

Android Studio 3.3

Development Essentials

Kotlin Edition

Android Studio 3.3 Development Essentials – Kotlin Edition

© 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
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 Google Play	37
5.5.13 Settings	37
5.5.14 Help.....	37
5.6 Working with Snapshots.....	37
5.7 Drag and Drop Support.....	38
5.8 Configuring Fingerprint Emulation	39
5.9 Summary	40
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. An Introduction to Kotlin	73
11.1 What is Kotlin?	73
11.2 Kotlin and Java.....	73
11.3 Converting from Java to Kotlin	73
11.4 Kotlin and Android Studio	74
11.5 Experimenting with Kotlin	74
11.6 Semi-colons in Kotlin	75
11.7 Summary	75
12. Kotlin Data Types, Variables and Nullability	77
12.1 Kotlin Data Types.....	77
12.1.1 Integer Data Types	78
12.1.2 Floating Point Data Types.....	78
12.1.3 Boolean Data Type.....	78
12.1.4 Character Data Type.....	78
12.1.5 String Data Type.....	78
12.1.6 Escape Sequences	79
12.2 Mutable Variables.....	80
12.3 Immutable Variables	80
12.4 Declaring Mutable and Immutable Variables.....	80
12.5 Data Types are Objects	80

Table of Contents

12.6 Type Annotations and Type Inference	81
12.7 Nullable Type.....	82
12.8 The Safe Call Operator	82
12.9 Not-Null Assertion.....	83
12.10 Nullable Types and the let Function.....	83
12.11 The Elvis Operator	84
12.12 Type Casting and Type Checking	85
12.13 Summary	85
13. Kotlin Operators and Expressions	87
13.1 Expression Syntax in Kotlin.....	87
13.2 The Basic Assignment Operator.....	87
13.3 Kotlin Arithmetic Operators	87
13.4 Augmented Assignment Operators	88
13.5 Increment and Decrement Operators	88
13.6 Equality Operators.....	89
13.7 Boolean Logical Operators	89
13.8 Range Operator	90
13.9 Bitwise Operators.....	90
13.9.1 Bitwise Inversion	91
13.9.2 Bitwise AND.....	91
13.9.3 Bitwise OR.....	91
13.9.4 Bitwise XOR.....	92
13.9.5 Bitwise Left Shift.....	92
13.9.6 Bitwise Right Shift.....	92
13.10 Summary	93
14. Kotlin Flow Control	95
14.1 Looping Flow Control	95
14.1.1 The Kotlin <i>for-in</i> Statement.....	95
14.1.2 The <i>while</i> Loop	96
14.1.3 The <i>do ... while</i> loop	97
14.1.4 Breaking from Loops	97
14.1.5 The <i>continue</i> Statement	98
14.1.6 Break and Continue Labels.....	98
14.2 Conditional Flow Control.....	99
14.2.1 Using the <i>if</i> Expressions	99
14.2.2 Using <i>if ... else ...</i> Expressions	100
14.2.3 Using <i>if ... else if ...</i> Expressions	100
14.2.4 Using the <i>when</i> Statement	100
14.3 Summary	101
15. An Overview of Kotlin Functions and Lambdas	103
15.1 What is a Function?	103
15.2 How to Declare a Kotlin Function	103
15.3 Calling a Kotlin Function.....	104
15.4 Single Expression Functions	104
15.5 Local Functions	104
15.6 Handling Return Values	105
15.7 Declaring Default Function Parameters.....	105
15.8 Variable Number of Function Parameters	105

15.9 Lambda Expressions	106
15.10 Higher-order Functions	107
15.11 Summary	108
16. The Basics of Object Oriented Programming in Kotlin	109
16.1 What is an Object?	109
16.2 What is a Class?	109
16.3 Declaring a Kotlin Class	109
16.4 Adding Properties to a Class	110
16.5 Defining Methods	110
16.6 Declaring and Initializing a Class Instance	110
16.7 Primary and Secondary Constructors	110
16.8 Initializer Blocks	113
16.9 Calling Methods and Accessing Properties	113
16.10 Custom Accessors	113
16.11 Nested and Inner Classes	114
16.12 Companion Objects	115
16.13 Summary	117
17. An Introduction to Kotlin Inheritance and Subclassing	119
17.1 Inheritance, Classes and Subclasses	119
17.2 Subclassing Syntax	119
17.3 A Kotlin Inheritance Example	120
17.4 Extending the Functionality of a Subclass	121
17.5 Overriding Inherited Methods	122
17.6 Adding a Custom Secondary Constructor	123
17.7 Using the SavingsAccount Class	123
17.8 Summary	123
18. Understanding Android Application and Activity Lifecycles	125
18.1 Android Applications and Resource Management	125
18.2 Android Process States	125
18.2.1 Foreground Process	126
18.2.2 Visible Process	126
18.2.3 Service Process	126
18.2.4 Background Process	126
18.2.5 Empty Process	127
18.3 Inter-Process Dependencies	127
18.4 The Activity Lifecycle	127
18.5 The Activity Stack	127
18.6 Activity States	128
18.7 Configuration Changes	128
18.8 Handling State Change	129
18.9 Summary	129
19. Handling Android Activity State Changes	131
19.1 New vs. Old Lifecycle Techniques	131
19.2 The Activity and Fragment Classes	131
19.3 Dynamic State vs. Persistent State	133
19.4 The Android Lifecycle Methods	134
19.5 Lifetimes	135

Table of Contents

19.6 Disabling Configuration Change Restarts	136
19.7 Lifecycle Method Limitations.....	136
19.8 Summary	137
20. Android Activity State Changes by Example	139
20.1 Creating the State Change Example Project	139
20.2 Designing the User Interface	140
20.3 Overriding the Activity Lifecycle Methods	141
20.4 Filtering the Logcat Panel.....	143
20.5 Running the Application.....	144
20.6 Experimenting with the Activity.....	145
20.7 Summary	146
21. Saving and Restoring the State of an Android Activity	147
21.1 Saving Dynamic State	147
21.2 Default Saving of User Interface State	147
21.3 The Bundle Class	148
21.4 Saving the State.....	149
21.5 Restoring the State	150
21.6 Testing the Application.....	150
21.7 Summary	151
22. Understanding Android Views, View Groups and Layouts	153
22.1 Designing for Different Android Devices	153
22.2 Views and View Groups	153
22.3 Android Layout Managers	153
22.4 The View Hierarchy	155
22.5 Creating User Interfaces	156
22.6 Summary	156
23. A Guide to the Android Studio Layout Editor	157
23.1 Basic vs. Empty Activity Templates	157
23.2 The Android Studio Layout Editor	159
23.3 Design Mode.....	159
23.4 The Palette.....	160
23.5 Design and Layout Views.....	161
23.6 Text Mode.....	161
23.7 Setting Attributes.....	162
23.8 Configuring Favorite Attributes	164
23.9 Converting Views.....	165
23.10 Displaying Sample Data	165
23.11 Creating a Custom Device Definition	166
23.12 Changing the Current Device.....	167
23.13 Summary.....	167
24. A Guide to the Android ConstraintLayout.....	169
24.1 How ConstraintLayout Works.....	169
24.1.1 Constraints.....	169
24.1.2 Margins.....	170
24.1.3 Opposing Constraints.....	170
24.1.4 Constraint Bias	171

24.1.5 Chains	172
24.1.6 Chain Styles.....	172
24.2 Baseline Alignment.....	173
24.3 Working with Guidelines	174
24.4 Configuring Widget Dimensions.....	174
24.5 Working with Barriers.....	174
24.6 Ratios	176
24.7 ConstraintLayout Advantages	176
24.8 ConstraintLayout Availability.....	176
24.9 Summary	177
25. A Guide to using ConstraintLayout in Android Studio	179
25.1 Design and Layout Views.....	179
25.2 Autoconnect Mode	180
25.3 Inference Mode.....	181
25.4 Manipulating Constraints Manually.....	181
25.5 Adding Constraints in the Inspector	183
25.6 Deleting Constraints.....	183
25.7 Adjusting Constraint Bias	184
25.8 Understanding ConstraintLayout Margins.....	184
25.9 The Importance of Opposing Constraints and Bias	186
25.10 Configuring Widget Dimensions.....	188
25.11 Adding Guidelines	189
25.12 Adding Barriers	191
25.13 Widget Group Alignment and Distribution.....	192
25.14 Converting other Layouts to ConstraintLayout.....	194
25.15 Summary	194
26. Working with ConstraintLayout Chains and Ratios in Android Studio	195
26.1 Creating a Chain.....	195
26.2 Changing the Chain Style	197
26.3 Spread Inside Chain Style.....	198
26.4 Packed Chain Style.....	198
26.5 Packed Chain Style with Bias.....	199
26.6 Weighted Chain.....	199
26.7 Working with Ratios.....	200
26.8 Summary	202
27. An Android Studio Layout Editor ConstraintLayout Tutorial	203
27.1 An Android Studio Layout Editor Tool Example	203
27.2 Creating a New Activity	203
27.3 Preparing the Layout Editor Environment.....	205
27.4 Adding the Widgets to the User Interface.....	205
27.5 Adding the Constraints	208
27.6 Testing the Layout	210
27.7 Using the Layout Inspector.....	211
27.8 Summary	211
28. Manual XML Layout Design in Android Studio	213
28.1 Manually Creating an XML Layout	213
28.2 Manual XML vs. Visual Layout Design.....	216

28.3 Summary	216
29. Managing Constraints using Constraint Sets.....	217
29.1 Kotlin Code vs. XML Layout Files.....	217
29.2 Creating Views.....	217
29.3 View Attributes.....	218
29.4 Constraint Sets.....	218
29.4.1 Establishing Connections.....	218
29.4.2 Applying Constraints to a Layout	218
29.4.3 Parent Constraint Connections.....	218
29.4.4 Sizing Constraints	219
29.4.5 Constraint Bias	219
29.4.6 Alignment Constraints.....	219
29.4.7 Copying and Applying Constraint Sets.....	219
29.4.8 ConstraintLayout Chains	219
29.4.9 Guidelines	220
29.4.10 Removing Constraints.....	220
29.4.11 Scaling.....	220
29.4.12 Rotation.....	221
29.5 Summary	221
30. An Android ConstraintSet Tutorial.....	223
30.1 Creating the Example Project in Android Studio	223
30.2 Adding Views to an Activity.....	223
30.3 Setting View Attributes.....	224
30.4 Creating View IDs.....	225
30.5 Configuring the Constraint Set	226
30.6 Adding the EditText View.....	227
30.7 Converting Density Independent Pixels (dp) to Pixels (px).....	228
30.8 Summary	229
31. A Guide to using Instant Run in Android Studio.....	231
31.1 Introducing Instant Run.....	231
31.2 Understanding Instant Run Swapping Levels.....	231
31.3 Enabling and Disabling Instant Run.....	232
31.4 Using Instant Run.....	232
31.5 An Instant Run Tutorial	233
31.6 Triggering an Instant Run Hot Swap	233
31.7 Triggering an Instant Run Warm Swap	233
31.8 Triggering an Instant Run Cold Swap	234
31.9 The Run Button	234
31.10 Summary.....	234
32. An Overview and Example of Android Event Handling.....	235
32.1 Understanding Android Events.....	235
32.2 Using the android:onClick Resource.....	235
32.3 Event Listeners and Callback Methods	236
32.4 An Event Handling Example	236
32.5 Designing the User Interface	237
32.6 The Event Listener and Callback Method.....	238
32.7 Consuming Events	239

32.8 Summary	240
33. Android Touch and Multi-touch Event Handling	241
33.1 Intercepting Touch Events	241
33.2 The MotionEvent Object	242
33.3 Understanding Touch Actions.....	242
33.4 Handling Multiple Touches	242
33.5 An Example Multi-Touch Application	243
33.6 Designing the Activity User Interface	243
33.7 Implementing the Touch Event Listener.....	243
33.8 Running the Example Application.....	246
33.9 Summary	246
34. Detecting Common Gestures using the Android Gesture Detector Class.....	247
34.1 Implementing Common Gesture Detection.....	247
34.2 Creating an Example Gesture Detection Project	248
34.3 Implementing the Listener Class.....	248
34.4 Creating the GestureDetectorCompat Instance.....	250
34.5 Implementing the onTouchEvent() Method.....	251
34.6 Testing the Application.....	251
34.7 Summary	251
35. Implementing Custom Gesture and Pinch Recognition on Android	253
35.1 The Android Gesture Builder Application.....	253
35.2 The GestureOverlayView Class	253
35.3 Detecting Gestures.....	253
35.4 Identifying Specific Gestures	253
35.5 Building and Running the Gesture Builder Application.....	254
35.6 Creating a Gestures File	254
35.7 Creating the Example Project.....	254
35.8 Extracting the Gestures File from the SD Card	255
35.9 Adding the Gestures File to the Project	255
35.10 Designing the User Interface	255
35.11 Loading the Gestures File	256
35.12 Registering the Event Listener.....	257
35.13 Implementing the onGesturePerformed Method.....	257
35.14 Testing the Application.....	258
35.15 Configuring the GestureOverlayView.....	258
35.16 Intercepting Gestures.....	259
35.17 Detecting Pinch Gestures.....	259
35.18 A Pinch Gesture Example Project.....	259
35.19 Summary	261
36. An Introduction to Android Fragments.....	263
36.1 What is a Fragment?	263
36.2 Creating a Fragment	263
36.3 Adding a Fragment to an Activity using the Layout XML File.....	264
36.4 Adding and Managing Fragments in Code	266
36.5 Handling Fragment Events	267
36.6 Implementing Fragment Communication.....	267
36.7 Summary	268

37. Using Fragments in Android Studio - An Example.....	271
37.1 About the Example Fragment Application	271
37.2 Creating the Example Project.....	271
37.3 Creating the First Fragment Layout.....	271
37.4 Creating the First Fragment Class	273
37.5 Creating the Second Fragment Layout.....	274
37.6 Adding the Fragments to the Activity	276
37.7 Making the Toolbar Fragment Talk to the Activity	277
37.8 Making the Activity Talk to the Text Fragment	280
37.9 Testing the Application.....	281
37.10 Summary.....	282
38. Modern Android App Architecture with Jetpack.....	283
38.1 What is Android Jetpack?	283
38.2 The “Old” Architecture.....	283
38.3 Modern Android Architecture.....	283
38.4 The ViewModel Component	284
38.5 The LiveData Component.....	284
38.6 LiveData and Data Binding.....	285
38.7 Android Lifecycles	286
38.8 Repository Modules.....	286
38.9 Summary	287
39. An Android Jetpack ViewModel Tutorial	289
39.1 About the Project	289
39.2 Creating the ViewModel Example Project.....	289
39.3 Reviewing the Project	290
39.3.1 The Main Activity.....	290
39.3.2 The Content Fragment	290
39.3.3 The ViewModel	291
39.4 Designing the Fragment Layout.....	292
39.5 Implementing the View Model.....	293
39.6 Associating the Fragment with the View Model.....	293
39.7 Modifying the Fragment	294
39.8 Accessing the ViewModel Data	294
39.9 Testing the Project.....	295
39.10 Summary.....	295
40. An Android Jetpack LiveData Tutorial	297
40.1 LiveData - A Recap	297
40.2 Adding LiveData to the ViewModel	297
40.3 Implementing the Observer.....	299
40.4 Summary	300
41. An Overview of Android Jetpack Data Binding	301
41.1 An Overview of Data Binding.....	301
41.2 The Key Components of Data Binding	301
41.2.1 The Project Build Configuration.....	301
41.2.2 The Data Binding Layout File.....	302
41.2.3 The Layout File Data Element	303
41.2.4 The Binding Classes.....	304

41.2.5 Data Binding Variable Configuration.....	304
41.2.6 Binding Expressions (One-Way).....	305
41.2.7 Binding Expressions (Two-Way).....	306
41.2.8 Event and Listener Bindings	306
41.3 Summary	307
42. An Android Jetpack Data Binding Tutorial.....	309
42.1 Removing the Redundant Code	309
42.2 Enabling Data Binding	310
42.3 Adding the Layout Element	311
42.4 Adding the Data Element to Layout File.....	312
42.5 Working with the Binding Class	313
42.6 Assigning the ViewModel Instance to the Data Binding Variable	314
42.7 Adding Binding Expressions	314
42.8 Adding the Conversion Method	315
42.9 Adding a Listener Binding.....	316
42.10 Testing the App.....	316
42.11 Summary	316
43. Working with Android Lifecycle-Aware Components	317
43.1 Lifecycle Awareness	317
43.2 Lifecycle Owners	317
43.3 Lifecycle Observers	318
43.4 Lifecycle States and Events.....	318
43.5 Summary	320
44. An Android Jetpack Lifecycle Awareness Tutorial	321
44.1 Creating the Example Lifecycle Project.....	321
44.2 Creating a Lifecycle Observer.....	321
44.3 Adding the Observer	323
44.4 Testing the Observer.....	323
44.5 Creating a Lifecycle Owner.....	324
44.6 Testing the Custom Lifecycle Owner.....	326
44.7 Summary	326
45. An Overview of the Navigation Architecture Component.....	327
45.1 Understanding Navigation.....	327
45.2 Declaring a Navigation Host.....	329
45.3 The Navigation Graph	330
45.4 Accessing the Navigation Controller.....	331
45.5 Triggering a Navigation Action	332
45.6 Passing Arguments.....	332
45.7 Summary	333
46. An Android Jetpack Navigation Component Tutorial	335
46.1 Creating the NavigationDemo Project.....	335
46.2 Adding Navigation to the Build Configuration.....	335
46.3 Creating the Navigation Graph Resource File.....	336
46.4 Declaring a Navigation Host.....	337
46.5 Adding Navigation Destinations.....	339
46.6 Designing the Destination Fragment Layouts.....	340

46.7 Adding an Action to the Navigation Graph.....	342
46.8 Implement the OnFragmentManagerInteractionListener	343
46.9 Triggering the Action	344
46.10 Passing Data Using Safeargs	344
46.11 Summary.....	347
47. Creating and Managing Overflow Menus on Android.....	349
47.1 The Overflow Menu	349
47.2 Creating an Overflow Menu	349
47.3 Displaying an Overflow Menu.....	350
47.4 Responding to Menu Item Selections.....	351
47.5 Creating Checkable Item Groups.....	351
47.6 Menus and the Android Studio Menu Editor.....	352
47.7 Creating the Example Project.....	353
47.8 Designing the Menu.....	353
47.9 Modifying the onOptionsItemSelected() Method.....	356
47.10 Testing the Application.....	357
47.11 Summary.....	357
48. Animating User Interfaces with the Android Transitions Framework.....	359
48.1 Introducing Android Transitions and Scenes	359
48.2 Using Interpolators with Transitions.....	360
48.3 Working with Scene Transitions	360
48.4 Custom Transitions and TransitionSets in Code	362
48.5 Custom Transitions and TransitionSets in XML.....	362
48.6 Working with Interpolators	364
48.7 Creating a Custom Interpolator	365
48.8 Using the beginDelayedTransition Method.....	366
48.9 Summary	366
49. An Android Transition Tutorial using beginDelayedTransition	367
49.1 Creating the Android Studio TransitionDemo Project.....	367
49.2 Preparing the Project Files.....	367
49.3 Implementing beginDelayedTransition Animation	367
49.4 Customizing the Transition	370
49.5 Summary	371
50. Implementing Android Scene Transitions – A Tutorial.....	373
50.1 An Overview of the Scene Transition Project	373
50.2 Creating the Android Studio SceneTransitions Project	373
50.3 Identifying and Preparing the Root Container	373
50.4 Designing the First Scene.....	373
50.5 Designing the Second Scene	374
50.6 Entering the First Scene	375
50.7 Loading Scene 2.....	376
50.8 Implementing the Transitions	377
50.9 Adding the Transition File	377
50.10 Loading and Using the Transition Set.....	377
50.11 Configuring Additional Transitions	378
50.12 Summary.....	379
51. Working with the Floating Action Button and Snackbar	381

51.1 The Material Design.....	381
51.2 The Design Library	381
51.3 The Floating Action Button (FAB)	381
51.4 The Snackbar.....	382
51.5 Creating the Example Project.....	383
51.6 Reviewing the Project.....	383
51.7 Changing the Floating Action Button	384
51.8 Adding the ListView to the Content Layout.....	385
51.9 Adding Items to the ListView	386
51.10 Adding an Action to the Snackbar.....	388
51.11 Summary	389
52. Creating a Tabbed Interface using the TabLayout Component	391
52.1 An Introduction to the ViewPager.....	391
52.2 An Overview of the TabLayout Component	391
52.3 Creating the TabLayoutDemo Project.....	392
52.4 Creating the First Fragment.....	392
52.5 Duplicating the Fragments.....	393
52.6 Adding the TabLayout and ViewPager.....	394
52.7 Creating the Pager Adapter.....	395
52.8 Performing the Initialization Tasks.....	396
52.9 Testing the Application.....	398
52.10 Customizing the TabLayout.....	399
52.11 Displaying Icon Tab Items.....	400
52.12 Summary	401
53. Working with the RecyclerView and CardView Widgets.....	403
53.1 An Overview of the RecyclerView	403
53.2 An Overview of the CardView	405
53.3 Adding the Libraries to the Project.....	406
53.4 Summary	406
54. An Android RecyclerView and CardView Tutorial.....	407
54.1 Creating the CardDemo Project.....	407
54.2 Removing the Floating Action Button	407
54.3 Adding the RecyclerView and CardView Libraries	407
54.4 Designing the CardView Layout	408
54.5 Adding the RecyclerView.....	409
54.6 Creating the RecyclerView Adapter.....	409
54.7 Adding the Image Files.....	411
54.8 Initializing the RecyclerView Component.....	412
54.9 Testing the Application.....	413
54.10 Responding to Card Selections	413
54.11 Summary	415
55. A Layout Editor Sample Data Tutorial	417
55.1 Adding Sample Data to a Project	417
55.2 Using Custom Sample Data	421
55.3 Summary	424
56. Working with the AppBar and Collapsing Toolbar Layouts	425

Table of Contents

56.1 The Anatomy of an AppBar	425
56.2 The Example Project	426
56.3 Coordinating the RecyclerView and Toolbar	426
56.4 Introducing the Collapsing Toolbar Layout	428
56.5 Changing the Title and Scrim Color	431
56.6 Summary	432
57. Implementing an Android Navigation Drawer	433
57.1 An Overview of the Navigation Drawer	433
57.2 Opening and Closing the Drawer	434
57.3 Responding to Drawer Item Selections	435
57.4 Using the Navigation Drawer Activity Template	436
57.5 Creating the Navigation Drawer Template Project.....	436
57.6 The Template Layout Resource Files.....	436
57.7 The Header Coloring Resource File.....	437
57.8 The Template Menu Resource File.....	437
57.9 The Template Code	437
57.10 Running the App	438
57.11 Summary.....	438
58. An Android Studio Master/Detail Flow Tutorial	439
58.1 The Master/Detail Flow	439
58.2 Creating a Master/Detail Flow Activity.....	440
58.3 The Anatomy of the Master/Detail Flow Template.....	441
58.4 Modifying the Master/Detail Flow Template	442
58.5 Changing the Content Model.....	442
58.6 Changing the Detail Pane	444
58.7 Modifying the WebsiteDetailFragment Class.....	445
58.8 Modifying the WebsiteListActivity Class	446
58.9 Adding Manifest Permissions.....	446
58.10 Running the Application.....	447
58.11 Summary.....	447
59. An Overview of Android Intents	449
59.1 An Overview of Intents	449
59.2 Explicit Intents.....	449
59.3 Returning Data from an Activity	450
59.4 Implicit Intents	451
59.5 Using Intent Filters.....	452
59.6 Checking Intent Availability	453
59.7 Summary	453
60. Android Explicit Intents – A Worked Example.....	455
60.1 Creating the Explicit Intent Example Application.....	455
60.2 Designing the User Interface Layout for MainActivity.....	455
60.3 Creating the Second Activity Class.....	456
60.4 Designing the User Interface Layout for ActivityB.....	457
60.5 Reviewing the Application Manifest File	458
60.6 Creating the Intent	459
60.7 Extracting Intent Data	460
60.8 Launching ActivityB as a Sub-Activity.....	460

60.9 Returning Data from a Sub-Activity.....	461
60.10 Testing the Application.....	462
60.11 Summary	462
61. Android Implicit Intents – A Worked Example	463
61.1 Creating the Android Studio Implicit Intent Example Project	463
61.2 Designing the User Interface	463
61.3 Creating the Implicit Intent	464
61.4 Adding a Second Matching Activity.....	464
61.5 Adding the Web View to the UI.....	465
61.6 Obtaining the Intent URL	465
61.7 Modifying the MyWebView Project Manifest File	466
61.8 Installing the MyWebView Package on a Device.....	468
61.9 Testing the Application.....	468
61.10 Summary	469
62. Android Broadcast Intents and Broadcast Receivers	471
62.1 An Overview of Broadcast Intents.....	471
62.2 An Overview of Broadcast Receivers	472
62.3 Obtaining Results from a Broadcast.....	473
62.4 Sticky Broadcast Intents	473
62.5 The Broadcast Intent Example.....	474
62.6 Creating the Example Application.....	474
62.7 Creating and Sending the Broadcast Intent.....	474
62.8 Creating the Broadcast Receiver	475
62.9 Registering the Broadcast Receiver.....	476
62.10 Testing the Broadcast Example	477
62.11 Listening for System Broadcasts.....	477
62.12 Summary	477
63. A Basic Overview of Threads and AsyncTasks.....	479
63.1 An Overview of Threads	479
63.2 The Application Main Thread.....	479
63.3 Thread Handlers	479
63.4 A Basic AsyncTask Example	479
63.5 Subclassing AsyncTask	481
63.6 Testing the App.....	484
63.7 Canceling a Task.....	484
63.8 Summary	484
64. An Overview of Android Started and Bound Services.....	485
64.1 Started Services.....	485
64.2 Intent Service	485
64.3 Bound Service.....	486
64.4 The Anatomy of a Service	486
64.5 Controlling Destroyed Service Restart Options.....	487
64.6 Declaring a Service in the Manifest File.....	487
64.7 Starting a Service Running on System Startup.....	488
64.8 Summary	488
65. Implementing an Android Started Service – A Worked Example	489

