

Android Studio Chipmunk Essentials

Java Edition

Android Studio Chipmunk Essentials – Java Edition

ISBN-13: 978-1-951442-48-4

© 2022 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

1. Introduction	1
1.1 Downloading the Code Samples	1
1.2 Feedback	1
1.3 Errata	2
2. Setting up an Android Studio Development Environment	3
2.1 System Requirements	3
2.2 Downloading the Android Studio Package	3
2.3 Installing Android Studio	4
2.3.1 Installation on Windows	4
2.3.2 Installation on macOS	4
2.3.3 Installation on Linux	5
2.4 The Android Studio Setup Wizard	5
2.5 Installing Additional Android SDK Packages	6
2.6 Making the Android SDK Tools Command-line Accessible	9
2.6.1 Windows 8.1	9
2.6.2 Windows 10	10
2.6.3 Windows 11	10
2.6.4 Linux	10
2.6.5 macOS	10
2.7 Android Studio Memory Management	11
2.8 Updating Android Studio and the SDK	12
2.9 Summary	12
3. Creating an Example Android App in Android Studio	13
3.1 About the Project	13
3.2 Creating a New Android Project	13
3.3 Creating an Activity	14
3.4 Defining the Project and SDK Settings	14
3.5 Modifying the Example Application	15
3.6 Modifying the User Interface	16
3.7 Reviewing the Layout and Resource Files	21
3.8 Adding Interaction	24
3.9 Summary	25
4. Creating an Android Virtual Device (AVD) in Android Studio	27
4.1 About Android Virtual Devices	27
4.2 Starting the Emulator	28
4.3 Running the Application in the AVD	29
4.4 Running on Multiple Devices	31
4.5 Stopping a Running Application	31
4.6 Supporting Dark Theme	32
4.7 Running the Emulator in a Separate Window	33
4.8 Enabling the Device Frame	34

Table of Contents

4.9 AVD Command-line Creation	35
4.10 Android Virtual Device Configuration Files	37
4.11 Moving and Renaming an Android Virtual Device	37
4.12 Summary	37
5. Using and Configuring the Android Studio AVD Emulator	39
5.1 The Emulator Environment	39
5.2 Emulator Toolbar Options	39
5.3 Working in Zoom Mode	41
5.4 Resizing the Emulator Window.....	41
5.5 Extended Control Options.....	41
5.5.1 Location	42
5.5.2 Displays.....	42
5.5.3 Cellular	42
5.5.4 Battery.....	42
5.5.5 Camera.....	42
5.5.6 Phone.....	42
5.5.7 Directional Pad.....	42
5.5.8 Microphone.....	42
5.5.9 Fingerprint	42
5.5.10 Virtual Sensors	43
5.5.11 Snapshots.....	43
5.5.12 Record and Playback	43
5.5.13 Google Play	43
5.5.14 Settings	43
5.5.15 Help.....	43
5.6 Working with Snapshots.....	43
5.7 Configuring Fingerprint Emulation	44
5.8 The Emulator in Tool Window Mode.....	45
5.9 Summary	46
6. A Tour of the Android Studio User Interface	47
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	53
6.7 Summary	54
7. Testing Android Studio Apps on a Physical Android Device.....	55
7.1 An Overview of the Android Debug Bridge (ADB)	55
7.2 Enabling USB Debugging ADB on Android Devices.....	55
7.2.1 macOS ADB Configuration	56
7.2.2 Windows ADB Configuration	57
7.2.3 Linux adb Configuration.....	58
7.3 Resolving USB Connection Issues	58
7.4 Enabling Wireless Debugging on Android Devices	59
7.5 Testing the adb Connection	61
7.6 Summary	61

8. The Basics of the Android Studio Code Editor.....	63
8.1 The Android Studio Editor.....	63
8.2 Splitting the Editor Window.....	65
8.3 Code Completion.....	66
8.4 Statement Completion.....	67
8.5 Parameter Information.....	68
8.6 Parameter Name Hints.....	68
8.7 Code Generation.....	68
8.8 Code Folding.....	69
8.9 Quick Documentation Lookup.....	70
8.10 Code Reformatting.....	71
8.11 Finding Sample Code.....	71
8.12 Live Templates.....	72
8.13 Summary.....	72
9. An Overview of the Android Architecture	73
9.1 The Android Software Stack.....	73
9.2 The Linux Kernel.....	74
9.3 Android Runtime – ART.....	74
9.4 Android Libraries.....	74
9.4.1 C/C++ Libraries.....	75
9.5 Application Framework.....	75
9.6 Applications.....	76
9.7 Summary.....	76
10. The Anatomy of an Android Application	77
10.1 Android Activities.....	77
10.2 Android Fragments.....	77
10.3 Android Intents.....	78
10.4 Broadcast Intents.....	78
10.5 Broadcast Receivers.....	78
10.6 Android Services.....	78
10.7 Content Providers.....	79
10.8 The Application Manifest.....	79
10.9 Application Resources.....	79
10.10 Application Context.....	79
10.11 Summary.....	79
11. An Overview of Android View Binding.....	81
11.1 Find View by Id.....	81
11.2 View Binding.....	81
11.3 Converting the AndroidSample project.....	82
11.4 Enabling View Binding.....	82
11.5 Using View Binding.....	82
11.6 Choosing an Option.....	83
11.7 View Binding in the Book Examples.....	83
11.8 Migrating a Project to View Binding.....	84
11.9 Summary.....	85
12. Understanding Android Application and Activity Lifecycles.....	87
12.1 Android Applications and Resource Management.....	87

Table of Contents

12.2 Android Process States	87
12.2.1 Foreground Process	88
12.2.2 Visible Process	88
12.2.3 Service Process	88
12.2.4 Background Process.....	88
12.2.5 Empty Process	89
12.3 Inter-Process Dependencies	89
12.4 The Activity Lifecycle.....	89
12.5 The Activity Stack.....	89
12.6 Activity States	90
12.7 Configuration Changes	90
12.8 Handling State Change.....	91
12.9 Summary	91
13. Handling Android Activity State Changes.....	93
13.1 New vs. Old Lifecycle Techniques.....	93
13.2 The Activity and Fragment Classes.....	93
13.3 Dynamic State vs. Persistent State.....	95
13.4 The Android Lifecycle Methods.....	96
13.5 Lifetimes	97
13.6 Foldable Devices and Multi-Resume	98
13.7 Disabling Configuration Change Restarts	98
13.8 Lifecycle Method Limitations.....	98
13.9 Summary	99
14. Android Activity State Changes by Example	101
14.1 Creating the State Change Example Project	101
14.2 Designing the User Interface	102
14.3 Overriding the Activity Lifecycle Methods	102
14.4 Filtering the Logcat Panel.....	104
14.5 Running the Application.....	105
14.6 Experimenting with the Activity.....	106
14.7 Summary	107
15. Saving and Restoring the State of an Android Activity	109
15.1 Saving Dynamic State	109
15.2 Default Saving of User Interface State	109
15.3 The Bundle Class	110
15.4 Saving the State.....	111
15.5 Restoring the State	112
15.6 Testing the Application.....	112
15.7 Summary	112
16. Understanding Android Views, View Groups and Layouts	115
16.1 Designing for Different Android Devices.....	115
16.2 Views and View Groups	115
16.3 Android Layout Managers	115
16.4 The View Hierarchy	117
16.5 Creating User Interfaces	118
16.6 Summary	118
17. A Guide to the Android Studio Layout Editor Tool	119

17.1 Basic vs. Empty Activity Templates	119
17.2 The Android Studio Layout Editor	123
17.3 Design Mode.....	123
17.4 The Palette	124
17.5 Design Mode and Layout Views.....	125
17.6 Night Mode	126
17.7 Code Mode.....	126
17.8 Split Mode	127
17.9 Setting Attributes.....	127
17.10 Transforms	129
17.11 Tools Visibility Toggles.....	130
17.12 Converting Views.....	132
17.13 Displaying Sample Data	132
17.14 Creating a Custom Device Definition	133
17.15 Changing the Current Device.....	134
17.16 Layout Validation (Multi Preview)	134
17.17 Summary	135
18. A Guide to the Android ConstraintLayout.....	137
18.1 How ConstraintLayout Works.....	137
18.1.1 Constraints.....	137
18.1.2 Margins.....	138
18.1.3 Opposing Constraints.....	138
18.1.4 Constraint Bias	139
18.1.5 Chains.....	140
18.1.6 Chain Styles.....	140
18.2 Baseline Alignment.....	141
18.3 Configuring Widget Dimensions.....	141
18.4 Guideline Helper	142
18.5 Group Helper	142
18.6 Barrier Helper	142
18.7 Flow Helper	144
18.8 Ratios	145
18.9 ConstraintLayout Advantages	145
18.10 ConstraintLayout Availability.....	146
18.11 Summary	146
19. A Guide to Using ConstraintLayout in Android Studio	147
19.1 Design and Layout Views.....	147
19.2 Autoconnect Mode	148
19.3 Inference Mode.....	149
19.4 Manipulating Constraints Manually.....	149
19.5 Adding Constraints in the Inspector	150
19.6 Viewing Constraints in the Attributes Window.....	151
19.7 Deleting Constraints.....	152
19.8 Adjusting Constraint Bias	152
19.9 Understanding ConstraintLayout Margins.....	153
19.10 The Importance of Opposing Constraints and Bias	154
19.11 Configuring Widget Dimensions.....	156
19.12 Design Time Tools Positioning	157

Table of Contents

19.13 Adding Guidelines	158
19.14 Adding Barriers	160
19.15 Adding a Group	161
19.16 Working with the Flow Helper	162
19.17 Widget Group Alignment and Distribution	163
19.18 Converting other Layouts to ConstraintLayout	164
19.19 Summary	164
20. Working with ConstraintLayout Chains and Ratios in Android Studio	165
20.1 Creating a Chain	165
20.2 Changing the Chain Style	167
20.3 Spread Inside Chain Style	167
20.4 Packed Chain Style	168
20.5 Packed Chain Style with Bias	168
20.6 Weighted Chain	168
20.7 Working with Ratios	169
20.8 Summary	171
21. An Android Studio Layout Editor ConstraintLayout Tutorial	173
21.1 An Android Studio Layout Editor Tool Example	173
21.2 Creating a New Activity	173
21.3 Preparing the Layout Editor Environment	175
21.4 Adding the Widgets to the User Interface	176
21.5 Adding the Constraints	179
21.6 Testing the Layout	180
21.7 Using the Layout Inspector	181
21.8 Summary	182
22. Manual XML Layout Design in Android Studio	183
22.1 Manually Creating an XML Layout	183
22.2 Manual XML vs. Visual Layout Design	186
22.3 Summary	186
23. Managing Constraints using Constraint Sets	187
23.1 Java Code vs. XML Layout Files	187
23.2 Creating Views	187
23.3 View Attributes	188
23.4 Constraint Sets	188
23.4.1 Establishing Connections	188
23.4.2 Applying Constraints to a Layout	188
23.4.3 Parent Constraint Connections	188
23.4.4 Sizing Constraints	189
23.4.5 Constraint Bias	189
23.4.6 Alignment Constraints	189
23.4.7 Copying and Applying Constraint Sets	189
23.4.8 ConstraintLayout Chains	189
23.4.9 Guidelines	190
23.4.10 Removing Constraints	190
23.4.11 Scaling	190
23.4.12 Rotation	191
23.5 Summary	191

24. An Android ConstraintSet Tutorial.....	193
24.1 Creating the Example Project in Android Studio	193
24.2 Adding Views to an Activity.....	193
24.3 Setting View Attributes.....	194
24.4 Creating View IDs.....	195
24.5 Configuring the Constraint Set	196
24.6 Adding the EditText View	197
24.7 Converting Density Independent Pixels (dp) to Pixels (px).....	198
24.8 Summary	199
25. A Guide to using Apply Changes in Android Studio.....	201
25.1 Introducing Apply Changes.....	201
25.2 Understanding Apply Changes Options	201
25.3 Using Apply Changes.....	202
25.4 Configuring Apply Changes Fallback Settings	203
25.5 An Apply Changes Tutorial.....	203
25.6 Using Apply Code Changes	203
25.7 Using Apply Changes and Restart Activity.....	204
25.8 Using Run App	204
25.9 Summary	204
26. An Overview and Example of Android Event Handling	205
26.1 Understanding Android Events.....	205
26.2 Using the android:onClick Resource.....	205
26.3 Event Listeners and Callback Methods	206
26.4 An Event Handling Example	206
26.5 Designing the User Interface	207
26.6 The Event Listener and Callback Method.....	208
26.7 Consuming Events	209
26.8 Summary	210
27. Android Touch and Multi-touch Event Handling	211
27.1 Intercepting Touch Events	211
27.2 The MotionEvent Object	211
27.3 Understanding Touch Actions.....	212
27.4 Handling Multiple Touches	212
27.5 An Example Multi-Touch Application	212
27.6 Designing the Activity User Interface	213
27.7 Implementing the Touch Event Listener.....	213
27.8 Running the Example Application.....	216
27.9 Summary	217
28. Detecting Common Gestures Using the Android Gesture Detector Class	219
28.1 Implementing Common Gesture Detection.....	219
28.2 Creating an Example Gesture Detection Project	220
28.3 Implementing the Listener Class.....	220
28.4 Creating the GestureDetectorCompat Instance.....	222
28.5 Implementing the onTouchEvent() Method.....	223
28.6 Testing the Application.....	223
28.7 Summary	224

29. Implementing Custom Gesture and Pinch Recognition on Android	225
29.1 The Android Gesture Builder Application.....	225
29.2 The GestureOverlayView Class	225
29.3 Detecting Gestures.....	225
29.4 Identifying Specific Gestures	225
29.5 Installing and Running the Gesture Builder Application	226
29.6 Creating a Gestures File	226
29.7 Creating the Example Project.....	226
29.8 Extracting the Gestures File from the SD Card	227
29.9 Adding the Gestures File to the Project	227
29.10 Designing the User Interface	227
29.11 Loading the Gestures File	228
29.12 Registering the Event Listener.....	229
29.13 Implementing the onGesturePerformed Method.....	229
29.14 Testing the Application.....	230
29.15 Configuring the GestureOverlayView.....	231
29.16 Intercepting Gestures.....	231
29.17 Detecting Pinch Gestures.....	231
29.18 A Pinch Gesture Example Project.....	232
29.19 Summary.....	234
30. An Introduction to Android Fragments.....	235
30.1 What is a Fragment?	235
30.2 Creating a Fragment	235
30.3 Adding a Fragment to an Activity using the Layout XML File.....	236
30.4 Adding and Managing Fragments in Code	238
30.5 Handling Fragment Events	239
30.6 Implementing Fragment Communication.....	240
30.7 Summary	241
31. Using Fragments in Android Studio - An Example.....	243
31.1 About the Example Fragment Application	243
31.2 Creating the Example Project.....	243
31.3 Creating the First Fragment Layout.....	243
31.4 Migrating a Fragment to View Binding	245
31.5 Adding the Second Fragment.....	246
31.6 Adding the Fragments to the Activity	247
31.7 Making the Toolbar Fragment Talk to the Activity	248
31.8 Making the Activity Talk to the Text Fragment	251
31.9 Testing the Application.....	252
31.10 Summary.....	253
32. Modern Android App Architecture with Jetpack.....	255
32.1 What is Android Jetpack?	255
32.2 The “Old” Architecture.....	255
32.3 Modern Android Architecture.....	255
32.4 The ViewModel Component	256
32.5 The LiveData Component.....	256
32.6 ViewModel Saved State.....	257
32.7 LiveData and Data Binding.....	258

32.8 Android Lifecycles	258
32.9 Repository Modules	258
32.10 Summary	259
33. An Android Jetpack ViewModel Tutorial	261
33.1 About the Project	261
33.2 Creating the ViewModel Example Project	261
33.3 Reviewing the Project	262
33.3.1 The Main Activity	262
33.3.2 The Content Fragment	262
33.3.3 The ViewModel	264
33.4 Designing the Fragment Layout	264
33.5 Implementing the View Model	265
33.6 Associating the Fragment with the View Model	265
33.7 Modifying the Fragment	266
33.8 Accessing the ViewModel Data	267
33.9 Testing the Project	268
33.10 Summary	268
34. An Android Jetpack LiveData Tutorial	269
34.1 LiveData - A Recap	269
34.2 Adding LiveData to the ViewModel	269
34.3 Implementing the Observer	271
34.4 Summary	273
35. An Overview of Android Jetpack Data Binding	275
35.1 An Overview of Data Binding	275
35.2 The Key Components of Data Binding	275
35.2.1 The Project Build Configuration	275
35.2.2 The Data Binding Layout File	276
35.2.3 The Layout File Data Element	277
35.2.4 The Binding Classes	278
35.2.5 Data Binding Variable Configuration	278
35.2.6 Binding Expressions (One-Way)	279
35.2.7 Binding Expressions (Two-Way)	280
35.2.8 Event and Listener Bindings	280
35.3 Summary	281
36. An Android Jetpack Data Binding Tutorial	283
36.1 Removing the Redundant Code	283
36.2 Enabling Data Binding	285
36.3 Adding the Layout Element	285
36.4 Adding the Data Element to Layout File	286
36.5 Working with the Binding Class	286
36.6 Assigning the ViewModel Instance to the Data Binding Variable	287
36.7 Adding Binding Expressions	288
36.8 Adding the Conversion Method	289
36.9 Adding a Listener Binding	289
36.10 Testing the App	289
36.11 Summary	290
37. An Android ViewModel Saved State Tutorial	291

Table of Contents

37.1 Understanding ViewModel State Saving.....	291
37.2 Implementing ViewModel State Saving	292
37.3 Saving and Restoring State.....	293
37.4 Adding Saved State Support to the ViewModelDemo Project.....	293
37.5 Summary	295
38. Working with Android Lifecycle-Aware Components	297
38.1 Lifecycle Awareness	297
38.2 Lifecycle Owners	297
38.3 Lifecycle Observers	298
38.4 Lifecycle States and Events.....	299
38.5 Summary	300
39. An Android Jetpack Lifecycle Awareness Tutorial	301
39.1 Creating the Example Lifecycle Project.....	301
39.2 Creating a Lifecycle Observer.....	301
39.3 Adding the Observer	302
39.4 Testing the Observer.....	303
39.5 Creating a Lifecycle Owner.....	303
39.6 Testing the Custom Lifecycle Owner.....	305
39.7 Summary	306
40. An Overview of the Navigation Architecture Component.....	307
40.1 Understanding Navigation	307
40.2 Declaring a Navigation Host.....	308
40.3 The Navigation Graph	310
40.4 Accessing the Navigation Controller	311
40.5 Triggering a Navigation Action	311
40.6 Passing Arguments.....	312
40.7 Summary	312
41. An Android Jetpack Navigation Component Tutorial	313
41.1 Creating the NavigationDemo Project.....	313
41.2 Adding Navigation to the Build Configuration.....	313
41.3 Creating the Navigation Graph Resource File.....	314
41.4 Declaring a Navigation Host.....	315
41.5 Adding Navigation Destinations.....	317
41.6 Designing the Destination Fragment Layouts.....	318
41.7 Adding an Action to the Navigation Graph.....	319
41.8 Implement the OnFragmentInteractionListener	321
41.9 Adding View Binding Support to the Destination Fragments	322
41.10 Triggering the Action	323
41.11 Passing Data Using Safeargs	323
41.12 Summary.....	327
42. An Introduction to MotionLayout.....	329
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
43. An Android MotionLayout Editor.....	341
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
44. A MotionLayout KeyCycle Tutorial	355
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
45. Working with the Floating Action Button and Snackbar	367
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
46. Creating a Tabbed Interface using the TabLayout Component	375
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

Table of Contents

46.7 Creating the Pager Adapter.....	380
46.8 Performing the Initialization Tasks.....	381
46.9 Testing the Application.....	383
46.10 Customizing the TabLayout.....	383
46.11 Summary.....	384
47. Working with the RecyclerView and CardView Widgets.....	385
47.1 An Overview of the RecyclerView.....	385
47.2 An Overview of the CardView.....	387
47.3 Summary.....	388
48. An Android RecyclerView and CardView Tutorial.....	389
48.1 Creating the CardDemo Project.....	389
48.2 Modifying the Basic Activity Project.....	389
48.3 Designing the CardView Layout.....	390
48.4 Adding the RecyclerView.....	391
48.5 Adding the Image Files.....	391
48.6 Creating the RecyclerView Adapter.....	392
48.7 Initializing the RecyclerView Component.....	394
48.8 Testing the Application.....	395
48.9 Responding to Card Selections.....	396
48.10 Summary.....	397
49. A Layout Editor Sample Data Tutorial.....	399
49.1 Adding Sample Data to a Project.....	399
49.2 Using Custom Sample Data.....	403
49.3 Summary.....	406
50. Working with the AppBar and Collapsing Toolbar Layouts.....	407
50.1 The Anatomy of an AppBar.....	407
50.2 The Example Project.....	408
50.3 Coordinating the RecyclerView and Toolbar.....	408
50.4 Introducing the Collapsing Toolbar Layout.....	410
50.5 Changing the Title and Scrim Color.....	413
50.6 Summary.....	414
51. An Android Studio Primary/Detail Flow Tutorial.....	415
51.1 The Primary/Detail Flow.....	415
51.2 Creating a Primary/Detail Flow Activity.....	416
51.3 Modifying the Primary/Detail Flow Template.....	417
51.4 Changing the Content Model.....	417
51.5 Changing the Detail Pane.....	419
51.6 Modifying the WebsiteDetailFragment Class.....	420
51.7 Modifying the WebsiteListFragment Class.....	421
51.8 Adding Manifest Permissions.....	422
51.9 Running the Application.....	422
51.10 Summary.....	422
52. An Overview of Android Intents.....	423
52.1 An Overview of Intents.....	423
52.2 Explicit Intents.....	423
52.3 Returning Data from an Activity.....	424

52.4 Implicit Intents	425
52.5 Using Intent Filters.....	426
52.6 Automatic Link Verification	427
52.7 Manually Enabling Links	429
52.8 Checking Intent Availability	431
52.9 Summary	431
53. Android Explicit Intents – A Worked Example	433
53.1 Creating the Explicit Intent Example Application.....	433
53.2 Designing the User Interface Layout for MainActivity.....	433
53.3 Creating the Second Activity Class.....	434
53.4 Designing the User Interface Layout for SecondActivity	435
53.5 Reviewing the Application Manifest File	435
53.6 Creating the Intent.....	436
53.7 Extracting Intent Data	437
53.8 Launching SecondActivity as a Sub-Activity.....	438
53.9 Returning Data from a Sub-Activity.....	439
53.10 Testing the Application.....	439
53.11 Summary	439
54. Android Implicit Intents – A Worked Example	441
54.1 Creating the Android Studio Implicit Intent Example Project	441
54.2 Designing the User Interface	441
54.3 Creating the Implicit Intent	442
54.4 Adding a Second Matching Activity.....	443
54.5 Adding the Web View to the UI.....	443
54.6 Obtaining the Intent URL	444
54.7 Modifying the MyWebView Project Manifest File	445
54.8 Installing the MyWebView Package on a Device.....	446
54.9 Testing the Application.....	447
54.10 Manually Enabling the Link	447
54.11 Automatic Link Verification	449
54.12 Summary	451
55. Android Broadcast Intents and Broadcast Receivers	453
55.1 An Overview of Broadcast Intents.....	453
55.2 An Overview of Broadcast Receivers	454
55.3 Obtaining Results from a Broadcast.....	455
55.4 Sticky Broadcast Intents	455
55.5 The Broadcast Intent Example.....	456
55.6 Creating the Example Application.....	456
55.7 Creating and Sending the Broadcast Intent.....	456
55.8 Creating the Broadcast Receiver	457
55.9 Registering the Broadcast Receiver.....	458
55.10 Testing the Broadcast Example	459
55.11 Listening for System Broadcasts.....	459
55.12 Summary	460
56. A Basic Overview of Java Threads, Handlers and Executors.....	461
56.1 An Overview of Threads	461
56.2 The Application Main Thread.....	461

