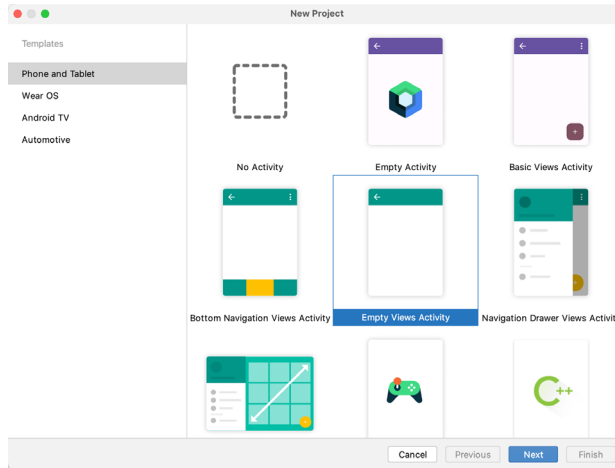


Figure 3-2

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

### 3.4 Defining the Project and SDK Settings

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

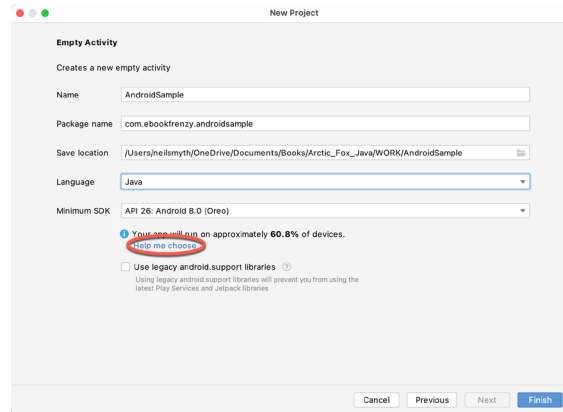


Figure 3-3

Finally, change the *Language* menu to *Java* and 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.

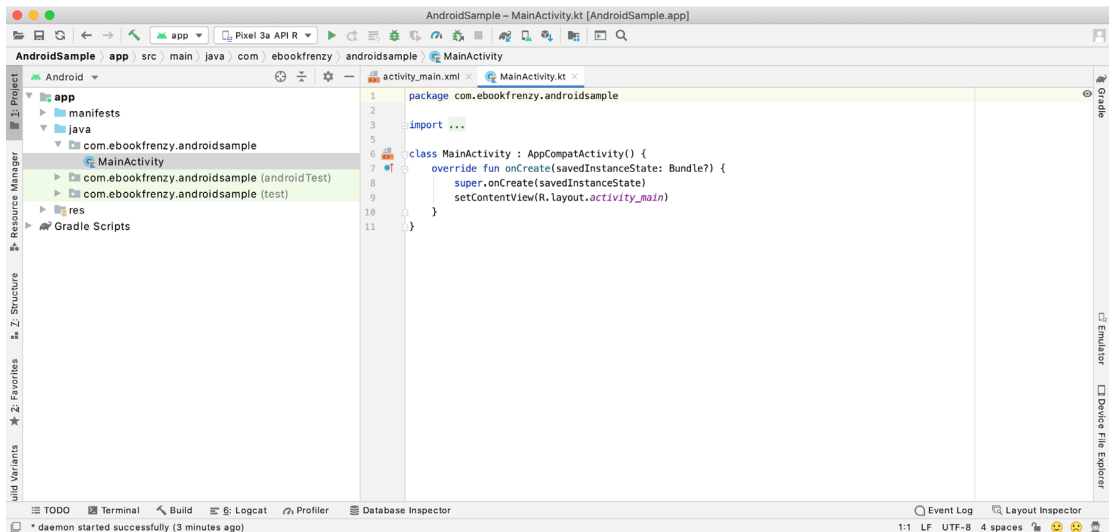


Figure 3-4

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:

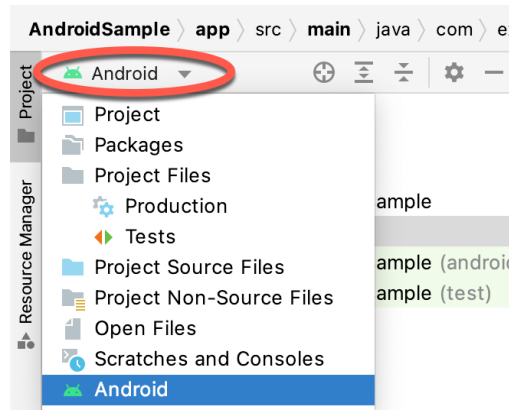


Figure 3-5

### 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:

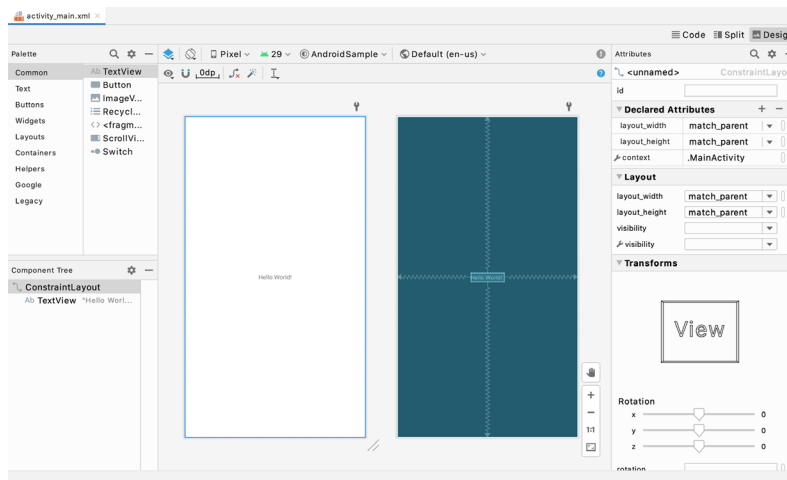




Figure 3-6

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. Use the night mode button (  ) to turn Night mode on and off.

As we can see in the device screen, the content layout already includes a label that displays a “Hello World!” message. Running down the left-hand side of the panel is a palette containing different categories of user interface components that may be used to construct a user interface, such as buttons, labels and text fields. 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:

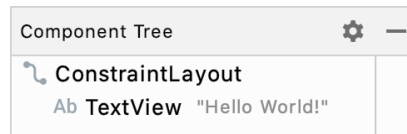


Figure 3-7

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:

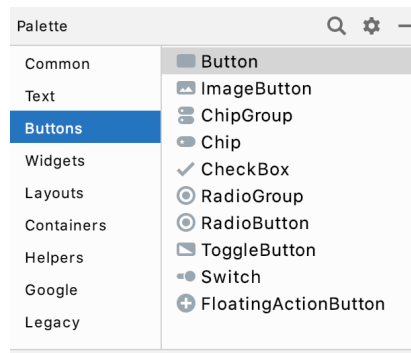


Figure 3-9

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:

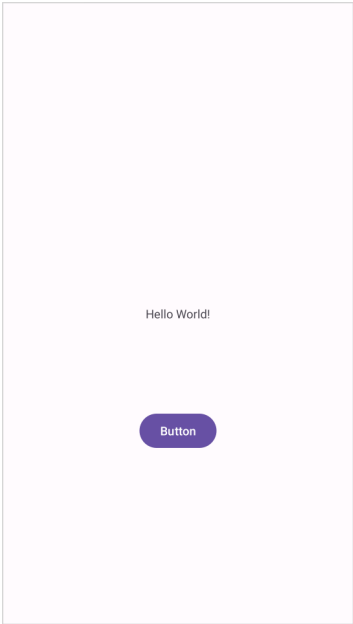


Figure 3-10

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:

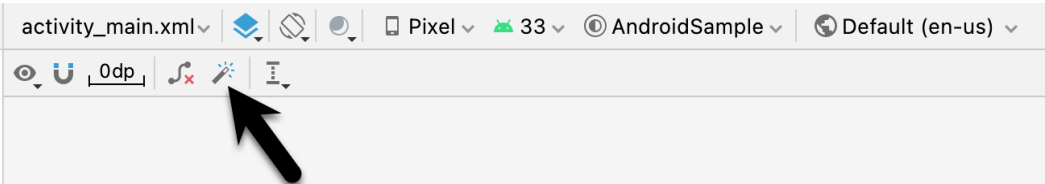


Figure 3-12

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:

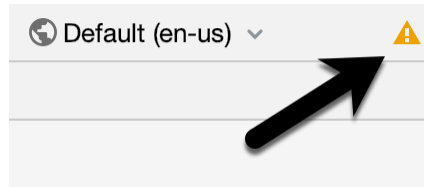


Figure 3-13

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

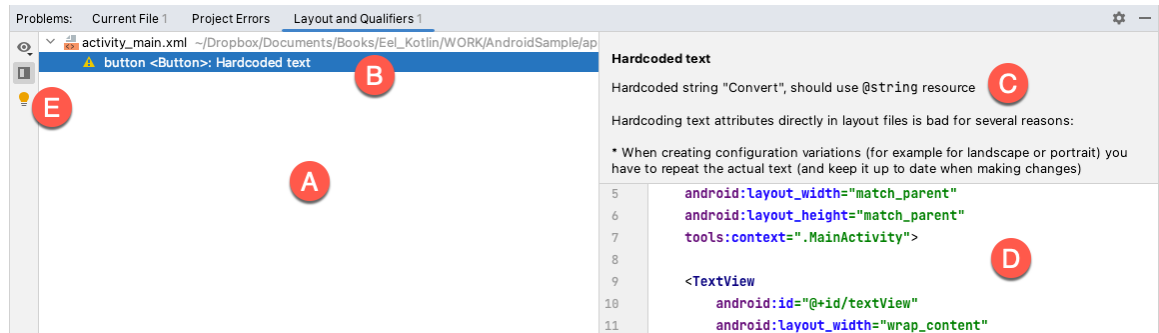


Figure 3-14

This tool window is divided into two panels. The left panel (marked A in the above figure) lists issues detected within the layout file. In our example, only the following problem is listed:

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

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

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

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

This I18N message 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”.

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



Figure 3-15

After this option has been selected, the *Extract Resource* panel (Figure 3-16) will appear. Within this panel, change the resource name field to *convert\_string* and leave the resource value set to *Convert* before clicking on the OK button.

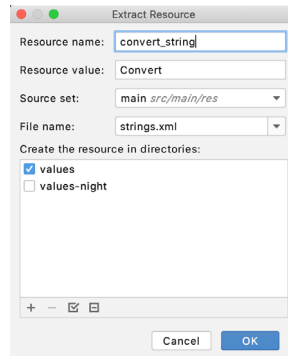


Figure 3-16

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

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

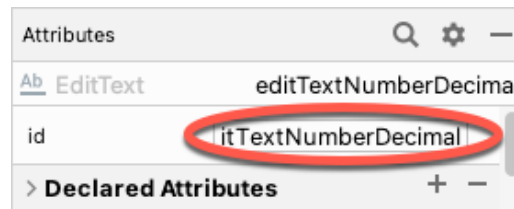


Figure 3-17

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



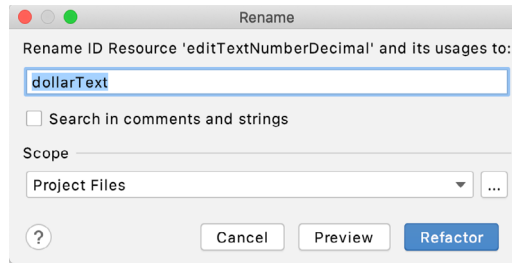


Figure 3-18

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

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

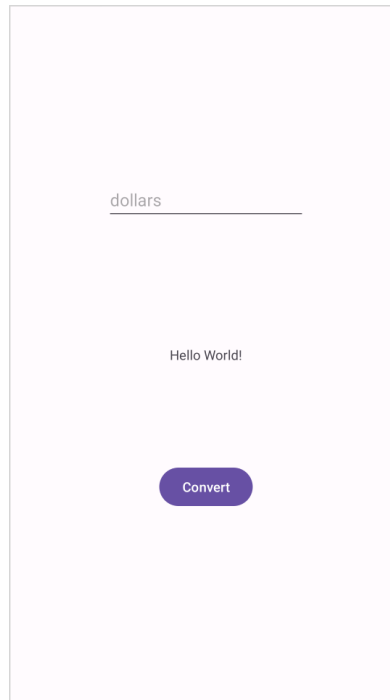


Figure 3-19

### 3.7 Reviewing the Layout and Resource Files

Before moving on to the next step, we 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 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 3-20 below:

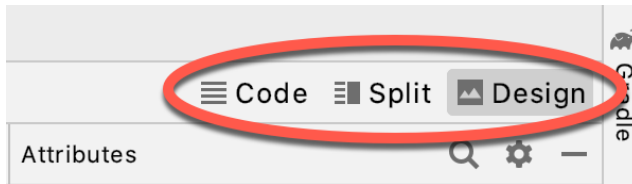


Figure 3-20

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 3-21:

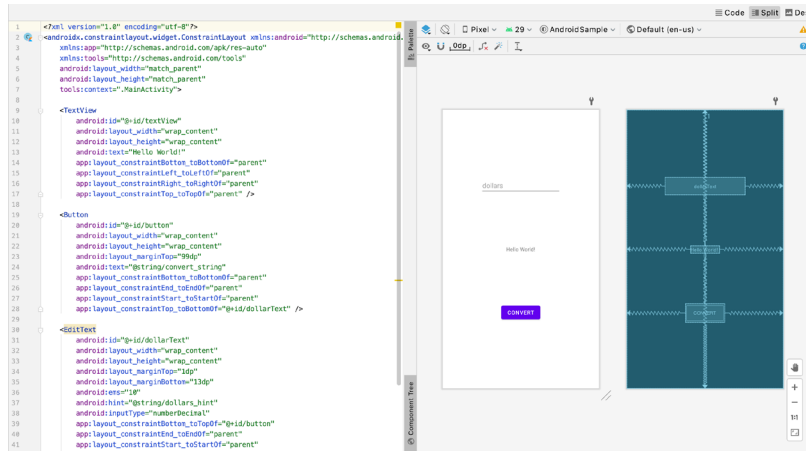


Figure 3-21

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. Clicking on the red square will display a color chooser allowing a different color to be selected:

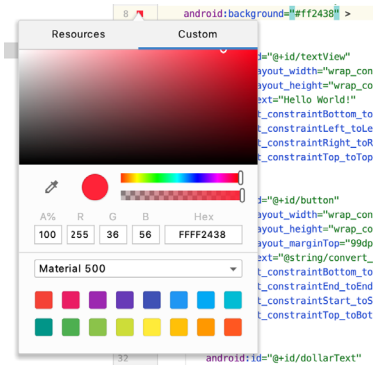


Figure 3-22

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

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

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

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 by clicking on the *Open editor* link in the top right-hand corner of the editor window. This will display the Translation Editor in the main panel of the Android Studio window:

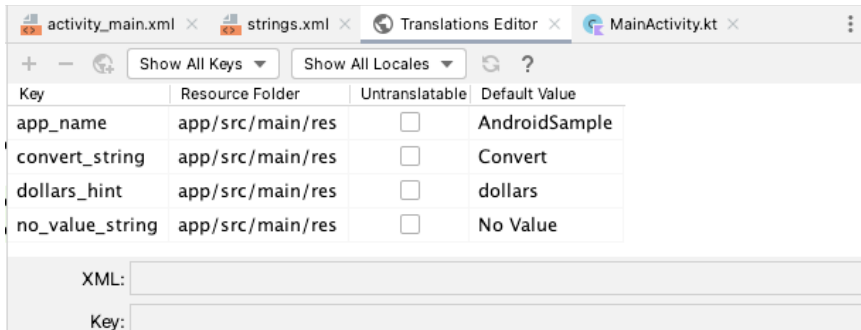


Figure 3-23

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

## 3.8 Adding Interaction

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

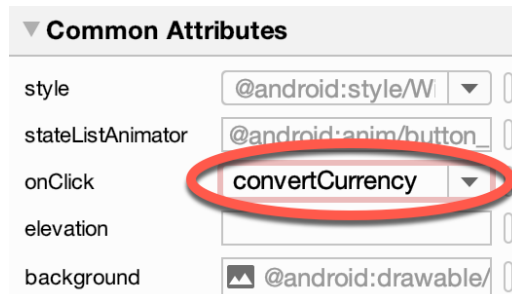


Figure 3-24

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

```
package com.ebookfrenzy.androidsample;

import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;
import android.view.View;
import android.widget.EditText;
import android.widget.TextView;
.
.
```

```
import java.util.Locale;

public class MainActivity extends AppCompatActivity {

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

    public void convertCurrency(View view) {

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

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

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

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

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

### 3.9 Summary

While not excessively complex, a number of steps are involved in setting up an Android development environment. Having performed those steps, it is worth working through an example to make sure the environment is correctly installed and configured. In this chapter, we have created an example application and then used the Android Studio Layout Editor tool to modify the user interface layout. In doing so, we explored the importance of using resources wherever possible, particularly 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

Creating an Example Android App in Android Studio  
detail in the next chapter.

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

Although the Android Studio Preview panel allows us to see the layout we are designing and test basic functionality using interactive mode, it be will necessary to compile and run an entire app to fully test that it works. 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. In this chapter, we will work through the creation of such a virtual device using the Pixel 4 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, several 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.

An AVD session can appear either as a separate window or embedded within the Android Studio window.

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. 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 -> Device Manager* menu option from within the main window.

Once launched, the manager will appear as a tool window as shown in Figure 4-1:

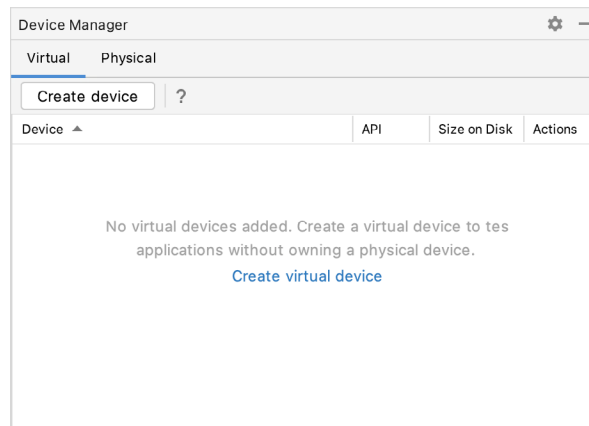


Figure 4-1

If you installed Android Studio for the first time on a computer (as opposed to upgrading an existing Android

## Creating an Android Virtual Device (AVD) in Android Studio

Studio installation), the installer might have created an initial AVD instance ready for use, as shown in Figure 4-2:

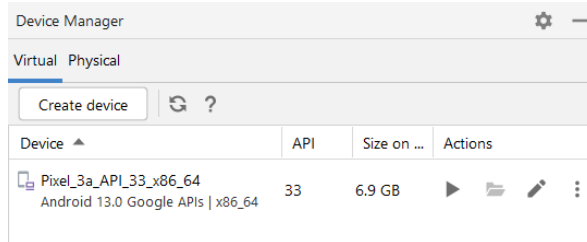


Figure 4-2

If this AVD is present on your system, you can use it to test apps. If no AVD was created, or you would like to create AVDs for different device types, follow the steps in the rest of this chapter.

To add a new AVD, begin by making sure that the *Virtual* tab is selected before clicking on the *Create device* button to open the *Virtual Device Configuration* dialog:

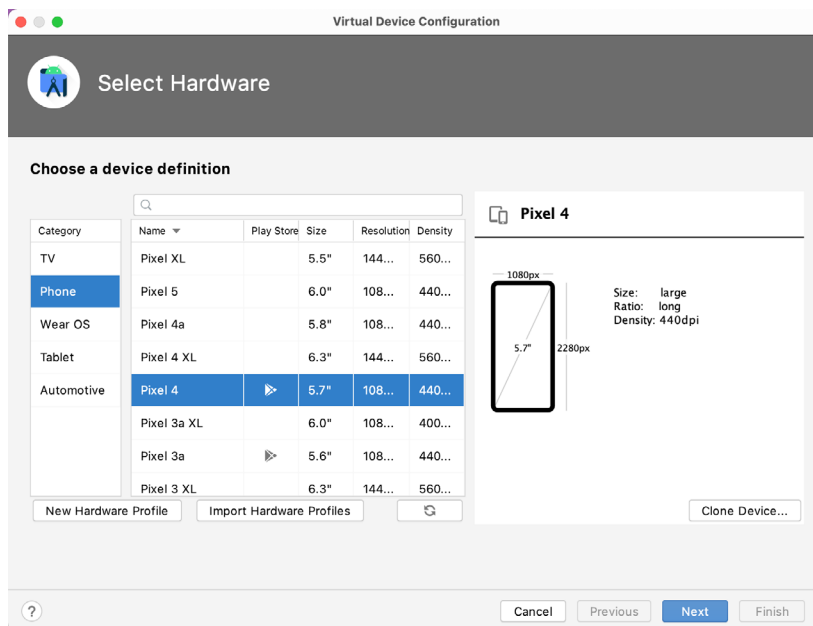


Figure 4-3

Within the dialog, perform the following steps to create a Pixel 4 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 4* device option and click *Next*.
3. On the *System Image* screen, select the latest version of Android. 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* (or *ARM images* if you are running a Mac with Apple Silicon) and *Other images* tabs to view alternative lists.



4. Click *Next* to proceed and enter a descriptive name (for example *Pixel 4 API 33*) into the name field or simply accept the default name.
5. Click *Finish* to create the AVD.
6. With the AVD created, the Device Manager may now be closed. If future modifications to the AVD are necessary, simply re-open the Device Manager, select the AVD from the list, and click on the pencil icon in the Actions column to edit the settings.

## 4.2 Starting the Emulator

To perform a test run of the newly created AVD emulator, simply select the emulator from the Device Manager and click on the launch button (the triangle in the Actions column). The emulator will appear embedded into the main Android Studio 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:

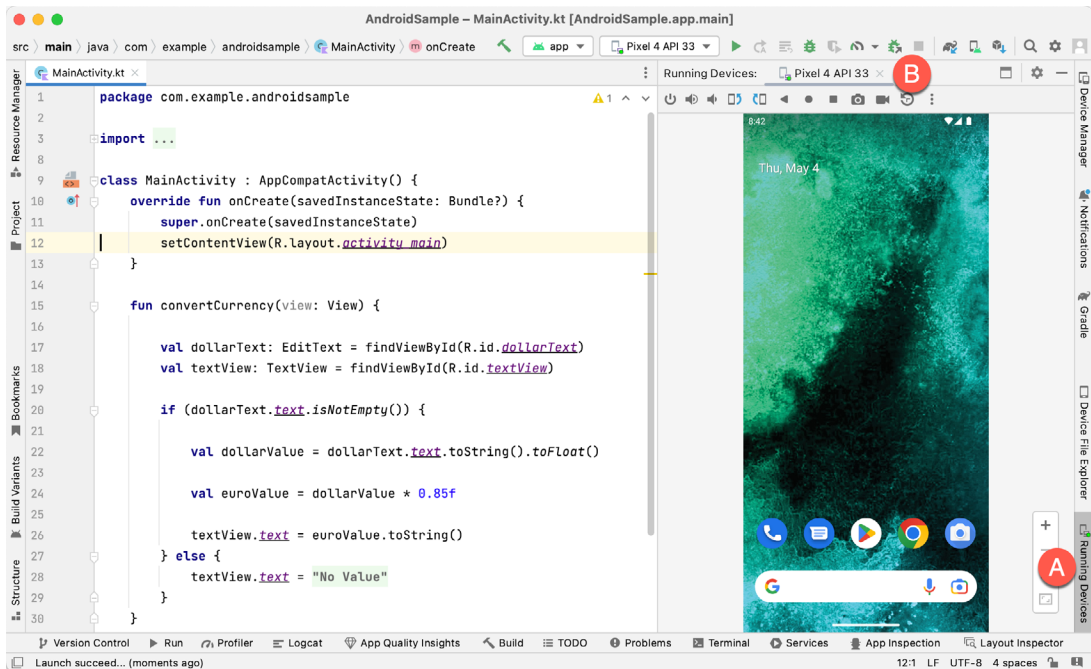


Figure 4-4

To hide and show the emulator tool window, click on the Running Devices tool window button (marked A above). Click on the “x” close button next to the tab (B) to exit the emulator. The emulator tool window can accommodate multiple emulator sessions, with each session represented by a tab. Figure 4-5, for example, shows a tool window with two emulator sessions:

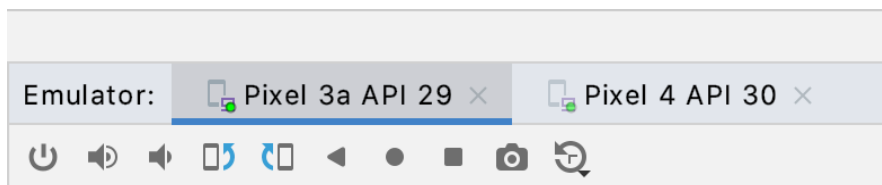


Figure 4-5

To switch between sessions, simply click on the corresponding tab.

## Creating an Android Virtual Device (AVD) in Android Studio

Although the emulator probably defaulted to appearing in portrait orientation, this and other default options can be changed. Within the Device Manager, select the new Pixel 4 entry and click on the pencil icon in the *Actions* column of the device row. In the configuration screen, locate the *Startup 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.

### 4.3 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 4 AVD is displayed in the device menu (marked A in Figure 4-6 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:

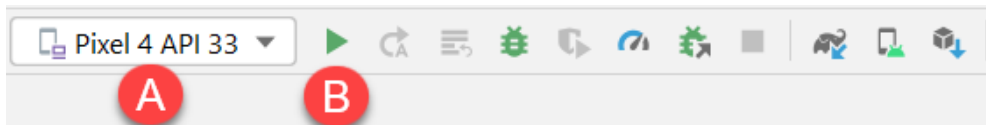


Figure 4-6

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 Device Manager as well as device connection configuration and troubleshooting options:

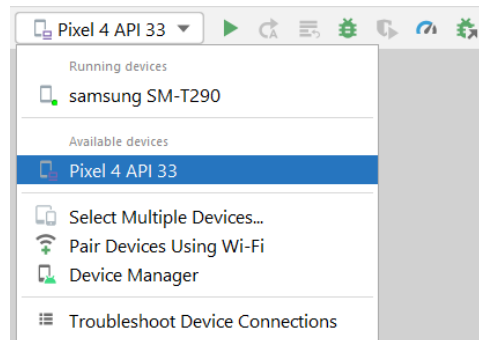


Figure 4-7

Once the application is installed and running, the user interface for the first fragment will appear within the emulator (a fragment is a reusable section of an Android project typically consisting of a user interface layout and some code, a topic which will be covered later in the chapter entitled “*An Introduction to Android Fragments*”):

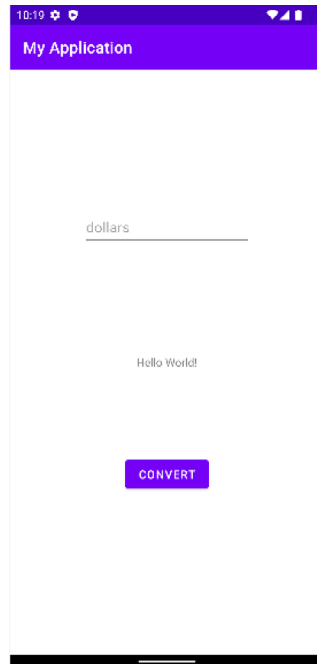


Figure 4-8

If 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 4-9 shows the Run tool window output from a typical successful application launch:

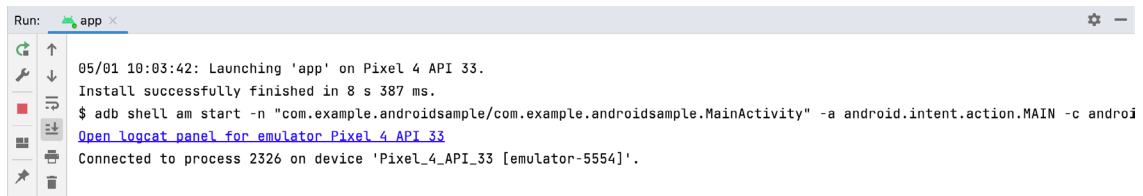


Figure 4-9

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. With the app now running, try performing a currency conversion to verify that the app works as intended.

## 4.4 Running on Multiple Devices

The run menu shown in Figure 4-7 above includes an option to run the app on multiple emulators and devices in parallel. When selected, this option displays the dialog shown in Figure 4-10 providing a list of both the AVDs configured on the system and any attached physical devices. Enable the checkboxes next to the emulators or devices to be targeted before clicking on the Run button:

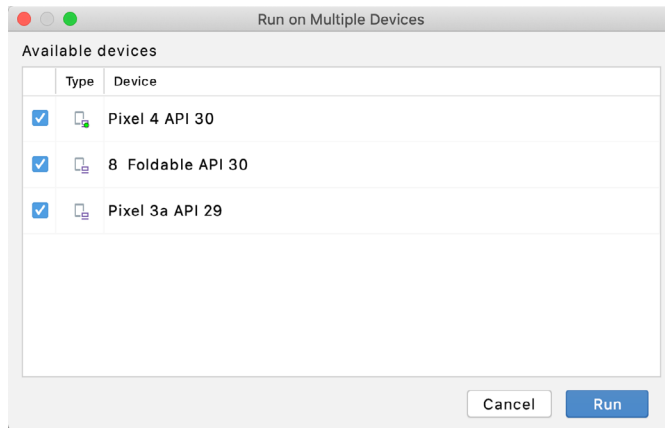


Figure 4-10

After the Run button is clicked, Android Studio will launch the app on the selected emulators and devices.

### 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 4-11:

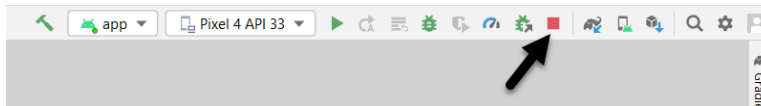


Figure 4-11

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 4-12 below:

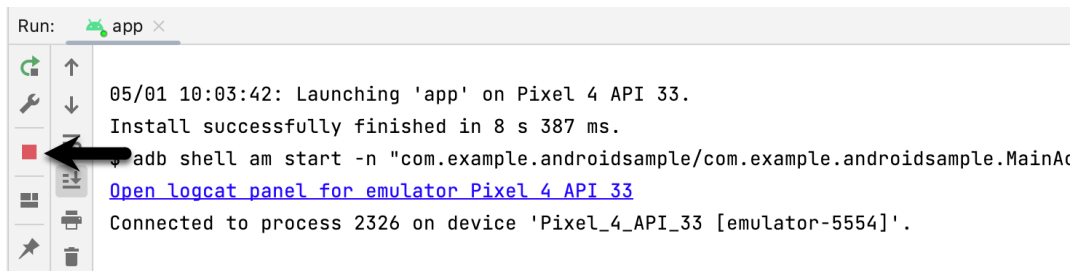


Figure 4-12

### 4.6 Supporting Dark Theme

Android 10 introduced the much-awaited dark theme, support for which is 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. Within the Settings app, choose the *Display* category and enable the *Dark theme* option as shown in Figure 4-13 so that the screen background turns black:

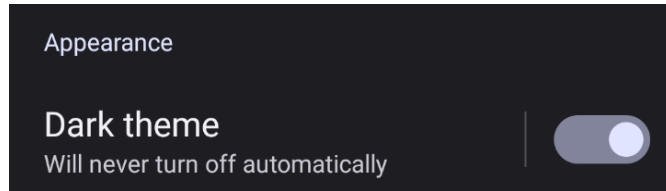


Figure 4-13

With dark theme enabled, run the AndroidSample app and note that it appears using a dark theme including a black background and a purple background color on the button as shown in Figure 4-14:

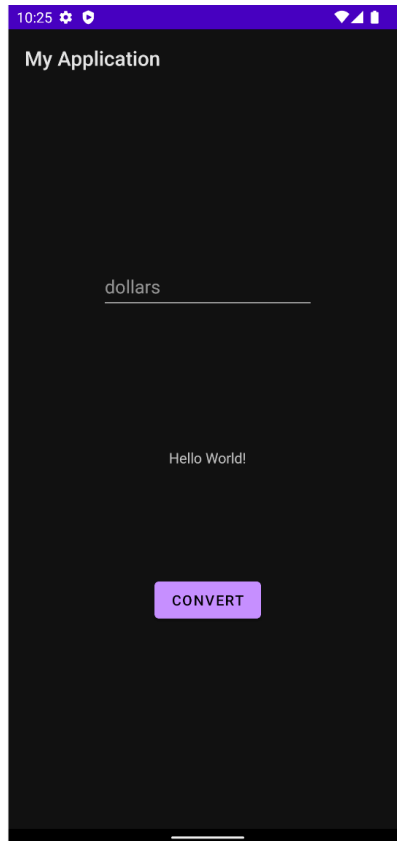


Figure 4-14

Return to the Settings app and turn off Dark theme mode before continuing.

## 4.7 Running the Emulator in a Separate Window

So far in this chapter, we have only used the emulator as a tool window embedded within the main Android Studio window. To run the emulator in a separate window, select the *File -> Settings...* menu option (*Android Studio -> Preferences...* on macOS), navigate to *Tools -> Emulator* in the left-hand navigation panel of the preferences dialog, and disable the *Launch in a tool window* option:

## Creating an Android Virtual Device (AVD) in Android Studio

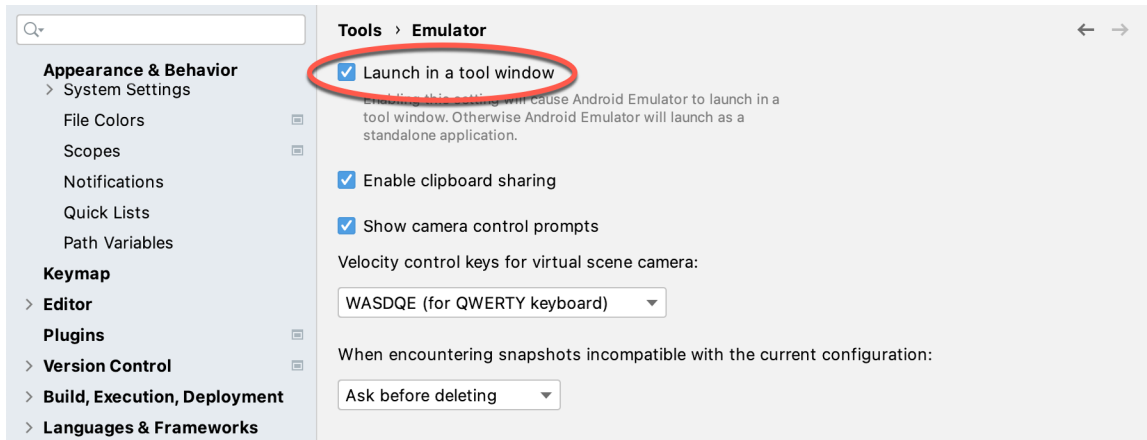


Figure 4-15

With the option disabled, click the Apply button followed by OK to commit the change, then exit the current emulator session by clicking on the close button on the tab marked B in Figure 4-4 above.

Run the sample app once again, at which point the emulator will appear as a separate window as shown below:

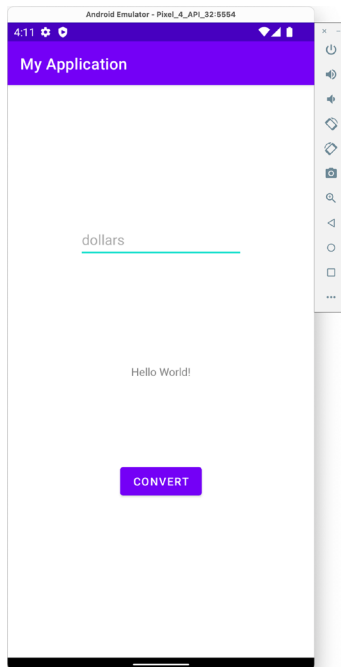


Figure 4-16

The choice of standalone or tool window mode is a matter of personal preference. If you prefer the emulator running in a tool window, return to the settings screen and re-enable the *Launch in a tool window* option. Before committing to standalone mode, however, keep in mind that the emulator tool window may also be detached from the main Android Studio window by clicking on the settings button (represented by the gear icon) in the tool emulator toolbar and selecting the *View Mode -> Float* menu option:

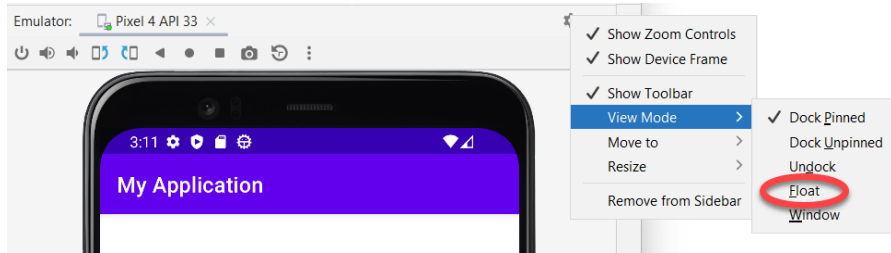


Figure 4-17

## 4.8 Enabling the Device Frame

The emulator can be configured to appear with (Figure 4-18) or without the device frame (Figure 4-16).

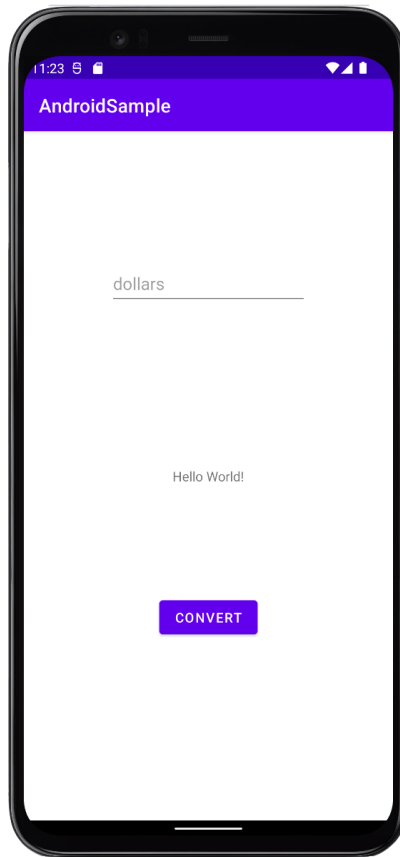


Figure 4-18

To change the setting, open the Device Manager, select the AVD from the list, and click on the pencil icon in the Actions column to edit the settings. In the settings screen, locate and change the Enable Device Frame option:

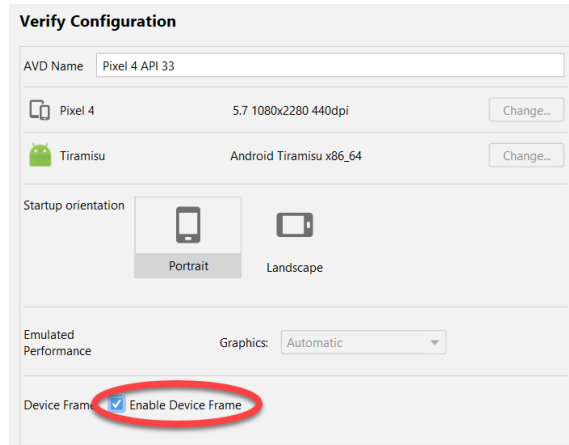


Figure 4-19

### 4.9 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 Studio Device Manager tool which may be used either as a command-line tool or via a graphical user interface. When creating an AVD to simulate a specific Android device model, the virtual device should be configured with a hardware specification matching that of the physical device.

The AVD emulator session may be displayed as a standalone window or embedded into the main Android Studio user interface.



## 5. Using and Configuring the Android Studio AVD Emulator

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 in both standalone and tool window modes.

### 5.1 The Emulator Environment

When launched in standalone mode, 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 Pixel 4 device):

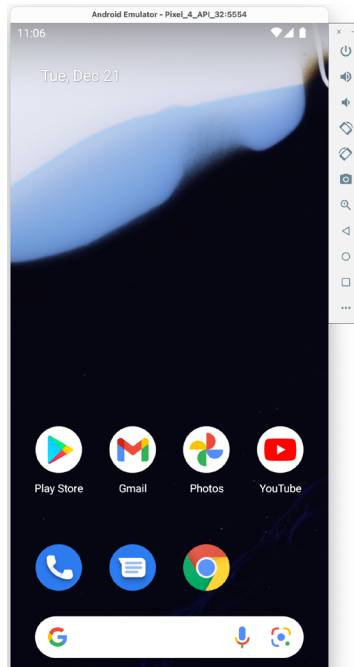


Figure 5-1

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

### 5.2 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.

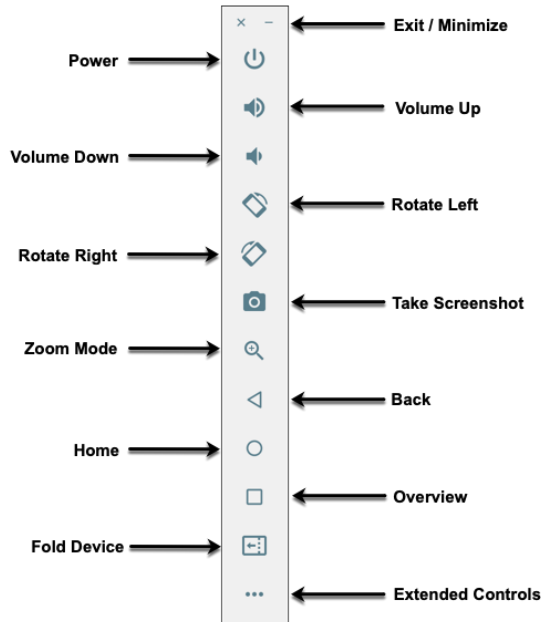


Figure 5-2

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** – Performs the standard Android “Back” navigation to return to a previous screen.
- **Home** – Displays the device home screen.
- **Overview** – Simulates selection of the standard Android “Overview” navigation 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:

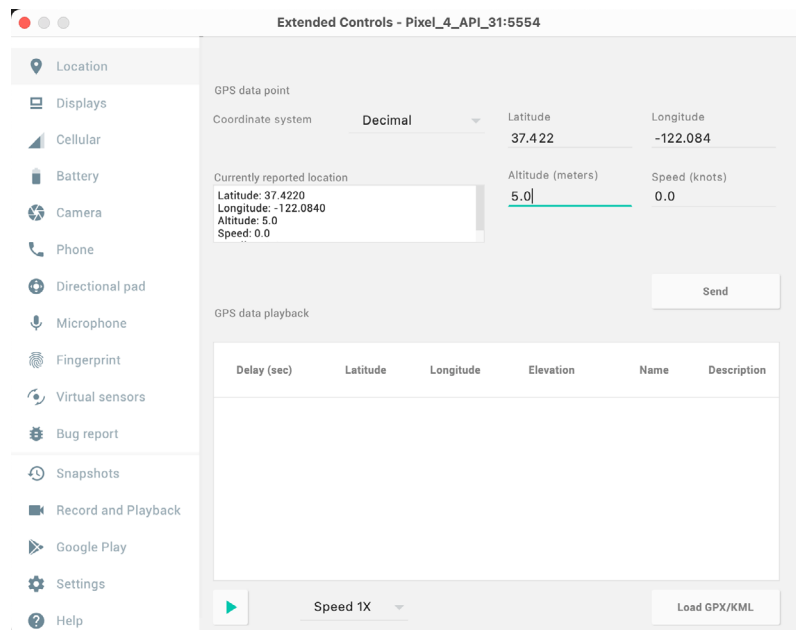


Figure 5-3

### 5.5.1 Location

The location controls allow simulated location information to be sent to the emulator in the form of decimal or sexagesimal 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 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.

### 5.5.5 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.

### 5.5.6 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 how 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 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 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:

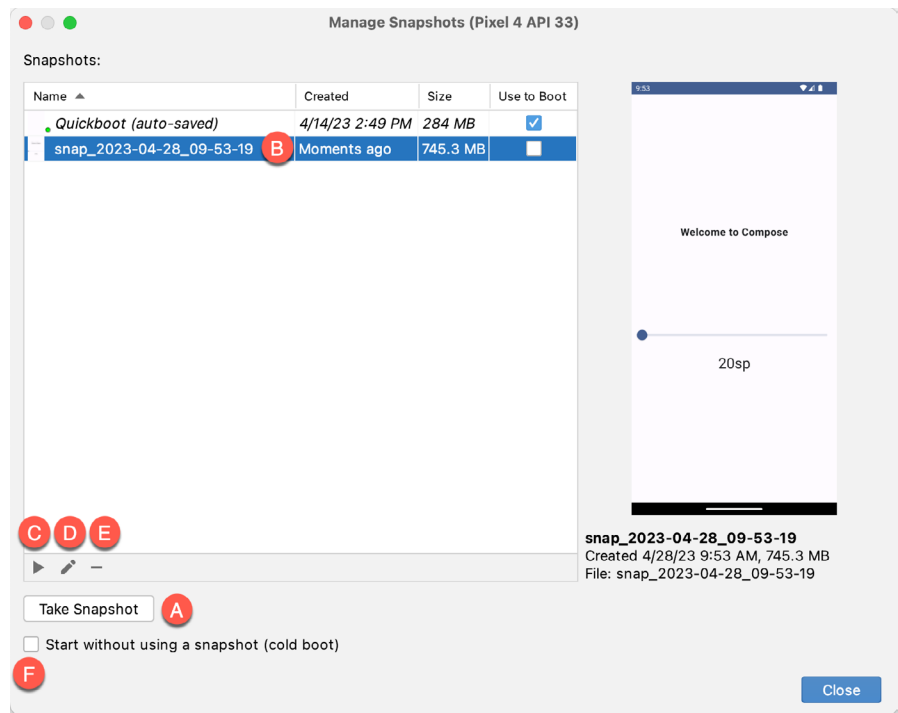


Figure 5-4

To force an emulator session to perform a cold boot instead of using a previous quick-boot snapshot, enable the checkbox marked F in the above figure.

You can also choose whether to start an emulator using either a cold boot, the most recent quick-boot snapshot, or a previous snapshot by making a selection from the run target menu in the main toolbar, as illustrated in Figure 5-5:

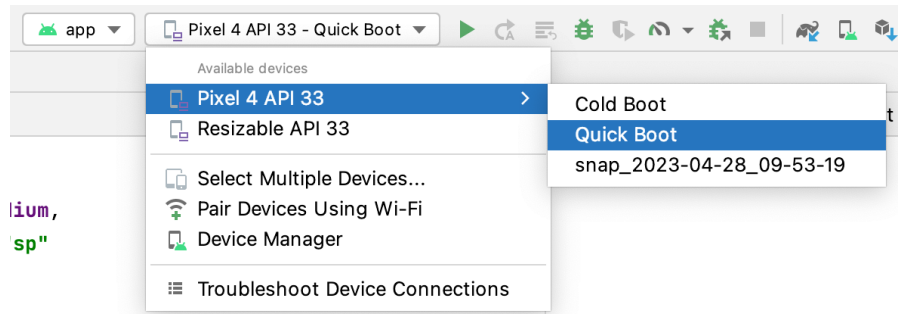


Figure 5-5

## 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) 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 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:

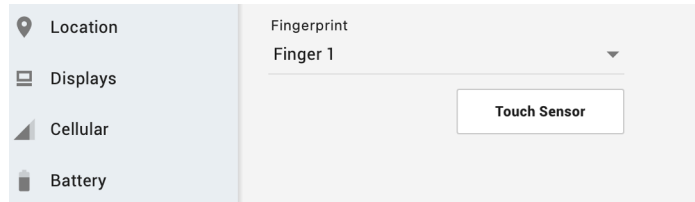


Figure 5-6

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

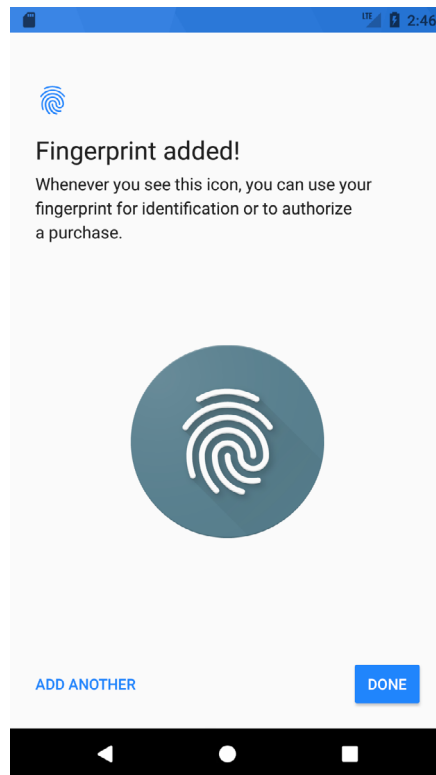


Figure 5-7

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 Sensor* button once again.

## 5.8 The Emulator in Tool Window Mode

As outlined in the previous chapter (*“Creating an Android Virtual Device (AVD) in Android Studio”*), Android Studio can be configured to launch the emulator as an embedded tool window so that it does not appear in a

## Using and Configuring the Android Studio AVD Emulator

separate window. When running in this mode, the same controls available in standalone mode are provided in the toolbar as shown in Figure 5-8:

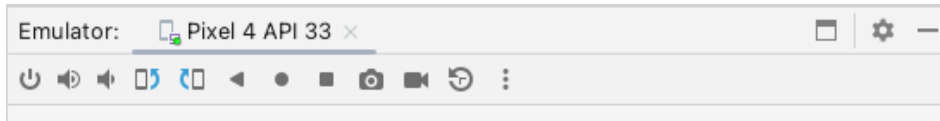


Figure 5-8

From left to right, these buttons perform the following tasks (details of which match those for standalone mode):

- Power
- Volume Up
- Volume Down
- Rotate Left
- Rotate Right
- Back
- Home
- Overview
- Screenshot
- Snapshots
- Extended Controls

## 5.9 Creating a Resizable Emulator

In addition to emulators configured to match specific Android device models, Android Studio also provides a resizable AVD that allows you to switch between phone, tablet and foldable device sizes. To create a resizable emulator, open the Device Manager and click the *Create device* button. Next, select the Resizable device definition illustrated in Figure 5-9, and follow the usual steps to create a new AVD:

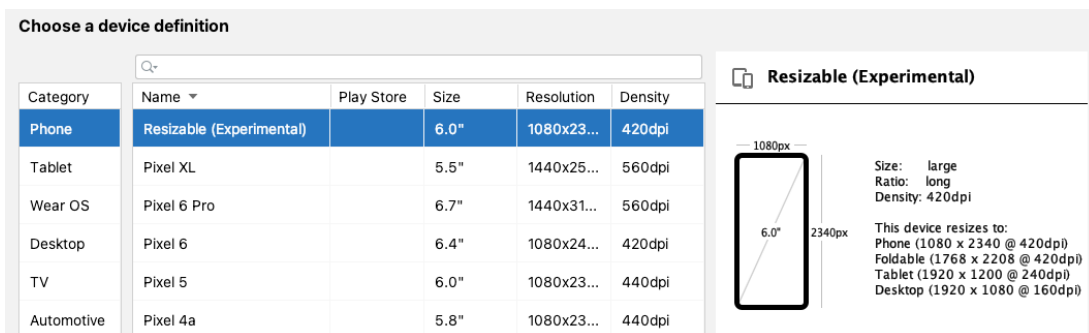


Figure 5-9

When you run an app on the new emulator within a tool window, the *Display mode* option will appear in the toolbar, allowing you to switch between emulator configurations as shown in Figure 5-10:



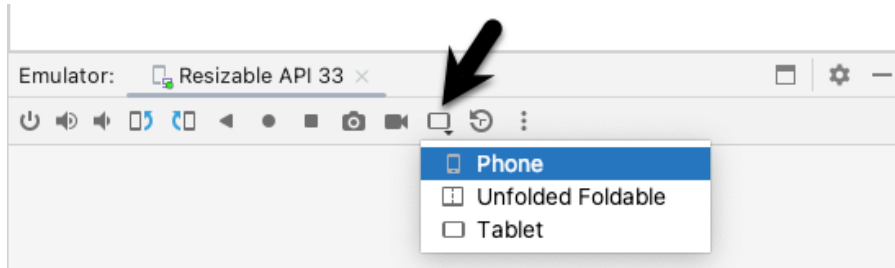


Figure 5-10

If the emulator is running in standalone mode, the Display mode option can be found in the side toolbar as shown below:

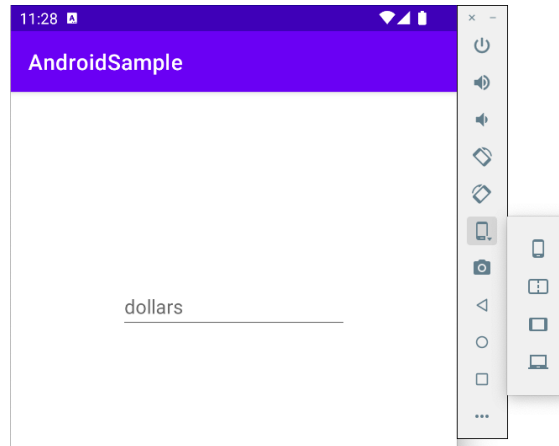


Figure 5-11

## 5.10 Summary

Android Studio contains an 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.



## 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 bypass the welcome screen next time it is launched, automatically opening the previously active project.

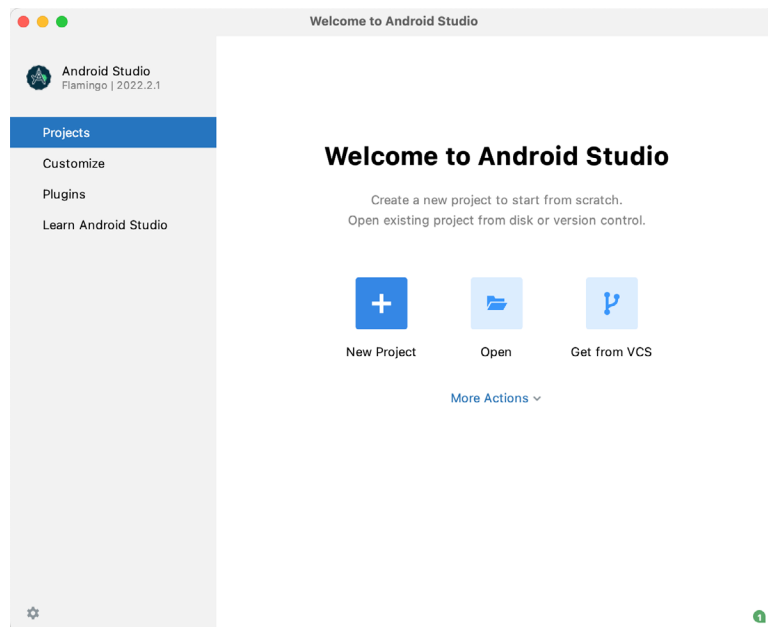


Figure 6-1

In addition to a list of recent projects, the welcome screen provides options for performing tasks such as opening and creating projects along with access to projects currently under version control. In addition, the *Customize* screen provides options to change the theme and font settings used by both the IDE and the editor. Android Studio plugins may be viewed, installed, and managed using the *Plugins* option.

Additional options are available by clicking on the menu button as shown in Figure 6-2:

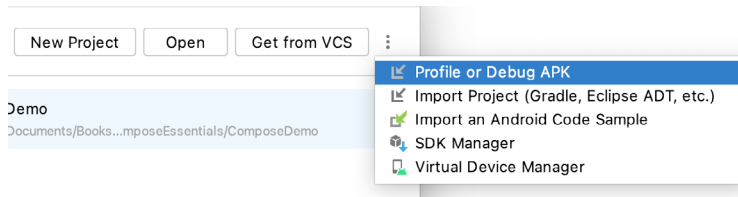


Figure 6-2

## 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 6-3.

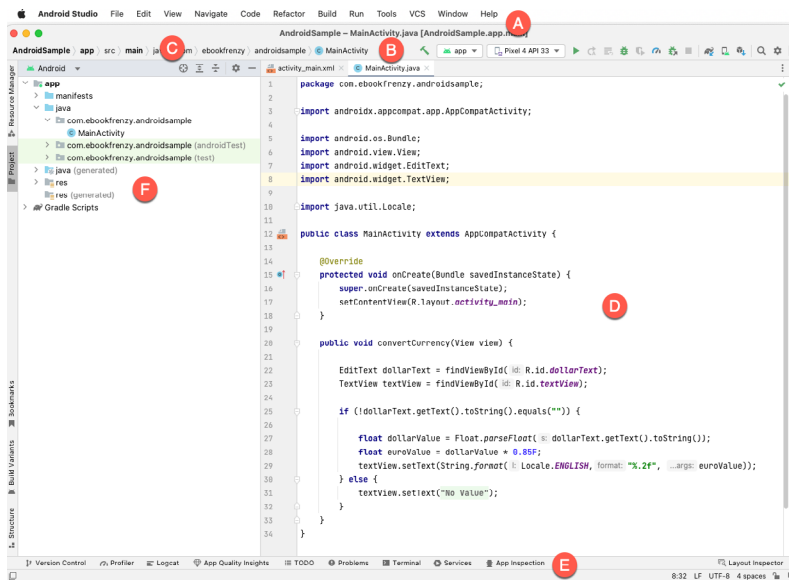


Figure 6-3

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 quick 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 sub-folders and files at that location ready for selection. Similarly, clicking on a class name displays a menu listing methods contained within that class. Select a method from the list to be taken to the corresponding location within the code editor. 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. When multiple files are open, each file is represented by a tab located along the top edge of the editor as shown in Figure 6-4:

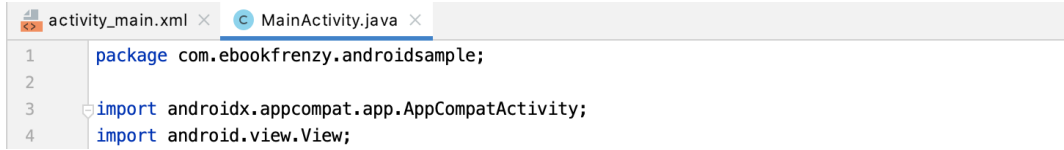


Figure 6-4

**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 display 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 several 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 many 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 many other windows which, when enabled, are displayed along the bottom and sides of the main window. The tool window quick access menu can be displayed by hovering the mouse pointer over the button located in the far left-hand corner of the status bar (Figure 6-5) without clicking the mouse button.

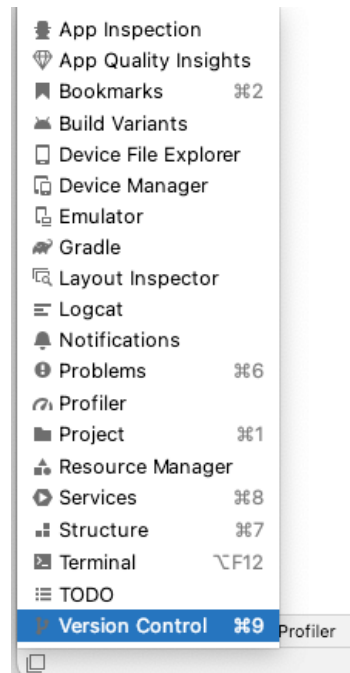


Figure 6-5

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

## A Tour of the Android Studio User Interface

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 6-6) 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.

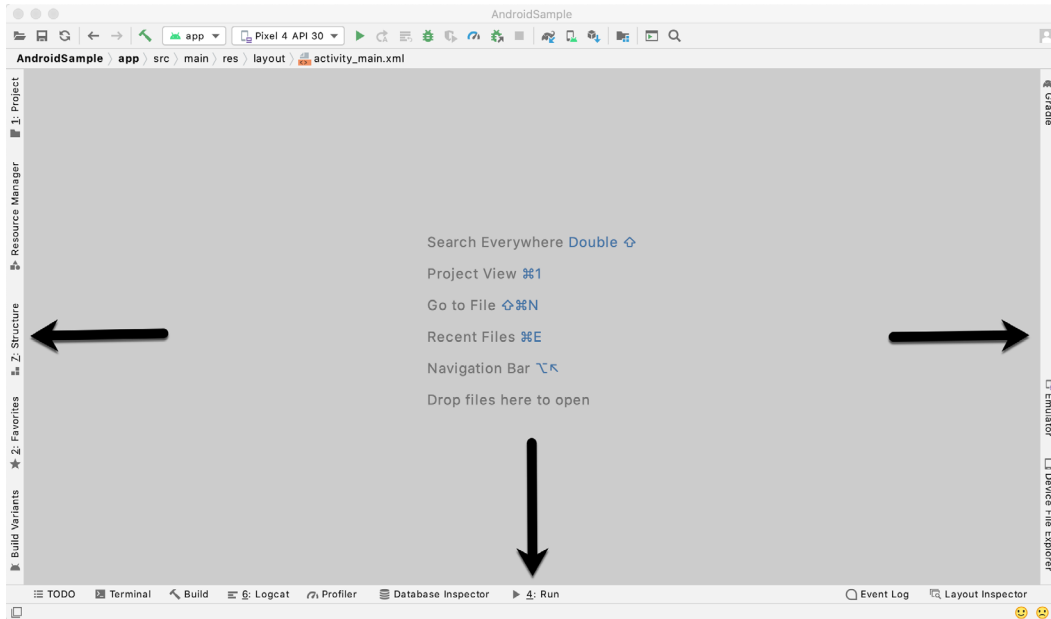


Figure 6-6

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 toolbars.

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 6-7 shows the settings menu for the Project 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.

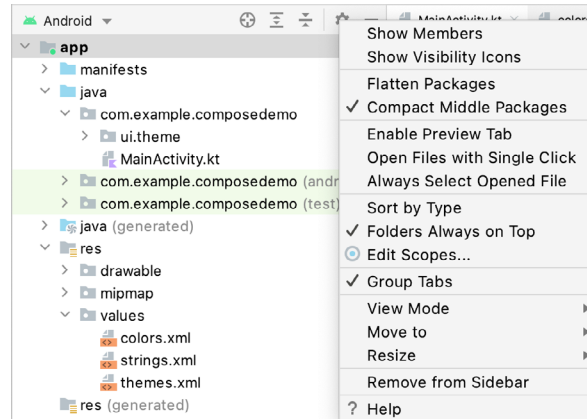


Figure 6-7

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 toolbar and items matching the search highlighted.

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

- **App Inspection** - Provides access to the Database and Background Task inspectors. The Database Inspector allows you to inspect, query, and modify your app's databases while the app is running. The Background Task Inspector allows background worker tasks created using WorkManager to be monitored and managed.
- **Build** - The build tool window displays information about the build process while a project is being compiled and packaged and displays details of any errors encountered.
- **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).
- **Device File Explorer** - Available via the *View -> Tool Windows -> Device File Explorer* menu, this 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.
- **Device Manager** - Provides access to the Device Manager tool window where physical Android device connections and emulators may be added, removed, and managed.
- **Emulator** - Contains the AVD emulator if the option has been enabled to run the emulator in a tool window as outlined in the chapter entitled *"Creating an Android Virtual Device (AVD) in Android Studio"*.
- **Bookmarks** - The Bookmarks tool window provides quick access to bookmarked files and code lines. For example, right-clicking on a file in the project view allows access to an Add to Bookmarks menu option. Similarly, you can bookmark a line of code in a source file by moving the cursor to that line and pressing the F11 key (F3 on macOS). All bookmarked items can be accessed through this tool window.
- **Gradle** - The Gradle tool window provides a view of 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.

- **Layout Inspector** - Provides a visual 3D rendering of the hierarchy of components that make up a user interface layout.
- **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.
- **Problems** - A central location in which to view all of the current errors or warnings within the project. Double-clicking on an item in the problem list will take you to the problem file and location.
- **Profiler** – The Android Profiler tool window provides real-time 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.
- **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.
- **Resource Manager** - A tool for adding and managing resources and assets such as images, colors, and layout files contained with the project.
- **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.
- **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.
- **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.
- **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*.
- **Version Control** - This tool window is used when the project files are under source code version control, allowing access to Git repositories and code change history.

## 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. You may also list and modify the keyboard shortcuts by selecting the *File -> Settings...* menu option (*Android Studio -> Preferences...* on macOS) and clicking on the Keymap entry as shown in Figure 6-8 below:



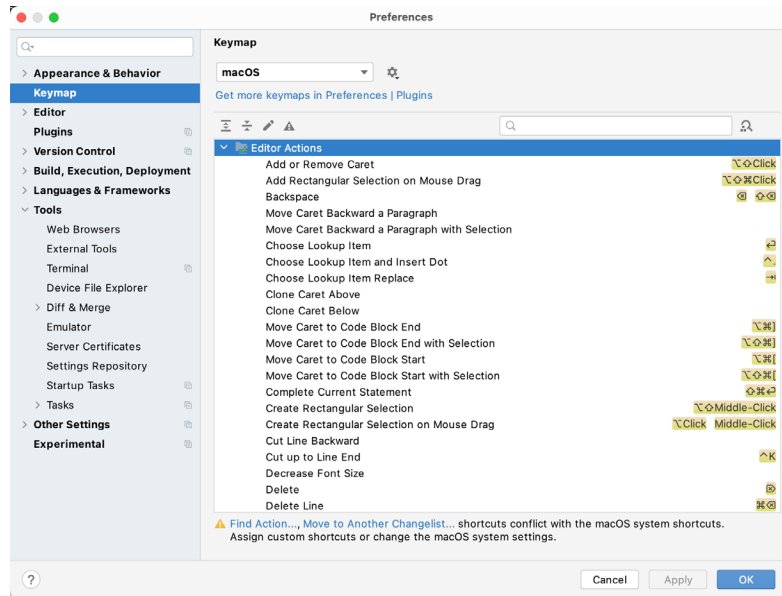


Figure 6-8

## 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 6-9).

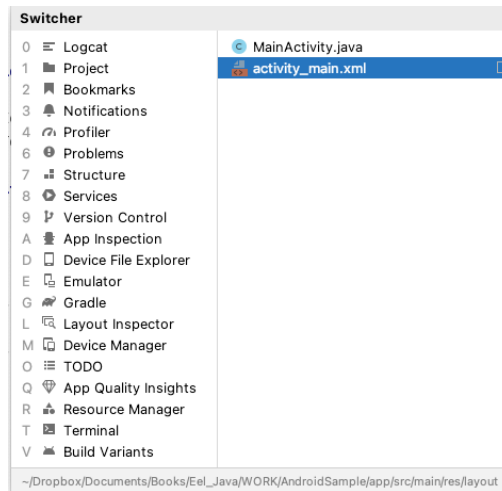


Figure 6-9

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 6-10). 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 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.

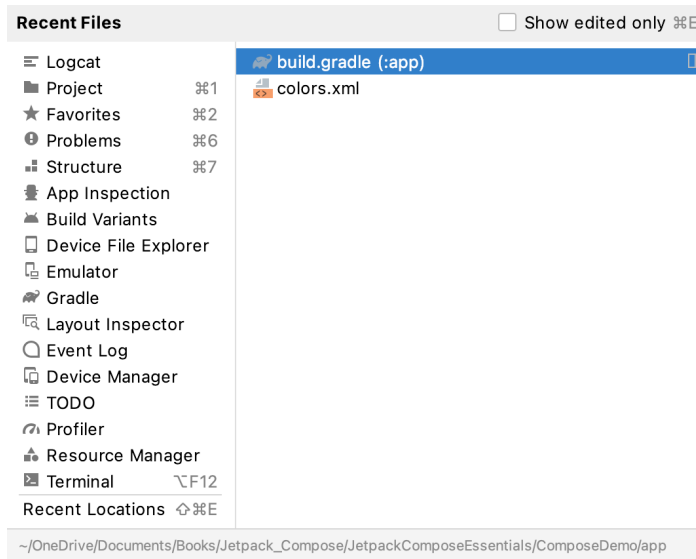


Figure 6-10

## 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 & Behavior* option followed by *Appearance* 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 6-11 shows an example of the main window with the Darcula theme selected:

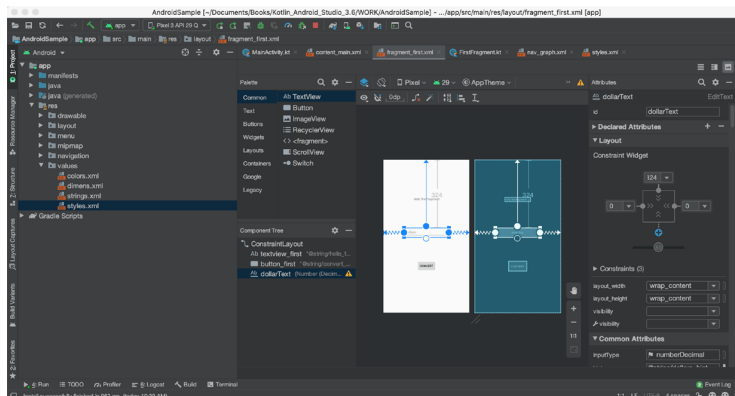


Figure 6-11

To synchronize the Android Studio theme with the operating system light and dark mode setting, enable the *Sync with OS* option and use the drop-down menu to control which theme to use for each mode:

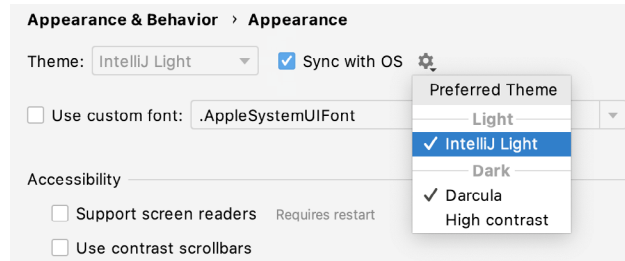


Figure 6-12

## 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 that 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.



## 26. An Overview and Example of Android Event Handling

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

The primary objective of this chapter, therefore, is to provide an overview of event handling in Android applications together with an Android Studio based example project.

### 26.1 Understanding Android Events

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

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

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

### 26.2 Using the android:onClick Resource

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