Table of Contents

65.1 Creating the Example Project.....	489
65.2 Creating the Service Class.....	489
65.3 Adding the Service to the Manifest File	490
65.4 Starting the Service	491
65.5 Testing the IntentService Example.....	491
65.6 Using the Service Class.....	492
65.7 Creating the New Service.....	492
65.8 Modifying the User Interface.....	493
65.9 Running the Application.....	494
65.10 Creating an AsyncTask for Service Tasks.....	495
65.11 Summary.....	496
66. Android Local Bound Services – A Worked Example	497
66.1 Understanding Bound Services.....	497
66.2 Bound Service Interaction Options.....	497
66.3 An Android Studio Local Bound Service Example	497
66.4 Adding a Bound Service to the Project	498
66.5 Implementing the Binder	498
66.6 Binding the Client to the Service	500
66.7 Completing the Example.....	502
66.8 Testing the Application.....	503
66.9 Summary	503
67. Android Remote Bound Services – A Worked Example	505
67.1 Client to Remote Service Communication.....	505
67.2 Creating the Example Application.....	505
67.3 Designing the User Interface	505
67.4 Implementing the Remote Bound Service.....	506
67.5 Configuring a Remote Service in the Manifest File.....	507
67.6 Launching and Binding to the Remote Service.....	507
67.7 Sending a Message to the Remote Service	509
67.8 Summary	509
68. An Android Notifications Tutorial	511
68.1 An Overview of Notifications.....	511
68.2 Creating the NotifyDemo Project.....	513
68.3 Designing the User Interface	513
68.4 Creating the Second Activity	513
68.5 Creating a Notification Channel	514
68.6 Creating and Issuing a Basic Notification	516
68.7 Launching an Activity from a Notification.....	518
68.8 Adding Actions to a Notification	520
68.9 Bundled Notifications.....	520
68.10 Summary.....	523
69. An Android Direct Reply Notification Tutorial	525
69.1 Creating the DirectReply Project	525
69.2 Designing the User Interface	525
69.3 Creating the Notification Channel.....	526
69.4 Building the RemoteInput Object.....	527
69.5 Creating the PendingIntent.....	528

69.6 Creating the Reply Action.....	528
69.7 Receiving Direct Reply Input.....	531
69.8 Updating the Notification	532
69.9 Summary	533
70. An Introduction to Android Multi-Window Support.....	535
70.1 Split-Screen, Freeform and Picture-in-Picture Modes.....	535
70.2 Entering Multi-Window Mode	536
70.3 Enabling Freeform Support	537
70.4 Checking for Freeform Support	537
70.5 Enabling Multi-Window Support in an App	538
70.6 Specifying Multi-Window Attributes	538
70.7 Detecting Multi-Window Mode in an Activity.....	539
70.8 Receiving Multi-Window Notifications	539
70.9 Launching an Activity in Multi-Window Mode	540
70.10 Configuring Freeform Activity Size and Position.....	540
70.11 Summary	541
71. An Android Studio Multi-Window Split-Screen and Freeform Tutorial.....	543
71.1 Creating the Multi-Window Project.....	543
71.2 Designing the MainActivity User Interface	543
71.3 Adding the Second Activity	544
71.4 Launching the Second Activity	544
71.5 Enabling Multi-Window Mode	545
71.6 Testing Multi-Window Support	545
71.7 Launching the Second Activity in a Different Window.....	547
71.8 Summary	548
72. An Overview of Android SQLite Databases	549
72.1 Understanding Database Tables	549
72.2 Introducing Database Schema	549
72.3 Columns and Data Types	549
72.4 Database Rows	550
72.5 Introducing Primary Keys	550
72.6 What is SQLite?	550
72.7 Structured Query Language (SQL).....	550
72.8 Trying SQLite on an Android Virtual Device (AVD)	551
72.9 The Android Room Persistence Library.....	553
72.10 Summary	553
73. The Android Room Persistence Library	555
73.1 Revisiting Modern App Architecture	555
73.2 Key Elements of Room Database Persistence.....	555
73.2.1 Repository	556
73.2.2 Room Database	556
73.2.3 Data Access Object (DAO)	556
73.2.4 Entities.....	556
73.2.5 SQLite Database	556
73.3 Understanding Entities.....	557
73.4 Data Access Objects.....	559
73.5 The Room Database	560

Table of Contents

73.6 The Repository.....	561
73.7 In-Memory Databases	562
73.8 Summary	562
74. An Android TableLayout and TableRow Tutorial	563
74.1 The TableLayout and TableRow Layout Views.....	563
74.2 Creating the Room Database Project	564
74.3 Converting to a LinearLayout.....	564
74.4 Adding the TableLayout to the User Interface.....	565
74.5 Configuring the TableRows	566
74.6 Adding the Button Bar to the Layout	567
74.7 Adding the RecyclerView.....	568
74.8 Adjusting the Layout Margins	569
74.9 Summary	569
75. An Android Room Database and Repository Tutorial.....	571
75.1 About the RoomDemo Project.....	571
75.2 Modifying the Build Configuration.....	571
75.3 Building the Entity	571
75.4 Creating the Data Access Object.....	573
75.5 Adding the Room Database.....	574
75.6 Adding the Repository	575
75.7 Modifying the ViewModel.....	578
75.8 Creating the Product Item Layout	579
75.9 Adding the RecyclerView Adapter.....	580
75.10 Preparing the Main Fragment	581
75.11 Adding the Button Listeners.....	582
75.12 Adding LiveData Observers	583
75.13 Initializing the RecyclerView.....	583
75.14 Testing the RoomDemo App.....	583
75.15 Summary	584
76. Accessing Cloud Storage using the Android Storage Access Framework.....	585
76.1 The Storage Access Framework.....	585
76.2 Working with the Storage Access Framework.....	586
76.3 Filtering Picker File Listings	586
76.4 Handling Intent Results.....	587
76.5 Reading the Content of a File	588
76.6 Writing Content to a File	588
76.7 Deleting a File.....	589
76.8 Gaining Persistent Access to a File.....	589
76.9 Summary	590
77. An Android Storage Access Framework Example	591
77.1 About the Storage Access Framework Example.....	591
77.2 Creating the Storage Access Framework Example.....	591
77.3 Designing the User Interface	591
77.4 Declaring Request Codes	592
77.5 Creating a New Storage File.....	593
77.6 The onActivityResult() Method	594
77.7 Saving to a Storage File.....	595

77.8 Opening and Reading a Storage File	598
77.9 Testing the Storage Access Application	600
77.10 Summary	600
78. Implementing Video Playback on Android using the VideoView and MediaController Classes	601
78.1 Introducing the Android VideoView Class	601
78.2 Introducing the Android MediaController Class	602
78.3 Creating the Video Playback Example	602
78.4 Designing the VideoPlayer Layout	602
78.5 Configuring the VideoView	603
78.6 Adding Internet Permission	604
78.7 Adding the MediaController to the Video View	605
78.8 Setting up the onPreparedListener	606
78.9 Summary	607
79. Android Picture-in-Picture Mode.....	609
79.1 Picture-in-Picture Features.....	609
79.2 Enabling Picture-in-Picture Mode.....	610
79.3 Configuring Picture-in-Picture Parameters	610
79.4 Entering Picture-in-Picture Mode	611
79.5 Detecting Picture-in-Picture Mode Changes	611
79.6 Adding Picture-in-Picture Actions.....	611
79.7 Summary	612
80. An Android Picture-in-Picture Tutorial.....	613
80.1 Adding Picture-in-Picture Support to the Manifest	613
80.2 Adding a Picture-in-Picture Button	613
80.3 Entering Picture-in-Picture Mode	614
80.4 Detecting Picture-in-Picture Mode Changes	615
80.5 Adding a Broadcast Receiver	616
80.6 Adding the PiP Action.....	617
80.7 Testing the Picture-in-Picture Action	619
80.8 Summary	620
81. Video Recording and Image Capture on Android using Camera Intents	621
81.1 Checking for Camera Support.....	621
81.2 Calling the Video Capture Intent.....	621
81.3 Calling the Image Capture Intent.....	622
81.4 Creating an Android Studio Video Recording Project.....	623
81.5 Designing the User Interface Layout	623
81.6 Checking for the Camera	624
81.7 Launching the Video Capture Intent.....	624
81.8 Handling the Intent Return	625
81.9 Testing the Application.....	626
81.10 Summary	626
82. Making Runtime Permission Requests in Android.....	627
82.1 Understanding Normal and Dangerous Permissions.....	627
82.2 Creating the Permissions Example Project.....	629
82.3 Checking for a Permission	629
82.4 Requesting Permission at Runtime.....	631

82.5 Providing a Rationale for the Permission Request	632
82.6 Testing the Permissions App.....	634
82.7 Summary	634
83. Android Audio Recording and Playback using MediaPlayer and MediaRecorder	635
83.1 Playing Audio	635
83.2 Recording Audio and Video using the MediaRecorder Class	636
83.3 About the Example Project	637
83.4 Creating the AudioApp Project	637
83.5 Designing the User Interface	637
83.6 Checking for Microphone Availability	638
83.7 Performing the Activity Initialization	639
83.8 Implementing the recordAudio() Method.....	640
83.9 Implementing the stopAudio() Method.....	640
83.10 Implementing the playAudio() method	641
83.11 Configuring and Requesting Permissions	641
83.12 Testing the Application.....	644
83.13 Summary	644
84. Working with the Google Maps Android API in Android Studio	645
84.1 The Elements of the Google Maps Android API	645
84.2 Creating the Google Maps Project.....	646
84.3 Obtaining Your Developer Signature	646
84.4 Adding the Apache HTTP Legacy Library Requirement	647
84.5 Testing the Application.....	647
84.6 Understanding Geocoding and Reverse Geocoding	648
84.7 Adding a Map to an Application.....	649
84.8 Requesting Current Location Permission	650
84.9 Displaying the User's Current Location	651
84.10 Changing the Map Type	652
84.11 Displaying Map Controls to the User.....	653
84.12 Handling Map Gesture Interaction.....	653
84.12.1 Map Zooming Gestures.....	653
84.12.2 Map Scrolling/Panning Gestures	654
84.12.3 Map Tilt Gestures.....	654
84.12.4 Map Rotation Gestures.....	654
84.13 Creating Map Markers.....	654
84.14 Controlling the Map Camera	655
84.15 Summary	657
85. Printing with the Android Printing Framework	659
85.1 The Android Printing Architecture	659
85.2 The Print Service Plugins	659
85.3 Google Cloud Print.....	660
85.4 Printing to Google Drive.....	660
85.5 Save as PDF	661
85.6 Printing from Android Devices	661
85.7 Options for Building Print Support into Android Apps.....	662
85.7.1 Image Printing	662
85.7.2 Creating and Printing HTML Content	663
85.7.3 Printing a Web Page.....	664

85.7.4 Printing a Custom Document	665
85.8 Summary	665
86. An Android HTML and Web Content Printing Example	667
86.1 Creating the HTML Printing Example Application	667
86.2 Printing Dynamic HTML Content	667
86.3 Creating the Web Page Printing Example	670
86.4 Removing the Floating Action Button	670
86.5 Designing the User Interface Layout	670
86.6 Loading the Web Page into the WebView	671
86.7 Adding the Print Menu Option	672
86.8 Summary	674
87. A Guide to Android Custom Document Printing	675
87.1 An Overview of Android Custom Document Printing	675
87.1.1 Custom Print Adapters	675
87.2 Preparing the Custom Document Printing Project	676
87.3 Creating the Custom Print Adapter	677
87.4 Implementing the onLayout() Callback Method	678
87.5 Implementing the onWrite() Callback Method	681
87.6 Checking a Page is in Range	683
87.7 Drawing the Content on the Page Canvas	684
87.8 Starting the Print Job	686
87.9 Testing the Application	687
87.10 Summary	687
88. An Introduction to Android App Links	689
88.1 An Overview of Android App Links	689
88.2 App Link Intent Filters	689
88.3 Handling App Link Intents	690
88.4 Associating the App with a Website	690
88.5 Summary	691
89. An Android Studio App Links Tutorial	693
89.1 About the Example App	693
89.2 The Database Schema	693
89.3 Loading and Running the Project	694
89.4 Adding the URL Mapping	695
89.5 Adding the Intent Filter	698
89.6 Adding Intent Handling Code	698
89.7 Testing the App Link	701
89.8 Associating an App Link with a Web Site	702
89.9 Summary	703
90. An Introduction to Android Instant Apps	705
90.1 An Overview of Android Instant Apps	705
90.2 Instant App Feature Modules	705
90.3 Instant App Project Structure	706
90.4 The Application and Feature Build Plugins	706
90.5 Installing the Instant Apps Development SDK	708
90.6 Summary	708

91. An Android Instant App Tutorial	709
91.1 Creating the Instant App Project.....	709
91.2 Reviewing the Project	710
91.3 Testing the Installable App.....	711
91.4 Testing the Instant App	712
91.5 Reviewing the Instant App APK Files	713
91.6 Summary	714
92. Adapting an Android Studio Project for Instant Apps.....	715
92.1 Getting Started.....	715
92.2 Adding the Application APK Module	716
92.3 Adding an Instant App Module.....	718
92.4 Testing the Instant App	719
92.5 Summary	719
93. A Guide to the Android Studio Profiler.....	721
93.1 Accessing the Android Profiler	721
93.2 Enabling Advanced Profiling.....	721
93.3 The Android Profiler Tool Window.....	722
93.4 The Sessions Panel.....	723
93.5 The CPU Profiler	724
93.6 Memory Profiler	727
93.7 Network Profiler	728
93.8 Energy Profiler.....	730
93.9 Summary	730
94. An Android Biometric Authentication Tutorial.....	731
94.1 An Overview of Biometric Authentication.....	731
94.2 Creating the Biometric Authentication Project	731
94.3 Configuring Device Fingerprint Authentication	732
94.4 Adding the Biometric Permission to the Manifest File.....	732
94.5 Designing the User Interface	733
94.6 Adding a Toast Convenience Method	733
94.7 Checking the Security Settings.....	734
94.8 Configuring the Authentication Callbacks.....	735
94.9 Adding the CancellationSignal.....	736
94.10 Starting the Biometric Prompt	736
94.11 Testing the Project.....	737
94.12 Summary	738
95. Creating, Testing and Uploading an Android App Bundle.....	739
95.1 The Release Preparation Process.....	739
95.2 Android App Bundles.....	739
95.3 Register for a Google Play Developer Console Account.....	740
95.4 Configuring the App in the Console	740
95.5 Enabling Google Play App Signing.....	741
95.6 Enabling ProGuard	742
95.7 Creating a Keystore File	742
95.8 Creating the Android App Bundle.....	744
95.9 Generating Test APK Files	745
95.10 Uploading the App Bundle to the Google Play Developer Console	746

95.11 Exploring the App Bundle	747
95.12 Managing Testers	749
95.13 Uploading Instant App Bundles.....	750
95.14 Uploading New App Bundle Revisions.....	751
95.15 Analyzing the App Bundle File	752
95.16 Enabling Google Play Signing for an Existing App	753
95.17 Summary	754
96. An Overview of Gradle in Android Studio.....	755
96.1 An Overview of Gradle	755
96.2 Gradle and Android Studio	755
96.2.1 Sensible Defaults	755
96.2.2 Dependencies.....	755
96.2.3 Build Variants	756
96.2.4 Manifest Entries	756
96.2.5 APK Signing.....	756
96.2.6 ProGuard Support.....	756
96.3 The Top-level Gradle Build File.....	756
96.4 Module Level Gradle Build Files.....	758
96.5 Configuring Signing Settings in the Build File.....	760
96.6 Running Gradle Tasks from the Command-line	761
96.7 Summary	761
Index.....	763

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 3.3 and Android 9, the goal of this book is to teach the skills necessary to develop Android based applications using the Kotlin programming language.

Beginning with the basics, this book provides an outline of the steps necessary to set up an Android development and testing environment followed by an introduction to programming in Kotlin including data types, flow control, functions, lambdas and object-oriented programming.

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 3.3 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, Instant Apps, the Android Studio Profiler and Gradle build configuration.

Assuming you already have some 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/as33kotlin/index.php>

Introduction

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

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

1.2 Feedback

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

1.3 Errata

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

<https://www.ebookfrenzy.com/errata/as33kotlin.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.

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), the Kotlin plug-in 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 3.3 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 3.3 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*, *Android SDK* 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:

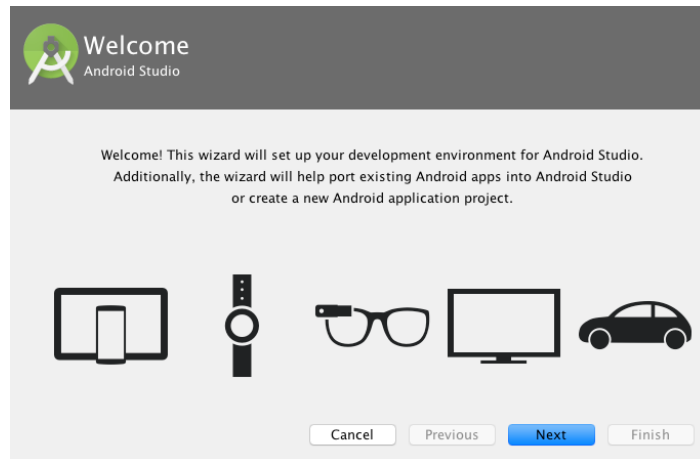


Figure 2-2

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:

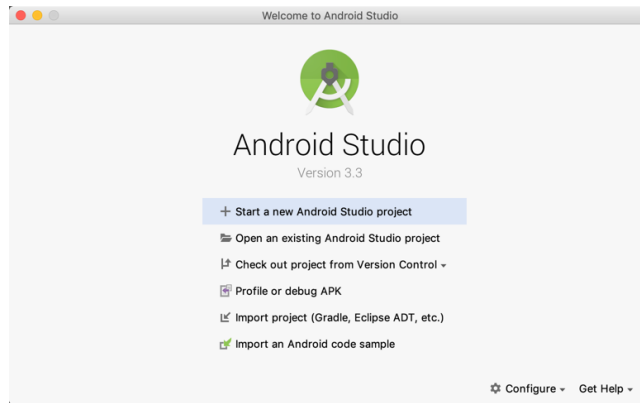


Figure 2-3

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:

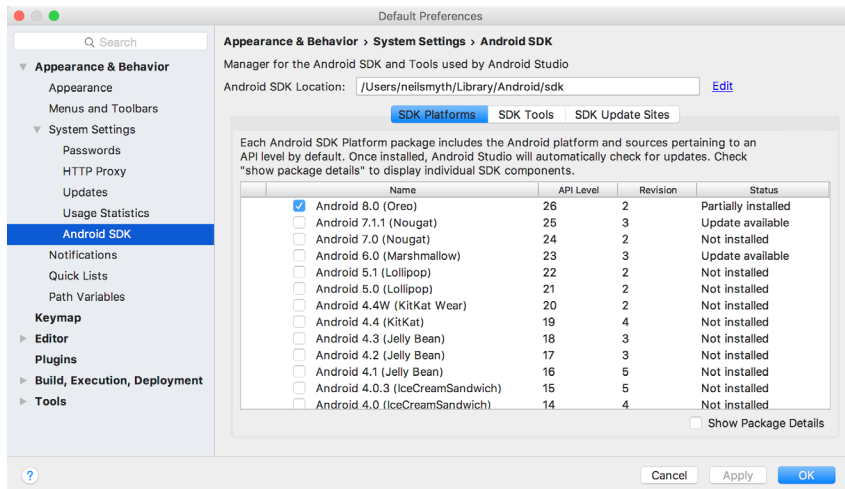


Figure 2-4

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 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:

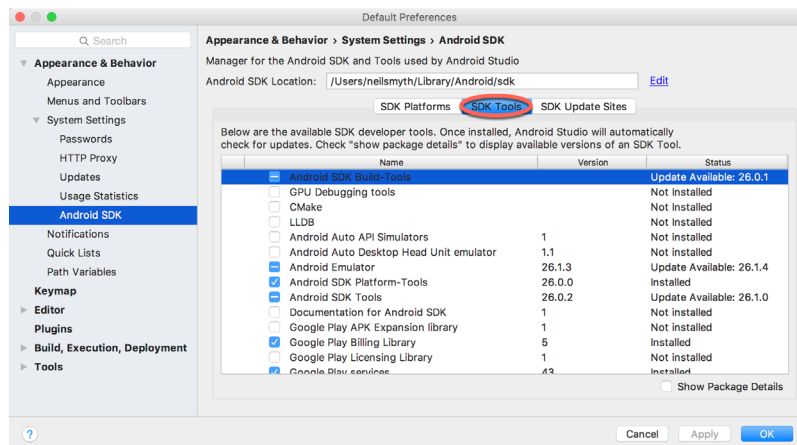


Figure 2-6

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
- Instant Apps Development SDK
- Intel x86 Emulator Accelerator (HAXM installer)
- ConstraintLayout for Android

Setting up an Android Studio Development Environment

- Solver for ConstraintLayout
- 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:

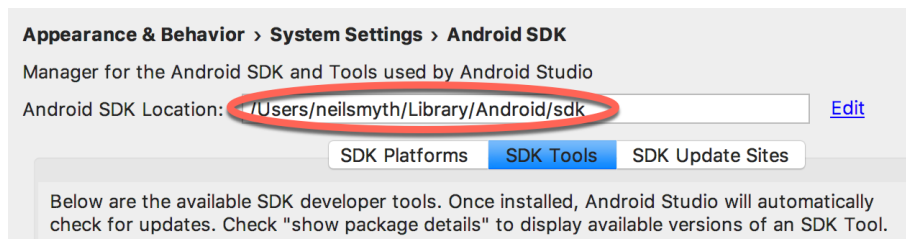


Figure 2-7

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

Setting up an Android Studio Development Environment

likely that an incorrect path was appended to the Path environment variable:

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

2.6.3 Windows 10

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

2.6.4 Linux

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

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

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

2.6.5 macOS

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

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

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

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

2.7 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:

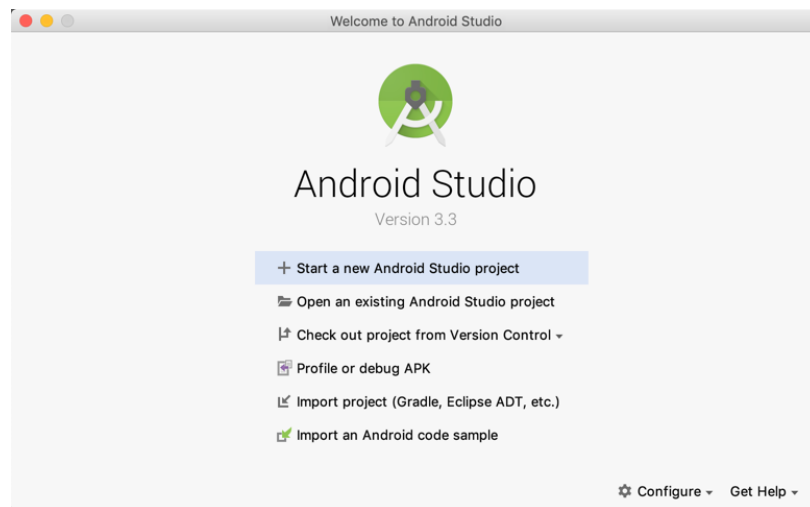


Figure 3-1

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.

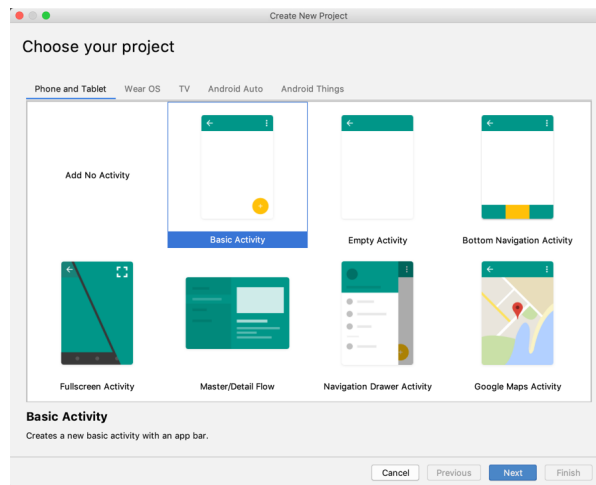


Figure 3-2

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.

Instant Apps will not be covered until later in this book so make sure that the *This project will support instant apps* option is disabled.

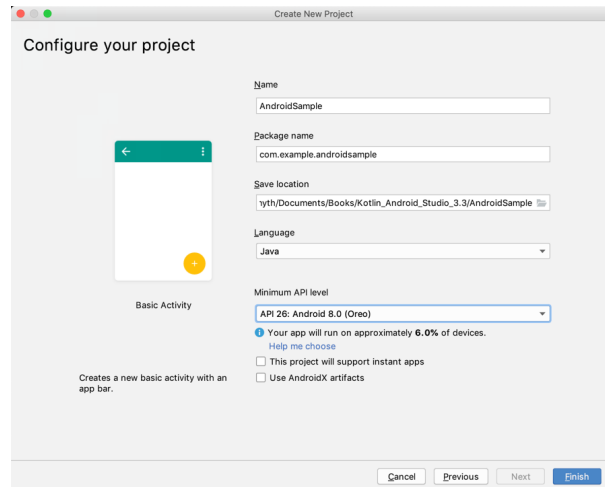


Figure 3-3

Finally, change the *Language* menu to *Kotlin* 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.

