

Android Studio 3.4

Development Essentials

Java Edition

Android Studio 3.4 Development Essentials – Java Edition

ISBN-13: 978-0-9600109-7-4

© 2019 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	2
1.3 Errata	2
1.4 Download the eBook	2
2. Setting up an Android Studio Development Environment	3
2.1 System Requirements	3
2.2 Downloading the Android Studio Package	3
2.3 Installing Android Studio	3
2.3.1 Installation on Windows	4
2.3.2 Installation on macOS	4
2.3.3 Installation on Linux	5
2.4 The Android Studio Setup Wizard	5
2.5 Installing Additional Android SDK Packages	6
2.6 Making the Android SDK Tools Command-line Accessible	8
2.6.1 Windows 7	8
2.6.2 Windows 8.1	9
2.6.3 Windows 10	10
2.6.4 Linux	10
2.6.5 macOS	10
2.7 Updating Android Studio and the SDK	10
2.8 Summary	10
3. Creating an Example Android App in Android Studio	11
3.1 About the Project	11
3.2 Creating a New Android Project	11
3.3 Creating an Activity	12
3.4 Defining the Project and SDK Settings	12
3.5 Modifying the Example Application	13
3.6 Reviewing the Layout and Resource Files	20
3.7 Adding Interaction	22
3.8 Summary	24
4. Creating an Android Virtual Device (AVD) in Android Studio	25
4.1 About Android Virtual Devices	25
4.2 Creating a New AVD	26
4.3 Starting the Emulator	27
4.4 Running the Application in the AVD	27
4.5 Run/Debug Configurations	29
4.6 Stopping a Running Application	30
4.7 AVD Command-line Creation	30
4.8 Android Virtual Device Configuration Files	32
4.9 Moving and Renaming an Android Virtual Device	32

4.10 Summary	32
5. Using and Configuring the Android Studio AVD Emulator	33
5.1 The Emulator Environment	33
5.2 The Emulator Toolbar Options.....	33
5.3 Working in Zoom Mode	35
5.4 Resizing the Emulator Window.....	35
5.5 Extended Control Options.....	35
5.5.1 Location.....	35
5.5.2 Cellular	36
5.5.3 Camera.....	36
5.5.4 Battery.....	36
5.5.5 Phone	36
5.5.6 Directional Pad.....	36
5.5.7 Microphone.....	36
5.5.8 Fingerprint	36
5.5.9 Virtual Sensors.....	36
5.5.10 Snapshots.....	36
5.5.11 Screen Record	36
5.5.12 Settings	37
5.5.13 Help.....	37
5.6 Working with Snapshots.....	37
5.7 Configuring Fingerprint Emulation	38
5.8 Summary	39
6. A Tour of the Android Studio User Interface	41
6.1 The Welcome Screen	41
6.2 The Main Window	42
6.3 The Tool Windows	43
6.4 Android Studio Keyboard Shortcuts	46
6.5 Switcher and Recent Files Navigation	46
6.6 Changing the Android Studio Theme	47
6.7 Summary	47
7. Testing Android Studio Apps on a Physical Android Device.....	49
7.1 An Overview of the Android Debug Bridge (ADB).....	49
7.2 Enabling ADB on Android based Devices.....	49
7.2.1 macOS ADB Configuration	50
7.2.2 Windows ADB Configuration.....	51
7.2.3 Linux adb Configuration.....	52
7.3 Testing the adb Connection.....	52
7.4 Summary	53
8. The Basics of the Android Studio Code Editor.....	55
8.1 The Android Studio Editor.....	55
8.2 Splitting the Editor Window	57
8.3 Code Completion	58
8.4 Statement Completion	59
8.5 Parameter Information	59
8.6 Parameter Name Hints	60
8.7 Code Generation	60

8.8 Code Folding.....	61
8.9 Quick Documentation Lookup	62
8.10 Code Reformatting.....	63
8.11 Finding Sample Code	63
8.12 Summary	64
9. An Overview of the Android Architecture	65
9.1 The Android Software Stack	65
9.2 The Linux Kernel.....	66
9.3 Android Runtime – ART.....	66
9.4 Android Libraries.....	66
9.4.1 C/C++ Libraries	67
9.5 Application Framework.....	67
9.6 Applications	68
9.7 Summary	68
10. The Anatomy of an Android Application	69
10.1 Android Activities.....	69
10.2 Android Fragments.....	69
10.3 Android Intents	70
10.4 Broadcast Intents.....	70
10.5 Broadcast Receivers	70
10.6 Android Services	70
10.7 Content Providers	71
10.8 The Application Manifest.....	71
10.9 Application Resources	71
10.10 Application Context.....	71
10.11 Summary	71
11. Understanding Android Application and Activity Lifecycles	73
11.1 Android Applications and Resource Management.....	73
11.2 Android Process States	73
11.2.1 Foreground Process	74
11.2.2 Visible Process	74
11.2.3 Service Process	74
11.2.4 Background Process.....	74
11.2.5 Empty Process	75
11.3 Inter-Process Dependencies	75
11.4 The Activity Lifecycle.....	75
11.5 The Activity Stack.....	75
11.6 Activity States	76
11.7 Configuration Changes	76
11.8 Handling State Change.....	77
11.9 Summary	77
12. Handling Android Activity State Changes.....	79
12.1 New vs. Old Lifecycle Techniques.....	79
12.2 The Activity and Fragment Classes.....	79
12.3 Dynamic State vs. Persistent State.....	82
12.4 The Android Lifecycle Methods.....	82
12.5 Lifetimes	84

Table of Contents

12.6 Disabling Configuration Change Restarts	85
12.7 Lifecycle Method Limitations.....	85
12.8 Summary	85
13. Android Activity State Changes by Example	87
13.1 Creating the State Change Example Project	87
13.2 Designing the User Interface	88
13.3 Overriding the Activity Lifecycle Methods	89
13.4 Filtering the Logcat Panel.....	92
13.5 Running the Application.....	93
13.6 Experimenting with the Activity.....	94
13.7 Summary	94
14. Saving and Restoring the State of an Android Activity	95
14.1 Saving Dynamic State	95
14.2 Default Saving of User Interface State	95
14.3 The Bundle Class	96
14.4 Saving the State.....	97
14.5 Restoring the State	98
14.6 Testing the Application.....	99
14.7 Summary	99
15. Understanding Android Views, View Groups and Layouts	101
15.1 Designing for Different Android Devices.....	101
15.2 Views and View Groups	101
15.3 Android Layout Managers	101
15.4 The View Hierarchy	103
15.5 Creating User Interfaces	104
15.6 Summary	104
16. A Guide to the Android Studio Layout Editor	105
16.1 Basic vs. Empty Activity Templates	105
16.2 The Android Studio Layout Editor	107
16.3 Design Mode.....	107
16.4 The Palette.....	108
16.5 Design and Layout Views.....	109
16.6 Text Mode.....	110
16.7 Setting Attributes.....	111
16.8 Converting Views.....	113
16.9 Displaying Sample Data	113
16.10 Creating a Custom Device Definition	114
16.11 Changing the Current Device.....	115
16.12 Summary.....	115
17. A Guide to the Android ConstraintLayout.....	117
17.1 How ConstraintLayout Works.....	117
17.1.1 Constraints.....	117
17.1.2 Margins.....	118
17.1.3 Opposing Constraints.....	118
17.1.4 Constraint Bias	119
17.1.5 Chains.....	120

17.1.6 Chain Styles.....	120
17.2 Baseline Alignment.....	121
17.3 Working with Guidelines.....	122
17.4 Configuring Widget Dimensions.....	122
17.5 Working with Barriers.....	122
17.6 Ratios.....	124
17.7 ConstraintLayout Advantages.....	124
17.8 ConstraintLayout Availability.....	124
17.9 Summary.....	125
18. A Guide to using ConstraintLayout in Android Studio.....	127
18.1 Design and Layout Views.....	127
18.2 Autoconnect Mode.....	128
18.3 Inference Mode.....	129
18.4 Manipulating Constraints Manually.....	129
18.5 Adding Constraints in the Inspector.....	131
18.6 Deleting Constraints.....	131
18.7 Adjusting Constraint Bias.....	132
18.8 Understanding ConstraintLayout Margins.....	132
18.9 The Importance of Opposing Constraints and Bias.....	134
18.10 Configuring Widget Dimensions.....	136
18.11 Adding Guidelines.....	137
18.12 Adding Barriers.....	138
18.13 Widget Group Alignment and Distribution.....	140
18.14 Converting other Layouts to ConstraintLayout.....	141
18.15 Summary.....	142
19. Working with ConstraintLayout Chains and Ratios in Android Studio.....	143
19.1 Creating a Chain.....	143
19.2 Changing the Chain Style.....	145
19.3 Spread Inside Chain Style.....	146
19.4 Packed Chain Style.....	146
19.5 Packed Chain Style with Bias.....	146
19.6 Weighted Chain.....	147
19.7 Working with Ratios.....	148
19.8 Summary.....	149
20. An Android Studio Layout Editor ConstraintLayout Tutorial.....	151
20.1 An Android Studio Layout Editor Tool Example.....	151
20.2 Creating a New Activity.....	151
20.3 Preparing the Layout Editor Environment.....	153
20.4 Adding the Widgets to the User Interface.....	154
20.5 Adding the Constraints.....	157
20.6 Testing the Layout.....	158
20.7 Using the Layout Inspector.....	159
20.8 Summary.....	160
21. Manual XML Layout Design in Android Studio.....	161
21.1 Manually Creating an XML Layout.....	161
21.2 Manual XML vs. Visual Layout Design.....	164
21.3 Summary.....	164

22. Managing Constraints using Constraint Sets	165
22.1 Java Code vs. XML Layout Files.....	165
22.2 Creating Views.....	165
22.3 View Attributes.....	166
22.4 Constraint Sets.....	166
22.4.1 Establishing Connections.....	166
22.4.2 Applying Constraints to a Layout.....	166
22.4.3 Parent Constraint Connections.....	166
22.4.4 Sizing Constraints.....	167
22.4.5 Constraint Bias.....	167
22.4.6 Alignment Constraints.....	167
22.4.7 Copying and Applying Constraint Sets.....	167
22.4.8 ConstraintLayout Chains.....	167
22.4.9 Guidelines.....	168
22.4.10 Removing Constraints.....	168
22.4.11 Scaling.....	168
22.4.12 Rotation.....	169
22.5 Summary.....	169
23. An Android ConstraintSet Tutorial	171
23.1 Creating the Example Project in Android Studio.....	171
23.2 Adding Views to an Activity.....	171
23.3 Setting View Attributes.....	172
23.4 Creating View IDs.....	173
23.5 Configuring the Constraint Set.....	174
23.6 Adding the EditText View.....	175
23.7 Converting Density Independent Pixels (dp) to Pixels (px).....	176
23.8 Summary.....	177
24. A Guide to using Instant Run in Android Studio	179
24.1 Introducing Instant Run.....	179
24.2 Understanding Instant Run Swapping Levels.....	179
24.3 Enabling and Disabling Instant Run.....	180
24.4 Using Instant Run.....	180
24.5 An Instant Run Tutorial.....	181
24.6 Triggering an Instant Run Hot Swap.....	181
24.7 Triggering an Instant Run Warm Swap.....	181
24.8 Triggering an Instant Run Cold Swap.....	182
24.9 The Run Button.....	182
24.10 Summary.....	182
25. An Overview and Example of Android Event Handling	183
25.1 Understanding Android Events.....	183
25.2 Using the android:onClick Resource.....	183
25.3 Event Listeners and Callback Methods.....	184
25.4 An Event Handling Example.....	184
25.5 Designing the User Interface.....	185
25.6 The Event Listener and Callback Method.....	186
25.7 Consuming Events.....	187
25.8 Summary.....	189

26. Android Touch and Multi-touch Event Handling	191
26.1 Intercepting Touch Events	191
26.2 The MotionEvent Object	191
26.3 Understanding Touch Actions.....	192
26.4 Handling Multiple Touches	192
26.5 An Example Multi-Touch Application	192
26.6 Designing the Activity User Interface	193
26.7 Implementing the Touch Event Listener.....	193
26.8 Running the Example Application.....	196
26.9 Summary	197
27. Detecting Common Gestures using the Android Gesture Detector Class.....	199
27.1 Implementing Common Gesture Detection.....	199
27.2 Creating an Example Gesture Detection Project	200
27.3 Implementing the Listener Class.....	200
27.4 Creating the GestureDetectorCompat Instance.....	203
27.5 Implementing the onTouchEvent() Method.....	203
27.6 Testing the Application.....	204
27.7 Summary	204
28. Implementing Custom Gesture and Pinch Recognition on Android	205
28.1 The Android Gesture Builder Application.....	205
28.2 The GestureOverlayView Class	205
28.3 Detecting Gestures.....	205
28.4 Identifying Specific Gestures	205
28.5 Building and Running the Gesture Builder Application.....	206
28.6 Creating a Gestures File	206
28.7 Creating the Example Project.....	206
28.8 Extracting the Gestures File from the SD Card	207
28.9 Adding the Gestures File to the Project	207
28.10 Designing the User Interface	207
28.11 Loading the Gestures File	208
28.12 Registering the Event Listener	209
28.13 Implementing the onGesturePerformed Method.....	209
28.14 Testing the Application.....	210
28.15 Configuring the GestureOverlayView.....	211
28.16 Intercepting Gestures.....	211
28.17 Detecting Pinch Gestures.....	211
28.18 A Pinch Gesture Example Project.....	212
28.19 Summary	214
29. An Introduction to Android Fragments.....	215
29.1 What is a Fragment?	215
29.2 Creating a Fragment	215
29.3 Adding a Fragment to an Activity using the Layout XML File.....	216
29.4 Adding and Managing Fragments in Code	218
29.5 Handling Fragment Events	219
29.6 Implementing Fragment Communication.....	219
29.7 Summary	221
30. Using Fragments in Android Studio - An Example.....	223

Table of Contents

30.1 About the Example Fragment Application	223
30.2 Creating the Example Project	223
30.3 Creating the First Fragment Layout.....	223
30.4 Creating the First Fragment Class	225
30.5 Creating the Second Fragment Layout.....	226
30.6 Adding the Fragments to the Activity	228
30.7 Making the Toolbar Fragment Talk to the Activity	229
30.8 Making the Activity Talk to the Text Fragment	233
30.9 Testing the Application.....	234
30.10 Summary	234
31. Modern Android App Architecture with Jetpack	235
31.1 What is Android Jetpack?	235
31.2 The “Old” Architecture.....	235
31.3 Modern Android Architecture	235
31.4 The ViewModel Component	236
31.5 The LiveData Component.....	236
31.6 LiveData and Data Binding.....	237
31.7 Android Lifecycles	238
31.8 Repository Modules.....	238
31.9 Summary	239
32. An Android Jetpack ViewModel Tutorial	241
32.1 About the Project	241
32.2 Creating the ViewModel Example Project.....	241
32.3 Reviewing the Project	242
32.3.1 The Main Activity.....	242
32.3.2 The Content Fragment	242
32.3.3 The ViewModel	244
32.4 Designing the Fragment Layout.....	244
32.5 Implementing the View Model.....	245
32.6 Associating the Fragment with the View Model.....	245
32.7 Modifying the Fragment	246
32.8 Accessing the ViewModel Data	247
32.9 Testing the Project.....	248
32.10 Summary	248
33. An Android Jetpack LiveData Tutorial	249
33.1 LiveData - A Recap	249
33.2 Adding LiveData to the ViewModel.....	249
33.3 Implementing the Observer.....	251
33.4 Summary	253
34. An Overview of Android Jetpack Data Binding	255
34.1 An Overview of Data Binding.....	255
34.2 The Key Components of Data Binding	255
34.2.1 The Project Build Configuration	255
34.2.2 The Data Binding Layout File.....	256
34.2.3 The Layout File Data Element	257
34.2.4 The Binding Classes.....	258
34.2.5 Data Binding Variable Configuration.....	258

34.2.6 Binding Expressions (One-Way).....	259
34.2.7 Binding Expressions (Two-Way).....	260
34.2.8 Event and Listener Bindings	260
34.3 Summary	261
35. An Android Jetpack Data Binding Tutorial.....	263
35.1 Removing the Redundant Code.....	263
35.2 Enabling Data Binding	265
35.3 Adding the Layout Element.....	266
35.4 Adding the Data Element to Layout File.....	267
35.5 Working with the Binding Class	267
35.6 Assigning the ViewModel Instance to the Data Binding Variable	269
35.7 Adding Binding Expressions	269
35.8 Adding the Conversion Method	270
35.9 Adding a Listener Binding.....	270
35.10 Testing the App.....	271
35.11 Summary	271
36. Working with Android Lifecycle-Aware Components	273
36.1 Lifecycle Awareness	273
36.2 Lifecycle Owners	273
36.3 Lifecycle Observers	274
36.4 Lifecycle States and Events.....	275
36.5 Summary	276
37. An Android Jetpack Lifecycle Awareness Tutorial	277
37.1 Creating the Example Lifecycle Project.....	277
37.2 Creating a Lifecycle Observer.....	277
37.3 Adding the Observer	279
37.4 Testing the Observer.....	279
37.5 Creating a Lifecycle Owner.....	280
37.6 Testing the Custom Lifecycle Owner.....	282
37.7 Summary	282
38. An Overview of the Navigation Architecture Component.....	283
38.1 Understanding Navigation.....	283
38.2 Declaring a Navigation Host.....	285
38.3 The Navigation Graph	286
38.4 Accessing the Navigation Controller.....	287
38.5 Triggering a Navigation Action	288
38.6 Passing Arguments.....	288
38.7 Summary	289
39. An Android Jetpack Navigation Component Tutorial	291
39.1 Creating the NavigationDemo Project.....	291
39.2 Adding Navigation to the Build Configuration.....	291
39.3 Creating the Navigation Graph Resource File.....	292
39.4 Declaring a Navigation Host.....	293
39.5 Adding Navigation Destinations.....	295
39.6 Designing the Destination Fragment Layouts.....	297
39.7 Adding an Action to the Navigation Graph.....	298

Table of Contents

39.8 Implement the OnFragmentInteractionListener	299
39.9 Triggering the Action	300
39.10 Passing Data Using Safeargs	301
39.11 Summary	304
40. Creating and Managing Overflow Menus on Android	305
40.1 The Overflow Menu	305
40.2 Creating an Overflow Menu	305
40.3 Displaying an Overflow Menu.....	306
40.4 Responding to Menu Item Selections.....	307
40.5 Creating Checkable Item Groups.....	307
40.6 Menus and the Android Studio Menu Editor.....	308
40.7 Creating the Example Project.....	309
40.8 Designing the Menu.....	309
40.9 Modifying the onOptionsItemSelected() Method.....	312
40.10 Testing the Application.....	313
40.11 Summary.....	313
41. Animating User Interfaces with the Android Transitions Framework.....	315
41.1 Introducing Android Transitions and Scenes	315
41.2 Using Interpolators with Transitions.....	316
41.3 Working with Scene Transitions	316
41.4 Custom Transitions and TransitionSets in Code	318
41.5 Custom Transitions and TransitionSets in XML.....	318
41.6 Working with Interpolators	320
41.7 Creating a Custom Interpolator	321
41.8 Using the beginDelayedTransition Method.....	322
41.9 Summary	322
42. An Android Transition Tutorial using beginDelayedTransition	323
42.1 Creating the Android Studio TransitionDemo Project.....	323
42.2 Preparing the Project Files.....	323
42.3 Implementing beginDelayedTransition Animation	323
42.4 Customizing the Transition	326
42.5 Summary	327
43. Implementing Android Scene Transitions – A Tutorial.....	329
43.1 An Overview of the Scene Transition Project	329
43.2 Creating the Android Studio SceneTransitions Project	329
43.3 Identifying and Preparing the Root Container	329
43.4 Designing the First Scene.....	329
43.5 Designing the Second Scene	330
43.6 Entering the First Scene	331
43.7 Loading Scene 2.....	332
43.8 Implementing the Transitions	333
43.9 Adding the Transition File	333
43.10 Loading and Using the Transition Set.....	334
43.11 Configuring Additional Transitions	335
43.12 Summary	336
44. Working with the Floating Action Button and Snackbar	337

44.1 The Material Design.....	337
44.2 The Design Library	337
44.3 The Floating Action Button (FAB)	337
44.4 The Snackbar.....	338
44.5 Creating the Example Project.....	339
44.6 Reviewing the Project.....	339
44.7 Changing the Floating Action Button.....	340
44.8 Adding the ListView to the Content Layout.....	341
44.9 Adding Items to the ListView.....	342
44.10 Adding an Action to the Snackbar.....	344
44.11 Summary.....	345
45. Creating a Tabbed Interface using the TabLayout Component	347
45.1 An Introduction to the ViewPager.....	347
45.2 An Overview of the TabLayout Component	347
45.3 Creating the TabLayoutDemo Project.....	348
45.4 Creating the First Fragment.....	348
45.5 Duplicating the Fragments.....	349
45.6 Adding the TabLayout and ViewPager.....	350
45.7 Creating the Pager Adapter.....	351
45.8 Performing the Initialization Tasks.....	352
45.9 Testing the Application.....	355
45.10 Customizing the TabLayout.....	355
45.11 Displaying Icon Tab Items.....	357
45.12 Summary.....	358
46. Working with the RecyclerView and CardView Widgets.....	359
46.1 An Overview of the RecyclerView.....	359
46.2 An Overview of the CardView	361
46.3 Adding the Libraries to the Project.....	362
46.4 Summary.....	362
47. An Android RecyclerView and CardView Tutorial.....	363
47.1 Creating the CardDemo Project.....	363
47.2 Removing the Floating Action Button	363
47.3 Adding the RecyclerView and CardView Libraries.....	364
47.4 Designing the CardView Layout.....	364
47.5 Adding the RecyclerView.....	365
47.6 Creating the RecyclerView Adapter.....	365
47.7 Adding the Image Files.....	368
47.8 Initializing the RecyclerView Component.....	368
47.9 Testing the Application.....	369
47.10 Responding to Card Selections	369
47.11 Summary.....	371
48. A Layout Editor Sample Data Tutorial	373
48.1 Adding Sample Data to a Project	373
48.2 Using Custom Sample Data	377
48.3 Summary.....	380
49. Working with the AppBar and Collapsing Toolbar Layouts	381