```
<Button
```

## An Overview and Example of Android Event Handling

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

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

### 26.3 Event Listeners and Callback Methods

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

- **onClickListener** – Used to detect click style events whereby the user touches and then releases an area of the device display occupied by a view. Corresponds to the `onClick()` callback method which is passed a reference to the view that received the event as an argument.
- **onLongClickListener** – Used to detect when the user maintains the touch over a view for an extended period. Corresponds to the `onLongClick()` callback method which is passed as an argument the view that received the event.
- **onTouchListener** – Used to detect any form of contact with the touch screen including individual or multiple touches and gesture motions. Corresponding with the `onTouch()` callback, this topic will be covered in greater detail in the chapter entitled “*Android Touch and Multi-touch Event Handling*”. The callback method is passed as arguments the view that received the event and a `MotionEvent` object.
- **onCreateContextMenuListener** – Listens for the creation of a context menu as the result of a long click. Corresponds to the `onCreateContextMenu()` callback method. The callback is passed the menu, the view that received the event and a menu context object.
- **onFocusChangeListener** – Detects when focus moves away from the current view as the result of interaction with a track-ball or navigation key. Corresponds to the `onFocusChange()` callback method which is passed the view that received the event and a Boolean value to indicate whether focus was gained or lost.
- **onKeyListener** – Used to detect when a key on a device is pressed while a view has focus. Corresponds to the `onKey()` callback method. Passed as arguments are the view that received the event, the `KeyCode` of the physical key that was pressed and a `KeyEvent` object.

### 26.4 An Event Handling Example

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

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

Enter *EventExample* into the Name field and specify *com.ebookfrenzy.eventexample* as the package name. Before

clicking on the Finish button, change the Minimum API level setting to API 26: Android 8.0 (Oreo) and the Language menu to Java. Using the steps outlined in section 11.8 *Migrating a Project to View Binding*, convert the project to use view binding.

## 26.5 Designing the User Interface

The user interface layout for the *MainActivity* class in this example is to consist of a *ConstraintLayout*, a *Button* and a *TextView* as illustrated in Figure 26-1.

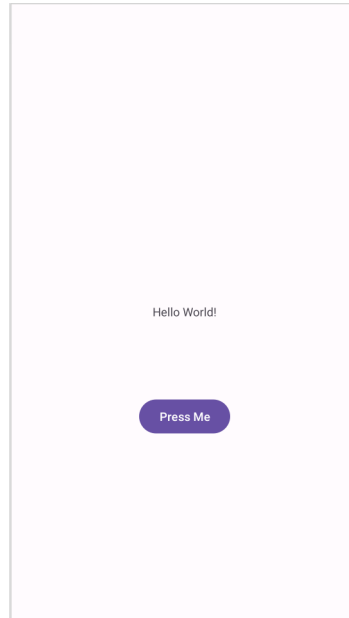


Figure 26-1

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

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

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

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

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

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

## 26.6 The Event Listener and Callback Method

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

## An Overview and Example of Android Event Handling

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

```
package com.ebookfrenzy.eventexample;

import androidx.appcompat.app.AppCompatActivity;

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

public class MainActivity extends AppCompatActivity {

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

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

                }
            }
        );
    }
}
```

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

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



```

binding.myButton.setOnClickListener(
    new Button.OnClickListener() {
        public void onClick(View v) {
            binding.statusText.setText("Button clicked");
        }
    }
);
}

```

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

## 26.7 Consuming Events

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

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

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

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

```

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

    binding.myButton.setOnLongClickListener(
        new Button.OnLongClickListener() {
            public boolean onLongClick(View v) {
                binding.statusText.setText("Long button click");
                return true;
            }
        }
    );
}
}

```

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

## An Overview and Example of Android Event Handling

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

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

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

## 26.8 Summary

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

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

Having covered the basics of event handling, the next chapter will explore in some depth the topic of touch events with a particular emphasis on handling multiple touches.

## 30. An Introduction to Android Fragments

As you progress through the chapters of this book it will become increasingly evident that many of the design concepts behind the Android system were conceived with the goal of promoting reuse of, and interaction between, the different elements that make up an application. One such area that will be explored in this chapter involves the use of Fragments.

This chapter will provide an overview of the basics of fragments in terms of what they are and how they can be created and used within applications. The next chapter will work through a tutorial designed to show fragments in action when developing applications in Android Studio, including the implementation of communication between fragments.

### 30.1 What is a Fragment?

A fragment is a self-contained, modular section of an application's user interface and corresponding behavior that can be embedded within an activity. Fragments can be assembled to create an activity during the application design phase, and added to or removed from an activity during application runtime to create a dynamically changing user interface.

Fragments may only be used as part of an activity and cannot be instantiated as standalone application elements. That being said, however, a fragment can be thought of as a functional "sub-activity" with its own lifecycle similar to that of a full activity.

Fragments are stored in the form of XML layout files and may be added to an activity either by placing appropriate <fragment> elements in the activity's layout file, or directly through code within the activity's class implementation.

### 30.2 Creating a Fragment

The two components that make up a fragment are an XML layout file and a corresponding Java class. The XML layout file for a fragment takes the same format as a layout for any other activity layout and can contain any combination and complexity of layout managers and views. The following XML layout, for example, is for a fragment consisting of a ConstraintLayout with a red background containing a single TextView with a white foreground:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.
android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/constraintLayout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@android:color/holo_red_dark"
    tools:context=".FragmentOne">
```

```
<TextView
    android:id="@+id/textView1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="My First Fragment"
    android:textAppearance="@style/TextAppearance.AppCompat.Large"
    android:textColor="@color/white"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

The corresponding class to go with the layout must be a subclass of the Android *Fragment* class. This class should, at a minimum, override the *onCreateView()* method which is responsible for loading the fragment layout. For example:

```
package com.example.myfragmentdemo;

import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import androidx.fragment.app.Fragment;

public class FragmentOne extends Fragment {

    @Override
    public View onCreateView(LayoutInflater inflater,
        ViewGroup container,
        Bundle savedInstanceState) {
        // Inflate the layout for this fragment
        binding = FragmentTextBinding.inflate(inflater, container, false);
        return binding.getRoot();
    }
}
```

In addition to the *onCreateView()* method, the class may also override the standard lifecycle methods.

Once the fragment layout and class have been created, the fragment is ready to be used within application activities.

### 30.3 Adding a Fragment to an Activity using the Layout XML File

Fragments may be incorporated into an activity either by writing Java code or by embedding the fragment into the activity's XML layout file. Regardless of the approach used, a key point to be aware of is that when the support library is being used for compatibility with older Android releases, any activities using fragments must be implemented as a subclass of *FragmentActivity* instead of the *AppCompatActivity* class:

```
package com.example.myfragmentdemo;
```

```
import android.os.Bundle;
import androidx.fragment.app.FragmentActivity;
import android.view.Menu;

public class MainActivity extends FragmentActivity {
    .
    .
}
```

Fragments are embedded into activity layout files using the `FragmentContainerView` class. The following example layout embeds the fragment created in the previous section of this chapter into an activity layout:

```
<?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">

    <androidx.fragment.app.FragmentContainerView
        android:id="@+id/fragment2"
        android:name="com.ebookfrenzy.myfragmentdemo.FragmentOne"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginStart="32dp"
        android:layout_marginEnd="32dp"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        tools:layout="@layout/fragment_one" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

The key properties within the `<fragment>` element are *android:name*, which must reference the class associated with the fragment, and *tools:layout*, which must reference the XML resource file containing the layout of the fragment.

Once added to the layout of an activity, fragments may be viewed and manipulated within the Android Studio Layout Editor tool. Figure 30-1, for example, shows the above layout with the embedded fragment within the Android Studio Layout Editor:



Figure 30-1

## 30.4 Adding and Managing Fragments in Code

The ease of adding a fragment to an activity via the activity's XML layout file comes at the cost of the activity not being able to remove the fragment at runtime. To achieve full dynamic control of fragments during runtime, those activities must be added via code. This has the advantage that the fragments can be added, removed and even made to replace one another dynamically while the application is running.

When using code to manage fragments, the fragment itself will still consist of an XML layout file and a corresponding class. The difference comes when working with the fragment within the hosting activity. There is a standard sequence of steps when adding a fragment to an activity using code:

1. Create an instance of the fragment's class.
2. Pass any additional intent arguments through to the class instance.
3. Obtain a reference to the fragment manager instance.
4. Call the *beginTransaction()* method on the fragment manager instance. This returns a fragment transaction instance.
5. Call the *add()* method of the fragment transaction instance, passing through as arguments the resource ID of the view that is to contain the fragment and the fragment class instance.
6. Call the *commit()* method of the fragment transaction.

The following code, for example, adds a fragment defined by the `FragmentOne` class so that it appears in the container view with an ID of `LinearLayout1`:

```
FragmentOne firstFragment = new FragmentOne();
firstFragment.setArguments(getIntent().getExtras());

FragmentManager fragmentManager = getSupportFragmentManager();
FragmentManager transaction = fragmentManager.beginTransaction();
```

```
transaction.add(R.id.LinearLayout1, firstFragment);
transaction.commit();
```

The above code breaks down each step into a separate statement for the purposes of clarity. The last four lines can, however, be abbreviated into a single line of code as follows:

```
getSupportFragmentManager().beginTransaction()
    .add(R.id.LinearLayout1, firstFragment).commit();
```

Once added to a container, a fragment may subsequently be removed via a call to the *remove()* method of the fragment transaction instance, passing through a reference to the fragment instance that is to be removed:

```
transaction.remove(firstFragment);
```

Similarly, one fragment may be replaced with another by a call to the *replace()* method of the fragment transaction instance. This takes as arguments the ID of the view containing the fragment and an instance of the new fragment. The replaced fragment may also be placed on what is referred to as the *back stack* so that it can be quickly restored if the user navigates back to it. This is achieved by making a call to the *addToBackStack()* method of the fragment transaction object before making the *commit()* method call:

```
FragmentTwo secondFragment = new FragmentTwo();
transaction.replace(R.id.LinearLayout1, secondFragment);
transaction.addToBackStack(null);
transaction.commit();
```

## 30.5 Handling Fragment Events

As previously discussed, a fragment is very much like a sub-activity with its own layout, class and lifecycle. The view components (such as buttons and text views) within a fragment are able to generate events just like those in a regular activity. This raises the question as to which class receives an event from a view in a fragment; the fragment itself, or the activity in which the fragment is embedded. The answer to this question depends on how the event handler is declared.

In the chapter entitled “*An Overview and Example of Android Event Handling*”, two approaches to event handling were discussed. The first method involved configuring an event listener and callback method within the code of the activity. For example:

```
binding.button.setOnClickListener(
    new Button.OnClickListener() {
        public void onClick(View v) {
            // Code to be performed when
            // the button is clicked
        }
    }
);
```

In the case of intercepting click events, the second approach involved setting the *android:onClick* property within the XML layout file:

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

The general rule for events generated by a view in a fragment is that if the event listener was declared in the fragment class using the event listener and callback method approach, then the event will be handled first by the fragment. If the *android:onClick* resource is used, however, the event will be passed directly to the activity containing the fragment.

### 30.6 Implementing Fragment Communication

Once one or more fragments are embedded within an activity, the chances are good that some form of communication will need to take place both between the fragments and the activity, and between one fragment and another. In fact, good practice dictates that fragments do not communicate directly with one another. All communication should take place via the encapsulating activity.

In order for an activity to communicate with a fragment, the activity must identify the fragment object via the ID assigned to it. Once this reference has been obtained, the activity can simply call the public methods of the fragment object.

Communicating in the other direction (from fragment to activity) is a little more complicated. In the first instance, the fragment must define a listener interface, which is then implemented within the activity class. For example, the following code declares an interface named *ToolbarListener* on a fragment class named *ToolbarFragment*. The code also declares a variable in which a reference to the activity will later be stored:

```
public class ToolbarFragment extends Fragment {

    ToolbarListener activityCallback;

    public interface ToolbarListener {
        public void onClick(int position, String text);
    }

    .
    .
}
```

The above code dictates that any class that implements the *ToolbarListener* interface must also implement a callback method named *onClick* which, in turn, accepts an integer and a String as arguments.

Next, the *onAttach()* method of the fragment class needs to be overridden and implemented. This method is called automatically by the Android system when the fragment has been initialized and associated with an activity. The method is passed a reference to the activity in which the fragment is contained. The method must store a local reference to this activity and verify that it implements the *ToolbarListener* interface:

```
@Override
public void onAttach(Context context) {
    super.onAttach(context);

    try {
        activityCallback = (ToolbarListener) activity;
    } catch (ClassCastException e) {
        throw new ClassCastException(activity.toString()
            + " must implement ToolbarListener");
    }
}
```



Upon execution of this example, a reference to the activity will be stored in the local *activityCallback* variable, and an exception will be thrown if that activity does not implement the *ToolbarListener* interface.

The next step is to call the callback method of the activity from within the fragment. When and how this happens is entirely dependent on the circumstances under which the activity needs to be contacted by the fragment. The following code, for example, calls the callback method on the activity when a button is clicked:

```
public void buttonClicked (View view) {
    activityCallback.onButtonClick(arg1, arg2);
}
```

All that remains is to modify the activity class so that it implements the *ToolbarListener* interface. For example:

```
public class MainActivity extends FragmentActivity
    implements ToolbarFragment.ToolbarListener {

    public void onButtonClick(String arg1, int arg2) {
        // Implement code for callback method
    }
    .
    .
}
```

As we can see from the above code, the activity declares that it implements the *ToolbarListener* interface of the *ToolbarFragment* class and then proceeds to implement the *onButtonClick()* method as required by the interface.

