

Android Studio Flamingo Essentials

Kotlin Edition

Android Studio Flamingo Essentials – Kotlin Edition

ISBN-13: 978-1-951442-68-2

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

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

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

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

Rev: 1.0

Table of Contents

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	4
2.3.1 Installation on Windows	4
2.3.2 Installation on macOS	4
2.3.3 Installation on Linux	5
2.4 The Android Studio setup wizard	5
2.5 Installing additional Android SDK packages	6
2.6 Installing the Android SDK Command-line Tools	9
2.6.1 Windows 8.1	10
2.6.2 Windows 10	10
2.6.3 Windows 11	11
2.6.4 Linux	11
2.6.5 macOS	11
2.7 Android Studio memory management	11
2.8 Updating Android Studio and the SDK	12
2.9 Summary	12
3. Creating an Example Android App in Android Studio	13
3.1 About the Project	13
3.2 Creating a New Android Project	13
3.3 Creating an Activity	14
3.4 Defining the Project and SDK Settings	14
3.5 Modifying the Example Application	15
3.6 Modifying the User Interface	16
3.7 Reviewing the Layout and Resource Files	21
3.8 Adding Interaction	24
3.9 Summary	25
4. Creating an Android Virtual Device (AVD) in Android Studio	27
4.1 About Android Virtual Devices	27
4.2 Starting the Emulator	29
4.3 Running the Application in the AVD	30
4.4 Running on Multiple Devices	31
4.5 Stopping a Running Application	32
4.6 Supporting Dark Theme	32
4.7 Running the Emulator in a Separate Window	33
4.8 Enabling the Device Frame	35

4.9 Summary	36
5. Using and Configuring the Android Studio AVD Emulator	37
5.1 The Emulator Environment	37
5.2 Emulator Toolbar Options	37
5.3 Working in Zoom Mode	39
5.4 Resizing the Emulator Window.....	39
5.5 Extended Control Options.....	39
5.5.1 Location	40
5.5.2 Displays.....	40
5.5.3 Cellular	40
5.5.4 Battery.....	40
5.5.5 Camera.....	40
5.5.6 Phone	40
5.5.7 Directional Pad.....	40
5.5.8 Microphone.....	40
5.5.9 Fingerprint	40
5.5.10 Virtual Sensors	41
5.5.11 Snapshots.....	41
5.5.12 Record and Playback	41
5.5.13 Google Play	41
5.5.14 Settings	41
5.5.15 Help.....	41
5.6 Working with Snapshots.....	41
5.7 Configuring Fingerprint Emulation	42
5.8 The Emulator in Tool Window Mode.....	43
5.9 Creating a Resizable Emulator.....	44
5.10 Summary	45
6. A Tour of the Android Studio User Interface	47
6.1 The Welcome Screen	47
6.2 The Main Window	48
6.3 The Tool Windows	49
6.4 Android Studio Keyboard Shortcuts	52
6.5 Switcher and Recent Files Navigation	53
6.6 Changing the Android Studio Theme	54
6.7 Summary	55
7. Testing Android Studio Apps on a Physical Android Device.....	57
7.1 An Overview of the Android Debug Bridge (ADB)	57
7.2 Enabling USB Debugging ADB on Android Devices.....	57
7.2.1 macOS ADB Configuration	58
7.2.2 Windows ADB Configuration	59
7.2.3 Linux adb Configuration.....	60
7.3 Resolving USB Connection Issues	60
7.4 Enabling Wireless Debugging on Android Devices	61
7.5 Testing the adb Connection	63
7.6 Device Mirroring.....	63
7.7 Summary	63
8. The Basics of the Android Studio Code Editor.....	65