Table of Contents

49.1 The Anatomy of an AppBar	381
49.2 The Example Project	382
49.3 Coordinating the RecyclerView and Toolbar	382
49.4 Introducing the Collapsing Toolbar Layout	384
49.5 Changing the Title and Scrim Color	387
49.6 Summary	388
50. Implementing an Android Navigation Drawer	389
50.1 An Overview of the Navigation Drawer	389
50.2 Opening and Closing the Drawer	390
50.3 Responding to Drawer Item Selections	391
50.4 Using the Navigation Drawer Activity Template	391
50.5 Creating the Navigation Drawer Template Project	392
50.6 The Template Layout Resource Files	392
50.7 The Header Coloring Resource File	393
50.8 The Template Menu Resource File	393
50.9 The Template Code	393
50.10 Running the App	394
50.11 Summary	394
51. An Android Studio Master/Detail Flow Tutorial	395
51.1 The Master/Detail Flow	395
51.2 Creating a Master/Detail Flow Activity	396
51.3 The Anatomy of the Master/Detail Flow Template	397
51.4 Modifying the Master/Detail Flow Template	398
51.5 Changing the Content Model	398
51.6 Changing the Detail Pane	400
51.7 Modifying the WebsiteDetailFragment Class	401
51.8 Modifying the WebsiteListActivity Class	403
51.9 Adding Manifest Permissions	403
51.10 Running the Application	403
51.11 Summary	404
52. An Overview of Android Intents	405
52.1 An Overview of Intents	405
52.2 Explicit Intents	405
52.3 Returning Data from an Activity	406
52.4 Implicit Intents	407
52.5 Using Intent Filters	408
52.6 Checking Intent Availability	409
52.7 Summary	409
53. Android Explicit Intents – A Worked Example	411
53.1 Creating the Explicit Intent Example Application	411
53.2 Designing the User Interface Layout for MainActivity	411
53.3 Creating the Second Activity Class	412
53.4 Designing the User Interface Layout for ActivityB	413
53.5 Reviewing the Application Manifest File	414
53.6 Creating the Intent	415
53.7 Extracting Intent Data	416
53.8 Launching ActivityB as a Sub-Activity	416

53.9 Returning Data from a Sub-Activity.....	417
53.10 Testing the Application.....	418
53.11 Summary.....	418
54. Android Implicit Intents – A Worked Example	419
54.1 Creating the Android Studio Implicit Intent Example Project	419
54.2 Designing the User Interface	419
54.3 Creating the Implicit Intent	420
54.4 Adding a Second Matching Activity.....	420
54.5 Adding the Web View to the UI.....	421
54.6 Obtaining the Intent URL.....	421
54.7 Modifying the MyWebView Project Manifest File	422
54.8 Installing the MyWebView Package on a Device.....	424
54.9 Testing the Application.....	424
54.10 Summary.....	425
55. Android Broadcast Intents and Broadcast Receivers	427
55.1 An Overview of Broadcast Intents.....	427
55.2 An Overview of Broadcast Receivers	428
55.3 Obtaining Results from a Broadcast.....	429
55.4 Sticky Broadcast Intents	429
55.5 The Broadcast Intent Example.....	430
55.6 Creating the Example Application.....	430
55.7 Creating and Sending the Broadcast Intent.....	430
55.8 Creating the Broadcast Receiver	431
55.9 Registering the Broadcast Receiver.....	432
55.10 Testing the Broadcast Example	433
55.11 Listening for System Broadcasts.....	433
55.12 Summary.....	434
56. A Basic Overview of Threads and AsyncTask.....	435
56.1 An Overview of Threads	435
56.2 The Application Main Thread.....	435
56.3 Thread Handlers.....	435
56.4 A Basic AsyncTask Example	435
56.5 Subclassing AsyncTask	437
56.6 Testing the App.....	440
56.7 Canceling a Task.....	440
56.8 Summary	441
57. An Overview of Android Started and Bound Services.....	443
57.1 Started Services.....	443
57.2 Intent Service	443
57.3 Bound Service.....	444
57.4 The Anatomy of a Service	444
57.5 Controlling Destroyed Service Restart Options.....	445
57.6 Declaring a Service in the Manifest File.....	445
57.7 Starting a Service Running on System Startup.....	446
57.8 Summary	446
58. Implementing an Android Started Service – A Worked Example	447

Table of Contents

58.1 Creating the Example Project.....	447
58.2 Creating the Service Class.....	447
58.3 Adding the Service to the Manifest File.....	449
58.4 Starting the Service.....	449
58.5 Testing the IntentService Example.....	450
58.6 Using the Service Class.....	450
58.7 Creating the New Service.....	450
58.8 Modifying the User Interface.....	452
58.9 Running the Application.....	453
58.10 Creating an AsyncTask for Service Tasks.....	453
58.11 Summary.....	455
59. Android Local Bound Services – A Worked Example.....	457
59.1 Understanding Bound Services.....	457
59.2 Bound Service Interaction Options.....	457
59.3 An Android Studio Local Bound Service Example.....	457
59.4 Adding a Bound Service to the Project.....	458
59.5 Implementing the Binder.....	458
59.6 Binding the Client to the Service.....	461
59.7 Completing the Example.....	462
59.8 Testing the Application.....	463
59.9 Summary.....	463
60. Android Remote Bound Services – A Worked Example.....	465
60.1 Client to Remote Service Communication.....	465
60.2 Creating the Example Application.....	465
60.3 Designing the User Interface.....	465
60.4 Implementing the Remote Bound Service.....	466
60.5 Configuring a Remote Service in the Manifest File.....	467
60.6 Launching and Binding to the Remote Service.....	468
60.7 Sending a Message to the Remote Service.....	469
60.8 Summary.....	470
61. An Android Notifications Tutorial.....	471
61.1 An Overview of Notifications.....	471
61.2 Creating the NotifyDemo Project.....	473
61.3 Designing the User Interface.....	473
61.4 Creating the Second Activity.....	473
61.5 Creating a Notification Channel.....	474
61.6 Creating and Issuing a Basic Notification.....	476
61.7 Launching an Activity from a Notification.....	478
61.8 Adding Actions to a Notification.....	480
61.9 Bundled Notifications.....	480
61.10 Summary.....	483
62. An Android Direct Reply Notification Tutorial.....	485
62.1 Creating the DirectReply Project.....	485
62.2 Designing the User Interface.....	485
62.3 Creating the Notification Channel.....	486
62.4 Building the RemoteInput Object.....	487
62.5 Creating the PendingIntent.....	488

62.6 Creating the Reply Action.....	488
62.7 Receiving Direct Reply Input.....	491
62.8 Updating the Notification	492
62.9 Summary	493
63. An Introduction to Android Multi-Window Support.....	495
63.1 Split-Screen, Freeform and Picture-in-Picture Modes.....	495
63.2 Entering Multi-Window Mode	496
63.3 Enabling Freeform Support	497
63.4 Checking for Freeform Support	497
63.5 Enabling Multi-Window Support in an App.....	498
63.6 Specifying Multi-Window Attributes	498
63.7 Detecting Multi-Window Mode in an Activity.....	499
63.8 Receiving Multi-Window Notifications	499
63.9 Launching an Activity in Multi-Window Mode	500
63.10 Configuring Freeform Activity Size and Position.....	500
63.11 Summary	501
64. An Android Studio Multi-Window Split-Screen and Freeform Tutorial.....	503
64.1 Creating the Multi-Window Project.....	503
64.2 Designing the MainActivity User Interface	503
64.3 Adding the Second Activity	504
64.4 Launching the Second Activity	504
64.5 Enabling Multi-Window Mode	505
64.6 Testing Multi-Window Support	505
64.7 Launching the Second Activity in a Different Window	507
64.8 Summary	508
65. An Overview of Android SQLite Databases	509
65.1 Understanding Database Tables	509
65.2 Introducing Database Schema	509
65.3 Columns and Data Types	509
65.4 Database Rows	510
65.5 Introducing Primary Keys	510
65.6 What is SQLite?	510
65.7 Structured Query Language (SQL).....	510
65.8 Trying SQLite on an Android Virtual Device (AVD)	511
65.9 The Android Room Persistence Library.....	513
65.10 Summary	513
66. The Android Room Persistence Library	515
66.1 Revisiting Modern App Architecture	515
66.2 Key Elements of Room Database Persistence.....	515
66.2.1 Repository	516
66.2.2 Room Database	516
66.2.3 Data Access Object (DAO)	516
66.2.4 Entities.....	516
66.2.5 SQLite Database	516
66.3 Understanding Entities.....	517
66.4 Data Access Objects.....	520
66.5 The Room Database.....	521

Table of Contents

66.6 The Repository.....	522
66.7 In-Memory Databases	523
66.8 Summary	523
67. An Android TableLayout and TableRow Tutorial	525
67.1 The TableLayout and TableRow Layout Views.....	525
67.2 Creating the Room Database Project	526
67.3 Converting to a LinearLayout.....	526
67.4 Adding the TableLayout to the User Interface.....	527
67.5 Configuring the TableRows	528
67.6 Adding the Button Bar to the Layout	529
67.7 Adding the RecyclerView.....	530
67.8 Adjusting the Layout Margins	531
67.9 Summary	531
68. An Android Room Database and Repository Tutorial.....	533
68.1 About the RoomDemo Project.....	533
68.2 Modifying the Build Configuration.....	533
68.3 Building the Entity	533
68.4 Creating the Data Access Object.....	535
68.5 Adding the Room Database.....	537
68.6 Adding the Repository	538
68.7 Modifying the ViewModel.....	542
68.8 Creating the Product Item Layout	543
68.9 Adding the RecyclerView Adapter.....	543
68.10 Preparing the Main Fragment	545
68.11 Adding the Button Listeners.....	546
68.12 Adding LiveData Observers	547
68.13 Initializing the RecyclerView.....	548
68.14 Testing the RoomDemo App.....	548
68.15 Summary	549
69. Accessing Cloud Storage using the Android Storage Access Framework.....	551
69.1 The Storage Access Framework.....	551
69.2 Working with the Storage Access Framework.....	552
69.3 Filtering Picker File Listings	552
69.4 Handling Intent Results.....	553
69.5 Reading the Content of a File	554
69.6 Writing Content to a File	554
69.7 Deleting a File.....	555
69.8 Gaining Persistent Access to a File.....	555
69.9 Summary	556
70. An Android Storage Access Framework Example	557
70.1 About the Storage Access Framework Example.....	557
70.2 Creating the Storage Access Framework Example.....	557
70.3 Designing the User Interface	557
70.4 Declaring Request Codes	558
70.5 Creating a New Storage File.....	559
70.6 The onActivityResult() Method	560
70.7 Saving to a Storage File.....	562

70.8 Opening and Reading a Storage File	564
70.9 Testing the Storage Access Application	566
70.10 Summary	566
71. Implementing Video Playback on Android using the VideoView and MediaController Classes	567
71.1 Introducing the Android VideoView Class	567
71.2 Introducing the Android MediaController Class	568
71.3 Creating the Video Playback Example	568
71.4 Designing the VideoPlayer Layout	568
71.5 Configuring the VideoView	569
71.6 Adding Internet Permission	570
71.7 Adding the MediaController to the Video View	571
71.8 Setting up the onPreparedListener	572
71.9 Summary	573
72. Android Picture-in-Picture Mode.....	575
72.1 Picture-in-Picture Features	575
72.2 Enabling Picture-in-Picture Mode	576
72.3 Configuring Picture-in-Picture Parameters	576
72.4 Entering Picture-in-Picture Mode	577
72.5 Detecting Picture-in-Picture Mode Changes	577
72.6 Adding Picture-in-Picture Actions	578
72.7 Summary	578
73. An Android Picture-in-Picture Tutorial.....	581
73.1 Adding Picture-in-Picture Support to the Manifest	581
73.2 Adding a Picture-in-Picture Button	581
73.3 Entering Picture-in-Picture Mode	582
73.4 Detecting Picture-in-Picture Mode Changes	583
73.5 Adding a Broadcast Receiver	584
73.6 Adding the PiP Action	585
73.7 Testing the Picture-in-Picture Action	588
73.8 Summary	588
74. Video Recording and Image Capture on Android using Camera Intents	589
74.1 Checking for Camera Support	589
74.2 Calling the Video Capture Intent	589
74.3 Calling the Image Capture Intent	590
74.4 Creating an Android Studio Video Recording Project	591
74.5 Designing the User Interface Layout	591
74.6 Checking for the Camera	592
74.7 Launching the Video Capture Intent	592
74.8 Handling the Intent Return	593
74.9 Testing the Application	594
74.10 Summary	594
75. Making Runtime Permission Requests in Android.....	595
75.1 Understanding Normal and Dangerous Permissions	595
75.2 Creating the Permissions Example Project	597
75.3 Checking for a Permission	597
75.4 Requesting Permission at Runtime	599

Table of Contents

75.5 Providing a Rationale for the Permission Request	600
75.6 Testing the Permissions App.....	602
75.7 Summary	602
76. Android Audio Recording and Playback using MediaPlayer and MediaRecorder	603
76.1 Playing Audio	603
76.2 Recording Audio and Video using the MediaRecorder Class.....	604
76.3 About the Example Project	605
76.4 Creating the AudioApp Project.....	605
76.5 Designing the User Interface	605
76.6 Checking for Microphone Availability	606
76.7 Performing the Activity Initialization	607
76.8 Implementing the recordAudio() Method.....	608
76.9 Implementing the stopAudio() Method.....	609
76.10 Implementing the playAudio() method.....	609
76.11 Configuring and Requesting Permissions	610
76.12 Testing the Application.....	613
76.13 Summary	613
77. Working with the Google Maps Android API in Android Studio	615
77.1 The Elements of the Google Maps Android API	615
77.2 Creating the Google Maps Project.....	616
77.3 Obtaining Your Developer Signature	616
77.4 Adding the Apache HTTP Legacy Library Requirement	617
77.5 Testing the Application.....	617
77.6 Understanding Geocoding and Reverse Geocoding	618
77.7 Adding a Map to an Application	619
77.8 Requesting Current Location Permission	620
77.9 Displaying the User's Current Location	621
77.10 Changing the Map Type	622
77.11 Displaying Map Controls to the User.....	623
77.12 Handling Map Gesture Interaction.....	624
77.12.1 Map Zooming Gestures.....	624
77.12.2 Map Scrolling/Panning Gestures	624
77.12.3 Map Tilt Gestures.....	624
77.12.4 Map Rotation Gestures.....	625
77.13 Creating Map Markers.....	625
77.14 Controlling the Map Camera	626
77.15 Summary	627
78. Printing with the Android Printing Framework	629
78.1 The Android Printing Architecture	629
78.2 The Print Service Plugins	629
78.3 Google Cloud Print.....	630
78.4 Printing to Google Drive.....	630
78.5 Save as PDF	631
78.6 Printing from Android Devices	631
78.7 Options for Building Print Support into Android Apps.....	632
78.7.1 Image Printing	632
78.7.2 Creating and Printing HTML Content	633
78.7.3 Printing a Web Page.....	635

78.7.4 Printing a Custom Document	635
78.8 Summary	635
79. An Android HTML and Web Content Printing Example	637
79.1 Creating the HTML Printing Example Application	637
79.2 Printing Dynamic HTML Content	637
79.3 Creating the Web Page Printing Example	640
79.4 Removing the Floating Action Button	640
79.5 Designing the User Interface Layout	640
79.6 Loading the Web Page into the WebView	642
79.7 Adding the Print Menu Option	643
79.8 Summary	645
80. A Guide to Android Custom Document Printing	647
80.1 An Overview of Android Custom Document Printing	647
80.1.1 Custom Print Adapters	647
80.2 Preparing the Custom Document Printing Project	648
80.3 Creating the Custom Print Adapter	649
80.4 Implementing the onLayout() Callback Method	650
80.5 Implementing the onWrite() Callback Method	653
80.6 Checking a Page is in Range	655
80.7 Drawing the Content on the Page Canvas	656
80.8 Starting the Print Job	658
80.9 Testing the Application	659
80.10 Summary	659
81. An Introduction to Android App Links	661
81.1 An Overview of Android App Links	661
81.2 App Link Intent Filters	661
81.3 Handling App Link Intents	662
81.4 Associating the App with a Website	662
81.5 Summary	663
82. An Android Studio App Links Tutorial	665
82.1 About the Example App	665
82.2 The Database Schema	665
82.3 Loading and Running the Project	666
82.4 Adding the URL Mapping	667
82.5 Adding the Intent Filter	670
82.6 Adding Intent Handling Code	670
82.7 Testing the App Link	674
82.8 Associating an App Link with a Web Site	674
82.9 Summary	676
83. A Guide to the Android Studio Profiler	677
83.1 Accessing the Android Profiler	677
83.2 Enabling Advanced Profiling	677
83.3 The Android Profiler Tool Window	678
83.4 The Sessions Panel	679
83.5 The CPU Profiler	680
83.6 Memory Profiler	683

Table of Contents

83.7 Network Profiler.....	684
83.8 Energy Profiler.....	686
83.9 Summary.....	686
84. An Android Biometric Authentication Tutorial.....	687
84.1 An Overview of Biometric Authentication.....	687
84.2 Creating the Biometric Authentication Project.....	687
84.3 Configuring Device Fingerprint Authentication.....	688
84.4 Adding the Biometric Permission to the Manifest File.....	688
84.5 Designing the User Interface.....	689
84.6 Adding a Toast Convenience Method.....	689
84.7 Checking the Security Settings.....	690
84.8 Configuring the Authentication Callbacks.....	691
84.9 Adding the CancellationSignal.....	692
84.10 Starting the Biometric Prompt.....	693
84.11 Testing the Project.....	693
84.12 Summary.....	695
85. Creating, Testing and Uploading an Android App Bundle.....	697
85.1 The Release Preparation Process.....	697
85.2 Android App Bundles.....	697
85.3 Register for a Google Play Developer Console Account.....	698
85.4 Configuring the App in the Console.....	699
85.5 Enabling Google Play App Signing.....	699
85.6 Creating a Keystore File.....	700
85.7 Creating the Android App Bundle.....	702
85.8 Generating Test APK Files.....	703
85.9 Uploading the App Bundle to the Google Play Developer Console.....	704
85.10 Exploring the App Bundle.....	704
85.11 Managing Testers.....	706
85.12 Uploading New App Bundle Revisions.....	707
85.13 Analyzing the App Bundle File.....	708
85.14 Enabling Google Play Signing for an Existing App.....	709
85.15 Summary.....	710
86. An Overview of Android Dynamic Feature Modules.....	711
86.1 An Overview of Dynamic Feature Modules.....	711
86.2 Dynamic Feature Module Architecture.....	711
86.3 Creating a Dynamic Feature Module.....	712
86.4 Converting an Existing Module for Dynamic Delivery.....	714
86.5 Working with Dynamic Feature Modules.....	716
86.6 Handling Large Dynamic Feature Modules.....	718
86.7 Summary.....	719
87. An Android Studio Dynamic Feature Tutorial.....	721
87.1 Creating the DynamicFeature Project.....	721
87.2 Adding Dynamic Feature Support to the Project.....	721
87.3 Designing the Base Activity User Interface.....	722
87.4 Adding the Dynamic Feature Module.....	723
87.5 Reviewing the Dynamic Feature Module.....	724
87.6 Adding the Dynamic Feature Activity.....	726

87.7 Implementing the <code>launchIntent()</code> Method.....	727
87.8 Uploading the App Bundle for Testing.....	728
87.9 Implementing the <code>installFeature()</code> Method	729
87.10 Adding the Update Listener.....	731
87.11 Handling Large Downloads.....	734
87.12 Using Deferred Installation	736
87.13 Removing a Dynamic Module	736
87.14 Summary.....	737
88. An Overview of Gradle in Android Studio.....	739
88.1 An Overview of Gradle	739
88.2 Gradle and Android Studio	739
88.2.1 Sensible Defaults	739
88.2.2 Dependencies.....	739
88.2.3 Build Variants	740
88.2.4 Manifest Entries	740
88.2.5 APK Signing.....	740
88.2.6 ProGuard Support.....	740
88.3 The Top-level Gradle Build File.....	740
88.4 Module Level Gradle Build Files.....	742
88.5 Configuring Signing Settings in the Build File.....	744
88.6 Running Gradle Tasks from the Command-line	745
88.7 Summary	745
Index.....	747

1. Introduction

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

Fully updated for Android Studio 3.4 and Android 9, the goal of this book is to teach the skills necessary to develop Android based applications using the Java programming language.

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

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

More advanced topics such as intents are also covered, as are touch screen handling, gesture recognition, camera access and the playback and recording of both video and audio. This edition of the book also covers printing, transitions and cloud-based file storage.

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

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

Other key features of Android Studio 3.4 and Android 9 are also covered in detail including the Layout Editor, the ConstraintLayout and ConstraintSet classes, constraint chains and barriers, direct reply notifications and multi-window support.

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

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

1.1 Downloading the Code Samples

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

<https://www.ebookfrenzy.com/retail/androidstudio34/index.php>

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

Introduction

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

1.2 Feedback

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

1.3 Errata

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

<https://www.ebookfrenzy.com/errata/androidstudio34.html>

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

1.4 Download the eBook

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

2. Setting up an Android Studio Development Environment

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

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

2.1 System Requirements

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

- Windows 7/8/10 (32-bit or 64-bit)
- macOS 10.10 or later (Intel based systems only)
- Linux systems with version 2.19 or later of GNU C Library (glibc)
- Minimum of 3GB of RAM (8GB is preferred)
- Approximately 4GB of available disk space
- 1280 x 800 minimum screen resolution

2.2 Downloading the Android Studio Package

Most of the work involved in developing applications for Android will be performed using the Android Studio environment. The content and examples in this book were created based on Android Studio version 3.4 which, at the time writing is the current version.

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

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

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

If this page provides instructions for downloading a newer version of Android Studio it is important to note that there may be some minor differences between this book and the software. A web search for Android Studio 3.4 should provide the option to download the older version in the event that these differences become a problem.

2.3 Installing Android Studio

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

2.3.1 Installation on Windows

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

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

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

On versions of Windows with a Start menu, the newly installed Android Studio can be launched from the entry added to that menu during the installation. The executable may be pinned to the task bar for easy access by navigating to the *Android Studio\bin* directory, right-clicking on the executable and selecting the *Pin to Taskbar* menu option. Note that the executable is provided in 32-bit (*studio*) and 64-bit (*studio64*) executable versions. If you are running a 32-bit system be sure to use the *studio* executable.