## 30.7 Summary

Fragments provide a powerful mechanism for creating re-usable modules of user interface layout and application behavior, which, once created, can be embedded in activities. A fragment consists of a user interface layout file and a class. Fragments may be utilized in an activity either by adding the fragment to the activity's layout file, or by writing code to manage the fragments at runtime. Fragments added to an activity in code can be removed and replaced dynamically at runtime. All communication between fragments should be performed via the activity within which the fragments are embedded.

Having covered the basics of fragments in this chapter, the next chapter will work through a tutorial designed to reinforce the techniques outlined in this chapter.



## 37. An Android ViewModel Saved State Tutorial

The preservation and restoration of app state is all about presenting the user with continuity in terms of appearance and behavior after an app is placed into the background. Users have come to expect to be able to switch from one app to another and, on returning to the original app, to find it in the exact state it was in before the switch took place.

As outlined in the chapter entitled “*Understanding Android Application and Activity Lifecycles*”, when the user places an app into the background that app becomes eligible for termination by the operating system if resources become constrained. When the user attempts to return the terminated app to the foreground, Android simply relaunches the app in a new process. Since this is all invisible to the user, it is the responsibility of the app to restore itself to the same state it was in when the app was originally placed in the background instead of presenting itself in its “initial launch” state. In the case of ViewModel-based apps, much of this behavior can be achieved using the *ViewModel Saved State module*.

### 37.1 Understanding ViewModel State Saving

As outlined in the previous chapters, the ViewModel brings many benefits to app development, including UI state restoration in the event of configuration changes such as a device rotation. To see this in action, run the *ViewModelDemo* app (or if you have not yet created the project, load into Android Studio the *ViewModelDemo\_LiveData* project from the sample code download that accompanies the book).

Once running, enter a dollar value and convert it to euros. With both the dollar and euro values displayed, rotate the device or emulator and note that, once the app has responded to the orientation change, both values are still visible.

Unfortunately, this behavior does not extend to the termination of a background app process. With the app still running, tap the device home button to place the *ViewModelDemo* app into the background, then terminate it by opening a terminal or command-prompt window and running the following command (where *<package name>* is the name you used when the project was created, for example, *com.ebookfrenzy.viewmodeldemo*):

```
adb shell am kill <package name>
```

If the *adb* command is not found, refer to the chapter titled “*Setting up an Android Studio Development Environment*” for steps on setting up your Android Studio environment.

Once the app has been terminated, return to the device or emulator and select the app from the launcher (do not simply re-run the app from within Android Studio). Once the app appears, it will do so as if it was just launched, with the previous dollar and euro values lost. From the perspective of the user, however, the app was simply restored from the background and should still have contained the original data. In this case, the app has failed to provide the continuity that users have come to expect from Android apps.

### 37.2 Implementing ViewModel State Saving

Basic ViewModel state saving is made possible through the introduction of the *ViewModel Saved State* library. This library essentially extends the *ViewModel* class to include support for maintaining state through the termination and subsequent relaunch of a background process.

## An Android ViewModel Saved State Tutorial

The key to saving state is the `SavedStateHandle` class which is used to save and restore the state of a view model instance. A `SavedStateHandle` object contains a key-value map that allows data values to be saved and restored by referencing corresponding keys.

To support saved state, a different kind of `ViewModel` subclass needs to be declared, in this case one containing a constructor which can receive a `SavedStateHandle` instance. Once declared, `ViewModel` instances of this type can be created by including a `SavedStateViewModelFactory` object at creation time. Consider the following code excerpt from a standard `ViewModel` declaration:

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

import androidx.lifecycle.ViewModel;
import androidx.lifecycle.MutableLiveData;

public class MainViewModel extends ViewModel {
    .
    .
}
```

The code to create an instance of this class would likely resemble the following:

```
private MainViewModel mViewModel;

mViewModel = new ViewModelProvider(this).get(MainViewModel.class);
```

A `ViewModel` subclass designed to support saved state, on the other hand, would need to be declared as follows:

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

import android.util.Log;

import androidx.lifecycle.ViewModel;
import androidx.lifecycle.MutableLiveData;
import androidx.lifecycle.SavedStateHandle;

public class MainViewModel extends ViewModel {

    private SavedStateHandle savedStateHandle;

    public MainViewModel(SavedStateHandle savedStateHandle) {
        this.savedStateHandle = savedStateHandle;
    }

    .
    .
}
```

When instances of the above `ViewModel` are created, the `ViewModelProvider` class initializer must be passed a `SavedStateViewModelFactory` instance as follows:

```
SavedStateViewModelFactory factory =
    new SavedStateViewModelFactory(getActivity().getApplication(), this);
```

```
mViewModel = new ViewModelProvider(this).get(MainViewModel.class);
```

### 37.3 Saving and Restoring State

An object or value can be saved from within the ViewModel by passing it through to the `set()` method of the `SavedStateHandle` instance, providing the key string by which it is to be referenced when performing a retrieval:

```
private static final String NAME_KEY = "Customer Name";
```

```
savedStateHandle.set(NAME_KEY, customerName);
```

When used with LiveData objects, a previously saved value may be restored using the `getLiveData()` method of the `SavedStateHandle` instance, once again referencing the corresponding key as follows:

```
MutableLiveData<String> restoredName = savedStateHandle.getLiveData(NAME_KEY);
```

To restore a normal (non-LiveData) object, simply use the `SavedStateHandle` `get()` method:

```
String restoredName = savedStateHandle.get(NAME_KEY);
```

Other useful `SavedStateHandle` methods include the following:

- **contains(String key)** - Returns a boolean value indicating whether the saved state contains a value for the specified key.
- **remove(String key)** - Removes the value and key from the saved state. Returns the value that was removed.
- **keys()** - Returns a String set of all the keys contained within the saved state.

### 37.4 Adding Saved State Support to the ViewModelDemo Project

With the basics of ViewModel Saved State covered, the ViewModelDemo app can be extended to include this support. Begin by loading the `ViewModelDemo_LiveData` project created in “*An Android Jetpack LiveData Tutorial*” into Android Studio (a copy of the project is also available in the sample code download), opening the *build.gradle* (*Module:app*) file and adding the Saved State library dependencies (checking, as always, if more recent library versions are available):

```
.
.
dependencies {
.
.
    implementation 'androidx.savedstate:savedstate:1.2.1'
    implementation 'androidx.lifecycle:lifecycle-viewmodel-savedstate:2.6.1'
.
.
}
```

Next, modify the *MainViewModel.java* file so that the constructor accepts and stores a `SavedStateHandle` instance. Also import `androidx.lifecycle.SavedStateHandle`, declare a key string constant and modify the *result* LiveData variable so that the value is now obtained from the saved state in the constructor:

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

```
import androidx.lifecycle.ViewModel;
import androidx.lifecycle.MutableLiveData;
import androidx.lifecycle.SavedStateHandle;
```

```
public class MainViewModel extends ViewModel {

    private static final String RESULT_KEY = "Euro Value";
    private static final Float rate = 0.74F;
    private String dollarText = "";
    final private SavedStateHandle savedStateHandle;
    final private MutableLiveData<Float> result = new MutableLiveData<>();

    public MainViewModel(SavedStateHandle savedStateHandle) {
        this.savedStateHandle = savedStateHandle;
        result = savedStateHandle.getLiveData(RESULT_KEY);
    }

    .
    .
}
```

Remaining within the *MainViewModel.java* file, modify the *setAmount()* method to include code to save the result value each time a new euro amount is calculated:

```
public void setAmount(String value) {
    this.dollarText = value;
    result.setValue(Float.valueOf(dollarText) * rate);
    Float convertedValue = Float.parseFloat(dollarText) * rate;
    result.setValue(convertedValue);
    savedStateHandle.set(RESULT_KEY, convertedValue);
}
```

With the changes to the ViewModel complete, open the *FirstFragment.java* file and make the following alterations to include a Saved State factory instance during the ViewModel creation process:

```
.
.
import androidx.lifecycle.SavedStateViewModelFactory;
.
.
@Override
public void onCreate(@Nullable Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    SavedStateViewModelFactory factory =
        new SavedStateViewModelFactory(
            getActivity().getApplication(), this);

    mViewModel = new ViewModelProvider(this, factory).get(MainViewModel.class);
    // TODO: Use the ViewModel
}
```

Note that the change to the ViewModelProvider call in the above code may cause Android Studio to generate

the following syntax error:

```
Cannot resolve constructor 'ViewModelProvider(FirstFragment,
SavedStateViewModelFactory)'
```

This syntax error is incorrect and can be ignored. The app will still compile and run successfully.

After completing the changes, build and run the app and perform a currency conversion. Note that the build may fail with the following error:

```
Duplicate class kotlin.collections.jdk8.CollectionsJDK8Kt found in modules
jetified-kotlin-stdlib-1.8.0
```

To correct this error, edit the *build.gradle (Module: app)* file, add the following dependency, and sync and rebuild the project:

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

With the screen UI populated with both the dollar and euro values, place the app into the background, terminate it using the *adb* tool and then relaunch it from the device or emulator screen. After restarting, the previous currency amounts should still be visible in the TextView and EditText components confirming that the state was successfully saved and restored.

## 37.5 Summary

A well designed app should always present the user with the same state when brought forward from the background, regardless of whether the process containing the app was terminated by the operating system in the interim. When working with ViewModels this can be achieved by taking advantage of the ViewModel Saved State module. This involves modifying the ViewModel constructor to accept a SavedStateHandle instance which, in turn, can be used to save and restore data values via a range of method calls. When the ViewModel instance is created, it must be passed a SavedStateViewModelFactory instance. Once these steps have been implemented, the app will automatically save and restore state during a background termination.





## 40. An Overview of the Navigation Architecture Component

Very few Android apps today consist of just a single screen. In reality, most apps comprise multiple screens through which the user navigates using screen gestures, button clicks and menu selections. Before the introduction of Android Jetpack, the implementation of navigation within an app was largely a manual coding process with no easy way to view and organize potentially complex navigation paths. This situation has improved considerably, however, with the introduction of the Android Navigation Architecture Component combined with support for navigation graphs in Android Studio.

### 40.1 Understanding Navigation

Every app has a home screen that appears after the app has launched and after any splash screen has appeared (a splash screen being the app branding screen that appears temporarily while the app loads). From this home screen, the user will typically perform tasks that will result in other screens appearing. These screens will usually take the form of other activities and fragments within the app. A messaging app, for example, might have a home screen listing current messages from which the user can navigate to either another screen to access a contact list or to a settings screen. The contacts list screen, in turn, might allow the user to navigate to other screens where new users can be added or existing contacts updated. Graphically, the app's *navigation graph* might be represented as shown in Figure 40-1:

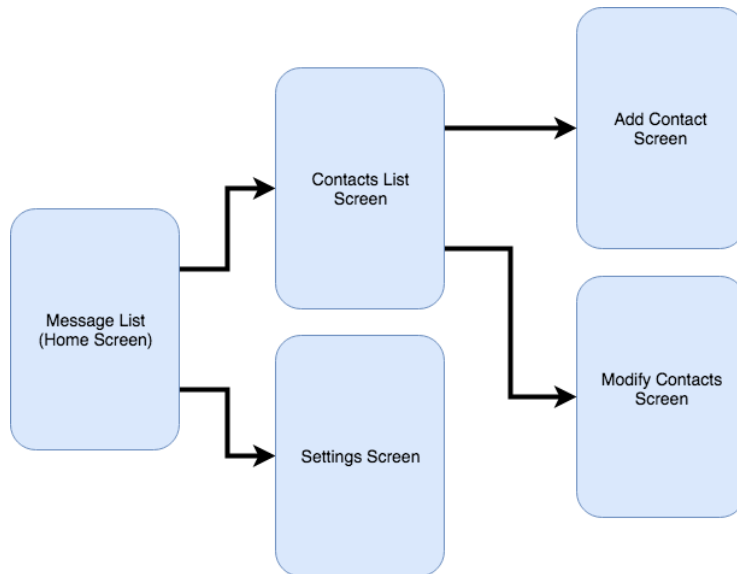


Figure 40-1

Each screen that makes up an app, including the home screen, is referred to as a *destination* and is usually a fragment or activity. The Android navigation architecture uses a *navigation stack* to track the user's path through the destinations within the app. When the app first launches, the home screen is the first destination placed onto the stack and becomes the *current destination*. When the user navigates to another destination, that screen

becomes the current destination and is *pushed* onto the stack above the home destination. As the user navigates to other screens, they are also pushed onto the stack. Figure 40-2, for example, shows the current state of the navigation stack for the hypothetical messaging app after the user has launched the app and is navigating to the “Add Contact” screen:

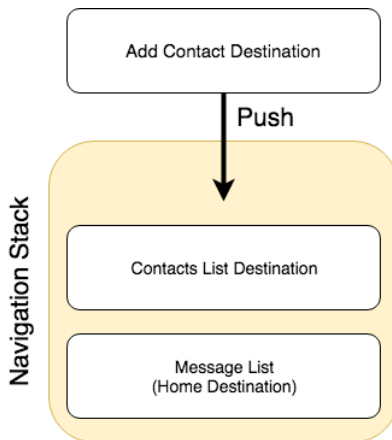


Figure 40-2

As the user navigates back through the screens using the system back button, each destination is *popped* off the stack until the home screen is once again the only destination on the stack. In Figure 40-3, the user has navigated back from the Add Contact screen, popping it off the stack and making the Contacts List screen the current destination:

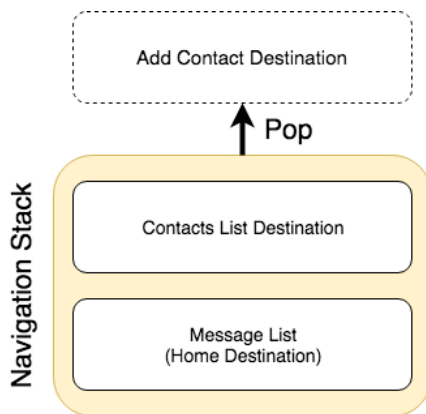


Figure 40-3

All of the work involved in navigating between destinations and managing the navigation stack is handled by a *navigation controller* which is represented by the `NavController` class.

Adding navigation to an Android project using the Navigation Architecture Component is a straightforward process involving a navigation host, navigation graph, navigation actions and a minimal amount of code writing to obtain a reference to, and interact with, the navigation controller instance.

### 40.2 Declaring a Navigation Host

A navigation host is simply a special fragment (`NavHostFragment`) that is embedded into the user interface layout of an activity and serves as a placeholder for the destinations through which the user will navigate. Figure 40-4, for example, shows a typical activity screen and highlights the area represented by the navigation host

fragment:



Figure 40-4

A `NavHostFragment` can be placed into an activity layout within the Android Studio layout editor either by dragging and dropping an instance from the Containers section of the palette, or by manually editing the XML as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/container"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity" >

    <androidx.fragment.app.FragmentContainerView
        android:id="@+id/demo_nav_host_fragment"
        android:name="androidx.navigation.fragment.NavHostFragment"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        app:defaultNavHost="true"
        app:navGraph="@navigation/navigation_graph" />

</FrameLayout>
```

The points of note in the above navigation host fragment element are the reference to the `NavHostFragment` in the *name* property, the setting of *defaultNavHost* to true and the assignment of the file containing the navigation graph to the *navGraph* property.

When the activity launches, this navigation host fragment is replaced by the home destination designated in the navigation graph. As the user navigates through the app screens, the host fragment will be replaced by the appropriate fragment for the destination.

## 40.3 The Navigation Graph

A navigation graph is an XML file which contains the destinations that will be included in the app navigation. In addition to these destinations, the file also contains navigation actions that define navigation between destinations, and optional arguments for passing data from one destination to another. Android Studio includes a navigation graph editor that can be used to design graphs and implement actions either visually or by manually editing the XML.

Figure 40-5, shows the Android Studio navigation graph editor in Design mode:

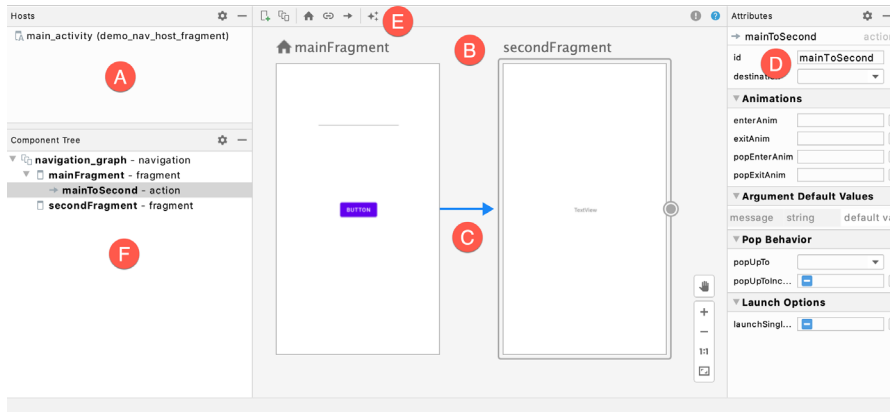


Figure 40-5

The destinations list (A) provides a list of all of the destinations currently contained within the graph. Selecting a destination from the list will locate and select the corresponding destination in the graph (particularly useful for locating specific destinations in a large graph). The navigation graph panel (B) contains a dialog for each destination showing a representation of the user interface layout. In this example, this graph contains two destinations named `mainFragment` and `secondFragment`. Arrows between destinations (C) represent navigation action connections. Actions are added by hovering the mouse pointer over the edge of the origin until a circle appears, then clicking and dragging from the circle to the destination. The Attributes panel (D) allows the properties of the currently selected destination or action connection to be viewed and modified. In the above figure, the attributes for the action are displayed. New destinations are added by clicking on the button marked E and selecting options from a menu. Options are available to add existing fragments or activities as destinations, or to create new blank fragment destinations. The Component Tree panel (F) provides a hierarchical overview of the navigation graph.

The underlying XML for the navigation graph can be viewed and modified by switching the editor into Code mode. The following XML listing represents the navigation graph for the destinations and action connection shown in Figure 40-5 above:

```
<?xml version="1.0" encoding="utf-8"?>
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/navigation_graph"
    app:startDestination="@id/mainFragment">

    <fragment
        android:id="@+id/mainFragment"
        android:name="com.ebookfrenzy.navigationdemo.ui.main.MainFragment"
```

```

        android:label="fragment_main"
        tools:layout="@layout/fragment_main" >
        <action
            android:id="@+id/mainToSecond"
            app:destination="@id/secondFragment" />
    </fragment>
    <fragment
        android:id="@+id/secondFragment"
        android:name="com.ebookfrenzy.navigationdemo.SecondFragment"
        android:label="fragment_second"
        tools:layout="@layout/fragment_second" >
    </fragment>
</navigation>

```

If necessary, navigation graphs can also be split over multiple files to improve organization and promote reuse. When structured in this way, *nested graphs* are embedded into *root graphs*. To create a nested graph, simply shift-click on the destinations to be nested, right-click over the first destination and select the *Move to Nested Graph -> New Graph* menu option. The nested graph will then appear as a new node in the graph. To access the nested graph, simply double-click on the nested graph node to load the graph file into the editor.

## 40.4 Accessing the Navigation Controller

Navigating from one destination to another will usually take place in response to an event of some kind within an app such as a button click or menu selection. Before a navigation action can be triggered, the code must first obtain a reference to the navigation controller instance. This requires a call to the *findNavController()* method of the Navigation or NavHostFragment classes. The following code, for example, can be used to access the navigation controller of an activity. Note that for the code to work, the activity must contain a navigation host fragment:

```

NavController controller =
    Navigation.findNavController(activity, R.id.demo_nav_host_fragment);

```

In this case, the method call is passed a reference to the activity and the id of the NavHostFragment embedded in the activity's layout.

Alternatively, the navigation controller associated with any view may be identified simply by passing that view to the method:

```

NavController controller = Navigation.findNavController(binding.button);

```

The final option finds the navigation controller for a fragment by calling the *findNavController()* method of the NavHostFragment class, passing through a reference to the fragment:

```

NavController controller = NavHostFragment.findNavController(fragment);

```

## 40.5 Triggering a Navigation Action

Once the navigation controller has been found, a navigation action is triggered by calling the controller's *navigate()* method and passing through the resource id of the action to be performed. For example:

```

controller.navigate(R.id.goToContactsList);

```

The id of the action is defined within the Attributes panel of the navigation graph editor when an action connection is selected.

## 40.6 Passing Arguments

Data may be passed from one destination to another during a navigation action by making use of arguments which are declared within the navigation graph file. An argument consists of a name, type and an optional default value and may be added manually within the XML or using the Attributes panel when an action arrow or destination is selected within the graph. In Figure 40-6, for example, an integer argument named *contactsCount* has been declared with a default value of 0:

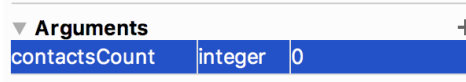


Figure 40-6

Once added, arguments are placed within the XML element of the receiving destination, for example:

```
<fragment
    android:id="@+id/secondFragment"
    android:name="com.ebookfrenzy.navigationdemo.SecondFragment"
    android:label="fragment_second"
    tools:layout="@layout/fragment_second" >
    <argument
        android:name="contactsCount"
        android:defaultValue=0
        app:type="integer" />
</fragment>
```

The Navigation Architecture Component provides two techniques for passing data between destinations. One approach involves placing the data into a Bundle object that is passed to the destination during an action where it is then unboxed and the arguments extracted.

The main drawback to this particular approach is that it is not “type safe”. In other words, if the receiving destination treats an argument as being a different type than it was declared (for example treating a string as an integer) this error will not be caught by the compiler and will likely cause problems at runtime.

A better option, and the one used in this book is to make use of *safeargs*. Safeargs is a plugin for the Android Studio Gradle build system which automatically generates special classes that allow arguments to be passed in a type safe way. The safeargs approach to argument passing will be described and demonstrated in the next chapter (“*An Android Jetpack Navigation Component Tutorial*”).

## 40.7 Summary

The term Navigation within the context of an Android app user interface refers to the ability of a user to move back and forth between different screens. Once time consuming to implement and difficult to organize, Android Studio and the Navigation Architecture Component now make it easier to implement and manage navigation within Android app projects.

The different screens within an app are referred to as destinations and are usually represented by fragments or activities. All apps have a home destination which includes the screen displayed when the app first loads. The content area of this layout is replaced by a navigation host fragment which is swapped out for other destination fragments as the user navigates the app. The navigation path is defined by the navigation graph file consisting of destinations and the actions that connect them together with any arguments to be passed between destinations. Navigation is handled by navigation controllers which, in addition to managing the navigation stack, provide methods to initiate navigation actions from within app code.

## 60. Making Runtime Permission Requests in Android

In a number of the example projects created in preceding chapters, changes have been made to the *AndroidManifest.xml* file to request permission for the app to perform a specific task. In a couple of instances, for example, internet access permission has been requested to allow the app to download and display web pages. In each case up until this point, the addition of the request to the manifest was all that was required for the app to obtain permission from the user to perform the designated task.

There are, however, a number of permissions for which additional steps are required in order for the app to function when running on Android 6.0 or later. The first of these so-called “dangerous” permissions will be encountered in the next chapter. Before reaching that point, however, this chapter will outline the steps involved in requesting such permissions when running on the latest generations of Android.

### 60.1 Understanding Normal and Dangerous Permissions

Android enforces security by requiring the user to grant permission for an app to perform certain tasks. Before the introduction of Android 6, permission was always sought at the point that the app was installed on the device. Figure 60-1, for example, shows a typical screen seeking a variety of permissions during the installation of an app via Google Play.

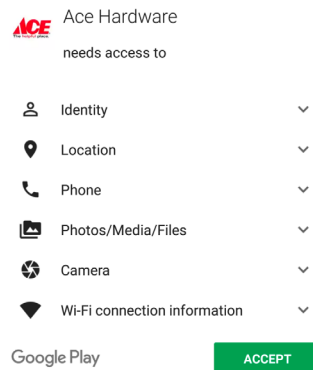


Figure 60-1

For many types of permissions this scenario still applies for apps on Android 6.0 or later. These permissions are referred to as *normal permissions* and are still required to be accepted by the user at the point of installation. A second type of permission, referred to as *dangerous permissions* must also be declared within the manifest file in the same way as a normal permission, but must also be requested from the user when the application is first launched. When such a request is made, it appears in the form of a dialog box as illustrated in Figure 60-2:

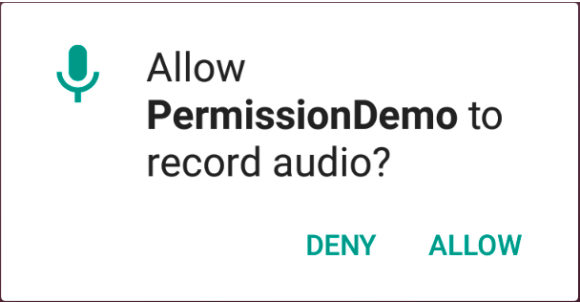


Figure 60-2

The full list of permissions that fall into the dangerous category is contained in Table 60-1:

Permission Group	Permission
Calendar	READ_CALENDAR
	WRITE_CALENDAR
Camera	CAMERA
Contacts	READ_CONTACTS
	WRITE_CONTACTS
	GET_ACCOUNTS
Location	ACCESS_FINE_LOCATION
	ACCESS_COARSE_LOCATION
Microphone	RECORD_AUDIO
Notifications	POST_NOTIFICATIONS
Phone	READ_PHONE_STATE
	CALL_PHONE
	READ_CALL_LOG
	WRITE_CALL_LOG
	ADD_VOICEMAIL
	USE_SIP
	PROCESS_OUTGOING_CALLS
Sensors	BODY_SENSORS
SMS	SEND_SMS
	RECEIVE_SMS
	READ_SMS
	RECEIVE_WAP_PUSH
	RECEIVE_MMS



Storage	MANAGE_EXTERNAL_STORAGE READ_EXTERNAL_STORAGE WRITE_EXTERNAL_STORAGE
---------	--

Table 60-1

The `MANAGE_EXTERNAL_STORAGE` permission gives the app access to all files located on the external storage of the device, including those belonging to other apps. Consequently, permission will only be enabled for your app once Google has verified during the review process that this level of access is needed. To test your app in advance of submitting it to the Google Play store, the following *adb* command can be executed to temporarily enable access for the app on the testing device:

```
adb shell appops set --uid <package name> MANAGE_EXTERNAL_STORAGE allow
```

This mode can be disabled as follows:

```
adb shell appops set --uid <package name> MANAGE_EXTERNAL_STORAGE default
```

## 60.2 Creating the Permissions Example Project

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

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

## 60.3 Checking for a Permission

The Android Support Library contains a number of methods that can be used to seek and manage dangerous permissions within the code of an Android app. These API calls can be made safely regardless of the version of Android on which the app is running, but will only perform meaningful tasks when executed on Android 6.0 or later.

Before an app attempts to make use of a feature that requires approval of a dangerous permission, and regardless of whether or not permission was previously granted, the code must check that the permission has been granted. This can be achieved via a call to the *checkSelfPermission()* method of the *ContextCompat* class, passing through as arguments a reference to the current activity and the permission being requested. The method will check whether the permission has been previously granted and return an integer value matching *PackageManager.PERMISSION\_GRANTED* or *PackageManager.PERMISSION\_DENIED*.

Within the *MainActivity.java* file of the example project, modify the code to check whether permission has been granted for the app to record audio:

```
package com.ebookfrenzy.permissiondemoactivity;

import androidx.appcompat.app.AppCompatActivity;
import androidx.core.content.ContextCompat;
import androidx.annotation.NonNull;

import android.os.Bundle;
import android.Manifest;
import android.content.pm.PackageManager;
import android.util.Log;
```

## Making Runtime Permission Requests in Android

```
public class MainActivity extends AppCompatActivity {  
  
    private static final String TAG = "PermissionDemo";  
  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_permission_demo);  
  
        setupPermissions();  
    }  
  
    private void setupPermissions() {  
        int permission = ContextCompat.checkSelfPermission(this,  
            Manifest.permission.RECORD_AUDIO);  
  
        if (permission != PackageManager.PERMISSION_GRANTED) {  
            Log.i(TAG, "Permission to record denied");  
        }  
    }  
}
```

} Edit the *AndroidManifest.xml* file (located in the Project tool window under *app -> manifests*) and add a line to request recording permission as follows:

```
<?xml version="1.0" encoding="utf-8"?>  
<manifest xmlns:android="http://schemas.android.com/apk/res/android"  
    package="com.ebookfrenzy.permissiondemoactivity" >  
  
    <uses-permission android:name="android.permission.RECORD_AUDIO" />  
  
    <application  
        android:allowBackup="true"  
        android:icon="@mipmap/ic_launcher"  
        android:label="@string/app_name"  
        android:supportRtl="true"  
        android:theme="@style/AppTheme" >  
        <activity android:name=".MainActivity" >  
            <intent-filter>  
                <action android:name="android.intent.action.MAIN" />  
  
                <category  
                    android:name="android.intent.category.LAUNCHER" />  
            </intent-filter>  
        </activity>  
    </application>  
</manifest>
```

Run the app on a device or emulator and open the Logcat tool window. Note that even though the permission has been added to the manifest file, the permission denied message appears. This is because Android requires that in addition to adding the request to the manifest file, the app must also request dangerous permissions at runtime.

## 60.4 Requesting Permission at Runtime

A permission request is made via a call to the *requestPermissions()* method of the *ActivityCompat* class. When this method is called, the permission request is handled asynchronously and a method named *onRequestPermissionsResult()* is called when the task is completed.

The *requestPermissions()* method takes as arguments a reference to the current activity, together with the identifier of the permission being requested and a request code. The request code can be any integer value and will be used to identify which request has triggered the call to the *onRequestPermissionsResult()* method. Modify the *MainActivity.java* file to declare a request code and request recording permission if the permission check failed:

```
.
.
import androidx.core.app.ActivityCompat;
.
.
public class MainActivity extends AppCompatActivity {

    private static final String TAG = "PermissionDemo";
    private static final int RECORD_REQUEST_CODE = 101;
.
.
    @Override
    private void setupPermissions() {

        int permission = ContextCompat.checkSelfPermission(this,
            Manifest.permission.RECORD_AUDIO);

        if (permission != PackageManager.PERMISSION_GRANTED) {
            Log.i(TAG, "Permission to record denied");
            makeRequest();
        }
    }

    protected void makeRequest() {
        ActivityCompat.requestPermissions(this,
            new String[]{Manifest.permission.RECORD_AUDIO},
            RECORD_REQUEST_CODE);
    }
}
```

Next, implement the *onRequestPermissionsResult()* method so that it reads as follows:

```
@Override
```

## Making Runtime Permission Requests in Android

```
public void onRequestPermissionsResult(int requestCode,
    @NonNull String[] permissions, @NonNull int[] grantResults) {
    super.onRequestPermissionsResult(requestCode, permissions, grantResults);
    if (requestCode == RECORD_REQUEST_CODE) {
        if (grantResults.length == 0
            || grantResults[0] !=
                PackageManager.PERMISSION_GRANTED) {
            Log.i(TAG, "Permission has been denied by user");
        } else {
            Log.i(TAG, "Permission has been granted by user");
        }
    }
}
```

Compile and run the app on an emulator or device and note that a dialog seeking permission to record audio appears as shown in Figure 60-3:

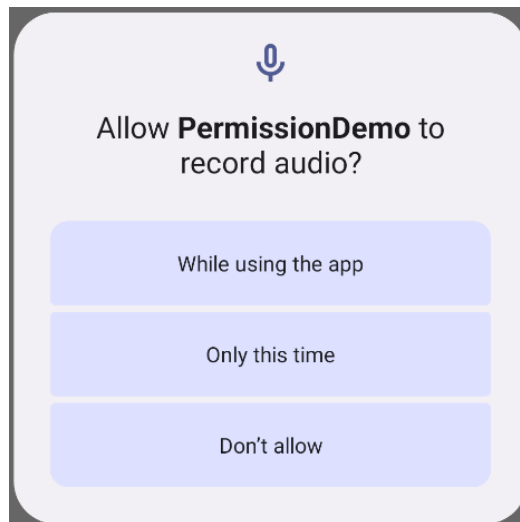


Figure 60-3

Tap the *While using the app* button and check that the “Permission has been granted by user” message appears in the Logcat panel.

Once the user has granted the requested permission, the *checkSelfPermission()* method call will return a `PERMISSION_GRANTED` result on future app invocations until the user uninstalls and re-installs the app or changes the permissions for the app in Settings.

### 60.5 Providing a Rationale for the Permission Request

As is evident from Figure 60-3, the user has the option to deny the requested permission. In this case, the app will continue to request the permission each time that it is launched by the user unless the user selected the “Never ask again” option before clicking on the Deny button. Repeated denials by the user may indicate that the user doesn’t understand why the permission is required by the app. The user might, therefore, be more likely to grant permission if the reason for the requirements is explained when the request is made. Unfortunately, it is not possible to change the content of the request dialog to include such an explanation.

An explanation is best included in a separate dialog which can be displayed before the request dialog is presented to the user. This raises the question as to when to display this explanation dialog. The Android documentation recommends that an explanation dialog only be shown if the user has previously denied the permission and provides a method to identify when this is the case.

A call to the *shouldShowRequestPermissionRationale()* method of the *ActivityCompat* class will return a true result if the user has previously denied a request for the specified permission, and a false result if the request has not previously been made. In the case of a true result, the app should display a dialog containing a rationale for needing the permission and, once the dialog has been read and dismissed by the user, the permission request should be repeated.

To add this functionality to the example app, modify the *onCreate()* method so that it reads as follows:

```
.
.
import android.app.AlertDialog;
.
.
private void setupPermissions() {

    int permission = ContextCompat.checkSelfPermission(this,
        Manifest.permission.RECORD_AUDIO);

    if (permission != PackageManager.PERMISSION_GRANTED) {
        Log.i(TAG, "Permission to record denied");

        if (ActivityCompat.shouldShowRequestPermissionRationale(this,
            Manifest.permission.RECORD_AUDIO)) {
            AlertDialog.Builder builder =
                new AlertDialog.Builder(this);
            builder.setMessage("Permission to access the microphone is required
for this app to record audio.")
                .setTitle("Permission required");

            builder.setPositiveButton("OK",
                (dialog, id) -> makeRequest());

            AlertDialog dialog = builder.create();
            dialog.show();
        } else {
            makeRequest();
        }
    }
}
```

The method still checks whether or not the permission has been granted, but now also identifies whether a rationale needs to be displayed. If the user has previously denied the request, a dialog is displayed containing an explanation and an OK button on which a listener is configured to call the *makeRequest()* method when the button is tapped. If the permission request has not previously been made, the code moves directly to seeking

permission.

### 60.6 Testing the Permissions App

On the device or emulator session on which testing is being performed, launch the Settings app, select the *Apps* option and scroll to and select the PermissionDemo app. On the app settings screen, tap the uninstall button to remove the app from the device.

Run the app once again and, when the permission request dialog appears, click on the *Don't allow* button. Stop and restart the app and verify that the rationale dialog appears. Tap the OK button and, when the permission request dialog appears, tap the *While using the app* button.

Return to the Settings app, select the Apps option and select the PermissionDemo app once again from the list. Once the settings for the app are listed, verify that the Permissions section lists the *Microphone* permission:

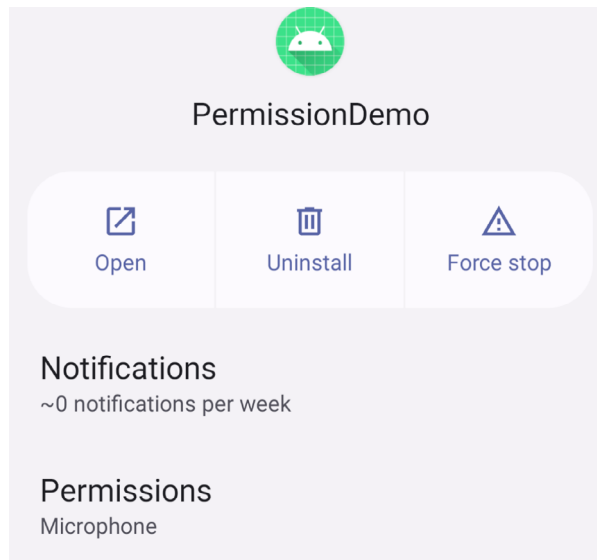


Figure 60-4

### 60.7 Summary

Before the introduction of Android 6.0 the only step necessary for an app to request permission to access certain functionality was to add an appropriate line to the application's manifest file. The user would then be prompted to approve the permission at the point that the app was installed. This is still the case for most permissions, with the exception of a set of permissions that are considered dangerous. Permissions that are considered dangerous usually have the potential to allow an app to violate the user's privacy such as allowing access to the microphone, contacts list or external storage.

As outlined in this chapter, apps based on Android 6 or later must now request dangerous permission approval from the user when the app launches in addition to including the permission request in the manifest file.

## 65. The Android Room Persistence Library

Included with the Android Architecture Components, the Room persistence library is designed specifically to make it easier to add database storage support to Android apps in a way that is consistent with the Android architecture guidelines. With the basics of SQLite databases covered in the previous chapter, this chapter will explore the basic concepts behind Room-based database management, the key elements that work together to implement Room support within an Android app and how these are implemented in terms of architecture and coding. Having covered these topics, the next two chapters will put this theory into practice in the form of an example Room database project.

### 65.1 Revisiting Modern App Architecture

The chapter entitled “*Modern Android App Architecture with Jetpack*” introduced the concept of modern app architecture and stressed the importance of separating different areas of responsibility within an app. The diagram illustrated in Figure 65-1 outlines the recommended architecture for a typical Android app:

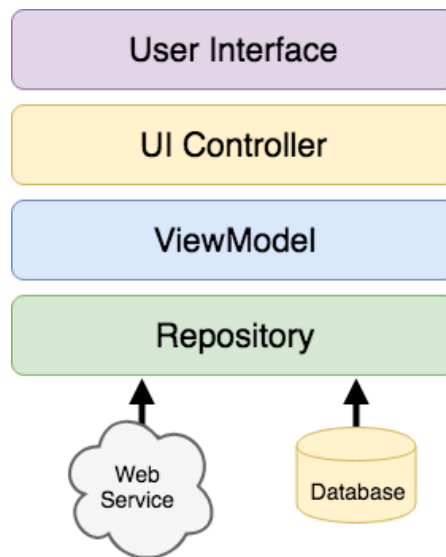


Figure 65-1

With the top three levels of this architecture covered in some detail in earlier chapters of this book, it is now time to begin exploration of the repository and database architecture levels in the context of the Room persistence library.

### 65.2 Key Elements of Room Database Persistence

Before going into greater detail later in the chapter, it is first worth summarizing the key elements involved in working with SQLite databases using the Room persistence library:

### 65.2.1 Repository

As previously discussed, the repository module contains all of the code necessary for directly handling all data sources used by the app. This avoids the need for the UI controller and ViewModel to contain code that directly accesses sources such as databases or web services.

### 65.2.2 Room Database

The room database object provides the interface to the underlying SQLite database. It also provides the repository with access to the Data Access Object (DAO). An app should only have one room database instance which may then be used to access multiple database tables.

### 65.2.3 Data Access Object (DAO)

The DAO contains the SQL statements required by the repository to insert, retrieve and delete data within the SQLite database. These SQL statements are mapped to methods which are then called from within the repository to execute the corresponding query.

### 65.2.4 Entities

An entity is a class that defines the schema for a table within the database and defines the table name, column names and data types, and identifies which column is to be the primary key. In addition to declaring the table schema, entity classes also contain getter and setter methods that provide access to these data fields. The data returned to the repository by the DAO in response to the SQL query method calls will take the form of instances of these entity classes. The getter methods will then be called to extract the data from the entity object. Similarly, when the repository needs to write new records to the database, it will create an entity instance, configure values on the object via setter calls, then call insert methods declared in the DAO, passing through entity instances to be saved.

### 65.2.5 SQLite Database

The actual SQLite database responsible for storing and providing access to the data. The app code, including the repository, should never make direct access to this underlying database. All database operations are performed using a combination of the room database, DAOs and entities.

The architecture diagram in Figure 65-2 illustrates the way in which these different elements interact to provide Room-based database storage within an Android app:

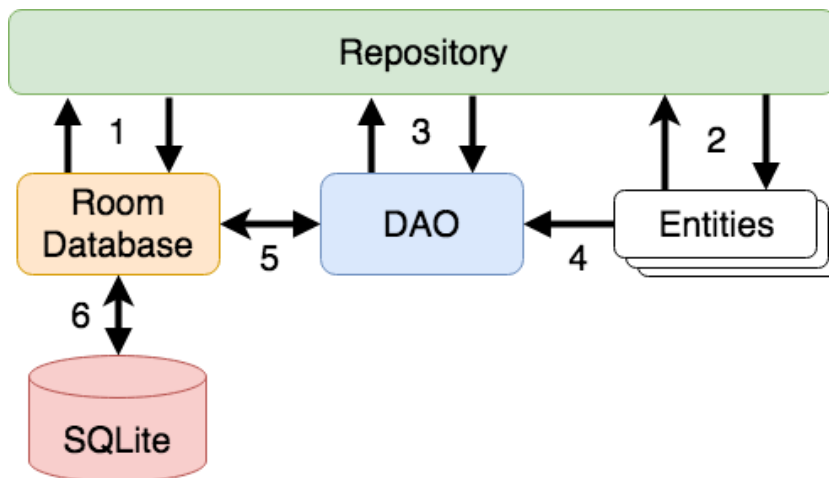


Figure 65-2



The numbered connections in the above architecture diagram can be summarized as follows:

1. The repository interacts with the Room Database to get a database instance which, in turn, is used to obtain references to DAO instances.
2. The repository creates entity instances and configures them with data before passing them to the DAO for use in search and insertion operations.
3. The repository calls methods on the DAO passing through entities to be inserted into the database and receives entity instances back in response to search queries.
4. When a DAO has results to return to the repository it packages those results into entity objects.
5. The DAO interacts with the Room Database to initiate database operations and handle results.
6. The Room Database handles all of the low level interactions with the underlying SQLite database, submitting queries and receiving results.

With a basic outline of the key elements of database access using the Room persistent library covered, it is now time to explore entities, DAOs, room databases and repositories in more detail.

## 65.3 Understanding Entities

Each database table will have associated with it an entity class. This class defines the schema for the table and takes the form of a standard Java class interspersed with some special Room annotations. An example Java class declaring the data to be stored within a database table might read as follows:

```
public class Customer {

    private int id;
    private String name;
    private String address;

    public Customer(String name, String address) {
        this.id = id;
        this.name = name;
        this.address = address;
    }

    public int getId() {
        return this.id;
    }

    public String getName() {
        return this.name;
    }

    public int getAddress() {
        return this.address;
    }

    public void setId(int id) {
```

```
        this.id = id;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void setAddress(int quantity) {
        this.address = address;
    }
}
```

As currently implemented, the above code declares a basic Java class containing a number of variables representing database table fields and a collection of getter and setter methods. This class, however, is not yet an entity. To make this class into an entity and to make it accessible within SQL statements, some Room annotations need to be added as follows:

```
@Entity(tableName = "customers")
public class Customer {

    @PrimaryKey(autoGenerate = true)
    @NonNull
    @ColumnInfo(name = "customerId")
    private int id;

    @ColumnInfo(name = "customerName")
    private String name;

    private String address;

    public Customer(String name, String address) {
        this.id = id;
        this.name = name;
        this.address = address;
    }

    public int getId() {
        return this.id;
    }

    public String getName() {
        return this.name;
    }

    public String getAddress() {
        return this.address;
    }
}
```

```

    public void setId(@NonNull int id) {
        this.id = id;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void setAddress(int quantity) {
        this.address = address;
    }
}

```

The above annotations begin by declaring that the class represents an entity and assigns a table name of “customers”. This is the name by which the table will be referenced in the DAO SQL statements:

```
@Entity(tableName = "customers")
```

Every database table needs a column to act as the primary key. In this case, the customer id is declared as the primary key. Annotations have also been added to assign a column name to be referenced in SQL queries and to indicate that the field cannot be used to store null values. Finally, the id value is configured to be auto-generated. This means that the id assigned to new records will be automatically generated by the system to avoid duplicate keys.

```

@PrimaryKey(autoGenerate = true)
@NonNull
@ColumnInfo(name = "customerId")
private int id;

```

A column name is also assigned to the customer name field. Note, however, that no column name was assigned to the address field. This means that the address data will still be stored within the database, but that it is not required to be referenced in SQL statements. If a field within an entity is not required to be stored within a database, simply use the `@Ignore` annotation:

```

@Ignore
private String myString;

```

Finally, the setter method for the id variable is modified to prevent attempts to assign a null value:

```

public void setId(@NonNull int id) {
    this.id = id;
}

```

Annotations may also be included within an entity class to establish relationships with other entities using a relational database concept referred to as *foreign keys*. Foreign keys allow a table to reference the primary key in another table. For example, a relationship could be established between an entity named Purchase and our existing Customer entity as follows:

```

@Entity(foreignKeys = {@ForeignKey(entity = Customer.class,
    parentColumns = "customerId",
    childColumns = "buyerId",
    onDelete = ForeignKey.CASCADE,
    onUpdate = ForeignKey.RESTRICT})

```

```
public class Purchase {

    @PrimaryKey(autoGenerate = true)
    @ColumnInfo(name = "purchaseId")
    private int purchaseId;

    @ColumnInfo(name = "buyerId")
    private int buyerId;

}
```

Note that the foreign key declaration also specifies the action to be taken when a parent record is deleted or updated. Available options are CASCADE, NO\_ACTION, RESTRICT, SET\_DEFAULT and SET\_NULL.

## 65.4 Data Access Objects

A Data Access Object provides a way to access the data stored within a SQLite database. A DAO is declared as a standard Java interface with some additional annotations that map specific SQL statements to methods that may then be called by the repository.

The first step is to create the interface and declare it as a DAO using the `@Dao` annotation:

```
@Dao
public interface CustomerDao {

}
```

Next, entries are added consisting of SQL statements and corresponding method names. The following declaration, for example, allows all of the rows in the customers table to be retrieved via a call to a method named `getAllCustomers()`:

```
@Dao
public interface CustomerDao {

    @Query("SELECT * FROM customers")
    LiveData<List<Customer>> getAllCustomers();

}
```

Note that the `getAllCustomers()` method returns a List object containing a Customer entity object for each record retrieved from the database table. The DAO is also making use of LiveData so that the repository is able to observe changes to the database.

Arguments may also be passed into the methods and referenced within the corresponding SQL statements. Consider the following DAO declaration which searches for database records matching a customer's name (note that the column name referenced in the WHERE condition is the name assigned to the column in the entity class):

```
@Query("SELECT * FROM customers WHERE name = :customerName")
List<Customer> findCustomer(String customerName);
```

In this example, the method is passed a string value which is, in turn, included within an SQL statement by prefixing the variable name with a colon (:).

A basic insertion operation can be declared as follows using the `@Insert convenience annotation`:

```
@Insert
void addCustomer(Customer customer);
```

This is referred to as a convenience annotation because the Room persistence library can infer that the Customer entity passed to the `addCustomer()` method is to be inserted into the database without the need for the SQL insert statement to be provided. Multiple database records may also be inserted in a single transaction as follows:

```
@Insert
public void insertCustomers(Customer... customers);
```

The following DAO declaration deletes all records matching the provided customer name:

```
@Query("DELETE FROM customers WHERE name = :name")
void deleteCustomer(String name);
```

As an alternative to using the `@Query` annotation to perform deletions, the `@Delete` convenience annotation may also be used. In the following example, all of the Customer records that match the set of entities passed to the `deleteCustomers()` method will be deleted from the database:

```
@Delete
public void deleteCustomers(Customer... customers);
```

The `@Update` convenience annotation provides similar behavior when updating records:

```
@Update
public void updateCustomers(Customer... customers);
```

The DAO methods for these types of database operations may also be declared to return an int value indicating the number of rows affected by the transaction, for example:

```
@Delete
public int deleteCustomers(Customer... customers);
```

## 65.5 The Room Database

The Room database class is created by extending the `RoomDatabase` class and acts as a layer on top of the actual SQLite database embedded into the Android operating system. The class is responsible for creating and returning a new room database instance and for providing access to the DAO instances associated with the database.

The Room persistence library provides a database builder for creating database instances. Each Android app should only have one room database instance, so it is best to implement defensive code within the class to prevent more than one instance being created.

An example Room Database implementation for use with the example customer table is outlined in the following code listing:

```
import android.content.Context;
import android.arch.persistence.room.Database;
import android.arch.persistence.room.Room;
import android.arch.persistence.room.RoomDatabase;

@Database(entities = {Customer.class}, version = 1)
public class CustomerRoomDatabase extends RoomDatabase {

    public abstract CustomerDao customerDao();

    private static CustomerRoomDatabase INSTANCE;
```

## The Android Room Persistence Library

```
static CustomerRoomDatabase getDatabase(final Context context) {
    if (INSTANCE == null) {
        synchronized (CustomerRoomDatabase.class) {
            if (INSTANCE == null) {
                INSTANCE = Room.databaseBuilder(
                    context.getApplicationContext(),
                    CustomerRoomDatabase.class, "customer_database")
                    .build();
            }
        }
    }
    return INSTANCE;
}
```

Important areas to note in the above example are the annotation above the class declaration declaring the entities with which the database is to work, the code to check that an instance of the class has not already been created and assignment of the name “customer\_database” to the instance.

## 65.6 The Repository

The repository is responsible for getting a Room Database instance, using that instance to access associated DAOs and then making calls to DAO methods to perform database operations. A typical constructor for a repository designed to work with a Room Database might read as follows:

```
public class CustomerRepository {

    private CustomerDao customerDao;
    private CustomerRoomDatabase db;

    public CustomerRepository(Application application) {
        db = CustomerRoomDatabase.getDatabase(application);
        customerDao = db.customerDao();
    }
    .
    .
    .
}
```

Once the repository has access to the DAO, it can make calls to the data access methods. The following code, for example, calls the *getAllCustomers()* DAO method:

```
private LiveData<List<Customer>> allCustomers;
allCustomers = customerDao.getAllCustomers();
```

When calling DAO methods, it is important to note that unless the method returns a LiveData instance (which automatically runs queries on a separate thread), the operation cannot be performed on the app’s main thread. In fact, attempting to do so will cause the app to crash with the following diagnostic output:

```
Cannot access database on the main thread since it may potentially lock the UI
for a long period of time
```

Since some database transactions may take a longer time to complete, running the operations on a separate thread avoids the app appearing to lock up. As will be demonstrated in the chapter entitled “An Android Room

*Database and Repository Tutorial*”, this problem can be easily resolved by making use of Java threads (for more information or a reminder of how to use threads, refer back to the chapter entitled “A Basic Overview of Java Threads, Handlers and Executors”).

## 65.7 In-Memory Databases

The examples outlined in this chapter involved the use of a SQLite database that exists as a database file on the persistent storage of an Android device. This ensures that the data persists even after the app process is terminated.

The Room database persistence library also supports *in-memory* databases. These databases reside entirely in memory and are lost when the app terminates. The only change necessary to work with an in-memory database is to call the `Room.inMemoryDatabaseBuilder()` method of the Room Database class instead of `Room.databaseBuilder()`. The following code shows the difference between the method calls (note that the in-memory database does not require a database name):

```
// Create a file storage based database
INSTANCE = Room.databaseBuilder(context.getApplicationContext(),
                                CustomerRoomDatabase.class, "customer_database")
                                .build();