8.1 The Android Studio Editor.....	65
8.2 Splitting the Editor Window.....	67
8.3 Code Completion.....	68
8.4 Statement Completion.....	69
8.5 Parameter Information.....	70
8.6 Parameter Name Hints.....	70
8.7 Code Generation.....	70
8.8 Code Folding.....	71
8.9 Quick Documentation Lookup.....	72
8.10 Code Reformatting.....	73
8.11 Finding Sample Code.....	73
8.12 Live Templates.....	74
8.13 Summary.....	74
9. An Overview of the Android Architecture	75
9.1 The Android Software Stack.....	75
9.2 The Linux Kernel.....	76
9.3 Android Runtime – ART.....	76
9.4 Android Libraries.....	76
9.4.1 C/C++ Libraries.....	77
9.5 Application Framework.....	77
9.6 Applications.....	78
9.7 Summary.....	78
10. The Anatomy of an Android Application	79
10.1 Android Activities.....	79
10.2 Android Fragments.....	79
10.3 Android Intents.....	80
10.4 Broadcast Intents.....	80
10.5 Broadcast Receivers.....	80
10.6 Android Services.....	80
10.7 Content Providers.....	81
10.8 The Application Manifest.....	81
10.9 Application Resources.....	81
10.10 Application Context.....	81
10.11 Summary.....	81
11. An Introduction to Kotlin.....	83
11.1 What is Kotlin?.....	83
11.2 Kotlin and Java.....	83
11.3 Converting from Java to Kotlin.....	83
11.4 Kotlin and Android Studio.....	84
11.5 Experimenting with Kotlin.....	84
11.6 Semi-colons in Kotlin.....	85
11.7 Summary.....	85
12. Kotlin Data Types, Variables, and Nullability	87
12.1 Kotlin Data Types.....	87
12.1.1 Integer Data Types.....	88
12.1.2 Floating-Point Data Types.....	88
12.1.3 Boolean Data Type.....	88

Table of Contents

12.1.4 Character Data Type.....	88
12.1.5 String Data Type.....	88
12.1.6 Escape Sequences	89
12.2 Mutable Variables.....	90
12.3 Immutable Variables	90
12.4 Declaring Mutable and Immutable Variables.....	90
12.5 Data Types are Objects	90
12.6 Type Annotations and Type Inference	91
12.7 Nullable Type.....	92
12.8 The Safe Call Operator	92
12.9 Not-Null Assertion.....	93
12.10 Nullable Types and the let Function.....	93
12.11 Late Initialization (lateinit)	94
12.12 The Elvis Operator	95
12.13 Type Casting and Type Checking	95
12.14 Summary.....	96
13. Kotlin Operators and Expressions	97
13.1 Expression Syntax in Kotlin.....	97
13.2 The Basic Assignment Operator.....	97
13.3 Kotlin Arithmetic Operators	97
13.4 Augmented Assignment Operators	98
13.5 Increment and Decrement Operators	98
13.6 Equality Operators	99
13.7 Boolean Logical Operators	99
13.8 Range Operator	100
13.9 Bitwise Operators.....	100
13.9.1 Bitwise Inversion	100
13.9.2 Bitwise AND	101
13.9.3 Bitwise OR.....	101
13.9.4 Bitwise XOR.....	101
13.9.5 Bitwise Left Shift.....	102
13.9.6 Bitwise Right Shift.....	102
13.10 Summary.....	103
14. Kotlin Control Flow	105
14.1 Looping Control flow	105
14.1.1 The Kotlin <i>for-in</i> Statement.....	105
14.1.2 The <i>while</i> Loop	106
14.1.3 The <i>do ... while</i> loop	107
14.1.4 Breaking from Loops	107
14.1.5 The <i>continue</i> Statement	108
14.1.6 Break and Continue Labels.....	108
14.2 Conditional Control Flow.....	109
14.2.1 Using the <i>if</i> Expressions	109
14.2.2 Using <i>if ... else ...</i> Expressions	110
14.2.3 Using <i>if ... else if ...</i> Expressions	110
14.2.4 Using the <i>when</i> Statement.....	110
14.3 Summary	111
15. An Overview of Kotlin Functions and Lambdas	113