Table of Contents

56.3 Thread Handlers.....	461
56.4 A Threading Example	461
56.5 Building the App	462
56.6 Creating a New Thread.....	463
56.7 Implementing a Thread Handler.....	464
56.8 Passing a Message to the Handler	465
56.9 Java Executor Concurrency	466
56.10 Working with Runnable Tasks.....	467
56.11 Shutting down an Executor Service.....	468
56.12 Working with Callable Tasks and Futures	468
56.13 Handling a Future Result	470
56.14 Scheduling Tasks	471
56.15 Summary	472
57. An Overview of Android Services.....	473
57.1 Started Services.....	473
57.2 Intent Service.....	473
57.3 Bound Service.....	474
57.4 The Anatomy of a Service	474
57.5 Controlling Destroyed Service Restart Options.....	475
57.6 Declaring a Service in the Manifest File.....	475
57.7 Starting a Service Running on System Startup.....	476
57.8 Summary	476
58. Implementing an Android Started Service – A Worked Example	477
58.1 Creating the Example Project.....	477
58.2 Designing the User Interface	477
58.3 Creating the Service Class.....	477
58.4 Adding the Service to the Manifest File	479
58.5 Starting the Service	479
58.6 Testing the IntentService Example.....	480
58.7 Using the Service Class.....	480
58.8 Creating the New Service	481
58.9 Launching the Service	482
58.10 Running the Application.....	482
58.11 Adding Threading to the Service	483
58.12 Summary	484
59. Android Local Bound Services – A Worked Example.....	485
59.1 Understanding Bound Services.....	485
59.2 Bound Service Interaction Options.....	485
59.3 A Local Bound Service Example.....	485
59.4 Adding a Bound Service to the Project	486
59.5 Implementing the Binder	486
59.6 Binding the Client to the Service	489
59.7 Completing the Example.....	490
59.8 Testing the Application.....	491
59.9 Summary	491
60. Android Remote Bound Services – A Worked Example	493
60.1 Client to Remote Service Communication.....	493

60.2 Creating the Example Application	493
60.3 Designing the User Interface	493
60.4 Implementing the Remote Bound Service.....	494
60.5 Configuring a Remote Service in the Manifest File.....	495
60.6 Launching and Binding to the Remote Service.....	496
60.7 Sending a Message to the Remote Service	497
60.8 Summary	498
61. An Android Notifications Tutorial	499
61.1 An Overview of Notifications.....	499
61.2 Creating the NotifyDemo Project.....	501
61.3 Designing the User Interface	501
61.4 Creating the Second Activity	501
61.5 Creating a Notification Channel	502
61.6 Creating and Issuing a Notification	504
61.7 Launching an Activity from a Notification.....	506
61.8 Adding Actions to a Notification	508
61.9 Bundled Notifications.....	508
61.10 Summary	510
62. An Android Direct Reply Notification Tutorial	513
62.1 Creating the DirectReply Project.....	513
62.2 Designing the User Interface	513
62.3 Creating the Notification Channel.....	514
62.4 Building the RemoteInput Object.....	515
62.5 Creating the PendingIntent.....	516
62.6 Creating the Reply Action.....	516
62.7 Receiving Direct Reply Input.....	519
62.8 Updating the Notification	520
62.9 Summary	521
63. Foldable Devices and Multi-Window Support	523
63.1 Foldables and Multi-Window Support.....	523
63.2 Using a Foldable Emulator.....	524
63.3 Entering Multi-Window Mode	525
63.4 Enabling and using Freeform Support	526
63.5 Checking for Freeform Support	526
63.6 Enabling Multi-Window Support in an App.....	526
63.7 Specifying Multi-Window Attributes	527
63.8 Detecting Multi-Window Mode in an Activity.....	528
63.9 Receiving Multi-Window Notifications.....	528
63.10 Launching an Activity in Multi-Window Mode	529
63.11 Configuring Freeform Activity Size and Position.....	529
63.12 Summary	530
64. An Overview of Android SQLite Databases	531
64.1 Understanding Database Tables.....	531
64.2 Introducing Database Schema	531
64.3 Columns and Data Types	531
64.4 Database Rows	532
64.5 Introducing Primary Keys	532

Table of Contents

64.6 What is SQLite?	532
64.7 Structured Query Language (SQL)	532
64.8 Trying SQLite on an Android Virtual Device (AVD)	533
64.9 The Android Room Persistence Library.....	535
64.10 Summary	535
65. The Android Room Persistence Library	537
65.1 Revisiting Modern App Architecture	537
65.2 Key Elements of Room Database Persistence.....	537
65.2.1 Repository	538
65.2.2 Room Database	538
65.2.3 Data Access Object (DAO)	538
65.2.4 Entities	538
65.2.5 SQLite Database	538
65.3 Understanding Entities.....	539
65.4 Data Access Objects.....	542
65.5 The Room Database.....	543
65.6 The Repository.....	544
65.7 In-Memory Databases	545
65.8 Database Inspector.....	545
65.9 Summary	545
66. An Android TableLayout and TableRow Tutorial	547
66.1 The TableLayout and TableRow Layout Views	547
66.2 Creating the Room Database Project	548
66.3 Converting to a LinearLayout.....	548
66.4 Adding the TableLayout to the User Interface.....	549
66.5 Configuring the TableRows	550
66.6 Adding the Button Bar to the Layout	551
66.7 Adding the RecyclerView.....	552
66.8 Adjusting the Layout Margins	553
66.9 Summary	553
67. An Android Room Database and Repository Tutorial.....	555
67.1 About the RoomDemo Project.....	555
67.2 Modifying the Build Configuration.....	555
67.3 Building the Entity	556
67.4 Creating the Data Access Object.....	557
67.5 Adding the Room Database.....	558
67.6 Adding the Repository	559
67.7 Modifying the ViewModel.....	562
67.8 Creating the Product Item Layout	563
67.9 Adding the RecyclerView Adapter.....	564
67.10 Preparing the Main Fragment	565
67.11 Adding the Button Listeners.....	566
67.12 Adding LiveData Observers	568
67.13 Initializing the RecyclerView.....	569
67.14 Testing the RoomDemo App.....	569
67.15 Using the Database Inspector.....	569
67.16 Summary	570

68. Accessing Cloud Storage using the Android Storage Access Framework.....	571
68.1 The Storage Access Framework.....	571
68.2 Working with the Storage Access Framework.....	572
68.3 Filtering Picker File Listings.....	572
68.4 Handling Intent Results.....	573
68.5 Reading the Content of a File.....	573
68.6 Writing Content to a File.....	574
68.7 Deleting a File.....	575
68.8 Gaining Persistent Access to a File.....	575
68.9 Summary.....	575
69. An Android Storage Access Framework Example.....	577
69.1 About the Storage Access Framework Example.....	577
69.2 Creating the Storage Access Framework Example.....	577
69.3 Designing the User Interface.....	577
69.4 Adding the Activity Launchers.....	578
69.5 Creating a New Storage File.....	580
69.6 Saving to a Storage File.....	582
69.7 Opening and Reading a Storage File.....	583
69.8 Testing the Storage Access Application.....	584
69.9 Summary.....	585
70. Video Playback on Android using the VideoView and MediaController Classes.....	587
70.1 Introducing the Android VideoView Class.....	587
70.2 Introducing the Android MediaController Class.....	588
70.3 Creating the Video Playback Example.....	588
70.4 Designing the VideoPlayer Layout.....	588
70.5 Downloading the Video File.....	589
70.6 Configuring the VideoView.....	589
70.7 Adding the MediaController to the Video View.....	591
70.8 Setting up the onPreparedListener.....	592
70.9 Summary.....	593
71. Android Picture-in-Picture Mode.....	595
71.1 Picture-in-Picture Features.....	595
71.2 Enabling Picture-in-Picture Mode.....	596
71.3 Configuring Picture-in-Picture Parameters.....	596
71.4 Entering Picture-in-Picture Mode.....	597
71.5 Detecting Picture-in-Picture Mode Changes.....	597
71.6 Adding Picture-in-Picture Actions.....	598
71.7 Summary.....	598
72. An Android Picture-in-Picture Tutorial.....	601
72.1 Adding Picture-in-Picture Support to the Manifest.....	601
72.2 Adding a Picture-in-Picture Button.....	601
72.3 Entering Picture-in-Picture Mode.....	601
72.4 Detecting Picture-in-Picture Mode Changes.....	603
72.5 Adding a Broadcast Receiver.....	604
72.6 Adding the PiP Action.....	605
72.7 Testing the Picture-in-Picture Action.....	607
72.8 Summary.....	608

73. Making Runtime Permission Requests in Android	609
73.1 Understanding Normal and Dangerous Permissions.....	609
73.2 Creating the Permissions Example Project.....	611
73.3 Checking for a Permission	611
73.4 Requesting Permission at Runtime.....	613
73.5 Providing a Rationale for the Permission Request	614
73.6 Testing the Permissions App.....	616
73.7 Summary	616
74. Android Audio Recording and Playback using MediaPlayer and MediaRecorder	617
74.1 Playing Audio	617
74.2 Recording Audio and Video using the MediaRecorder Class	618
74.3 About the Example Project	619
74.4 Creating the AudioApp Project.....	619
74.5 Designing the User Interface	619
74.6 Checking for Microphone Availability.....	620
74.7 Initializing the Activity.....	621
74.8 Implementing the recordAudio() Method.....	622
74.9 Implementing the stopAudio() Method.....	623
74.10 Implementing the playAudio() method.....	623
74.11 Configuring and Requesting Permissions	623
74.12 Testing the Application.....	626
74.13 Summary.....	626
75. Working with the Google Maps Android API in Android Studio	627
75.1 The Elements of the Google Maps Android API	627
75.2 Creating the Google Maps Project.....	628
75.3 Obtaining Your Developer Signature	628
75.4 Adding the Apache HTTP Legacy Library Requirement	629
75.5 Testing the Application.....	629
75.6 Understanding Geocoding and Reverse Geocoding	630
75.7 Adding a Map to an Application.....	632
75.8 Requesting Current Location Permission.....	632
75.9 Displaying the User's Current Location	633
75.10 Changing the Map Type.....	634
75.11 Displaying Map Controls to the User.....	635
75.12 Handling Map Gesture Interaction.....	636
75.12.1 Map Zooming Gestures.....	636
75.12.2 Map Scrolling/Panning Gestures	636
75.12.3 Map Tilt Gestures.....	637
75.12.4 Map Rotation Gestures.....	637
75.13 Creating Map Markers.....	637
75.14 Controlling the Map Camera	638
75.15 Summary.....	639
76. Printing with the Android Printing Framework	641
76.1 The Android Printing Architecture	641
76.2 The Print Service Plugins	641
76.3 Google Cloud Print.....	642
76.4 Printing to Google Drive.....	642

76.5 Save as PDF	643
76.6 Printing from Android Devices	643
76.7 Options for Building Print Support into Android Apps	644
76.7.1 Image Printing.....	644
76.7.2 Creating and Printing HTML Content	645
76.7.3 Printing a Web Page.....	646
76.7.4 Printing a Custom Document	647
76.8 Summary	647
77. An Android HTML and Web Content Printing Example	649
77.1 Creating the HTML Printing Example Application	649
77.2 Printing Dynamic HTML Content	649
77.3 Creating the Web Page Printing Example.....	652
77.4 Removing the Floating Action Button	652
77.5 Removing Navigation Features.....	652
77.6 Designing the User Interface Layout	654
77.7 Accessing the WebView from the Main Activity	654
77.8 Loading the Web Page into the WebView	655
77.9 Adding the Print Menu Option.....	656
77.10 Summary	657
78. A Guide to Android Custom Document Printing	659
78.1 An Overview of Android Custom Document Printing	659
78.1.1 Custom Print Adapters.....	659
78.2 Preparing the Custom Document Printing Project.....	660
78.3 Creating the Custom Print Adapter.....	661
78.4 Implementing the onLayout() Callback Method	662
78.5 Implementing the onWrite() Callback Method	665
78.6 Checking a Page is in Range	667
78.7 Drawing the Content on the Page Canvas	668
78.8 Starting the Print Job	670
78.9 Testing the Application.....	671
78.10 Summary	671
79. An Introduction to Android App Links.....	673
79.1 An Overview of Android App Links	673
79.2 App Link Intent Filters	673
79.3 Handling App Link Intents	674
79.4 Associating the App with a Website.....	674
79.5 Summary	675
80. An Android Studio App Links Tutorial	677
80.1 About the Example App	677
80.2 The Database Schema	677
80.3 Loading and Running the Project.....	678
80.4 Adding the URL Mapping.....	679
80.5 Adding the Intent Filter.....	682
80.6 Adding Intent Handling Code.....	682
80.7 Testing the App.....	685
80.8 Creating the Digital Asset Links File	685
80.9 Testing the App Link.....	686

80.10 Summary	686
81. An Android Biometric Authentication Tutorial.....	687
81.1 An Overview of Biometric Authentication.....	687
81.2 Creating the Biometric Authentication Project	687
81.3 Configuring Device Fingerprint Authentication	688
81.4 Adding the Biometric Permission to the Manifest File.....	688
81.5 Designing the User Interface	689
81.6 Adding a Toast Convenience Method	689
81.7 Checking the Security Settings.....	690
81.8 Configuring the Authentication Callbacks	691
81.9 Adding the CancellationSignal.....	692
81.10 Starting the Biometric Prompt	693
81.11 Testing the Project.....	693
81.12 Summary	694
82. Creating, Testing and Uploading an Android App Bundle.....	695
82.1 The Release Preparation Process.....	695
82.2 Android App Bundles.....	695
82.3 Register for a Google Play Developer Console Account.....	696
82.4 Configuring the App in the Console	697
82.5 Enabling Google Play App Signing.....	698
82.6 Creating a Keystore File	698
82.7 Creating the Android App Bundle.....	700
82.8 Generating Test APK Files	701
82.9 Uploading the App Bundle to the Google Play Developer Console.....	702
82.10 Exploring the App Bundle	703
82.11 Managing Testers	704
82.12 Rolling the App Out for Testing.....	704
82.13 Uploading New App Bundle Revisions.....	705
82.14 Analyzing the App Bundle File	706
82.15 Summary	706
83. An Overview of Android In-App Billing	709
83.1 Preparing a Project for In-App Purchasing.....	709
83.2 Creating In-App Products and Subscriptions	709
83.3 Billing Client Initialization.....	710
83.4 Connecting to the Google Play Billing Library.....	711
83.5 Displaying Available Products.....	712
83.6 Starting the Purchase Process.....	712
83.7 Completing the Purchase	713
83.8 Querying Previous Purchases.....	714
83.9 Summary	715
84. An Android In-App Purchasing Tutorial	717
84.1 About the In-App Purchasing Example Project.....	717
84.2 Creating the InAppPurchase Project	717
84.3 Adding Libraries to the Project	717
84.4 Designing the User Interface	718
84.5 Adding the App to the Google Play Store	718
84.6 Creating an In-App Product.....	719

84.7 Enabling License Testers	719
84.8 Initializing the Billing Client	720
84.9 Querying the Product.....	722
84.10 Launching the Purchase Flow	723
84.11 Handling Purchase Updates	723
84.12 Consuming the Product.....	724
84.13 Testing the App.....	725
84.14 Troubleshooting	726
84.15 Summary	726
85. An Overview of Android Dynamic Feature Modules.....	727
85.1 An Overview of Dynamic Feature Modules.....	727
85.2 Dynamic Feature Module Architecture	727
85.3 Creating a Dynamic Feature Module	728
85.4 Converting an Existing Module for Dynamic Delivery.....	730
85.5 Working with Dynamic Feature Modules.....	733
85.6 Handling Large Dynamic Feature Modules	735
85.7 Summary	736
86. An Android Studio Dynamic Feature Tutorial.....	737
86.1 Creating the DynamicFeature Project.....	737
86.2 Adding Dynamic Feature Support to the Project	737
86.3 Designing the Base Activity User Interface	738
86.4 Adding the Dynamic Feature Module.....	739
86.5 Reviewing the Dynamic Feature Module.....	740
86.6 Adding the Dynamic Feature Activity.....	741
86.7 Implementing the <code>launchIntent()</code> Method.....	744
86.8 Uploading the App Bundle for Testing.....	745
86.9 Implementing the <code>installFeature()</code> Method	746
86.10 Adding the Update Listener.....	748
86.11 Using Deferred Installation	751
86.12 Removing a Dynamic Module	751
86.13 Summary	751
87. Working with Material Design 3 Theming	753
87.1 Material Design 2 vs Material Design 3	753
87.2 Understanding Material Design Theming.....	753
87.3 Material Design 2 Theming	753
87.4 Material Design 3 Theming	755
87.5 Building a Custom Theme.....	756
87.6 Summary	757
88. Migrating from Material Design 2 to Material Design 3.....	759
88.1 Creating the ThemeMigration Project	759
88.2 Designing the User Interface	759
88.3 Migrating to Material Design 3	761
88.4 Building a New Theme	762
88.5 Adding the Theme to the Project.....	763
88.6 Enabling Dynamic Color Support	764
88.7 Summary	765
89. An Overview of Gradle in Android Studio.....	767

Table of Contents

89.1 An Overview of Gradle	767
89.2 Gradle and Android Studio	767
89.2.1 Sensible Defaults	767
89.2.2 Dependencies.....	767
89.2.3 Build Variants	768
89.2.4 Manifest Entries	768
89.2.5 APK Signing.....	768
89.2.6 ProGuard Support.....	768
89.3 The Property and Settings Gradle Build File	768
89.4 The Top-level Gradle Build File.....	769
89.5 Module Level Gradle Build Files.....	770
89.6 Configuring Signing Settings in the Build File.....	772
89.7 Running Gradle Tasks from the Command-line	773
89.8 Summary	773
Index	775

1. Introduction

Fully updated for Android Studio Chipmunk, the goal of this book is to teach you how to develop Android-based applications using the Java programming language.

Beginning with the basics, this book provides an outline of the steps necessary to set up an Android development and testing environment. An overview of Android Studio is included covering areas such as tool windows, the code editor, and the Layout Editor tool. An introduction to the architecture of Android is followed by an in-depth look at the design of Android applications and user interfaces using the Android Studio environment.

Chapters are also included covering the Android Architecture Components including view models, lifecycle management, Room database access, 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 edition of the book 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 Chipmunk 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, 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/chipmunkjava/index.php>

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

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

1.2 Feedback

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

1.3 Errata

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

<https://www.ebookfrenzy.com/errata/chipmunkjava.html>

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

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 (see below)
- Approximately 8GB of available disk space
- 1280 x 800 minimum screen resolution

Although Android Studio will run on computers with 8GB of RAM, performance will be greatly improved on systems containing more memory. This is particularly an issue if you plan to test your apps using the Android Virtual Device emulator (AVD).

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 Chipmunk 2021.2.1 using the Android API 32 SDK which, at the time of writing, are the current versions.

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

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

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

If this page provides instructions for downloading a newer version of Android Studio it is important to note that there may be some minor differences between this book and the software. A web search for “Android Studio Chipmunk” 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 Chipmunk 2021.2.1 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:

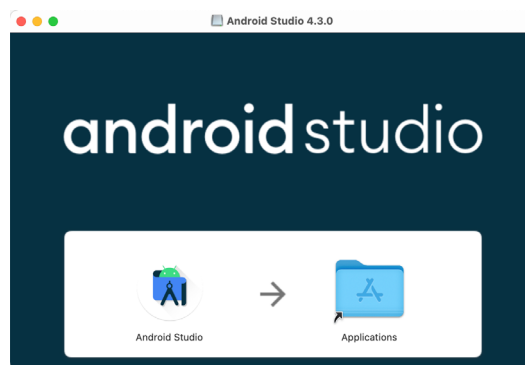


Figure 2-1

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 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:

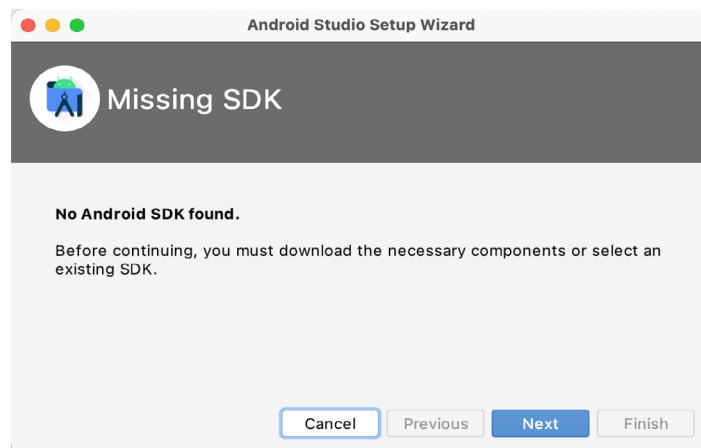


Figure 2-2

If this dialog appears, click the Next button to display the SDK Components Setup dialog (Figure 2-3). Within this dialog, make sure that the Android SDK option is selected along with the latest API package before clicking on the Next button:

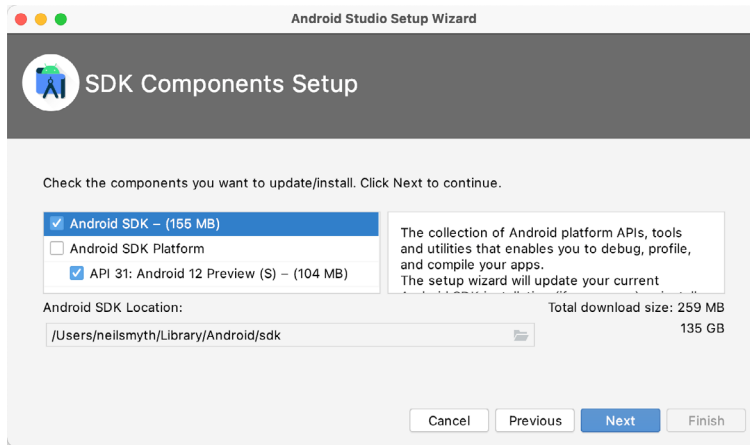


Figure 2-3

After clicking Next, Android Studio will download and install the Android SDK and tools.

If you have previously installed an earlier version of Android Studio, the first time that 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.

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

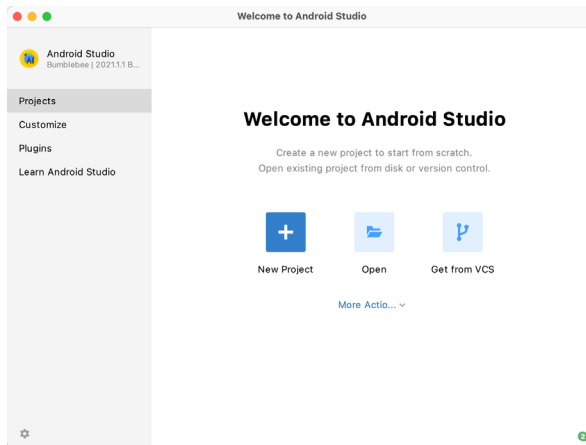


Figure 2-4

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:

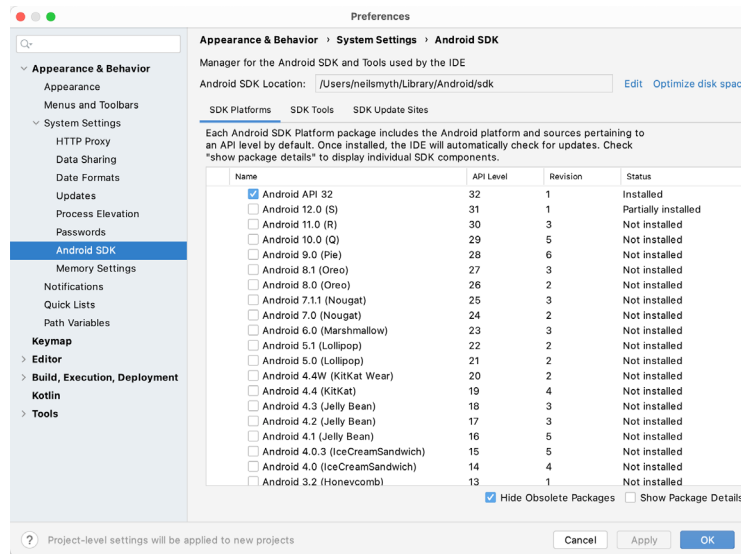


Figure 2-5

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.