2.3.2 Installation on macOS

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



Figure 2-1

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

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

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

2.3.3 Installation on Linux

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

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

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

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

```
./studio.sh
```

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

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

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

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

2.4 The Android Studio Setup Wizard

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

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

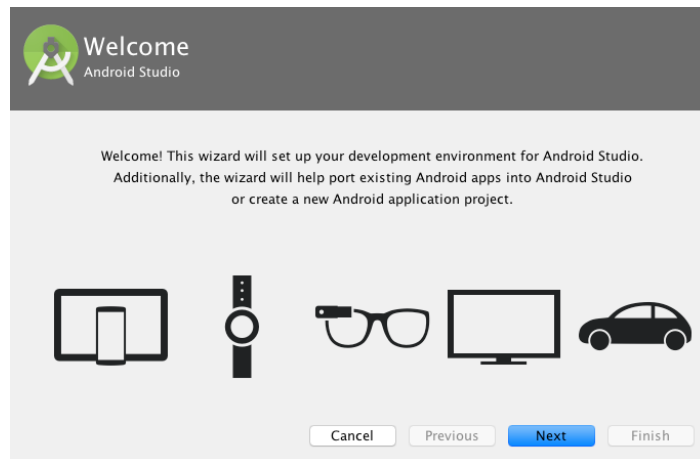


Figure 2-2

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

Android Studio will proceed to download and configure the latest Android SDK and some additional components and packages. Once this process has completed, click on the *Finish* button in the *Downloading Components* dialog at which point the Welcome to Android Studio screen should then appear:

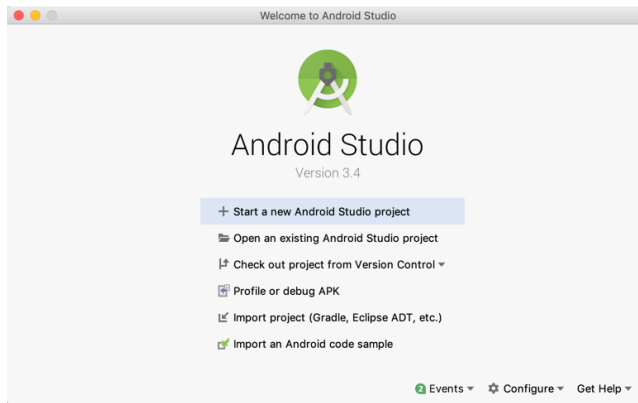


Figure 2-3

2.5 Installing Additional Android SDK Packages

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

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

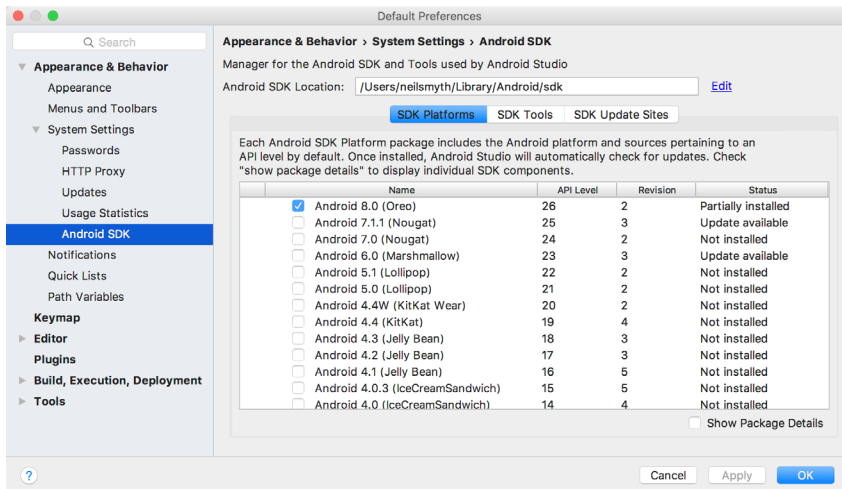


Figure 2-4

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

It is also possible that updates will be listed as being available for the latest SDK. To access detailed information about the packages that are available for update, enable the *Show Package Details* option located in the lower right-hand corner of the screen. This will display information similar to that shown in Figure 2-5:

	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 2-5

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

In addition to the Android SDK packages, a number of tools are also installed for building Android applications. To view the currently installed packages and check for updates, remain within the SDK settings screen and select the SDK Tools tab as shown in Figure 2-6:

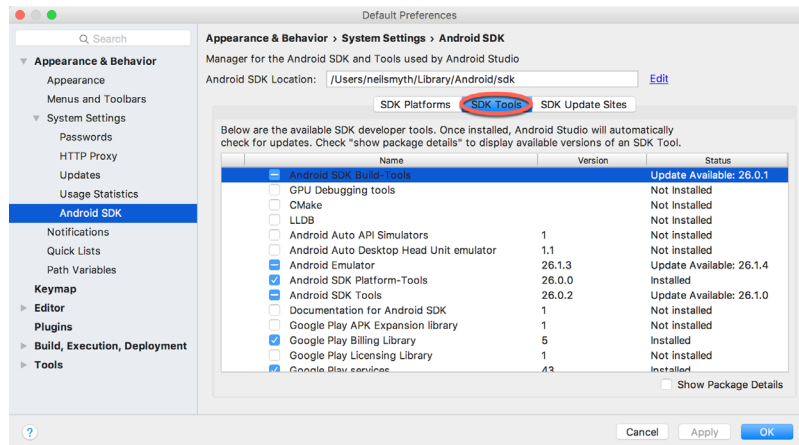


Figure 2-6

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

- Android SDK Build-tools
- Android Emulator
- Android SDK Platform-tools
- Android SDK Tools
- Google Play Services
- Intel x86 Emulator Accelerator (HAXM installer)
- ConstraintLayout for Android
- Solver for ConstraintLayout

Setting up an Android Studio Development Environment

- Android Support Repository
- Google Repository
- Google USB Driver (Windows only)

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

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

2.6 Making the Android SDK Tools Command-line Accessible

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

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

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

The location of the SDK on your system can be identified by launching the SDK Manager and referring to the *Android SDK Location*: field located at the top of the settings panel as highlighted in Figure 2-7:

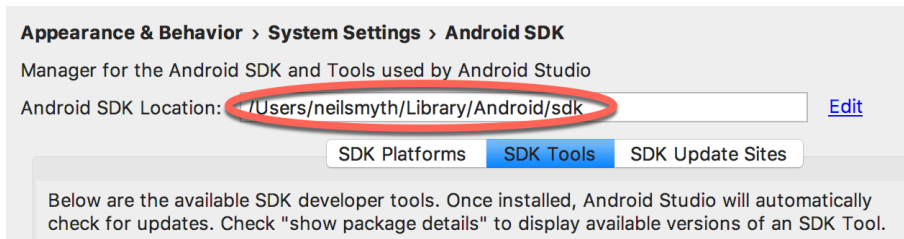


Figure 2-7

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

2.6.1 Windows 7

1. Right-click on Computer in the desktop start menu and select Properties from the resulting menu.
2. In the properties panel, select the Advanced System Settings link and, in the resulting dialog, click on the Environment Variables... button.
3. In the Environment Variables dialog, locate the Path variable in the System variables list, select it and click on *Edit...* Locate the end of the current variable value string and append the path to the Android platform tools to the end, using a semicolon to separate the path from the preceding values. For example, assuming the Android SDK was installed into C:\Users\demo\AppData\Local\Android\sdk, the following would be

appended to the end of the current Path value:

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

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

Once the above steps are complete, verify that the path is correctly set by opening a *Command Prompt* window (*Start -> All Programs -> Accessories -> Command Prompt*) and at the prompt enter:

```
echo %Path%
```

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

```
adb
```

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

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

```
avdmanager
```

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

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

2.6.2 Windows 8.1

1. On the start screen, move the mouse to the bottom right-hand corner of the screen and select Search from the resulting menu. In the search box, enter Control Panel. When the Control Panel icon appears in the results area, click on it to launch the tool on the desktop.
2. Within the Control Panel, use the Category menu to change the display to Large Icons. From the list of icons select the one labeled System.
3. Follow the steps outlined for Windows 7 starting from step 2 through to step 4.

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

Within the Command Prompt window, enter:

```
echo %Path%
```

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

```
adb
```

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

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

```
avdmanager
```

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

```
'adb' is not recognized as an internal or external command,
```

Setting up an Android Studio Development Environment

operable program or batch file.

2.6.3 Windows 10

Right-click on the Start menu, select Settings from the resulting menu and enter “Edit the system environment variables” into the Find a setting text field. In the System Properties dialog, click the *Environment Variables...* button. Follow the steps outlined for Windows 7 starting from step 3.

2.6.4 Linux

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

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

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

2.6.5 macOS

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

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

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

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

2.7 Updating Android Studio and the SDK

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

To manually check for Android Studio updates, click on the *Configure -> Check for Update* menu option within the Android Studio welcome screen, or use the *Help -> Check for Update* menu option accessible from within the Android Studio main window.

2.8 Summary

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

3. Creating an Example Android App in Android Studio

The preceding chapters of this book have covered the steps necessary to configure an environment suitable for the development of Android applications using the Android Studio IDE. Before moving on to slightly more advanced topics, now is a good time to validate that all of the required development packages are installed and functioning correctly. The best way to achieve this goal is to create an Android application and compile and run it. This chapter will cover the creation of a simple Android application project using Android Studio. Once the project has been created, a later chapter will explore the use of the Android emulator environment to perform a test run of the application.

3.1 About the Project

The project created in this chapter takes the form of a very simple currency conversion calculator (so simple, in fact, that it only converts from dollars to euros and does so using an estimated conversion rate). The project will also make use of 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:

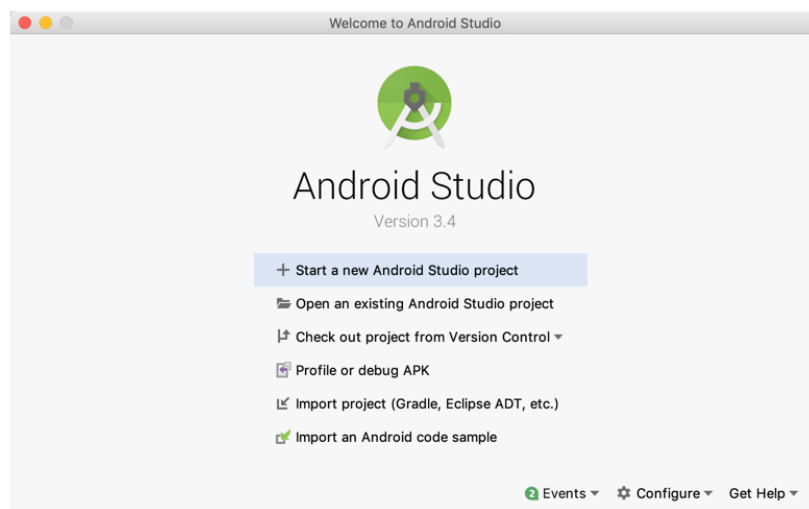


Figure 3-1

Creating an Example Android App in Android Studio

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

3.3 Creating an Activity

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

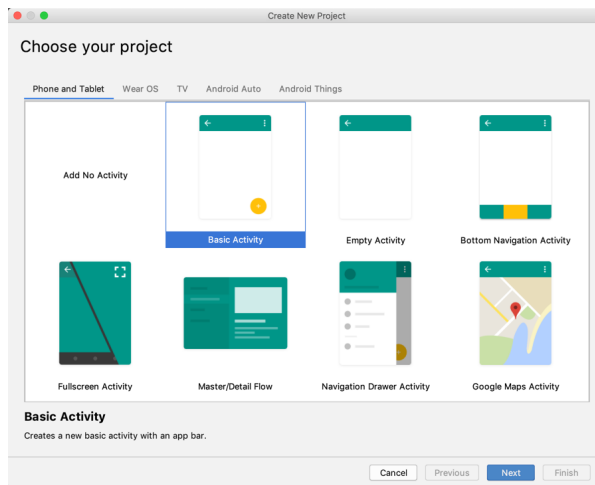


Figure 3-2

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

3.4 Defining the Project and SDK Settings

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

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

```
com.mycompany.androidsample
```

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

```
com.example.androidsample
```

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

Set the minimum SDK setting to API 26: Android 8.0 (Oreo). This is the SDK that will be used in most of the projects created in this book unless a necessary feature is only available in a more recent version. While Android Studio allows older SDK versions to be selected, many of the security and privacy features built into Android were only introduced after the API 25 SDK was released. To improve app security, Google announced that starting in August 2018 the Google Play store will only accept new apps built using API 26 or newer. This same restriction was also applied to updates of existing apps after October 2018.

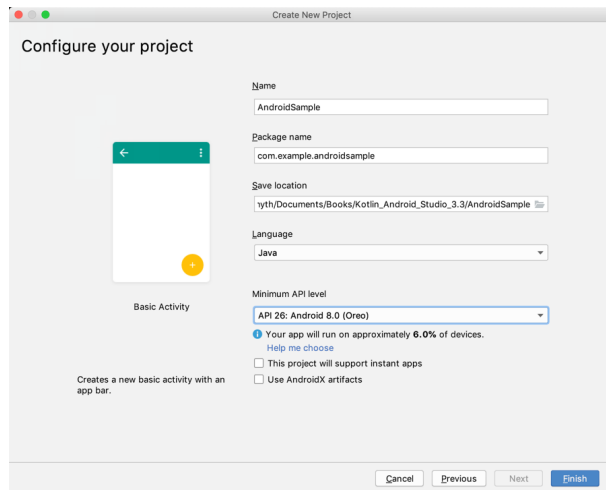


Figure 3-3

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.

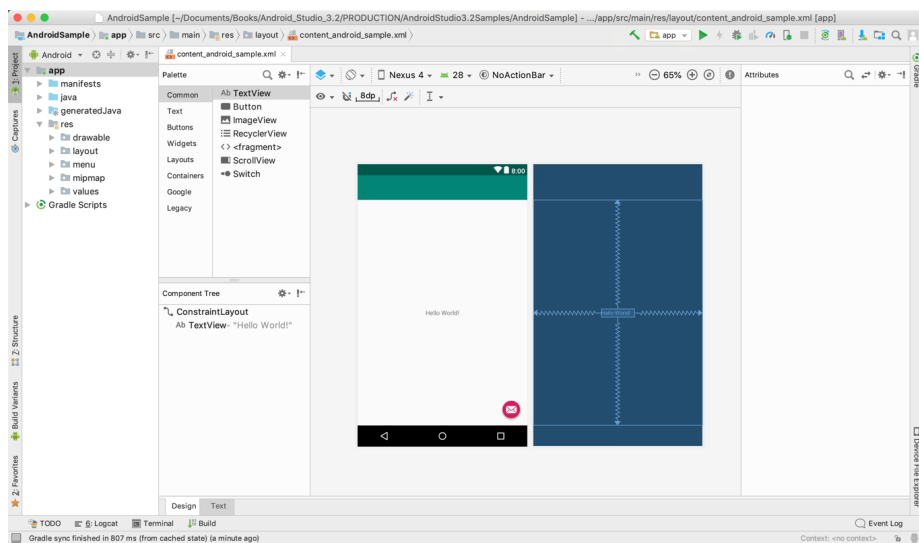


Figure 3-4

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

Creating an Example Android App in Android Studio

information can be displayed. By default, this panel will 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:

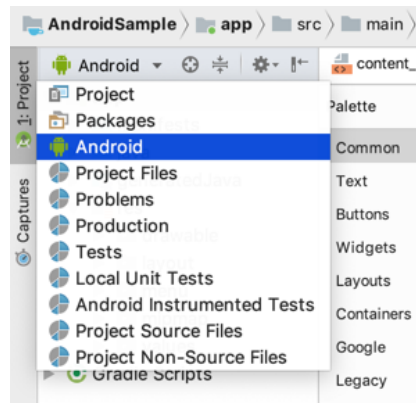


Figure 3-5

The example project created for us when we selected the option to create an activity consists of a user interface containing a label that will read “Hello World!” when the application is executed.

The next step in this tutorial is to modify the user interface of our application so that it displays a larger text view object with a different message to the one provided for us by Android Studio.

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. This layout file includes the app bar (also known as an action bar) that appears across the top of the device screen (marked A in Figure 3-6) and the floating action button (the email button marked B). In addition to these items, the *activity_main.xml* layout file contains a reference to a second file containing the content layout (marked C):

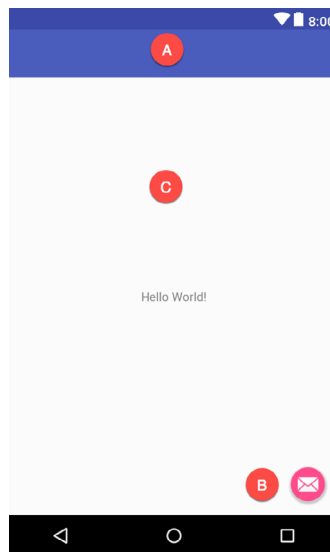


Figure 3-6

By default, the content layout is contained within a file named *content_main.xml* and it is within this file that

changes to the layout of the activity are made. Using the Project tool window, locate this file as illustrated in Figure 3-7:

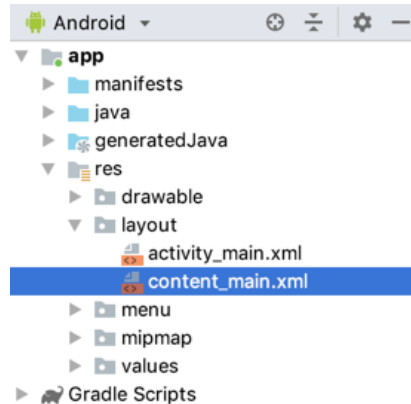


Figure 3-7

Once located, double-click on the file to load it into the user interface Layout Editor tool which will appear in the center panel of the Android Studio main window:

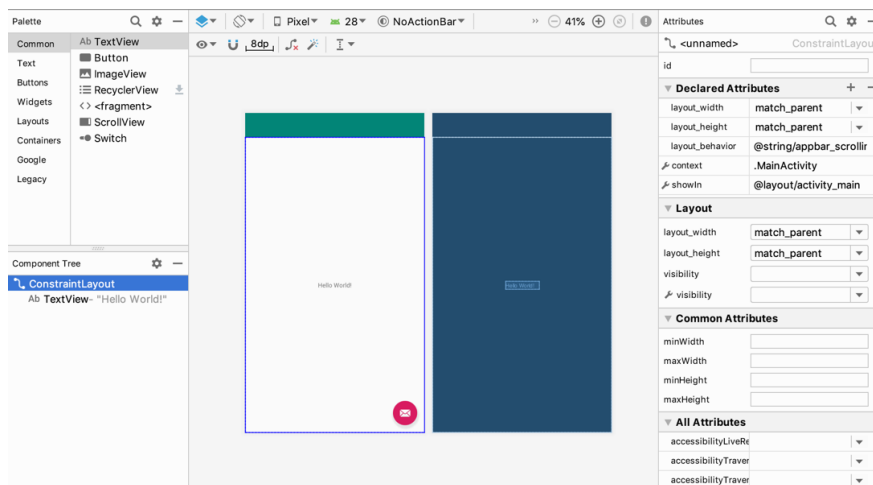



Figure 3-8

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

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

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

Creating an Example Android App in Android Studio

user interface components are positioned and managed on the screen. Though it is difficult to tell from looking at the visual representation of the user interface, the current design has been created using a `ConstraintLayout`. This can be confirmed by reviewing the information in the *Component Tree* panel which, by default, is located in the lower left-hand corner of the Layout Editor panel and is shown in Figure 3-9:

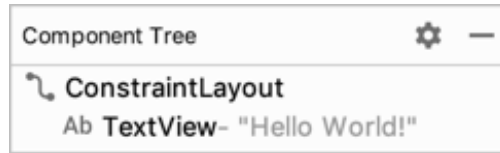


Figure 3-9

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

Before proceeding, check that the Layout Editor's Autoconnect mode is enabled. This means that as components are added to the layout, the Layout Editor will automatically add constraints to 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 3-10). If necessary, re-enable Autoconnect mode by clicking on this button.