15.1 What is a Function?	113
15.2 How to Declare a Kotlin Function	113
15.3 Calling a Kotlin Function.....	114
15.4 Single Expression Functions.....	114
15.5 Local Functions	114
15.6 Handling Return Values	115
15.7 Declaring Default Function Parameters.....	115
15.8 Variable Number of Function Parameters	115
15.9 Lambda Expressions	116
15.10 Higher-order Functions	117
15.11 Summary	118
16. The Basics of Object Oriented Programming in Kotlin	119
16.1 What is an Object?	119
16.2 What is a Class?	119
16.3 Declaring a Kotlin Class.....	119
16.4 Adding Properties to a Class.....	120
16.5 Defining Methods	120
16.6 Declaring and Initializing a Class Instance.....	120
16.7 Primary and Secondary Constructors.....	120
16.8_INITIALIZER Blocks.....	123
16.9 Calling Methods and Accessing Properties	123
16.10 Custom Accessors	123
16.11 Nested and Inner Classes	124
16.12 Companion Objects.....	125
16.13 Summary	127
17. An Introduction to Kotlin Inheritance and Subclassing.....	129
17.1 Inheritance, Classes and Subclasses.....	129
17.2 Subclassing Syntax	129
17.3 A Kotlin Inheritance Example.....	130
17.4 Extending the Functionality of a Subclass	131
17.5 Overriding Inherited Methods.....	132
17.6 Adding a Custom Secondary Constructor.....	133
17.7 Using the SavingsAccount Class	133
17.8 Summary	133
18. An Overview of Android View Binding.....	135
18.1 Find View by Id	135
18.2 View Binding	135
18.3 Converting the AndroidSample project.....	136
18.4 Enabling View Binding.....	136
18.5 Using View Binding	136
18.6 Choosing an Option	137
18.7 View Binding in the Book Examples	137
18.8 Migrating a Project to View Binding.....	138
18.9 Summary	138
19. Understanding Android Application and Activity Lifecycles.....	139
19.1 Android Applications and Resource Management.....	139
19.2 Android Process States	139

Table of Contents

19.2.1 Foreground Process	140
19.2.2 Visible Process	140
19.2.3 Service Process	140
19.2.4 Background Process.....	140
19.2.5 Empty Process	141
19.3 Inter-Process Dependencies	141
19.4 The Activity Lifecycle.....	141
19.5 The Activity Stack.....	141
19.6 Activity States	142
19.7 Configuration Changes	142
19.8 Handling State Change.....	143
19.9 Summary	143
20. Handling Android Activity State Changes.....	145
20.1 New vs. Old Lifecycle Techniques.....	145
20.2 The Activity and Fragment Classes.....	145
20.3 Dynamic State vs. Persistent State.....	147
20.4 The Android Lifecycle Methods	147
20.5 Lifetimes	149
20.6 Foldable Devices and Multi-Resume	150
20.7 Disabling Configuration Change Restarts	150
20.8 Lifecycle Method Limitations.....	150
20.9 Summary	151
21. Android Activity State Changes by Example	153
21.1 Creating the State Change Example Project	153
21.2 Designing the User Interface	154
21.3 Overriding the Activity Lifecycle Methods	155
21.4 Filtering the Logcat Panel.....	157
21.5 Running the Application.....	158
21.6 Experimenting with the Activity.....	159
21.7 Summary	160
22. Saving and Restoring the State of an Android Activity	161
22.1 Saving Dynamic State	161
22.2 Default Saving of User Interface State	161
22.3 The Bundle Class	162
22.4 Saving the State.....	163
22.5 Restoring the State	164
22.6 Testing the Application.....	164
22.7 Summary	164
23. Understanding Android Views, View Groups and Layouts	165
23.1 Designing for Different Android Devices	165
23.2 Views and View Groups	165
23.3 Android Layout Managers	165
23.4 The View Hierarchy	167
23.5 Creating User Interfaces	168
23.6 Summary	168
24. A Guide to the Android Studio Layout Editor Tool	169