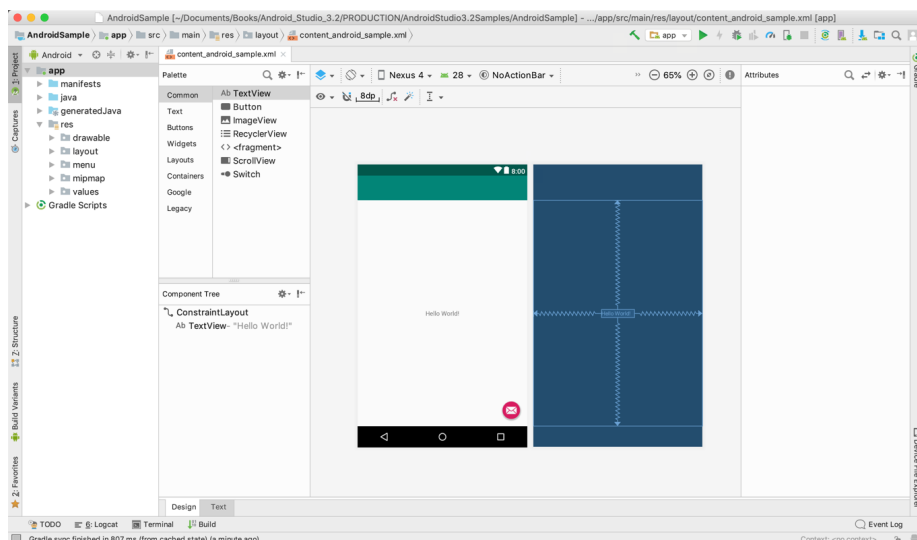


Figure 3-4

Creating an Example Android App in Android Studio

The newly created project and references to associated files are listed in the *Project* tool window located on the left-hand side of the main project window. The Project tool window has a number of modes in which information can be displayed. By default, this panel 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:

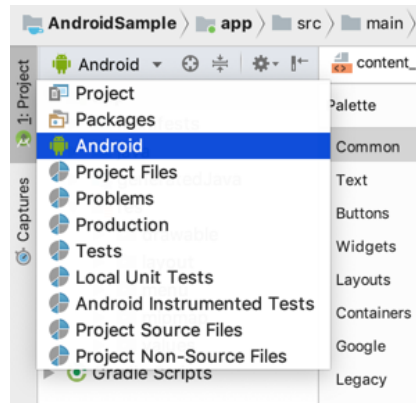


Figure 3-5

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):

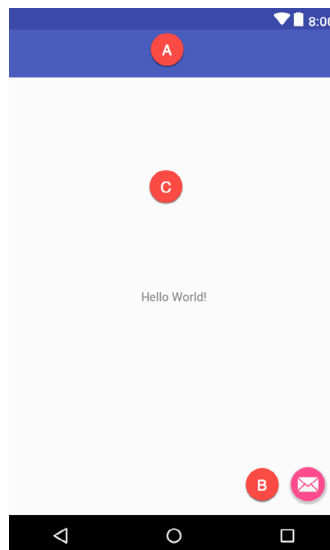


Figure 3-6

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:

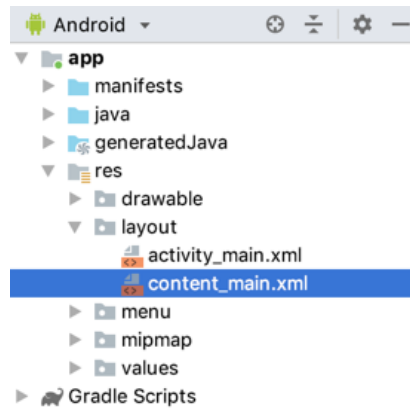


Figure 3-7

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:

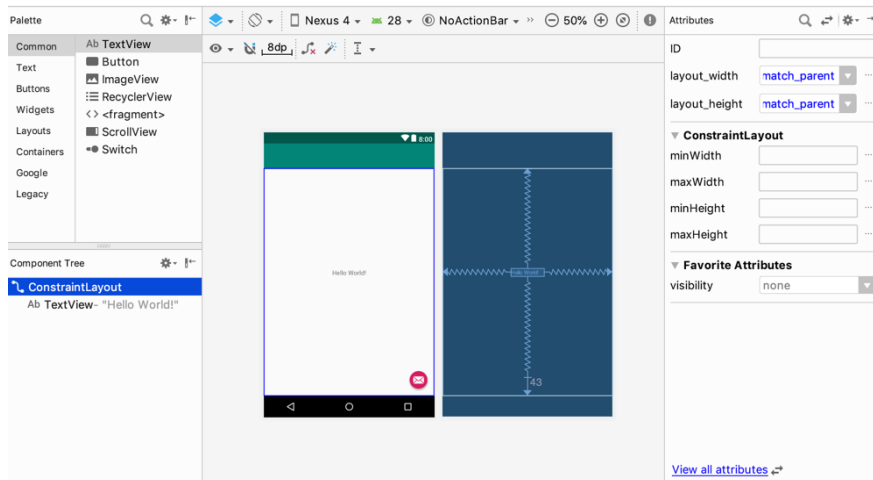



Figure 3-8

In the toolbar across the top of the Layout Editor window is a menu (currently set to *Nexus 4* 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

Creating an Example Android App in Android Studio

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

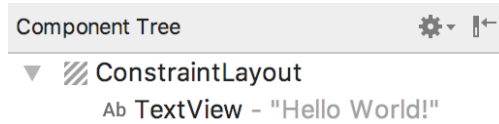


Figure 3-9

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

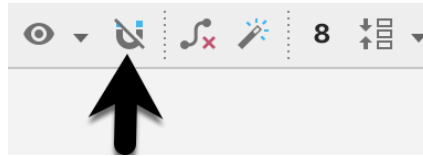


Figure 3-10

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:

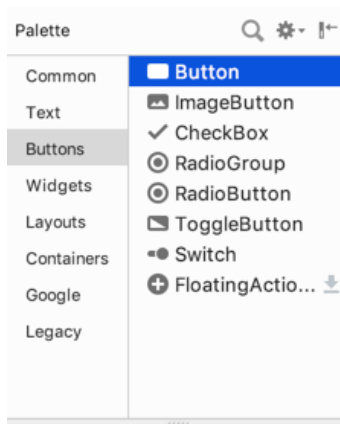


Figure 3-11

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:

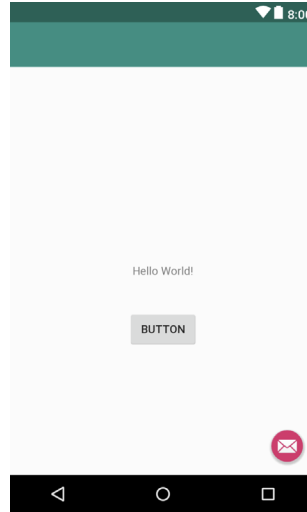


Figure 3-12

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 and change the current value from “Button” to “Convert” as shown in Figure 3-13:



Figure 3-13

A useful shortcut to changing the text property of a component is to double-click on it in the layout. This will automatically locate the attribute in the attributes panel and select it ready for editing.

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

Just in case the Autoconnect system failed to set all of the layout connections, click on the Infer constraints button (Figure 3-14) to add any missing constraints to the layout:

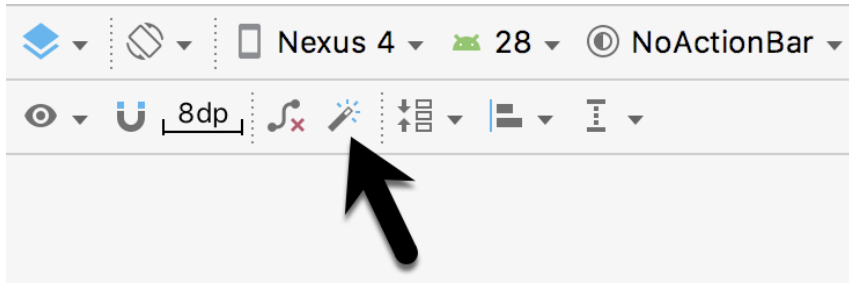


Figure 3-14

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:

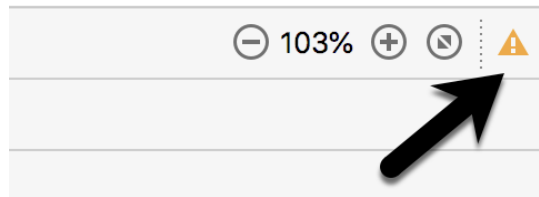


Figure 3-15

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

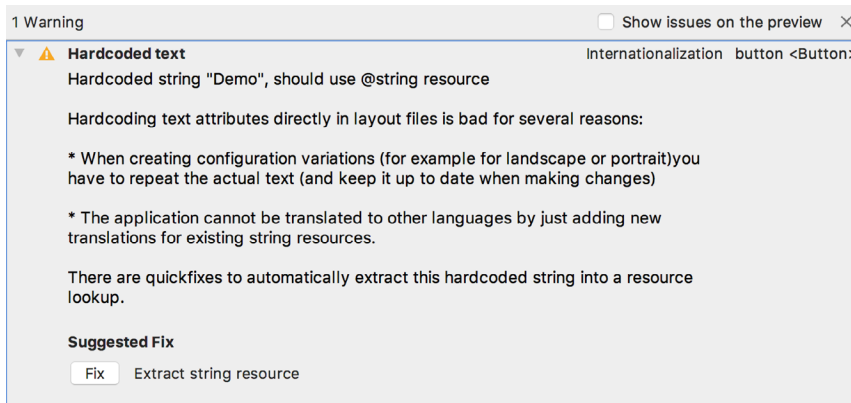


Figure 3-16

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.

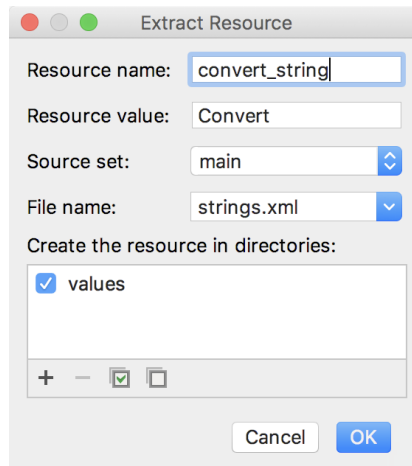


Figure 3-17

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 button displaying three dots 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:

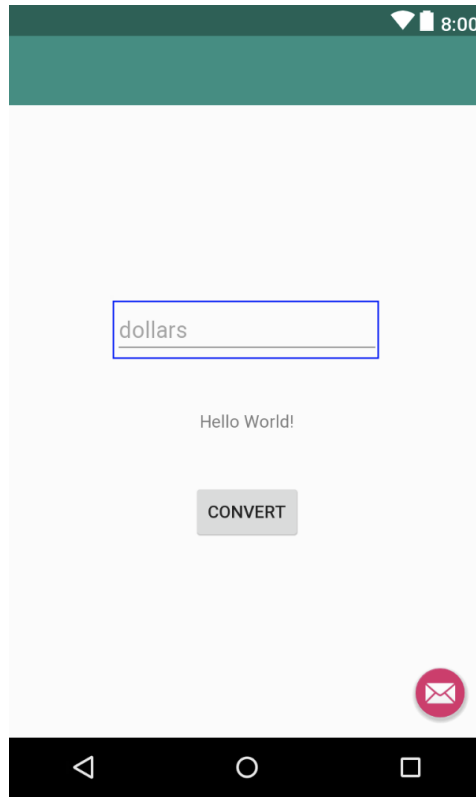


Figure 3-18

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:

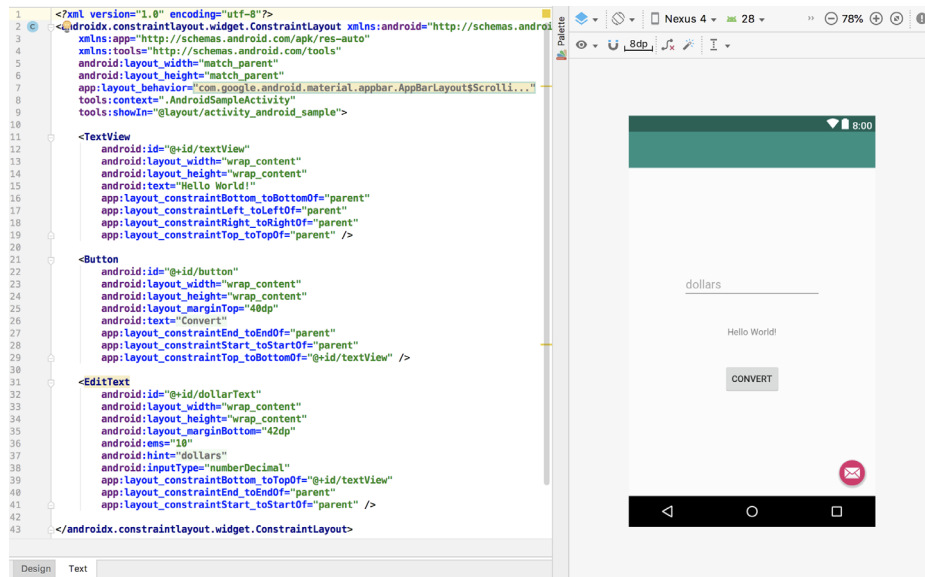


Figure 3-20

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>
```

Creating an Example Android App in Android Studio

Note that the color of the preview changes in real-time to match the new setting in the XML file. Note also that a small red square appears in the left-hand margin (also referred to as the *gutter*) of the XML editor next to the line containing the color setting. This is a visual cue to the fact that the color red has been set on a property. Change the color value to #a0ff28 and note that both the small square in the margin and the preview change to green.

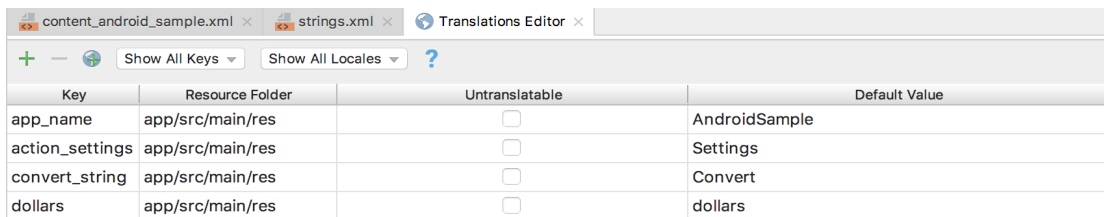
Finally, use the Project view to locate the *app -> res -> values -> strings.xml* file and double-click on it to load it into the editor. Currently the XML should read as follows:

```
<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 Editor* menu option. This will display the Translation Editor in the main panel of the Android Studio window:



The screenshot shows the Android Studio Translations Editor interface. At the top, there are tabs for 'content_android_sample.xml', 'strings.xml', and 'Translations Editor'. Below the tabs, there are buttons for '+', '-', a globe icon, 'Show All Keys', and 'Show All Locales'. A table with four columns is displayed: 'Key', 'Resource Folder', 'Untranslatable', and 'Default Value'. The table contains four rows of data.

Key	Resource Folder	Untranslatable	Default Value
app_name	app/src/main/res	<input type="checkbox"/>	AndroidSample
action_settings	app/src/main/res	<input type="checkbox"/>	Settings
convert_string	app/src/main/res	<input type="checkbox"/>	Convert
dollars	app/src/main/res	<input type="checkbox"/>	dollars

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:

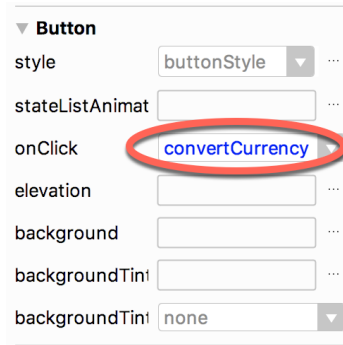


Figure 3-22

Next, double-click on the *MainActivity.kt* 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 com.google.android.material.snackbar.Snackbar
import android.support.v7.app.AppCompatActivity
import android.view.Menu
import android.view.MenuItem
import android.view.View

import kotlinx.android.synthetic.main.activity_main.*
import kotlinx.android.synthetic.main.content_main.*

class MainActivity : AppCompatActivity() {
    .
    .

    fun convertCurrency(view: View) {

        if (dollarText.text.isNotEmpty()) {

            val dollarValue = dollarText.text.toString().toFloat()

            val euroValue = dollarValue * 0.85f

            textView.text = euroValue.toString()
        } else {
            textView.text = "No Value"
        }
    }
    .
    .
}
```

```
}
```

The method begins by checking the text property of the `dollarText` `EditText` view to make sure that it is not empty (in other words that the user has entered a dollar value). If a value has not been entered, a “No Value” string is displayed on the `resultText` view. If, on the other hand, a dollar amount has been entered, it is converted into a floating point value and the equivalent euro value calculated. This floating point value is then converted into a string and displayed on the `resultText` view. 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.

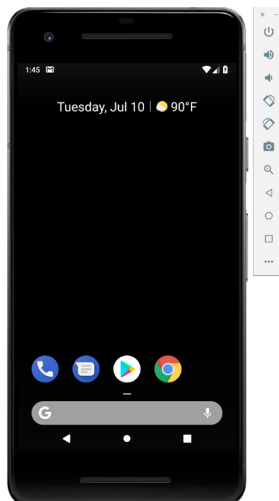


Figure 4-1

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 -> Android -> 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:

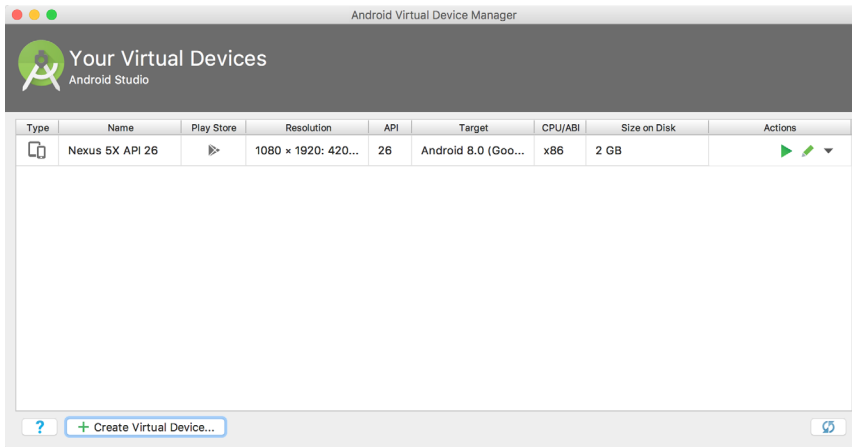


Figure 4-2

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

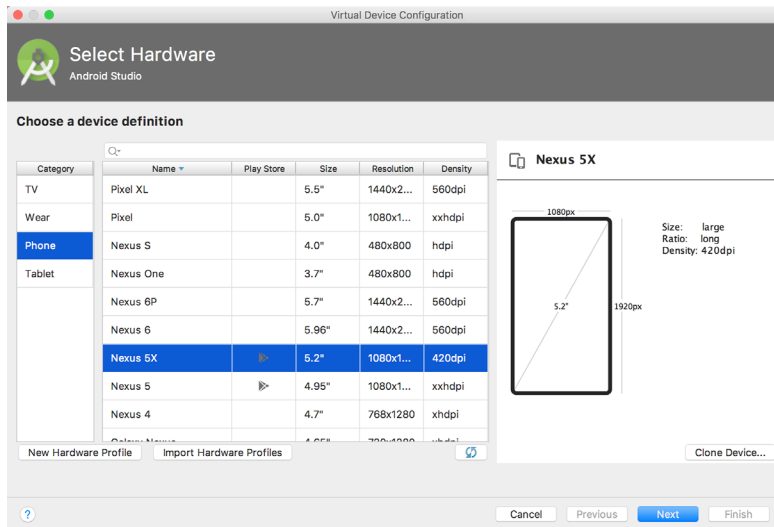


Figure 4-3

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:

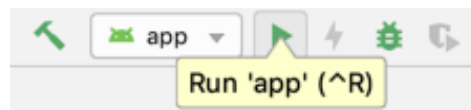


Figure 4-4

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

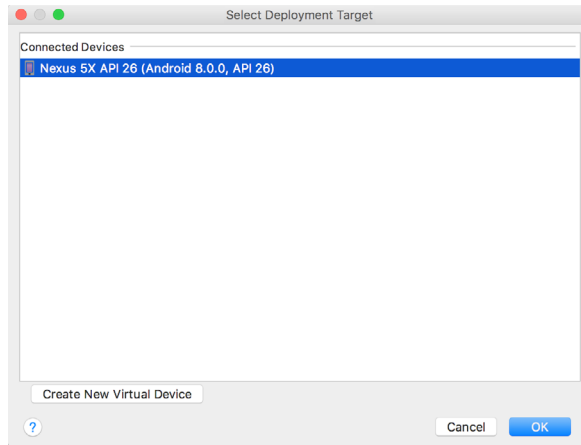


Figure 4-5

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

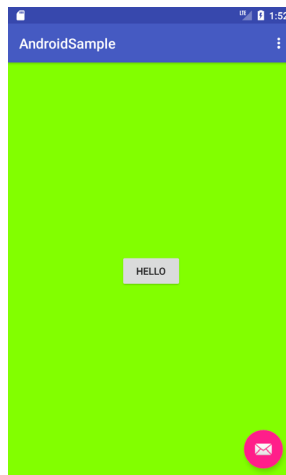


Figure 4-6

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:

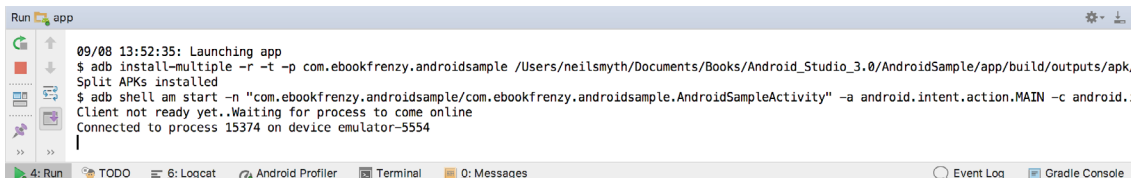


Figure 4-7

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:

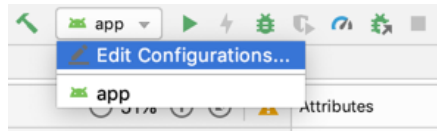


Figure 4-8

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:

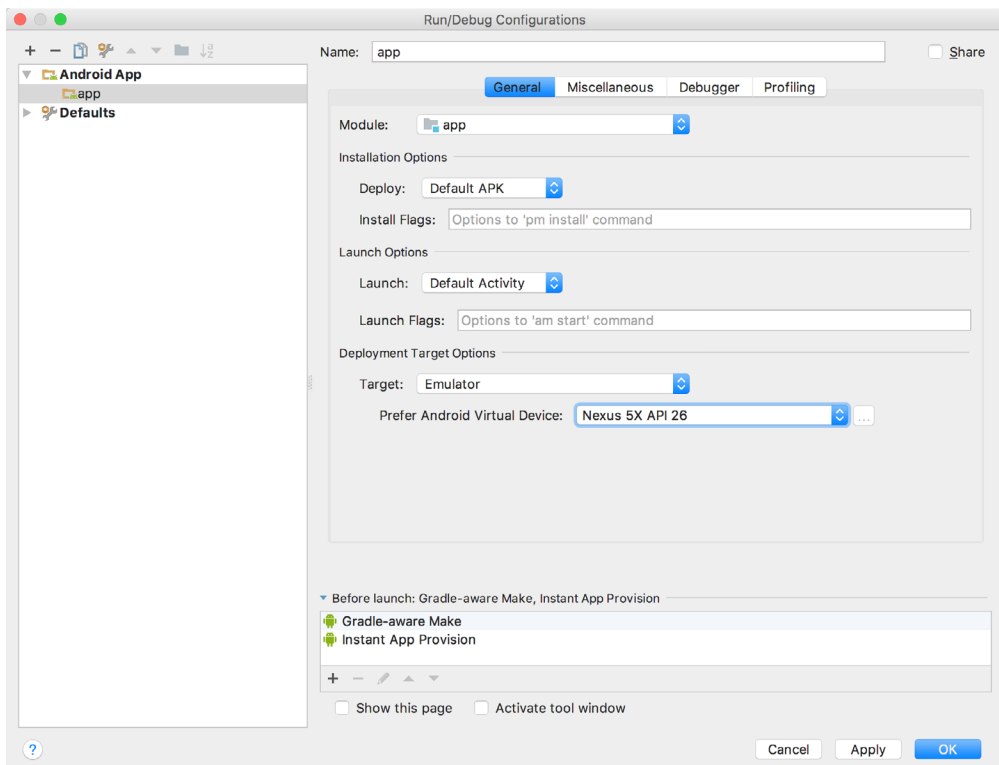


Figure 4-9

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:

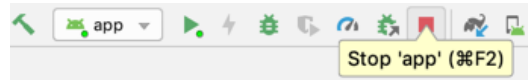


Figure 4-10

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:

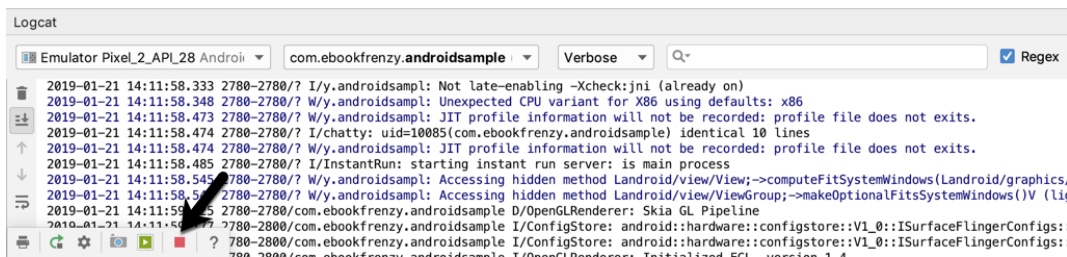


Figure 4-11

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:

```
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.

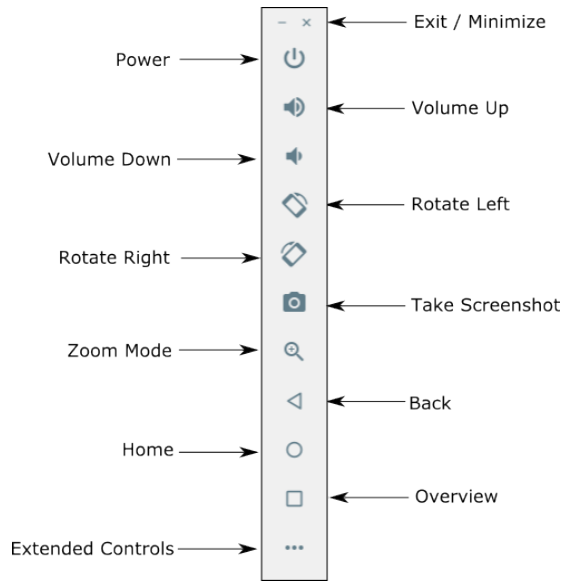


Figure 5-2

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:

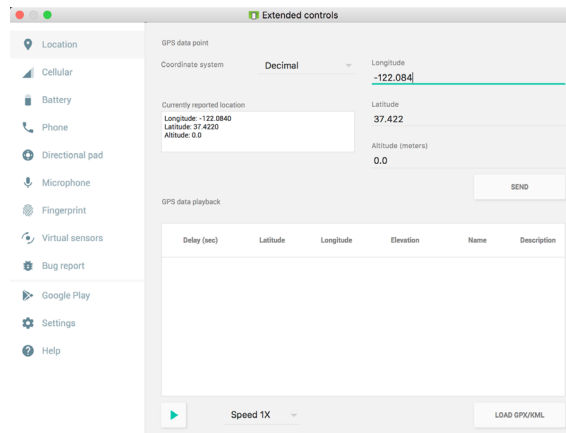


Figure 5-3

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 Google Play

Google Play will need to be present on AVD instances on which Google services such as in-app purchasing need to be tested. This extended controls screen displays the version of Google Play installed on the current AVD instance and provides the option to update or deactivate the Google Play installation. When the Update button is clicked, the appropriate Google Play Store page will load on the emulator allowing changes to be made to the installation, including deactivating Google Play.