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 Intel x86 Atom System Image	25	6	Not installed
<input type="checkbox"/>	Android Wear for China ARM EABI v7a System Image	25	3	Not installed
<input type="checkbox"/>	Android Wear for China Intel x86 Atom System Image	25	3	Not installed
<input type="checkbox"/>	Android Wear ARM EABI v7a System Image	25	3	Not installed
<input type="checkbox"/>	Android Wear Intel x86 Atom System Image	25	3	Not installed
<input type="checkbox"/>	Google APIs ARM 64 v8a System Image	25	8	Not installed
<input type="checkbox"/>	Google APIs ARM EABI v7a System Image	25	8	Not installed
<input type="checkbox"/>	Google APIs Intel x86 Atom System Image	25	8	Not installed
<input checked="" type="checkbox"/>	Google APIs Intel x86 Atom_64 System Image	25	6	Update Available: 8
▼ <input type="checkbox"/>	Android 7.0 (Nougat)			
<input type="checkbox"/>	Google APIs	24	1	Not 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 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:

Setting up an Android Studio Development Environment

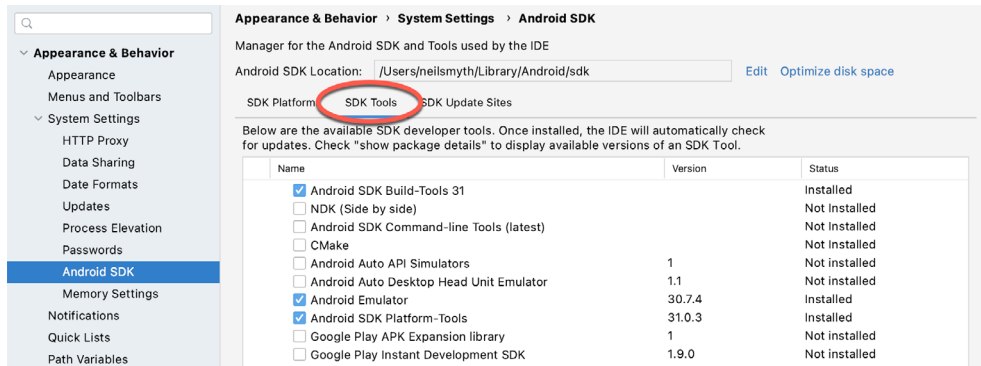


Figure 2-7

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 S

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:

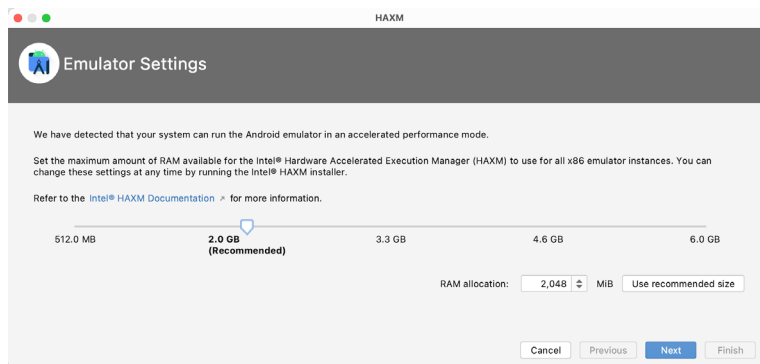


Figure 2-8

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 Making the Android SDK Tools Command-line Accessible

Most of the time, the underlying tools of the Android SDK will be accessed from within the Android Studio environment. That being said, however, there will also be instances where it will be useful to be able to invoke those tools from a command prompt or terminal window. For the operating system on which you are developing to be able to find these tools, it will be necessary to add them to the system's *PATH* environment variable.

Regardless of operating system, the *PATH* variable needs to be configured to include the following paths (where *<path_to_android_sdk_installation>* represents the file system location into which the Android SDK was installed):

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

The location of the SDK on your system can be identified by launching the SDK Manager and referring to the *Android SDK Location*: field located at the top of the settings panel as highlighted in Figure 2-9:

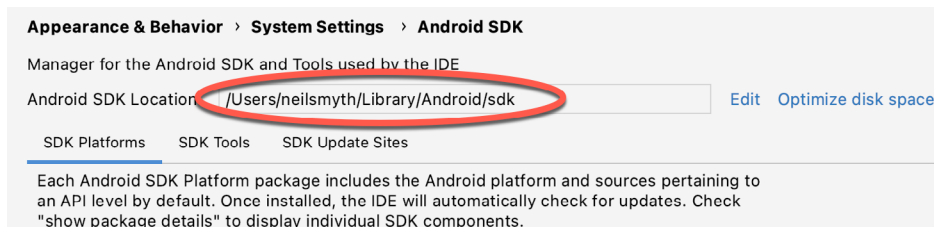


Figure 2-9

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 three new entries to the path. For example, assuming the Android SDK was installed into C:\Users\demo\AppData\Local\Android\Sdk, the following entries would need to be added:

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

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

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

Setting up an Android Studio Development Environment

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/tools:/  
home/demo/Android/sdk/tools/bin:/home/demo/android-studio/bin:$PATH
```

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

2.6.5 macOS

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

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

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

2.7 Android Studio Memory Management

Android Studio is a large and complex software application that consists of many background processes. Although Android Studio has been criticized in the past for providing less than optimal performance, Google has made significant performance improvements in recent releases and continues to do so with each new version. 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:

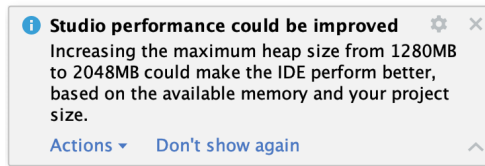


Figure 2-10

To view and modify the current memory configuration, select the *File -> Settings...* (*Android Studio -> Preferences...* on macOS) menu option and, in the resulting dialog, select the *Memory Settings* option listed under *System Settings* in the left-hand navigation panel as illustrated in Figure 2-11 below.

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

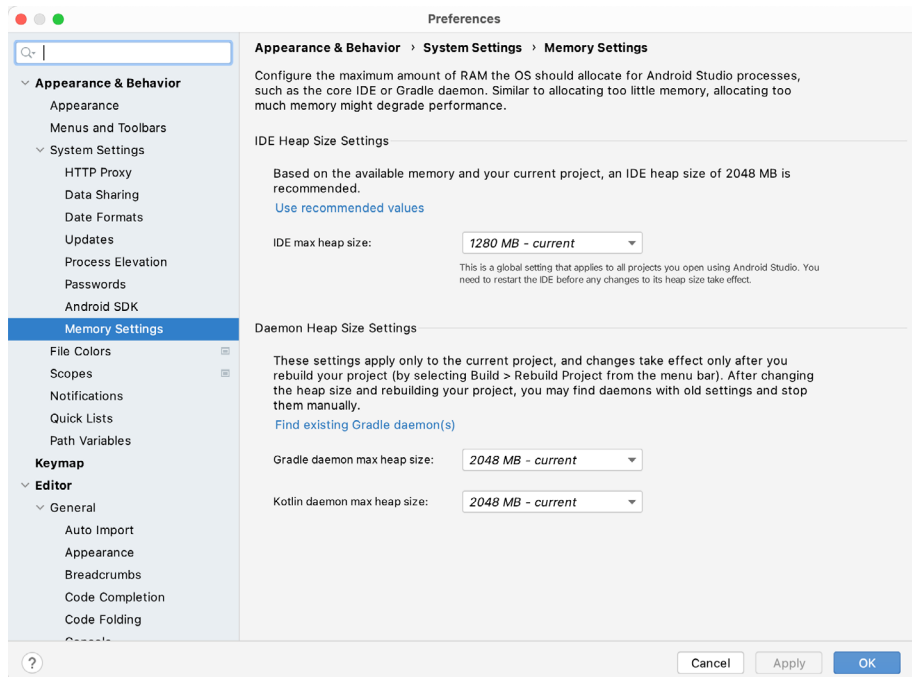


Figure 2-11

Setting up an Android Studio Development Environment

The IDE memory setting adjusts the memory allocated to Android Studio and applies regardless of the currently loaded project. When a project is built and run from within Android Studio, on the other hand, a number of background processes (referred to as daemons) perform the task of compiling and running the app. When compiling and running large and complex projects, build time may potentially be improved by adjusting the daemon heap settings. Unlike the IDE heap settings, these settings apply only to the current project and can only be accessed when a project is open in Android Studio.

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:

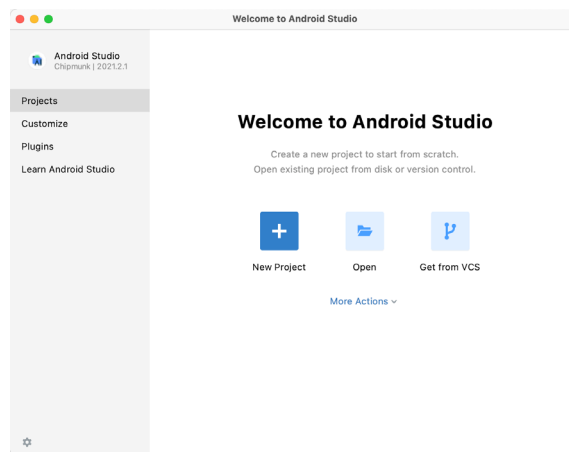


Figure 3-1

Once this window appears, Android Studio is ready for a new project to be created. To create the new project, 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 Activity*. The Empty Activity option creates a template user interface consisting of a single TextView object.

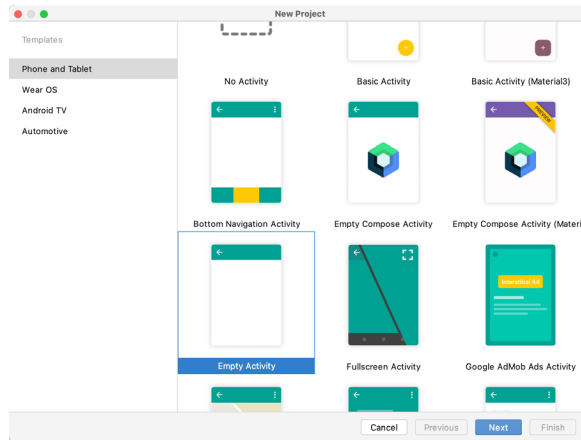


Figure 3-2

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

3.4 Defining the Project and SDK Settings

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

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

```
com.mycompany.androidsample
```

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

```
com.example.androidsample
```

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

Set the minimum SDK setting to API 26: Android 8.0 (Oreo). This is the 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:

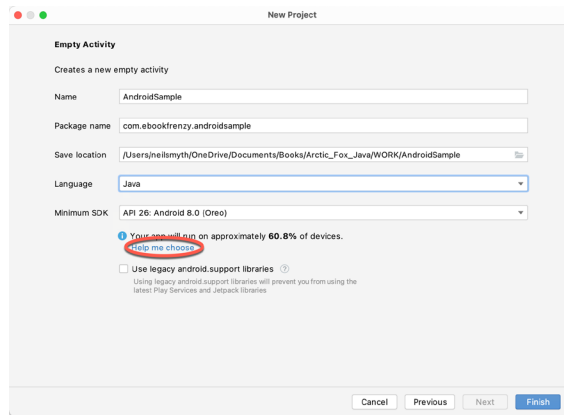


Figure 3-3

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.

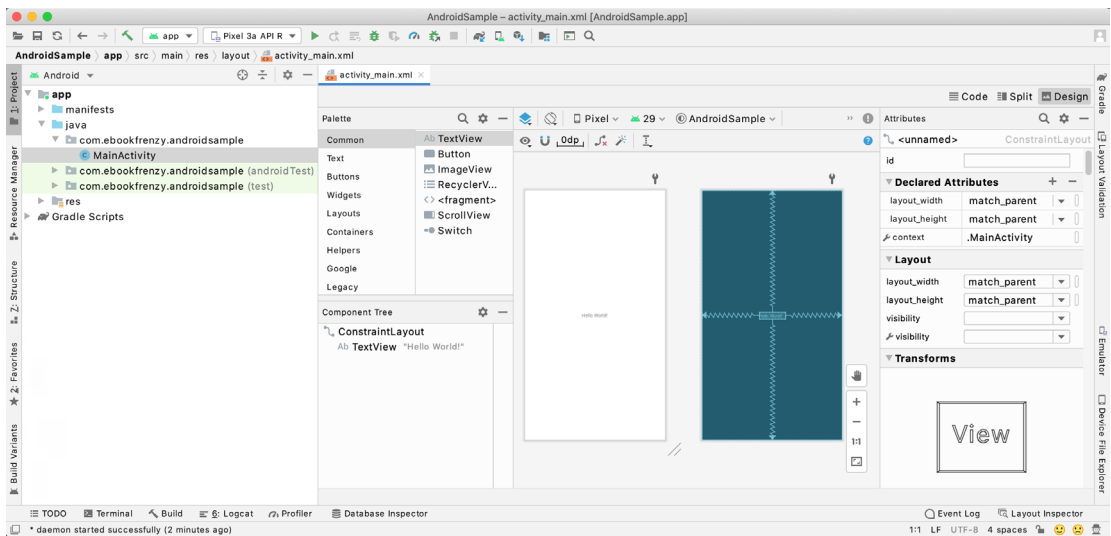


Figure 3-4

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:

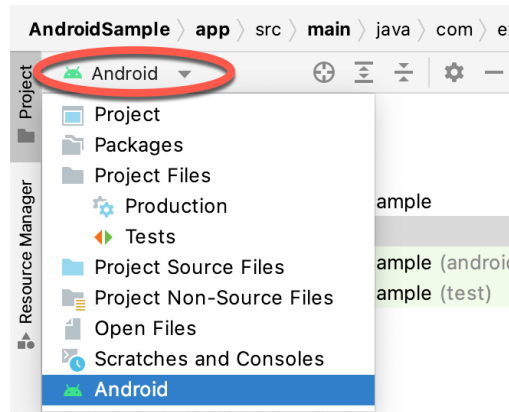


Figure 3-5

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:

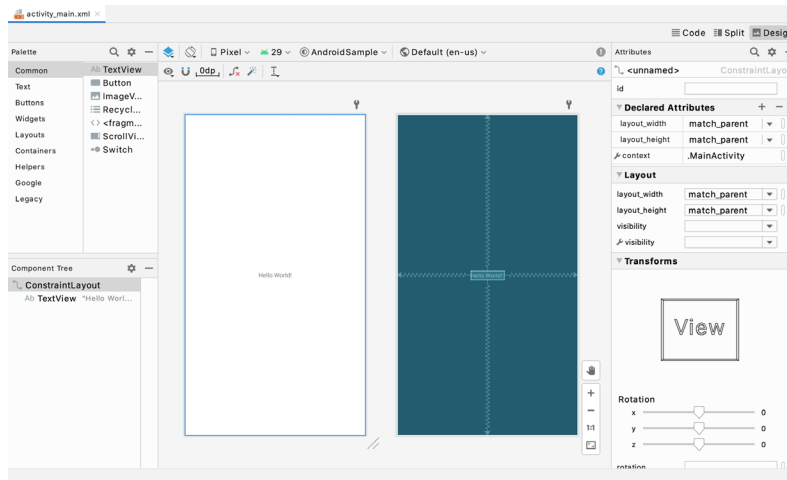




Figure 3-6

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 button () to turn Night mode on and off.

As can be seen in the device screen, the content layout already includes a label that displays a “Hello World!” message. Running down the left-hand side of the panel is a palette containing different categories of user interface components that may be used to construct a user interface, such as buttons, labels and text fields. It should be noted, however, that not all user interface components are obviously visible to the user. One such category

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

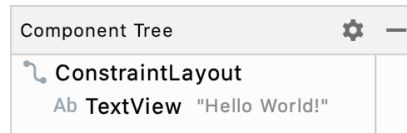


Figure 3-7

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

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



Figure 3-8

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

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

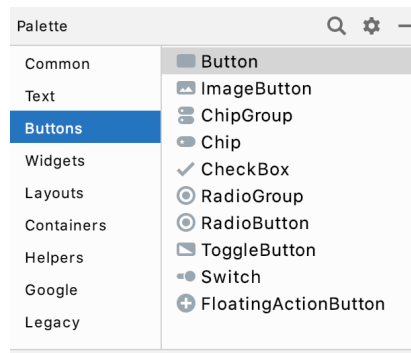


Figure 3-9

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

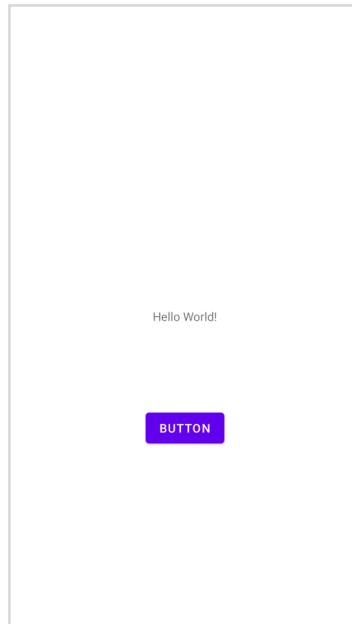


Figure 3-10

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

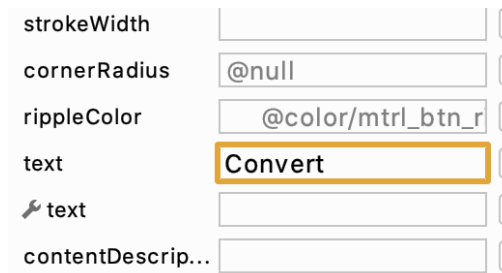


Figure 3-11

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

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

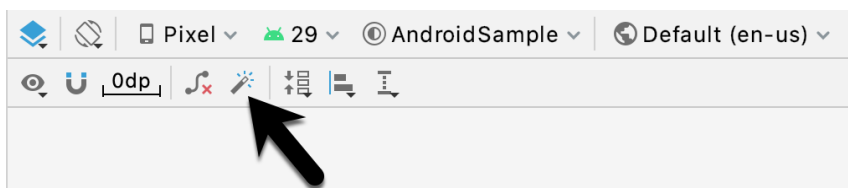


Figure 3-12

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

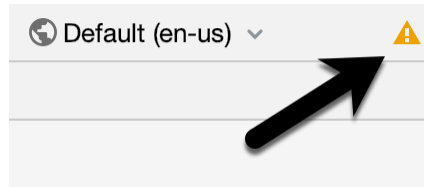


Figure 3-13

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

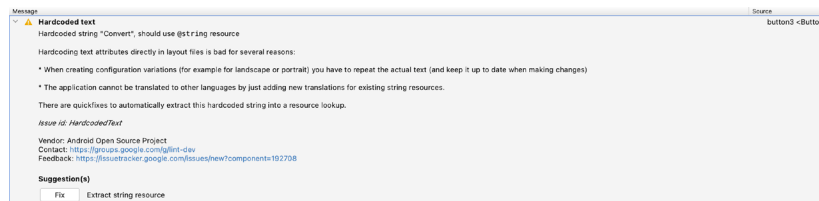


Figure 3-14

Currently, the only warning listed reads as follows:

Hardcoded string "Convert", should use @string resource

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

Click on the *Fix* button in the Issue Explanation panel to display the *Extract Resource* panel (Figure 3-15). Within this panel, change the resource name field to *convert_string* and leave the resource value set to *Convert* before clicking on the OK button.

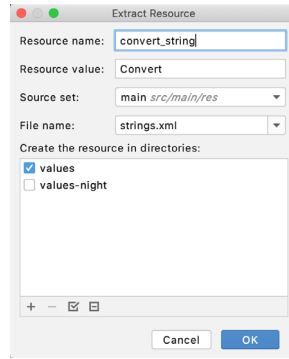


Figure 3-15

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 3-16:

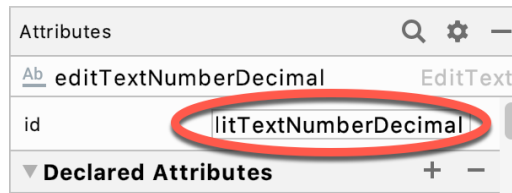


Figure 3-16

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:

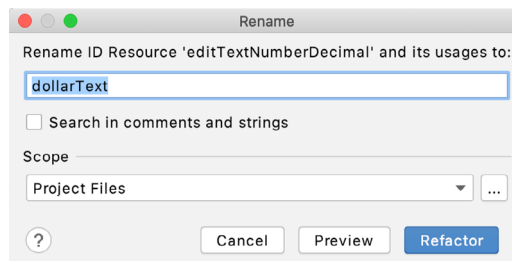


Figure 3-17

Add any missing layout constraints by clicking on the *Infer constraints* button. At this point the layout should resemble that shown in Figure 3-18:

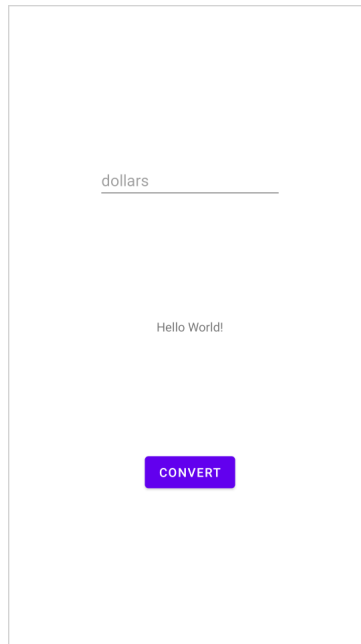


Figure 3-18

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 3-19 below:

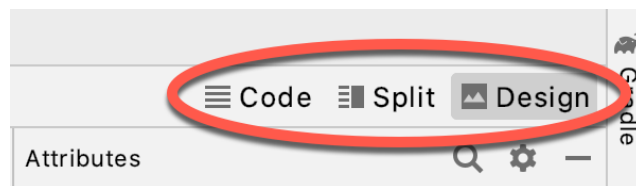


Figure 3-19

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 3-20:

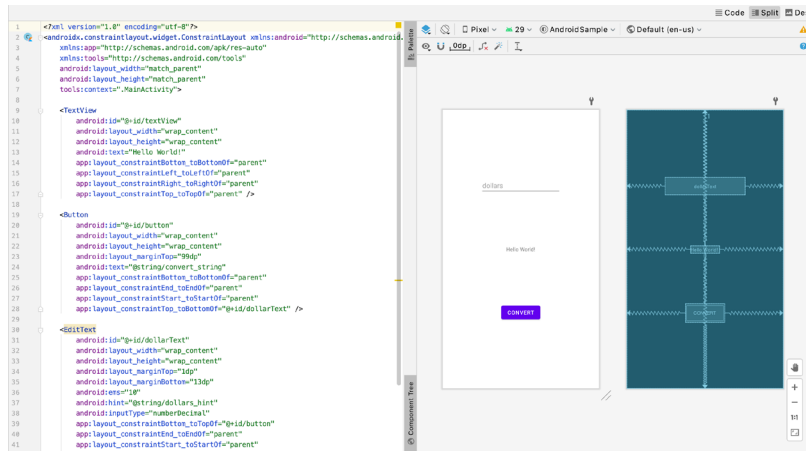


Figure 3-20

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:

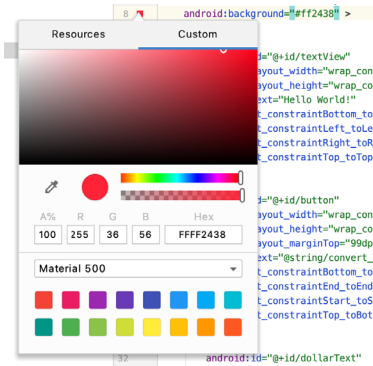


Figure 3-21

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

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

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

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

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

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

Resource strings may also be edited using the Android Studio Translations Editor. To open this editor, right-click on the *app* -> *res* -> *values* -> *strings.xml* file and select the *Open editor* menu option. This will display the Translation Editor in the main panel of the Android Studio window:

Creating an Example Android App in Android Studio

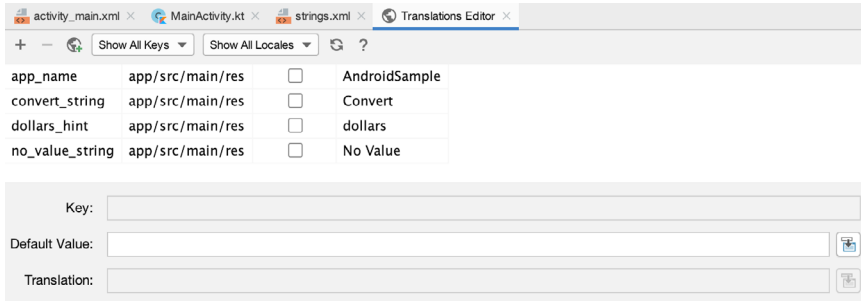


Figure 3-22

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:

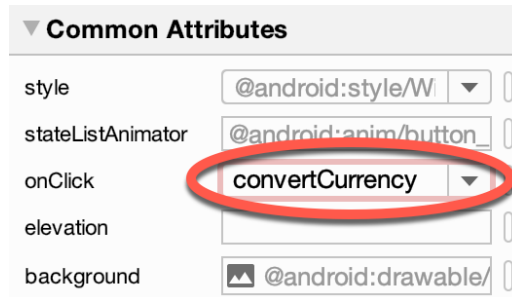


Figure 3-23

Note that the text field for the *onClick* property is now highlighted with a red border to warn us that the button has been configured to call a method which does not yet exist. To address this, double-click on the *MainActivity.java* file in the Project tool window (*app* -> *java* -> *<package name>* -> *MainActivity*) to load it into the code editor and add the code for the *convertCurrency* method to the class file so that it reads as follows, noting that it is also necessary to import some additional Android packages:

```
package com.ebookfrenzy.androidsample;

import androidx.appcompat.app.AppCompatActivity;

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

```

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, "%f", 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.

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 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:

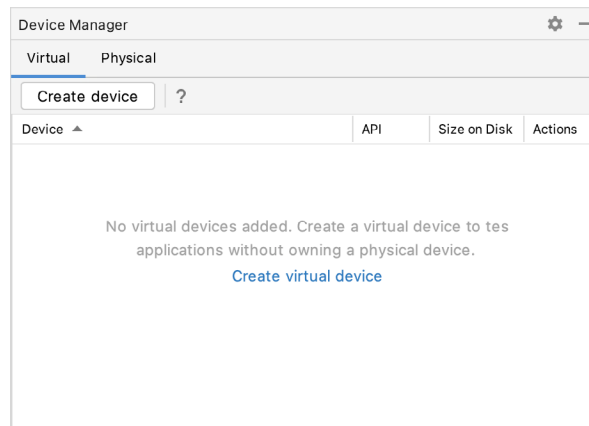


Figure 4-1

To add a new AVD, begin by making sure that the Virtual tab is selected before clicking on the *Create device*

Creating an Android Virtual Device (AVD) in Android Studio

button to open the *Virtual Device Configuration* dialog:

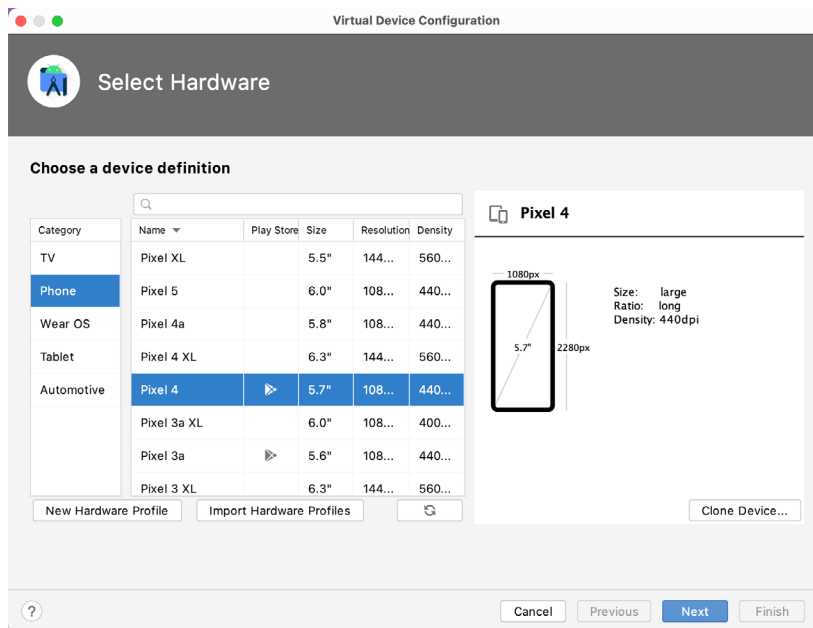


Figure 4-2

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 32*) 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:

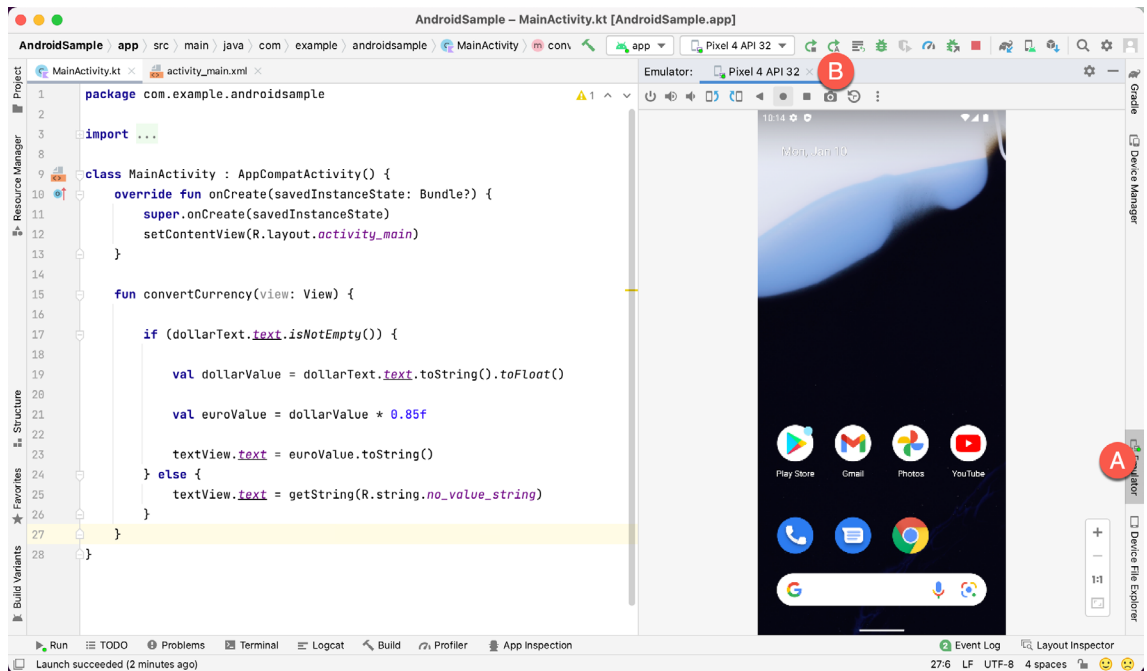


Figure 4-3

To hide and show the emulator tool window, click on the Emulator 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 4-4, for example, shows a tool window with two emulator sessions:

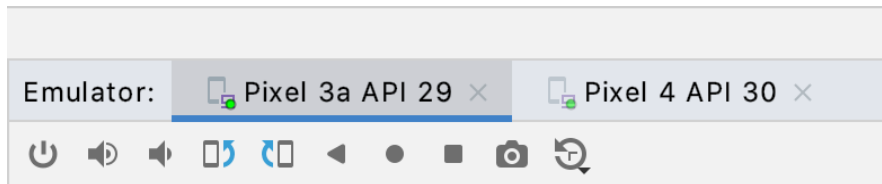


Figure 4-4

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

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 4-5 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

Creating an Android Virtual Device (AVD) in Android Studio

keyboard shortcut:

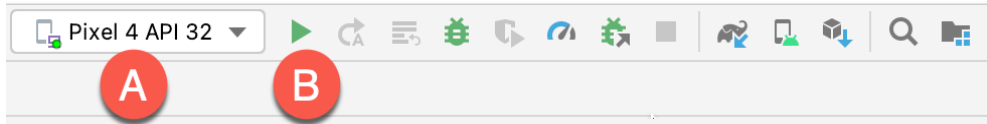


Figure 4-5

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:

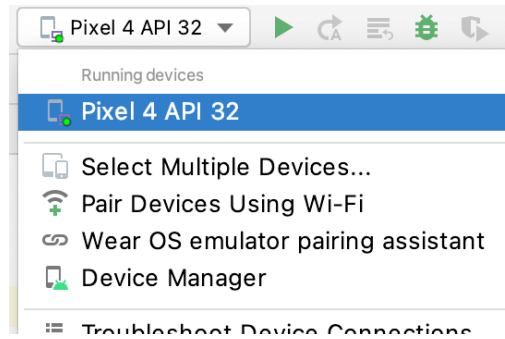


Figure 4-6

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*”):

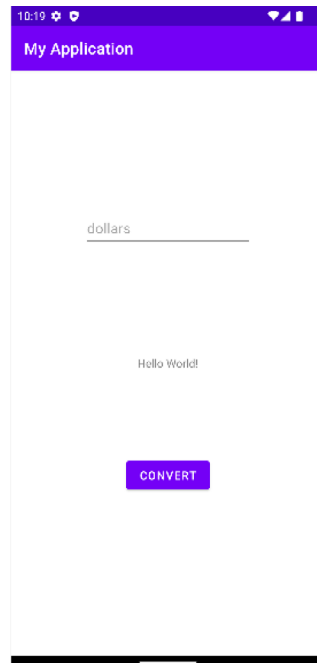


Figure 4-7

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 4-8 shows the Run tool window output from a typical successful application launch:



Figure 4-8

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 temperature conversion to verify that the app works as intended.

4.4 Running on Multiple Devices

The run menu shown in Figure 4-6 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 4-9 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 4-9

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 4-10:

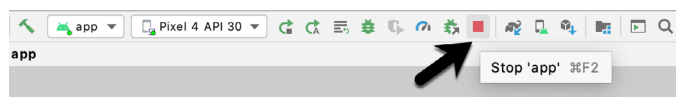


Figure 4-10

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

Creating an Android Virtual Device (AVD) in Android Studio

the stop button highlighted in Figure 4-11 below:



Figure 4-11

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 4-12 so that the screen background turns black:

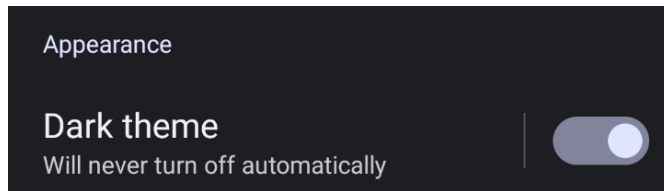


Figure 4-12

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 4-13:

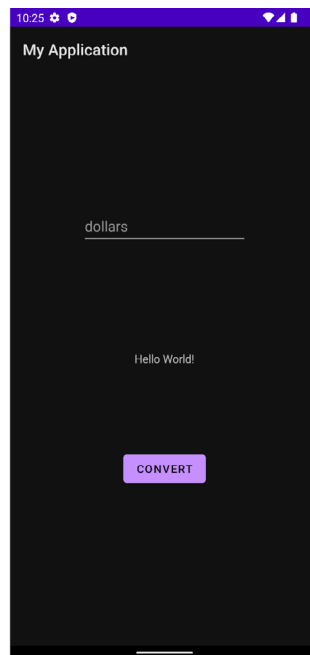


Figure 4-13

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:

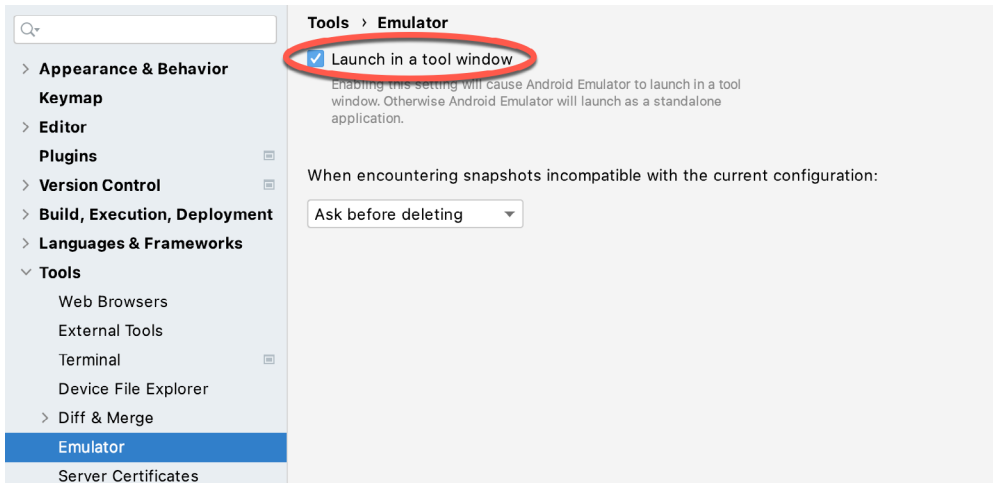


Figure 4-14

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 4-3 above.

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

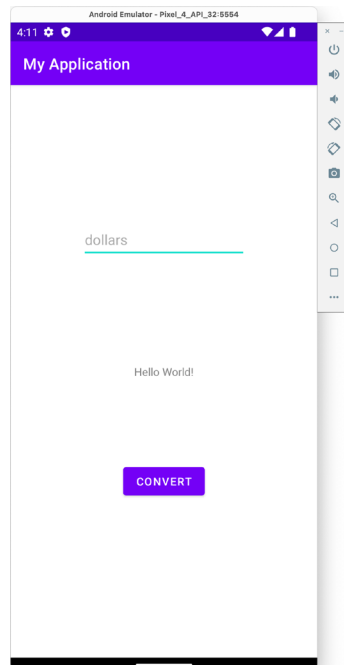


Figure 4-15

Creating an Android Virtual Device (AVD) in Android Studio

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:

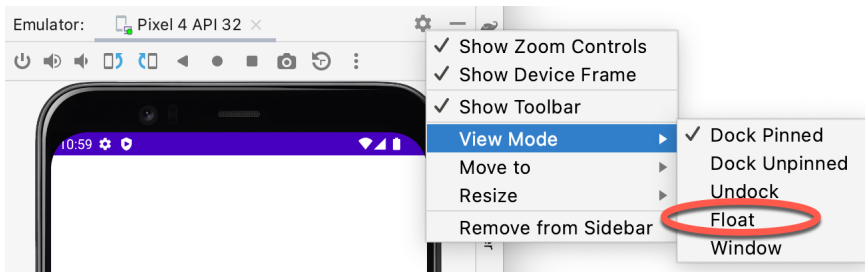


Figure 4-16

4.8 Enabling the Device Frame

The emulator can be configured to appear with (Figure 4-17) or without the device frame (Figure 4-15).

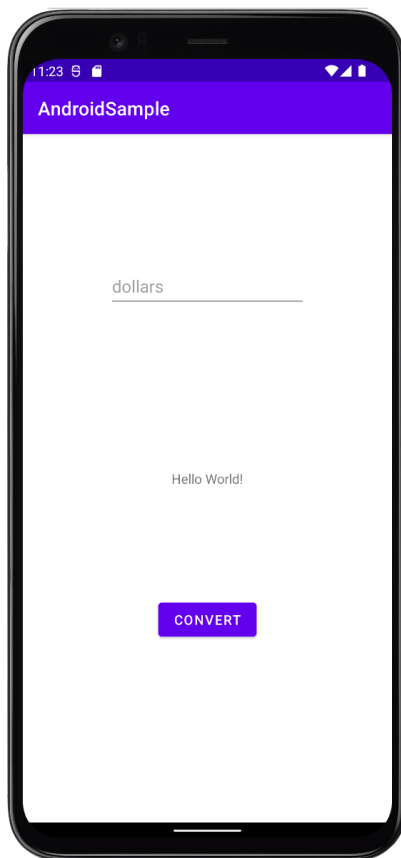


Figure 4-17

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:

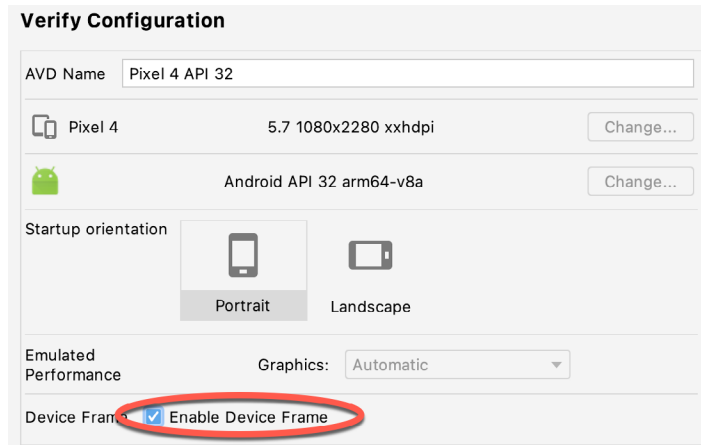


Figure 4-18

4.9 AVD Command-line Creation

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

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

1. Launch Android Studio and open the ComposeDemo project created earlier in the book.
2. Select the *File -> Settings...* menu option (*Android Studio -> Preferences...* on macOS).
3. Navigate to the Build, Execution, Deployment section and select the Gradle option listed under the Build Tools category.
4. Click on the *Gradle JDK* setting and make a note of the path for *Android Studio default JDK*:

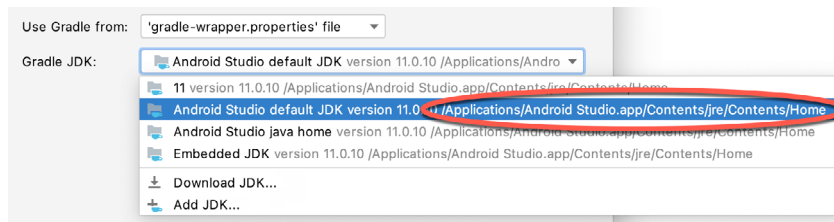


Figure 4-19

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

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

On macOS or Linux, execute the following command:

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

If you expect to use the *avdmanager* tool frequently, follow the environment variable steps for your operating

Creating an Android Virtual Device (AVD) in Android Studio

system outlined in the chapter entitled “*Setting up an Android Studio Development Environment*” to configure `JAVA_HOME` on a system-wide basis.

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

```
avdmanager list targets
```

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

```
Available Android targets:
```

```
-----
```

```
id: 1 or "android-29"
```

```
    Name: Android API 29
```

```
    Type: Platform
```

```
    API level: 29
```

```
    Revision: 1
```

```
-----
```

```
id: 2 or "android-26"
```

```
    Name: Android API 26
```

```
    Type: Platform
```

```
    API level: 26
```

```
    Revision: 1
```

The `avdmanager` tool also allows new AVD instances to be created from the command-line. For example, to create a new AVD named *myAVD* using the target ID for the Android API level 29 device using the x86 ABI, the following command may be used:

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

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

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

```
avdmanager list avd
```

```
Available Android Virtual Devices:
```

```
    Name: Pixel_XL_API_28_No_Play
```

```
    Device: pixel_xl (Google)
```

```
    Path: /Users/neilsmyth/.android/avd/Pixel_XL_API_28_No_Play.avd
```

```
    Target: Google APIs (Google Inc.)
```

```
        Based on: Android API 28 Tag/ABI: google_apis/x86
```

```
    Skin: pixel_xl_silver
```

```
    Sdcard: 512M
```

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


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

4.10 Android Virtual Device Configuration Files

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

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

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

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

4.11 Moving and Renaming an Android Virtual Device

The current name or the location of the AVD files may be altered from the command-line using the *avdmanager* tool's *move avd* argument. For example, to rename an AVD named Pixel4 to Pixel4a, the following command may be executed:

```
avdmanager move avd -n Pixel4 -r Pixel4a
```

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

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

For example, to move an AVD from its current file system location to */tmp/Pixel4Test*:

```
avdmanager move avd -n Pixel4 -p /tmp/Pixel4Test
```

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

4.12 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):

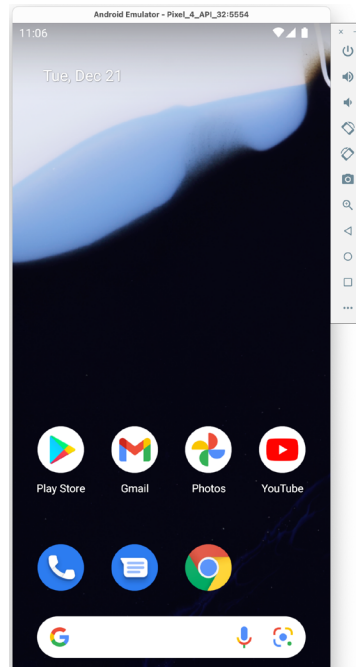


Figure 5-1

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.

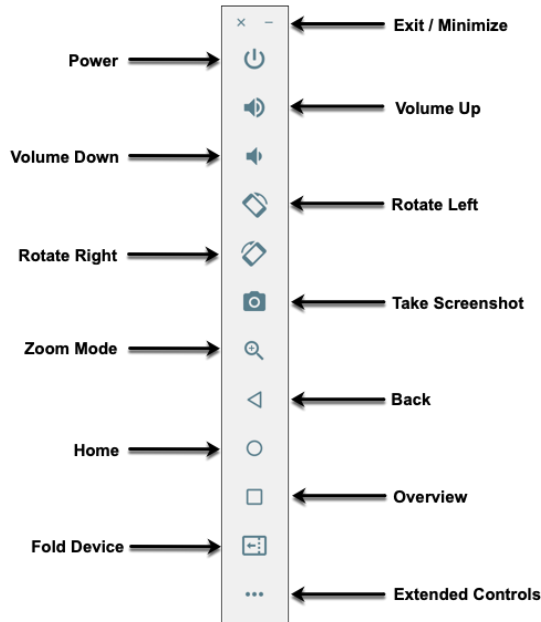


Figure 5-2

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:

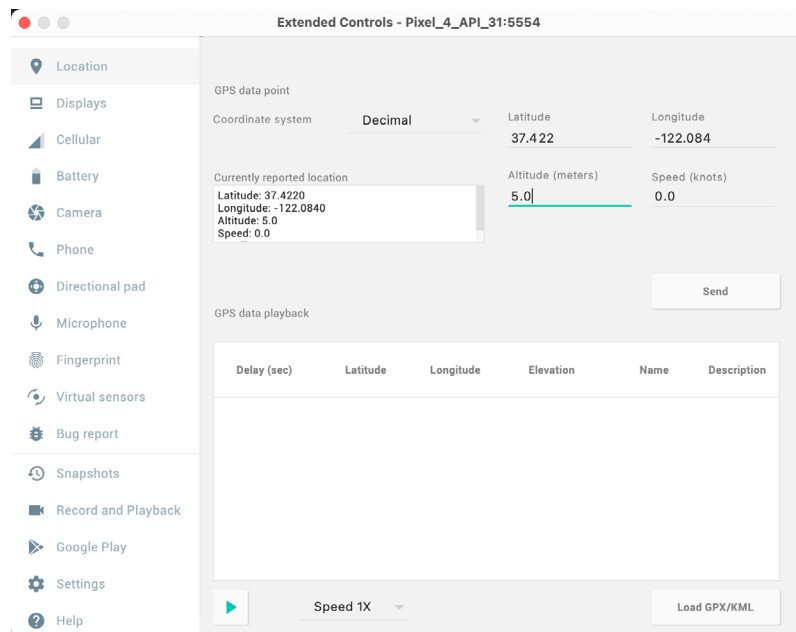


Figure 5-3

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:

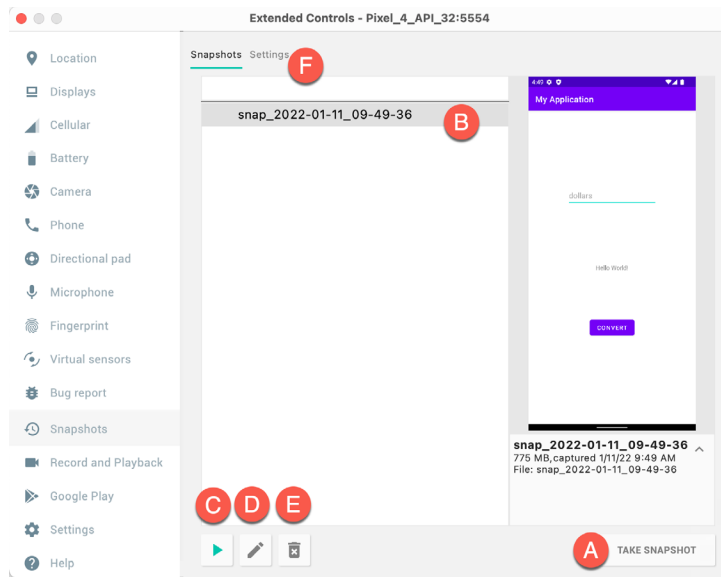


Figure 5-4

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

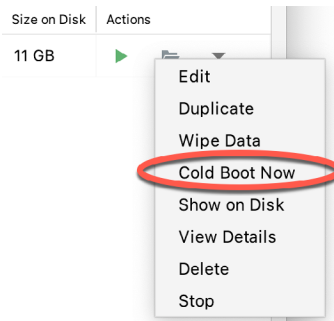


Figure 5-5

5.7 Configuring Fingerprint Emulation

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

Within the Security settings screen, select the *Use fingerprint* option. On the resulting information screen click on the *Next* button to proceed to the Fingerprint setup screen. Before fingerprint security can be enabled a backup screen unlocking method (such as a PIN) 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:

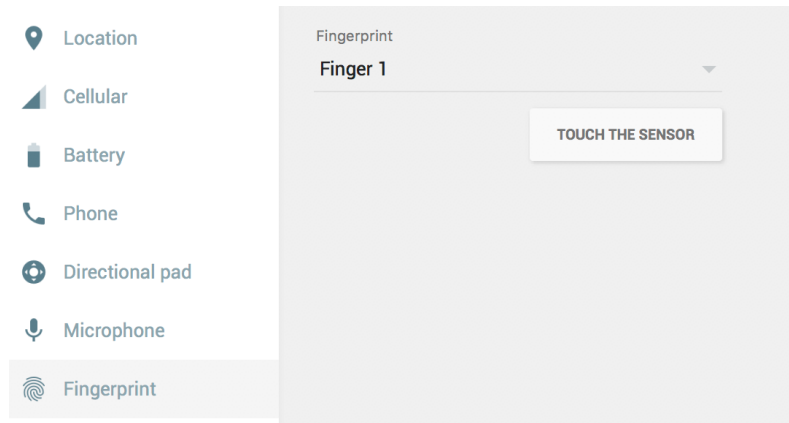


Figure 5-6

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

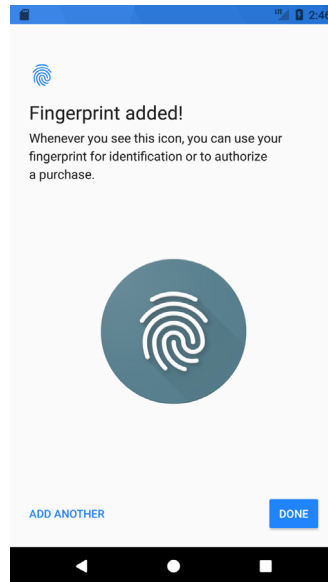


Figure 5-7

To add additional fingerprints click on the *Add Another* button and select another finger from the extended controls panel menu before clicking on the *Touch the Sensor* button once again.

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 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:

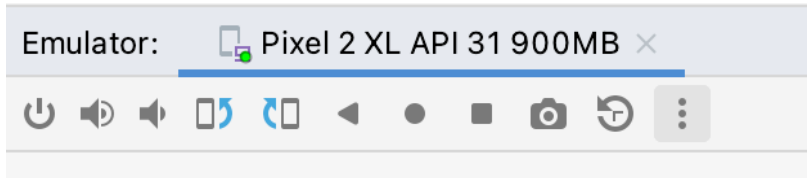


Figure 5-8

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 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 by-pass the welcome screen next time it is launched, automatically opening the previously active project.

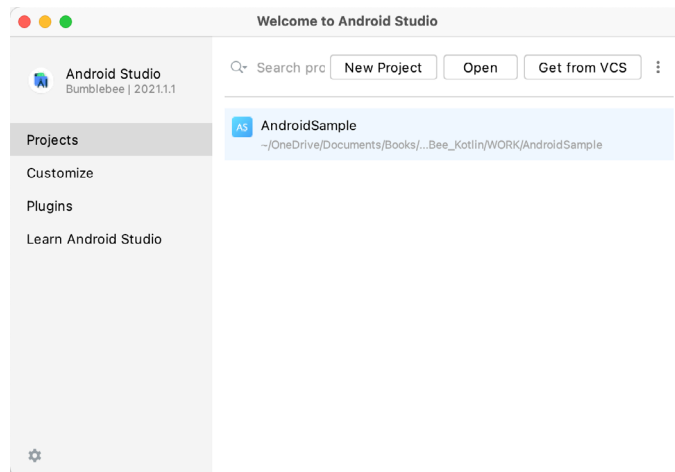


Figure 6-1

In addition to a list of recent projects, the welcome screen provides a range of 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 via the *Plugins* option.

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

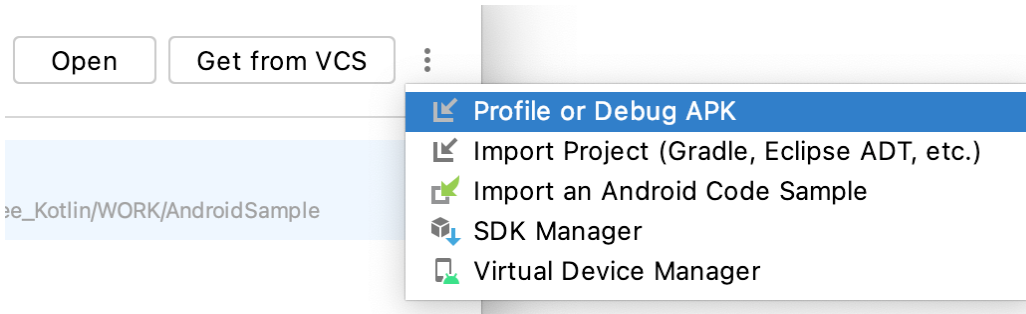


Figure 6-2

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.

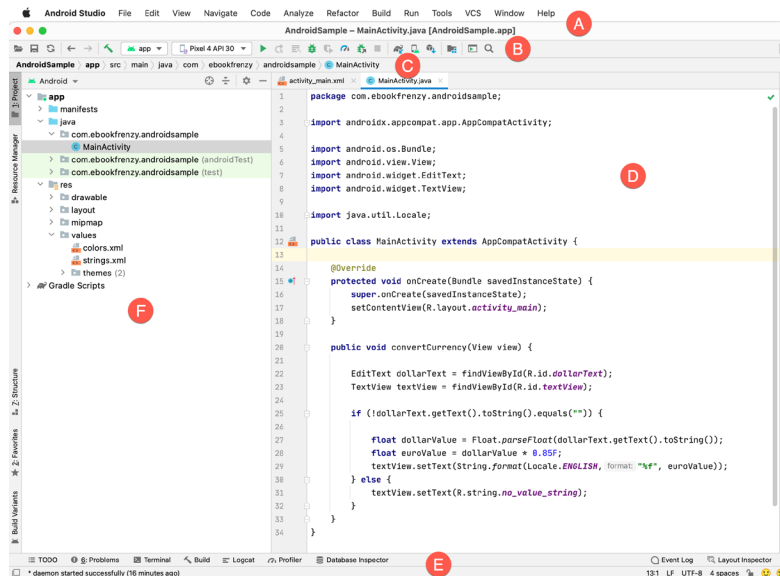


Figure 6-3

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

C – Navigation Bar – The navigation bar provides a convenient way to move around the files and folders that make up the project. Clicking on an element in the navigation bar will drop down a menu listing the 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. What gets displayed in this location, however, is subject to context. When editing code, for example, the code editor will appear. When working on a user interface layout file, on the other hand, the user interface Layout Editor tool will appear. When multiple files are open, each file is represented by a tab located along the top edge of the editor as shown in Figure 6-4.

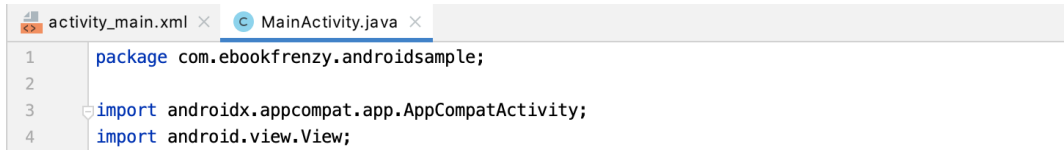


Figure 6-4

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 provide a description of that field. Many fields are interactive, allowing the user to click to perform tasks or obtain more detailed status information.

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

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

6.3 The Tool Windows

In addition to the project view tool window, Android Studio also includes a number of other windows which, when enabled, are displayed along the bottom and sides of the main window. The tool window quick access menu can be accessed by hovering the mouse pointer over the button located in the far left-hand corner of the status bar (Figure 6-5) without clicking the mouse button.

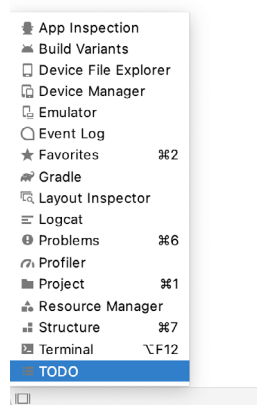


Figure 6-5

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

Alternatively, a set of *tool window bars* can be displayed by clicking on the quick access menu icon in the status

A Tour of the Android Studio User Interface

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.

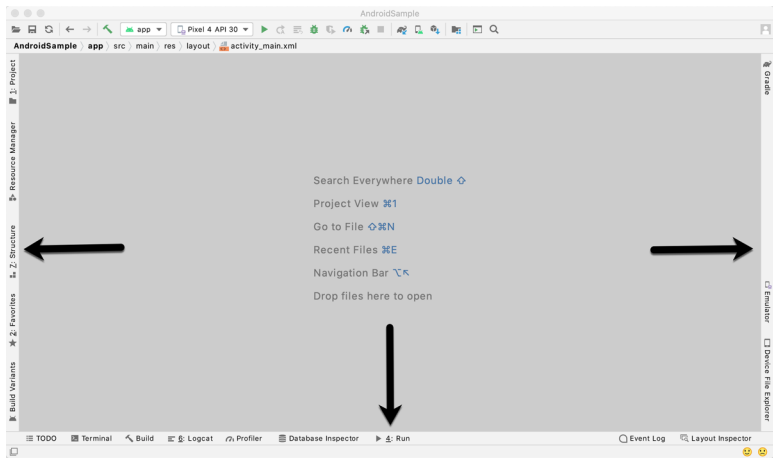


Figure 6-6

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 tool bars.

Each tool window has its own toolbar along the top edge. The buttons within these toolbars vary from one tool to the next, though all tool windows contain a settings option, represented by the cog icon, which allows various aspects of the window to be changed. Figure 6-7 shows the settings menu for the project view tool window. Options are available, for example, to undock a window and to allow it to float outside of the boundaries of the Android Studio main window and to move and resize the tool panel.

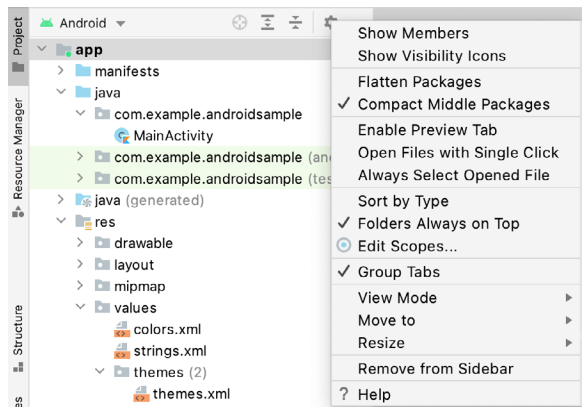


Figure 6-7

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 tool bar and items matching the search highlighted.

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

- **App Inspector** - 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*”.
- **Event Log** – The event log window displays messages relating to events and activities performed within Android Studio. The successful build of a project, for example, or the fact that an application is now running will be reported within this tool window.
- **Favorites** – A variety of project items can be added to the favorites list. Right-clicking on a file in the project view, for example, provides access to an *Add to Favorites* menu option. Similarly, a method in a source file can be added as a favorite by right-clicking on it in the Structure tool window. Anything added to a Favorites list can be accessed through this Favorites tool window.
- **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*.

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:

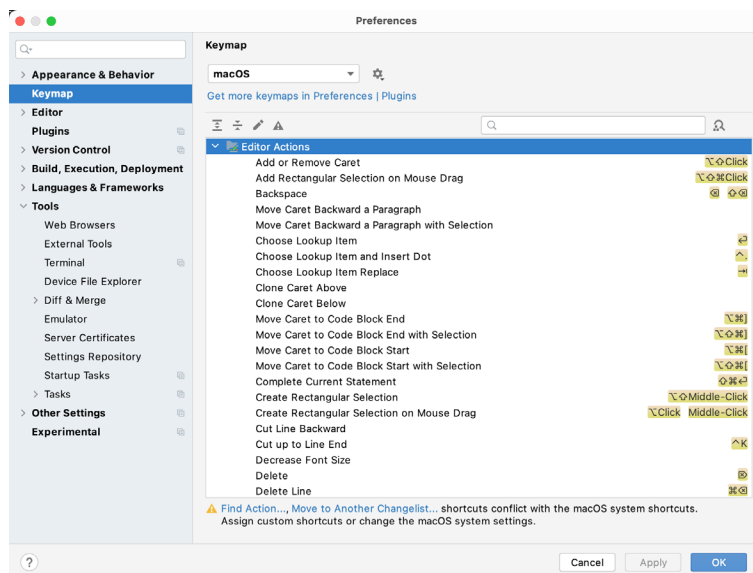


Figure 6-8

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).

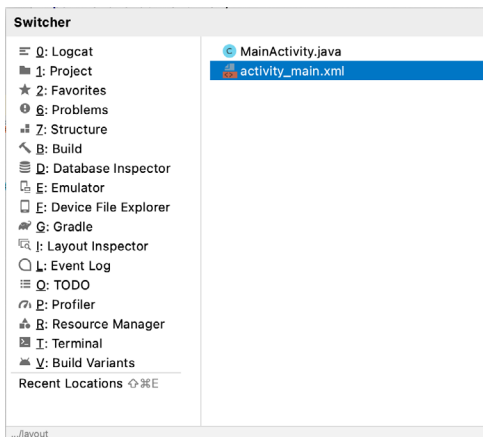


Figure 6-9

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, alternatively, the keyboard arrow keys used to scroll through the file name and tool window options. Pressing the Enter key will select the currently highlighted item.

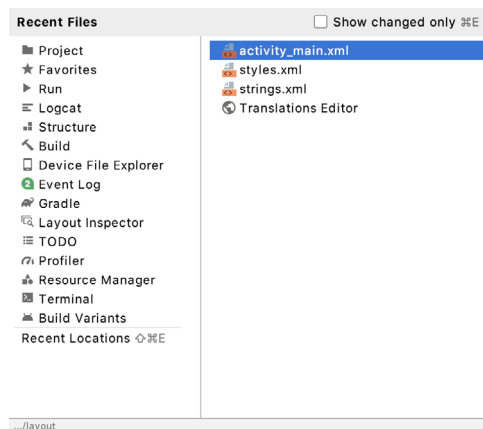


Figure 6-10

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

A Tour of the Android Studio User Interface

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:

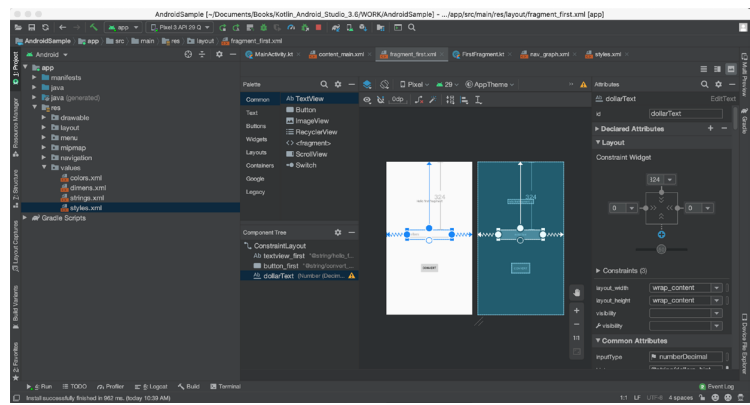


Figure 6-11

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:

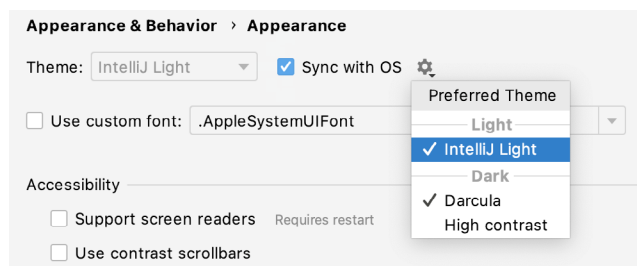


Figure 6-12

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

7. Testing Android Studio Apps on a Physical Android Device

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

Communication with both AVD instances and connected Android devices is handled by the *Android Debug Bridge (ADB)*. In this chapter, we explain how to configure the adb environment to enable application testing on an Android device with macOS, Windows, and Linux-based systems.

7.1 An Overview of the Android Debug Bridge (ADB)

The primary purpose of the ADB is to facilitate interaction between a development system, in this case, Android Studio, and both AVD emulators and Android devices to run and debug applications. ADB allows you to connect to devices either over a WiFi network or directly using a USB cable.

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

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

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

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

7.2 Enabling USB Debugging ADB on Android Devices

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

1. Open the Settings app on the device and select the *About tablet* or *About phone* option (on some versions of Android this can be found on the *System* page of the Settings app).
2. On the *About* screen, scroll down to the *Build number* field (Figure 7-1) and tap on it seven times until a message appears indicating that developer mode has been enabled. If the Build number is not listed on the About screen it may be available via the *Software information* option. Alternatively, unfold the Advanced section of the list if available.

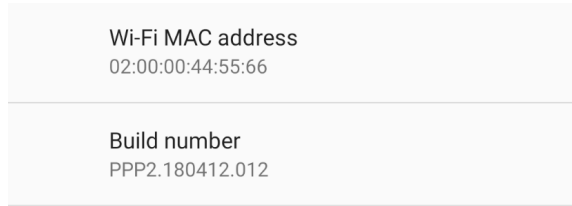


Figure 7-1

3. Return to the main Settings screen and note the appearance of a new option titled Developer options (on newer versions of Android this option is listed on the System settings screen). Select this option and on the resulting screen, locate the USB debugging option as illustrated in Figure 7-2:

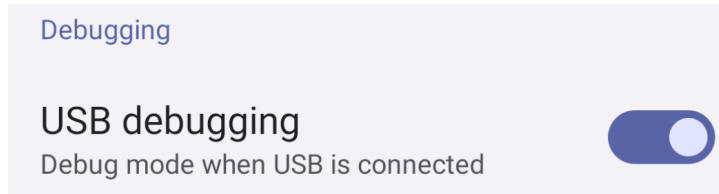


Figure 7-2

4. Enable the USB debugging option and tap the Allow button when confirmation is requested.

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

7.2.1 macOS ADB Configuration

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

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

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

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

If the device is listed as *offline*, go to the Android device and check for the presence of the dialog shown in Figure 7-3 seeking permission to *Allow USB debugging*. Enable the checkbox next to the option that reads *Always allow from this computer*, before clicking on OK.

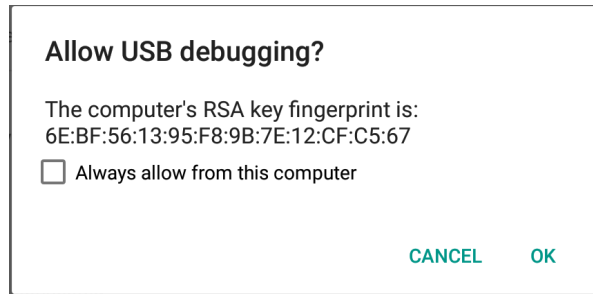


Figure 7-3

Repeating the `adb devices` command should now list the device as being available:

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

If the device is not listed, try logging out and then back into the macOS desktop and, if the problem persists, rebooting the system.

7.2.2 Windows ADB Configuration

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

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

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

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

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

```
adb devices
```

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

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

If the device is listed as *offline* or *unauthorized*, go to the device display and check for the dialog shown in Figure 7-3 seeking permission to *Allow USB debugging*. Enable the checkbox next to the option that reads *Always allow from this computer*, before clicking on *OK*. Repeating the `adb devices` command should now list the device as being ready:

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

If the device is not listed, execute the following commands to restart the ADB server:

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

Testing Android Studio Apps on a Physical Android Device

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