24.1 Basic vs. Empty Views Activity Templates	169
24.2 The Android Studio Layout Editor	173
24.3 Design Mode.....	173
24.4 The Palette	174
24.5 Design Mode and Layout Views.....	175
24.6 Night Mode	176
24.7 Code Mode.....	176
24.8 Split Mode	176
24.9 Setting Attributes.....	177
24.10 Transforms	178
24.11 Tools Visibility Toggles.....	179
24.12 Converting Views.....	181
24.13 Displaying Sample Data	182
24.14 Creating a Custom Device Definition	182
24.15 Changing the Current Device.....	183
24.16 Layout Validation	184
24.17 Summary	184
25. A Guide to the Android ConstraintLayout.....	185
25.1 How ConstraintLayout Works.....	185
25.1.1 Constraints.....	185
25.1.2 Margins.....	186
25.1.3 Opposing Constraints.....	186
25.1.4 Constraint Bias	187
25.1.5 Chains.....	188
25.1.6 Chain Styles.....	188
25.2 Baseline Alignment.....	189
25.3 Configuring Widget Dimensions.....	189
25.4 Guideline Helper	190
25.5 Group Helper	190
25.6 Barrier Helper	190
25.7 Flow Helper	192
25.8 Ratios	193
25.9 ConstraintLayout Advantages	193
25.10 ConstraintLayout Availability.....	194
25.11 Summary	194
26. A Guide to Using ConstraintLayout in Android Studio	195
26.1 Design and Layout Views.....	195
26.2 Autoconnect Mode	196
26.3 Inference Mode.....	197
26.4 Manipulating Constraints Manually.....	197
26.5 Adding Constraints in the Inspector	198
26.6 Viewing Constraints in the Attributes Window.....	199
26.7 Deleting Constraints.....	200
26.8 Adjusting Constraint Bias	200
26.9 Understanding ConstraintLayout Margins.....	201
26.10 The Importance of Opposing Constraints and Bias	202
26.11 Configuring Widget Dimensions.....	204
26.12 Design Time Tools Positioning	205

Table of Contents

26.13 Adding Guidelines	206
26.14 Adding Barriers	208
26.15 Adding a Group	209
26.16 Working with the Flow Helper	210
26.17 Widget Group Alignment and Distribution	211
26.18 Converting other Layouts to ConstraintLayout	212
26.19 Summary	212
27. Working with ConstraintLayout Chains and Ratios in Android Studio	213
27.1 Creating a Chain	213
27.2 Changing the Chain Style	215
27.3 Spread Inside Chain Style	215
27.4 Packed Chain Style	216
27.5 Packed Chain Style with Bias	216
27.6 Weighted Chain	216
27.7 Working with Ratios	217
27.8 Summary	219
28. An Android Studio Layout Editor ConstraintLayout Tutorial	221
28.1 An Android Studio Layout Editor Tool Example	221
28.2 Preparing the Layout Editor Environment	221
28.3 Adding the Widgets to the User Interface	222
28.4 Adding the Constraints	225
28.5 Testing the Layout	226
28.6 Using the Layout Inspector	227
28.7 Summary	228
29. Manual XML Layout Design in Android Studio	229
29.1 Manually Creating an XML Layout	229
29.2 Manual XML vs. Visual Layout Design	232
29.3 Summary	232
30. Managing Constraints using Constraint Sets	233
30.1 Kotlin Code vs. XML Layout Files	233
30.2 Creating Views	233
30.3 View Attributes	234
30.4 Constraint Sets	234
30.4.1 Establishing Connections	234
30.4.2 Applying Constraints to a Layout	234
30.4.3 Parent Constraint Connections	234
30.4.4 Sizing Constraints	235
30.4.5 Constraint Bias	235
30.4.6 Alignment Constraints	235
30.4.7 Copying and Applying Constraint Sets	235
30.4.8 ConstraintLayout Chains	235
30.4.9 Guidelines	236
30.4.10 Removing Constraints	236
30.4.11 Scaling	236
30.4.12 Rotation	237
30.5 Summary	237
31. An Android ConstraintSet Tutorial	239