// Create an in-memory database
INSTANCE = Room.inMemoryDatabaseBuilder(context.getApplicationContext(),
                                         CustomerRoomDatabase.class)
                                         .build();
```

## 65.8 Database Inspector

Android Studio includes a Database Inspector tool window which allows the Room databases associated with running apps to be viewed, searched and modified as shown in Figure 65-3:

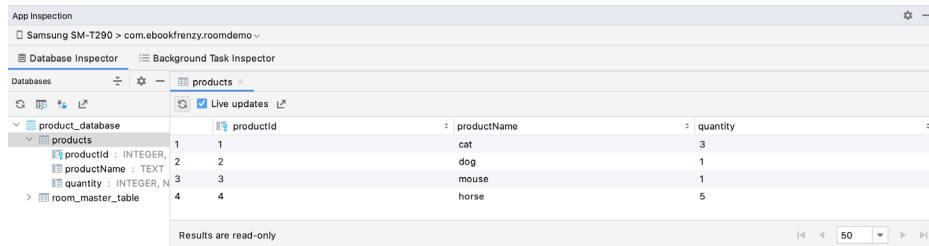


Figure 65-3

Use of the Database Inspector will be covered in the chapter entitled “An Android Room Database and Repository Tutorial”.

## 65.9 Summary

The Android Room persistence library is bundled with the Android Architecture Components and acts as an abstract layer above the lower level SQLite database. The library is designed to make it easier to work with databases while conforming to the Android architecture guidelines. This chapter has introduced the different elements that interact to build Room-based database storage into Android app projects including entities, repositories, data access objects, annotations and Room Database instances.

With the basics of SQLite and the Room architecture component covered, the next step is to create an example app that puts this theory into practice. Since the user interface for the example application will require a forms

based layout, the next chapter, entitled “*An Android TableLayout and TableRow Tutorial*”, will detour slightly from the core topic by introducing the basics of the TableLayout and TableRow views.



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

When Google decided to introduce a map service many years ago, it is hard to say whether or not they ever anticipated having a version available for integration into mobile applications. When the first web based version of what would eventually be called Google Maps was introduced in 2005, the iPhone had yet to ignite the smartphone revolution and the company that was developing the Android operating system would not be acquired by Google for another six months. Whatever aspirations Google had for the future of Google Maps, it is remarkable to consider that all of the power of Google Maps can now be accessed directly via Android applications using the Google Maps Android API.

This chapter is intended to provide an overview of the Google Maps system and Google Maps Android API. The chapter will provide an overview of the different elements that make up the API, detail the steps necessary to configure a development environment to work with Google Maps and then work through some code examples demonstrating some of the basics of Google Maps Android integration.

### 74.1 The Elements of the Google Maps Android API

The Google Maps Android API consists of a core set of classes that combine to provide mapping capabilities in Android applications. The key elements of a map are as follows:

- **GoogleMap** – The main class of the Google Maps Android API. This class is responsible for downloading and displaying map tiles and for displaying and responding to map controls. The GoogleMap object is not created directly by the application but is instead created when MapView or MapFragment instances are created. A reference to the GoogleMap object can be obtained within application code via a call to the *getMap()* method of a MapView, MapFragment or SupportMapFragment instance.
- **MapView** - A subclass of the View class, this class provides the view canvas onto which the map is drawn by the GoogleMap object, allowing a map to be placed in the user interface layout of an activity.
- **SupportMapFragment** – A subclass of the Fragment class, this class allows a map to be placed within a Fragment in an Android layout.
- **Marker** – The purpose of the Marker class is to allow locations to be marked on a map. Markers are added to a map by obtaining a reference to the GoogleMap object associated with a map and then making a call to the *addMarker()* method of that object instance. The position of a marker is defined via Longitude and Latitude. Markers can be configured in a number of ways, including specifying a title, text and an icon. Markers may also be made to be “draggable”, allowing the user to move the marker to different positions on a map.
- **Shapes** – The drawing of lines and shapes on a map is achieved through the use of the *Polyline*, *Polygon* and *Circle* classes.
- **UiSettings** – The UiSettings class provides a level of control from within an application of which user interface controls appear on a map. Using this class, for example, the application can control whether or not the zoom, current location and compass controls appear on a map. This class can also be used to configure which touch screen gestures are recognized by the map.

- **My Location Layer** – When enabled, the My Location Layer displays a button on the map which, when selected by the user, centers the map on the user's current geographical location. If the user is stationary, this location is represented on the map by a blue marker. If the user is in motion the location is represented by a chevron indicating the user's direction of travel.

The best way to gain familiarity with the Google Maps Android API is to work through an example. The remainder of this chapter will create a Google Maps based application while highlighting the key areas of the API.

## 74.2 Creating the Google Maps Project

Select the *New Project* option from the welcome screen and, within the resulting new project dialog, choose the No Activity template before clicking on the Next button.

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

Next, right-click on the *app -> java -> com.ebookfrenzy.mapdemo* entry in the Project tool window and select the *New -> Google -> Google Maps Views Activity* menu option. Finally, enable the Launcher Activity checkbox in the New Android Activity dialog before clicking the Finish button:

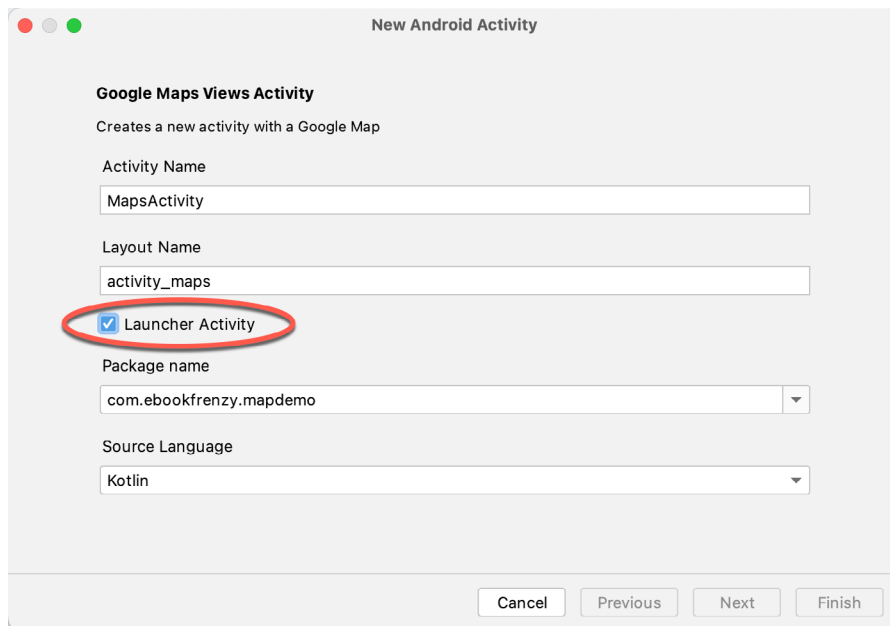


Figure 74-1

## 74.3 Creating a Google Cloud Billing Account

Before you can use the Google Map APIs you must first create Google Cloud billing account (unless you already have one, in which case you can skip to the next section). To do this, open a browser and use the following link to navigate to the Google Cloud Console:

<https://console.cloud.google.com/>

Next, click on the menu button in the top left-hand corner of the console page and select the Billing entry as illustrated in Figure 74-2 below:

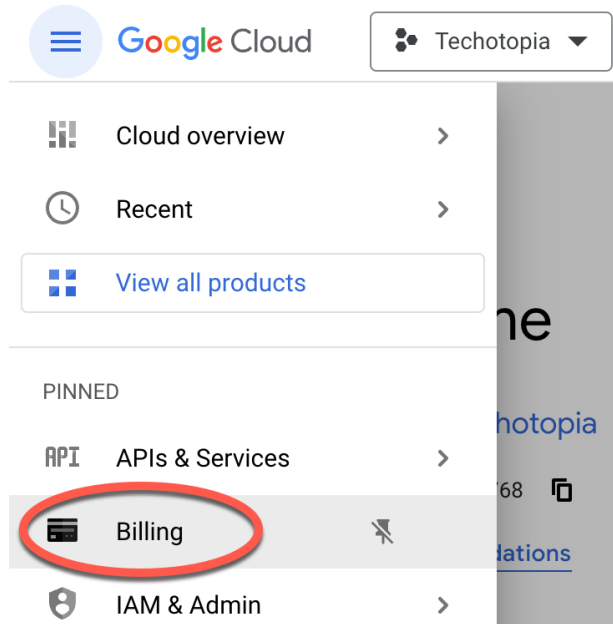


Figure 74-2

On the Billing page, select the option to add a new billing account and then follow the steps to start a free trial. You will need to provide a credit card to open the account, but Google won't charge you when the free trial ends without your consent.

## 74.4 Creating a New Google Cloud Project

The next step is to create a Google Cloud project to be associated with the MapDemo app. To do this, return to the Google Cloud Console dashboard by using the following URL:

<https://console.cloud.google.com/home/dashboard>

Within the dashboard, click the *Select a project* button located in the top toolbar:

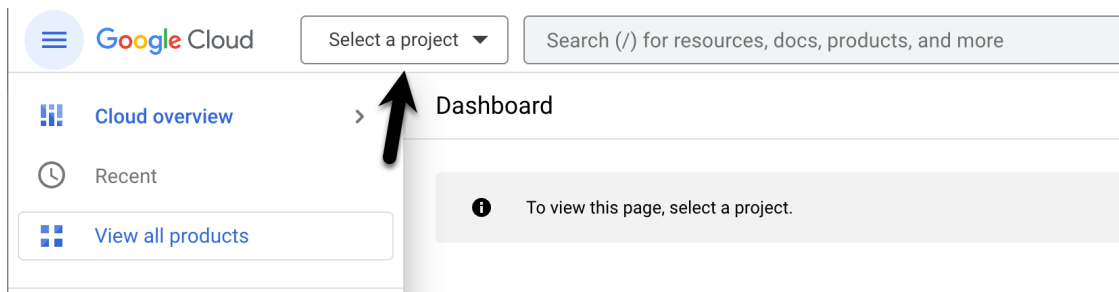


Figure 74-3

When the project selection dialog appears, click on the New Project button (highlighted in Figure 74-4):



Figure 74-4

When the new project screen appears, provide a name for the project. The console will display a default id for the project beneath the project name field. If you don't like the default id, click the Edit button to change it:

### New Project

Project name \*  
My Example Project ?

Project ID: my-example-project-375214. It cannot be changed later. [EDIT](#)

Location \*  
No organization [BROWSE](#)

Parent organization or folder

[CREATE](#) [CANCEL](#)

Figure 74-5

Click the Create button, and after a brief pause, you will be returned to the dashboard where your new project will be listed.

## 74.5 Enabling the Google Maps SDK

Now that we have created a new Google Cloud project, the next step is to allow the project to use the Google Maps SDK. To enable Google Maps support, select your project in the Google Cloud Console, click the menu button in the top left-hand corner and select the Google Maps Platform entry. Then, from the resulting menu, select the APIs option as shown in Figure 74-6:

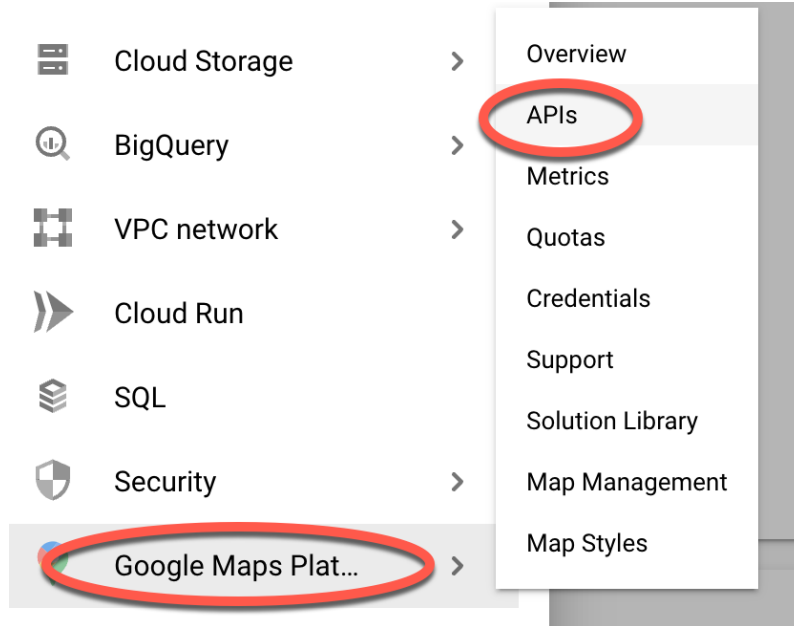


Figure 74-6

On the APIs screen, click on the *Maps SDK for Android* option and, on the resulting screen, click the Enable button:

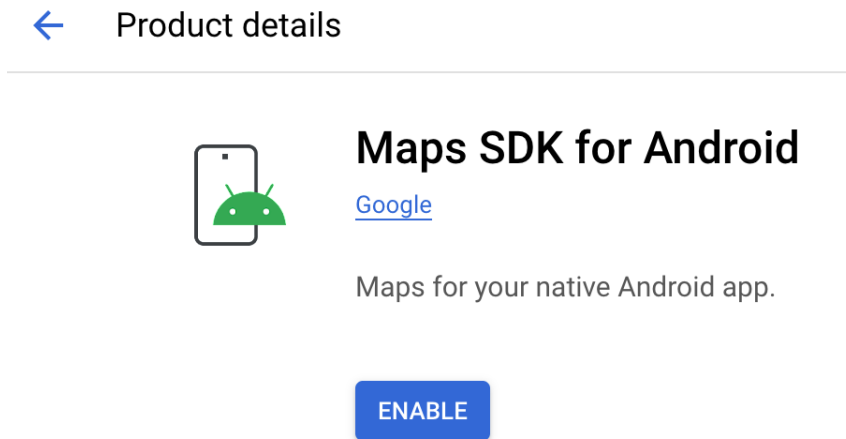


Figure 74-7

Repeat the above steps to enable the Geocoding API credential, which will be needed later in the chapter to allow our app to display the user's current location.

Once you have enabled the credentials for your project, click the back arrow to return to the product details page in preparation for the next step.

## 74.6 Generating a Google Maps API Key

Before an application can use the Google Maps Android SDK, it must first be configured with an API key that will associate it with a Maps-enabled Google Cloud project. To generate an API key, select the Credentials menu

option (marked A in Figure 74-8) followed by Create Credentials button (B):

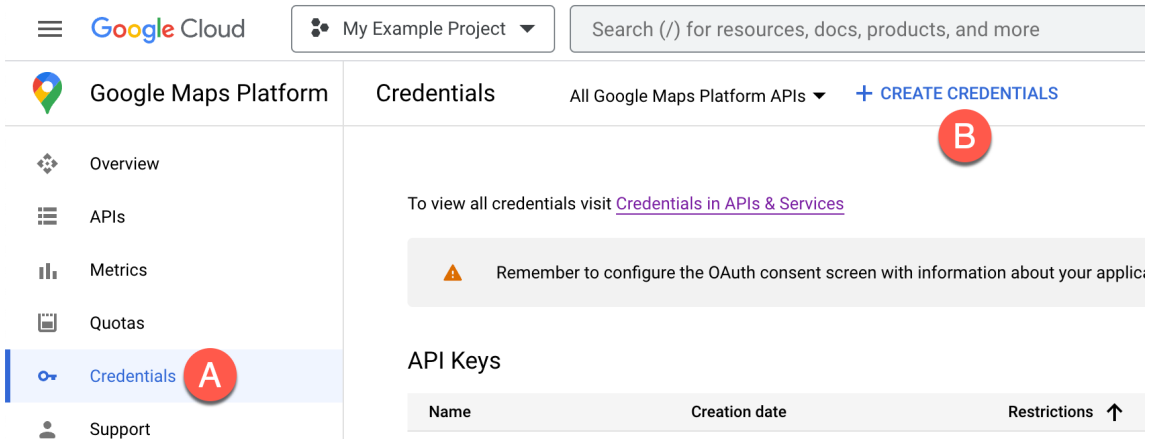


Figure 74-8

After the credential has been created, a dialog will appear displaying the API key. Copy the key before closing the API dialog:

## API key created

Use this key in your application by passing it with the `key=API_KEY` parameter.




 This key is unrestricted. To prevent unauthorized use, we recommend restricting where and for which APIs it can be used. [Edit API key](#) to add restrictions. [Learn more](#)

Figure 74-9

### 74.7 Adding the API Key to the Android Studio Project

Now that we have generated an API key that will allow our app to use the Google Maps SDK, we need to add it to our project. Return to Android Studio, edit the *manifests* -> *AndroidManifest.xml* file, and locate the API key entry, which will read as follows:

```
<meta-data
    android:name="com.google.android.geo.API_KEY"
    android:value="YOUR_API_KEY" />
```

Delete the text that reads “YOUR\_API\_KEY” and replace it with the API key created in the Google Play Console.

Next, edit the *Gradle Scripts* -> *local.properties* file and add a new line that reads as follows (where the API key for your project replaces YOUR\_API\_KEY):

```
MAPS_API_KEY=YOUR_API_KEY
```

## 74.8 Testing the Application

Perform a test run of the application to verify that the API key is correctly configured. Assuming the configuration is correct, the application will run and display a map on the screen.

If a map is not displayed, check the following areas:

- If the application is running on an emulator, make sure that the emulator is running a version of Android that includes the Google APIs. The current operating system can be changed for an AVD configuration by selecting the *Tools -> Android -> AVD Manager* menu option, clicking on the pencil icon in the *Actions* column of the AVD followed by the *Change...* button next to the current Android version. Within the system image dialog, select a target that includes the Google APIs.
- Check the Logcat output for any areas relating to authentication problems with regard to the Google Maps API. This usually means the API key was entered incorrectly. Ensure that the API key in both the *AndroidManifest.xml* and *local.properties* files matches the key generated in the Google Cloud console.
- Verify within the Google API Console that *Maps SDK for Android* has been enabled in the Credentials panel.

## 74.9 Understanding Geocoding and Reverse Geocoding

It is impossible to talk about maps and geographical locations without first covering the subject of Geocoding. Geocoding can best be described as the process of converting a textual-based geographical location (such as a street address) into geographical coordinates expressed in terms of longitude and latitude.

Geocoding can be achieved using the Android Geocoder class. An instance of the Geocoder class can, for example, be passed a string representing a location such as a city name, street address or airport code. The Geocoder will attempt to find a match for the location and return a list of Address objects that potentially match the location string, ranked in order with the closest match at position 0 in the list. A variety of information can then be extracted from the Address objects, including the longitude and latitude of the potential matches.

The following code, for example, requests the location of the National Air and Space Museum in Washington, D.C.:

```
import java.io.IOException;
import java.util.List;

import android.location.Address;
import android.location.Geocoder;

.
.
double latitude;
double longitude;

List<Address> geocodeMatches = null;

try {
    geocodeMatches =
        new Geocoder(this).getFromLocationName(
            "600 Independence Ave SW, Washington, DC 20560", 1);
} catch (IOException e) {
    // TODO Auto-generated catch block
```

## Working with the Google Maps Android API in Android Studio

```
        e.printStackTrace();
    }

    if (!geocodeMatches.isEmpty())
    {
        latitude = geocodeMatches.get(0).getLatitude();
        longitude = geocodeMatches.get(0).getLongitude();
    }
}
```

Note that the value of 1 is passed through as the second argument to the *getFromLocationName()* method. This simply tells the Geocoder to return only one result in the array. Given the specific nature of the address provided, there should only be one potential match. For more vague location names, however, it may be necessary to request more potential matches and allow the user to choose the correct one.

The above code is an example of *forward-geocoding* in that coordinates are calculated based on a text location description. *Reverse-geocoding*, as the name suggests, involves the translation of geographical coordinates into a human readable address string. Consider, for example, the following code:

```
import java.io.IOException;
import java.util.List;

import android.location.Address;
import android.location.Geocoder;

.
.
List<Address> geocodeMatches = null;
String Address1;
String Address2;
String State;
String Zipcode;
String Country;

try {
    geocodeMatches =
        new Geocoder(this).getFromLocation(38.8874245, -77.0200729, 1);
} catch (IOException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}

if (!geocodeMatches.isEmpty())
{
    Address1 = geocodeMatches.get(0).getAddressLine(0);
    Address2 = geocodeMatches.get(0).getAddressLine(1);
    State = geocodeMatches.get(0).getAdminArea();
    Zipcode = geocodeMatches.get(0).getPostalCode();
    Country = geocodeMatches.get(0).getCountryName();
}
```



In this case the Geocoder object is initialized with latitude and longitude values via the *getFromLocation()* method. Once again, only a single matching result is requested. The text based address information is then extracted from the resulting Address object.

It should be noted that the geocoding is not actually performed on the Android device, but rather on a server to which the device connects when a translation is required and the results subsequently returned when the translation is complete. As such, geocoding can only take place when the device has an active internet connection.

## 74.10 Adding a Map to an Application

The simplest way to add a map to an application is to specify it in the user interface layout XML file for an activity. The following example layout file shows the SupportMapFragment instance added to the *activity\_maps.xml* file created by Android Studio:

```
<fragment xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:id="@+id/map"
    tools:context=".MapsActivity"
    android:name="com.google.android.gms.maps.SupportMapFragment"/>
```

## 74.11 Requesting Current Location Permission

As outlined in the chapter entitled “*Making Runtime Permission Requests in Android*”, certain permissions are considered dangerous and require special handling for Android 6.0 or later. One set of permissions allows applications to identify the user’s current location. Edit the *AndroidManifest.xml* file located under *app -> manifests* in the Project tool window and add the following permission lines:

```
<uses-permission
    android:name="android.permission.ACCESS_FINE_LOCATION" />

<uses-permission
    android:name="android.permission.ACCESS_COARSE_LOCATION" />
```

These settings will ensure that the app can provide permission for the app to obtain location information when installed on older versions of Android. To support Android 6.0 or later, however, we need to add some code to the *MapsActivity.java* file to request this permission at runtime.

Begin by adding some import directives and a constant to act as the permission request code:

```
package com.ebookfrenzy.mapdemo;

.
.
import androidx.annotation.NonNull;
import androidx.core.content.ContextCompat;
import androidx.core.app.ActivityCompat;
import android.Manifest;
import android.widget.Toast;
import android.content.pm.PackageManager;

.
.

public class MapsActivity extends FragmentActivity implements OnMapReadyCallback
```

```

{
    private static final int LOCATION_REQUEST_CODE = 101;
    private GoogleMap mMap;
    .
    .
}

```

Next, a method needs to be added to the class to request a specified permission from the user. Remaining within the *MapsActivity.java* class file, implement this method as follows:

```

protected void requestPermission(String permissionType,
                                int requestCode) {

    ActivityCompat.requestPermissions(this,
        new String[]{permissionType}, requestCode
    );
}

```

When the user has responded to the permission request, the *onRequestPermissionsResult()* method will be called on the activity. Remaining in the *MapsActivity.java* file, implement this method now so that it reads as follows:

```

@Override
public void onRequestPermissionsResult(int requestCode,
    @NonNull String[] permissions, @NonNull int[] grantResults) {

    super.onRequestPermissionsResult(requestCode, permissions, grantResults);
    if (requestCode == LOCATION_REQUEST_CODE) {
        if (grantResults.length == 0
            || grantResults[0] !=
                PackageManager.PERMISSION_GRANTED) {
            Toast.makeText(this,
                "Unable to show location - permission required",
                Toast.LENGTH_LONG).show();
        } else {

            SupportMapFragment mapFragment =
                (SupportMapFragment) getSupportFragmentManager()
                    .findFragmentById(R.id.map);
            mapFragment.getMapAsync(this);
        }
    }
}

```

If permission has not been granted by the user, the app displays a message indicating that the current location cannot be displayed. If, on the other hand, permission was granted, the map is refreshed to provide an opportunity for the location marker to be displayed.

## 74.12 Displaying the User's Current Location

Once the appropriate permission has been granted, the user's current location may be displayed on the map by obtaining a reference to the *GoogleMap* object associated with the displayed map and calling the

*setMyLocationEnabled()* method of that instance, passing through a value of *true*.

When the map is ready to display, the *onMapReady()* method of the activity is called. This method will also be called when the map is refreshed within the *onRequestPermissionsResult()* method above. By default, Android Studio has implemented this method and added some code to orient the map over Australia with a marker positioned over the city of Sidney. Locate and edit the *onMapReady()* method in the *MapsActivity.java* file to remove this template code and to add code to check the location permission has been granted before enabling display of the user's current location. If permission has not been granted, a request is made to the user via a call to the previously added *requestPermission()* method:

```
@Override
public void onMapReady(GoogleMap googleMap) {
    mMap = googleMap;

    // Add a marker in Sydney and move the camera
    LatLng sydney = new LatLng(-34, 151);
    mMap.addMarker(new MarkerOptions().position(sydney).title("Marker in
    Sydney"));
    mMap.moveCamera(CameraUpdateFactory.newLatLng(sydney));

    if (mMap != null) {
        int permission = ContextCompat.checkSelfPermission(this,
            Manifest.permission.ACCESS_FINE_LOCATION);

        if (permission == PackageManager.PERMISSION_GRANTED) {
            mMap.setMyLocationEnabled(true);
        } else {
            requestPermission(
                Manifest.permission.ACCESS_FINE_LOCATION,
                LOCATION_REQUEST_CODE);
        }
    }
}
```

When the app is now run, the dialog shown in Figure 74-10 will appear requesting location permission. If permission is granted, a blue dot will appear on the map indicating the current location of the device.

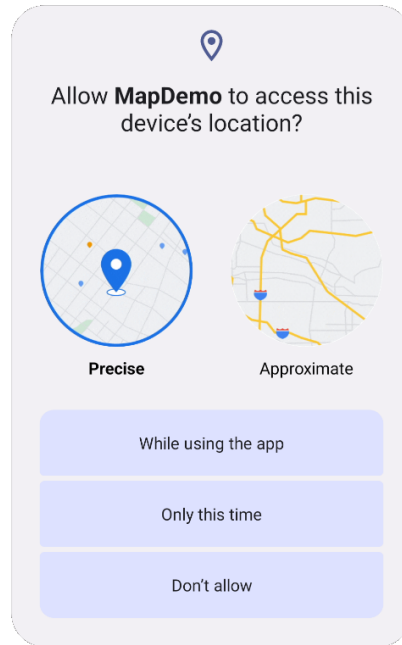


Figure 74-10

### 74.13 Changing the Map Type

The type of map displayed can be modified dynamically by making a call to the `setMapType()` method of the corresponding `GoogleMap` object, passing through one of the following values:

- **`GoogleMap.MAP_TYPE_NONE`** – An empty grid with no mapping tiles displayed.
- **`GoogleMap.MAP_TYPE_NORMAL`** – The standard view consisting of the classic road map.
- **`GoogleMap.MAP_TYPE_SATELLITE`** – Displays the satellite imagery of the map region.
- **`GoogleMap.MAP_TYPE_HYBRID`** – Displays satellite imagery with the road map superimposed.
- **`GoogleMap.MAP_TYPE_TERRAIN`** – Displays topographical information such as contour lines and colors.

The following code change to the `onMapReady()` method, for example, switches a map to Satellite mode:

```
.
.
if (mMap != null) {
    int permission = ContextCompat.checkSelfPermission(
        this, Manifest.permission.ACCESS_FINE_LOCATION);

    if (permission == PackageManager.PERMISSION_GRANTED) {
        mMap.setMyLocationEnabled(true);
    } else {
        requestPermission(Manifest.permission.ACCESS_FINE_LOCATION,
            LOCATION_REQUEST_CODE);
    }
}
```

```

mMap.setMapType(GoogleMap.MAP_TYPE_SATELLITE);
}
.
.

```

Alternatively, the map type may be specified in the XML layout file in which the map is embedded using the *map:mapType* property together with a value of *none*, *normal*, *hybrid*, *satellite* or *terrain*. For example:

```

<?xml version="1.0" encoding="utf-8"?>
<fragment xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:map="http://schemas.android.com/apk/res-auto"
    android:id="@+id/map"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    map:mapType="hybrid"
    android:name="com.google.android.gms.maps.SupportMapFragment"/>

```

## 74.14 Displaying Map Controls to the User

The Google Maps Android API provides a number of controls that may be optionally displayed to the user consisting of zoom in and out buttons, a “my location” button and a compass.

Whether or not the zoom and compass controls are displayed may be controlled either programmatically or within the map element in XML layout resources. To configure the controls programmatically, a reference to the *UiSettings* object associated with the *GoogleMap* object must be obtained:

```

import com.google.android.gms.maps.UiSettings;
.
.
UiSettings mapSettings;
mapSettings = mMap.getUiSettings();

```

The zoom controls are enabled and disabled via the calls to the *setZoomControlsEnabled()* method of the *UiSettings* object. For example:

```
mapSettings.setZoomControlsEnabled(true);
```

Alternatively, the *map:uiZoomControls* property may be set within the map element of the XML resource file:

```
map:uiZoomControls="false"
```

The compass may be displayed either via a call to the *setCompassEnabled()* method of the *UiSettings* instance, or through XML resources using the *map:uiCompass* property. Note the compass icon only appears when the map camera is tilted or rotated away from the default orientation.

The “My Location” button will only appear when *MyLocation* mode is enabled as outlined earlier in this chapter. The button may be prevented from appearing even when in this mode via a call to the *setMyLocationButtonEnabled()* method of the *UiSettings* instance.

## 74.15 Handling Map Gesture Interaction

The Google Maps Android API is capable of responding to a number of different user interactions. These interactions can be used to change the area of the map displayed, the zoom level and even the angle of view (such that a 3D representation of the map area is displayed for certain cities).

### 74.15.1 Map Zooming Gestures

Support for gestures relating to zooming in and out of a map may be enabled or disabled using the *setZoomGesturesEnabled()* method of the *UiSettings* object associated with the *GoogleMap* instance. For example, the following code disables zoom gestures for our example map:

```
UiSettings mapSettings;  
mapSettings = map.getUiSettings();  
mapSettings.setZoomGesturesEnabled(false);
```

The same result can be achieved within an XML resource file by setting the *map:uiZoomGestures* property to true or false.

When enabled, zooming will occur when the user makes pinching gestures on the screen. Similarly, a double tap will zoom in while a two finger tap will zoom out. One finger zooming gestures, on the other hand, are performed by tapping twice but not releasing the second tap and then sliding the finger up and down on the screen to zoom in and out respectively.

### 74.15.2 Map Scrolling/Panning Gestures

A scrolling, or panning gesture allows the user to move around the map by dragging the map around the screen with a single finger motion. Scrolling gestures may be enabled within code via a call to the *setScrollGesturesEnabled()* method of the *UiSettings* instance:

```
UiSettings mapSettings;  
mapSettings = mMap.getUiSettings();  
mapSettings.setScrollGesturesEnabled(true);
```

Alternatively, scrolling on a map instance may be enabled in an XML resource layout file using the *map:uiScrollGestures* property.

### 74.15.3 Map Tilt Gestures

Tilt gestures allow the user to tilt the angle of projection of the map by placing two fingers on the screen and moving them up and down to adjust the tilt angle. Tilt gestures may be enabled or disabled via a call to the *setTiltGesturesEnabled()* method of the *UiSettings* instance, for example:

```
UiSettings mapSettings;  
mapSettings = mMap.getUiSettings();  
mapSettings.setTiltGesturesEnabled(true);
```

Tilt gestures may also be enabled and disabled using the *map:uiTiltGestures* property in an XML layout resource file.

### 74.15.4 Map Rotation Gestures

By placing two fingers on the screen and rotating them in a circular motion, the user may rotate the orientation of a map when map rotation gestures are enabled. This gesture support is enabled and disabled in code via a call to the *setRotateGesturesEnabled()* method of the *UiSettings* instance, for example:

```
UiSettings mapSettings;  
mapSettings = mMap.getUiSettings();  
mapSettings.setRotateGesturesEnabled(true);
```

Rotation gestures may also be enabled or disabled using the *map:uiRotateGestures* property in an XML layout resource file.

## 74.16 Creating Map Markers

Markers are used to notify the user of locations on a map and take the form of either a standard or custom icon. Markers may also include a title and optional text (referred to as a snippet) and may be configured such that they can be dragged to different locations on the map by the user. When tapped by the user an *info window* will appear displaying additional information about the marker location.

Markers are represented by instances of the `Marker` class and are added to a map via a call to the `addMarker()` method of the corresponding `GoogleMap` object. Passed through as an argument to this method is a `MarkerOptions` class instance containing the various options required for the marker such as the title and snippet text. The location of a marker is defined by specifying latitude and longitude values, also included as part of the `MarkerOptions` instance. For example, the following code adds a marker including a title, snippet and a position to a specific location on the map:

```
import com.google.android.gms.maps.model.Marker;
import com.google.android.gms.maps.model.LatLng;
import com.google.android.gms.maps.model.MarkerOptions;
.
.
.
LatLng position = new LatLng(38.8874245, -77.0200729);
Marker museum = mMap.addMarker(new MarkerOptions()
    .position(position)
    .title("Museum")
    .snippet("National Air and Space Museum"));
```

When executed, the above code will mark the location specified which, when tapped, will display an info window containing the title and snippet as shown in Figure 74-11:

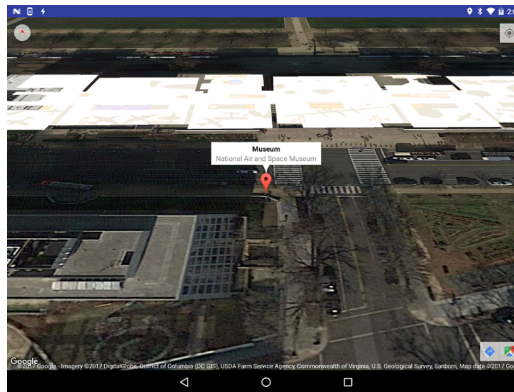


Figure 74-11

## 74.17 Controlling the Map Camera

Because Android device screens are flat and the world is a sphere, the Google Maps Android API uses the Mercator projection to represent the earth on a flat surface. The default view of the map is presented to the user as though through a *camera* suspended above the map and pointing directly down at the map. The Google Maps Android API allows the *target*, *zoom*, *bearing* and *tilt* of this camera to be changed in real-time from within the application:

- **Target** – The location of the center of the map within the device display specified in terms of longitude and

latitude.

- **Zoom** – The zoom level of the camera specified in levels. Increasing the zoom level by 1.0 doubles the width of the amount of the map displayed.
- **Tilt** – The viewing angle of the camera specified as a position on an arc spanning directly over the center of the viewable map area measured in degrees from the top of the arc (this being the nadir of the arc where the camera points directly down to the map).
- **Bearing** – The orientation of the map in degrees measured in a clockwise direction from North.

Camera changes are made by creating an instance of the `CameraUpdate` class with the appropriate settings. `CameraUpdate` instances are created by making method calls to the `CameraUpdateFactory` class. Once a `CameraUpdate` instance has been created, it is applied to the map via a call to the `moveCamera()` method of the `GoogleMap` instance. To obtain a smooth animated effect as the camera changes, the `animateCamera()` method may be called instead of `moveCamera()`.

A summary of `CameraUpdateFactory` methods is as follows:

- **`CameraUpdateFactory.zoomIn()`** – Provides a `CameraUpdate` instance zoomed in by one level.
- **`CameraUpdateFactory.zoomOut()`** – Provides a `CameraUpdate` instance zoomed out by one level.
- **`CameraUpdateFactory.zoomTo(float)`** – Generates a `CameraUpdate` instance that changes the zoom level to the specified value.
- **`CameraUpdateFactory.zoomBy(float)`** – Provides a `CameraUpdate` instance with a zoom level increased or decreased by the specified amount.
- **`CameraUpdateFactory.zoomBy(float, Point)`** – Creates a `CameraUpdate` instance that increases or decreases the zoom level by the specified value.
- **`CameraUpdateFactory.newLatLng(LatLng)`** – Creates a `CameraUpdate` instance that changes the camera's target latitude and longitude.
- **`CameraUpdateFactory.newLatLngZoom(LatLng, float)`** – Generates a `CameraUpdate` instance that changes the camera's latitude, longitude and zoom.
- **`CameraUpdateFactory.newCameraPosition(CameraPosition)`** – Returns a `CameraUpdate` instance that moves the camera to the specified position. A `CameraPosition` instance can be obtained using `CameraPosition.Builder()`.

The following code, for example, zooms in the camera by one level using animation:

```
mMap.animateCamera(CameraUpdateFactory.zoomIn());
```

The following code, on the other hand, moves the camera to a new location and adjusts the zoom level to 10 without animation:

```
private static final LatLng position =  
    new LatLng(38.8874245, -77.0200729);
```

```
mMap.moveCamera(CameraUpdateFactory.newLatLngZoom(position, 10));
```

Finally, the next code example uses `CameraPosition.Builder()` to create a `CameraPosition` object with changes to the target, zoom, bearing and tilt. This change is then applied to the camera using animation:

```
import com.google.android.gms.maps.model.CameraPosition;
```



```
import com.google.android.gms.maps.CameraUpdateFactory;
.
.
CameraPosition cameraPosition = new CameraPosition.Builder()
    .target(position)
    .zoom(50)
    .bearing(70)
    .tilt(25)
    .build();
mMap.animateCamera(CameraUpdateFactory.newCameraPosition(
    cameraPosition));
```

## 74.18 Summary

This chapter has provided an overview of the key classes and methods that make up the Google Maps Android API and outlined the steps involved in preparing both the development environment and an application project to make use of the API.



## 80. An Android Biometric Authentication Tutorial

Touch sensors are now built into many Android devices to identify the user and provide access to both the device and application functionality such as in-app payment options using fingerprint recognition. Fingerprint recognition is, of course, just one of a number of different authentication methods including passwords, PIN numbers and, more recently, facial recognition.

Although only a few Android devices currently on the market provide facial recognition, it is likely that this will become more common in the near future. In recognition of this, Google has begun to transition away from what was a fingerprint-centric approach to adding authentication to apps to a less specific approach that is referred to as *biometric authentication*. In the initial release of Android 8, these biometric features only cover fingerprint authentication but this will change in future releases and updates of the Android operating system and SDK.

This chapter provides both an overview of biometric authentication and a detailed, step by step tutorial that demonstrates a practical approach to implementing biometric authentication within an Android app project.

### 80.1 An Overview of Biometric Authentication

The key biometric authentication component is the `BiometricPrompt` class. This class performs much of the work that previously had to be performed by writing code in earlier Android versions, including displaying a standard dialog to guide the user through the authentication process, performing the authentication and reporting the results to the app. The class also handles excessive failed authentication attempts and enforces a timeout before the user can try again.

The `BiometricPrompt` class includes a companion `Builder` class that can be used to configure and create `BiometricPrompt` instances, including defining the text that is to appear within the biometric authentication dialog and the customization of the cancel button (also referred to as the *negative button*) that appears in the dialog.

The `BiometricPrompt` instance is also assigned a set of authentication callbacks that will be called to provide the app with the results of an authentication operation. A `CancellationSignal` instance is also used to allow the app to cancel the authentication while it is in process.

With these basics covered, the remainder of this chapter will implement fingerprint-based biometric authentication within an example project.

### 80.2 Creating the Biometric Authentication Project

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

Enter *BiometricDemo* into the Name field and specify *com.ebookfrenzy.biometricdemo* as the package name. Before clicking on the Finish button, change the Minimum API level setting to API 29: Android (Q) and the Language menu to Java.

## 80.3 Configuring Device Fingerprint Authentication

Fingerprint authentication is only available on devices containing a touch sensor and on which the appropriate configuration steps have been taken to secure the device and enroll at least one fingerprint. For steps on configuring an emulator session to test fingerprint authentication, refer to the chapter entitled “*Using and Configuring the Android Studio AVD Emulator*”.

To configure fingerprint authentication on a physical device begin by opening the Settings app and selecting the *Security* option. Within the Security settings screen, select the *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. If the lock screen is not already secured, follow the steps to configure either PIN, pattern or password security.

With the lock screen secured, proceed to the fingerprint detection screen and touch the sensor when prompted to do so (Figure 80-1), repeating the process to add additional fingerprints if required.

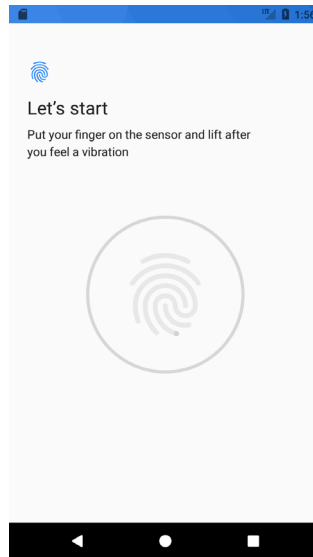


Figure 80-1

## 80.4 Adding the Biometric Permission to the Manifest File

Biometric authentication requires that the app request the *USE\_BIOMETRIC* permission within the project manifest file. Within the Android Studio Project tool window locate and edit the *app -> manifests -> AndroidManifest.xml* file to add the permission request as follows:

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

    <uses-permission
        android:name="android.permission.USE_BIOMETRIC" />

    .
    .
```

## 80.5 Designing the User Interface

In the interests of keeping the example as simple as possible, the only visual element within the user interface will be a Button view. Locate and select the *activity\_main.xml* layout resource file to load it into the Layout Editor tool.

Delete the sample TextView object, drag and drop a Button object from the *Common* category of the palette and position it in the center of the layout canvas. Using the Attributes tool window, change the text property on the button to “Authenticate” and extract the string to a resource. Finally, configure the onClick property to call a method named *authenticateUser*.

On completion of the above steps the layout should match that shown in Figure 80-2:

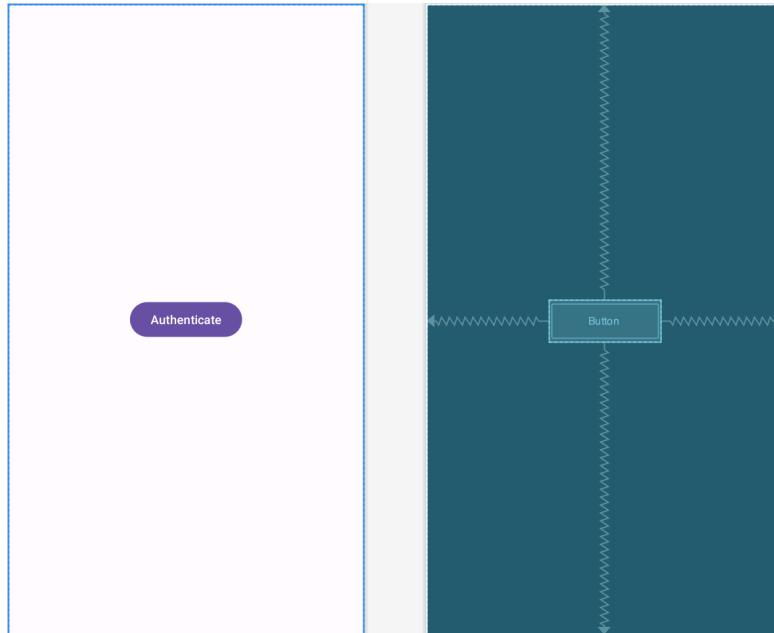


Figure 80-2

## 80.6 Adding a Toast Convenience Method

At various points throughout the code in this example the app will be designed to display information to the user via Toast messages. Rather than repeat the same Toast code multiple times, a convenience method named *notifyUser()* will be added to the main activity. This method will accept a single String value and display it to the user in the form of a Toast message. Edit the *MainActivity.java* file now and add this method as follows:

```
.
.
import android.widget.Toast;
.
.

private void notifyUser(String message) {
    Toast.makeText(this,
        message,
        Toast.LENGTH_LONG).show();
}
```

.

.

## 80.7 Checking the Security Settings

Earlier in this chapter steps were taken to configure the lock screen and register fingerprints on the device or emulator on which the app is going to be tested. It is important, however, to include defensive code in the app to make sure that these requirements have been met before attempting to seek fingerprint authentication. These steps will be performed within the *onCreate* method residing in the *MainActivity.java* file, making use of the Keyguard and PackageManager manager services. Note that code has also been added to verify that the *USE\_BIOMETRIC* permission has been configured for the app:

```
package com.ebookfrenzy.biometricdemo;

import androidx.appcompat.app.AppCompatActivity;
import androidx.core.app.ActivityCompat;

import android.widget.Toast;
import android.Manifest;
import android.content.pm.PackageManager;

import android.app.KeyguardManager;

public class MainActivity extends AppCompatActivity {

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

    private void checkBiometricSupport() {

        KeyguardManager keyguardManager =
            (KeyguardManager) getSystemService(KEYGUARD_SERVICE);

        if (!keyguardManager.isKeyguardSecure()) {
            notifyUser("Lock screen security not enabled in Settings");
        }

        if (ActivityCompat.checkSelfPermission(this,
            Manifest.permission.USE_BIOMETRIC) !=
            PackageManager.PERMISSION_GRANTED) {

            notifyUser("Fingerprint authentication permission not enabled");
        }
    }
}
```

```
.
.
}
```

The above code changes begin by using the Keyguard manager to verify that a backup screen unlocking method has been configured (in other words a PIN or other authentication method can be used as an alternative to fingerprint authentication to unlock the screen). If the lock screen is not secured the code reports the problem to the user and returns from the method.

The method then checks that the user has biometric authentication permission enabled for the app before using the package manager to verify that fingerprint authentication is available on the device.

## 80.8 Configuring the Authentication Callbacks

When the biometric prompt dialog is configured, it will need to be assigned a set of authentication callback methods that can be called to notify the app of the success or failure of the authentication process. These methods need to be wrapped in a `BiometricPrompt.AuthenticationCallback` class instance. Remaining in the *MainActivity.java* file, add a method to create and return an instance of this class with the appropriate methods implemented:

```
.
.
import android.hardware.biometrics.BiometricPrompt;
.
.

    private BiometricPrompt.AuthenticationCallback getAuthenticationCallback() {

        return new BiometricPrompt.AuthenticationCallback() {

            @Override
            public void onAuthenticationError(int errorCode,
                                             CharSequence errString) {
                notifyUser("Authentication error: " + errString);
                super.onAuthenticationError(errorCode, errString);
            }

            @Override
            public void onAuthenticationHelp(int helpCode,
                                             CharSequence helpString) {
                super.onAuthenticationHelp(helpCode, helpString);
            }

            @Override
            public void onAuthenticationFailed() {
                super.onAuthenticationFailed();
            }

            @Override
            public void onAuthenticationSucceeded(
                BiometricPrompt.AuthenticationResult result) {
```

```

        notifyUser("Authentication Succeeded");
        super.onAuthenticationSucceeded(result);
    }
};
}
.
.
}

```

## 80.9 Adding the CancellationSignal

Once initiated, the biometric authentication process is performed independently of the app. To provide the app with a way to cancel the operation, an instance of the `CancellationSignal` class is created and passed to the biometric authentication process. This `CancellationSignal` instance can then be used to cancel the process if necessary. The cancellation signal instance may be configured with a listener which will be called when the cancellation is completed. Add a new method to the activity class to configure and return a `CancellationSignal` object as follows:

```

.
.
import android.os.CancellationSignal;
.
.
    private CancellationSignal cancellationSignal;
.
.
    private CancellationSignal getCancellationSignal() {

        cancellationSignal = new CancellationSignal();
        cancellationSignal.setOnCancelListener(() ->
            notifyUser("Cancelled via signal"));
        return cancellationSignal;
    }
.
.

```

## 80.10 Starting the Biometric Prompt

All that remains is to add code to the `authenticateUser()` method to create and configure a `BiometricPrompt` instance and initiate the authentication. Add the `authenticateUser()` method as follows:

```

.
.
import android.view.View;
.
.
public void authenticateUser(View view) {
    BiometricPrompt biometricPrompt = new BiometricPrompt.Builder(this)
        .setTitle("Biometric Demo")
        .setSubtitle("Authentication is required to continue")

```



```

        .setDescription(
            "This app uses biometric authentication to protect your data.")
        .setNegativeButton("Cancel", this.getMainExecutor(),
            (dialogInterface, i) ->
                notifyUser("Authentication cancelled"))
        .build();

    biometricPrompt.authenticate(getCancellationSignal(), getMainExecutor(),
        getAuthenticationCallback());
}

```

The `BiometricPrompt.Builder` class is used to create a new `BiometricPrompt` instance configured with title, subtitle and description text to appear in the prompt dialog. The negative button is configured to display text which reads “Cancel” and a listener configured to display a message when this button is clicked. Finally, the `authenticate()` method of the `BiometricPrompt` instance is called and passed the `AuthenticationCallback` and `CancellationSignal` instances. The `BiometricPrompt` also needs to know which thread to perform the authentication on. This is defined by passing through an `Executor` object configured for the required thread. In this case, the `getMainExecutor()` method is used to pass a main `Executor` object to the `BiometricPrompt` instance so that the authentication process takes place on the app’s main thread.

## 80.11 Testing the Project

With the project now complete, run the app on a physical Android device or emulator session and click on the `Authenticate` button to display the `BiometricPrompt` dialog as shown in Figure 80-3:

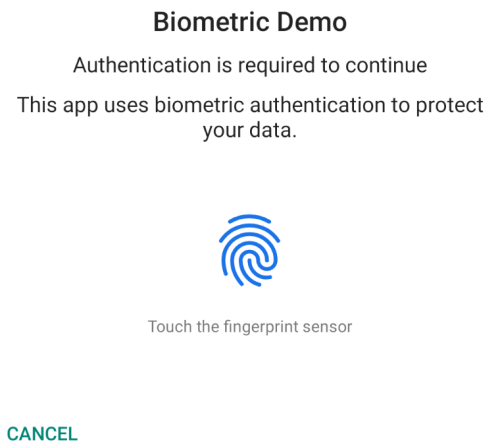


Figure 80-3

Once running, either touch the fingerprint sensor or use the extended controls panel within the emulator to simulate a fingerprint touch as outlined in the chapter entitled “*Using and Configuring the Android Studio AVD Emulator*”. Assuming a registered fingerprint is detected the prompt dialog will return to the main activity where the toast message from the successful authentication callback method will appear.

Click the `Authenticate` button once again, this time using an unregistered fingerprint to attempt the authentication. This time the biometric prompt dialog will indicate that the fingerprint was not recognized:

## Biometric Demo

Authentication is required to continue

This app uses biometric authentication to protect  
your data.



Not recognized

Figure 80-4

Verify that the error handling callback is working by clicking on the activity outside of the biometric prompt dialog. The prompt dialog will disappear and the toast message will appear with the following message:

Authentication error: Fingerprint operation cancelled by user.

Check that canceling the prompt dialog using the Cancel button triggers the “Authentication Canceled” toast message. Finally, attempt to authenticate multiple times using an unregistered fingerprint and note that after a number of attempts the prompt dialog indicates that too many failures have occurred and that future attempts cannot be made until later.

## 80.12 Summary

This chapter has outlined how to integrate biometric authentication into an Android app project. This involves the use of the BiometricPrompt class which, once configured with appropriate message text and callbacks, automatically handles most of the authentication process.

## Index

### Symbols

<application> 428  
 <fragment> 239  
 <fragment> element 239  
 <receiver> 462  
 <service> 428, 472, 479  
 Code Reformatting 73  
 .well-known folder 435, 458, 674

### A

AbsoluteLayout 120  
 ACCESS\_COARSE\_LOCATION permission 496  
 ACCESS\_FINE\_LOCATION permission 496  
 acknowledgePurchase() method 713  
 ACTION\_CREATE\_DOCUMENT 588  
 ACTION\_CREATE\_INTENT 588  
 ACTION\_DOWN 214  
 ACTION\_MOVE 214  
 ACTION\_OPEN\_DOCUMENT intent 580  
 ACTION\_POINTER\_DOWN 214  
 ACTION\_POINTER\_UP 214  
 ACTION\_UP 214  
 ACTION\_VIEW 453  
 Active / Running state 94  
 Activity 81, 97  
   adding views in Java code 195  
   class 97  
   creation 14  
   Entire Lifetime 101  
   Foreground Lifetime 101  
   lifecycle methods 100  
   lifecycles 91  
   returning data from 432  
   state change example 105  
   state changes 97

states 94  
   Visible Lifetime 101  
 ActivityCompat class 501  
 Activity Lifecycle 93  
 Activity Manager 80  
 ActivityResultLauncher 433  
 Activity Stack 93  
 Actual screen pixels 186  
 adb  
   command-line tool 57  
   connection testing 63  
   device pairing 61  
   enabling on Android devices 57  
   Linux configuration 60  
   list devices 57  
   macOS configuration 58  
   overview 57  
   restart server 58  
   testing connection 63  
   WiFi debugging 61  
   Windows configuration 59  
   Wireless debugging 61  
   Wireless pairing 61  
 addCategory() method 461  
 addMarker() method 637  
 addView() method 190  
 ADD\_VOICEMAIL permission 496  
 android  
   exported 429  
   gestureColor 233  
   layout\_behavior property 411  
   onClick 241  
   process 429, 479  
   uncertainGestureColor 233  
 Android  
   Activity 81  
   architecture 77  
   events 207  
   intents 82

## Index

- onClick Resource 207
- runtime 78
- SDK Packages 6
- android.app 78
- Android Architecture Components 257
- android.content 78
- android.content.Intent 431
- android.database 78
- Android Debug Bridge. *See* ADB
- Android Design Support Library 367
- Android Development
  - System Requirements 3
- Android Devices
  - designing for different 119
- android.graphics 78
- android.hardware 78
- android.intent.action 467
- android.intent.action.BOOT\_COMPLETED 430
- android.intent.action.MAIN 453
- android.intent.category.LAUNCHER 453
- Android Libraries 78
- android.media 79
- Android Monitor tool window 32
- Android Native Development Kit 79
- android.net 79
- android.opengl 79
- android.os 79
- android.permission.RECORD\_AUDIO 615
- android.print 79
- Android Project
  - create new 13
- android.provider 79
- Android SDK Location
  - identifying 9
- Android SDK Manager 8, 10
- Android SDK Packages
  - version requirements 8
- Android SDK Tools
  - command-line access 9
  - Linux 11
  - macOS 11
  - Windows 7 10
  - Windows 8 10
- Android Software Stack 77
- Android Storage Access Framework 580
- Android Studio
  - changing theme 54
  - downloading 3
  - Editor Window 48
  - installation 4
  - Linux installation 5
  - macOS installation 4
  - Main Window 48
  - Menu Bar 48
  - Navigation Bar 48
  - Project tool window 49
  - setup wizard 5
  - Status Bar 49
  - Toolbar 48
  - Tool window bars 50
  - tool windows 49
  - updating 12
  - Welcome Screen 47
  - Windows installation 4
- android.text 79
- android.util 79
- android.view 79
- android.view.View 122
- android.view.ViewGroup 119, 122
- Android Virtual Device. *See* AVD
  - overview 27
- Android Virtual Device Manager 27
- android.webkit 79
- android.widget 79
- AndroidX libraries 768
- API Key 627
- APK analyzer 706
- APK file 700
  - split 730
- APK File
  - analyzing 706
- APK Signing 768
- APK Wizard dialog 698
- App Architecture

- modern 257
- AppBar
  - anatomy of 409
- appbar\_scrolling\_view\_behavior 411
- App Bundles 695
  - creating 700
  - overview 695
  - revisions 705
  - uploading 702
- AppCompatActivity class 98
- App Inspector 51
- Application
  - stopping 32
- Application Context 83
- Application Framework 79
- Application Manifest 83
- Application Resources 83
- App Link
  - Adding Intent Filter 682
  - Assistant 677
  - Digital Asset Links file 674, 435
  - Intent Filter Handling 682
  - Intent Filters 673
  - Intent Handling 674
  - Testing 686
  - tutorial 677
  - URL Mapping 679
- App Link Assistant 677
- App Links 673
  - auto verification 435
  - autoVerify 435
  - manually enabling 437
  - overview 673
- Apply Changes 203
  - Apply Changes and Restart Activity 203
  - Apply Code Changes 203
  - fallback settings 205
  - options 203
  - Run App 203
  - tutorial 205
- applyToActivitiesIfAvailable() method 763
- Architecture Components 257

- ART 78
- assetlinks.json , 674, 435
- Attribute Keyframes 334
- Audio
  - supported formats 613
- Audio Playback 613
- Audio Recording 613
- Autoconnect Mode 152
- Automatic Link Verification 435, 457
- autoVerify 435, 682

- AVD
  - cold boot 42
  - command-line creation 27
  - creation 27
  - device frame 35
  - Display mode 44
  - launch in tool window 35
  - overview 27
  - quickboot 42
  - Resizable 44
  - running an application 30
  - Snapshots 41
  - standalone 33
  - starting 29
  - Startup size and orientation 30

## B

- Background Process 92
- Barriers 146
  - adding 164
  - constrained views 146
- Base APK file 730
- Baseline Alignment 145
- beginTransaction() method 240
- BillingClient 714
  - acknowledgePurchase() method 713
  - consumeAsync() method 713
  - getPurchaseState() method 713
  - initialization 710, 720
  - launchBillingFlow() method 712
  - queryProductDetailsAsync() method 712
  - queryPurchasesAsync() method 714

## Index

- startConnection() method 711
- BillingResult 727
  - getDebugMessage() 727
- Binding Expressions 281
  - one-way 281
  - two-way 282
- BIND\_JOB\_SERVICE permission 429
- bindService() method 427, 469, 474
- Biometric Authentication 687
  - callbacks 691
  - overview 687
  - tutorial 687
- Biometric Prompt 692
- BitmapFactory 581
- black activity 14
- Blank template 123
- Blueprint view 151
- BODY\_SENSORS permission 496
- Bookmarks 51
- Bound Service 427, 469
  - adding to a project 470
  - Implementing the Binder 470
  - Interaction options 469
- BoundService class 471
- Broadcast Intent 461
  - example 464
  - overview 82, 461
  - sending 464
  - Sticky 463
- Broadcast Receiver 461
  - adding to manifest file 466
  - creation 465
  - overview 82, 462
- BroadcastReceiver class 462
- BroadcastReceiver superclass 465
- BufferedReader object 591
- Build tool window 51
- Build Variants 51, 768
  - tool window 51
- Bundle class 114
- Bundled Notifications 515

## C

- Calendar permissions 496
- CALL\_PHONE permission 496
- CAMERA permission 496
- Camera permissions 496
- CameraUpdateFactory class
  - methods 638
- CancellationSignal 692
- Canvas class 668
- CardView
  - example 391
  - layout file 389
  - responding to selection of 397
- CardView class 389
- CATEGORY\_OPENABLE 580
- C/C++ Libraries 79
- Chain bias 172
- chain head 144
- chains 144
- Chains
  - creation of 169
- Chain style
  - changing 171
- chain styles 144
- CharSequence 115
- CheckBox 119
- checkSelfPermission() method 500
- Circle class 623
- Code completion 68
- Code Editor
  - basics 65
  - Code completion 68
  - Code Generation 70
  - Code Reformatting 73
  - Document Tabs 65
  - Editing area 66
  - Gutter Area 66
  - Live Templates 74
  - Splitting 67
  - Statement Completion 69
  - Status Bar 67
- Code Generation 70