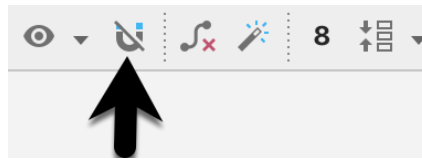


Figure 3-10

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

The Palette panel consists of two columns with the left-hand column containing a list of view component categories. The right-hand column lists the components contained within the currently selected category. In Figure 3-11, for example, the `Button` view is currently selected within the `Buttons` category:

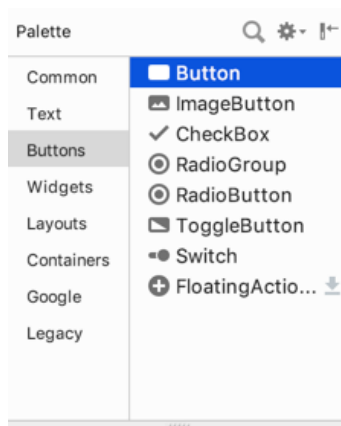


Figure 3-11

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

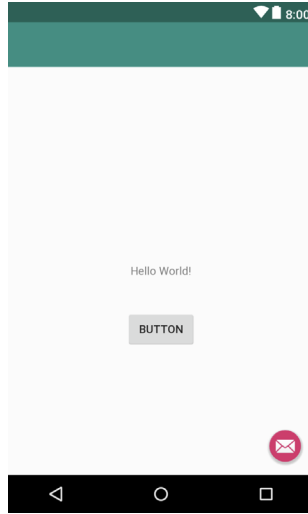


Figure 3-12

The next step is to change the text 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 3-13:

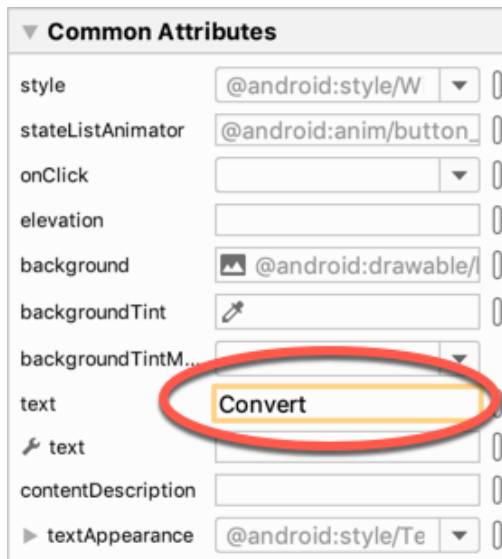


Figure 3-13

The second text property with a wrench next to it allows a text property to be set which only appears within the Layout Editor tool but is not shown at runtime. This is useful for testing 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

Creating an Example Android App in Android Studio

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

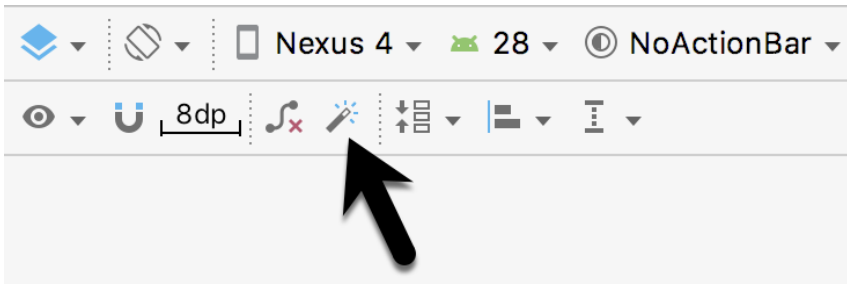


Figure 3-14

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 3-15. Obviously, this is indicating potential problems with the layout. For details on any problems, click on the button:

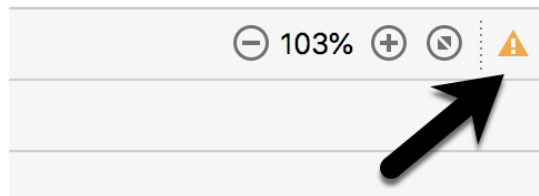


Figure 3-15

When clicked, a panel (Figure 3-16) will appear describing the nature of the problems and offering some possible corrective measures:

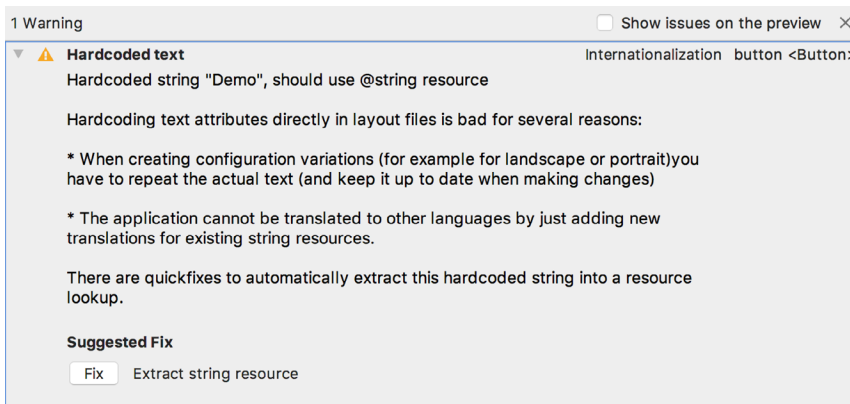


Figure 3-16

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 3-17). 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.

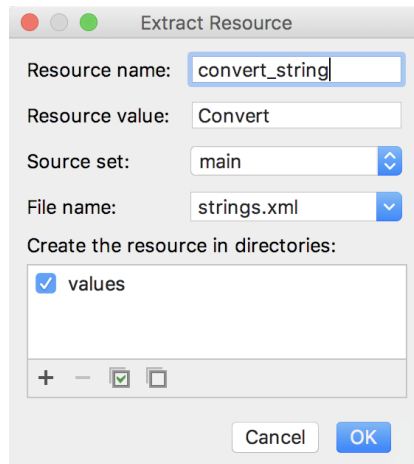


Figure 3-17

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

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

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:

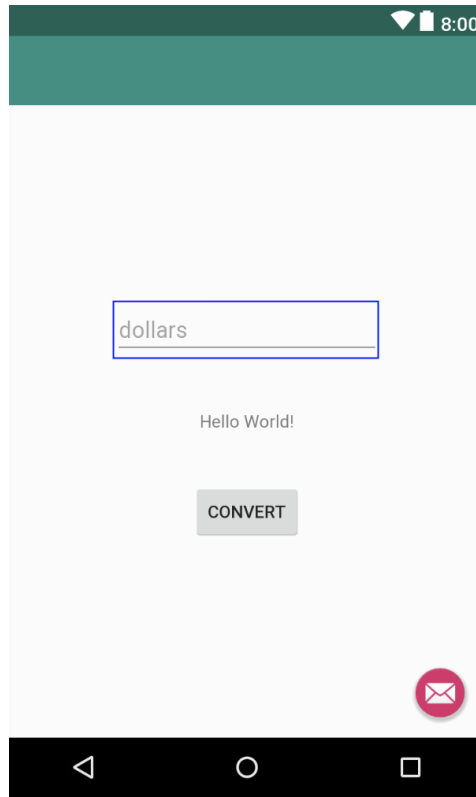


Figure 3-18

The code written later in this chapter will need to access the dollar value entered by the user into the EditText field. It will do this by referencing the id assigned to the widget in the user interface layout. The default id assigned to the widget by Android Studio can be viewed and changed from within the Attributes tool window when the widget is selected in the layout as shown in Figure 3-19:

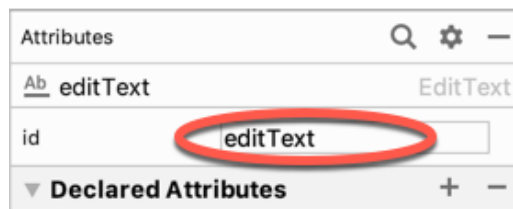


Figure 3-19

Change the id to *dollarText* before proceeding.

3.6 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 *content_main.xml* file using the Layout Editor tool. In fact, all that the Layout Editor was doing was providing a user-friendly way to edit the underlying XML content of the file. In practice, there is no reason why you cannot modify the XML directly in order to make user interface changes and, in some instances, this may actually be quicker than using the Layout Editor tool. At the bottom of the Layout Editor panel are two tabs labeled *Design*

and *Text* respectively. To switch to the XML view simply select the *Text* tab as shown in Figure 3-20:

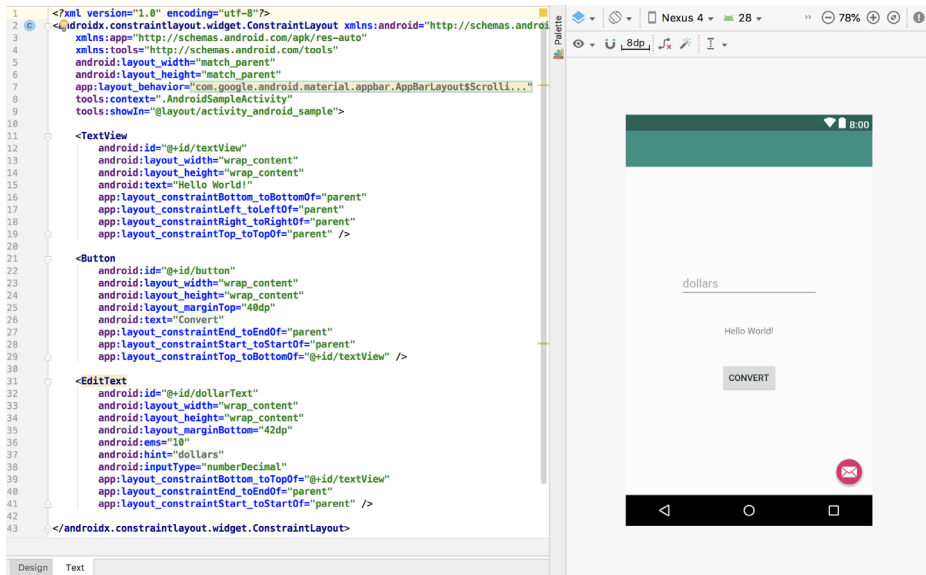


Figure 3-20

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 *Button* object. We can also see 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.

One of the more powerful features of Android Studio can be found to the right-hand side of the XML editing panel. If the panel is not visible, display it by selecting the *Preview* button located along the right-hand edge of the Android Studio window. This is the Preview panel and shows the current visual state of the layout. As changes are made to the XML layout, these will be reflected in the preview panel. The layout may also be modified visually from within the Preview panel with the changes appearing in the XML listing. To see this in action, 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"?>
<android.support.constraint.ConstraintLayout xmlns:android="http://schemas.
android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:layout_behavior="@string/appbar_scrolling_view_behavior"
    tools:context=".MainActivity"
    tools:showIn="@layout/activity_android_sample"
    android:background="#ff2438" >
    .
    .
</android.support.constraint.ConstraintLayout>
```

Note that the color of the preview changes in real-time to match the new setting in the XML file. Note also that a

Creating an Example Android App in Android Studio

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. Change the color value to #a0ff28 and note that both the small square in the margin and the preview change to green.

Finally, use the Project view 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="action_settings">Settings</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 Text 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.

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

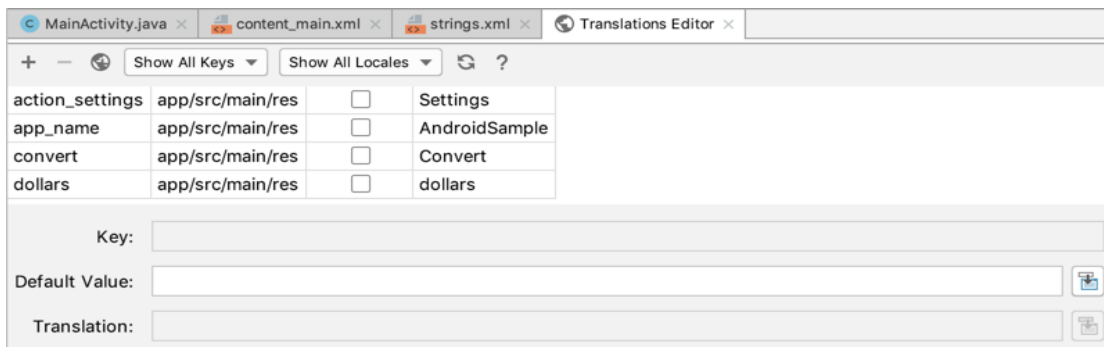


Figure 3-21

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

3.7 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:

Declared Attributes		+	-
layout_width	wrap_content	▼	0
layout_height	wrap_content	▼	0
layout_constraintTr	@+id/textView	▼	0
layout_constraintEr	parent	▼	0
layout_constraintSt	parent	▼	0
layout_marginTop	87dp		0
id	button		
onClick	convertCurrency	▶	0
text	@string/convert		

Figure 3-22

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 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 android.os.Bundle;
import android.support.design.widget.FloatingActionButton;
import android.support.design.widget.Snackbar;
import android.support.v7.app.AppCompatActivity;
import android.support.v7.widget.Toolbar;
import android.view.View;
import android.view.Menu;
import android.view.MenuItem;
import android.widget.EditText;
import android.widget.TextView;

public class MainActivity extends AppCompatActivity {
    .
    .

    public void convertCurrency(View view) {

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

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

            Float dollarValue = Float.valueOf(dollarText.getText().toString());
            Float euroValue = dollarValue * 0.85F;
            textView.setText(euroValue.toString());
        } else {
            textView.setText("No Value");
        }
    }
}
```

```
.  
.   
}
```

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.

The project is now complete and ready to run, a task that will be performed in the next chapter after an AVD emulator session as been created for testing purposes.

3.8 Summary

While not excessively complex, a number of steps are involved in setting up an Android development environment. Having performed those steps, it is worth working through a simple example to make sure the environment is correctly installed and configured. In this chapter, we have created a simple application and then used the Android Studio Layout Editor tool to modify the user interface layout. In doing so, we explored the importance of using resources wherever possible, particularly in the case of string values, and briefly touched on the topic of layouts. Next we looked at the underlying XML that is used to store the user interface designs of Android applications.

While it is useful to be able to preview a layout from within the Android Studio Layout Editor tool, there is no substitute for testing an application by compiling and running it.

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

In the course of developing Android apps in Android Studio it will be necessary to compile and run an application multiple times. An Android application may be tested by installing and running it either on a physical device or in an *Android Virtual Device (AVD)* emulator environment. Before an AVD can be used, it must first be created and configured to match the specifications of a particular device model. The goal of this chapter, therefore, is to work through the steps involved in creating such a virtual device using the Nexus 5X phone as a reference example.

4.1 About Android Virtual Devices

AVDs are essentially emulators that allow Android applications to be tested without the necessity to install the application on a physical Android based device. An AVD may be configured to emulate a variety of hardware features including options such as screen size, memory capacity and the presence or otherwise of features such as a camera, GPS navigation support or an accelerometer. As part of the standard Android Studio installation, a number of emulator templates are installed allowing AVDs to be configured for a range of different devices. Additional templates may be loaded or custom configurations 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. Check the online developer documentation for your device to find out if emulator definitions are available for download and installation into the AVD environment.

When launched, an AVD will appear as a window containing an emulated Android device environment. Figure 4-1, for example, shows an AVD session configured to emulate the Google Nexus 5X model.

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

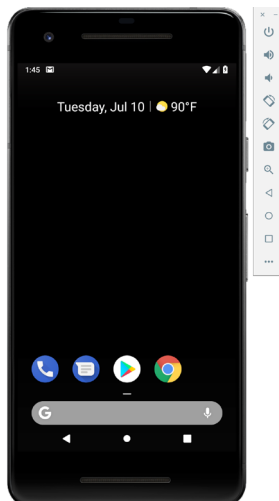


Figure 4-1

4.2 Creating a New AVD

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

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

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

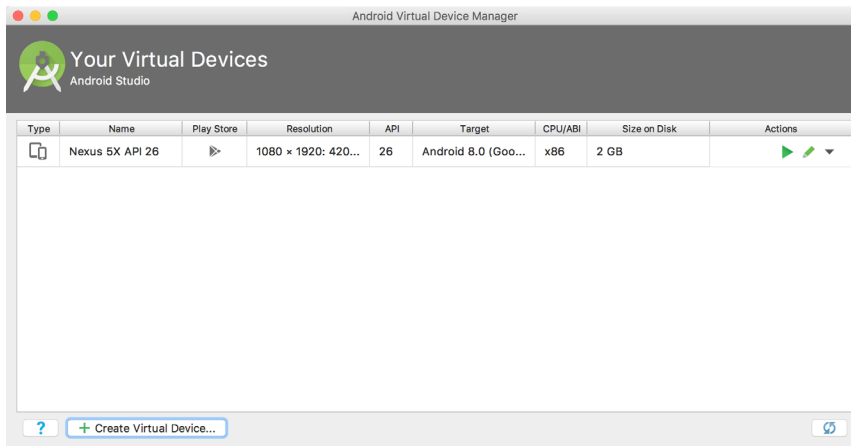


Figure 4-2

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

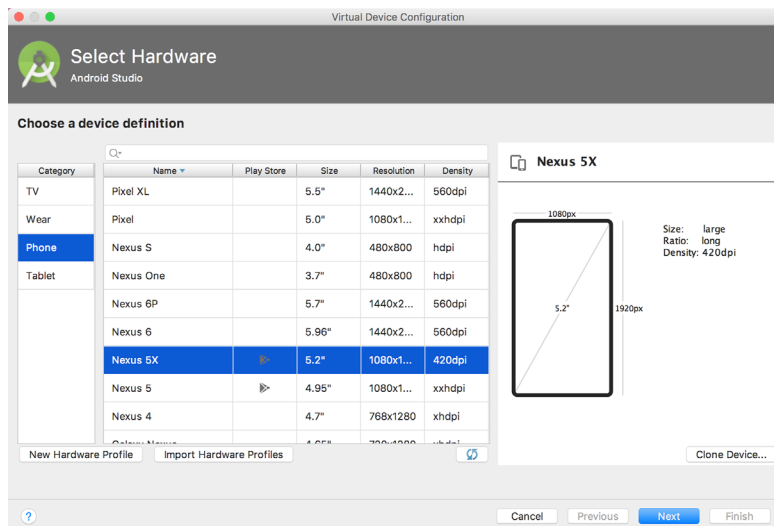


Figure 4-3

Within the dialog, perform the following steps to create a Nexus 5X compatible emulator:

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

2. Select the *Nexus 5X* device option and click *Next*.
3. On the System Image screen, select the latest version of Android (at time of writing this is API level 28, Android 9.0 with Google Play) for the *x86* ABI. Note that if the system image has not yet been installed a *Download* link will be provided next to the Release Name. Click this link to download and install the system image before selecting it. If the image you need is not listed, click on the *x86 images* and *Other images* tabs to view alternative lists.
4. Click *Next* to proceed and enter a descriptive name (for example *Nexus 5X API 28*) into the name field or simply accept the default name.
5. Click *Finish* to create the AVD.
6. With the AVD created, the AVD Manager may now be closed. If future modifications to the AVD are necessary, simply re-open the AVD Manager, select the AVD from the list and click on the pencil icon in the *Actions* column of the device row in the AVD Manager.

4.3 Starting the Emulator

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

Although the emulator probably defaulted to appearing in portrait orientation, this and other default options can be changed. Within the AVD Manager, select the new Nexus 5X entry and click on the pencil icon in the *Actions* column of the device row. In the configuration screen locate the *Startup and orientation* section and change the orientation setting. Exit and restart the emulator session to see this change take effect. More details on the emulator are covered in the next chapter (“*Using and Configuring the Android Studio AVD Emulator*”).

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

4.4 Running the Application in the AVD

With an AVD emulator configured, the example *AndroidSample* application created in the earlier chapter now can be compiled and run. With the *AndroidSample* project loaded into Android Studio, simply click on the run button represented by a green triangle located in the Android Studio toolbar as shown in Figure 4-4 below, select the *Run -> Run 'app'* menu option or use the *Ctrl-R* keyboard shortcut:

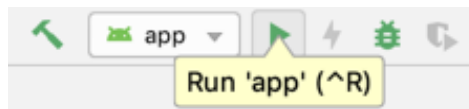


Figure 4-4

By default, Android Studio will respond to the run request by displaying the *Select Deployment Target* dialog. This provides the option to execute the application on an AVD instance that is already running, or to launch a new AVD session specifically for this application. Figure 4-5 lists the previously created Nexus 5X AVD as a running device as a result of the steps performed in the preceding section. With this device selected in the dialog, click on *OK* to install and run the application on the emulator.

Creating an Android Virtual Device (AVD) in Android Studio

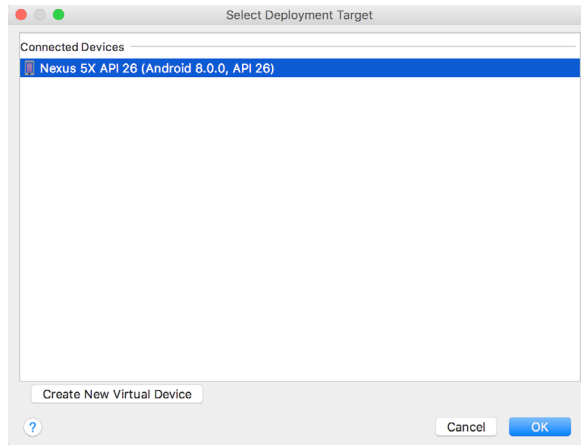


Figure 4-5

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



Figure 4-6

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

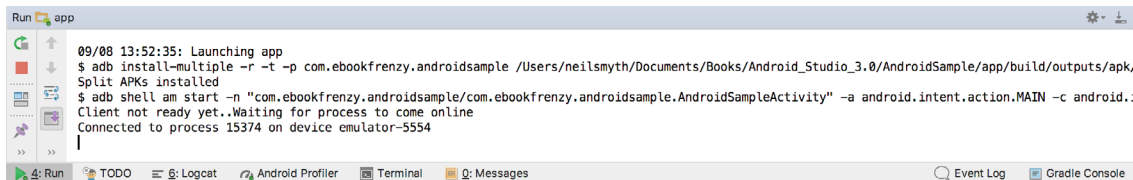


Figure 4-7

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

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

4.5 Run/Debug Configurations

A particular project can be configured such that a specific device or emulator is used automatically each time it is run from within Android Studio. This avoids the necessity to make a selection from the device chooser each time the application is executed. To review and modify the Run/Debug configuration, click on the button to the left of the run button in the Android Studio toolbar and select the *Edit Configurations...* option from the resulting menu:

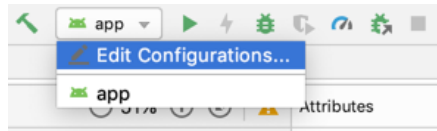


Figure 4-8

In the *Run/Debug Configurations* dialog, the application may be configured to always use a preferred emulator by selecting *Emulator* from the *Target* menu located in the *Deployment Target Options* section and selecting the emulator from the drop down menu. Figure 4-9, for example, shows the *AndroidSample* application configured to run by default on the previously created *Nexus 5X* emulator:

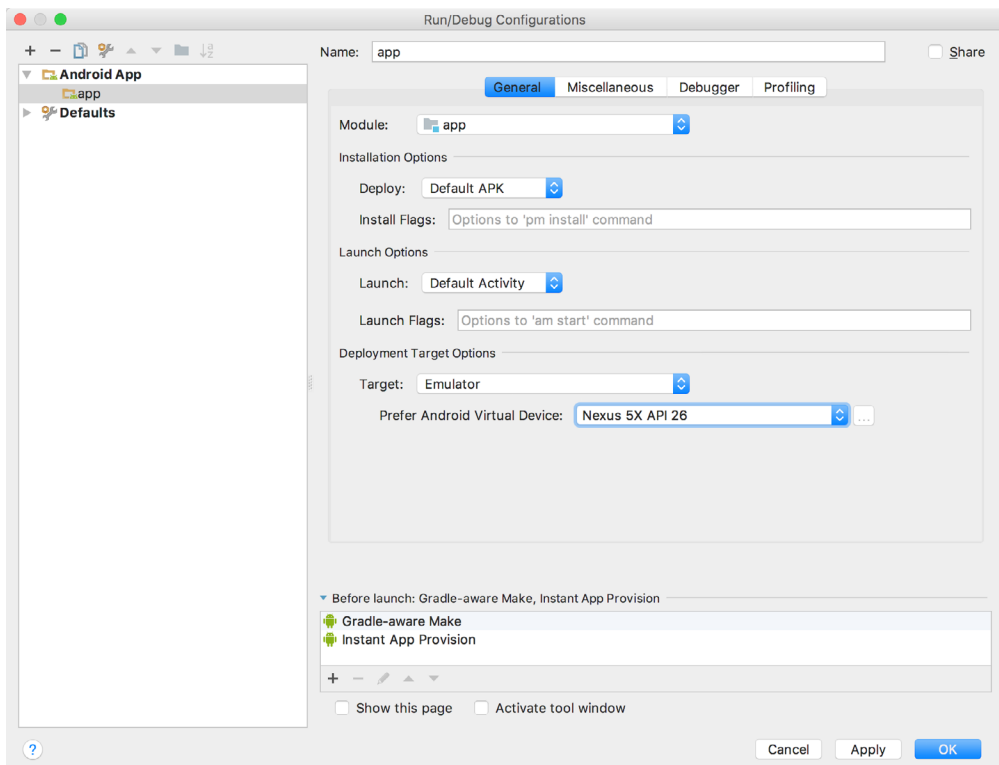


Figure 4-9

Creating an Android Virtual Device (AVD) in Android Studio

Be sure to switch the Target menu setting back to “Open Select Deployment Target Dialog” mode before moving on to the next chapter of the book.