```
android update adb
```

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

7.2.3 Linux adb Configuration

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

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

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

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

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

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

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

If the device is listed as *offline* or *unauthorized*, go to the Android device and check for the dialog shown in Figure 7-3 seeking permission to *Allow USB debugging*.

7.3 Resolving USB Connection Issues

If you are unable to successfully connect to the device using the above steps, display the run target menu (Figure 7-4) and select the *Troubleshoot Device Connections* option:

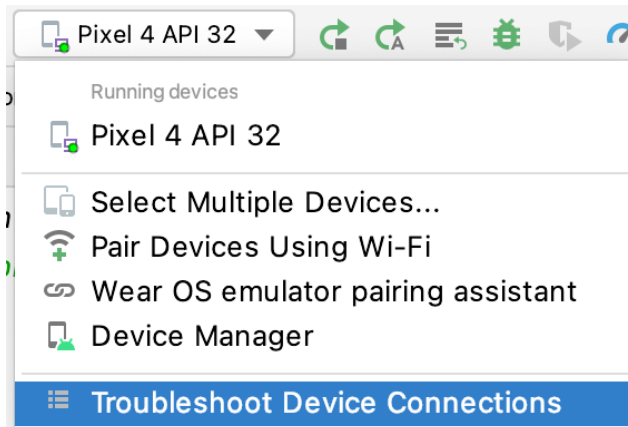


Figure 7-4

The connection assistant will scan for devices and report problems and possible solutions.

7.4 Enabling Wireless Debugging on Android Devices

Follow steps 1 through 3 from section 7.2 above, this time enabling the Wireless Debugging option as shown in Figure 7-5:

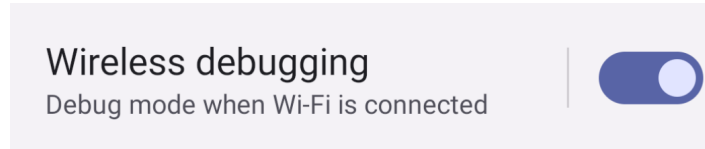


Figure 7-5

Next, tap the above Wireless debugging entry to display the screen shown in Figure 7-6:

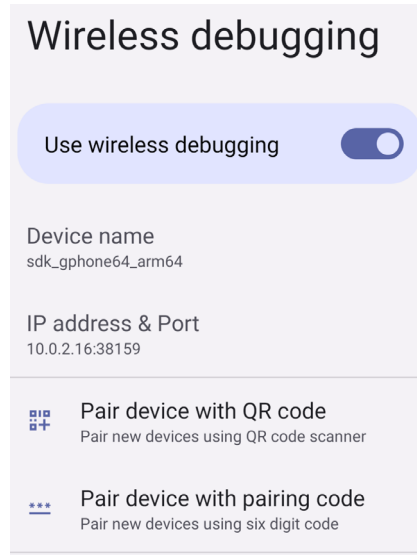


Figure 7-6

If the device you are using has a camera, select *Pair device with QR code*, otherwise select the *Pair device with pairing code* option. Depending on your selection, the Settings app will either start a camera session or display a pairing code as shown in Figure 7-7:

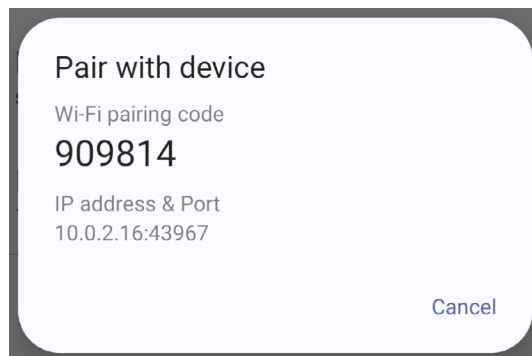


Figure 7-7

With an option selected, return to Android Studio and select the *Pair Devices Using WiFi* option from the run target menu as illustrated in Figure 7-8:

Testing Android Studio Apps on a Physical Android Device

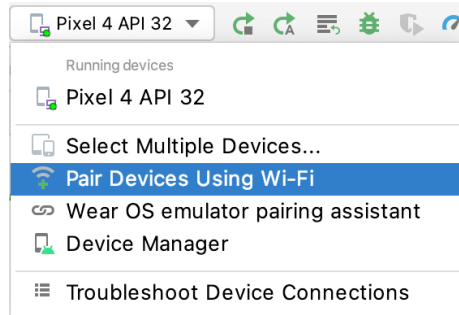


Figure 7-8

In the pairing dialog, select either *Pair using QR code* or *Pair using pairing code* depending on your previous selection in the Settings app on the device:

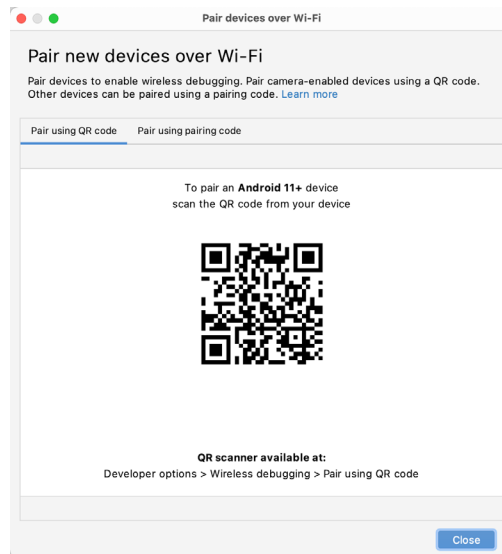


Figure 7-9

Either scan the QR code using the Android device or enter the pairing code displayed on the device screen into the Android Studio dialog (Figure 7-10) to complete the pairing process:

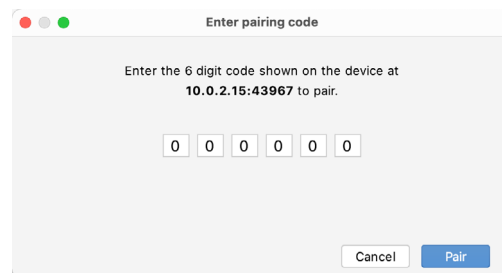


Figure 7-10

If the pairing process fails, try rebooting both the development system and Android device and try again.

7.5 Testing the adb Connection

Assuming that the adb configuration has been successful on your chosen development platform, the next step is to try running the test application created in the chapter entitled “*Creating an Example Android App in Android Studio*” on the device. Launch Android Studio, open the AndroidSample project, and verify that the device appears in the device selection menu as highlighted in Figure 7-11:

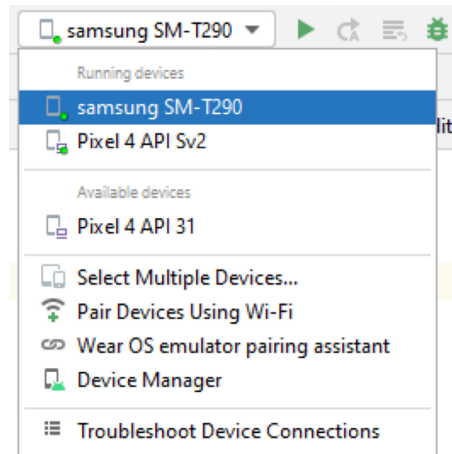


Figure 7-11

Select the device from the list and click on the run button (the green arrow button located immediately to the right of the device menu) to install and run the app.

7.6 Summary

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

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

8. The Basics of the Android Studio Code Editor

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

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

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

8.1 The Android Studio Editor

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

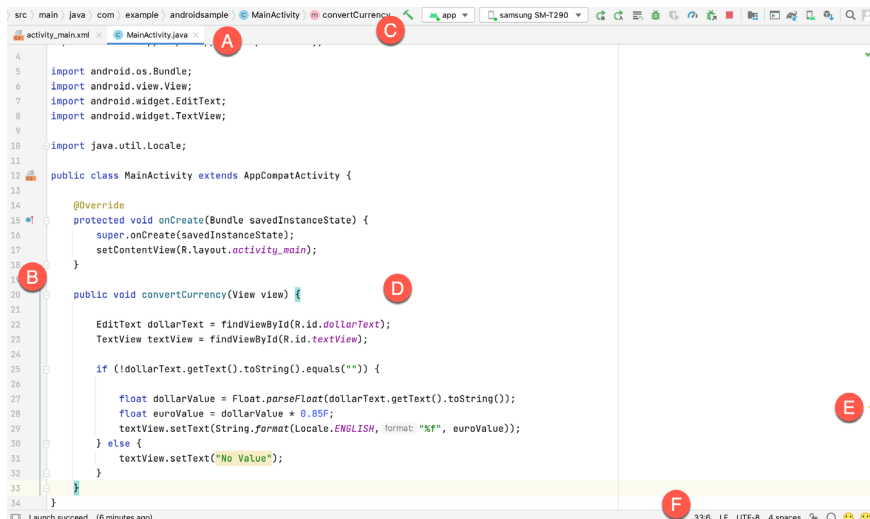


Figure 8-1

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

A – Document Tabs – Android Studio is capable of holding multiple files open for editing at any one time. As each file is opened, it is assigned a document tab displaying the file name in the tab bar located along the top

edge of the editor window. A small drop-down menu will appear in the far right-hand corner of the tab bar when there is insufficient room to display all of the tabs. Clicking on this menu will drop down a list of additional open files. A wavy red line underneath a file name in a tab indicates that the code in the file contains one or more errors that need to be addressed before the project can be compiled and run.

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

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

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

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

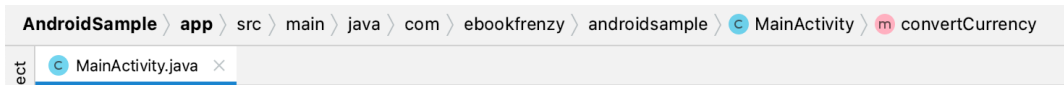


Figure 8-2

Double-clicking an element within the bar will move the cursor to the corresponding location within the code file. For example, double-clicking on the `convertCurrency` entry will move the cursor to the top of the `convertCurrency` method within the source code. Similarly clicking on the `MainActivity` entry will drop down a list of available code navigation points for selection:

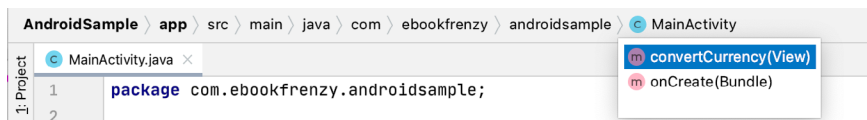


Figure 8-3

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

E – The Validation and Marker Sidebar – Android Studio incorporates a feature referred to as “on-the-fly code analysis”. What this essentially means is that as you are typing code, the editor is analyzing the code to check for warnings and syntax errors. The indicators at the top of the validation sidebar will update in real-time to indicate the number of errors and warnings found as code is added. Clicking on this indicator will display a popup containing a summary of the issues found with the code in the editor as illustrated in Figure 8-4:

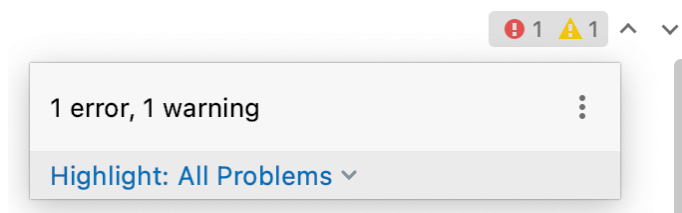


Figure 8-4

The up and down arrows may be used to move between the error locations within the code. A green check mark indicates that no warnings or errors have been detected.

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

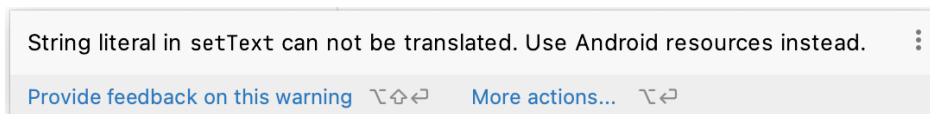


Figure 8-5

Hovering the mouse pointer over a marker for a line of code which is currently scrolled out of the viewing area of the editor will display a “lens” overlay containing the block of code where the problem is located (Figure 8-6) allowing it to be viewed without the necessity to scroll to that location in the editor:

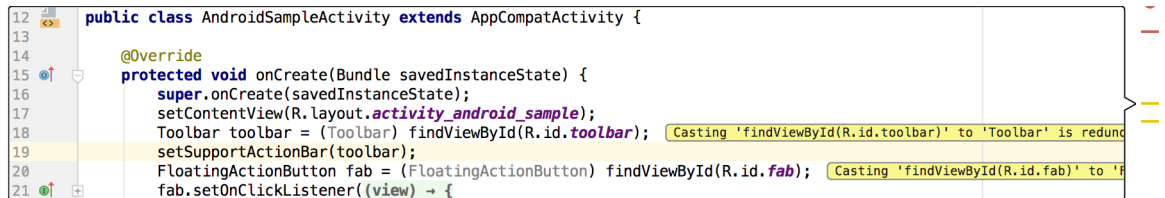


Figure 8-6

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

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

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

8.2 Splitting the Editor Window

By default, the editor will display a single panel showing the content of the currently selected file. A particularly useful feature when working simultaneously with multiple source code files is the ability to split the editor into multiple panes. To split the editor, right-click on a file tab within the editor window and select either the Split Vertically or Split Horizontally menu option. Figure 8-7, for example, shows the splitter in action with the editor

split into three panels:

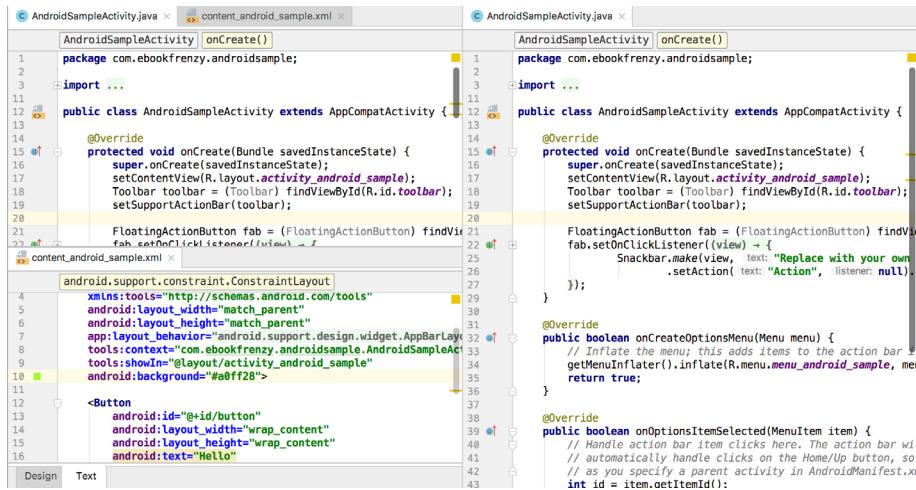


Figure 8-7

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

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

8.3 Code Completion

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

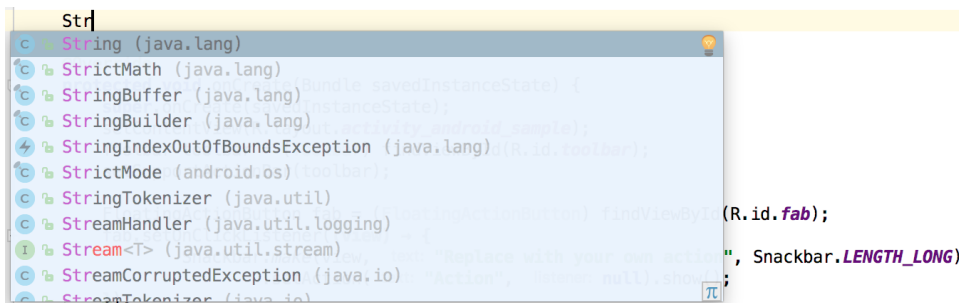


Figure 8-8

If none of the auto completion suggestions are correct, simply keep typing and the editor will continue to refine the suggestions where appropriate. To accept the top most suggestion, simply press the Enter or Tab key on the keyboard. To select a different suggestion, use the arrow keys to move up and down the list, once again using the Enter or Tab key to select the highlighted item.

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

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

Code completion can be a matter of personal preference for many programmers. In recognition of this fact, Android Studio provides a high level of control over the auto completion settings. These can be viewed and modified by selecting the *File -> Settings...* menu option (or *Android Studio -> Preferences...* on macOS) and choosing *Editor -> General -> Code Completion* from the settings panel as shown in Figure 8-9:

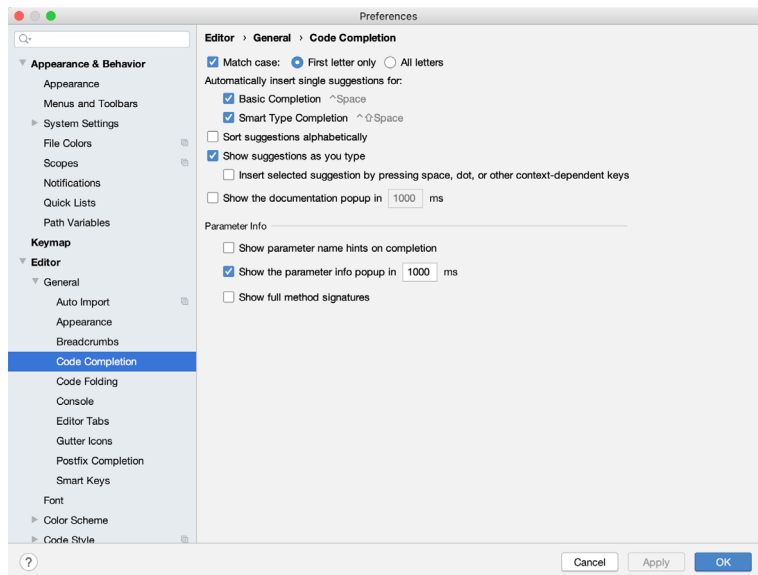


Figure 8-9

8.4 Statement Completion

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

```
myMethod()
```

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

```
myMethod() {  
  
}
```

8.5 Parameter Information

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

```
String myButtonText = mystring.replaceAll();
```

@NonNull String regex, @NonNull String replacement

Figure 8-10

8.6 Parameter Name Hints

The code editor may be configured to display parameter name hints within method calls. Figure 8-11, for example, highlights the parameter name hints within the calls to the *make()* and *setAction()* methods of the *Snackbar* class:

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

Figure 8-11

The settings for this mode may be configured by selecting the *File -> Settings* menu (*Android Studio -> Preferences* on macOS) option followed by *Editor -> Inlay Hints -> Java* in the left-hand panel. On the resulting screen, select the *Parameter Hints* item from the list and enable or disable the *Show parameter hints* option. To adjust the hint settings, click on the *Exclude list...* link and make any necessary adjustments.

8.7 Code Generation

In addition to completing code as it is typed the editor can, under certain conditions, also generate code for you. The list of available code generation options shown in Figure 8-12 can be accessed using the Alt-Insert (Cmd-N on macOS) keyboard shortcut when the cursor is at the location in the file where the code is to be generated.

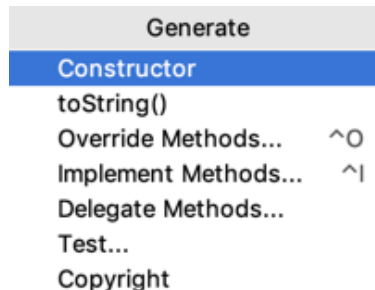


Figure 8-12

For the purposes of an example, consider a situation where we want to be notified when an Activity in our project is about to be destroyed by the operating system. As will be outlined in a later chapter of this book, this can be achieved by overriding the *onStop()* lifecycle method of the Activity superclass. To have Android Studio generate a stub method for this, simply select the *Override Methods...* option from the code generation list and

select the `onStop()` method from the resulting list of available methods:

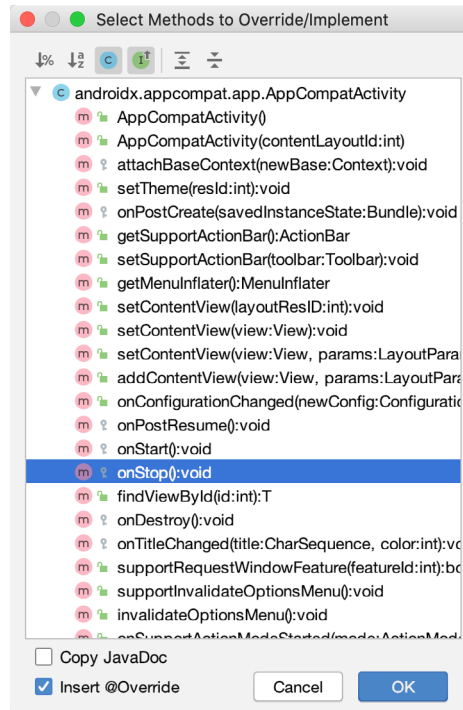


Figure 8-13

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

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

8.8 Code Folding

Once a source code file reaches a certain size, even the most carefully formatted and well organized code can become overwhelming and difficult to navigate. Android Studio takes the view that it is not always necessary to have the content of every code block visible at all times. Code navigation can be made easier through the use of the code folding feature of the Android Studio editor. Code folding is controlled using markers appearing in the editor gutter at the beginning and end of each block of code in a source file. Figure 8-14, for example, highlights the start and end markers for a method declaration which is not currently folded:

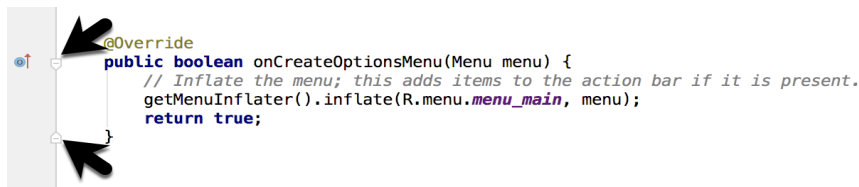


Figure 8-14

Clicking on either of these markers will fold the statement such that only the signature line is visible as shown

in Figure 8-15:

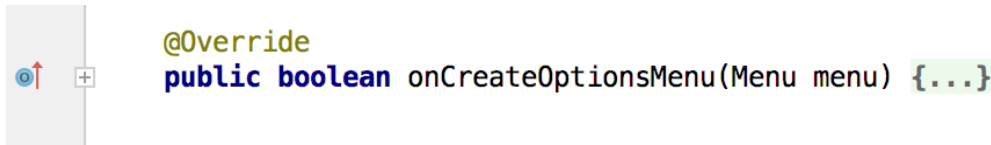


Figure 8-15

To unfold a collapsed section of code, simply click on the '+' marker in the editor gutter. To see the hidden code without unfolding it, hover the mouse pointer over the "{...}" indicator as shown in Figure 8-16. The editor will then display the lens overlay containing the folded code block:

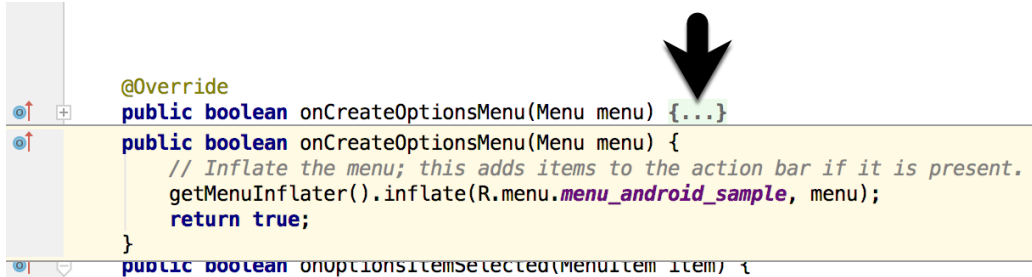


Figure 8-16

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

By default, the Android Studio editor will automatically fold some code when a source file is opened. To configure the conditions under which this happens, select File -> Settings... (Android Studio -> Preferences... on macOS) and choose the Editor -> General -> Code Folding entry in the resulting settings panel (Figure 8-17):

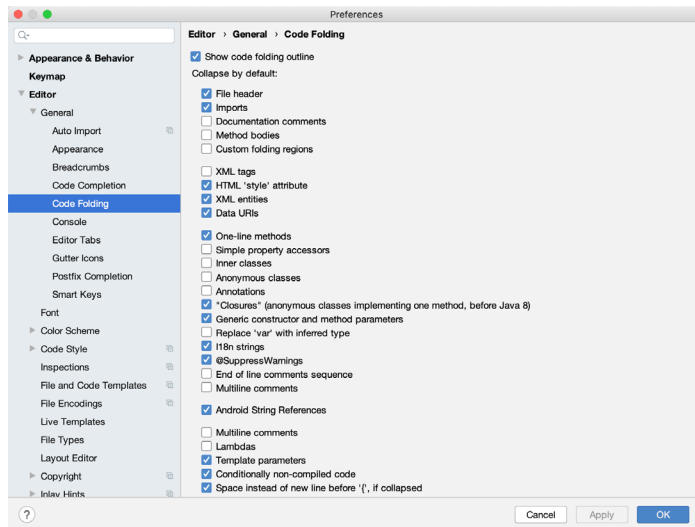


Figure 8-17

8.9 Quick Documentation Lookup

Context sensitive Java and Android documentation can be accessed by placing the cursor over the declaration for which documentation is required and pressing the Ctrl-Q keyboard shortcut (Ctrl-J on macOS). This will

display a popup containing the relevant reference documentation for the item. Figure 8-18, for example, shows the documentation for the Android `FloatingActionButton` class.



Figure 8-18

Once displayed, the documentation popup can be moved around the screen as needed.

8.10 Code Reformatting

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

To reformat source code, press the `Ctrl-Alt-L` (`Cmd-Opt-L` on macOS) keyboard shortcut sequence. To display the Reformat Code dialog (Figure 8-19) use the `Ctrl-Alt-Shift-L` (`Cmd-Opt-Shift-L` on macOS). This dialog provides the option to reformat only the currently selected code, the entire source file currently active in the editor or only code that has changed as the result of a source code control update.

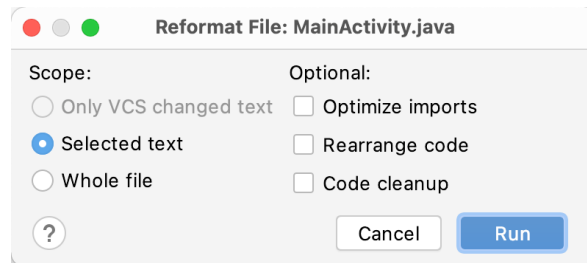


Figure 8-19

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

8.11 Finding Sample Code

The Android Studio editor provides a way to access sample code relating to the currently highlighted entry within the code listing. This feature can be useful for learning how a particular Android class or method is used. To find sample code, highlight a method or class name in the editor, right-click on it and select the *Find Sample Code* menu option. The Find Sample Code panel (Figure 8-20) will appear beneath the editor with a list of matching samples. Selecting a sample from the list will load the corresponding code into the right-hand panel:

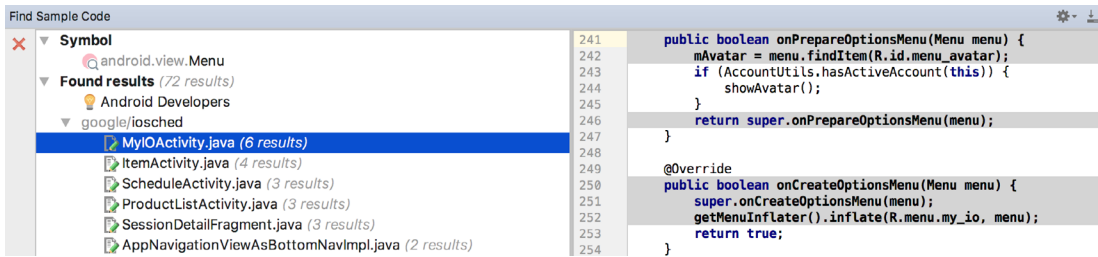


Figure 8-20

8.12 Live Templates

As you write Android code you will find that there are common constructs that are used frequently. For example, a common requirement is to display a popup message to the user using the Android Toast class. Live templates are a collection of common code constructs that can be entered into the editor by typing the initial characters followed by a special key (set to the Tab key by default) to insert template code. To experience this in action, type toast in the code editor followed by the Tab key and Android Studio will insert the following code at the cursor position ready for editing:

```
Toast.makeText(, "", Toast.LENGTH_SHORT).show();
```

To list and edit existing templates, change the special key, or add your own templates, open the Preferences dialog and select Live Templates from the Editor section of the left-hand navigation panel:

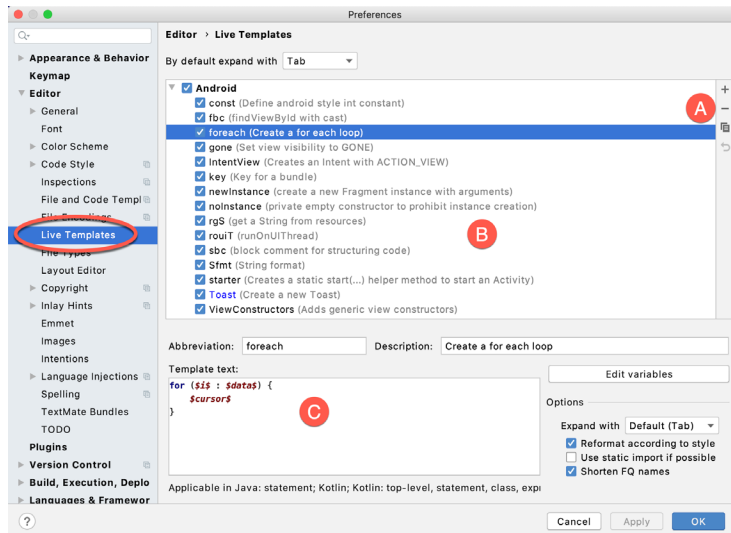


Figure 8-21

Add, remove, duplicate or reset templates using the buttons marked A in Figure 8-21 above. To modify a template, select it from the list (B) and change the settings in the panel marked C.

8.13 Summary

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

9. An Overview of the Android Architecture

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

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

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

9.1 The Android Software Stack

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

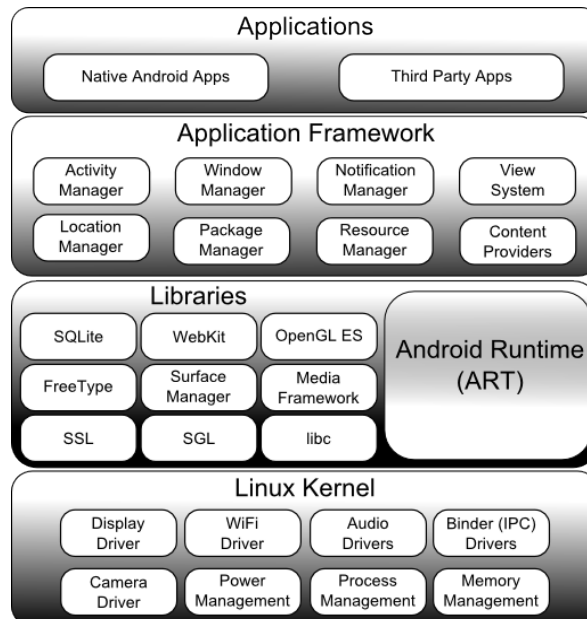


Figure 9-1

9.2 The Linux Kernel

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

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

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

9.3 Android Runtime – ART

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

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

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

9.4 Android Libraries

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

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

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

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

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

9.4.1 C/C++ Libraries

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

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

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

9.5 Application Framework

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

found and reused by other applications.

The Android framework includes the following key services:

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

9.6 Applications

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

9.7 Summary

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

10. The Anatomy of an Android Application

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

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

10.1 Android Activities

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

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

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

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

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

10.2 Android Fragments

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

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

10.3 Android Intents

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

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

10.4 Broadcast Intents

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

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

10.5 Broadcast Receivers

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

10.6 Android Services

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

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

Example situations where a Service might be a practical solution include, as previously mentioned, the streaming of audio that should continue when the application is no longer active, or a stock market tracking application

that needs to notify the user when a share hits a specified price.

10.7 Content Providers

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

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

10.8 The Application Manifest

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

10.9 Application Resources

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

10.10 Application Context

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

10.11 Summary

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

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

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

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

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

11. An Overview of Android View Binding

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

11.1 Find View by Id

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

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

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

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

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

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

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

11.2 View Binding

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

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

Android Studio Chipmunk is inconsistent in using view bindings within project templates. The Empty Activity template used when we created the *AndroidSample* project, for example, does not use view bindings. The Basic Activity template, on the other hand, is implemented using view binding. If you use a template that does not use view binding, it is important to know how to migrate that project away from synthetic properties.

11.3 Converting the AndroidSample project

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

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

11.4 Enabling View Binding

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

```
plugins {  
    id 'com.android.application'  
    .  
    .  
    android {  
  
        buildFeatures {  
            viewBinding true  
        }  
        .  
        .  
    }  
}
```

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

11.5 Using View Binding

The first step in this process is to “inflate” the view binding class so that we can access the root view within the layout. This root view will then be used as the content view for the layout.

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

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

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

```
.  
.br/>import android.widget.EditText;  
import android.widget.TextView;
```

```

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

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

```

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

```

public void convertCurrency(View view) {

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

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

        Float dollarValue = Float.valueOf(binding.dollarText.getText().
toString());
        Float euroValue = dollarValue * 0.85F;
        binding.textView.setText(euroValue.toString());
    } else {
        binding.textView.setText(R.string.no_value_string);
    }
}

```

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

11.6 Choosing an Option

Their failure to adopt view bindings in the Empty Activity project template notwithstanding, Google strongly recommends the use of view binding wherever possible. In fact, support for synthetic properties is now deprecated and will likely be removed in a future release of Android Studio. When developing your own projects, therefore, view binding should probably be used.

11.7 View Binding in the Book Examples

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

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

11.8 Migrating a Project to View Binding

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

1. Edit the module level Gradle build script file listed in the Project tool window as *Gradle Scripts* -> *build.gradle* (Module: *<project name>.app*) where *<project name>* is the name of the project (for example *AndroidSample*).
2. Locate the *android* section of the file and add an entry to enable the *viewBinding* property as follows:

```
android {
```

```
    buildFeatures {
        viewBinding true
    }
}
```

3. Click on the *Sync Now* link at the top of the editor to resynchronize the project with these new build settings.
4. Edit the *MainActivity.java* file and modify it to read as follows (where *<reverse domain>* represents the domain name used when the project was created and *<project name>* is replaced by the lowercase name of the project, for example *androidsample*) and *<binding name>* is the name of the binding for the corresponding layout resource file (for example the binding for *activity_main.xml* is *ActivityMainBinding*).

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

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

```
public class MainActivity extends AppCompatActivity {
```

```
    private <binding name> binding;

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

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

11.9 Summary

Before the introduction of Android Studio 3.6, access to layout views from within the code of an app involved the use of the *findViewById()* method. An alternative is now available in the form of view bindings. View bindings consist of classes which are automatically generated by Android Studio for each XML layout file. These classes contain bindings to each of the views in the corresponding layout, providing a safer option to that offered by the *findViewById()* method. As of Android Studio Chipmunk, however, view bindings are not enabled by default in some project templates and additional steps are required to manually enable and configure support within each project module.

12. Understanding Android Application and Activity Lifecycles

In earlier chapters we have learned that Android applications run within processes and that they are comprised of multiple components in the form of activities, services and broadcast receivers. The goal of this chapter is to expand on this knowledge by looking at the lifecycle of applications and activities within the Android runtime system.

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

An important factor in developing Android applications, therefore, is to gain an understanding of both the application and activity lifecycle management models of Android, and the ways in which an application can react to the state changes that are likely to be imposed upon it during its execution lifetime.

12.1 Android Applications and Resource Management

Each running Android application is viewed by the operating system as a separate process. If the system identifies that resources on the device are reaching capacity it will take steps to terminate processes to free up memory.

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

12.2 Android Process States

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

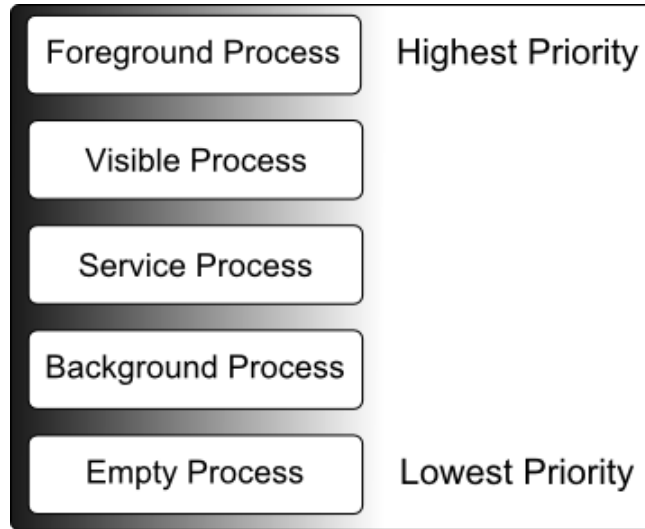


Figure 12-1

12.2.1 Foreground Process

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

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

12.2.2 Visible Process

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

12.2.3 Service Process

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

12.2.4 Background Process

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

12.2.5 Empty Process

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

12.3 Inter-Process Dependencies

The situation with regard to determining the highest priority process is slightly more complex than outlined in the preceding section for the simple reason that processes can often be inter-dependent. As such, when making a determination as to the priority of a process, the Android system will also take into consideration whether the process is in some way serving another process of higher priority (for example, a service process acting as the content provider for a foreground process). As a basic rule, the Android documentation states that a process can never be ranked lower than another process that it is currently serving.

12.4 The Activity Lifecycle

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

12.5 The Activity Stack

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

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

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

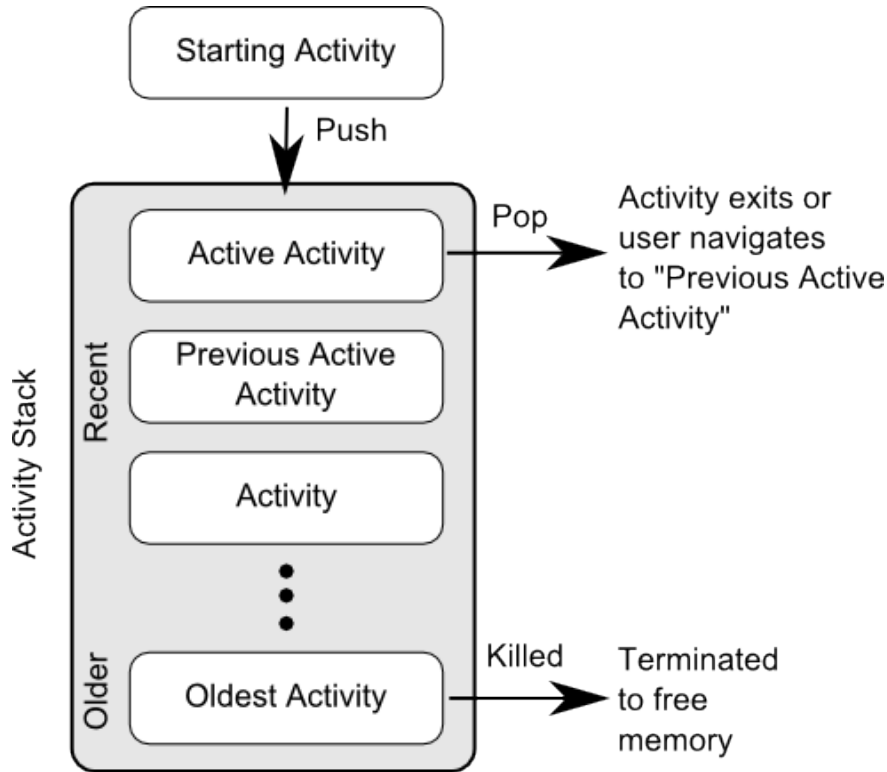


Figure 12-2

12.6 Activity States

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

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

12.7 Configuration Changes

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

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

12.8 Handling State Change

If nothing else, it should be clear from this chapter that an application and, by definition, the components contained therein will transition through many states during the course of its lifespan. Of particular importance is the fact that these state changes (up to and including complete termination) are imposed upon the application by the Android runtime subject to the actions of the user and the availability of resources on the device.

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

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

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

12.9 Summary

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

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

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

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

13. Handling Android Activity State Changes

Based on the information outlined in the chapter entitled “*Understanding Android Application and Activity Lifecycles*” it is now evident that the activities and fragments that make up an application pass through a variety of different states during the course of the application’s lifespan. The change from one state to the other is imposed by the Android runtime system and is, therefore, largely beyond the control of the activity itself. That does not, however, mean that the app cannot react to those changes and take appropriate actions.

The primary objective of this chapter is to provide a high-level overview of the ways in which an activity may be notified of a state change and to outline the areas where it is advisable to save or restore state information. Having covered this information, the chapter will then touch briefly on the subject of *activity lifetimes*.

13.1 New vs. Old Lifecycle Techniques

Up until recently, there was a standard way to build lifecycle awareness into an app. This is the approach covered in this chapter and involves implementing a set of methods (one for each lifecycle state) within an activity or fragment instance that get called by the operating system when the lifecycle status of that object changes. This approach has remained unchanged since the early years of the Android operating system and, while still a viable option today, it does have some limitations which will be explained later in this chapter.

With the introduction of the lifecycle classes with the Jetpack Android Architecture Components, a better approach to lifecycle handling is now available. This modern approach to lifecycle management (together with the Jetpack components and architecture guidelines) will be covered in detail in later chapters. It is still important, however, to understand the traditional lifecycle methods for a couple of reasons. First, as an Android developer you will not be completely insulated from the traditional lifecycle methods and will still make use of some of them. More importantly, understanding the older way of handling lifecycles will provide a good knowledge foundation on which to begin learning the new approach later in the book.

13.2 The Activity and Fragment Classes

With few exceptions, activities and fragments in an application are created as subclasses of the Android `AppCompatActivity` class and `Fragment` classes respectively.

Consider, for example, the *AndroidSample* project created in “*Creating an Example Android App in Android Studio*” and subsequently converted to use view binding. Load this project into the Android Studio environment and locate the *MainActivity.java* file (located in `app -> java -> com -> <your domain> -> androidsample`). Having located the file, double-click on it to load it into the editor where it should read as follows:

```
package com.example.androidsample;

import androidx.appcompat.app.AppCompatActivity;
import android.view.View;
import android.os.Bundle;

import java.util.Locale;
```

Handling Android Activity State Changes

```
import com.example.androidsample.databinding.ActivityMainBinding;

public class MainActivity extends AppCompatActivity {

    private ActivityMainBinding binding;

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

        public void convertCurrency(View view) {

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

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

When the project was created, we instructed Android Studio also to create an initial activity named *MainActivity.java*. As is evident from the above code, the *MainActivity* class is a subclass of the *AppCompatActivity* class.

A review of the reference documentation for the *AppCompatActivity* class would reveal that it is itself a subclass of the *Activity* class. This can be verified within the Android Studio editor using the *Hierarchy* tool window. With the *MainActivity.java* file loaded into the editor, click on *AppCompatActivity* in the *class* declaration line and press the *Ctrl-H* keyboard shortcut. The hierarchy tool window will subsequently appear displaying the class hierarchy for the selected class. As illustrated in Figure 13-1, *AppCompatActivity* is clearly subclassed from the *FragmentActivity* class which is itself ultimately a subclass of the *Activity* class:

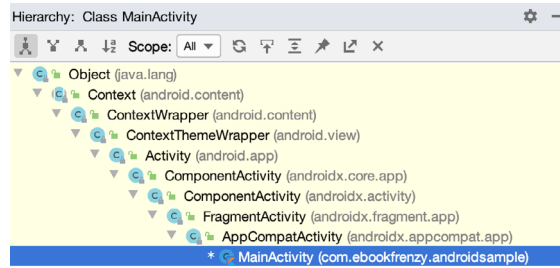


Figure 13-1

The Activity and Fragment classes contain a range of methods that are intended to be called by the Android runtime to notify the object when its state is changing. For the purposes of this chapter, we will refer to these as the *lifecycle methods*. An activity or fragment class simply needs to *override* these methods and implement the necessary functionality within them to react accordingly to state changes.

One such method is named *onCreate()* and, turning once again to the above code fragment, we can see that this method has already been overridden and implemented for us in the *MainActivity* class. In a later section we will explore in detail both *onCreate()* and the other relevant lifecycle methods of the Activity and Fragment classes.

13.3 Dynamic State vs. Persistent State

A key objective of lifecycle management is ensuring that the state of the activity is saved and restored at appropriate times. When talking about *state* in this context we mean the data that is currently being held within the activity and the appearance of the user interface. The activity might, for example, maintain a data model in memory that needs to be saved to a database, content provider or file. Such state information, because it persists from one invocation of the application to another, is referred to as the *persistent state*.

The appearance of the user interface (such as text entered into a text field but not yet committed to the application's internal data model) is referred to as the *dynamic state*, since it is typically only retained during a single invocation of the application (and also referred to as *user interface state* or *instance state*).

Understanding the differences between these two states is important because both the ways they are saved, and the reasons for doing so, differ.

The purpose of saving the persistent state is to avoid the loss of data that may result from an activity being killed by the runtime system while in the background. The dynamic state, on the other hand, is saved and restored for reasons that are slightly more complex.

Consider, for example, that an application contains an activity (which we will refer to as *Activity A*) containing a text field and some radio buttons. During the course of using the application, the user enters some text into the text field and makes a selection from the radio buttons. Before performing an action to save these changes, however, the user then switches to another activity causing *Activity A* to be pushed down the Activity Stack and placed into the background. After some time, the runtime system ascertains that memory is low and consequently kills *Activity A* to free up resources. As far as the user is concerned, however, *Activity A* was simply placed into the background and is ready to be moved to the foreground at any time. On returning *Activity A* to the foreground the user would, quite reasonably, expect the entered text and radio button selections to have been retained. In this scenario, however, a new instance of *Activity A* will have been created and, if the dynamic state was not saved and restored, the previous user input lost.

The main purpose of saving dynamic state, therefore, is to give the perception of seamless switching between foreground and background activities, regardless of the fact that activities may actually have been killed and restarted without the user's knowledge.

The mechanisms for saving persistent and dynamic state will become clearer in the following sections of this chapter.

13.4 The Android Lifecycle Methods

As previously explained, the Activity and Fragment classes contain a number of lifecycle methods which act as event handlers when the state of an instance changes. The primary methods supported by the Android Activity and Fragment class are as follows:

- **onCreate(Bundle savedInstanceState)** – The method that is called when the activity is first created and the ideal location for most initialization tasks to be performed. The method is passed an argument in the form of a *Bundle* object that may contain dynamic state information (typically relating to the state of the user interface) from a prior invocation of the activity.
- **onRestart()** – Called when the activity is about to restart after having previously been stopped by the runtime system.
- **onStart()** – Always called immediately after the call to the *onCreate()* or *onRestart()* methods, this method indicates to the activity that it is about to become visible to the user. This call will be followed by a call to *onResume()* if the activity moves to the top of the activity stack, or *onStop()* in the event that it is pushed down the stack by another activity.
- **onResume()** – Indicates that the activity is now at the top of the activity stack and is the activity with which the user is currently interacting.
- **onPause()** – Indicates that a previous activity is about to become the foreground activity. This call will be followed by a call to either the *onResume()* or *onStop()* method depending on whether the activity moves back to the foreground or becomes invisible to the user. Steps may be taken within this method to store *persistent state* information not yet saved by the app. To avoid delays in switching between activities, time consuming operations such as storing data to a database or performing network operations should be avoided within this method. This method should also ensure that any CPU intensive tasks such as animation are stopped.
- **onStop()** – The activity is now no longer visible to the user. The two possible scenarios that may follow this call are a call to *onRestart()* in the event that the activity moves to the foreground again, or *onDestroy()* if the activity is being terminated.
- **onDestroy()** – The activity is about to be destroyed, either voluntarily because the activity has completed its tasks and has called the *finish()* method or because the runtime is terminating it either to release memory or due to a configuration change (such as the orientation of the device changing). It is important to note that a call will not always be made to *onDestroy()* when an activity is terminated.
- **onConfigurationChanged()** – Called when a configuration change occurs for which the activity has indicated it is not to be restarted. The method is passed a Configuration object outlining the new device configuration and it is then the responsibility of the activity to react to the change.

The following lifecycle methods only apply to the Fragment class:

- **onAttach()** - Called when the fragment is assigned to an activity.
- **onCreateView()** - Called to create and return the fragment's user interface layout view hierarchy.
- **onActivityCreated()** - The *onCreate()* method of the activity with which the fragment is associated has completed execution.
- **onViewStateRestored()** - The fragment's saved view hierarchy has been restored.

In addition to the lifecycle methods outlined above, there are two methods intended specifically for saving and restoring the *dynamic state* of an activity:

- **onRestoreInstanceState(Bundle savedInstanceState)** – This method is called immediately after a call to the *onStart()* method in the event that the activity is restarting from a previous invocation in which state was saved. As with *onCreate()*, this method is passed a Bundle object containing the previous state data. This method is typically used in situations where it makes more sense to restore a previous state after the initialization of the activity has been performed in *onCreate()* and *onStart()*.
- **onSaveInstanceState(Bundle outState)** – Called before an activity is destroyed so that the current *dynamic state* (usually relating to the user interface) can be saved. The method is passed the Bundle object into which the state should be saved and which is subsequently passed through to the *onCreate()* and *onRestoreInstanceState()* methods when the activity is restarted. Note that this method is only called in situations where the runtime ascertains that dynamic state needs to be saved.

When overriding the above methods, it is important to remember that, with the exception of *onRestoreInstanceState()* and *onSaveInstanceState()*, the method implementation must include a call to the corresponding method in the super class. For example, the following method overrides the *onRestart()* method but also includes a call to the super class instance of the method:

```
protected void onRestart() {
    super.onRestart();
    Log.i(TAG, "onRestart");
}
```

Failure to make this super class call in method overrides will result in the runtime throwing an exception during execution. While calls to the super class in the *onRestoreInstanceState()* and *onSaveInstanceState()* methods are optional (they can, for example, be omitted when implementing custom save and restoration behavior) there are considerable benefits to using them, a subject that will be covered in the chapter entitled “*Saving and Restoring the State of an Android Activity*”.

13.5 Lifetimes

The final topic to be covered involves an outline of the *entire*, *visible* and *foreground* lifetimes through which an activity or fragment will transition during execution:

- **Entire Lifetime** – The term “entire lifetime” is used to describe everything that takes place between the initial call to the *onCreate()* method and the call to *onDestroy()* prior to the object terminating.
- **Visible Lifetime** – Covers the periods of execution between the call to *onStart()* and *onStop()*. During this period the activity or fragment is visible to the user though may not be the object with which the user is currently interacting.
- **Foreground Lifetime** – Refers to the periods of execution between calls to the *onResume()* and *onPause()* methods.

It is important to note that an activity or fragment may pass through the *foreground* and *visible* lifetimes multiple times during the course of the *entire* lifetime.

The concepts of lifetimes and lifecycle methods are illustrated in Figure 13-2:

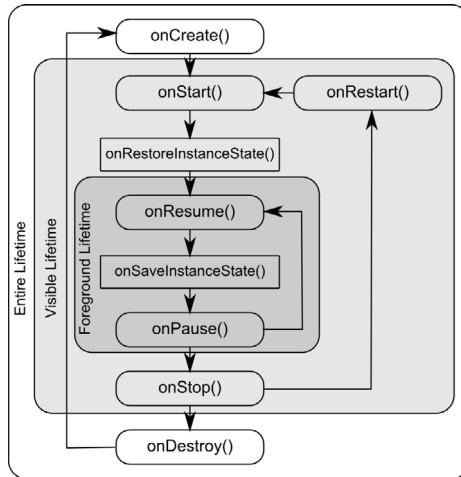


Figure 13-2

13.6 Foldable Devices and Multi-Resume

As discussed previously, an activity is considered to be in the resumed state when it has moved to the foreground and is the activity with which the user is currently interacting. On standard devices an app can have one activity in the resumed state at any one time and all other activities are likely to be in the paused or stopped state.

For some time now, Android has included multi-window support, allowing multiple activities to appear simultaneously in either split-screen or freeform configurations. Although originally used primarily on large screen tablet devices, this feature is likely to become more popular with the introduction of foldable devices.

On devices running Android 10 and on which multi-window support is enabled (as will be the case for most foldables), it will be possible for multiple app activities to be in the resumed state at the same time (a concept referred to as *multi-resume*) allowing those visible activities to continue functioning (for example streaming content or updating visual data) even when another activity currently has focus. Although multiple activities can be in the resumed state, only one of these activities will be considered to be the *topmost resumed activity* (in other words, the activity with which the user most recently interacted).

An activity can receive notification that it has gained or lost the topmost resumed status by implementing the `onTopResumedActivityChanged()` callback method.

13.7 Disabling Configuration Change Restarts

As previously outlined, an activity may indicate that it is not to be restarted in the event of certain configuration changes. This is achieved by adding an `android:configChanges` directive to the activity element within the project manifest file. The following manifest file excerpt, for example, indicates that the activity should not be restarted in the event of configuration changes relating to orientation or device-wide font size:

```
<activity android:name=".MainActivity"
    android:configChanges="orientation|fontScale"
    android:label="@string/app_name">
```

13.8 Lifecycle Method Limitations

As discussed at the start of this chapter, lifecycle methods have been in use for many years and, until recently, were the only mechanism available for handling lifecycle state changes for activities and fragments. There are, however, shortcomings to this approach.

One issue with the lifecycle methods is that they do not provide an easy way for an activity or fragment to find out its current lifecycle state at any given point during app execution. Instead the object would need to track the state internally, or wait for the next lifecycle method call.

Also, the methods do not provide a simple way for one object to observe the lifecycle state changes of other objects within an app. This is a serious consideration since many other objects within an app can potentially be impacted by a lifecycle state change in a given activity or fragment.

The lifecycle methods are also only available on subclasses of the `Fragment` and `Activity` classes. It is not possible, therefore, to build custom classes that are truly lifecycle aware.

Finally, the lifecycle methods result in most of the lifecycle handling code being written within the activity or fragment which can lead to complex and error prone code. Ideally, much of this code should reside in the other classes that are impacted by the state change. An app that streams video, for example, might include a class designed specifically to manage the incoming stream. If the app needs to pause the stream when the main activity is stopped, the code to do so should reside in the streaming class, not the main activity.

All of these problems and more are resolved by using *lifecycle-aware* components, a topic which will be covered starting with the chapter entitled “*Modern Android App Architecture with Jetpack*”.

13.9 Summary

All activities are derived from the Android *Activity* class which, in turn, contains a number of lifecycle methods that are designed to be called by the runtime system when the state of an activity changes. Similarly, the `Fragment` class contains a number of comparable methods. By overriding these methods, activities and fragments can respond to state changes and, where necessary, take steps to save and restore the current state of both the activity and the application. Lifecycle state can be thought of as taking two forms. The persistent state refers to data that needs to be stored between application invocations (for example to a file or database). Dynamic state, on the other hand, relates instead to the current appearance of the user interface.

Although lifecycle methods have a number of limitations that can be avoided by making use of lifecycle-aware components, an understanding of these methods is important to fully understand the new approaches to lifecycle management covered later in this book.

In this chapter, we have highlighted the lifecycle methods available to activities and covered the concept of activity lifetimes. In the next chapter, entitled “*Android Activity State Changes by Example*”, we will implement an example application that puts much of this theory into practice.

14. Android Activity State Changes by Example

The previous chapters have discussed in some detail the different states and lifecycles of the activities that comprise an Android application. In this chapter, we will put the theory of handling activity state changes into practice through the creation of an example application. The purpose of this example application is to provide a real world demonstration of an activity as it passes through a variety of different states within the Android runtime. In the next chapter, entitled “*Saving and Restoring the State of an Android Activity*”, the example project constructed in this chapter will be extended to demonstrate the saving and restoration of dynamic activity state.

14.1 Creating the State Change Example Project

The first step in this exercise is to create the new project. Begin by launching Android Studio and, if necessary, closing any currently open projects using the *File -> Close Project* menu option so that the Welcome screen appears.

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

Enter *StateChange* into the Name field and specify *com.ebookfrenzy.statechange* 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. Upon completion of the project creation process, the *StateChange* project should be listed in the Project tool window located along the left-hand edge of the Android Studio main window. Use the steps outlined in section 11.8 *Migrating a Project to View Binding* to convert the project to use view binding.

The next action to take involves the design of the user interface for the activity. This is stored in a file named *activity_main.xml* which should already be loaded into the Layout Editor tool. If it is not, navigate to it in the project tool window where it can be found in the *app -> res -> layout* folder. Once located, double-clicking on the file will load it into the Android Studio Layout Editor tool.

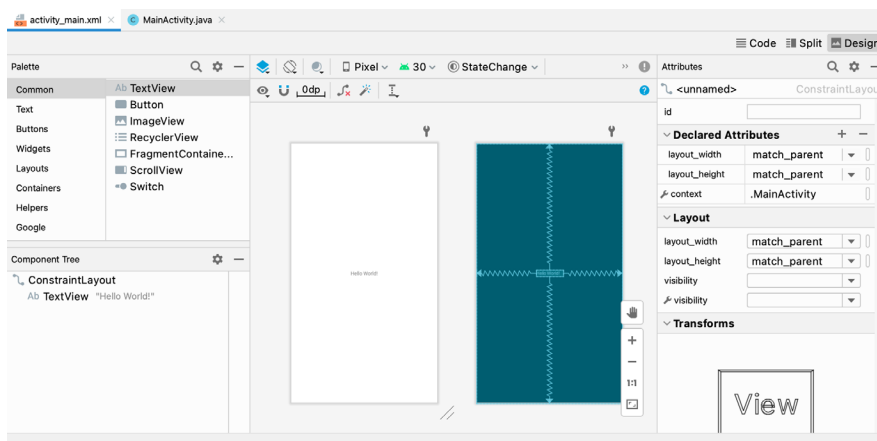


Figure 14-1

14.2 Designing the User Interface

With the user interface layout loaded into the Layout Editor tool, it is now time to design the user interface for the example application. Instead of the “Hello World!” TextView currently present in the user interface design, the activity actually requires an EditText view. Select the TextView object in the Layout Editor canvas and press the Delete key on the keyboard to remove it from the design.

From the Palette located on the left side of the Layout Editor, select the *Text* category and, from the list of text components, click and drag a *Plain Text* component over to the visual representation of the device screen. Move the component to the center of the display so that the center guidelines appear and drop it into place so that the layout resembles that of Figure 14-2.

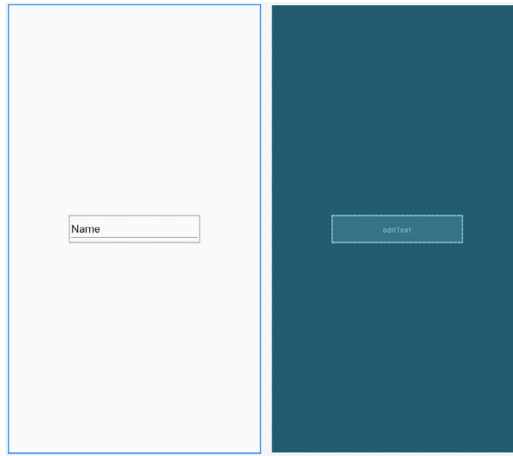


Figure 14-2

When using the EditText widget it is necessary to specify an *input type* for the view. This simply defines the type of text or data that will be entered by the user. For example, if the input type is set to *Phone*, the user will be restricted to entering numerical digits into the view. Alternatively, if the input type is set to *TextCapCharacters*, the input will default to upper case characters. Input type settings may also be combined.

For the purposes of this example, we will set the input type to support general text input. To do so, select the EditText widget in the layout and locate the *inputType* entry within the Attributes tool window. Click on the flag icon to the left of the current setting to open the list of options and, within the list, switch off *textPersonName* and enable *text* before clicking on the Apply button. Remaining in the Attributes tool window, change the id of the view to *editText* and click on the Refactor button in the resulting dialog.

By default the EditText is displaying text which reads “Name”. Remaining within the Attributes panel, delete this from the *text* property field so that the view is blank within the layout.

Before continuing, click on the *Infer Constraints* button in the layout editor toolbar to add any missing constraints.

14.3 Overriding the Activity Lifecycle Methods

At this point, the project contains a single activity named *MainActivity*, which is derived from the Android *AppCompatActivity* class. The source code for this activity is contained within the *MainActivity.java* file which should already be open in an editor session and represented by a tab in the editor tab bar. If the file is no longer open, navigate to it in the Project tool window panel (*app -> java -> com -> ebookfrenzy -> statechange -> MainActivity*) and double-click on it to load the file into the editor.

So far the only lifecycle method overridden by the activity is the *onCreate()* method which has been implemented

to call the super class instance of the method before setting up the user interface for the activity. We will now modify this method so that it outputs a diagnostic message in the Android Studio Logcat panel each time it executes. For this, we will use the *Log* class, which requires that we import *android.util.Log* and declare a tag that will enable us to filter these messages in the log output:

```
package com.ebookfrenzy.statechange;

.
.
import android.util.Log;
import androidx.annotation.NonNull;
import android.view.View;

public class MainActivity extends AppCompatActivity {

    private ActivityMainBinding binding;
    private static final String TAG = "StateChange";

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

        Log.i(TAG, "onCreate");
    }
}
```

The next task is to override some more methods, with each one containing a corresponding log call. These override methods may be added manually or generated using the *Alt-Insert* keyboard shortcut as outlined in the chapter entitled “*The Basics of the Android Studio Code Editor*”. Note that the Log calls will still need to be added manually if the methods are being auto-generated:

```
@Override
protected void onStart() {
    super.onStart();
    Log.i(TAG, "onStart");
}

@Override
protected void onResume() {
    super.onResume();
    Log.i(TAG, "onResume");
}

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

Android Activity State Changes by Example

```
        Log.i(TAG, "onPause");
    }

    @Override
    protected void onStop() {
        super.onStop();
        Log.i(TAG, "onStop");
    }

    @Override
    protected void onRestart() {
        super.onRestart();
        Log.i(TAG, "onRestart");
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        Log.i(TAG, "onDestroy");
    }

    @Override
    protected void onSaveInstanceState(@NonNull Bundle outState) {
        super.onSaveInstanceState(outState);
        Log.i(TAG, "onSaveInstanceState");
    }

    @Override
    protected void onRestoreInstanceState(@NonNull Bundle savedInstanceState) {
        super.onRestoreInstanceState(savedInstanceState);
        Log.i(TAG, "onRestoreInstanceState");
    }
}
```

14.4 Filtering the Logcat Panel

The purpose of the code added to the overridden methods in *MainActivity.java* is to output logging information to the *Logcat* tool window. This output can be configured to display all events relating to the device or emulator session, or restricted to those events that relate to the currently selected app. The output can also be further restricted to only those log events that match a specified filter.

Display the Logcat tool window and click on the filter menu (marked as B in Figure 14-3) to review the available options. When this menu is set to *Show only selected application*, only those messages relating to the app selected in the menu marked as A will be displayed in the Logcat panel. Choosing *No Filters*, on the other hand, will display all the messages generated by the device or emulator.

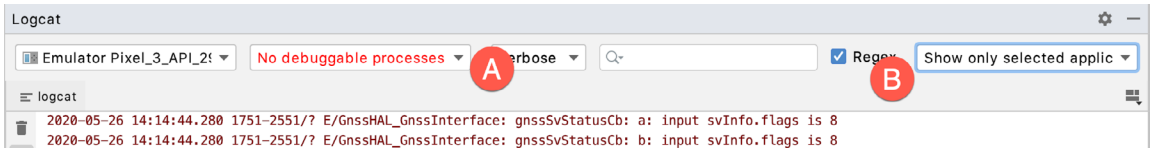


Figure 14-3

Before running the application, it is worth demonstrating the creation of a filter which, when selected, will further restrict the log output to ensure that only those log messages containing the tag declared in our activity are displayed.

From the filter menu (B), select the *Edit Filter Configuration* menu option. In the *Create New Logcat Filter* dialog (Figure 14-4), name the filter *Lifecycle* and, in the *Log Tag* field, enter the Tag value declared in *MainActivity.java* (in the above code example this was *StateChange*).

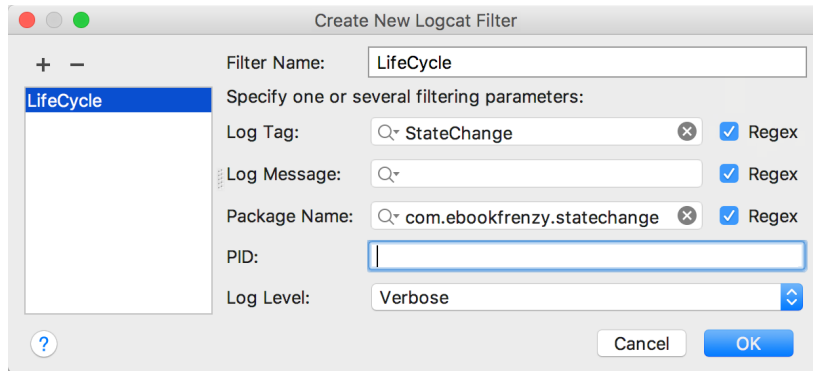


Figure 14-4

Enter the package identifier in the *Package Name* field and, when the changes are complete, click on the OK button to create the filter and dismiss the dialog. Instead of listing *No Filters*, the newly created filter should now be selected in the Logcat tool window.

14.5 Running the Application

For optimal results, the application should be run on a physical Android device or emulator. With the device configured and connected to the development computer, click on the run button represented by a green triangle located in the Android Studio toolbar as shown in Figure 14-5 below, select the *Run -> Run...* menu option or use the Shift+F10 keyboard shortcut:

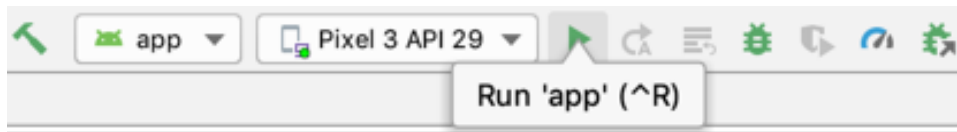


Figure 14-5

Select the physical Android device or emulator from the *Choose Device* dialog if it appears (assuming that you have not already configured it to be the default target). After Android Studio has built the application and installed it on the device it should start up and be running in the foreground.

A review of the Logcat panel should indicate which methods have so far been triggered (taking care to ensure that the *Lifecycle* filter created in the preceding section is selected to filter out log events that are not currently of interest to us):

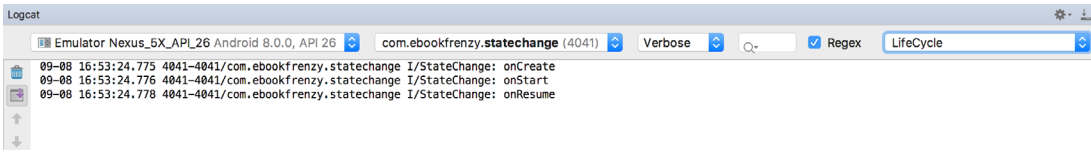


Figure 14-6

14.6 Experimenting with the Activity

With the diagnostics working, it is now time to exercise the application with a view to gaining an understanding of the activity lifecycle state changes. To begin with, consider the initial sequence of log events in the Logcat panel:

```
onCreate
onStart
onResume
```

Clearly, the initial state changes are exactly as outlined in “*Understanding Android Application and Activity Lifecycles*”. Note, however, that a call was not made to `onRestoreInstanceState()` since the Android runtime detected that there was no state to restore in this situation.

Tap on the Home icon in the bottom status bar on the device display and note the sequence of method calls reported in the log as follows:

```
onPause
onStop
onSaveInstanceState
```

In this case, the runtime has noticed that the activity is no longer in the foreground, is not visible to the user and has stopped the activity, but not without providing an opportunity for the activity to save the dynamic state. Depending on whether the runtime ultimately destroyed the activity or simply restarted it, the activity will either be notified it has been restarted via a call to `onRestart()` or will go through the creation sequence again when the user returns to the activity.

As outlined in “*Understanding Android Application and Activity Lifecycles*”, the destruction and recreation of an activity can be triggered by making a configuration change to the device, such as rotating from portrait to landscape. To see this in action, simply rotate the device while the *StateChange* application is in the foreground. When using the emulator, device rotation may be simulated using the rotation button located in the emulator toolbar. To complete the rotation, it may also be necessary to tap on the rotation button. This appears at the bottom of the device or emulator screen as shown in Figure 14-7:

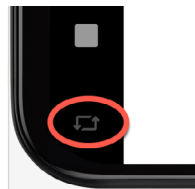


Figure 14-7

The resulting sequence of method calls in the log should read as follows:

```
onPause
onStop
onSaveInstanceState
```

```
onDestroy  
onCreate  
onStart  
onRestoreInstanceState  
onResume
```

Clearly, the runtime system has given the activity an opportunity to save state before being destroyed and restarted.

14.7 Summary

The old adage that a picture is worth a thousand words holds just as true for examples when learning a new programming paradigm. In this chapter, we have created an example Android application for the purpose of demonstrating the different lifecycle states through which an activity is likely to pass. In the course of developing the project in this chapter, we also looked at a mechanism for generating diagnostic logging information from within an activity.

In the next chapter, we will extend the *StateChange* example project to demonstrate how to save and restore an activity's dynamic state.

15. Saving and Restoring the State of an Android Activity

If the previous few chapters have achieved their objective, it should now be a little clearer as to the importance of saving and restoring the state of a user interface at particular points in the lifetime of an activity.

In this chapter, we will extend the example application created in “*Android Activity State Changes by Example*” to demonstrate the steps involved in saving and restoring state when an activity is destroyed and recreated by the runtime system.

A key component of saving and restoring dynamic state involves the use of the Android SDK *Bundle* class, a topic that will also be covered in this chapter.

15.1 Saving Dynamic State

An activity, as we have already learned, is given the opportunity to save dynamic state information via a call from the runtime system to the activity’s implementation of the *onSaveInstanceState()* method. Passed through as an argument to the method is a reference to a *Bundle* object into which the method will need to store any dynamic data that needs to be saved. The *Bundle* object is then stored by the runtime system on behalf of the activity and subsequently passed through as an argument to the activity’s *onCreate()* and *onRestoreInstanceState()* methods if and when they are called. The data can then be retrieved from the *Bundle* object within these methods and used to restore the state of the activity.

15.2 Default Saving of User Interface State

In the previous chapter, the diagnostic output from the *StateChange* example application showed that an activity goes through a number of state changes when the device on which it is running is rotated sufficiently to trigger an orientation change.

Launch the *StateChange* application once again, this time entering some text into the *EditText* field. Before performing the device rotation (on devices or emulators running Android 9 or later it may be necessary to tap the rotation button in the located in the status bar to complete the rotation). Having rotated the device, the following state change sequence should appear in the Logcat window:

```
onPause
onStop
onSaveInstanceState
onDestroy
onCreate
onStart
onRestoreInstanceState
onResume
```

Clearly this has resulted in the activity being destroyed and re-created. A review of the user interface of the running application, however, should show that the text entered into the *EditText* field has been preserved. Given that the activity was destroyed and recreated, and that we did not add any specific code to make sure the text was saved and restored, this behavior requires some explanation.

Saving and Restoring the State of an Android Activity

In fact, most of the view widgets included with the Android SDK already implement the behavior necessary to automatically save and restore state when an activity is restarted. The only requirement to enable this behavior is for the `onSaveInstanceState()` and `onRestoreInstanceState()` override methods in the activity to include calls to the equivalent methods of the super class:

```
@Override
protected void onSaveInstanceState(@NonNull Bundle outState) {
    super.onSaveInstanceState(outState);
}

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

The automatic saving of state for a user interface view can be disabled in the XML layout file by setting the `android:saveEnabled` property to `false`. For the purposes of an example, we will disable the automatic state saving mechanism for the `EditText` view in the user interface layout and then add code to the application to manually save and restore the state of the view.

To configure the `EditText` view such that state will not be saved and restored if the activity is restarted, edit the `activity_main.xml` file so that the entry for the view reads as follows (note that the XML can be edited directly by clicking on the *Text* tab on the bottom edge of the Layout Editor panel):

```
<EditText
    android:id="@+id/editText"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:ems="10"
    android:inputType="text"
    android:saveEnabled="false"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
```

After making the change, run the application, enter text and rotate the device to verify that the text is no longer saved and restored before proceeding.

15.3 The Bundle Class

For situations where state needs to be saved beyond the default functionality provided by the user interface view components, the `Bundle` class provides a container for storing data using a *key-value pair* mechanism. The *keys* take the form of string values, while the *values* associated with those *keys* can be in the form of a primitive value or any object that implements the Android *Parcelable* interface. A wide range of classes already implements the `Parcelable` interface. Custom classes may be made “parcelable” by implementing the set of methods defined in the `Parcelable` interface, details of which can be found in the Android documentation at:

<https://developer.android.com/reference/android/os/Parcelable.html>

The `Bundle` class also contains a set of methods that can be used to get and set key-value pairs for a variety of data types including both primitive types (including `Boolean`, `char`, `double` and `float` values) and objects (such

as Strings and CharSequences).

For the purposes of this example, and having disabled the automatic saving of text for the EditText view, we need to make sure that the text entered into the EditText field by the user is saved into the Bundle object and subsequently restored. This will serve as a demonstration of how to manually save and restore state within an Android application and will be achieved using the *putCharSequence()* and *getCharSequence()* methods of the Bundle class respectively.

15.4 Saving the State

The first step in extending the *StateChange* application is to make sure that the text entered by the user is extracted from the EditText component within the *onSaveInstanceState()* method of the *MainActivity* activity, and then saved as a key-value pair into the Bundle object.

To extract the text from the EditText object we first need to identify that object in the user interface. Clearly, this involves bridging the gap between the Java code for the activity (contained in the *MainActivity.java* source code file) and the XML representation of the user interface (contained within the *activity_main.xml* resource file). To extract the text entered into the EditText component we need to gain access to that user interface object.

Each component within a user interface has associated with it a unique identifier. By default, the Layout Editor tool constructs the id for a newly added component from the object type. If more than one view of the same type is contained in the layout the type name is followed by a sequential number (though this can, and should, be changed to something more meaningful by the developer). As can be seen by checking the *Component Tree* panel within the Android Studio main window when the *activity_main.xml* file is selected and the Layout Editor tool displayed, the EditText component has been assigned the id *editText*:

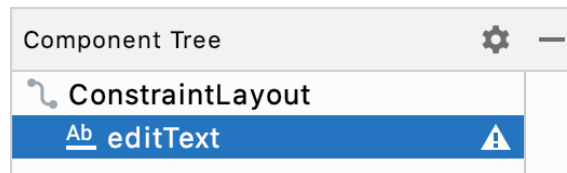


Figure 15-1

We can now obtain the text that the editText view contains via the object's *getText()* method, which, in turn, returns the current text:

```
CharSequence userText = binding.editText.getText();
```

Finally, we can save the text using the Bundle object's *putCharSequence()* method, passing through the key (this can be any string value but in this instance, we will declare it as "savedText") and the *userText* object as arguments:

```
outState.putCharSequence("savedText", userText);
```

Bringing this all together gives us a modified *onSaveInstanceState()* method in the *MainActivity.java* file that reads as follows:

```
package com.ebookfrenzy.statechange;
.
.
public class MainActivity extends AppCompatActivity {
.
.
.
}
```

Saving and Restoring the State of an Android Activity

```
protected void onSaveInstanceState(@NonNull Bundle outState) {
    super.onSaveInstanceState(outState);
    Log.i(TAG, "onSaveInstanceState");

    CharSequence userText = binding.editText.getText();
    outState.putCharSequence("savedText", userText);
}
.
```

Now that steps have been taken to save the state, the next phase is to ensure that it is restored when needed.

15.5 Restoring the State

The saved dynamic state can be restored in those lifecycle methods that are passed the Bundle object as an argument. This leaves the developer with the choice of using either *onCreate()* or *onRestoreInstanceState()*. The method to use will depend on the nature of the activity. In instances where state is best restored after the activity's initialization tasks have been performed, the *onRestoreInstanceState()* method is generally more suitable. For the purposes of this example we will add code to the *onRestoreInstanceState()* method to extract the saved state from the Bundle using the "savedText" key. We can then display the text on the editText component using the object's *setText()* method:

```
@Override
protected void onRestoreInstanceState(@NonNull Bundle savedInstanceState) {
    super.onRestoreInstanceState(savedInstanceState);
    Log.i(TAG, "onRestoreInstanceState");

    CharSequence userText =
        savedInstanceState.getCharSequence("savedText");

    binding.editText.setText(userText);
}
```

15.6 Testing the Application

All that remains is once again to build and run the *StateChange* application. Once running and in the foreground, touch the EditText component and enter some text before rotating the device to another orientation. Whereas the text changes were previously lost, the new text is retained within the editText component thanks to the code we have added to the activity in this chapter.

Having verified that the code performs as expected, comment out the *super.onSaveInstanceState()* and *super.onRestoreInstanceState()* calls from the two methods, re-launch the app and note that the text is still preserved after a device rotation. The default save and restoration system has essentially been replaced by a custom implementation, thereby providing a way to dynamically and selectively save and restore state within an activity.

15.7 Summary

The saving and restoration of dynamic state in an Android application is simply a matter of implementing the appropriate code in the appropriate lifecycle methods. For most user interface views, this is handled automatically by the Activity super class. In other instances, this typically consists of extracting values and settings within the *onSaveInstanceState()* method and saving the data as key-value pairs within the Bundle object passed through to the activity by the runtime system.

State can be restored in either the *onCreate()* or the *onRestoreInstanceState()* methods of the activity by extracting values from the Bundle object and updating the activity based on the stored values.

In this chapter, we have used these techniques to update the *StateChange* project so that the Activity retains changes through the destruction and subsequent recreation of an activity.