- code samples
  - download 1
- cold boot 42
- CollapsingToolbarLayout
  - example 413
  - introduction 412
  - parallax mode 412
  - pin mode 412
  - setting scrim color 415
  - setting title 415
  - with image 412
- Color class 669
- COLOR\_MODE\_COLOR 644, 664
- COLOR\_MODE\_MONOCHROME 644, 664
- com.android.application 733
- com.android.dynamic-feature 733
- Common Gestures 221
  - detection 221
- Component tree 17
- Configuration APK file 730
- Constraint Bias 143
  - adjusting 156
- ConstraintLayout
  - advantages of 149
  - Availability 150
  - Barriers 146
  - Baseline Alignment 145
  - chain bias 172
  - chain head 144
  - chains 144
  - chain styles 144
  - Constraint Bias 143
  - Constraints 141
  - conversion to 168
  - convert to MotionLayout 341
  - deleting constraints 156
  - guidelines 162
  - Guidelines 146
  - manual constraint manipulation 153
  - Margins 142, 157
  - Opposing Constraints 142, 158
  - overview of 141
  - Packed chain 145, 172
  - ratios 149, 173
  - Spread chain 144
  - Spread inside 171
  - Spread inside chain 144
  - tutorial 177
  - using in Android Studio 151
  - Weighted chain 144, 172
  - Widget Dimensions 145, 160
  - Widget Group Alignment 167
- ConstraintLayout chains
  - creation of 169
  - in layout editor 169
- ConstraintLayout Chain style
  - changing 171
- Constraints
  - deleting 156
- ConstraintSet
  - addToHorizontalChain() method 192
  - addToVerticalChain() method 192
  - alignment constraints 191
  - apply to layout 190
  - applyTo() method 190
  - centerHorizontally() method 191
  - centerVertically() method 191
  - chains 191
  - clear() method 192
  - clone() method 191
  - connect() method 190
  - connect to parent 190
  - constraint bias 191
  - copying constraints 191
  - create 190
  - create connection 190
  - createHorizontalChain() method 191
  - createVerticalChain() method 191
  - guidelines 192
  - removeFromHorizontalChain() method 192
  - removeFromVerticalChain() method 192
  - removing constraints 192
  - rotation 193
  - scaling 192

## Index

- setGuidelineBegin() method 192
- setGuidelineEnd() method 192
- setGuidelinePercent() method 192
- setHorizontalBias() method 191
- setRotationX() method 193
- setRotationY() method 193
- setScaleX() method 192
- setScaleY() method 192
- setTransformPivot() method 193
- setTransformPivotX() method 193
- setTransformPivotY() method 193
- setVerticalBias() method 191
- sizing constraints 191
- tutorial 195
- view IDs 197
- ConstraintSet class 189, 190
- ConstraintSet.PARENT\_ID 190
- Constraint Sets 190
- ConstraintSets
  - configuring 330
- consumeAsync() method 713
- ConsumeParams 725
- ConsumeResponseListener 713
- Contacts permissions 496
- container view 119
- Content Provider 80
  - overview 83
- Context class 83
- CoordinatorLayout 120, 409, 411
- createPrintDocumentAdapter() method 659
- Custom Attribute 331
- Custom Document Printing 647, 659
- Custom Gesture
  - recognition 227
- Custom Print Adapter
  - implementation 661
- Custom Print Adapters 659
- Custom Theme
  - building 757
- Cycle Editor 359
- Cycle Keyframe 339
- Cycle Keyframes

overview 355

## D

- dangerous permissions 495
  - list of 496
- Dark Theme 32
  - enable on device 32
- Data Access Object (DAO) 546
- Data Access Objects (DAO) 550
- Database Inspector 553, 576
  - live updates 577
  - SQL query 577
- Database Rows 540
- Database Schema 539
- Database Tables 539
- Data binding
  - binding expressions 281
- Data Binding 260
  - binding classes 280
  - enabling 286
  - event and listener binding 282
  - key components 277
  - overview 277
  - tutorial 285
  - with LiveData 260
- DDMS 32
- Debugging
  - enabling on device 57
- debug.keystore file 435, 457
- DefaultLifecycleObserver 300, 303
- deltaRelative 335
- Density-independent pixels 185
- Density Independent Pixels
  - converting to pixels 200
- Device Definition
  - custom 137
- Device File Explorer 51
- device frame 35
- Device Manager 51
- Device Mirroring 63
  - enabling 63
- device pairing 61



- Digital Asset Links file 435, 674, 435
- Direct Reply Input 526
- Direct Reply Notification 519
- document provider 579
- dp 185
- Dynamic Colors
  - applyToActivitiesIfAvailable() method 763
  - enabling 763
  - enabling in Android 763
- Dynamic Delivery 732
- Dynamic Feature APK 730
- Dynamic Feature Module
  - architecture 729
  - overview 729
  - removal 753
  - tutorial 739
- Dynamic Feature Modules
  - deferred installation 735
  - handling of large 737
- Dynamic Feature Support
  - adding to project 739
- Dynamic State 99
  - saving 113

## E

- Empty Process 93
- Empty template 123
- Emulator 51
  - battery 40
  - cellular configuration 40
  - configuring fingerprints 42
  - directional pad 40
  - extended control options 39
  - Extended controls 39
  - fingerprint 40
  - location configuration 40
  - phone settings 40
  - Resizable 44
  - resize 39
  - rotate 38
  - Screen Record 41
  - Snapshots 41

- starting 29
- take screenshot 38
- toolbar 37
- toolbar options 37
- tool window mode 43
- Virtual Sensors 41
- zoom 38
- enablePendingPurchases() method 713
- enabling ADB support 57
- ettings.gradle file 768
- Event Handling 207
  - example 208
- Event Listener 209
- Event Listeners 208
- Event Log 51
- Events
  - consuming 211
- explicit
  - intent 82
- explicit intent 431
- Explicit Intent 431
- Extended Control
  - options 39

## F

- Favorites
  - tool window 51
- Files
  - switching between 66
- findPointerIndex() method 214
- findViewById() 85
- Fingerprint
  - emulation 42
- Fingerprint authentication
  - device configuration 688
  - permission 688
  - steps to implement 687
- Fingerprint Authentication
  - overview 687
  - tutorial 687
- FLAG\_INCLUDE\_STOPPED\_PACKAGES 461
- flexible space area 409

## Index

floating action button 14, 124, 367

    changing appearance of 371

    margins 368

    overview of 367

    removing 125

    sizes 368

Foldable Devices 102

    multi-resume 102

Foldable Emulator 532

Foldables 531

Foreground Process 92

Forward-geocoding 630

Fragment

    creation 237

    event handling 241

    XML file 237, 238

FragmentManager class 98

Fragment Communication 242

Fragments 237

    adding in code 240

    duplicating 378

    example 245

    overview 237

FragmentManager class 381

FrameLayout 120

## G

Geocoder class 629

Geocoder object 631

Geocoding 629

Gesture Builder Application 227

    building and running 228

Gesture Detector class 221

GestureDetectorCompat 224

    instance creation 224

GestureDetectorCompat class 221

GestureDetector.OnDoubleTapListener 221, 222

GestureDetector.OnGestureListener 222

GestureLibrary 227

GestureLibrary class 227

GestureOverlayView 227

    configuring color 233

    configuring multiple strokes 233

GestureOverlayView class 227

GesturePerformedListener 227

Gestures

    interception of 233

Gestures File

    creation 228

    extract from SD card 229

    loading into application 230

GET\_ACCOUNTS permission 496

getAction() method 467

getDebugMessage() 727

getFromLocation() method 631

getId() method 190

getIntent() method 432

getPointerCount() method 214

getPointerId() method 214

getPurchaseState() method 713

getService() method 474

GNU/Linux 78

Google Cloud

    billing account 624

    Console 624

    new project 625

Google Cloud Print 642

Google Drive 580

    printing to 642

GoogleMap 623

    map types 634

GoogleMap.MAP\_TYPE\_HYBRID 634

GoogleMap.MAP\_TYPE\_NONE 634

GoogleMap.MAP\_TYPE\_NORMAL 634

GoogleMap.MAP\_TYPE\_SATELLITE 634

GoogleMap.MAP\_TYPE\_TERRAIN 634

Google Maps Android API 623

    Controlling the Map Camera 637

    displaying controls 635

    gesture handling 635

    Map Markers 637

    overview 623

Google Maps SDK 623

    API Key 627

- Credentials 627
- enabling 626
- Maps SDK for Android 627
- Google Play Billing Library 709
- Google Play Console 718
  - Creating an in-app product 718
  - License Testers 719
- Google Play Developer Console 696
- Go to Line:Column 67
- Gradle
  - APK signing settings 772
  - Build Variants 768
  - command line tasks 773
  - dependencies 767
  - Manifest Entries 768
  - overview 767
  - sensible defaults 767
  - tool window 51
- Gradle Build File
  - top level 769
- Gradle Build Files
  - module level 770
- gradle.properties file 768
- GridLayout 120
- LayoutManager 387

## H

- Handler class 478
- HP Print Services Plugin 641
- HTML printing 645
- HTML Printing
  - example 649

## I

- IBinder 427, 471
- IBinder object 469, 477, 479
- Image Printing 644
- implicit
  - intent 82
- implicit intent 431
- Implicit Intent 433
- Implicit Intents

- example 449
- in 185
- INAPP 714
- In-App Products 709
- In-App Purchasing 717
  - acknowledgePurchase() method 713
  - BillingClient 710
  - BillingResult 727
  - consumeAsync() method 713
  - ConsumeParams 725
  - ConsumeResponseListener 713
  - Consuming purchases 724
  - enablePendingPurchases() method 713
  - getPurchaseState() method 713
  - Google Play Billing Library 709
  - launchBillingFlow() method 712
- Libraries 717
  - newBuilder() method 710
- onBillingServiceDisconnected() callback 722
- onBillingServiceDisconnected() method 711
- onBillingSetupFinished() listener 721
- onProductDetailsResponse() callback 722
- Overview 709
- ProductDetail 712
- ProductDetails 723
- products 709
- ProductType 714
- ProductType.INAPP 714
- ProductType.SUBS 714
- Purchase Flow 723
- PurchaseResponseListener 714
- PurchasesUpdatedListener 713
- PurchaseUpdatedListener 723
- purchase updates 723
- queryProductDetailsAsync() 722
- queryProductDetailsAsync() method 712
- queryPurchasesAsync() 725
- queryPurchasesAsync() method 714
- runOnUiThread() 723
- startConnection() method 711
- subscriptions 709
- tutorial 717

## Index

In-Memory Database 553  
Instant Dynamic Feature Module 730  
Intent 82

- explicit 82
- implicit 82

Intent Availability

- checking for 438

Intent.CATEGORY\_OPENABLE 588  
intent filters 431  
Intent Filters 434

- App Link 673

intent resolution 434  
Intents 431

- ActivityResultLauncher 433
- overview 431
- registerForActivityResult() 446

Intent Service 427  
IntentService class 427, 430  
Intent URL 452

## J

Java Native Interface 79  
Jetpack 257

- overview 257

JobIntentService 427

- BIND\_JOB\_SERVICE permission 429
- onHandleWork() method 427

## K

KeyAttribute 334  
Keyboard Shortcuts 52  
KeyCycle 339, 355

- Cycle Editor 359
- tutorial 355

Keyframe 347  
Keyframes 334  
KeyFrameSet 364  
KeyPosition 335

- deltaRelative 335
- parentRelative 335
- pathRelative 336

Keystore File

creation 698  
KeyTimeCycle 339, 355  
keytool 435  
KeyTrigger 338  
Killed state 94

## L

launchBillingFlow() method 712  
layout\_collapseMode

- parallax 414
- pin 414

layout\_constraintDimentionRatio 174  
layout\_constraintHorizontal\_bias 172  
layout\_constraintVertical\_bias 172  
layout editor

- ConstraintLayout chains 169

Layout Editor 16, 177

- Autoconnect Mode 152
- code mode 130
- Component Tree 128
- design mode 127
- device screen 128
- example project 177
- Inference Mode 153
- palette 128
- properties panel 128
- Sample Data 136
- Setting Properties 131
- toolbar 128
- user interface design 177
- view conversion 135

Layout Editor Tool

- changing orientation 16
- overview 127

Layout Inspector 52  
Layout Managers 119  
LayoutResultCallback object 665  
Layouts 119  
layout\_scrollFlags

- enterAlwaysCollapsed mode 411
- enterAlways mode 411
- exitUntilCollapsed mode 411

- scroll mode 411
- Layout Validation 138
- libc 79
- License Testers 719
- Lifecycle
  - awareness 299
  - components 260
  - owners 299
  - states and events 301
  - tutorial 303
- Lifecycle-Aware Components 299
- Lifecycle Methods 100
- Lifecycle Observer 303
  - creating a 303
- Lifecycle Owner
  - creating a 305
- Lifecycles
  - modern 260
- LinearLayout 120
- LinearLayoutManager 387
- LinearLayoutManager layout 396
- Linux Kernel 78
- list devices 57
- LiveData 258, 271
  - adding to ViewModel 271
  - observer 273
  - tutorial 271
- Live Templates 74
- Local Bound Service 469
  - example 469
- Location Manager 80
- Location permission 496
- Logcat
  - tool window 52
- LogCat
  - enabling 109

## M

- MANAGE\_EXTERNAL\_STORAGE 497
  - adb enabling 497
  - testing 497
- Manifest File

- permissions 453
- Maps 623
- MapView 623
  - adding to a layout 631
- Marker class 623
- Master/Detail Flow
  - creation 418
  - two pane mode 417
- match\_parent properties 185
- Material design 367
- Material Design 2 755
- Material Design 2 Theming 755
- Material Design 3 755
- Material Theme Builder 757
- Material You 755
- MediaController
  - adding to VideoView instance 597
- MediaController class 594
  - methods 594
- MediaPlayer class 613
  - methods 613
- MediaRecorder class 613
  - methods 614
  - recording audio 614
- Memory Indicator 67
- Messenger object 479
- Microphone
  - checking for availability 616
- Microphone permissions 496
- mm 185
- MotionEvent 213, 214, 235
  - getActionMasked() 214
- MotionLayout 329
  - arc motion 334
  - Attribute Keyframes 334
  - ConstraintSets 330
  - Custom Attribute 350
  - Custom Attributes 331
  - Cycle Editor 359
  - Cycle Keyframes 339
  - Editor 341
  - KeyAttribute 334

## Index

- KeyCycle 355
- Keyframes 334
- KeyFrameSet 364
- KeyPosition 335
- KeyTimeCycle 355
- KeyTrigger 338
- OnClick 333, 346
- OnSwipe 333
- overview 329
- Position Keyframes 335
- previewing animation 345
- starting animation 332
- Trigger Keyframe 338
- Tutorial 341
- MotionScene
  - ConstraintSets 330
  - Custom Attributes 331
  - file 330
  - overview 329
  - transition 330
- moveCamera() method 638
- multiple devices
  - testing app on 31
- Multiple Touches
  - handling 214
- multi-resume 102
- Multi-Touch
  - example 214
- Multi-touch Event Handling 213
- Multi-Window
  - attributes 535
- Multi-Window Mode
  - detecting 536
  - entering 533
  - launching activity into 537
- Multi-Window Notifications 536
- multi-window support 102
- Multi-Window Support
  - enabling 534
- My Location Layer 624

## N

- Navigation 309
  - adding destinations 318
  - overview 309
  - pass data with safeargs 325
  - passing arguments 314
  - safeargs 314
  - stack 309
  - tutorial 315
- Navigation Action
  - triggering 313
- Navigation Architecture Component 309
- Navigation Component
  - tutorial 315
- Navigation Controller
  - accessing 313
- Navigation Graph 312, 316
  - adding actions 321
  - creating a 316
- Navigation Host 310
  - declaring 317
- newBuilder() method 710
- normal permissions 495
- Notification
  - adding actions 514
  - direct reply 519
  - Direct Reply Input 526
  - issuing a basic 510
  - launch activity from a 512
  - PendingIntent 522
  - Reply Action 524
  - updating direct reply 527
- Notifications 503
  - bundled 515
  - overview 503
- Notifications Manager 80

## O

- Observer
  - implementing a LiveData 273
- onAttach() method 242
- onBillingServiceDisconnected() callback 722
- onBillingServiceDisconnected() method 711

- onBillingSetupFinished() listener 721
- onBind() method 428, 469, 477
- onBindViewHolder() method 395
- OnClick 333
- onClickListener 208, 209, 212
- onClick() method 207
- onCreateContextMenuListener 208
- onCreate() method 92, 100, 428
- onCreateView() method 100
- on-demand modules 729
- onDestroy() method 100, 428
- onDoubleTap() method 221
- onDown() method 221
- onFling() method 221
- onFocusChangeListener 208
- OnFragmentInteractionListener
  - implementation 323
- onGesturePerformed() method 227
- onHandleWork() method 427, 428
- onKeyListener 208
- onLayoutFailed() method 665
- onLayoutFinished() method 665
- onLongClickListener 208, 211
- onLongClick() method 211
- onLongPress() method 221
- onMapReady() method 633
- onPageFinished() callback 650
- onPause() method 100
- onProductDetailsResponse() callback 722
- onReceive() method 92, 462, 463, 465
- onRequestPermissionsResult() method 499, 620, 508, 520
- onRestart() method 100
- onRestoreInstanceState() method 101
- onResume() method 92, 100
- onSaveInstanceState() method 101
- onScaleBegin() method 233
- onScaleEnd() method 233
- onScale() method 233
- onScroll() method 221
- OnSeekBarChangeListener 252
- onServiceConnected() method 469, 473, 480
- onServiceDisconnected() method 469, 473, 480

- onShowPress() method 221
- onSingleTapUp() method 221
- onStartCommand() method 428
- onStart() method 100
- onStop() method 100
- onTouchEvent() method 221, 233
- onTouchListener 208, 213
- onTouch() method 213
- onViewCreated() method 100
- onViewStatusRestored() method 100
- openFileDescriptor() method 580
- OpenJDK 3

## P

- Package Explorer 15
- Package Manager 80
- PackageManager class 616
- PackageManager.FEATURE\_MICROPHONE 616
- PackageManager.PERMISSION\_DENIED 497
- PackageManager.PERMISSION\_GRANTED 497
- Package Name 14
- Packed chain 145, 172
- PageRange 666, 667
- Paint class 669
- parentRelative 335
- parent view 121
- pathRelative 336
- Paused state 94
- PdfDocument 647
- PdfDocument.Page 659, 666
- PendingIntent class 522
- Permission
  - checking for 497
- permissions
  - dangerous 495
  - normal 495
- Persistent State 99
- Phone permissions 496
- picker 579
- Pinch Gesture
  - detection 233
  - example 234

## Index

- Pinch Gesture Recognition 227
- Play Core Library 735, 739
- Polygon class 623
- Polyline class 623
- Position Keyframes 335
- POST\_NOTIFICATIONS permission 496, 520
- PrintAttributes 664
- PrintDocumentAdapter 647, 659
- PrintDocumentInfo 664
- Printing
  - color 644
  - monochrome 644
- Printing framework
  - architecture 641
- Printing Framework 641
- Print Job
  - starting 670
- Print Manager 641
- PrintManager service 651
- Problems
  - tool window 52
- PROCESS\_OUTGOING\_CALLS permission 496
- Process States 91
- ProductDetail 712
- ProductDetails 723
- ProductType 714
- Profiler
  - tool window 52
- ProgressBar 119
- proguard-rules.pro file 772
- ProGuard Support 768
- Project
  - tool window 52
- Project Name 14
- Project tool window 15, 52
- pt 185
- PurchaseResponseListener 714
- PurchasesUpdatedListener 713
- PurchaseUpdatedListener 723
- putExtra() method 431, 461
- px 186

## Q

- queryProductDetailsAsync() 722
- queryProductDetailsAsync() method 712
- queryPurchaseHistoryAsync() method 714
- queryPurchasesAsync() 725
- queryPurchasesAsync() method 714
- quickboot snapshot 42
- Quick Documentation 72

## R

- RadioButton 119
- ratios 173
- READ\_CALENDAR permission 496
- READ\_CALL\_LOG permission 496
- READ\_CONTACTS permission 496
- READ\_EXTERNAL\_STORAGE permission 497
- READ\_PHONE\_STATE permission 496
- READ\_SMS permission 496
- RECEIVE\_MMS permission 496
- RECEIVE\_SMS permission 496
- RECEIVE\_WAP\_PUSH permission 496
- Recent Files Navigation 53
- RECORD\_AUDIO permission 496
- Recording Audio
  - permission 615
- RecyclerView 387
  - adding to layout file 388
  - example 391
  - LayoutManager 387
    - initializing 396
    - LinearLayoutManager 387
    - StaggeredLayoutManager 387
- RecyclerView Adapter
  - creation of 394
- RecyclerView.Adapter 388, 394
  - getItemCount() method 388
  - onBindViewHolder() method 388
  - onCreateViewHolder() method 388
- RecyclerView.ViewHolder
  - getAdapterPosition() method 398
- registerForActivityResult() method 433, 446
- registerReceiver() method 463



- RelativeLayout 120
- release mode 695
- releasePersistableUriPermission() method 583
- Release Preparation 695
- Remote Bound Service 477
  - client communication 477
  - implementation 478
  - manifest file declaration 479
- RemoteInput.Builder() method 522
- RemoteInput Object 522
- Remote Service
  - launching and binding 480
  - sending a message 481
- Repository
  - tutorial 563
- Repository Modules 260
- requestPermissions() method 499
- Resizable Emulator 44
- Resource
  - string creation 19
- Resource File 21
- Resource Management 91
- Resource Manager 52, 80
- result receiver 463
- Reverse-geocoding 630
- Reverse Geocoding 629
- Room
  - Data Access Object (DAO) 546
  - entities 546, 547
  - In-Memory Database 553
  - Repository 546
- Room Database 546
  - tutorial 563
- Room Database Persistence 545
- Room Persistence Library 543, 545
- root element 119
- root view 121
- Run
  - tool window 52
- Running Devices
  - tool window 63
- runOnUiThread() 723

- Runtime Permission Requests 495

## S

- safeargs 314, 325
- Sample Data 136, 401
  - tutorial 401
- Saved State 259, 293
  - library dependencies 295
- SavedStateHandle 294, 295
  - contains() method 295
  - keys() method 295
  - remove() method 295
- Saved State module 293
- SavedStateViewModelFactory 294
- ScaleGestureDetector class 233
- Scale-independent 185
- SDK Packages 6
- Secure Sockets Layer (SSL) 79
- SeekBar 245
- sendBroadcast() method 461, 463
- sendOrderedBroadcast() method 461, 463
- SEND\_SMS permission 496
- sendStickyBroadcast() method 461
- Sensor permissions 496
- Service
  - anatomy 428
  - launch at system start 430
  - manifest file entry 428
  - overview 82
  - run in separate process 429
- ServiceConnection class 480
- Service Process 92
- Service Restart Options 428
- setAudioEncoder() method 614
- setAudioSource() method 614
- setBackgroundColor() 190
- setCompassEnabled() method 635
- setContentView() method 189, 195
- setId() method 190
- setMyLocationButtonEnabled() method 635
- setOnClickListener() method 207, 209
- setOnDoubleTapListener() method 221, 224

## Index

setOutputFile() method 614  
setOutputFormat() method 614  
setResult() method 433  
setRotateGesturesEnabled() method 636  
setScrollGesturesEnabled() method 636  
setText() method 116  
setTiltGesturesEnabled() method 636  
setTransition() 339  
setVideoSource() method 614  
setZoomControlsEnabled() method 635, 636  
SHA-256 certificate fingerprint 435  
shouldOverrideUrlLoading() method 650  
shouldShowRequestPermissionRationale() method 501  
SimpleOnScaleGestureListener 233  
SimpleOnScaleGestureListener class 235  
SMS permissions 496  
Snackbar 367, 368, 369  
    overview of 368  
Snapshots  
    emulator 41  
sp 185  
Space class 120  
split APK files 730  
SplitCompatApplication 734  
SplitInstallManager 735  
Spread chain 144  
Spread inside 171  
Spread inside chain 144  
SQL 540  
SQLite 539  
    AVD command-line use 541  
    Columns and Data Types 539  
    overview 540  
    Primary keys 540  
StaggeredGridLayoutManager 387  
startActivity() method 431  
startConnection() method 711  
startForeground() method 92  
START\_NOT\_STICKY 428  
START\_REDELIVER\_INTENT 428  
START\_STICKY 428  
State

    restoring 116  
State Change  
    handling 95  
Statement Completion 69  
status bar 409  
Status Bar Widgets 67  
    Memory Indicator 67  
Sticky Broadcast Intents 463  
Stopped state 94  
Storage Access Framework 579  
    ACTION\_CREATE\_DOCUMENT 580  
    ACTION\_OPEN\_DOCUMENT 580  
    deleting a file 583  
    example 585  
    file creation 588  
    file filtering 580  
    file reading 581  
    file writing 582  
    intents 580  
    MIME Types 581  
    Persistent Access 583  
    picker 579  
Storage permissions 497  
StringBuilder object 591  
strings.xml file 23  
Structure  
    tool window 52  
Structured Query Language 540  
Structure tool window 52  
SUBS 714  
subscriptions 709  
SupportMapFragment class 623  
Switcher 53  
System Broadcasts 467  
system requirements 3

## T

tab bar 409  
TabLayout 375  
    adding to layout 379  
    app  
        tabGravity property 384

- tabMode property 384
- example 376
- fixed mode 383
- getItemCount() method 375
- overview 375
- scrollable mode 384
- TableLayout 120, 555
- TableRow 555
- Telephony Manager 80
- Templates
  - blank vs. empty 123
- Terminal
  - tool window 52
- Theme
  - building a custom 757
- Theming 755
  - Material Theme Builder 757
  - tutorial 759
- Time Cycle Keyframes 339
- TODO
  - tool window 52
- toolbar 409
- ToolBarListener 242
- tools
  - layout 239
- Tool window bars 50
- Tool windows 49
- Touch Actions 214
- Touch Event Listener
  - implementation 215
- Touch Events
  - intercepting 213
- Touch handling 213

## U

- UiSettings class 623
- unbindService() method 427
- unregisterReceiver() method 463
- URL Mapping 679
- USB connection issues
  - resolving 60
- USE\_BIOMETRIC 688

- user interface state 99
- USE\_SIP permission 496

## V

- Video Playback 593
- VideoView class 593
  - methods 593
  - supported formats 593
- view bindings 85
  - enabling 86
  - using 86
- View class
  - setting properties 196
- view conversion 135
- ViewGroup 119
- View Groups 119
- View Hierarchy 121
- ViewHolder class 388
  - sample implementation 395
- ViewModel
  - adding LiveData 271
  - data access 268
  - fragment association 266
  - overview 258
  - saved state 293
  - Saved State 259, 293
  - tutorial 263
- ViewModelProvider 266
- ViewModel Saved State 293
- ViewPager 375, 380
  - adapter 380
  - adding to layout 379
  - example 376
- Views 119
  - Java creation 189
- View System 80
- Virtual Device Configuration dialog 28
- Virtual Sensors 41
- Visible Process 92

## W

- WebViewClient 645, 650

## Index

WebView view 451  
Weighted chain 144, 172  
Welcome screen 47  
Widget Dimensions 145  
Widget Group Alignment 167  
Widgets palette 178  
WiFi debugging 61  
Wireless debugging 61  
Wireless pairing 61  
wrap\_content properties 187  
WRITE\_CALENDAR permission 496  
WRITE\_CALL\_LOG permission 496  
WRITE\_CONTACTS permission 496  
WRITE\_EXTERNAL\_STORAGE permission 497

## X

XML Layout File  
    manual creation 185  
    vs. Java Code 189