4.6 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:

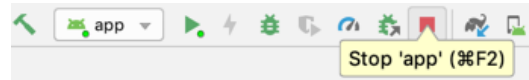


Figure 4-10

An app may also be terminated using the Logcat tool window. Begin by displaying the *Logcat* tool window using the window bar button that becomes available when the app is running. Once the Logcat tool window appears, select the *androidsample* app menu highlighted in Figure 4-11 below:

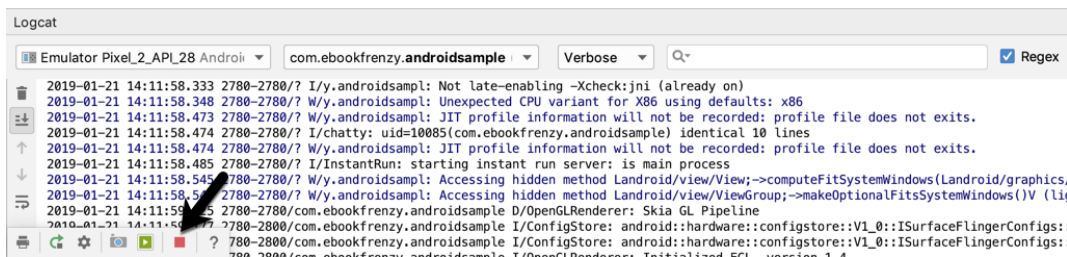


Figure 4-11

With the process selected, stop it by clicking on the red *Terminate Application* button in the toolbar to the left of the process list indicated by the arrow in the above figure.

4.7 AVD Command-line Creation

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

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

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

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

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

On macOS or Linux, execute the following command:

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

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

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

```
avdmanager list targets
```

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

```
Available Android targets:
```

```
-----
id: 1 or "android-28"
    Name: Android API 28
    Type: Platform
    API level: 28
    Revision: 3
-----
```

```
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 *Nexus9* using the target ID for the Android API level 26 device using the x86 ABI, the following command may be used:

```
avdmanager create avd -n Nexus9 -k "system-images;android-26;google_apis;x86"
```

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

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

```
avdmanager list avd
```

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

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

Creating an Android Virtual Device (AVD) in Android Studio

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

4.8 Android Virtual Device Configuration Files

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

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

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

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

4.9 Moving and Renaming an Android Virtual Device

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

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

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

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

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

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

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

4.10 Summary

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

5. Using and Configuring the Android Studio AVD Emulator

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

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

5.1 The Emulator Environment

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



Figure 5-1

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

5.2 The Emulator Toolbar Options

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

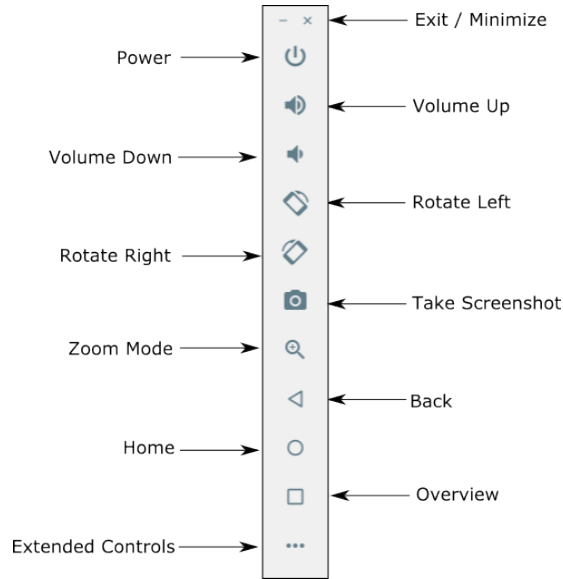


Figure 5-2

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

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

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

- **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:

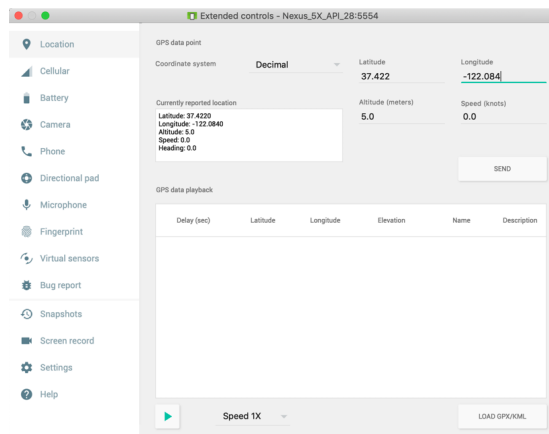


Figure 5-3

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.

A single location is transmitted to the emulator when the *Send* button is clicked. The transmission of GPS data points begins once the “play” button located beneath the data table is selected. The speed at which the GPS data points are fed to the emulator can be controlled using the speed menu adjacent to the play button.

5.5.2 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.

5.5.3 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.

5.5.4 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 Phone

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

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

5.5.6 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.

5.5.7 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.

5.5.8 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.

5.5.9 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.

5.5.10 Snapshots

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

5.5.11 Screen Record

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

5.5.12 Settings

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

5.5.13 Help

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

5.6 Working with Snapshots

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

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

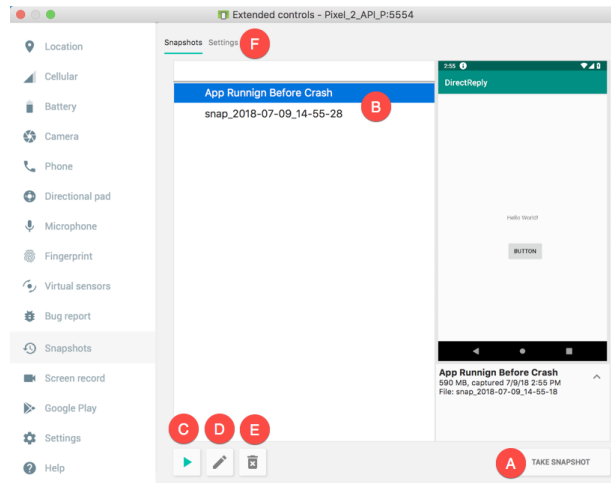


Figure 5-4

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

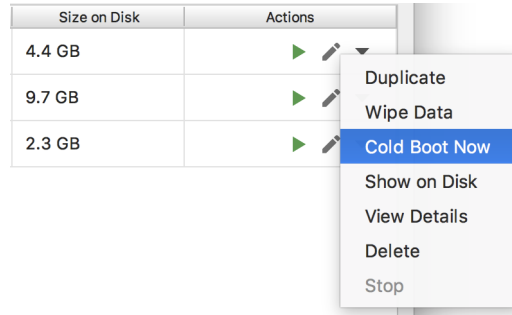


Figure 5-5

5.7 Configuring Fingerprint Emulation

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

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

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

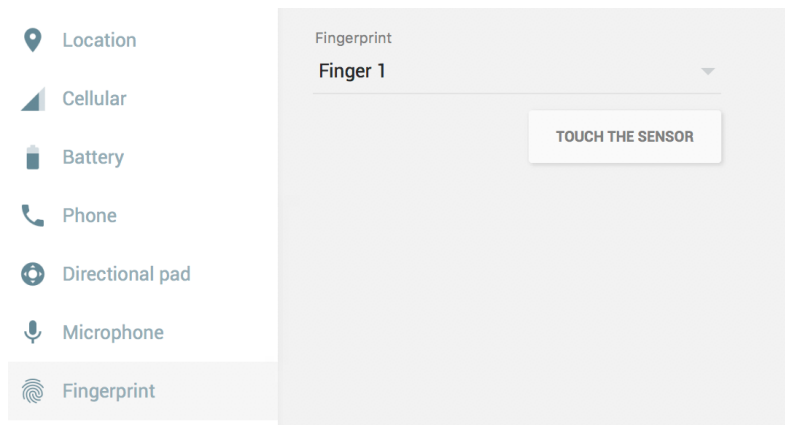


Figure 5-6

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

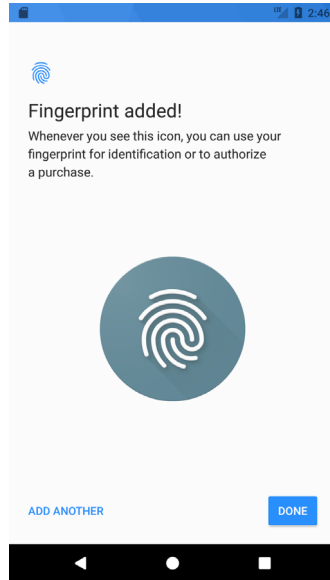


Figure 5-7

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

5.8 Summary

Android Studio 3.4 contains a new and improved Android Virtual Device emulator environment designed to make it easier to test applications without the need to run on a physical Android device. This chapter has provided a brief tour of the emulator and highlighted key features that are available to configure and customize the environment to simulate different testing conditions

6. A Tour of the Android Studio User Interface

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

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

6.1 The Welcome Screen

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

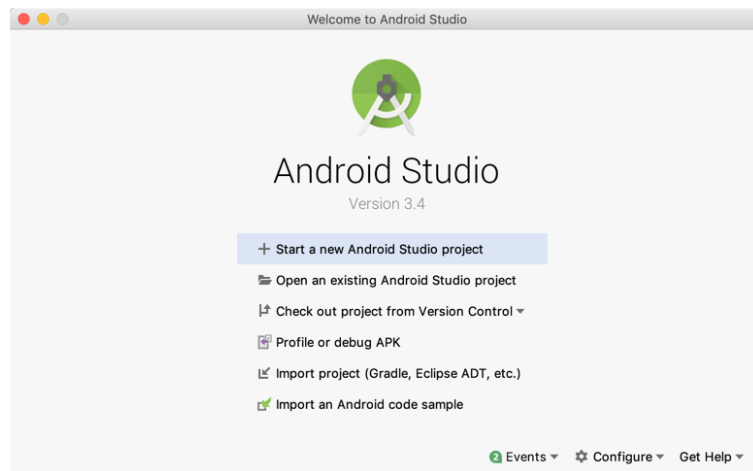


Figure 6-1

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

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

6.2 The Main Window

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

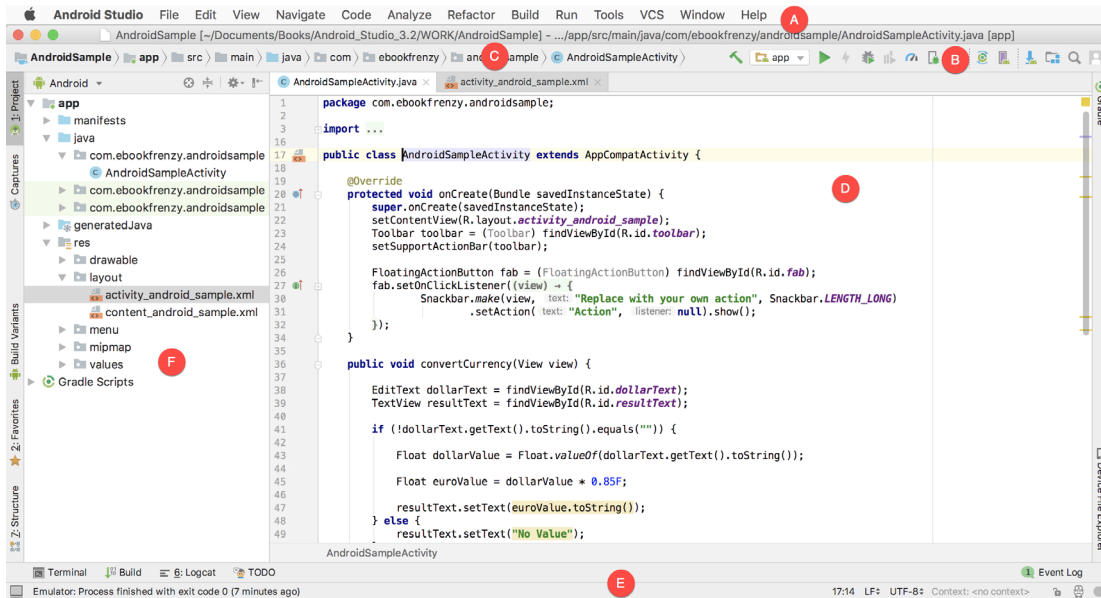


Figure 6-2

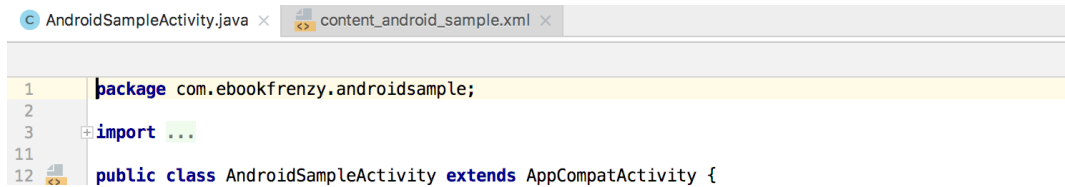
The various elements of the main window can be summarized as follows:

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

B – Toolbar – A selection of shortcuts to frequently performed actions. The toolbar buttons provide quicker access to a select group of menu bar actions. The toolbar can be customized by right-clicking on the bar and selecting the *Customize Menus and Toolbars...* menu option.

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

D – Editor Window – The editor window displays the content of the file on which the developer is currently working. What gets displayed in this location, however, is subject to context. When editing code, for example, the code editor will appear. When working on a user interface layout file, on the other hand, the user interface Layout Editor tool will appear. When multiple files are open, each file is represented by a tab located along the top edge of the editor as shown in Figure 6-3.



```

1 package com.ebookfrenzy.androidsample;
2
3 import ...
11
12 public class AndroidSampleActivity extends AppCompatActivity {

```

Figure 6-3

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

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

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

6.3 The Tool Windows

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

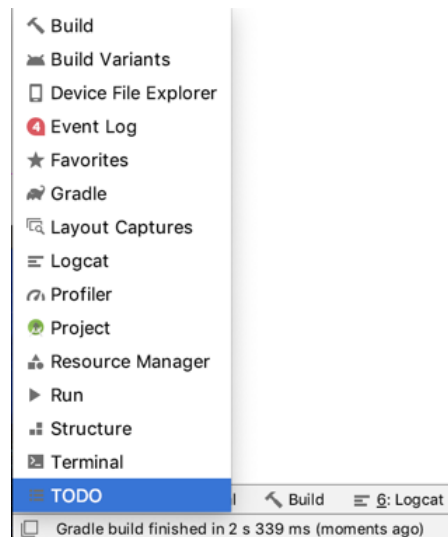


Figure 6-4

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

Alternatively, a set of *tool window bars* can be displayed by clicking on the quick access menu icon in the status bar. These bars appear along the left, right and bottom edges of the main window (as indicated by the arrows in Figure 6-5) and contain buttons for showing and hiding each of the tool windows. When the tool window bars

A Tour of the Android Studio User Interface

are displayed, a second click on the button in the status bar will hide them.

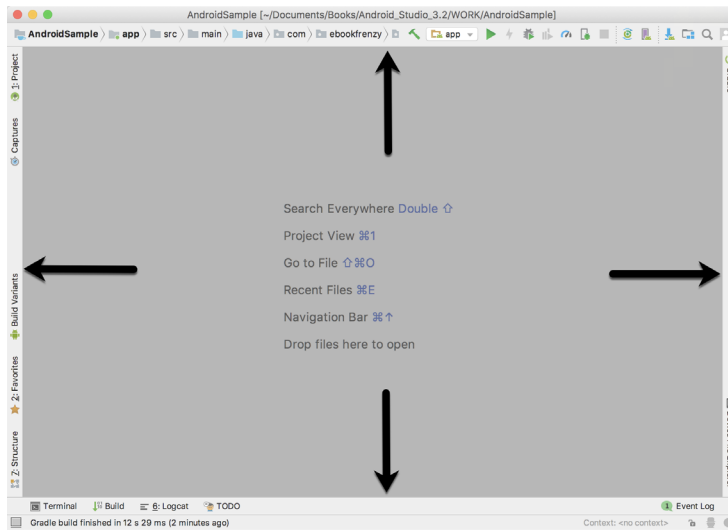


Figure 6-5

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

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

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

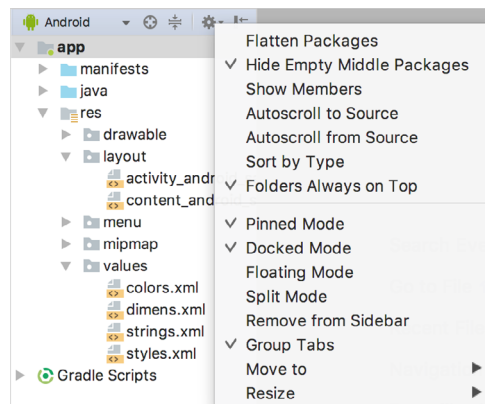


Figure 6-6

All of the windows also include a far right button on the toolbar providing an additional way to hide the tool

window from view. A search of the items within a tool window can be performed simply by giving that window focus by clicking in it and then typing the search term (for example the name of a file in the Project tool window). A search box will appear in the window's tool bar and items matching the search highlighted.

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

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

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

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

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

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

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

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

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

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

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

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

Gradle – The Gradle tool window provides a view onto the Gradle tasks that make up the project build configuration. The window lists the tasks that are involved in compiling the various elements of the project into an executable application. Right-click on a top level Gradle task and select the *Open Gradle Config* menu option

to load the Gradle build file for the current project into the editor. Gradle will be covered in greater detail later in this book.

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

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

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

6.4 Android Studio Keyboard Shortcuts

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

6.5 Switcher and Recent Files Navigation

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

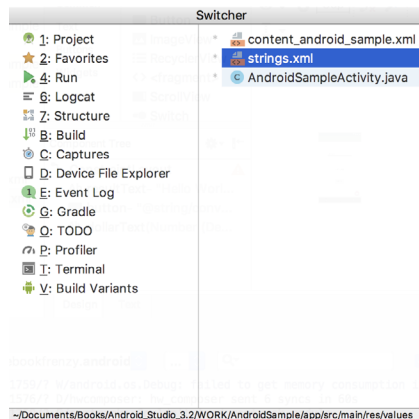


Figure 6-7

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

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

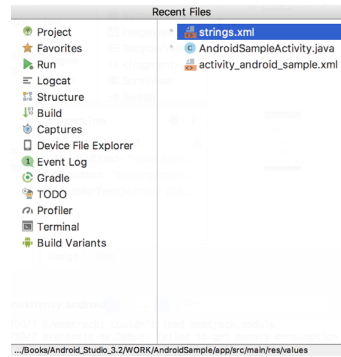


Figure 6-8

6.6 Changing the Android Studio Theme

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

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

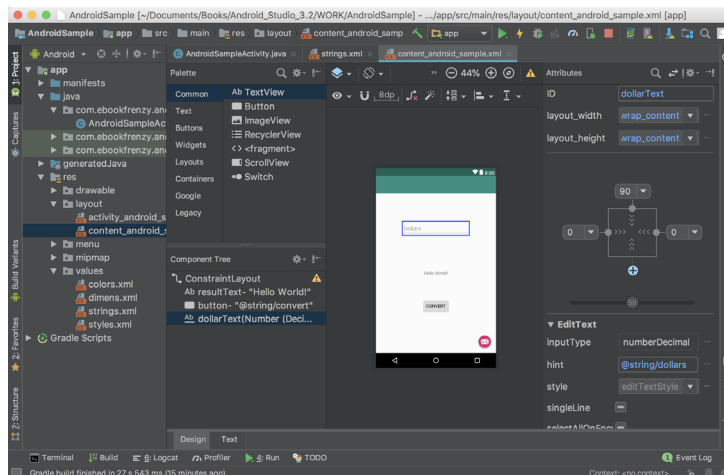


Figure 6-9

6.7 Summary

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

There are very few actions within Android Studio which cannot be triggered via a keyboard shortcut. A keypad 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 a number of Android features that are only available on physical Android devices.

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

7.1 An Overview of the Android Debug Bridge (ADB)

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

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

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

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

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

7.2 Enabling ADB on Android based Devices

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

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

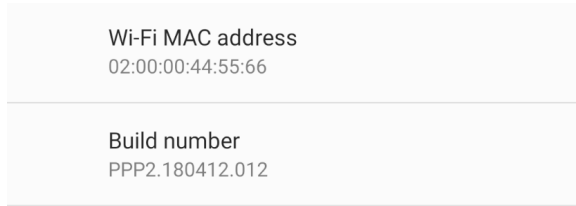


Figure 7-1

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

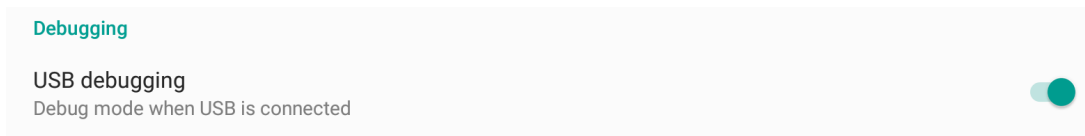


Figure 7-2

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

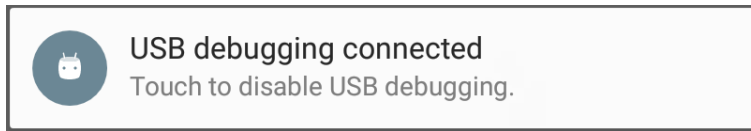


Figure 7-3

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

7.2.1 macOS ADB Configuration

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

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

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

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

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

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

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

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

7.2.2 Windows ADB Configuration

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

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

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

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

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