5.5.13 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.

5.5.14 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:

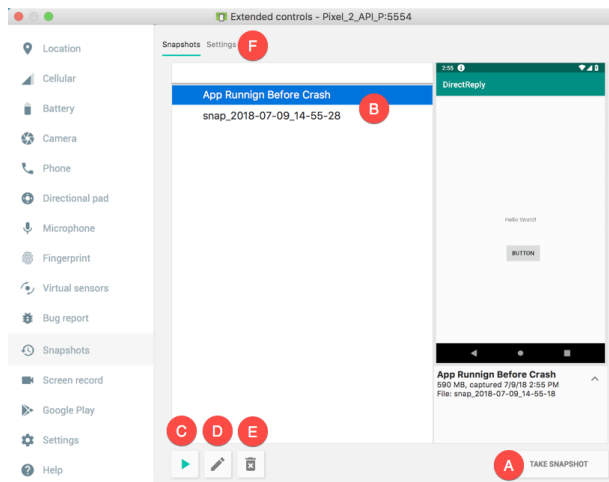


Figure 5-4

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.

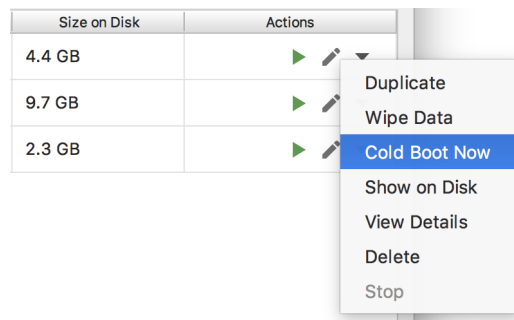


Figure 5-5

5.7 Drag and Drop Support

An Android application is packaged into an APK file when it is built. When Android Studio built and ran the AndroidSample app created earlier in this book, for example, the application was compiled and packaged into an APK file. That APK file was then transferred to the emulator and launched.

The Android Studio emulator also supports installation of apps by dragging and dropping the corresponding APK file onto the emulator window. To experience this in action, start the emulator, open Settings and select the *Apps & notifications* option followed by the *App Info* option on the subsequent screen. Within the list of installed apps, locate and select the AndroidSample app and, in the app detail screen, uninstall the app from the emulator.

Open the file system navigation tool for your operating system (e.g. Windows Explorer for Windows or Finder for macOS) and navigate to the folder containing the AndroidSample project. Within this folder locate the *app/build/outputs/apk/debug* subfolder which should contain an APK file named *app-debug.apk*. Drag this file and drop it onto the emulator window. The dialog shown in (Figure 5-6) will subsequently appear as the APK file is installed.

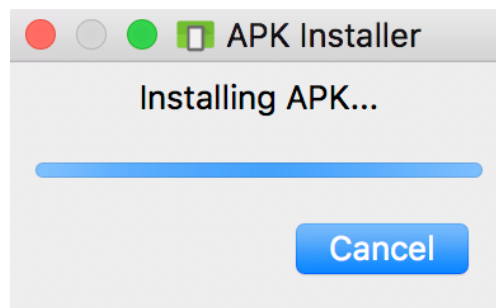


Figure 5-6

Once the APK file installation has completed, locate the app on the device and click on it to launch it.

In addition to APK files, any other type of file such as image, video or data files can be installed onto the emulator using this drag and drop feature. Such files are added to the SD card storage area of the emulator where they may subsequently be accessed from within app code.

5.8 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:

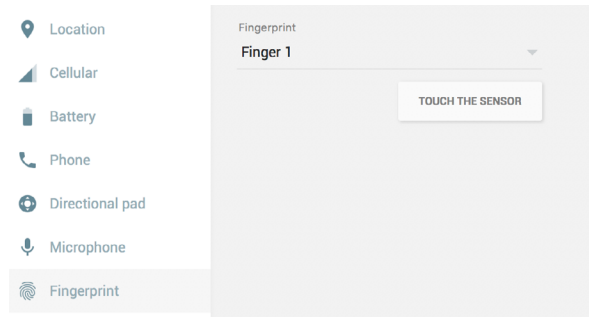


Figure 5-7

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:

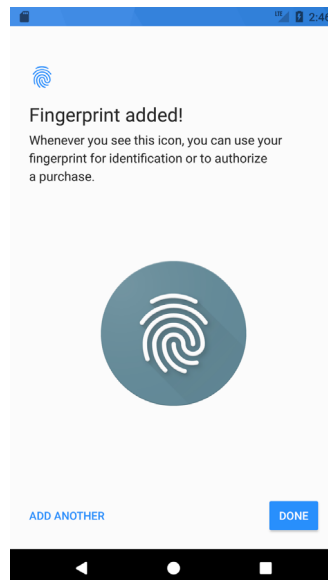


Figure 5-8

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*”.

5.9 Summary

Android Studio 3.3 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.

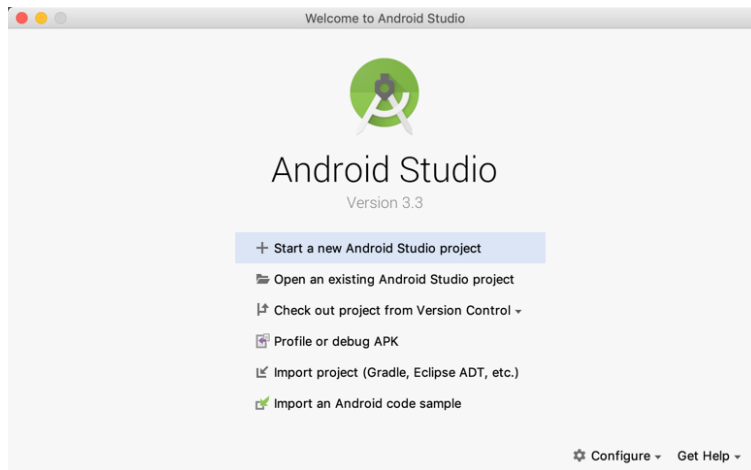


Figure 6-1

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.

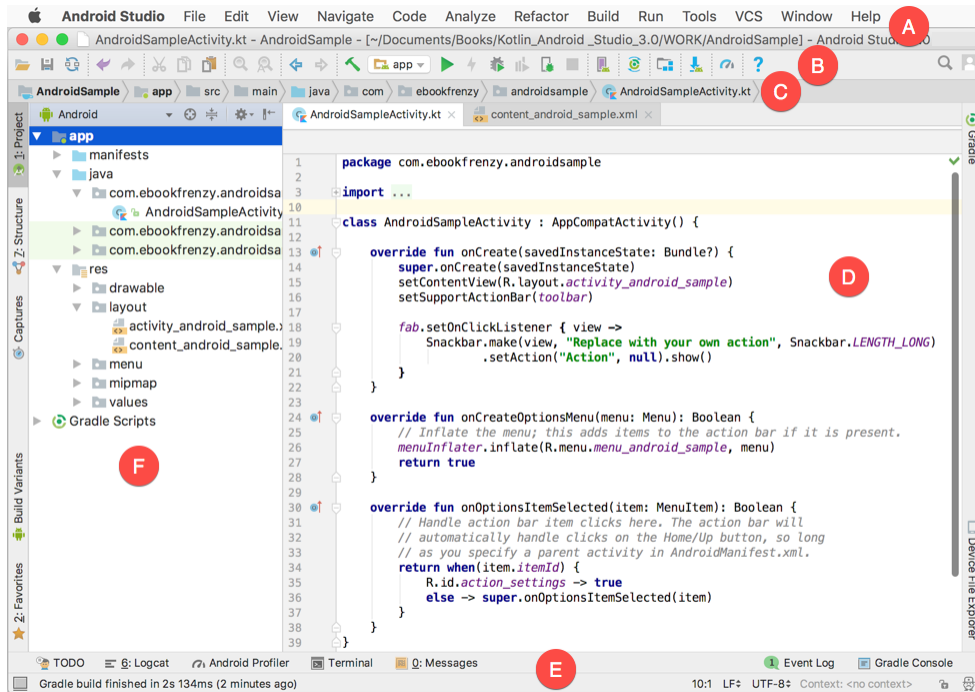


Figure 6-2

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.

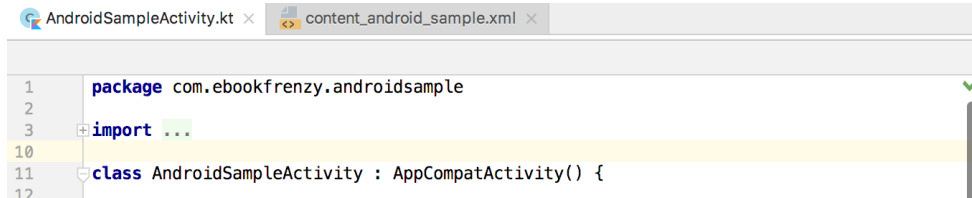


Figure 6-3

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.

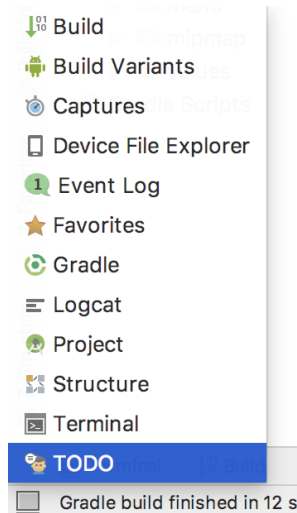


Figure 6-4

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.

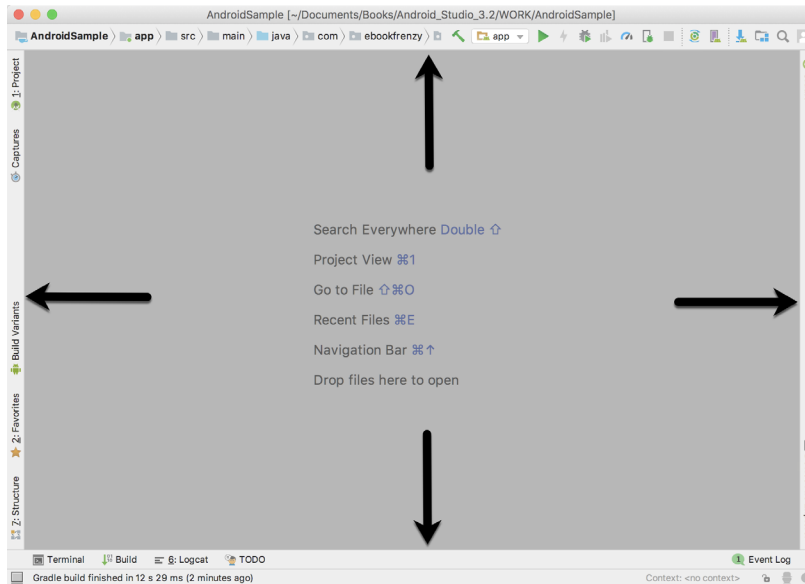


Figure 6-5

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.

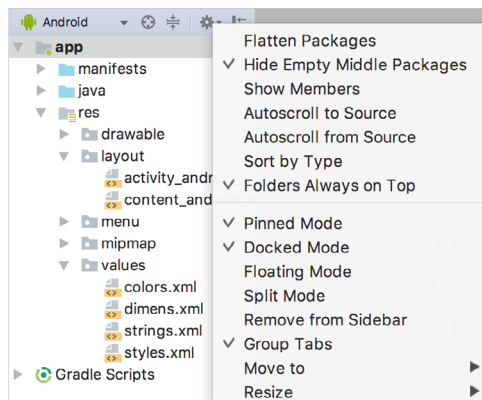


Figure 6-6

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.

Captures – The captures tool window provides access to performance data files that have been generated by the monitoring tools contained within Android Studio.

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*.

Messages – The messages tool window records output from the Gradle build system (Gradle is the underlying system used by Android Studio for building the various parts of projects into runnable applications) and can be useful for identifying the causes of build problems when compiling application projects.

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.

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.

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.

Android 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.

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).

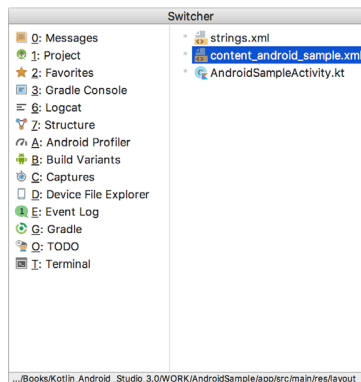


Figure 6-7

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.

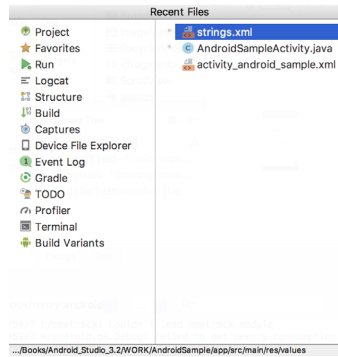


Figure 6-8

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, Default and Darcula. Figure 6-9 shows an example of the main window with the Darcula theme selected:

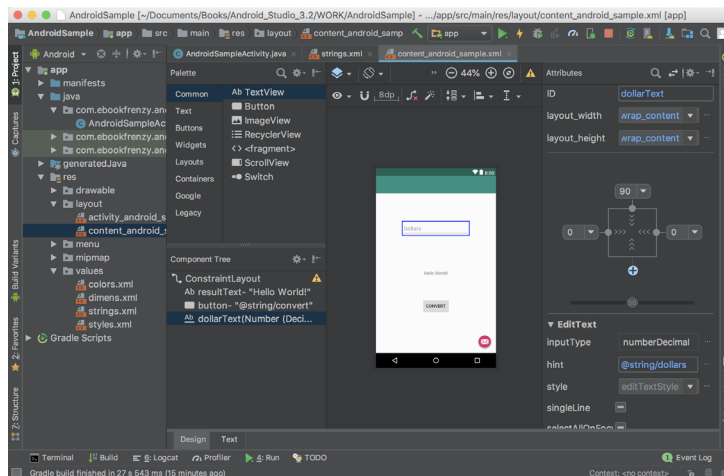


Figure 6-9

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.

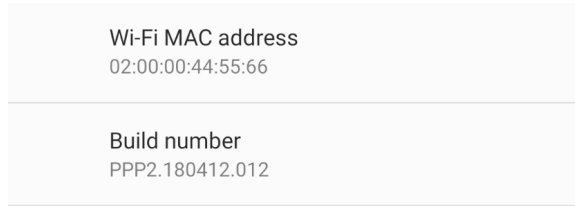


Figure 7-1

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

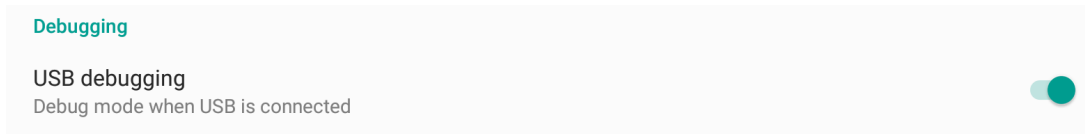


Figure 7-2

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

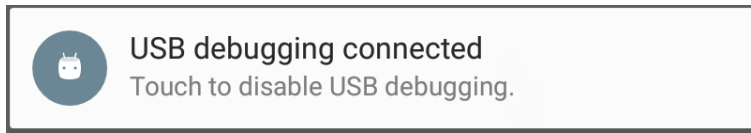


Figure 7-3

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

7.2.1 macOS ADB Configuration

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

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

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

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

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

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

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

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

7.2.2 Windows ADB Configuration

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

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

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

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

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

```
adb devices
```

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

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

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

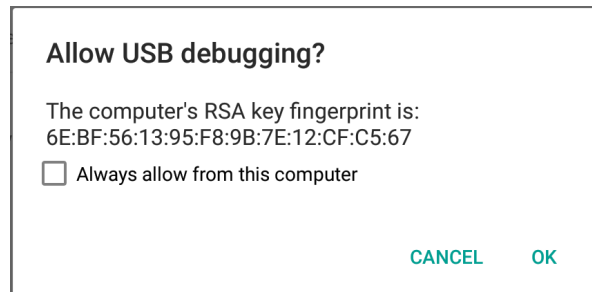


Figure 7-4

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

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

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

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).

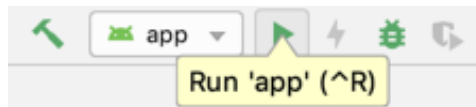


Figure 7-5

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.

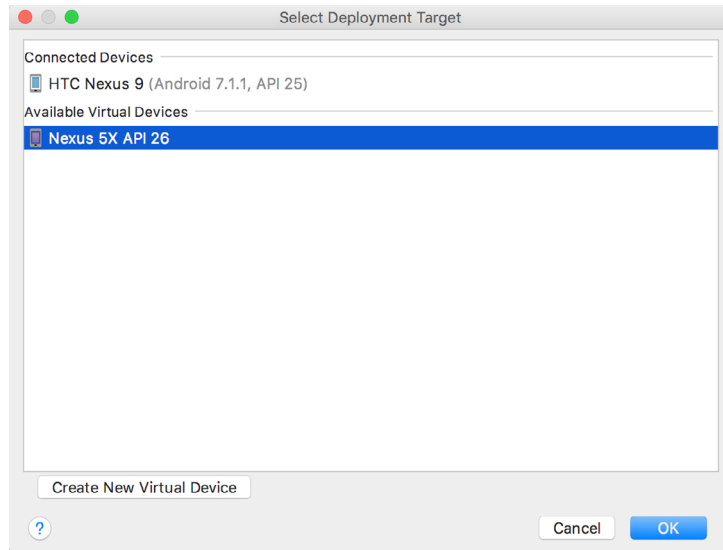


Figure 7-6

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 Kotlin source code file loaded:

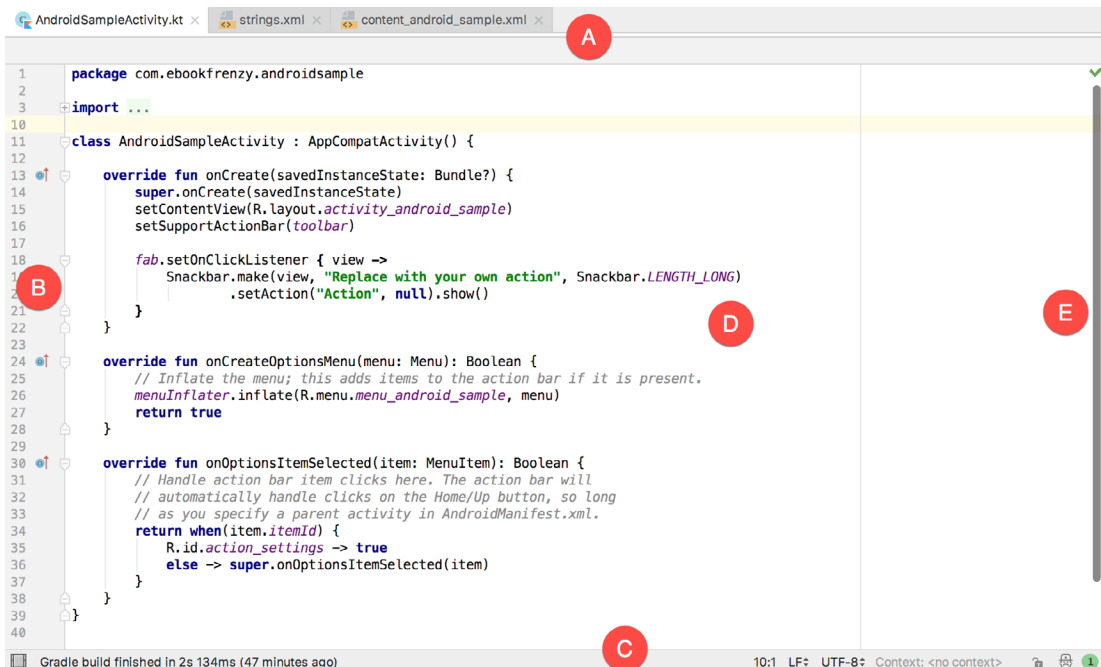


Figure 8-1

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):

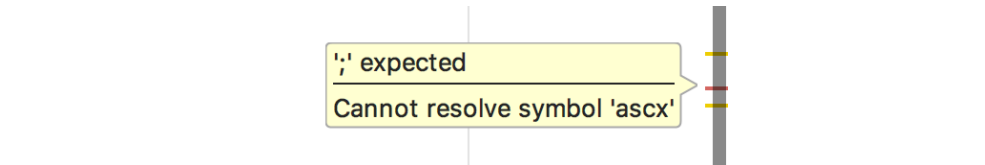


Figure 8-3

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:

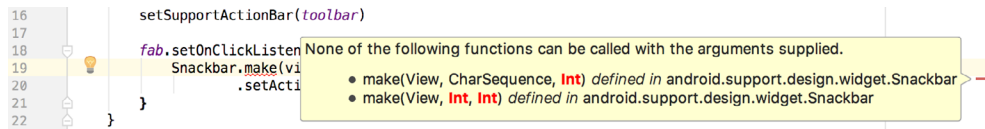


Figure 8-4

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:

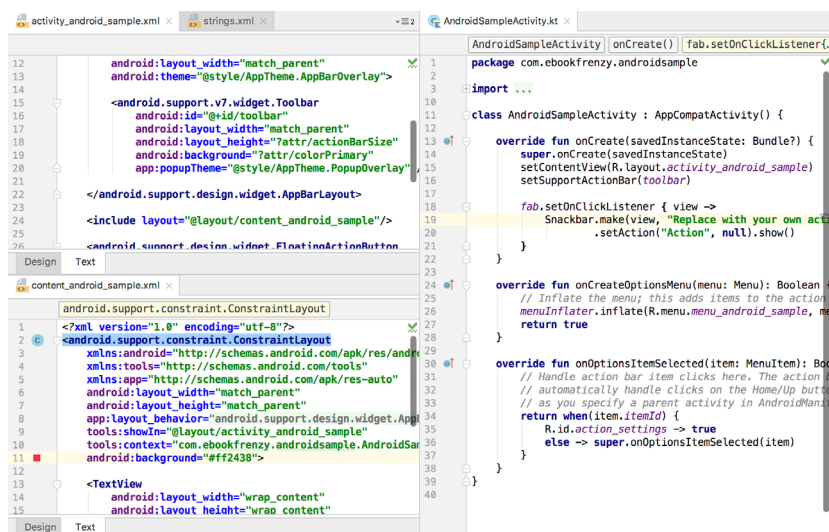


Figure 8-5

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 Kotlin 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:

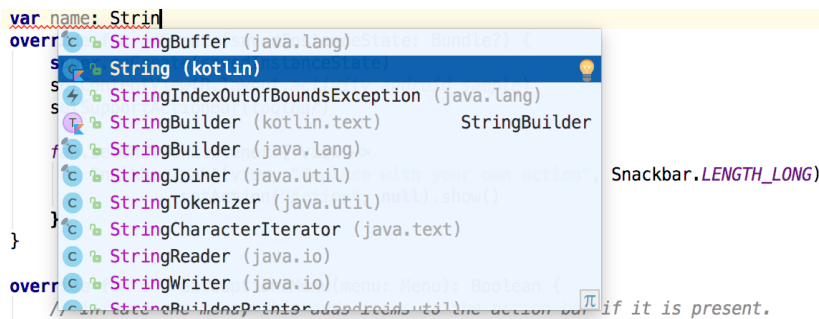


Figure 8-6

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:

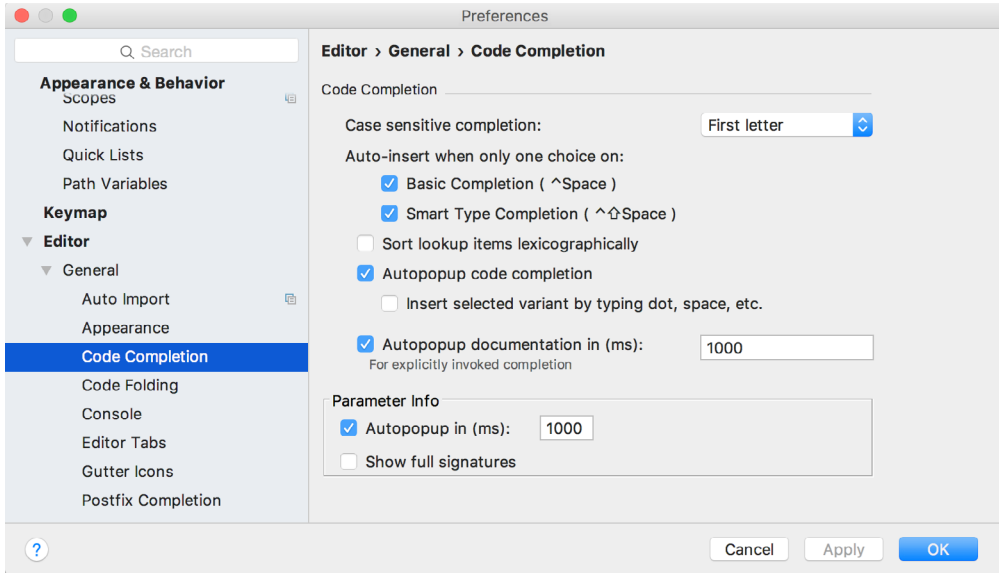


Figure 8-7

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:

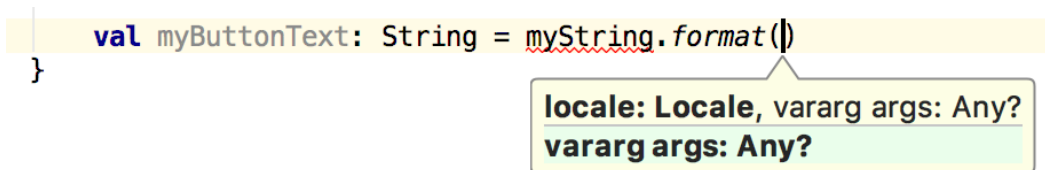


Figure 8-8

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.