```
adb devices
```

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

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

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

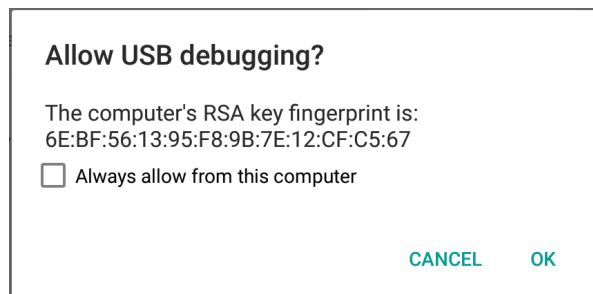


Figure 7-4

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

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

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

Testing Android Studio Apps on a Physical Android Device

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

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

```
android update adb
```

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

7.2.3 Linux adb Configuration

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

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

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

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

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

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

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

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

7.3 Testing the adb Connection

Assuming that the adb configuration has been successful on your chosen development platform, the next step is to try running the test application created in the chapter entitled “*Creating an Example Android App in Android Studio*” on the device.

Launch Android Studio, open the AndroidSample project and, once the project has loaded, click on the run button located in the Android Studio toolbar (Figure 7-5).

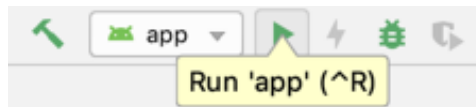


Figure 7-5

Assuming that the project has not previously been configured to run automatically in an emulator environment, the deployment target selection dialog will appear with the connected Android device listed as a currently running device. Figure 7-6, for example, lists a Nexus 9 device as a suitable target for installing and executing the application.

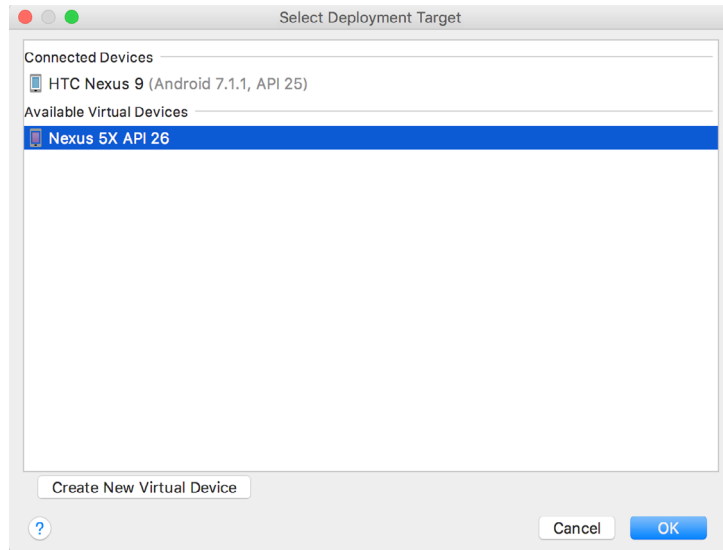


Figure 7-6

To make this the default device for testing, enable the *Use same device for future launches* option. With the device selected, click on the *OK* button to install and run the application on the device. As with the emulator environment, diagnostic output relating to the installation and launch of the application on the device will be logged in the Run tool window.

7.4 Summary

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

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

8. The Basics of the Android Studio Code Editor

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

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

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

8.1 The Android Studio Editor

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

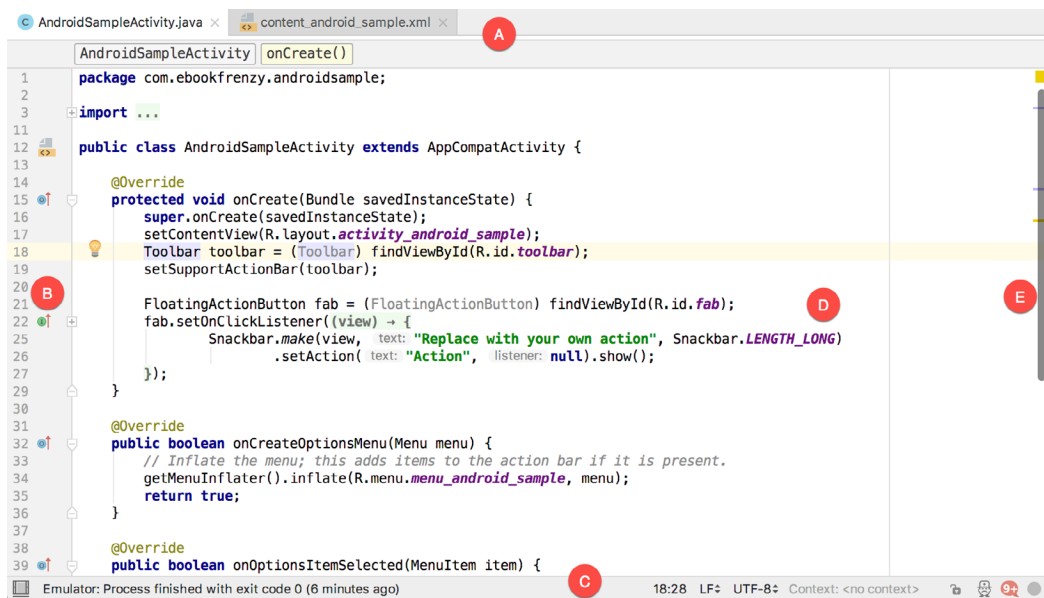


Figure 8-1

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

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

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

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

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

C – 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.

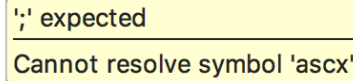
D – The Editor Area – This is the main area where the code is displayed, entered and edited by the user. Later sections of this chapter will cover the key features of the editing area in detail.

E – The Validation and Marker Sidebar – Android Studio incorporates a feature referred to as “on-the-fly code analysis”. What this essentially means is that as you are typing code, the editor is analyzing the code to check for warnings and syntax errors. The indicator at the top of the validation sidebar will change from a green check mark (no warnings or errors detected) to a yellow square (warnings detected) or red alert icon (errors have been detected). Clicking on this indicator will display a popup containing a summary of the issues found with the code in the editor as illustrated in Figure 8-2:



Figure 8-2

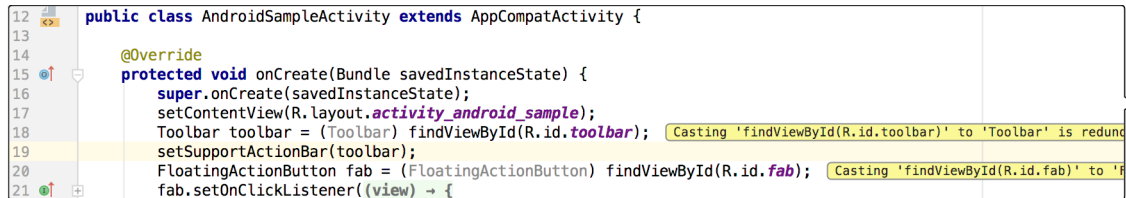
The sidebar also displays markers at the locations where issues have been detected using the same color coding. Hovering the mouse pointer over a marker when the line of code is visible in the editor area will display a popup containing a description of the issue (Figure 8-3):



```
';' expected
Cannot resolve symbol 'ascx'
```

Figure 8-3

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 8-4) allowing it to be viewed without the necessity to scroll to that location in the editor:



```
public class AndroidSampleActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_android_sample);
        Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
        setSupportActionBar(toolbar);
        FloatingActionButton fab = (FloatingActionButton) findViewById(R.id.fab);
        fab.setOnClickListener((view) -> {
            // ...
        });
    }
}
```

Figure 8-4

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.

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 8-5, for example, shows the splitter in action with the editor split into three panels:

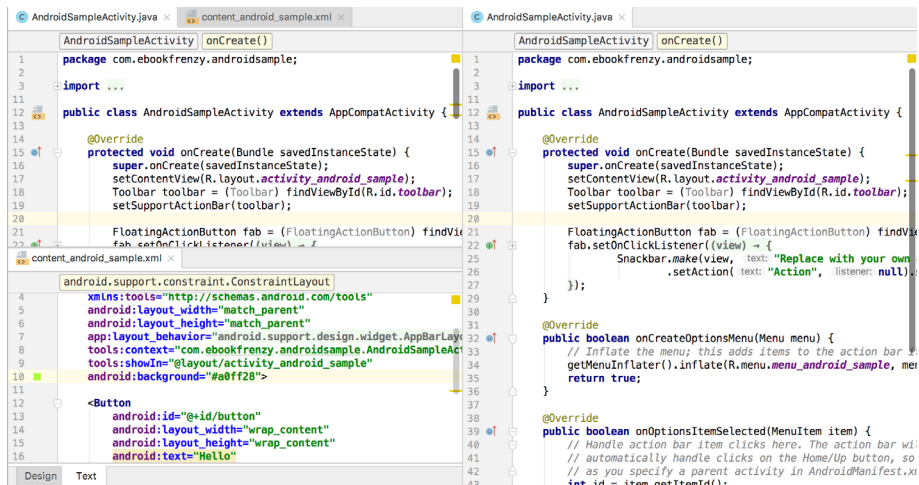


Figure 8-5

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 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 8-6, for example, the editor is suggesting possibilities for the beginning of a String declaration:

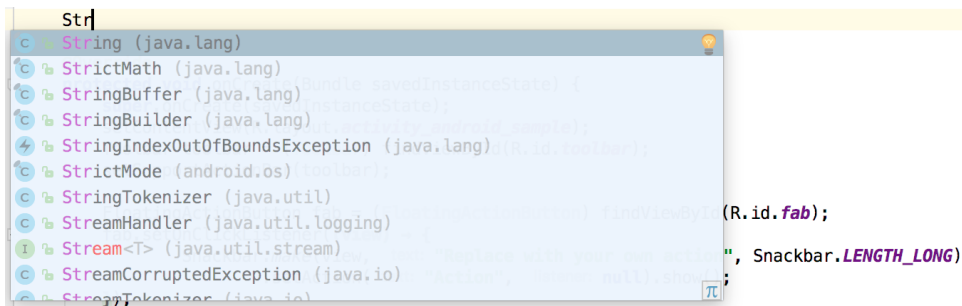


Figure 8-6

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 8-7:

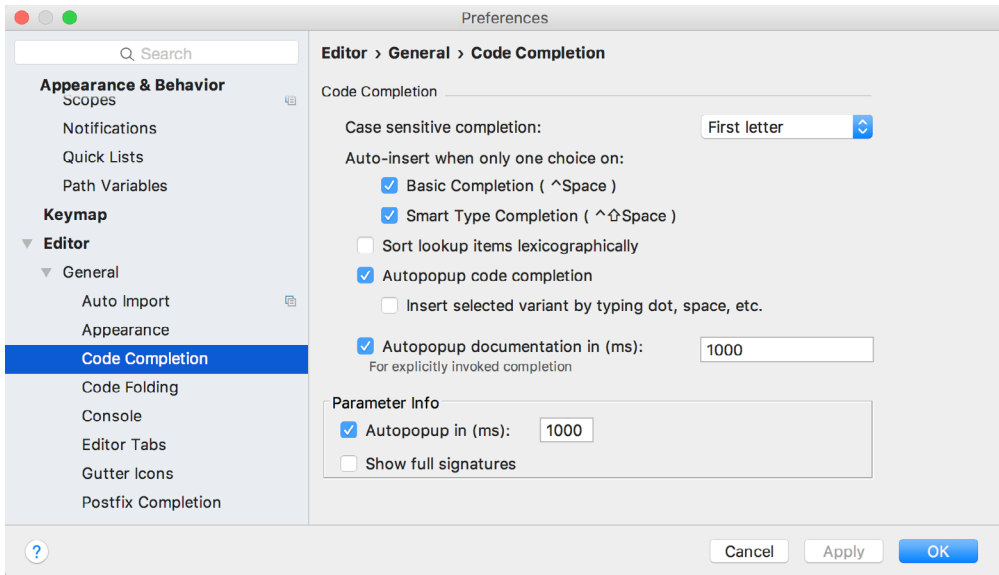


Figure 8-7

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 8-8

8.6 Parameter Name Hints

The code editor may be configured to display parameter name hints within method calls. Figure 8-9, 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 8-9

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

8.7 Code Generation

In addition to completing code as it is typed the editor can, under certain conditions, also generate code for you. The list of available code generation options shown in Figure 8-10 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.

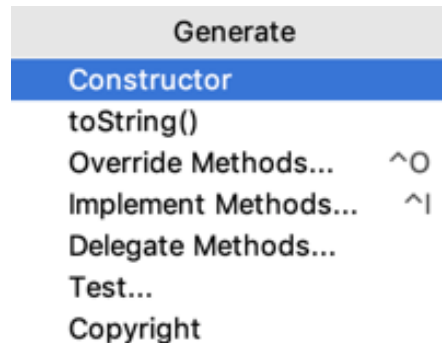


Figure 8-10

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:

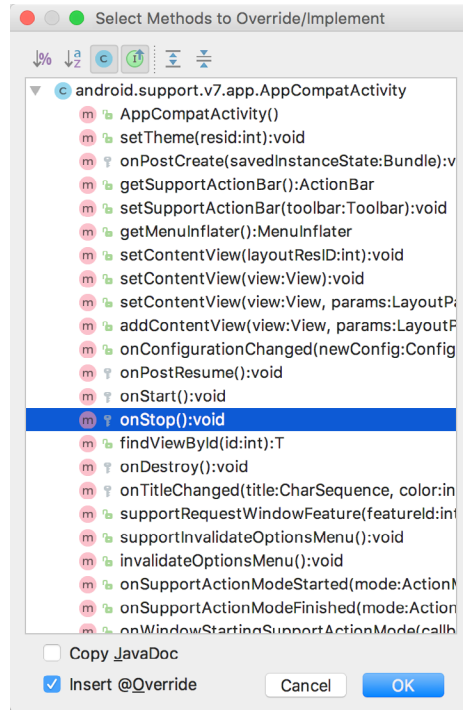


Figure 8-11

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 8-12, for example, highlights the start and end markers for a method declaration which is not currently folded:

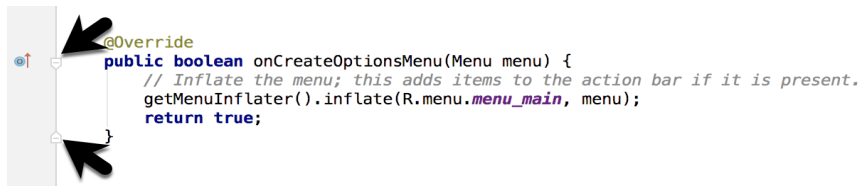


Figure 8-12

Clicking on either of these markers will fold the statement such that only the signature line is visible as shown in Figure 8-13:

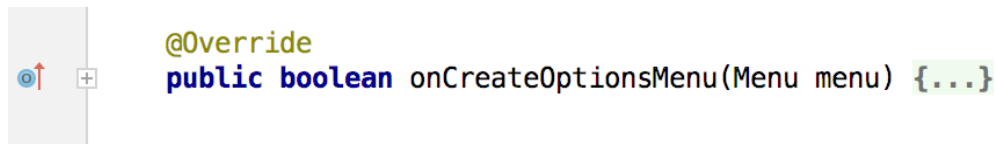


Figure 8-13

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 8-14. The editor will then display the lens overlay containing the folded code block:

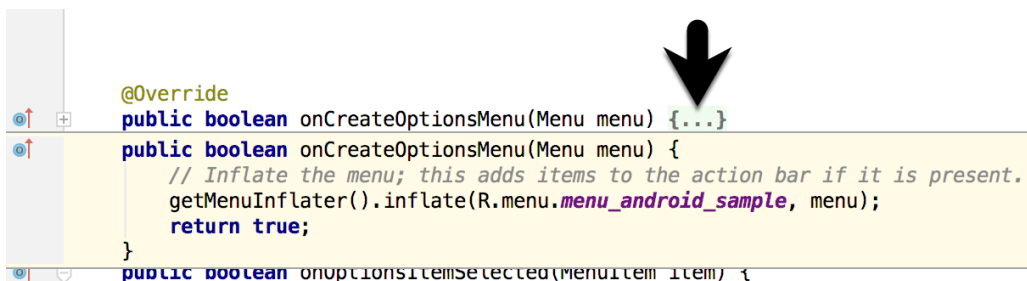


Figure 8-14

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 8-15):

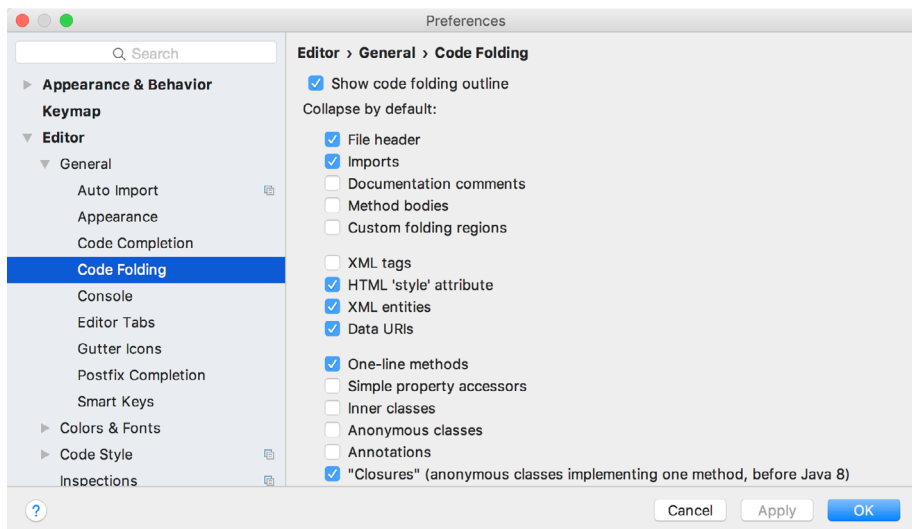


Figure 8-15

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 8-16, for example, shows the documentation for the Android Snackbar class.



Figure 8-16

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

8.10 Code Reformatting

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

To reformat source code, press the *Ctrl-Alt-L* (*Cmd-Opt-L* on macOS) keyboard shortcut sequence. To display the *Reformat Code* dialog (Figure 8-17) 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.

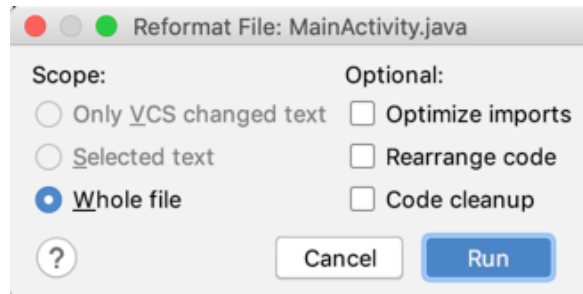


Figure 8-17

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.

The Basics of the Android Studio Code Editor

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 8-18) 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:

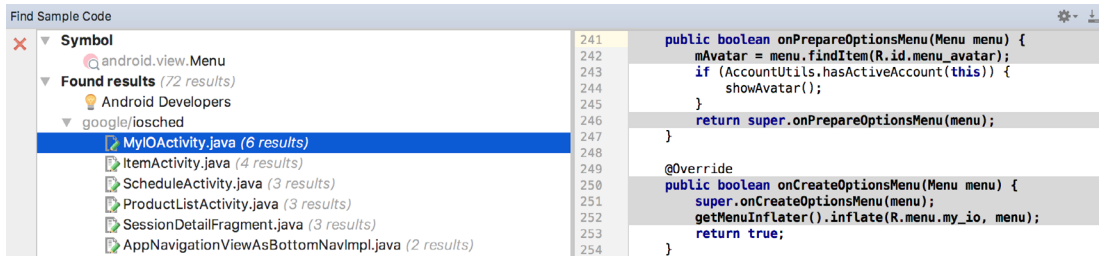


Figure 8-18

8.12 Summary

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

9. An Overview of the Android Architecture

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

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

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

9.1 The Android Software Stack

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

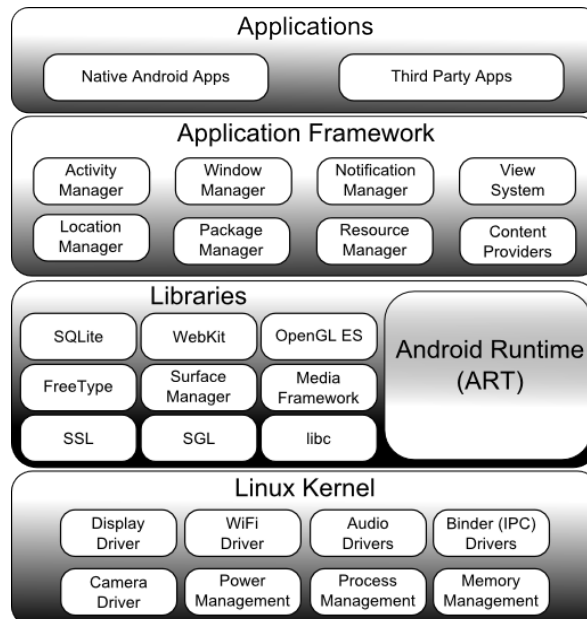


Figure 9-1

9.2 The Linux Kernel

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

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

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

9.3 Android Runtime – ART

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

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

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

9.4 Android Libraries

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

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

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

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

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

9.4.1 C/C++ Libraries

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

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

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

9.5 Application Framework

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

An Overview of the Android Architecture

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

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

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

10.7 Content Providers

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

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

10.8 The Application Manifest

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

10.9 Application Resources

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

10.10 Application Context

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

10.11 Summary

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

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

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

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

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

11. 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. In order 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.

11.1 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 in order 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.

11.2 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 11-1, a process can be in one of the following five states at any given time:

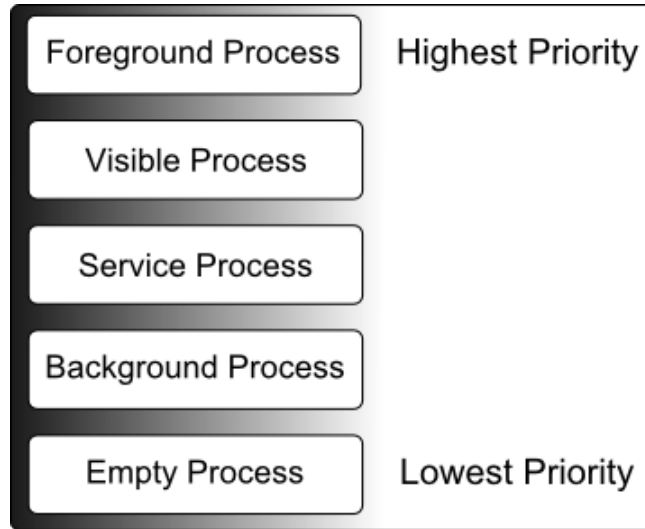


Figure 11-1

11.2.1 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.

11.2.2 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.

11.2.3 Service Process

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

11.2.4 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 in the event that 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.

11.2.5 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.

11.3 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.

11.4 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*.

11.5 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 11-2.

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. In the event that 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.

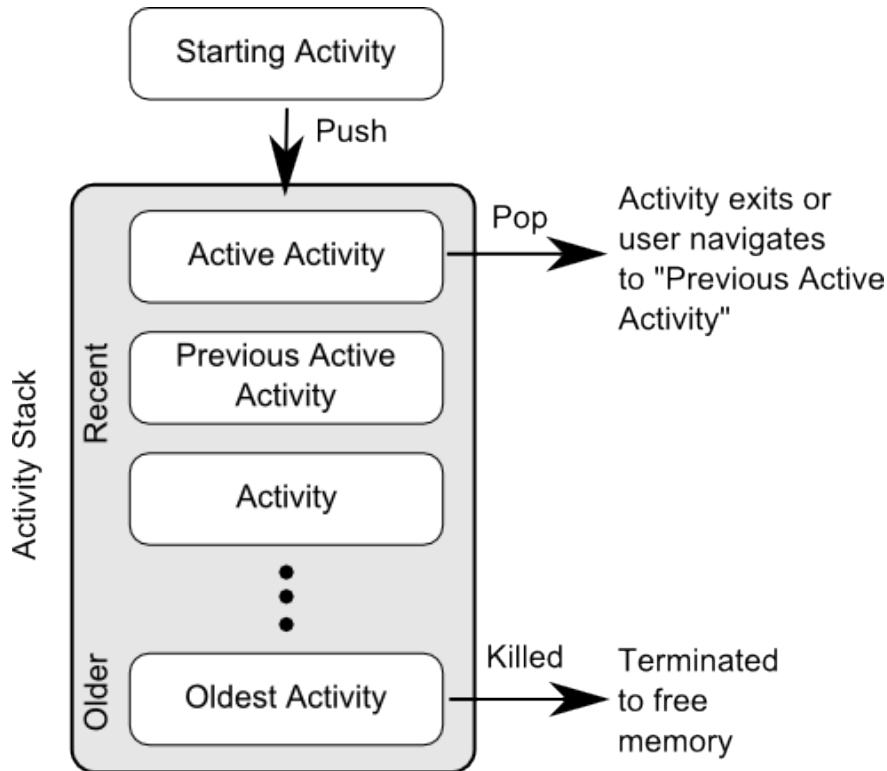


Figure 11-2

11.6 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.

11.7 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 in order 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.

11.8 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*”.

11.9 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 in order 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.

12. 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*.

12.1 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.

12.2 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 simple *AndroidSample* project created in “*Creating an Example Android App in Android Studio*”. 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.ebookfrenzy.androidsample;

import android.os.Bundle;
import android.support.design.widget.FloatingActionButton;
import android.support.design.widget.Snackbar;
import android.support.v7.app.AppCompatActivity;
import android.support.v7.widget.Toolbar;
```

Handling Android Activity State Changes

```
import android.view.View;
import android.view.Menu;
import android.view.MenuItem;
import android.widget.EditText;
import android.widget.TextView;

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_android_sample);
        Toolbar toolbar = findViewById(R.id.toolbar);
        setSupportActionBar(toolbar);

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

    public void convertCurrency(View view) {

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

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

            Float dollarValue = Float.valueOf(dollarText.getText().toString());
            Float euroValue = dollarValue * 0.85F;
            textView.setText(euroValue.toString());
        } else {
            textView.setText("No Value");
        }
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the action bar if it is present.
        getMenuInflater().inflate(R.menu.menu_main, menu);
    }
}
```



```

        return true;
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        // Handle action bar item clicks here. The action bar will
        // automatically handle clicks on the Home/Up button, so long
        // as you specify a parent activity in AndroidManifest.xml.
        int id = item.getItemId();

        //noinspection SimplifiableIfStatement
        if (id == R.id.action_settings) {
            return true;
        }

        return super.onOptionsItemSelected(item);
    }
}

```

When the project was created, we instructed Android Studio also to create an initial activity named *MainActivity*. 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 12-1, *AppCompatActivity* is clearly subclassed from the *FragmentActivity* class which is itself ultimately a subclass of the *Activity* class:

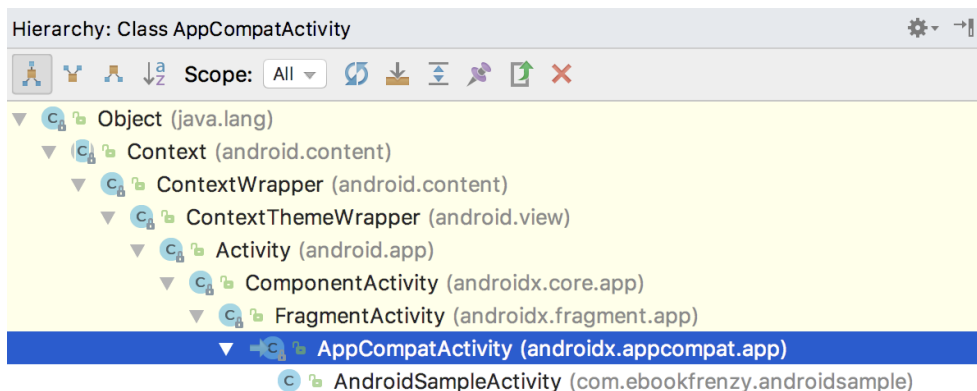


Figure 12-1

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 in order 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.

12.3 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.

12.4 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

Handling Android Activity State Changes

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*”.

12.5 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 12-2:

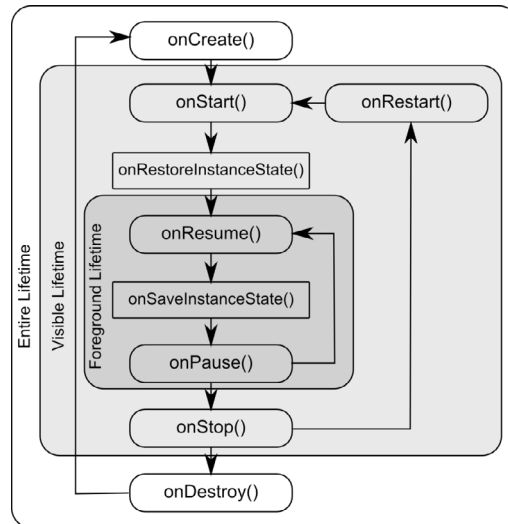


Figure 12-2

12.6 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">
```

12.7 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*”.

12.8 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 in order 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.

13. 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.

13.1 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 *Start a new Android Studio project* quick start option from the welcome screen and, within the resulting new project dialog, choose the Basic 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.

The next action to take involves the design of the content area of the user interface for the activity. This is stored in a file named *content_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.

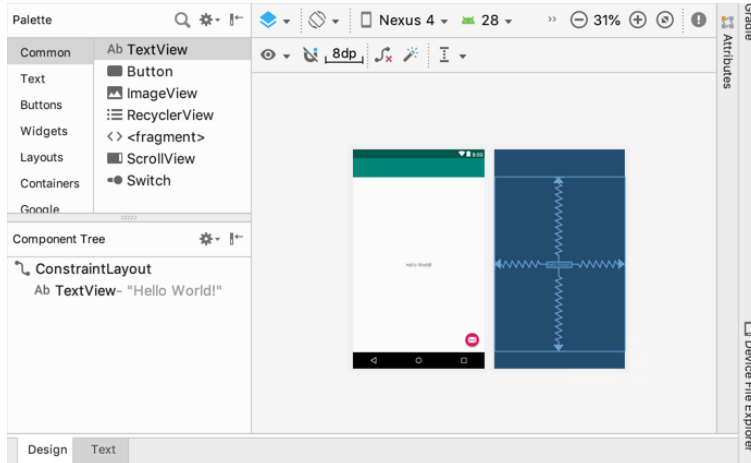


Figure 13-1

13.2 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 13-2.

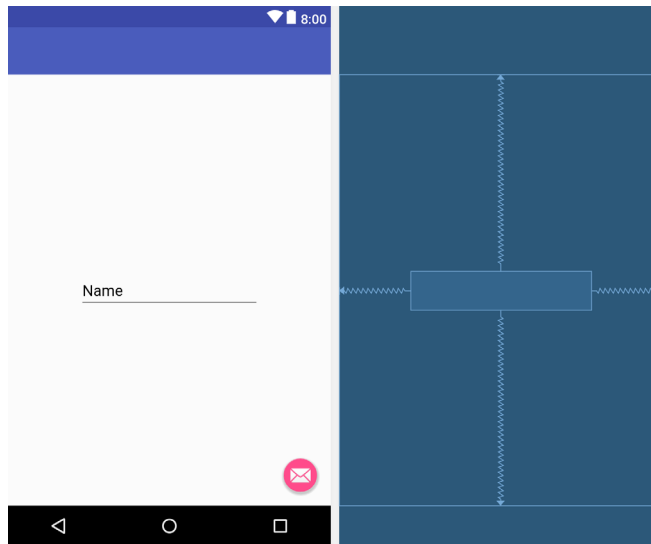


Figure 13-2

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 current setting to open the list of options and, within the list, switch off *textPersonName* and enable *text* before clicking on the OK button.

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.

13.3 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. In the event that 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. Once loaded the code should read as follows:

```
package com.ebookfrenzy.statechange;

import android.os.Bundle;
import android.support.design.widget.FloatingActionButton;
import android.support.design.widget.Snackbar;
import android.support.v7.app.AppCompatActivity;
import android.support.v7.widget.Toolbar;
import android.view.View;
import android.view.Menu;
import android.view.MenuItem;

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_state_change);
        Toolbar toolbar = findViewById(R.id.toolbar);
        setSupportActionBar(toolbar);

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

Android Activity State Changes by Example

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    // Inflate the menu; this adds items to the action bar if it is present.
    getMenuInflater().inflate(R.menu.menu_main, menu);
    return true;
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
    // Handle action bar item clicks here. The action bar will
    // automatically handle clicks on the Home/Up button, so long
    // as you specify a parent activity in AndroidManifest.xml.
    int id = item.getItemId();

    //noinspection SimplifiableIfStatement
    if (id == R.id.action_settings) {
        return true;
    }

    return super.onOptionsItemSelected(item);
}
}
```

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.os.Bundle;
import android.support.design.widget.FloatingActionButton;
import android.support.design.widget.Snackbar;
import android.support.v7.app.AppCompatActivity;
import android.view.View;
import android.view.Menu;
import android.view.MenuItem;
import android.util.Log;

public class MainActivity extends AppCompatActivity {

    private static final String TAG = "StateChange";

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

```

setContentview(R.layout.activity_state_change);
Toolbar toolbar = findViewById(R.id.toolbar);
setSupportActionBar(toolbar);

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

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

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

```

Android Activity State Changes by Example

```
}

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

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

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

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

13.4 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 13-3) 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 Filter*, on the other hand, will display all the messages generated by the device or emulator.

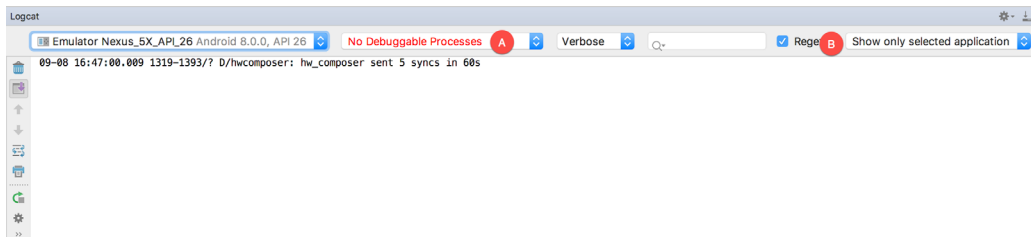


Figure 13-3

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 13-4), 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*).

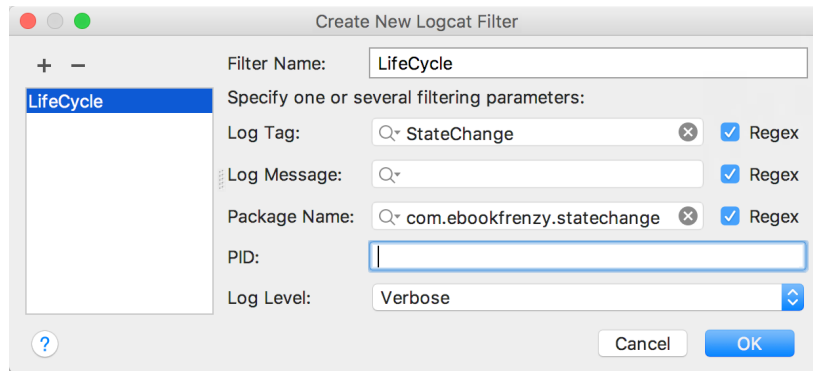


Figure 13-4

Enter the package identifier in the *Package Name* field (clicking on the search icon in the text field will drop down a menu from which the package name may be selected) 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.

13.5 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 13-5 below, select the *Run* -> *Run...* menu option or use the Shift+F10 keyboard shortcut:

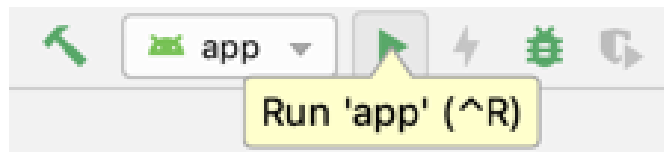


Figure 13-5

Select the physical Android device 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):

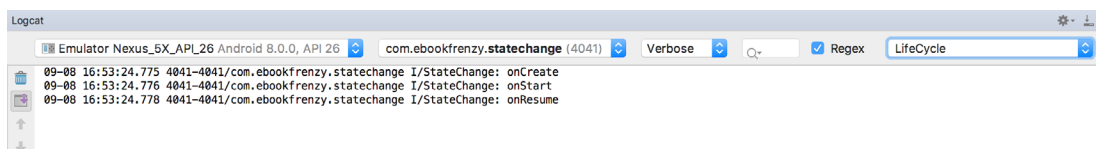


Figure 13-6

13.6 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. 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.

13.7 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.

14. 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.

14.1 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.

14.2 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 prior to performing the device rotation (on devices or emulators running Android 9 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 actual 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(Bundle outState) {
    super.onSaveInstanceState(outState);
}

@Override
protected void onRestoreInstanceState(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 in the event that the activity is restarted, edit the `content_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.

14.3 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.

14.4 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.

In order 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 *content_main.xml* resource file). In order 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 *content_main.xml* file is selected and the Layout Editor tool displayed, the EditText component has been assigned the ID *editText*:

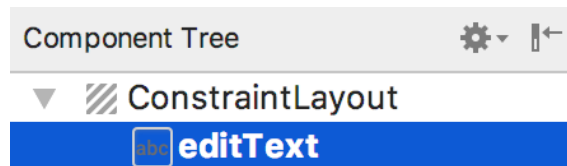


Figure 14-1

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, including the layout for our current activity. Within the *R* class is a subclass named *layout*, which contains the layout resources, and within that subclass is our *content_main* layout. With this knowledge, we can make a call to the *findViewById()* method of our activity object to get a reference to the editText object as follows:

```
final EditText editText = findViewById(R.id.editText);
```

Having either obtained a reference to the EditText object and assigned it to a variable, we can now obtain the text that it contains via the object's *getText()* method, which, in turn, returns the current text:

```
CharSequence userText = 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

Saving and Restoring the State of an Android Activity

reads as follows (noting also the additional import directive for *android.widget.EditText*):

```
package com.ebookfrenzy.statechange;

import android.os.Bundle;
import android.support.design.widget.FloatingActionButton;
import android.support.design.widget.Snackbar;
import android.support.v7.app.AppCompatActivity;
import android.support.v7.widget.Toolbar;
import android.view.View;
import android.view.Menu;
import android.view.MenuItem;
import android.util.Log;
import android.widget.EditText;

public class MainActivity extends AppCompatActivity {
    .
    .
    .

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

        final EditText editText =
            findViewById(R.id.editText);
        CharSequence userText = 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.

14.5 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(Bundle savedInstanceState) {
    super.onRestoreInstanceState(savedInstanceState);
    Log.i(TAG, "onRestoreInstanceState");

    final EditText editText =
```

```

        findViewById(R.id.editText);

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

        editText.setText(userText);
    }

```

14.6 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.

14.7 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.

15. Understanding Android Views, View Groups and Layouts

With the possible exception of listening to streaming audio, a user's interaction with an Android device is primarily visual and tactile in nature. All of this interaction takes place through the user interfaces of the applications installed on the device, including both the built-in applications and any third party applications installed by the user. It should come as no surprise, therefore, that a key element of developing Android applications involves the design and creation of user interfaces.

Within this chapter, the topic of Android user interface structure will be covered, together with an overview of the different elements that can be brought together to make up a user interface; namely Views, View Groups and Layouts.

15.1 Designing for Different Android Devices

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

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

15.2 Views and View Groups

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

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

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

15.3 Android Layout Managers

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

sole purpose of controlling how child views are positioned on the screen.

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

- **ConstraintLayout** – Introduced in Android 7, use of this layout manager is recommended for most layout requirements. ConstraintLayout allows the positioning and behavior of the views in a layout to be defined by simple constraint settings assigned to each child view. The flexibility of this layout allows complex layouts to be quickly and easily created without the necessity to nest other layout types inside each other, resulting in improved layout performance. ConstraintLayout is also tightly integrated into the Android Studio Layout Editor tool. Unless otherwise stated, this is the layout of choice for the majority of examples in this book.
- **LinearLayout** – Positions child views in a single row or column depending on the orientation selected. A *weight* value can be set on each child to specify how much of the layout space that child should occupy relative to other children.
- **TableLayout** – Arranges child views into a grid format of rows and columns. Each row within a table is represented by a *TableRow* object child, which, in turn, contains a view object for each cell.
- **FrameLayout** – The purpose of the FrameLayout is to allocate an area of screen, typically for the purposes of displaying a single view. If multiple child views are added they will, by default, appear on top of each other positioned in the top left-hand corner of the layout area. Alternate positioning of individual child views can be achieved by setting gravity values on each child. For example, setting a *center_vertical* gravity value on a child will cause it to be positioned in the vertical center of the containing FrameLayout view.
- **RelativeLayout** – The RelativeLayout allows child views to be positioned relative both to each other and the containing layout view through the specification of alignments and margins on child views. For example, child *View A* may be configured to be positioned in the vertical and horizontal center of the containing RelativeLayout view. *View B*, on the other hand, might also be configured to be centered horizontally within the layout view, but positioned 30 pixels above the top edge of *View A*, thereby making the vertical position *relative* to that of *View A*. The RelativeLayout manager can be of particular use when designing a user interface that must work on a variety of screen sizes and orientations.
- **AbsoluteLayout** – Allows child views to be positioned at specific X and Y coordinates within the containing layout view. Use of this layout is discouraged since it lacks the flexibility to respond to changes in screen size and orientation.
- **GridLayout** – A GridLayout instance is divided by invisible lines that form a grid containing rows and columns of cells. Child views are then placed in cells and may be configured to cover multiple cells both horizontally and vertically allowing a wide range of layout options to be quickly and easily implemented. Gaps between components in a GridLayout may be implemented by placing a special type of view called a *Space* view into adjacent cells, or by setting margin parameters.
- **CoordinatorLayout** – Introduced as part of the Android Design Support Library with Android 5.0, the CoordinatorLayout is designed specifically for coordinating the appearance and behavior of the app bar across the top of an application screen with other view elements. When creating a new activity using the Basic Activity template, the parent view in the main layout will be implemented using a CoordinatorLayout instance. This layout manager will be covered in greater detail starting with the chapter entitled “*Working with the Floating Action Button and Snackbar*”.

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

15.4 The View Hierarchy

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

A user interface screen is comprised of a view hierarchy with a *root view* positioned at the top of the tree and child views positioned on branches below. The child of a container view appears on top of its parent view and is constrained to appear within the bounds of the parent view's display area. Consider, for example, the user interface illustrated in Figure 15-1:

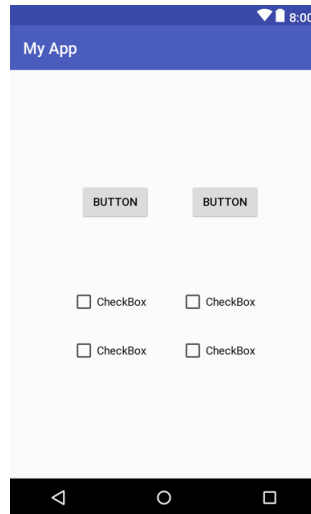


Figure 15-1

In addition to the visible button and checkbox views, the user interface actually includes a number of layout views that control how the visible views are positioned. Figure 15-2 shows an alternative view of the user interface, this time highlighting the presence of the layout views in relation to the child views:

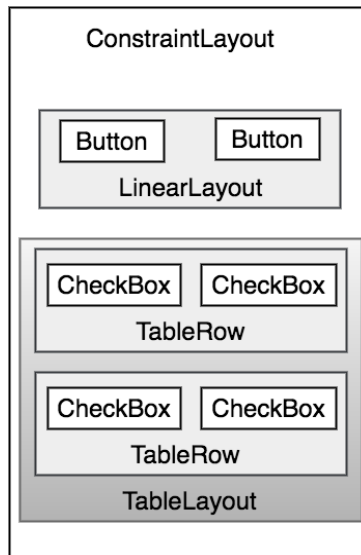


Figure 15-2

As was previously discussed, user interfaces are constructed in the form of a view hierarchy with a root view at the top. This being the case, we can also visualize the above user interface example in the form of the view tree illustrated in Figure 15-3:

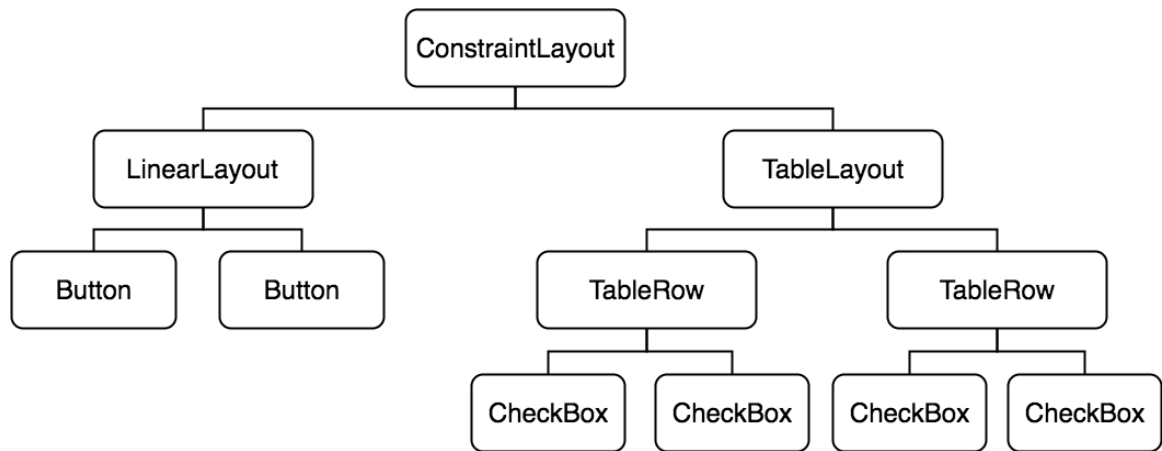


Figure 15-3

The view hierarchy diagram gives probably the clearest overview of the relationship between the various views that make up the user interface shown in Figure 15-1. When a user interface is displayed to the user, the Android runtime walks the view hierarchy, starting at the root view and working down the tree as it renders each view.

15.5 Creating User Interfaces

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

15.6 Summary

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

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

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

16. A Guide to the Android Studio Layout Editor Tool

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

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

16.1 Basic vs. Empty Activity Templates

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

When creating a new Android Studio project, a number of different templates are available to be used as the starting point for the user interface of the main activity. The most basic of these templates are the Basic Activity and Empty Activity templates. Although these seem similar at first glance, there are actually considerable differences between the two options.

The Empty Activity template creates a single layout file consisting of a ConstraintLayout manager instance containing a TextView object as shown in Figure 16-1:

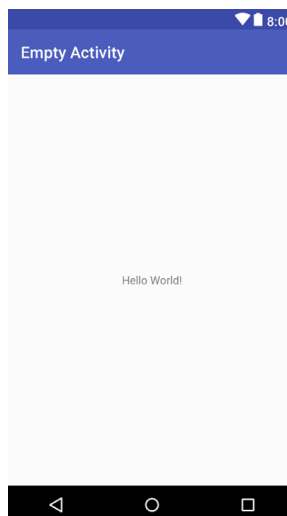


Figure 16-1

The Basic Activity, on the other hand, consists of two layout files. The top level layout file has a `CoordinatorLayout` as the root view, a configurable app bar, a menu preconfigured with a single menu item (A in Figure 16-2), a floating action button (B) and a reference to the second layout file in which the layout for the content area of the activity user interface is declared:

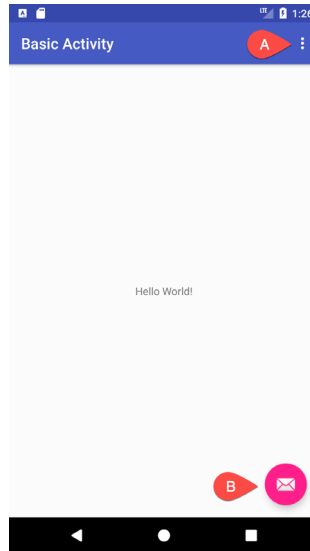


Figure 16-2

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

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

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

1. Double-click on the main *activity* layout file located in the Project tool window under *app -> res -> layout* to load it into the Layout Editor. This will be the layout file prefixed with *activity_* and not the content file prefixed with *content_*.
2. With the layout loaded into the Layout Editor tool, select the floating action button and tap the keyboard *Delete* key to remove the object from the layout.
3. Locate and edit the Java code for the activity (located under *app -> java -> <package name> -> <activity class name>*) and remove the floating action button code from the `onCreate` method as follows:

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