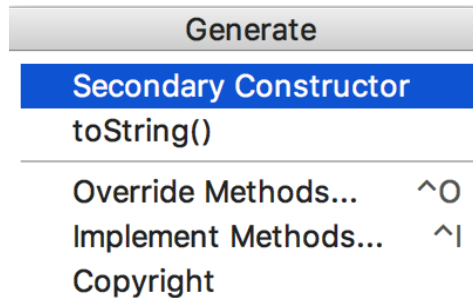


Figure 8-10

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:

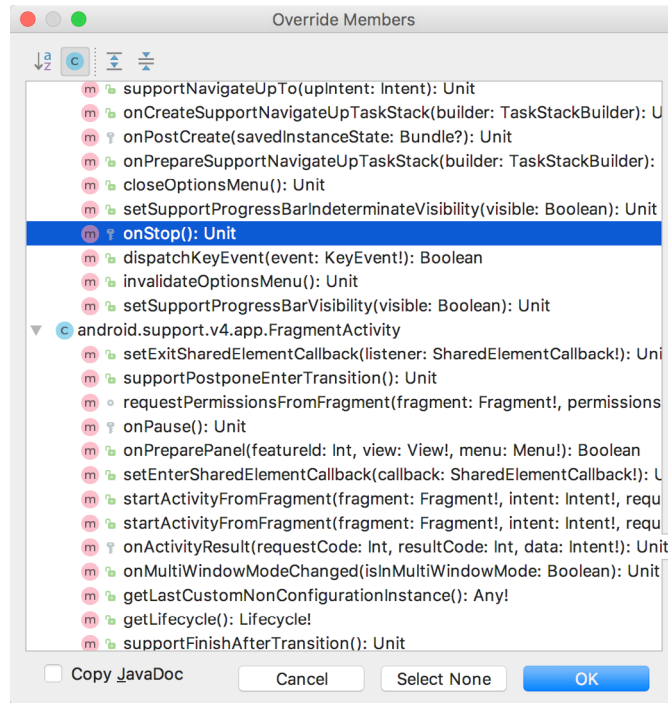


Figure 8-11

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

```
override fun 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:

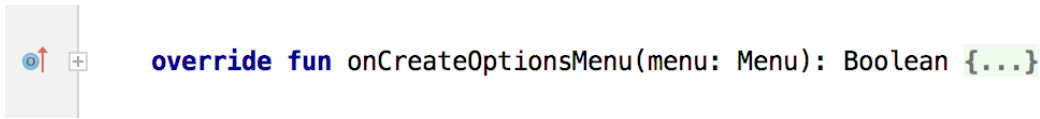


Figure 8-13

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:

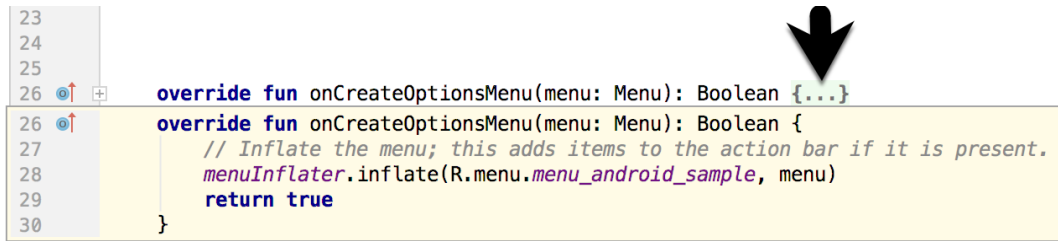


Figure 8-14

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):

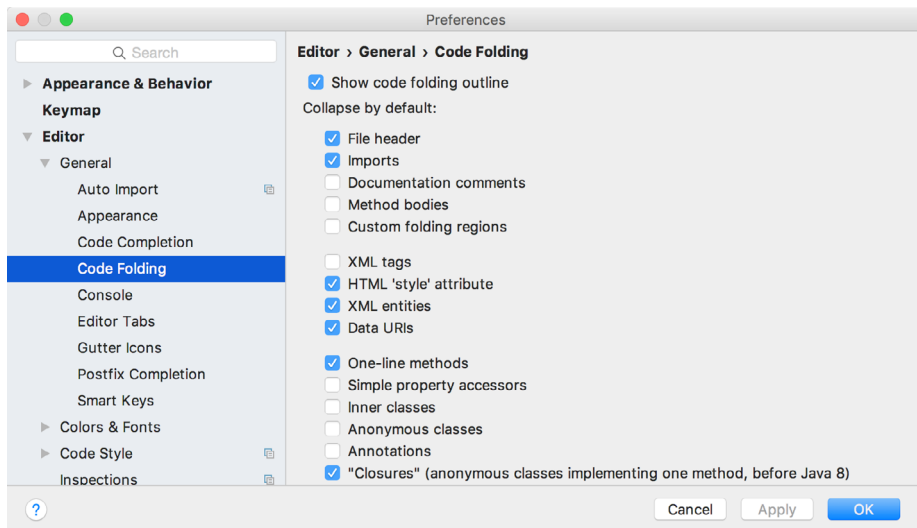


Figure 8-15

8.9 Quick Documentation Lookup

Context sensitive Kotlin 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-Alt-L* on macOS) keyboard shortcut sequence. To display the *Reformat Code* dialog (Figure 8-17) use the *Ctrl-Alt-Shift-L* (*Cmd-Alt-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.

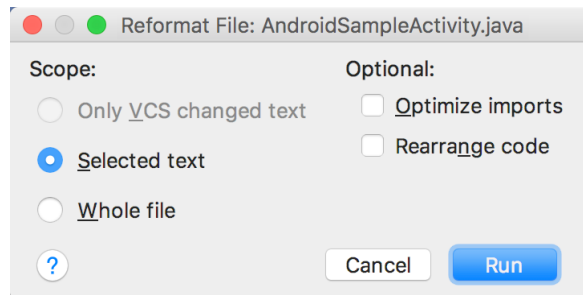


Figure 8-17

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 Kotlin and, from the Kotlin settings, select the *Arrangement* tab.

8.11 Finding Sample Code

The Android Studio editor provides a way to access sample code relating to the currently highlighted entry within the code listing. This feature can be useful for learning how a particular Android class or method is used. To find sample code, highlight a method or class name in the editor, right-click on it and select the *Find Sample*

Code menu option. The Find Sample Code panel (Figure 8-18) 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:

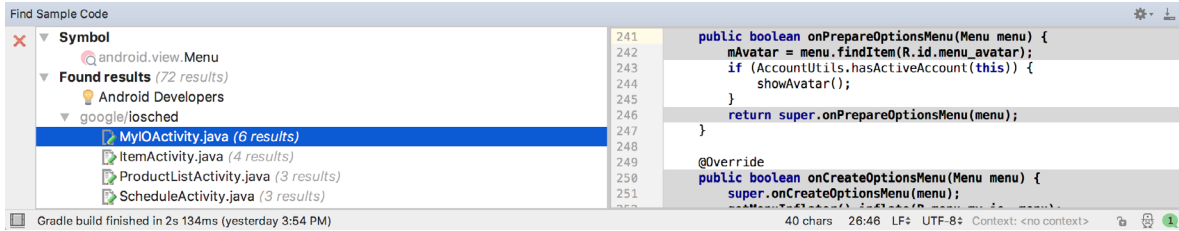


Figure 8-18

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.

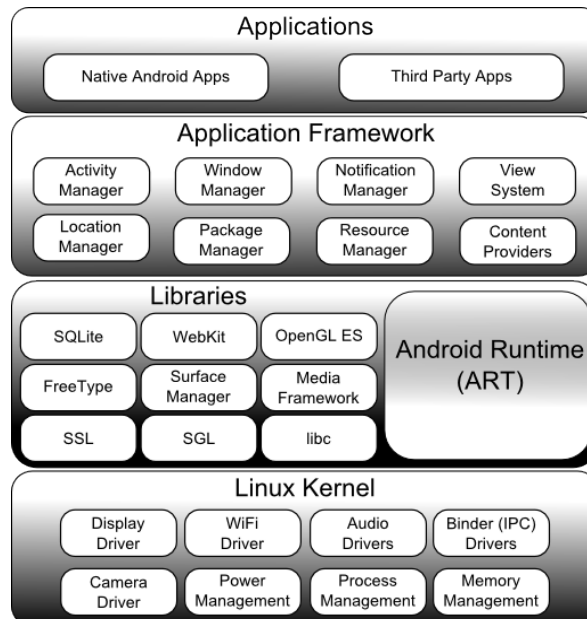


Figure 9-1

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

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

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

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

11. An Introduction to Kotlin

Android development is performed primarily using Android Studio which is, in turn, based on the IntelliJ IDEA development environment created by a company named JetBrains. Prior to the release of Android Studio 3.0, all Android apps were written using Android Studio and the Java programming language (with some occasional C++ code when needed).

Since the introduction of Android Studio 3.0, however, developers now have the option of creating Android apps using another programming language called Kotlin. Although detailed coverage of all features of this language is beyond the scope of this book (entire books can and have been written covering solely Kotlin), the objective of this and the following six chapters is to provide enough information to begin programming in Kotlin and quickly get up to speed developing Android apps using this programming language.

11.1 What is Kotlin?

Named after an island located in the Baltic Sea, Kotlin is a programming language created by JetBrains and follows Java in the tradition of naming programming languages after islands. Kotlin code is intended to be easier to understand and write and also safer than many other programming languages. The language, compiler and related tools are all open source and available for free under the Apache 2 license.

The primary goals of the Kotlin language are to make code both concise and safe. Code is generally considered concise when it can be easily read and understood. Conciseness also plays a role when writing code, allowing code to be written more quickly and with greater efficiency. In terms of safety, Kotlin includes a number of features that improve the chances that potential problems will be identified when the code is being written instead of causing runtime crashes.

A third objective in the design and implementation of Kotlin involves interoperability with Java.

11.2 Kotlin and Java

Originally introduced by Sun Microsystems in 1995 Java is still by far the most popular programming language in use today. Until the introduction of Kotlin, it is quite likely that every Android app available on the market was written in Java. Since acquiring the Android operating system, Google has invested heavily in tuning and optimizing compilation and runtime environments for running Java-based code on Android devices.

Rather than try to re-invent the wheel, Kotlin is design to both integrate with and work alongside Java. When Kotlin code is compiled it generates the same bytecode as that generated by the Java compiler enabling projects to be built using a combination of Java and Kotlin code. This compatibility also allows existing Java frameworks and libraries to be used seamlessly from within Kotlin code and also for Kotlin code to be called from within Java.

Kotlin's creators also acknowledged that while there were ways to improve on existing languages, there are many features of Java that did not need to be changed. Consequently, those familiar with programming in Java will find many of these skills to be transferable to Kotlin-based development. Programmers with Swift programming experience will also find much that is familiar when learning Kotlin.

11.3 Converting from Java to Kotlin

Given the high level of interoperability between Kotlin and Java it is not essential to convert existing Java code to Kotlin since these two languages will comfortably co-exist within the same project. That being said, Java code

can be converted to Kotlin from within Android Studio using a built-in Java to Kotlin converter. To convert an entire Java source file to Kotlin, load the file into the Android Studio code editor and select the *Code -> Convert Java File to Kotlin File* menu option. Alternatively, blocks of Java code may be converted to Kotlin by cutting the code and pasting it into an existing Kotlin file within the Android Studio code editor. Note when performing Java to Kotlin conversions that the Java code will not always convert to the best possible Kotlin code and that time should be taken to review and tidy up the code after conversion.

11.4 Kotlin and Android Studio

Support for Kotlin is provided within Android Studio via the Kotlin Plug-in which is integrated by default into Android Studio 3.0 or later.

11.5 Experimenting with Kotlin

When learning a new programming language, it is often useful to be able to enter and execute snippets of code. One of the best ways to do this with Kotlin is to use the online playground (Figure 11-1) located at <https://try.kotlin.in>. In addition to providing an environment in which Kotlin code may be quickly entered and executed, the online playground also includes a set of examples demonstrating key Kotlin features in action.

The panel on the left-hand side (marked A in Figure 11-1) contains a list of coding examples together with any examples you create. Code is typed into the main panel (B) and executed by clicking the Run button (C). Any output from the code execution appears in the console panel (D). Arguments may be passed through to the main function by entering them into the field marked E.

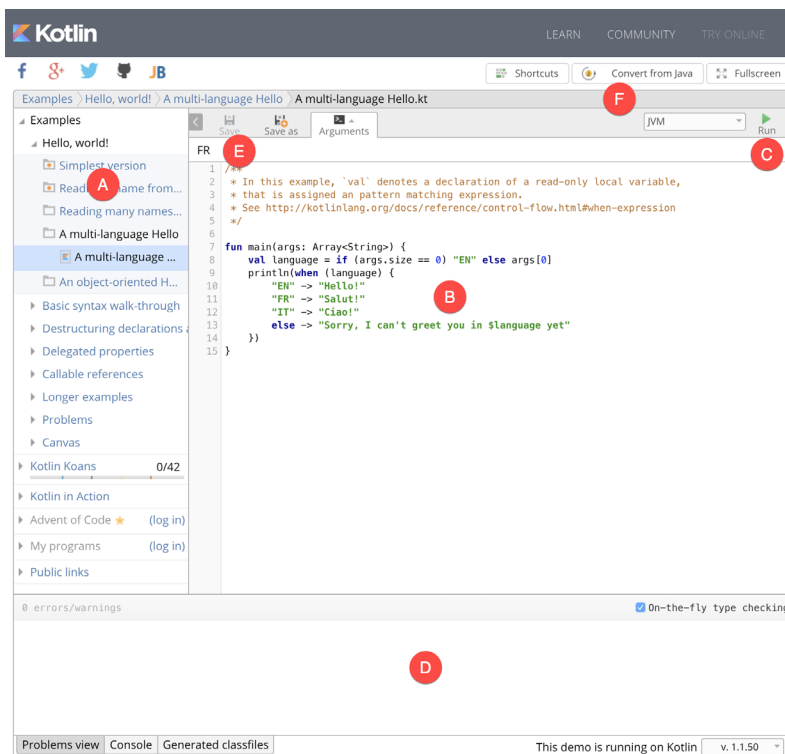


Figure 11-1

Try out some Kotlin code by opening a browser window, navigating to the online playground and entering the following into the main code panel:


```
fun main(args: Array<String>) {

    println("Welcome to Kotlin")

    for (i in 1..8) {
        println("i = $i")
    }
}
```

After entering the code, click on the Run button and note the output in the console panel:

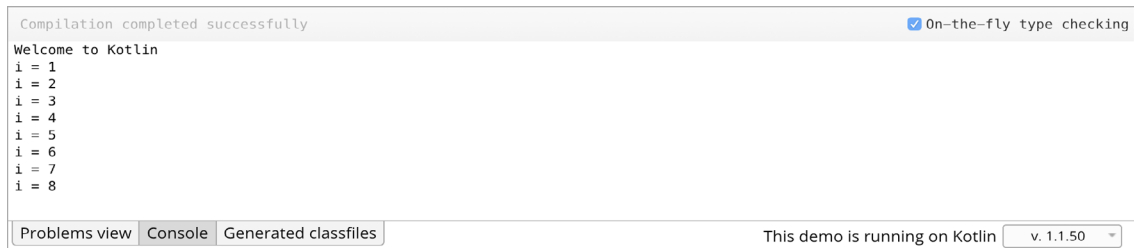


Figure 11-2

The online playground may also be used to find the Kotlin equivalent for fragments of Java code. Simply enter (or cut and paste) the Java code into the main panel and click on the Convert from Java button (marked E).

11.6 Semi-colons in Kotlin

Unlike programming languages such as Java and C++, Kotlin does not require semi-colons at the end of each statement or expression line. The following, therefore, is valid Kotlin code:

```
val mynumber = 10
println(mynumber)
```

Semi-colons are only required when multiple statements appear on the same line:

```
val mynumber = 10; println(mynumber)
```

11.7 Summary

For the first time since the Android operating system was introduced, developers now have an alternative to writing apps in Java code. Kotlin is a programming language developed by JetBrains, the company that created the development environment on which Android Studio is based. Kotlin is intended to make code safer and easier to understand and write. Kotlin is also highly compatible with Java, allowing Java and Kotlin code to co-exist within the same projects. This interoperability ensures that most of the standard Java and Java-based Android libraries and frameworks are available for use when developing using Kotlin.

Kotlin support for Android Studio is provided via a plug-in bundled with Android Studio 3.0 or later. This plug-in also provides a converter to translate Java code to Kotlin.

When learning Kotlin, the online playground provides a useful environment for quickly trying out Kotlin code.

12. Kotlin Data Types, Variables and Nullability

Both this and the following few chapters are intended to introduce the basics of the Kotlin programming language. This chapter will focus on the various data types available for use within Kotlin code. This will also include an explanation of constants, variables, type casting and Kotlin's handling of null values.

As outlined in the previous chapter, entitled “*An Introduction to Kotlin*” a useful way to experiment with the language is to use the Kotlin online playground environment. Before starting this chapter, therefore, open a browser window, navigate to <https://try.kotl.in> and use the playground to try out the code in both this and the other Kotlin introductory chapters that follow.

12.1 Kotlin Data Types

When we look at the different types of software that run on computer systems and mobile devices, from financial applications to graphics intensive games, it is easy to forget that computers are really just binary machines. Binary systems work in terms of 0 and 1, true or false, set and unset. All the data sitting in RAM, stored on disk drives and flowing through circuit boards and buses are nothing more than sequences of 1s and 0s. Each 1 or 0 is referred to as a bit and bits are grouped together in blocks of 8, each group being referred to as a byte. When people talk about 32-bit and 64-bit computer systems they are talking about the number of bits that can be handled simultaneously by the CPU bus. A 64-bit CPU, for example, is able to handle data in 64-bit blocks, resulting in faster performance than a 32-bit based system.

Humans, of course, don't think in binary. We work with decimal numbers, letters and words. In order for a human to easily ('easily' being a relative term in this context) program a computer, some middle ground between human and computer thinking is needed. This is where programming languages such as Kotlin come into play. Programming languages allow humans to express instructions to a computer in terms and structures we understand, and then compile that down to a format that can be executed by a CPU.

One of the fundamentals of any program involves data, and programming languages such as Kotlin define a set of *data types* that allow us to work with data in a format we understand when programming. For example, if we want to store a number in a Kotlin program we could do so with syntax similar to the following:

```
val mynumber = 10
```

In the above example, we have created a variable named *mynumber* and then assigned to it the value of 10. When we compile the source code down to the machine code used by the CPU, the number 10 is seen by the computer in binary as:

```
1010
```

Similarly, we can express a letter, the visual representation of a digit ('0' through to '9') or punctuation mark (referred to in computer terminology as *characters*) using the following syntax:

```
val myletter = 'c'
```

Once again, this is understandable by a human programmer, but gets compiled down to a binary sequence for the CPU to understand. In this case, the letter 'c' is represented by the decimal number 99 using the ASCII table (an internationally recognized standard that assigns numeric values to human readable characters). When

converted to binary, it is stored as:

```
10101100011
```

Now that we have a basic understanding of the concept of data types and why they are necessary we can take a closer look at some of the more commonly used data types supported by Kotlin.

12.1.1 Integer Data Types

Kotlin integer data types are used to store whole numbers (in other words a number with no decimal places). All integers in Kotlin are signed (in other words capable of storing positive, negative and zero values).

Kotlin provides support for 8, 16, 32 and 64 bit integers (represented by the Byte, Short, Int and Long types respectively).

12.1.2 Floating Point Data Types

The Kotlin floating point data types are able to store values containing decimal places. For example, 4353.1223 would be stored in a floating point data type. Kotlin provides two floating point data types in the form of Float and Double. Which type to use depends on the size of value to be stored and the level of precision required. The Double type can be used to store up to 64-bit floating point numbers. The Float data type, on the other hand, is limited to 32-bit floating point numbers.

12.1.3 Boolean Data Type

Kotlin, like other languages, includes a data type for the purpose of handling true or false (1 or 0) conditions. Two Boolean constant values (*true* and *false*) are provided by Kotlin specifically for working with Boolean data types.

12.1.4 Character Data Type

The Kotlin Char data type is used to store a single character of rendered text such as a letter, numerical digit, punctuation mark or symbol. Internally characters in Kotlin are stored in the form of 16-bit Unicode grapheme clusters. A grapheme cluster is made of two or more Unicode code points that are combined to represent a single visible character.

The following lines assign a variety of different characters to Character type variables:

```
val myChar1 = 'f'  
val myChar2 = ':'  
val myChar3 = 'X'
```

Characters may also be referenced using Unicode code points. The following example assigns the 'X' character to a variable using Unicode:

```
val myChar4 = '\u0058'
```

Note the use of single quotes when assigning a character to a variable. This indicates to Kotlin that this is a Char data type as opposed to double quotes which indicate a String data type.

12.1.5 String Data Type

The String data type is a sequence of characters that typically make up a word or sentence. In addition to providing a storage mechanism, the String data type also includes a range of string manipulation features allowing strings to be searched, matched, concatenated and modified. Double quotes are used to surround single line strings during assignment, for example:

```
val message = "You have 10 new messages."
```

Alternatively, a multi-line string may be declared using triple quotes

```
val message = """You have 10 new messages,
                5 old messages
                and 6 spam messages."""
```

The leading spaces on each line of a multi-line string can be removed by making a call to the *trimMargin()* function of the String data type:

```
val message = """You have 10 new messages,
                5 old messages
                and 6 spam messages.""".trimMargin()
```

Strings can also be constructed using combinations of strings, variables, constants, expressions, and function calls using a concept referred to as string interpolation. For example, the following code creates a new string from a variety of different sources using string interpolation before outputting it to the console:

```
val username = "John"
val inboxCount = 25
val maxcount = 100
val message = "$username has $inboxCount message. Message capacity remaining is
${maxcount - inboxCount}"

println(message)
```

When executed, the code will output the following message:

```
John has 25 messages. Message capacity remaining is 75 messages.
```

12.1.6 Escape Sequences

In addition to the standard set of characters outlined above, there is also a range of special characters (also referred to as escape characters) available for specifying items such as a new line, tab or a specific Unicode value within a string. These special characters are identified by prefixing the character with a backslash (a concept referred to as escaping). For example, the following assigns a new line to the variable named *newline*:

```
var newline = '\n'
```

In essence, any character that is preceded by a backslash is considered to be a special character and is treated accordingly. This raises the question as to what to do if you actually want a backslash character. This is achieved by escaping the backslash itself:

```
var backslash = '\\'
```

The complete list of special characters supported by Kotlin is as follows:

- \n - New line
- \r - Carriage return
- \t - Horizontal tab
- \\ - Backslash
- \" - Double quote (used when placing a double quote into a string declaration)
- \' - Single quote (used when placing a single quote into a string declaration)
- \\$ - Used when a character sequence containing a \$ is misinterpreted as a variable in a string template.
- \unnnn - Double byte Unicode scalar where nnnn is replaced by four hexadecimal digits representing the

Unicode character.

12.2 Mutable Variables

Variables are essentially locations in computer memory reserved for storing the data used by an application. Each variable is given a name by the programmer and assigned a value. The name assigned to the variable may then be used in the Kotlin code to access the value assigned to that variable. This access can involve either reading the value of the variable or, in the case of *mutable variables*, changing the value.

12.3 Immutable Variables

Often referred to as a *constant*, an immutable variable is similar to a mutable variable in that it provides a named location in memory to store a data value. Immutable variables differ in one significant way in that once a value has been assigned it cannot subsequently be changed.

Immutable variables are particularly useful if there is a value which is used repeatedly throughout the application code. Rather than use the value each time, it makes the code easier to read if the value is first assigned to a constant which is then referenced in the code. For example, it might not be clear to someone reading your Kotlin code why you used the value 5 in an expression. If, instead of the value 5, you use an immutable variable named *interestRate* the purpose of the value becomes much clearer. Immutable values also have the advantage that if the programmer needs to change a widely used value, it only needs to be changed once in the constant declaration and not each time it is referenced.

12.4 Declaring Mutable and Immutable Variables

Mutable variables are declared using the *var* keyword and may be initialized with a value at creation time. For example:

```
var userCount = 10
```

If the variable is declared without an initial value, the type of the variable must also be declared (a topic which will be covered in more detail in the next section of this chapter). The following, for example, is a typical declaration where the variable is initialized after it has been declared:

```
var userCount: Int
userCount = 42
```

Immutable variables are declared using the *val* keyword.