```

setContentView(R.layout.activity_main);
Toolbar toolbar = findViewById(R.id.toolbar);
setSupportActionBar(toolbar);

```

```


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


```

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

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

16.2 The Android Studio Layout Editor

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

Under the surface, the Layout Editor tool actually constructs an XML resource file containing the definition of the user interface that is being designed. As such, the Layout Editor tool operates in two distinct modes referred to as *Design mode* and *Text mode*.

16.3 Design Mode

In design mode, the user interface can be visually manipulated by directly working with the view palette and the graphical representation of the layout. Figure 16-3 highlights the key areas of the Android Studio Layout Editor tool in design mode:

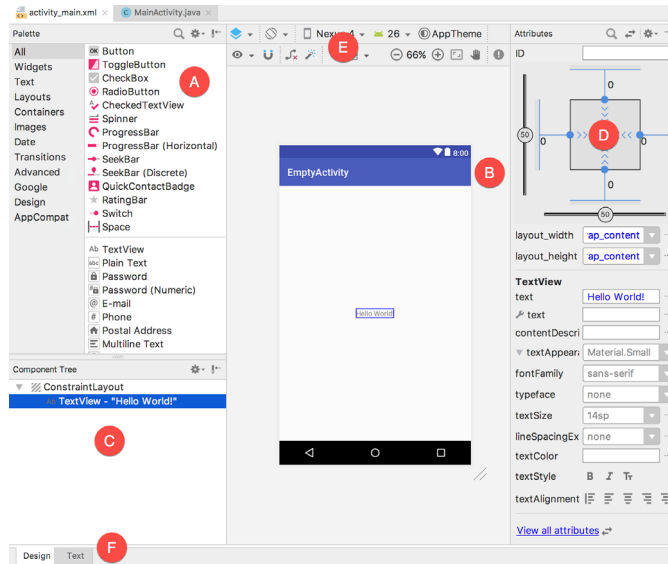


Figure 16-3

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

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

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

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

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

F – Mode Switching Tabs – The tabs located along the lower edge of the Layout Editor provide a way to switch back and forth between the Layout Editor tool’s text and design modes.

16.4 The Palette

The Layout Editor palette is organized into two panels designed to make it easy to locate and preview view components for addition to a layout design. The category panel (marked A in Figure 16-4) lists the different

categories of view components supported by the Android SDK. When a category is selected from the list, the second panel (B) updates to display a list of the components that fall into that category:

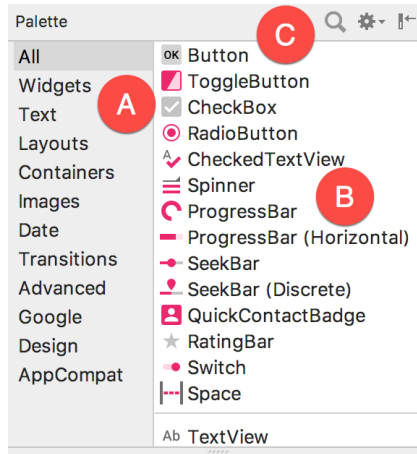


Figure 16-4

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

A search for a specific component within the currently selected category may be initiated by clicking on the search button (marked C in Figure 16-4 above) in the palette toolbar and typing in the component name. As characters are typed, matching results will appear in real-time within the component list panel. If you are unsure of the category in which the component resides, simply select the All category either before or during the search operation.

16.5 Design and Layout Views

When the Layout Editor tool is in Design mode, the layout can be viewed in two different ways. The view shown in Figure 16-3 above is the Design view and shows the layout and widgets as they will appear in the running app. A second mode, referred to as Layout or Blueprint view can be shown either instead of, or concurrently with the Design view. The toolbar menu shown in Figure 16-5 provides options to display the Design, Blueprint, or both views. A fourth option, *Force Refresh Layout*, causes the layout to rebuild and redraw. This can be useful when the layout enters an unexpected state or is not accurately reflecting the current design settings:

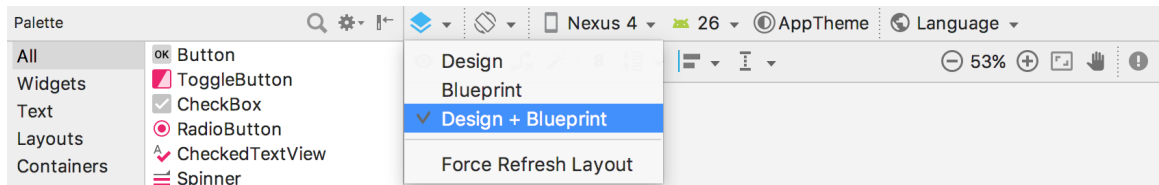


Figure 16-5

Whether to display the layout view, design view or both is a matter of personal preference. A good approach is to begin with both displayed as shown in Figure 16-6:

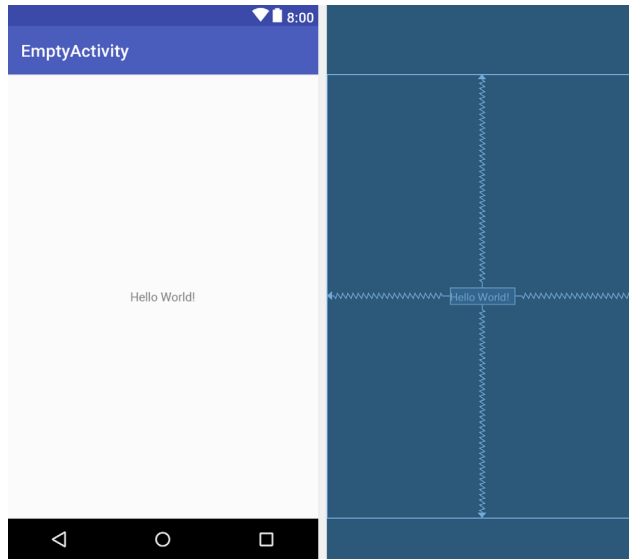


Figure 16-6

16.6 Text Mode

It is important to keep in mind when using the Android Studio Layout Editor tool that all it is really doing is providing a user friendly approach to creating XML layout resource files. At any time during the design process, the underlying XML can be viewed and directly edited simply by clicking on the *Text* tab located at the bottom of the Layout Editor tool panel. To return to design mode, simply click on the *Design* tab.

Figure 16-7 highlights the key areas of the Android Studio Layout Editor tool in text mode:

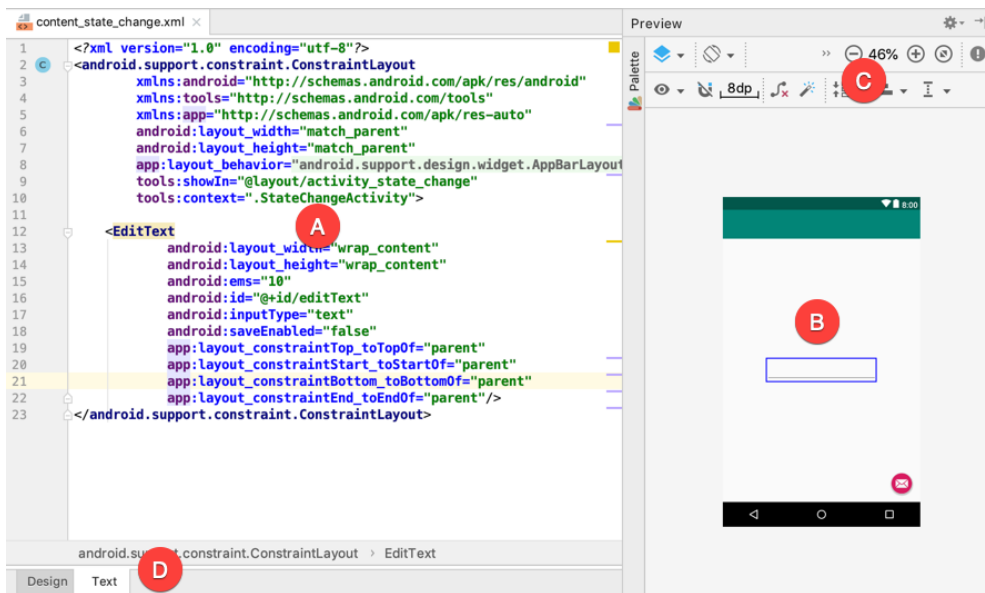


Figure 16-7

A – Editor – The editor panel displays the XML that makes up the current user interface layout design. This is

the full Android Studio editor environment containing all of the features previously outlined in the “*The Basics of the Android Studio Code Editor*” chapter of this book.

B – Preview – As changes are made to the XML in the editor, these changes are visually reflected in the preview window. This provides instant visual feedback on the XML changes as they are made in the editor, thereby avoiding the need to switch back and forth between text and design mode to see changes. The preview also allows direct manipulation and design of the layout just as if the layout were in Design mode, with visual changes being reflected in the editor panel in real-time. As with Design mode, both the Design and Layout views may be displayed using the toolbar buttons highlighted in Figure 16-5 above.

C – Toolbar – The toolbar in text mode provides access to the same functions available in design mode.

D - Mode Switching Tabs – The tabs located along the lower edge of the Layout Editor provide a way to switch back and forth between the Layout Editor tool’s Text and Design modes.

16.7 Setting Attributes

The Attributes panel provides access to all of the available settings for the currently selected component. Figure 16-8, for example, shows the attributes for the TextView widget:

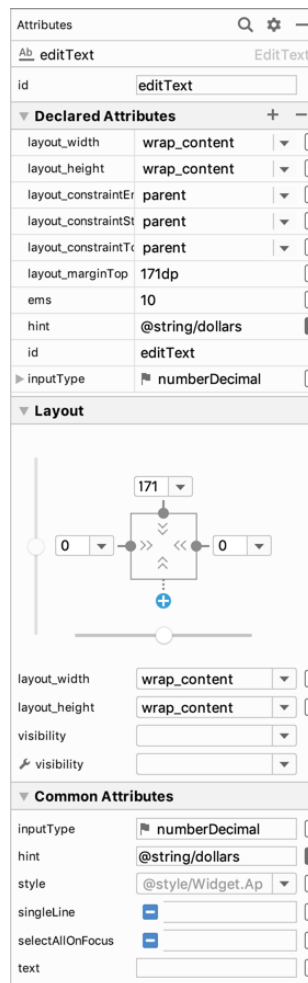


Figure 16-8

The Attributes tool window is divided into the following different sections.

- **id** - Contains the id property which defines the name by which the currently selected object will be referenced in the source code of the app.
- **Declared Attributes** - Contains all of the properties which have already been assigned a value.
- **Layout** - The settings that define how the currently selected view object is positioned and sized in relation to the screen and other objects in the layout.
- **Common Attributes** - A list of attributes that commonly need to be changed for the class of view object currently selected.
- **All Attributes** - A complete list of all of the attributes available for the currently selected object.

A search for a specific attribute may also be performed by selecting the search button in the toolbar of the attributes tool window and typing in the attribute name.

Some attributes contain a narrow button to the right of the value field. This indicates that the Resources dialog is available to assist in selecting a suitable property value. To display the dialog, simply click on the button. The appearance of this button changes to reflect whether or not the corresponding property value is stored in a resource file or hardcoded. If the value is stored in a resource file, the button to the right of the text property field will be filled in to indicate that the value is not hard coded as highlighted in Figure 16-9 below:



Figure 16-9

Attributes for which a finite number of valid options are available will present a drop down menu (Figure 16-10) from which a selection may be made.

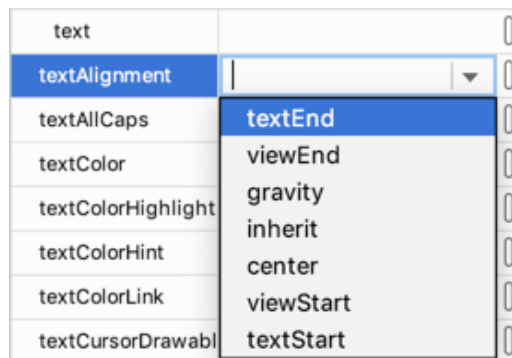


Figure 16-10

16.8 Converting Views

Changing a view in a layout from one type to another (such as converting a `TextView` to an `EditText`) can be performed easily within the Android Studio layout editor simply by right-clicking on the view either within the screen layout or Component tree window and selecting the *Convert view...* menu option (Figure 16-11):

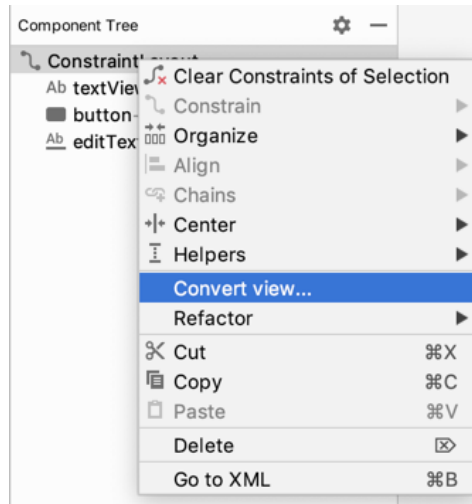


Figure 16-11

Once selected, a dialog will appear containing a list of compatible view types to which the selected object is eligible for conversion. Figure 16-12, for example shows the types to which an existing `TextView` view may be converted:

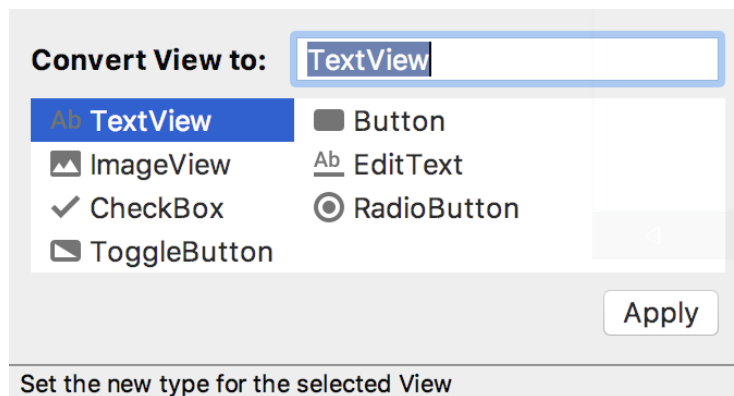


Figure 16-12

This technique is also useful for converting layouts from one type to another (for example converting a `ConstraintLayout` to a `LinearLayout`).

16.9 Displaying Sample Data

When designing layouts in Android Studio situations will arise where the content to be displayed within the user interface will not be available until the app is completed and running. This can sometimes make it difficult to assess from within the layout editor how the layout will appear at app runtime. To address this issue, the layout editor allows sample data to be specified that will populate views within the layout editor with sample images

and data. This sample data only appears within the layout editor and is not displayed when the app runs. Sample data may be configured either by directly editing the XML for the layout, or visually using the design-time helper by right-clicking on the widget in the design area and selecting the *Set Sample Data* menu option. The design-time helper panel will display a range of preconfigured options for sample data to be displayed on the selected view item including combinations of text and images in a variety of configurations. Figure 16-13, for example, shows the sample data options displayed when selecting sample data to appear in a RecyclerView list:

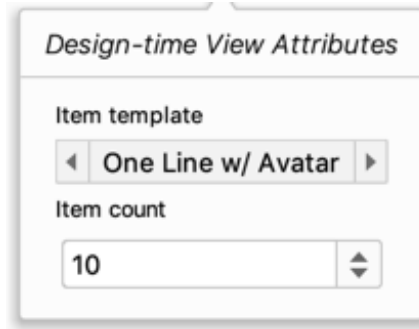


Figure 16-13

Alternatively, custom text and images may be provided for display during the layout design process. An example of using sample data within the layout editor is included in a later chapter entitled “*A Layout Editor Sample Data Tutorial*”.

16.10 Creating a Custom Device Definition

The device menu in the Layout Editor toolbar (Figure 16-14) provides a list of preconfigured device types which, when selected, will appear as the device screen canvas. In addition to the pre-configured device types, any AVD instances that have previously been configured within the Android Studio environment will also be listed within the menu. To add additional device configurations, display the device menu, select the *Add Device Definition...* option and follow the steps outlined in the chapter entitled “*Creating an Android Virtual Device (AVD) in Android Studio*”.

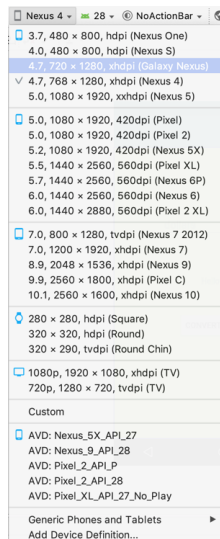


Figure 16-14

16.11 Changing the Current Device

As an alternative to the device selection menu, the current device format may be changed by selecting the *Custom* option from the device menu, clicking on the resize handle located next to the bottom right-hand corner of the device screen (Figure 16-15) and dragging to select an alternate device display format. As the screen resizes, markers will appear indicating the various size options and orientations available for selection:

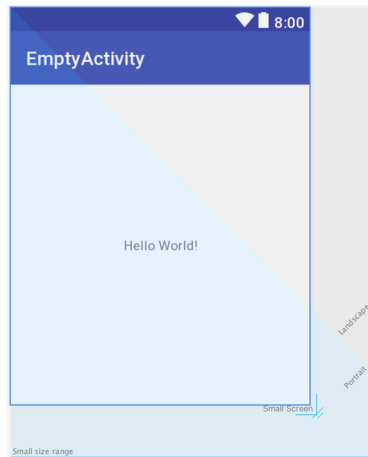


Figure 16-15

16.12 Summary

A key part of developing Android applications involves the creation of the user interface. Within the Android Studio environment, this is performed using the Layout Editor tool which operates in two modes. In design mode, view components are selected from a palette and positioned on a layout representing an Android device screen and configured using a list of attributes. In text mode, the underlying XML that represents the user interface layout can be directly edited, with changes reflected in a preview screen. These modes combine to provide an extensive and intuitive user interface design environment.