```
val maxUserCount = 20
```

As with mutable variables, the type must also be specified when declaring the variable without initializing it:

```
val maxUserCount: Int
maxUserCount = 20
```

When writing Kotlin code, immutable variables should always be used in preference to mutable variables whenever possible.

12.5 Data Types are Objects

All of the above data types are actually objects, each of which provides a range of functions and properties that may be used to perform a variety of different type specific tasks. These functions and properties are accessed using so-called dot notation. Dot notation involves accessing a function or property of an object by specifying the variable name followed by a dot followed in turn by the name of the property to be accessed or function to be called.

A string variable, for example, can be converted to uppercase via a call to the *toUpperCase()* function of the *String* class:

```
val myString = "The quick brown fox"
val uppercase = myString.toUpperCase()
```

Similarly, the length of a string is available by accessing the length property:

```
val length = myString.length
```

Functions are also available within the String class to perform tasks such as comparisons and checking for the presence of a specific word. The following code, for example, will return a *true* Boolean value since the word “fox” appears within the string assigned to the *myString* variable:

```
val result = myString.contains("fox")
```

All of the number data types include functions for performing tasks such as converting from one data type to another such as converting an Int to a Float:

```
val myInt = 10
val myFloat = myInt.toFloat()
```

A detailed overview of all of the properties and functions provided by the Kotlin data type classes is beyond the scope of this book (there are hundreds). An exhaustive list for all data types can, however, be found within the Kotlin reference documentation available online at:

<https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/>

12.6 Type Annotations and Type Inference

Kotlin is categorized as a statically typed programming language. This essentially means that once the data type of a variable has been identified, that variable cannot subsequently be used to store data of any other type without inducing a compilation error. This contrasts to loosely typed programming languages where a variable, once declared, can subsequently be used to store other data types.

There are two ways in which the type of a variable will be identified. One approach is to use a type annotation at the point the variable is declared in the code. This is achieved by placing a colon after the variable name followed by the type declaration. The following line of code, for example, declares a variable named *userCount* as being of type Int:

```
val userCount: Int = 10
```

In the absence of a type annotation in a declaration, the Kotlin compiler uses a technique referred to as *type inference* to identify the type of the variable. When relying on type inference, the compiler looks to see what type of value is being assigned to the variable at the point that it is initialized and uses that as the type. Consider, for example, the following variable declarations:

```
var signalStrength = 2.231
val companyName = "My Company"
```

During compilation of the above lines of code, Kotlin will infer that the *signalStrength* variable is of type Double (type inference in Kotlin defaults to Double for all floating point numbers) and that the *companyName* constant is of type String.

When a constant is declared without a type annotation it must be assigned a value at the point of declaration:

```
val bookTitle = "Android Studio Development Essentials"
```

If a type annotation is used when the constant is declared, however, the value can be assigned later in the code. For example:

```
val iosBookType = false
val bookTitle: String
```

```

if (iosBookType) {
    bookTitle = "iOS App Development Essentials"
} else {
    bookTitle = "Android Studio Development Essentials"
}

```

12.7 Nullable Type

Kotlin nullable types are a concept that does not exist in most other programming languages (with the exception of the optional type in Swift). The purpose of nullable types is to provide a safe and consistent approach to handling situations where a variable may have a null value assigned to it. In other words, the objective is to avoid the common problem of code crashing with the null pointer exception errors that occur when code encounters a null value where one was not expected.

By default, a variable in Kotlin cannot have a null value assigned to it. Consider, for example, the following code:

```
val username: String = null
```

An attempt to compile the above code will result in a compilation error similar to the following:

```
Error: Null cannot be a value of a non-null string type String
```

If a variable is required to be able to store a null value, it must be specifically declared as a nullable type by placing a question mark (?) after the type declaration:

```
val username: String? = null
```

The *username* variable can now have a null value assigned to it without triggering a compiler error. Once a variable has been declared as nullable, a range of restrictions are then imposed on that variable by the compiler to prevent it being used in situations where it might cause a null pointer exception to occur. A nullable variable, cannot, for example, be assigned to a variable of non-null type as is the case in the following code:

```
val username: String? = null
val firstname: String = username
```

The above code will elicit the following error when encountered by the compiler:

```
Error: Type mismatch: inferred type is String? but String was expected
```

The only way that the assignment will be permitted is if some code is added to check that the value assigned to the nullable variable is non-null:

```

val username: String? = null

if (username != null) {
    val firstname: String = username
}

```

In the above case, the assignment will only take place if the *username* variable references a non-null value.

12.8 The Safe Call Operator

A nullable variable also cannot be used to call a function or to access a property in the usual way. Earlier in this chapter the *toUpperCase()* function was called on a *String* object. Given the possibility that this could cause a function to be called on a null reference, the following code will be disallowed by the compiler:

```

val username: String? = null
val uppercase = username.toUpperCase()

```


The exact error message generated by the compiler in this situation reads as follows:

Error: (Only safe (?.) or non-null asserted (!!.) calls are allowed on a nullable receiver of type String?

In this instance, the compiler is essentially refusing to allow the function call to be made because no attempt has been made to verify that the variable is non-null. One way around this is to add some code to verify that something other than null value has been assigned to the variable prior to making the function call:

```
if (username != null) {
    val uppercase = username.toUpperCase()
}
```

A much more efficient way to achieve this same verification, however, is to call the function using the *safe call operator* (represented by `?.`) as follows:

```
val uppercase = username?.toUpperCase()
```

In the above example, if the `username` variable is null, the `toUpperCase()` function will not be called and execution will proceed at the next line of code. If, on the other hand, a non-null value is assigned the `toUpperCase()` function will be called and the result assigned to the `uppercase` variable.

In addition to function calls, the safe call operator may also be used when accessing properties:

```
val uppercase = username?.length
```

12.9 Not-Null Assertion

The *not-null assertion* removes all of the compiler restrictions from a nullable type, allowing it to be used in the same ways as a non-null type, even if it has been assigned a null value. This assertion is implemented using double exclamation marks after the variable name, for example:

```
val username: String? = null
val length = username!!.length
```

The above code will now compile, but will crash with the following exception at runtime since an attempt is being made to call a function on a non-existent object:

```
Exception in thread "main" kotlin.KotlinNullPointerException
```

Clearly, this causes the very issue that nullable types are designed to avoid. Use of the not-null assertion is generally discouraged and should only be used in situations where you are certain that the value will not be null.

12.10 Nullable Types and the let Function

Earlier in this chapter we looked at how the safe call operator can be used when making a call to a function belonging to a nullable type. This technique makes it easier to check if a value is null without having to write an *if* statement every time the variable is accessed. A similar problem occurs when passing a nullable type as an argument to a function which is expecting a non-null parameter. As an example, consider the `times()` function of the `Int` data type. When called on an `Int` object and passed another integer value as an argument, the function multiplies the two values and returns the result. When the following code is executed, for example, the value of 200 will be displayed within the console:

```
val firstNumber = 10
val secondNumber = 20

val result = firstNumber.times(secondNumber)
print(result)
```

The above example works because the `secondNumber` variable is a non-null type. A problem, however, occurs if

the `secondNumber` variable is declared as being of nullable type:

```
val firstNumber = 10
val secondNumber: Int? = 20

val result = firstNumber.times(secondNumber)
print(result)
```

Now the compilation will fail with the following error message because a nullable type is being passed to a function that is expecting a non-null parameter:

Error: Type mismatch: inferred type is Int? but Int was expected

A possible solution to this problem is to simply write an *if* statement to verify that the value assigned to the variable is non-null before making the call to the function:

```
val firstNumber = 10
val secondNumber: Int? = 20

if (secondNumber != null) {
    val result = firstNumber.times(secondNumber)
    print(result)
}
```

A more convenient approach to addressing the issue, however, involves use of the *let* function. When called on a nullable type object, the *let* function converts the nullable type to a non-null variable named *it* which may then be referenced within a lambda statement.

```
secondNumber?.let {
    val result = firstNumber.times(it)
    print(result)
}
```

Note the use of the safe call operator when calling the *let* function on `secondVariable` in the above example. This ensures that the function is only called when the variable is assigned a non-null value.

12.11 The Elvis Operator

The Kotlin Elvis operator can be used in conjunction with nullable types to define a default value that is to be returned in the event that a value or expression result is null. The Elvis operator (`?:`) is used to separate two expressions. If the expression on the left does not resolve to a null value that value is returned, otherwise the result of the rightmost expression is returned. This can be thought of as a quick alternative to writing an if-else statement to check for a null value. Consider the following code:

```
if (myString != null) {
    return myString
} else {
    return "String is null"
}
```

The same result can be achieved with less coding using the Elvis operator as follows:

```
return myString ?: "String is null"
```

12.12 Type Casting and Type Checking

When compiling Kotlin code, the compiler can typically infer the type of an object. Situations will occur, however, where the compiler is unable to identify the specific type. This is often the case when a value type is ambiguous or an unspecified object is returned from a function call. In this situation it may be necessary to let the compiler know the type of object that your code is expecting or to write code that checks whether the object is of a particular type.

Letting the compiler know the type of object that is expected is known as *type casting* and is achieved within Kotlin code using the *as* cast operator. The following code, for example, lets the compiler know that the result returned from the *getSystemService()* method needs to be treated as a *KeyguardManager* object:

```
val keyMgr = getSystemService(Context.KEYGUARD_SERVICE) as KeyguardManager
```

The Kotlin language includes both safe and unsafe cast operators. The above cast is an unsafe cast and will cause the app to throw an exception if the cast cannot be performed. A safe cast, on the other hand, uses the *as?* operator and returns null if the cast cannot be performed:

```
val keyMgr = getSystemService(Context.KEYGUARD_SERVICE) as? KeyguardManager
```

A type check can be performed to verify that an object conforms to a specific type using the *is* operator, for example:

```
if (keyMgr is KeyguardManager) {
    // It is a KeyguardManager object
}
```

12.13 Summary

This chapter has begun the introduction to Kotlin by exploring data types together with an overview of how to declare variables. The chapter has also introduced concepts such as nullable types, type casting and type checking and the Elvis operator, each of which is an integral part of Kotlin programming and designed specifically to make code writing less prone to error.

13. Kotlin Operators and Expressions

So far we have looked at using variables and constants in Kotlin and also described the different data types. Being able to create variables is only part of the story however. The next step is to learn how to use these variables in Kotlin code. The primary method for working with data is in the form of *expressions*.

13.1 Expression Syntax in Kotlin

The most basic expression consists of an *operator*, two *operands* and an *assignment*. The following is an example of an expression:

```
val myresult = 1 + 2
```

In the above example, the (+) operator is used to add two operands (1 and 2) together. The *assignment operator* (=) subsequently assigns the result of the addition to a variable named *myresult*. The operands could just have easily been variables (or a mixture of values and variables) instead of the actual numerical values used in the example.

In the remainder of this chapter we will look at the basic types of operators available in Kotlin.

13.2 The Basic Assignment Operator

We have already looked at the most basic of assignment operators, the = operator. This assignment operator simply assigns the result of an expression to a variable. In essence, the = assignment operator takes two operands. The left-hand operand is the variable to which a value is to be assigned and the right-hand operand is the value to be assigned. The right-hand operand is, more often than not, an expression which performs some type of arithmetic or logical evaluation or a call to a function, the result of which will be assigned to the variable. The following examples are all valid uses of the assignment operator:

```
var x: Int // Declare a mutable Int variable
val y = 10 // Declare and initialize an immutable Int variable

x = 10 // Assign a value to x
x = x + y // Assign the result of x + y to x
x = y // Assign the value of y to x
```

13.3 Kotlin Arithmetic Operators

Kotlin provides a range of operators for the purpose of creating mathematical expressions. These operators primarily fall into the category of *binary operators* in that they take two operands. The exception is the *unary negative operator* (-) which serves to indicate that a value is negative rather than positive. This contrasts with the *subtraction operator* (-) which takes two operands (i.e. one value to be subtracted from another). For example:

```
var x = -10 // Unary - operator used to assign -10 to variable x
x = x - 5 // Subtraction operator. Subtracts 5 from x
```

The following table lists the primary Kotlin arithmetic operators:

Operator	Description
-(unary)	Negates the value of a variable or expression
*	Multiplication

/	Division
+	Addition
-	Subtraction
%	Remainder/Modulo

Table 13-1

Note that multiple operators may be used in a single expression.

For example:

```
x = y * 10 + z - 5 / 4
```

13.4 Augmented Assignment Operators

In an earlier section we looked at the basic assignment operator (=). Kotlin provides a number of operators designed to combine an assignment with a mathematical or logical operation. These are primarily of use when performing an evaluation where the result is to be stored in one of the operands. For example, one might write an expression as follows:

```
x = x + y
```

The above expression adds the value contained in variable x to the value contained in variable y and stores the result in variable x. This can be simplified using the addition augmented assignment operator:

```
x += y
```

The above expression performs exactly the same task as `x = x + y` but saves the programmer some typing.

Numerous augmented assignment operators are available in Kotlin. The most frequently used of which are outlined in the following table:

Operator	Description
x += y	Add x to y and place result in x
x -= y	Subtract y from x and place result in x
x *= y	Multiply x by y and place result in x
x /= y	Divide x by y and place result in x
x %= y	Perform Modulo on x and y and place result in x

Table 13-2

13.5 Increment and Decrement Operators

Another useful shortcut can be achieved using the Kotlin increment and decrement operators (also referred to as unary operators because they operate on a single operand). Consider the code fragment below:

```
x = x + 1 // Increase value of variable x by 1
x = x - 1 // Decrease value of variable x by 1
```

These expressions increment and decrement the value of x by 1. Instead of using this approach, however, it is quicker to use the ++ and -- operators. The following examples perform exactly the same tasks as the examples above:

```
x++ // Increment x by 1
x-- // Decrement x by 1
```

These operators can be placed either before or after the variable name. If the operator is placed before the

variable name, the increment or decrement operation is performed before any other operations are performed on the variable. For example, in the following code, `x` is incremented before it is assigned to `y`, leaving `y` with a value of 10:

```
var x = 9
val y = ++x
```

In the next example, however, the value of `x` (9) is assigned to variable `y` before the decrement is performed. After the expression is evaluated the value of `y` will be 9 and the value of `x` will be 8.

```
var x = 9
val y = x--
```

13.6 Equality Operators

Kotlin also includes a set of logical operators useful for performing comparisons. These operators all return a Boolean result depending on the result of the comparison. These operators are *binary operators* in that they work with two operands.

Equality operators are most frequently used in constructing program flow control logic. For example an *if* statement may be constructed based on whether one value matches another:

```
if x == y {
    // Perform task
}
```

The result of a comparison may also be stored in a Boolean variable. For example, the following code will result in a *true* value being stored in the variable `result`:

```
var result: Bool
val x = 10
val y = 20
```

```
result = x < y
```

Clearly 10 is less than 20, resulting in a *true* evaluation of the `x < y` expression. The following table lists the full set of Kotlin comparison operators:

Operator	Description
<code>x == y</code>	Returns true if <code>x</code> is equal to <code>y</code>
<code>x > y</code>	Returns true if <code>x</code> is greater than <code>y</code>
<code>x >= y</code>	Returns true if <code>x</code> is greater than or equal to <code>y</code>
<code>x < y</code>	Returns true if <code>x</code> is less than <code>y</code>
<code>x <= y</code>	Returns true if <code>x</code> is less than or equal to <code>y</code>
<code>x != y</code>	Returns true if <code>x</code> is not equal to <code>y</code>

Table 13-3

13.7 Boolean Logical Operators

Kotlin also provides a set of so called logical operators designed to return Boolean *true* or *false* values. These operators both return Boolean results and take Boolean values as operands. The key operators are NOT (!), AND (&&) and OR (||).

The NOT (!) operator simply inverts the current value of a Boolean variable, or the result of an expression. For

Kotlin Operators and Expressions

example, if a variable named *flag* is currently true, prefixing the variable with a '!' character will invert the value to false:

```
val flag = true // variable is true
val secondFlag = !flag // secondFlag set to false
```

The OR (||) operator returns true if one of its two operands evaluates to true, otherwise it returns false. For example, the following code evaluates to true because at least one of the expressions either side of the OR operator is true:

```
if ((10 < 20) || (20 < 10)) {
    print("Expression is true")
}
```

The AND (&&) operator returns true only if both operands evaluate to be true. The following example will return false because only one of the two operand expressions evaluates to true:

```
if ((10 < 20) && (20 < 10)) {
    print("Expression is true")
}
```

13.8 Range Operator

Kotlin includes a useful operator that allows a range of values to be declared. As will be seen in later chapters, this operator is invaluable when working with looping in program logic.

The syntax for the range operator is as follows:

```
x..y
```

This operator represents the range of numbers starting at x and ending at y where both x and y are included within the range (referred to as a closed range). The range operator 5..8, for example, specifies the numbers 5, 6, 7 and 8.

13.9 Bitwise Operators

As previously discussed, computer processors work in binary. These are essentially streams of ones and zeros, each one referred to as a bit. Bits are formed into groups of 8 to form bytes. As such, it is not surprising that we, as programmers, will occasionally end up working at this level in our code. To facilitate this requirement, Kotlin provides a range of *bit operators*.

Those familiar with bitwise operators in other languages such as C, C++, C#, Objective-C and Java will find nothing new in this area of the Kotlin language syntax. For those unfamiliar with binary numbers, now may be a good time to seek out reference materials on the subject in order to understand how ones and zeros are formed into bytes to form numbers. Other authors have done a much better job of describing the subject than we can do within the scope of this book.

For the purposes of this exercise we will be working with the binary representation of two numbers. First, the decimal number 171 is represented in binary as:

```
10101011
```

Second, the number 3 is represented by the following binary sequence:

```
00000011
```

Now that we have two binary numbers with which to work, we can begin to look at the Kotlin bitwise operators:

13.9.1 Bitwise Inversion

The Bitwise inversion (also referred to as NOT) is performed using the *inv()* operation and has the effect of inverting all of the bits in a number. In other words, all the zeros become ones and all the ones become zeros. Taking our example 3 number, a Bitwise NOT operation has the following result:

```
00000011 NOT
=====
11111100
```

The following Kotlin code, therefore, results in a value of -4:

```
val y = 3
val z = y.inv()

print("Result is $z")
```

13.9.2 Bitwise AND

The Bitwise AND is performed using the *and()* operation. It makes a bit by bit comparison of two numbers. Any corresponding position in the binary sequence of each number where both bits are 1 results in a 1 appearing in the same position of the resulting number. If either bit position contains a 0 then a zero appears in the result. Taking our two example numbers, this would appear as follows:

```
10101011 AND
00000011
=====
00000011
```

As we can see, the only locations where both numbers have 1s are the last two positions. If we perform this in Kotlin code, therefore, we should find that the result is 3 (00000011):

```
val x = 171
val y = 3
val z = x.and(y)

print("Result is $z")
```

13.9.3 Bitwise OR

The bitwise OR also performs a bit by bit comparison of two binary sequences. Unlike the AND operation, the OR places a 1 in the result if there is a 1 in the first or second operand. Using our example numbers, the result will be as follows:

```
10101011 OR
00000011
=====
10101011
```

If we perform this operation in Kotlin using the *or()* operation the result will be 171:

```
val x = 171
val y = 3
val z = x.or(y)

print("Result is $z")
```

13.9.4 Bitwise XOR

The bitwise XOR (commonly referred to as *exclusive OR* and performed using the *xor()* operation) performs a similar task to the OR operation except that a 1 is placed in the result if one or other corresponding bit positions in the two numbers is 1. If both positions are a 1 or a 0 then the corresponding bit in the result is set to a 0. For example:

```
10101011 XOR
00000011
=====
10101000
```

The result in this case is 10101000 which converts to 168 in decimal. To verify this we can, once again, try some Kotlin code:

```
val x = 171
val y = 3
val z = x.xor(y)
```

```
print("Result is $z")
```

When executed, we get the following output from print:

```
Result is 168
```

13.9.5 Bitwise Left Shift

The bitwise left shift moves each bit in a binary number a specified number of positions to the left. Shifting an integer one position to the left has the effect of doubling the value.

As the bits are shifted to the left, zeros are placed in the vacated right most (low order) positions. Note also that once the left most (high order) bits are shifted beyond the size of the variable containing the value, those high order bits are discarded:

```
10101011 Left Shift one bit
=====
101010110
```

In Kotlin the bitwise left shift operator is performed using the *shl()* operation, passing through the number of bit positions to be shifted. For example, to shift left by 1 bit:

```
val x = 171
val z = x.shl(1)
```

```
print("Result is $z")
```

When compiled and executed, the above code will display a message stating that the result is 342 which, when converted to binary, equates to 101010110.

13.9.6 Bitwise Right Shift

A bitwise right shift is, as you might expect, the same as a left except that the shift takes place in the opposite direction. Shifting an integer one position to the right has the effect of halving the value.

Note that since we are shifting to the right there is no opportunity to retain the lower most bits regardless of the data type used to contain the result. As a result the low order bits are discarded. Whether or not the vacated high order bit positions are replaced with zeros or ones depends on whether the *sign bit* used to indicate positive and negative numbers is set or not.

```
10101011 Right Shift one bit
=====
01010101
```

The bitwise right shift is performed using the *shr()* operation passing through the shift count:

```
val x = 171
val z = x.shr(1)

print("Result is $z")
```

When executed, the above code will report the result of the shift as being 85, which equates to binary 01010101.

13.10 Summary

Operators and expressions provide the underlying mechanism by which variables and constants are manipulated and evaluated within Kotlin code. This can take the simplest of forms whereby two numbers are added using the addition operator in an expression and the result stored in a variable using the assignment operator. Operators fall into a range of categories, details of which have been covered in this chapter.

14. Kotlin Flow Control

Regardless of the programming language used, application development is largely an exercise in applying logic, and much of the art of programming involves writing code that makes decisions based on one or more criteria. Such decisions define which code gets executed, how many times it is executed and, conversely, which code gets by-passed when the program is executing. This is often referred to as *flow control* since it controls the *flow* of program execution. Flow control typically falls into the categories of *looping control* (how often code is executed) and *conditional flow control* (whether or not code is executed). This chapter is intended to provide an introductory overview of both types of flow control in Kotlin.

14.1 Looping Flow Control

This chapter will begin by looking at flow control in the form of loops. Loops are essentially sequences of Kotlin statements which are to be executed repeatedly until a specified condition is met. The first looping statement we will explore is the *for* loop.

14.1.1 The Kotlin *for-in* Statement

The for-in loop is used to iterate over a sequence of items contained in a collection or number range.

The syntax of the for-in loop is as follows:

```
for variable name in collection or range {  
    // code to be executed  
}
```

In this syntax, *variable name* is the name to be used for a variable that will contain the current item from the collection or range through which the loop is iterating. The code in the body of the loop will typically use this name as a reference to the current item in the loop cycle. The *collection or range* references the item through which the loop is iterating. This could, for example, be an array of string values, a range operator or even a string of characters.

Consider, for example, the following for-in loop construct:

```
for (index in 1..5) {  
    println("Value of index is $index")  
}
```

The loop begins by stating that the current item is to be assigned to a constant named *index*. The statement then declares a closed range operator to indicate that the for loop is to iterate through a range of numbers, starting at 1 and ending at 5. The body of the loop simply prints out a message to the console indicating the current value assigned to the *index* constant, resulting in the following output:

```
Value of index is 1  
Value of index is 2  
Value of index is 3  
Value of index is 4  
Value of index is 5
```

The for-in loop is of particular benefit when working with collections such as arrays. In fact, the for-in loop can be used to iterate through any object that contains more than one item. The following loop, for example, outputs

Kotlin Flow Control

each of the characters in the specified string:

```
for (index in "Hello") {  
    println("Value of index is $index")  
}
```

The operation of a for-in loop may be configured using the *downTo* and *until* functions. The *downTo* function causes the for loop to work backwards through the specified collection until the specified number is reached. The following for loop counts backwards from 100 until the number 90 is reached:

```
for (index in 100 downTo 90) {  
    print("$index.. ")  
}
```

When executed, the above loop will generate the following output:

```
100.. 99.. 98.. 97.. 96.. 95.. 94.. 93.. 92.. 91.. 90..
```

The *until* function operates in much the same way with the exception that counting starts from the bottom of the collection range and works up until (but not including) the specified end point (a concept referred to as a half closed range):

```
for (index in 1 until 10) {  
    print("$index.. ")  
}
```

The output from the above code will range from the start value of 1 through to 9:

```
1.. 2.. 3.. 4.. 5.. 6.. 7.. 8.. 9..
```

The increment used on each iteration through the loop may also be defined using the *step* function as follows:

```
for (index in 0 until 100 step 10) {  
    print("$index.. ")  
}
```

The above code will result in the following console output:

```
0.. 10.. 20.. 30.. 40.. 50.. 60.. 70.. 80.. 90..
```

14.1.2 The *while* Loop

The Kotlin *for* loop described previously works well when it is known in advance how many times a particular task needs to be repeated in a program. There will, however, be instances where code needs to be repeated until a certain condition is met, with no way of knowing in advance how many repetitions are going to be needed to meet that criteria. To address this need, Kotlin includes the *while* loop.

Essentially, the while loop repeats a set of tasks while a specified condition is met. The *while* loop syntax is defined as follows:

```
while condition {  
    // Kotlin statements go here  
}
```

In the above syntax, *condition* is an expression that will return either *true* or *false* and the *// Kotlin statements go here* comment represents the code to be executed while the condition expression is true. For example:

```
var myCount = 0  
  
while (myCount < 100) {
```

```

        myCount++
        println(myCount)
    }

```

In the above example, the *while* expression will evaluate whether the *myCount* variable is less than 100. If it is already greater than 100, the code in the braces is skipped and the loop exits without performing any tasks.

If, on the other hand, *myCount* is not greater than 100 the code in the braces is executed and the loop returns to the *while* statement and repeats the evaluation of *myCount*. This process repeats until the value of *myCount* is greater than 100, at which point the loop exits.

14.1.3 The *do ... while* loop

It is often helpful to think of the *do ... while* loop as an inverted *while* loop. The *while* loop evaluates an expression before executing the code contained in the body of the loop. If the expression evaluates to *false* on the first check then the code is not executed. The *do ... while* loop, on the other hand, is provided for situations where you know that the code contained in the body of the loop will *always* need to be executed at least once. For example, you may want to keep stepping through the items in an array until a specific item is found. You know that you have to at least check the first item in the array to have any hope of finding the entry you need. The syntax for the *do ... while* loop is as follows:

```

do {
    // Kotlin statements here
} while conditional expression

```

In the *do ... while* example below the loop will continue until the value of a variable named *i* equals 0:

```

var i = 10

do {
    i--
    println(i)
} while (i > 0)

```

14.1.4 Breaking from Loops

Having created a loop, it is possible that under certain conditions you might want to break out of the loop before the completion criteria have been met (particularly if you have created an infinite loop). One such example might involve continually checking for activity on a network socket. Once activity has been detected it will most likely be necessary to break out of the monitoring loop and perform some other task.

For the purpose of breaking out of a loop, Kotlin provides the *break* statement which breaks out of the current loop and resumes execution at the code directly after the loop. For example:

```

var j = 10

for (i in 0..100)
{
    j += j

    if (j > 100) {
        break
    }
}

```

Kotlin Flow Control

```
        println("j = $j")
    }
```

In the above example the loop will continue to execute until the value of *j* exceeds 100 at which point the loop will exit and execution will continue with the next line of code after the loop.

14.1.5 The *continue* Statement

The *continue* statement causes all remaining code statements in a loop to be skipped, and execution to be returned to the top of the loop. In the following example, the `println` function is only called when the value of variable *i* is an even number:

```
var i = 1

while (i < 20)
{
    i += 1

    if (i % 2 != 0) {
        continue
    }

    println("i = $i")
}
```

The *continue* statement in the above example will cause the `println` call to be skipped unless the value of *i* can be divided by 2 with no remainder. If the *continue* statement is triggered, execution will skip to the top of the while loop and the statements in the body of the loop will be repeated (until the value of *i* exceeds 19).

14.1.6 Break and Continue Labels

Kotlin expressions may be assigned a label by preceding the expression with a label name followed by the `@` sign. This label may then be referenced when using `break` and `continue` statements to designate where execution is to resume. This is particularly useful when breaking out of nested loops. The following code contains a `for` loop nested within another `for` loop. The inner loop contains a `break` statement which is executed when the value of *j* reaches 10:

```
for (i in 1..100) {

    println("Outer loop i = $i")

    for (j in 1..100) {
        println("Inner loop j = $j")
        if (j == 10) break
    }
}
```

As currently implemented, the `break` statement will exit the inner `for` loop but execution will resume at the top of the outer `for` loop. Suppose, however, that the `break` statement is required to also exit the outer loop. This can be achieved by assigning a label to the outer loop and referencing that label with the `break` statement as follows:

```
outerloop@ for (i in 1..100) {
```



```
println("Outer loop i = $i")

for (j in 1..100) {

    println("Inner loop j = $j")

    if (j == 10) break@outerloop
}
}
```

Now when the value assigned to variable `j` reaches 10 the `break` statement will break out of both loops and resume execution at the line of code immediately following the outer loop.

14.2 Conditional Flow Control

In the previous chapter we looked at how to use logical expressions in Kotlin to determine whether something is *true* or *false*. Since programming is largely an exercise in applying logic, much of the art of programming involves writing code that makes decisions based on one or more criteria. Such decisions define which code gets executed and, conversely, which code gets by-passed when the program is executing.

14.2.1 Using the *if* Expressions

The *if* expression is perhaps the most basic of flow control options available to the Kotlin programmer. Programmers who are familiar with C, Swift, C++ or Java will immediately be comfortable using Kotlin *if* statements, although there are some subtle differences.

The basic syntax of the Kotlin *if* expression is as follows:

```
if (boolean expression) {
    // Kotlin code to be performed when expression evaluates to true
}
```

Unlike some other programming languages, it is important to note that the braces are optional in Kotlin if only one line of code is associated with the *if* expression. In fact, in this scenario, the statement is often placed on the same line as the *if* expression.

Essentially if the *Boolean expression* evaluates to *true* then the code in the body of the statement is executed. If, on the other hand, the expression evaluates to *false* the code in the body of the statement is skipped.

For example, if a decision needs to be made depending on whether one value is greater than another, we would write code similar to the following:

```
val x = 10

if (x > 9) println("x is greater than 9!")
```

Clearly, `x` is indeed greater than 9 causing the message to appear in the console panel.

At this point it is important to notice that we have been referring to the *if* expression instead of the *if* statement. The reason for this is that unlike the *if* statement in other programming languages, the Kotlin *if* returns a result. This allows *if* constructs to be used within expressions. As an example, a typical *if* expression to identify the largest of two numbers and assign the result to a variable might read as follows:

```
if (x > y)
    largest = x
else
```

Kotlin Flow Control

```
largest = y
```

The same result can be achieved using the *if* statement within an expression using the following syntax:

```
variable = if (condition) return_val_1 else return_val_2
```

The original example can, therefore be re-written as follows:

```
val largest = if (x > y) x else y
```

The technique is not limited to returning the values contained within the condition. The following example is also a valid use of *if* in an expression, in this case assigning a string value to the variable:

```
val largest = if (x > y) "x is greatest" else "y is greatest"
println(largest)
```

For those familiar with programming languages such as Java, this feature allows code constructs similar to ternary statements to be implemented in Kotlin.

14.2.2 Using *if... else ...* Expressions

The next variation of the *if* expression allows us to also specify some code to perform if the expression in the *if* expression evaluates to *false*. The syntax for this construct is as follows:

```
if (boolean expression) {
    // Code to be executed if expression is true
} else {
    // Code to be executed if expression is false
}
```

The braces are, once again, optional if only one line of code is to be executed.

Using the above syntax, we can now extend our previous example to display a different message if the comparison expression evaluates to be *false*:

```
val x = 10

if (x > 9) println("x is greater than 9!")
    else println("x is less than 9!")
```

In this case, the second `println` statement will execute if the value of `x` was less than 9.

14.2.3 Using *if ... else if ...* Expressions

So far we have looked at *if* statements which make decisions based on the result of a single logical expression. Sometimes it becomes necessary to make decisions based on a number of different criteria. For this purpose, we can use the *if... else if... else if... else* construct, an example of which is as follows:

```
var x = 9

if (x == 10) println("x is 10")
    else if (x == 9) println("x is 9")
        else if (x == 8) println("x is 8")
            else println("x is less than 8")
}
```

14.2.4 Using the *when* Statement

The Kotlin *when* statement provides a cleaner alternative to the *if ... else if ...* construct and uses the following syntax:

```

when (value) {
    match1 -> // code to be executed on match
    match2 -> // code to be executed on match
    .
    .
    else -> // default code to executed if no match
}

```

Using this syntax, the previous *if... else if...* construct can be rewritten to use the *when* statement:

```

when (x) {
    10 -> println ("x is 10")
    9 -> println("x is 9")
    8 -> println("x is 8")
    else ->  println("x is less than 8")
}

```

The *when* statement is similar to the *switch* statement found in many other programming languages.

14.3 Summary

The term *flow control* is used to describe the logic that dictates the execution path that is taken through the source code of an application as it runs. This chapter has looked at the two types of flow control provided by Kotlin (looping and conditional) and explored the various Kotlin constructs that are available to implement both forms of flow control logic.

15. An Overview of Kotlin Functions and Lambdas

Kotlin functions and lambdas are a vital part of writing well-structured and efficient code and provide a way to organize programs while avoiding code repetition. In this chapter we will look at how functions and lambdas are declared and used within Kotlin.

15.1 What is a Function?

A function is a named block of code that can be called upon to perform a specific task. It can be provided data on which to perform the task and is capable of returning results to the code that called it. For example, if a particular arithmetic calculation needs to be performed in a Kotlin program, the code to perform the arithmetic can be placed in a function. The function can be programmed to accept the values on which the arithmetic is to be performed (referred to as parameters) and to return the result of the calculation. At any point in the program code where the calculation is required the function is simply called, parameter values passed through as arguments and the result returned.

The terms parameter and argument are often used interchangeably when discussing functions. There is, however, a subtle difference. The values that a function is able to accept when it is called are referred to as parameters. At the point that the function is actually called and passed those values, however, they are referred to as arguments.

15.2 How to Declare a Kotlin Function

A Kotlin function is declared using the following syntax:

```
fun <function name> (<para name>: <para type>, <para name>: <para type>, ... ):  
<return type> {  
    // Function code  
}
```

This combination of function name, parameters and return type are referred to as the function *signature* or *type*. Explanations of the various fields of the function declaration are as follows:

- `fun` – The prefix keyword used to notify the Kotlin compiler that this is a function.
- `<function name>` - The name assigned to the function. This is the name by which the function will be referenced when it is called from within the application code.
- `<para name>` - The name by which the parameter is to be referenced in the function code.
- `<para type>` - The type of the corresponding parameter.
- `<return type>` - The data type of the result returned by the function. If the function does not return a result then no return type is specified.
- Function code - The code of the function that does the work.

As an example, the following function takes no parameters, returns no result and simply displays a message:

```
fun sayHello() {
```

An Overview of Kotlin Functions and Lambdas

```
println("Hello")
}
```

The following sample function, on the other hand, takes an integer and a string as parameters and returns a string result:

```
fun buildMessageFor(name: String, count: Int): String {
    return("$name, you are customer number $count")
}
```

15.3 Calling a Kotlin Function

Once declared, functions are called using the following syntax:

```
<function name> (<arg1>, <arg2>, ... )
```

Each argument passed through to a function must match the parameters the function is configured to accept. For example, to call a function named `sayHello` that takes no parameters and returns no value, we would write the following code:

```
sayHello()
```

In the case of a message that accepts parameters, the function could be called as follows:

```
buildMessageFor("John", 10)
```

15.4 Single Expression Functions

When a function contains a single expression, it is not necessary to include the braces around the expression. All that is required is an equals sign (=) after the function declaration followed by the expression. The following function contains a single expression declared in the usual way:

```
fun multiply(x: Int, y: Int): Int {
    return x * y
}
```

Below is the same function expressed as a single line expression:

```
fun multiply(x: Int, y: Int): Int = x * y
```

When using single line expressions, the return type may be omitted in situations where the compiler is able to infer the type returned by the expression making for even more compact code:

```
fun multiply(x: Int, y: Int) = x * y
```

15.5 Local Functions

A local function is a function that is embedded within another function. In addition, a local function has access to all of the variables contained within the enclosing function:

```
fun main(args: Array<String>) {

    val name = "John"
    val count = 5

    fun displayString() {
        for (index in 0..count) {
            println(name)
        }
    }
}
```

```
        displayString()
    }
}
```

15.6 Handling Return Values

To call a function named `buildMessage` that takes two parameters and returns a result, on the other hand, we might write the following code:

```
val message = buildMessageFor("John", 10)
```

To improve code readability, the parameter names may also be specified when making the function call:

```
val message = buildMessageFor(name = "John", count = 10)
```

In the above examples, we have created a new variable called `message` and then used the assignment operator (`=`) to store the result returned by the function.

15.7 Declaring Default Function Parameters

Kotlin provides the ability to designate a default parameter value to be used in the event that the value is not provided as an argument when the function is called. This simply involves assigning the default value to the parameter when the function is declared.

To see default parameters in action the `buildMessageFor` function will be modified so that the string “Customer” is used as a default in the event that a customer name is not passed through as an argument. Similarly, the `count` parameter is declared with a default value of 0:

```
fun buildMessageFor(name: String = "Customer", count: Int = 0): String {
    return("$name, you are customer number $count")
}
```

When parameter names are used when making the function call, any parameters for which defaults have been specified may be omitted. The following function call, for example, omits the customer name argument but still compiles because the parameter name has been specified for the second argument:

```
val message = buildMessageFor(count = 10)
```

If parameter names are not used within the function call, however, only the trailing arguments may be omitted:

```
val message = buildMessageFor("John") // Valid
val message = buildMessageFor(10) // Invalid
```

15.8 Variable Number of Function Parameters

It is not always possible to know in advance the number of parameters a function will need to accept when it is called within application code. Kotlin handles this possibility through the use of the `vararg` keyword to indicate that the function accepts an arbitrary number of parameters of a specified data type. Within the body of the function, the parameters are made available in the form of an array object. The following function, for example, takes as parameters a variable number of `String` values and then outputs them to the console panel:

```
fun displayStrings(vararg strings: String)
{
    for (string in strings) {
        println(string)
    }
}
```

```
displayStrings("one", "two", "three", "four")
```

An Overview of Kotlin Functions and Lambdas

Kotlin does not permit multiple vararg parameters within a function and any single parameters supported by the function must be declared before the vararg declaration:

```
fun displayStrings(name: String, vararg strings: String)
{
    for (string in strings) {
        println(string)
    }
}
```

15.9 Lambda Expressions

Having covered the basics of functions in Kotlin it is now time to look at the concept of lambda expressions. Essentially, lambdas are self-contained blocks of code. The following code, for example, declares a lambda, assigns it to a variable named `sayHello` and then calls the function via the lambda reference:

```
val sayHello = { println("Hello") }
sayHello()
```

Lambda expressions may also be configured to accept parameters and return results. The syntax for this is as follows:

```
{<para name>: <para type>, <para name> <para type>, ... ->
    // Lambda expression here
}
```

The following lambda expression, for example, accepts two integer parameters and returns an integer result:

```
val multiply = { val1: Int, val2: Int -> val1 * val2 }
val result = multiply(10, 20)
```

Note that the above lambda examples have assigned the lambda code block to a variable. This is also possible when working with functions. Of course, the following syntax will execute the function and assign the result of that execution to a variable, instead of assigning the function itself to the variable:

```
val myvar = myfunction()
```

To assign a function reference to a variable, simply remove the parentheses and prefix the function name with double colons (`::`) as follows. The function may then be called simply by referencing the variable name:

```
val mavar = ::myfunction
myvar()
```

A lambda block may be executed directly by placing parentheses at the end of the expression including any arguments. The following lambda directly executes the multiplication lambda expression multiplying 10 by 20.

```
val result = { val1: Int, val2: Int -> val1 * val2 }(10, 20)
```

The last expression within a lambda serves as the expressions return value (hence the value of 200 being assigned to the result variable in the above multiplication examples). In fact, unlike functions, lambdas do not support the *return* statement. In the absence of an expression that returns a result (such as an arithmetic or comparison expression), simply declaring the value as the last item in the lambda will cause that value to be returned. The following lambda returns the Boolean true value after printing a message:

```
val result = { println("Hello"); true }()
```

Similarly, the following lambda simply returns a string literal:

```
val nextmessage = { println("Hello"); "Goodbye" }()
```


A particularly useful feature of lambdas and the ability to create function references is that they can be both passed to functions as arguments and returned as results. This concept, however, requires an understanding of function types and higher-order functions.

15.10 Higher-order Functions

On the surface, lambdas and function references do not seem to be particularly compelling features. The possibilities that these features offer become more apparent, however, when we consider that lambdas and function references have the same capabilities of many other data types. In particular, these may be passed through as arguments to another function, or even returned as a result from a function.

A function that is capable of receiving a function or lambda as an argument, or returning one as a result is referred to as a *higher-order function*.

Before we look at what is, essentially, the ability to plug one function into another, it is first necessary to explore the concept of *function types*. The type of a function is dictated by a combination of the parameters it accepts and the type of result it returns. A function which accepts an Int and a Double as parameters and returns a String result for example is considered to have the following function type:

```
(Int, Double) -> String
```

In order to accept a function as a parameter, the receiving function simply declares the type of the function it is able to accept.

For the purposes of an example, we will begin by declaring two unit conversion functions:

```
fun inchesToFeet (inches: Double): Double {
    return inches * 0.0833333
}

fun inchesToYards (inches: Double): Double {
    return inches * 0.0277778
}
```

The example now needs an additional function, the purpose of which is to perform a unit conversion and print the result in the console panel. This function needs to be as general purpose as possible, capable of performing a variety of different measurement unit conversions. In order to demonstrate functions as parameters, this new function will take as a parameter a function type that matches both the inchesToFeet and inchesToYards functions together with a value to be converted. Since the type of these functions is equivalent to (Double) -> Double, our general purpose function can be written as follows:

```
fun outputConversion(converterFunc: (Double) -> Double, value: Double) {
    val result = converterFunc(value)
    println("Result of conversion is $result")
}
```

When the outputConversion function is called, it will need to be passed a function matching the declared type. That function will be called to perform the conversion and the result displayed in the console panel. This means that the same function can be called to convert inches to both feet and yards, simply by “plugging in” the appropriate converter function as a parameter, keeping in mind that it is the function reference that is being passed as an argument:

```
outputConversion(::inchesToFeet, 22.45)
outputConversion(::inchesToYards, 22.45)
```

Functions can also be returned as a data type simply by declaring the type of the function as the return type.

An Overview of Kotlin Functions and Lambdas

The following function is configured to return either our `inchesToFeet` or `inchesToYards` function type (in other words a function which accepts and returns a `Double` value) based on the value of a `Boolean` parameter:

```
fun decideFunction(feet: Boolean): (Double) -> Double
{
    if (feet) {
        return ::inchesToFeet
    } else {
        return ::inchesToYards
    }
}
```

When called, the function will return a function reference which can then be used to perform the conversion:

```
val converter = decideFunction(true)
val result = converter(22.4)
println(result)
```

15.11 Summary

Functions and lambda expressions are self-contained blocks of code that can be called upon to perform a specific task and provide a mechanism for structuring code and promoting reuse. This chapter has introduced the basic concepts of function and lambda declaration and implementation in addition to the use of higher-order functions that allow lambdas and functions to be passed as arguments and returned as results.

16. The Basics of Object Oriented Programming in Kotlin

Kotlin provides extensive support for developing object-oriented applications. The subject area of object oriented programming is, however, large. As such, a detailed overview of object oriented software development is beyond the scope of this book. Instead, we will introduce the basic concepts involved in object oriented programming and then move on to explaining the concept as it relates to Kotlin application development.

16.1 What is an Object?

Objects (also referred to as instances) are self-contained modules of functionality that can be easily used, and re-used as the building blocks for a software application.

Objects consist of data variables (called properties) and functions (called methods) that can be accessed and called on the object or instance to perform tasks and are collectively referred to as class members.

16.2 What is a Class?

Much as a blueprint or architect's drawing defines what an item or a building will look like once it has been constructed, a class defines what an object will look like when it is created. It defines, for example, what the methods will do and what the properties will be.

16.3 Declaring a Kotlin Class

Before an object can be instantiated, we first need to define the class 'blueprint' for the object. In this chapter we will create a bank account class to demonstrate the basic concepts of Kotlin object oriented programming.

In declaring a new Kotlin class we specify an optional parent class from which the new class is derived and also define the properties and methods that the class will contain. The basic syntax for a new class is as follows:

```
class NewClassName: ParentClass {  
    // Properties  
    // Methods  
}
```

The Properties section of the declaration defines the variables and constants that are to be contained within the class. These are declared in the same way that any other variable would be declared in Kotlin.

The Methods sections define the methods that are available to be called on the class and instances of the class. These are essentially functions specific to the class that perform a particular operation when called upon and will be described in greater detail later in this chapter.

To create an example outline for our BankAccount class, we would use the following:

```
class BankAccount {  
  
}
```

Now that we have the outline syntax for our class, the next step is to add some properties to it.

16.4 Adding Properties to a Class

A key goal of object oriented programming is a concept referred to as data encapsulation. The idea behind data encapsulation is that data should be stored within classes and accessed only through methods defined in that class. Data encapsulated in a class are referred to as properties or instance variables.

Instances of our BankAccount class will be required to store some data, specifically a bank account number and the balance currently held within the account. Properties are declared in the same way any other variables are declared in Kotlin. We can, therefore, add these variables as follows:

```
class BankAccount {  
    var accountBalance: Double = 0.0  
    var accountNumber: Int = 0  
}
```

Having defined our properties, we can now move on to defining the methods of the class that will allow us to work with our properties while staying true to the data encapsulation model.

16.5 Defining Methods

The methods of a class are essentially code routines that can be called upon to perform specific tasks within the context of that class.

Methods are declared within the opening and closing braces of the class to which they belong and are declared using the standard Kotlin function declaration syntax.

For example, the declaration of a method to display the account balance in our example might read as follows:

```
class BankAccount {  
    var accountBalance: Double = 0.0  
    var accountNumber: Int = 0  
  
    fun displayBalance()  
    {  
        println("Number $accountNumber")  
        println("Current balance is $accountBalance")  
    }  
}
```

16.6 Declaring and Initializing a Class Instance

So far all we have done is define the blueprint for our class. In order to do anything with this class, we need to create instances of it. The first step in this process is to declare a variable to store a reference to the instance when it is created. We do this as follows:

```
val account1: BankAccount = BankAccount()
```

When executed, an instance of our BankAccount class will have been created and will be accessible via the account1 variable. Of course, the Kotlin compiler will be able to use inference here, making the type declaration optional:

```
val account1 = BankAccount()
```

16.7 Primary and Secondary Constructors

A class will often need to perform some initialization tasks at the point of creation. These tasks can be implemented using constructors within the class. In the case of the BankAccount class, it would be useful to be

able to initialize the account number and balance properties with values when a new class instance is created. To achieve this, a *secondary constructor* can be declared within the class header as follows:

```
class BankAccount {

    var accountBalance: Double = 0.0
    var accountNumber: Int = 0

    constructor(number: Int, balance: Double) {
        accountNumber = number
        accountBalance = balance
    }

    .
    .
}
```

When creating an instance of the class, it will now be necessary to provide initialization values for the account number and balance properties as follows:

```
val account1: BankAccount = BankAccount(456456234, 342.98)
```

A class can contain multiple secondary constructors allowing instances of the class to be initiated with different value sets. The following variation of the BankAccount class includes an additional secondary constructor for use when initializing an instance with the customer's last name in addition to the corresponding account number and balance:

```
class BankAccount {

    var accountBalance: Double = 0.0
    var accountNumber: Int = 0
    var lastName: String = ""

    constructor(number: Int,
                balance: Double) {
        accountNumber = number
        accountBalance = balance
    }

    constructor(number: Int,
                balance: Double,
                name: String ) {
        accountNumber = number
        accountBalance = balance
        lastName = name
    }

    .
    .
}
```

Instances of the BankAccount may now also be created as follows:

The Basics of Object Oriented Programming in Kotlin

```
val account1: BankAccount = BankAccount(456456234, 342.98, "Smith")
```

It is also possible to use a *primary constructor* to perform basic initialization tasks. The primary constructor for a class is declared within the class header as follows:

```
class BankAccount (val accountNumber: Int, var accountBalance: Double) {  
    .  
    .  
    fun displayBalance()  
    {  
        println("Number $accountNumber")  
        println("Current balance is $accountBalance")  
    }  
}
```

Note that now both properties have been declared in the primary constructor, it is no longer necessary to also declare the variables within the body of the class. Since the account number will now not change after an instance of the class has been created, this property is declared as being immutable using the *val* keyword.

Although a class may only contain one primary constructor, Kotlin allows multiple secondary constructors to be declared in addition to the primary constructor. In the following class declaration the constructor that handles the account number and balance is declared as the primary constructor while the variation that also accepts the user's last name is declared as a secondary constructor:

```
class BankAccount (val accountNumber: Int, var accountBalance: Double) {  
  
    var lastName: String = ""  
  
    constructor(accountNumber: Int,  
                accountBalance: Double,  
                name: String ) : this(accountNumber, accountBalance) {  
  
        lastName = name  
    }  
    .  
    .  
}
```

In the above example there are two key points which need to be noted. First, since the `lastName` property is referenced by a secondary constructor, the variable is not handled automatically by the primary constructor and must be declared within the body of the class and initialized within the constructor.

```
var lastName: String = ""  
.  
.  
lastName = name
```

Second, although the `accountNumber` and `accountBalance` properties are accepted as parameters to the secondary constructor, the variable declarations are still handled by the primary constructor and do not need to be declared. To associate the references to these properties in the secondary constructor with the primary constructor, however, they must be linked back to the primary constructor using the *this* keyword:

```
... this(accountNumber, accountBalance)...
```

16.8 Initializer Blocks

In addition to the primary and secondary constructors, a class may also contain *initializer blocks* which are called after the constructors. Since a primary constructor cannot contain any code, these methods are a particularly useful location for adding code to perform initialization tasks when an instance of the class is created. Initializer blocks are declared using the *init* keyword with the initialization code enclosed in braces:

```
class BankAccount (val accountNumber: Int, var accountBalance: Double) {

    init {
        // Initialization code goes here
    }

    .
    .
}
```

16.9 Calling Methods and Accessing Properties

Now is probably a good time to recap what we have done so far in this chapter. We have now created a new Kotlin class named `BankAccount`. Within this new class we declared primary and secondary constructors to accept and initialize account number, balance and customer name properties. In the preceding sections we also covered the steps necessary to create and initialize an instance of our new class. The next step is to learn how to call the instance methods and access the properties we built into our class. This is most easily achieved using dot notation.

Dot notation involves accessing a property, or calling a method by specifying a class instance followed by a dot followed in turn by the name of the property or method:

```
classInstance.propertyname
classInstance.methodname()
```

For example, to get the current value of our `accountBalance` instance variable:

```
val balance1 = account1.accountBalance
```

Dot notation can also be used to set values of instance properties:

```
account1.accountBalance = 6789.98
```

The same technique is used to call methods on a class instance. For example, to call the `displayBalance` method on an instance of the `BankAccount` class:

```
account1.displayBalance()
```

16.10 Custom Accessors

When accessing the `accountBalance` property in the previous section, the code is making use of property accessors that are provided automatically by Kotlin. In addition to these default accessors it is also possible to implement *custom accessors* that allow calculations or other logic to be performed before the property is returned or set.

Custom accessors are implemented by creating getter and optional corresponding setter methods containing the code to perform any tasks before returning the property. Consider, for example, that the `BankAccount` class might need an additional property to contain the current balance less any recent banking fees. Rather than use a standard accessor, it makes more sense to use a custom accessor which calculates this value on request. The modified `BankAccount` class might now read as follows:

```
class BankAccount (val accountNumber: Int, var accountBalance: Double) {
```

```
val fees: Double = 25.00

val balanceLessFees: Double
    get() {
        return accountBalance - fees
    }

fun displayBalance()
{
    println("Number $accountNumber")
    println("Current balance is $accountBalance")
}
}
```

The above code adds a getter that returns a computed property based on the current balance minus a fee amount. An optional setter could also be declared in much the same way to set the balance value less fees:

```
val fees: Double = 25.00

var balanceLessFees: Double
    get() {
        return accountBalance - fees
    }
    set(value) {
        accountBalance = value - fees
    }
.
.
}
```

The new setter takes as a parameter a Double value from which it deducts the fee value before assigning the result to the current balance property. Regardless of the fact that these are custom accessors, they are accessed in the same way as stored properties using dot-notation. The following code gets the current balance less the fees value before setting the property to a new value:

```
val balance1 = account1.balanceLessFees
account1.balanceLessFees = 12123.12
```

16.11 Nested and Inner Classes

Kotlin allows one class to be nested within another class. In the following code, for example, ClassB is nested inside ClassA:

```
class ClassA {
    class ClassB {
    }
}
```

In the above example, ClassB does not have access to any of the properties within the outer class. If access is required, the nested class must be declared using the *inner* directive. In the example below ClassB now has access to the myProperty variable belonging to ClassA:


```

class ClassA {
    var myProperty: Int = 10

    inner class ClassB {
        val result = 20 + myProperty
    }
}

```

16.12 Companion Objects

A Kotlin class can also contain a companion object. A companion object contains methods and variables that are common to all instances of the class. In addition to being accessible via class instances, these properties are also accessible at the class level (in other words without the need to create an instance of the class).

The syntax for declaring a companion object within a class is as follows:

```

class ClassName: ParentClass {
    // Properties
    // Methods

    companion object {
        // properties
        // methods
    }
}

```

To experience a simple companion object example in action, enter the following into the Kotlin online playground at <https://try.kotl.in>:

```

class MyClass {

    fun showCount() {
        println("counter = " + counter)
    }

    companion object {
        var counter = 1

        fun counterUp() {
            counter += 1
        }
    }
}

fun main(args: Array<String>) {
    println(MyClass.counter)
}

```

The class contains a companion object consisting of a counter variable and a method to increment that variable. The class also contains a method to display the current counter value. The *main()* method simply displays the

The Basics of Object Oriented Programming in Kotlin

current value of the counter variable, but does so by calling the method on the class itself instead of a class instance:

```
println(MyClass.counter)
```

Modify the *main()* method to also increment the counter, displaying the current value both before and after:

```
fun main(args: Array<String>) {  
    println(MyClass.counter)  
    MyClass.counterUp()  
    println(MyClass.counter)  
}
```

Run the code and verify that the following output appears in the console:

```
1  
2
```

Next, add some code to create an instance of *MyClass* before making a call to the *showCount()* method:

```
fun main(args: Array<String>) {  
    println(MyClass.counter)  
    MyClass.counterUp()  
    println(MyClass.counter)  
  
    val instanceA = MyClass()  
    instanceA.showCount()  
}
```

When executed, the following output will appear in the console:

```
1  
2  
counter = 2
```

Clearly, the class has access to the variables and methods contained within the companion object.

Another useful aspect of companion objects is that all instances of the containing class see the same companion object, including current variable values. To see this in action, create a second instance of *MyClass* and call the *showCount()* method on that instance:

```
fun main(args: Array<String>) {  
    println(MyClass.counter)  
    MyClass.counterUp()  
    println(MyClass.counter)  
  
    val instanceA = MyClass()  
    instanceA.showCount()  
  
    val instanceB = MyClass()  
    instanceB.showCount()  
}
```

When run, the code will produce the following console output:

```
1
```

```
2
```

```
counter = 2
```

```
counter = 2
```

Note that both instances return the incremented value of 2, showing that the two class instances are sharing the same companion object data.

16.13 Summary

Object oriented programming languages such as Kotlin encourage the creation of classes to promote code reuse and the encapsulation of data within class instances. This chapter has covered the basic concepts of classes and instances within Kotlin together with an overview of primary and secondary constructors, initializer blocks, properties, methods, companion objects and custom accessors.

17. An Introduction to Kotlin Inheritance and Subclassing

In “*The Basics of Object Oriented Programming in Kotlin*” we covered the basic concepts of object-oriented programming and worked through an example of creating and working with a new class using Kotlin. In that example, our new class was not specifically derived from a base class (though in practice, all Kotlin classes are ultimately derived from the *Any* class). In this chapter we will provide an introduction to the concepts of subclassing, inheritance and extensions in Kotlin.

17.1 Inheritance, Classes and Subclasses

The concept of inheritance brings something of a real-world view to programming. It allows a class to be defined that has a certain set of characteristics (such as methods and properties) and then other classes to be created which are derived from that class. The derived class inherits all of the features of the parent class and typically then adds some features of its own. In fact, all classes in Kotlin are ultimately subclasses of the *Any* superclass which provides the basic foundation on which all classes are based.

By deriving classes we create what is often referred to as a class hierarchy. The class at the top of the hierarchy is known as the base class or root class and the derived classes as subclasses or child classes. Any number of subclasses may be derived from a class. The class from which a subclass is derived is called the parent class or superclass.

Classes need not only be derived from a root class. For example, a subclass can also inherit from another subclass with the potential to create large and complex class hierarchies.

In Kotlin a subclass can only be derived from a single direct parent class. This is a concept referred to as single inheritance.

17.2 Subclassing Syntax

As a safety measure designed to make Kotlin code less prone to error, before a subclass can be derived from a parent class, the parent class must be declared as open. This is achieved by placing the *open* keyword within the class header:

```
open class MyParentClass {  
    var myProperty: Int = 0  
}
```

With a simple class of this type, the subclass can be created as follows:

```
class MySubClass : MyParentClass() {  
  
}
```

For classes containing primary or secondary constructors, the rules for creating a subclass are slightly more complicated. Consider the following parent class which contains a primary constructor:

```
open class MyParentClass(var myProperty: Int) {
```

```
}
```

In order to create a subclass of this class, the subclass declaration references any base class parameters while also initializing the parent class using the following syntax:

```
class MySubClass(myProperty: Int) : MyParentClass(myProperty) {  
  
}
```

If, on the other hand, the parent class contains one or more secondary constructors, the constructors must also be implemented within the subclass declaration and include a call to the secondary constructors of the parent class, passing through as arguments the values passed to the subclass secondary constructor. When working with subclasses, the parent class can be referenced using the *super* keyword. A parent class with a secondary constructor might read as follows:

```
open class MyParentClass {  
    var myProperty: Int = 0  
  
    constructor(number: Int) {  
        myProperty = number  
    }  
}
```

The code for the corresponding subclass would need to be implemented as follows:

```
class MySubClass : MyParentClass {  
    constructor(number: Int) : super(number)  
}
```

If additional tasks need to be performed within the constructor of the subclass, this can be placed within curly braces after the constructor declaration:

```
class MySubClass : MyParentClass {  
  
    constructor(number: Int) : super(number) {  
        // Subclass constructor code here  
    }  
}
```

17.3 A Kotlin Inheritance Example

As with most programming concepts, the subject of inheritance in Kotlin is perhaps best illustrated with an example. In *“The Basics of Object Oriented Programming in Kotlin”* we created a class named `BankAccount` designed to hold a bank account number and corresponding current balance. The `BankAccount` class contained both properties and methods. A simplified declaration for this class is reproduced below and will be used for the basis of the subclassing example in this chapter:

```
class BankAccount {  
  
    var accountNumber = 0  
    var accountBalance = 0.0  
  
    constructor(number: Int, balance: Double) {  
        accountNumber = number  
    }  
}
```

```

        accountBalance = balance
    }

    open fun displayBalance()
    {
        println("Number $accountNumber")
        println("Current balance is $accountBalance")
    }
}

```

Though this is a somewhat rudimentary class, it does everything necessary if all you need it to do is store an account number and account balance. Suppose, however, that in addition to the `BankAccount` class you also needed a class to be used for savings accounts. A savings account will still need to hold an account number and a current balance and methods will still be needed to access that data. One option would be to create an entirely new class, one that duplicates all of the functionality of the `BankAccount` class together with the new features required by a savings account. A more efficient approach, however, would be to create a new class that is a subclass of the `BankAccount` class. The new class will then inherit all the features of the `BankAccount` class but can then be extended to add the additional functionality required by a savings account. Before a subclass of the `BankAccount` class can be created, the declaration needs to be modified to declare the class as open:

```
open class BankAccount {
```

To create a subclass of `BankAccount` that we will call `SavingsAccount`, we simply declare the new class, this time specifying `BankAccount` as the parent class and add code to call the constructor on the parent class:

```
class SavingsAccount : BankAccount {
    constructor(accountNumber: Int, accountBalance: Double) :
        super(accountNumber, accountBalance)
}
```

Note that although we have yet to add any properties or methods, the class has actually inherited all the methods and properties of the parent `BankAccount` class. We could, therefore, create an instance of the `SavingsAccount` class and set variables and call methods in exactly the same way we did with the `BankAccount` class in previous examples. That said, we haven't really achieved anything unless we actually take steps to extend the class.

17.4 Extending the Functionality of a Subclass

So far we have been able to create a subclass that contains all the functionality of the parent class. In order for this exercise to make sense, however, we now need to extend the subclass so that it has the features we need to make it useful for storing savings account information. To do this, we simply add the properties and methods that provide the new functionality, just as we would for any other class we might wish to create:

```
class SavingsAccount : BankAccount {
    var interestRate: Double = 0.0

    constructor(accountNumber: Int, accountBalance: Double) :
        super(accountNumber, accountBalance)

    fun calculateInterest(): Double
    {
        return interestRate * accountBalance
    }
}
```

```
}
```

17.5 Overriding Inherited Methods

When using inheritance it is not unusual to find a method in the parent class that almost does what you need, but requires modification to provide the precise functionality you require. That being said, it is also possible you'll inherit a method with a name that describes exactly what you want to do, but it actually does not come close to doing what you need. One option in this scenario would be to ignore the inherited method and write a new method with an entirely new name. A better option is to override the inherited method and write a new version of it in the subclass.

Before proceeding with an example, there are three rules that must be obeyed when overriding a method. First, the overriding method in the subclass must take exactly the same number and type of parameters as the overridden method in the parent class. Second, the new method must have the same return type as the parent method. Finally, the original method in the parent class must be declared as open before the compiler will allow it to be overridden.

In our `BankAccount` class we have a method named `displayBalance` that displays the bank account number and current balance held by an instance of the class. In our `SavingsAccount` subclass we might also want to output the current interest rate assigned to the account. To achieve this, we simply declare a new version of the `displayBalance` method in our `SavingsAccount` subclass, prefixed with the *override* keyword:

```
class SavingsAccount : BankAccount {
    var interestRate: Double = 0.0

    constructor(accountNumber: Int, accountBalance: Double) :
        super(accountNumber, accountBalance)

    fun calculateInterest(): Double
    {
        return interestRate * accountBalance
    }

    override fun displayBalance()
    {
        println("Number $accountNumber")
        println("Current balance is $accountBalance")
        println("Prevailing interest rate is $interestRate")
    }
}
```

Before this code will compile, the `displayBalance` method in the `BankAccount` class must be declared as open:

```
open fun displayBalance()
{
    println("Number $accountNumber")
    println("Current balance is $accountBalance")
}
```

It is also possible to make a call to the overridden method in the super class from within a subclass. The `displayBalance` method of the super class could, for example, be called to display the account number and balance, before the interest rate is displayed, thereby eliminating further code duplication:


```

override fun displayBalance()
{
    super.displayBalance()
    println("Prevailing interest rate is $interestRate")
}

```

17.6 Adding a Custom Secondary Constructor

As the SavingsAccount class currently stands, it makes a call to the secondary constructor from the parent BankAccount class which was implemented as follows:

```

constructor(accountNumber: Int, accountBalance: Double) :
    super(accountNumber, accountBalance)

```

Clearly this constructor takes the necessary steps to initialize both the account number and balance properties of the class. The SavingsAccount class, however, contains an additional property in the form of the interest rate variable. The SavingsAccount class, therefore, needs its own constructor to ensure that the interestRate property is initialized when instances of the class are created. Modify the SavingsAccount class one last time to add an additional secondary constructor allowing the interest rate to also be specified when class instances are initialized:

```

class SavingsAccount : BankAccount {

    var interestRate: Double = 0.0

    constructor(accountNumber: Int, accountBalance: Double) :
        super(accountNumber, accountBalance)

    constructor(accountNumber: Int, accountBalance: Double, rate: Double) :
        super(accountNumber, accountBalance) {
            interestRate = rate
        }
}

```

17.7 Using the SavingsAccount Class

Now that we have completed work on our SavingsAccount class, the class can be used in some example code in much the same way as the parent BankAccount class:

```

val savings1 = SavingsAccount(12311, 600.00, 0.07)

println(savings1.calculateInterest())
savings1.displayBalance()

```

17.8 Summary

Inheritance extends the concept of object re-use in object oriented programming by allowing new classes to be derived from existing classes, with those new classes subsequently extended to add new functionality. When an existing class provides some, but not all, of the functionality required by the programmer, inheritance allows that class to be used as the basis for a new subclass. The new subclass will inherit all the capabilities of the parent class, but may then be extended to add the missing functionality.

18. 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.

18.1 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.

18.2 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 18-1, a process can be in one of the following five states at any given time:

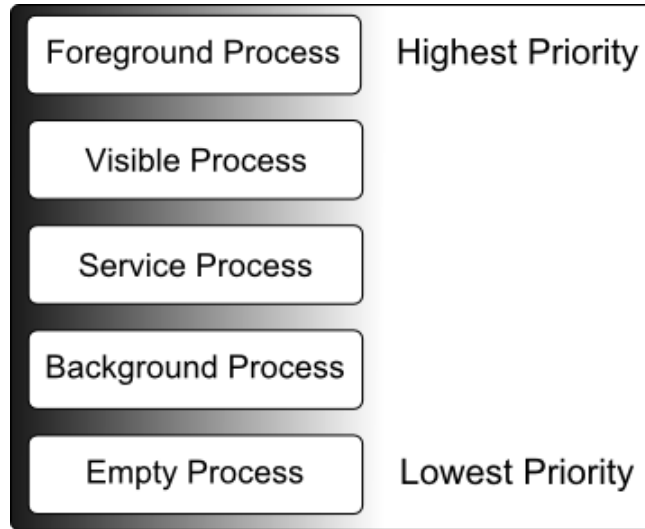


Figure 18-1

18.2.1 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.

18.2.2 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.

18.2.3 Service Process

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

18.2.4 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.

18.2.5 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.

18.3 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.

18.4 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*.

18.5 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 18-2.

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.

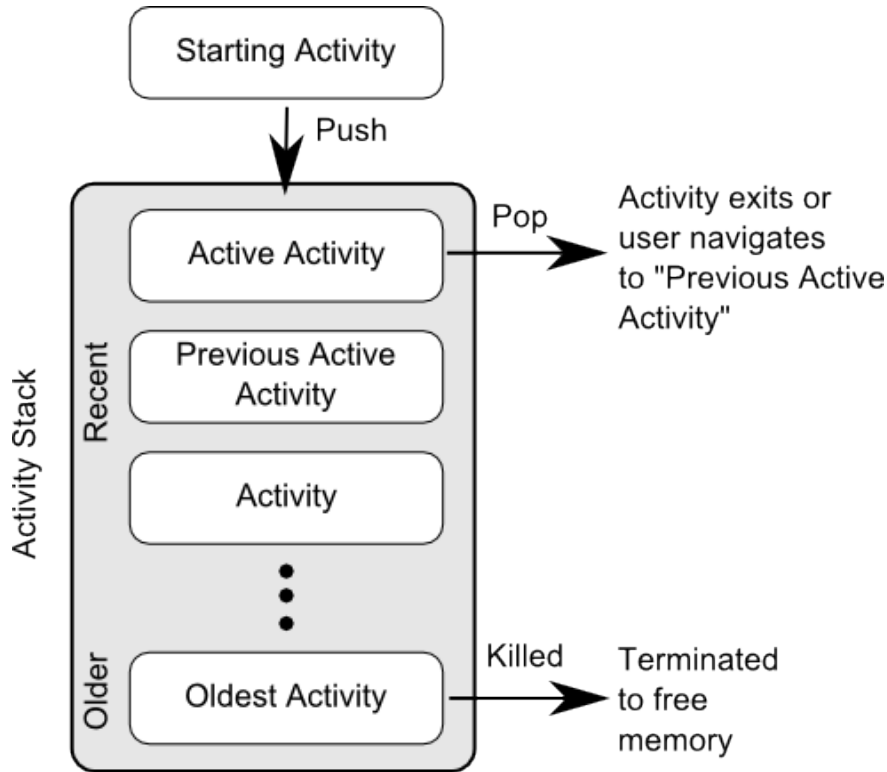


Figure 18-2

18.6 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.

18.7 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.

18.8 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*”.

18.9 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.

19. 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*.

19.1 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.

19.2 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.kt* 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.Snackbar
import android.support.v7.app.AppCompatActivity
import android.view.Menu
import android.view.MenuItem
```

Handling Android Activity State Changes

```
import kotlinx.android.synthetic.main.activity_main.*
import kotlinx.android.synthetic.main.content_main.*

class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_android_sample)
        setSupportActionBar(toolbar)
    }

    fun convertCurrency(view: View) {

        if (dollarText.text.isNotEmpty()) {
            val dollarValue = dollarText.text.toString().toFloat()
            val euroValue = dollarValue * 0.85f
            textView.text = euroValue.toString()
        } else {
            textView.text = "No Value"
        }
    }

    fab.setOnClickListener { view ->
        Snackbar.make(view, "Replace with your own action",
            Snackbar.LENGTH_LONG)
            .setAction("Action", null).show()
    }

    override fun onCreateOptionsMenu(menu: Menu): Boolean {
        // Inflate the menu; this adds items to the action bar if it is present.
        menuInflater.inflate(R.menu.menu_main, menu)
        return true
    }

    override fun onOptionsItemSelected(item: MenuItem): Boolean {
        // 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.
        return when(item.itemId) {
            R.id.action_settings -> true
            else -> 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.kt* 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 19-1, *AppCompatActivity* is clearly subclassed from the *FragmentActivity* class which is itself ultimately a subclass of the *Activity* class:

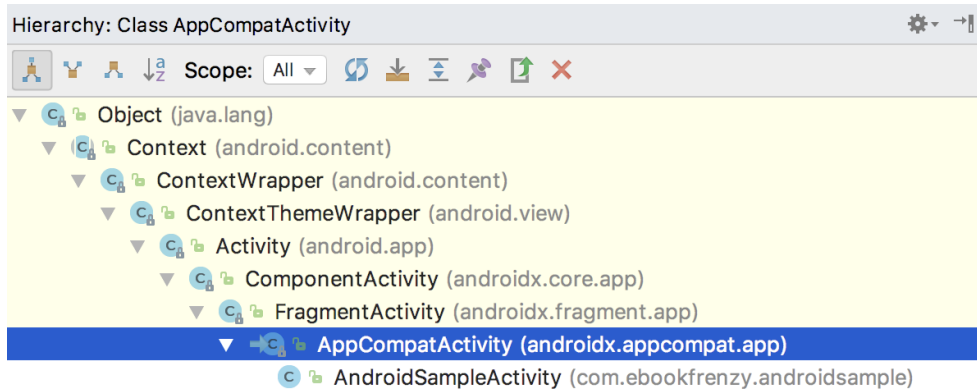


Figure 19-1

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.

19.3 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.

19.4 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(savedInstanceState: Bundle?)** – 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.
- **onViewStatusRestored()** - 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(savedInstanceState: Bundle?)** – 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(outState: Bundle?)** – Called before an activity is destroyed so that the current *dynamic state* (usually relating to the user interface) can be saved. The method is passed the Bundle object into which the state should be saved and which is subsequently passed through to the *onCreate()* and *onRestoreInstanceState()* methods when the activity is restarted. Note that this method is only called in situations where the runtime ascertains that dynamic state needs to be saved.

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

```
override fun 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*”.

19.5 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 19-2:

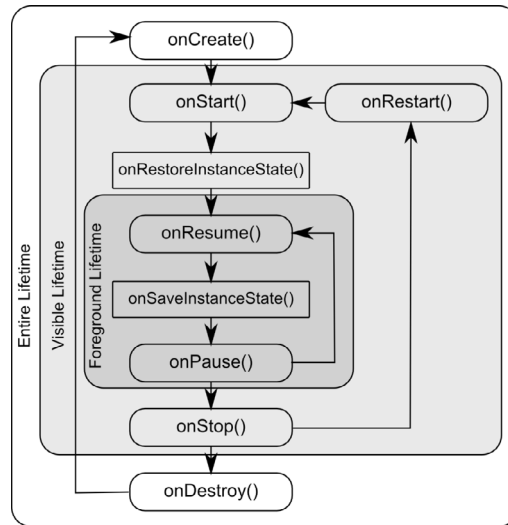


Figure 19-2

19.6 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">
```

19.7 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*”.

19.8 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.

20. 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.

20.1 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 Kotlin.

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.

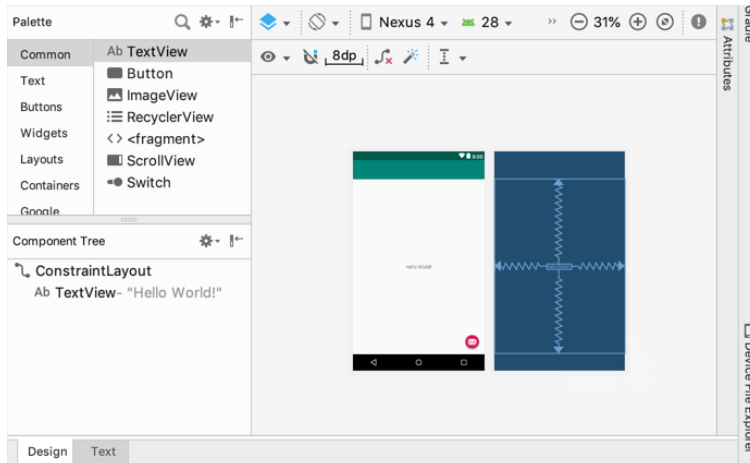


Figure 20-1

20.2 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 20-2.

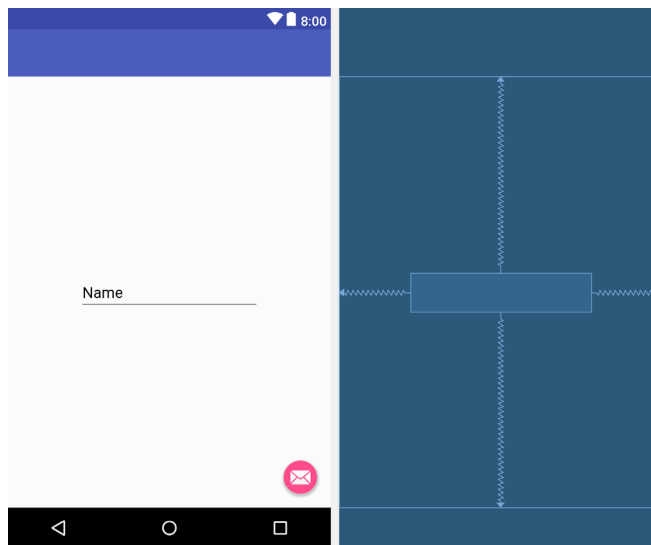


Figure 20-2

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.

20.3 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.kt* 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 com.google.android.material.snackbar.Snackbar
import android.support.v7.app.AppCompatActivity
import android.view.Menu
import android.view.MenuItem

import kotlinx.android.synthetic.main.activity_main.*

class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_state_change)
        setSupportActionBar(toolbar)

        fab.setOnClickListener { view ->
            Snackbar.make(view, "Replace with your own action",
                Snackbar.LENGTH_LONG)
                .setAction("Action", null).show()
        }
    }

    override fun onCreateOptionsMenu(menu: Menu): Boolean {
        // Inflate the menu; this adds items to the action bar if it is present.
        menuInflater.inflate(R.menu.menu_main, menu)
        return true
    }

    override fun onOptionsItemSelected(item: MenuItem): Boolean {
```

Android Activity State Changes by Example

```
// 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.
return when (item.itemId) {
    R.id.action_settings -> true
    else -> 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 com.google.android.material.snackbar.Snackbar
import android.support.v7.app.AppCompatActivity
import android.view.Menu
import android.view.MenuItem
import android.util.Log

import kotlinx.android.synthetic.main.activity_main.*

class MainActivity : AppCompatActivity() {

    val TAG = "StateChange"

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_state_change)
        setSupportActionBar(toolbar)

        fab.setOnClickListener { 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 fun onStart() {
    super.onStart()
    Log.i(TAG, "onStart")
}

override fun onResume() {
    super.onResume()
    Log.i(TAG, "onResume")
}

override fun onPause() {
    super.onPause()
    Log.i(TAG, "onPause")
}

override fun onStop() {
    super.onStop()
    Log.i(TAG, "onStop")
}

override fun onRestart() {
    super.onRestart()
    Log.i(TAG, "onRestart")
}

override fun onDestroy() {
    super.onDestroy()
    Log.i(TAG, "onDestroy")
}

override fun onSaveInstanceState(outState: Bundle?) {
    super.onSaveInstanceState(outState)
    Log.i(TAG, "onSaveInstanceState")
}

override fun onRestoreInstanceState(savedInstanceState: Bundle?) {
    super.onRestoreInstanceState(savedInstanceState)
    Log.i(TAG, "onRestoreInstanceState")
}

```

20.4 Filtering the Logcat Panel

The purpose of the code added to the overridden methods in *MainActivity.kt* 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

Android Activity State Changes by Example

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 20-3) 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.

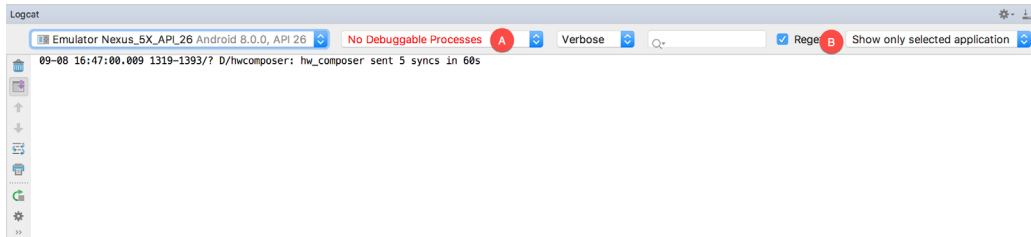


Figure 20-3

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 20-4), name the filter *Lifecycle* and, in the *Log Tag* field, enter the Tag value declared in *MainActivity.kt* (in the above code example this was *StateChange*).

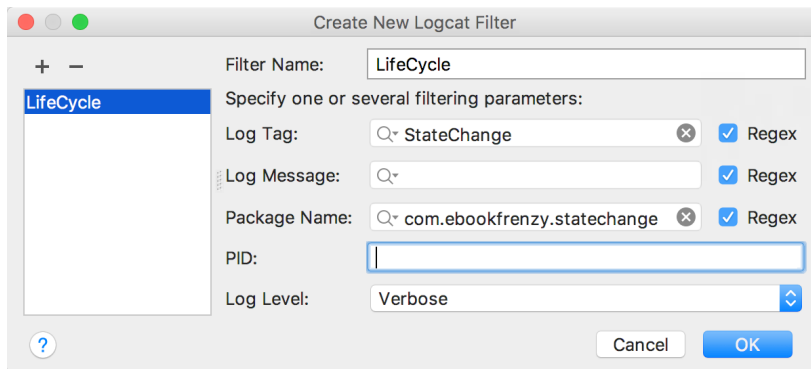


Figure 20-4

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.

20.5 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 20-5 below, select the *Run -> Run...* menu option or use the Shift+F10 keyboard shortcut:

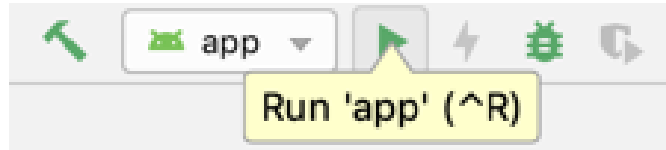


Figure 20-5

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):

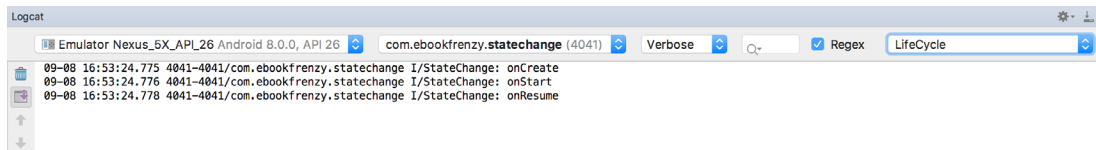


Figure 20-6

20.6 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:

Android Activity State Changes by Example

```
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.

20.7 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.