31.1 Creating the Example Project in Android Studio	239
31.2 Adding Views to an Activity.....	239
31.3 Setting View Attributes.....	240
31.4 Creating View IDs.....	241
31.5 Configuring the Constraint Set	242
31.6 Adding the EditText View	243
31.7 Converting Density Independent Pixels (dp) to Pixels (px).....	244
31.8 Summary	245
32. A Guide to using Apply Changes in Android Studio.....	247
32.1 Introducing Apply Changes.....	247
32.2 Understanding Apply Changes Options	247
32.3 Using Apply Changes.....	248
32.4 Configuring Apply Changes Fallback Settings.....	249
32.5 An Apply Changes Tutorial.....	249
32.6 Using Apply Code Changes	249
32.7 Using Apply Changes and Restart Activity.....	250
32.8 Using Run App	250
32.9 Summary	250
33. An Overview and Example of Android Event Handling	251
33.1 Understanding Android Events.....	251
33.2 Using the android:onClick Resource	251
33.3 Event Listeners and Callback Methods	252
33.4 An Event Handling Example	252
33.5 Designing the User Interface	253
33.6 The Event Listener and Callback Method.....	253
33.7 Consuming Events	255
33.8 Summary	256
34. Android Touch and Multi-touch Event Handling	257
34.1 Intercepting Touch Events	257
34.2 The MotionEvent Object.....	258
34.3 Understanding Touch Actions.....	258
34.4 Handling Multiple Touches	258
34.5 An Example Multi-Touch Application	259
34.6 Designing the Activity User Interface	259
34.7 Implementing the Touch Event Listener.....	259
34.8 Running the Example Application.....	262
34.9 Summary	262
35. Detecting Common Gestures Using the Android Gesture Detector Class	263
35.1 Implementing Common Gesture Detection.....	263
35.2 Creating an Example Gesture Detection Project	264
35.3 Implementing the Listener Class.....	264
35.4 Creating the GestureDetectorCompat Instance.....	266
35.5 Implementing the onTouchEvent() Method.....	266
35.6 Testing the Application.....	267
35.7 Summary	267
36. Implementing Custom Gesture and Pinch Recognition on Android	269

Table of Contents

36.1 The Android Gesture Builder Application.....	269
36.2 The GestureOverlayView Class	269
36.3 Detecting Gestures.....	269
36.4 Identifying Specific Gestures	269
36.5 Installing and Running the Gesture Builder Application	270
36.6 Creating a Gestures File	270
36.7 Creating the Example Project.....	270
36.8 Extracting the Gestures File from the SD Card	271
36.9 Adding the Gestures File to the Project	271
36.10 Designing the User Interface	271
36.11 Loading the Gestures File	272
36.12 Registering the Event Listener.....	273
36.13 Implementing the onGesturePerformed Method.....	273
36.14 Testing the Application.....	274
36.15 Configuring the GestureOverlayView.....	274
36.16 Intercepting Gestures.....	275
36.17 Detecting Pinch Gestures.....	275
36.18 A Pinch Gesture Example Project.....	275
36.19 Summary.....	277
37. An Introduction to Android Fragments	279
37.1 What is a Fragment?	279
37.2 Creating a Fragment	279
37.3 Adding a Fragment to an Activity using the Layout XML File.....	280
37.4 Adding and Managing Fragments in Code	282
37.5 Handling Fragment Events	283
37.6 Implementing Fragment Communication.....	283
37.7 Summary	285
38. Using Fragments in Android Studio - An Example.....	287
38.1 About the Example Fragment Application	287
38.2 Creating the Example Project.....	287
38.3 Creating the First Fragment Layout.....	287
38.4 Migrating a Fragment to View Binding	289
38.5 Adding the Second Fragment.....	290
38.6 Adding the Fragments to the Activity	291
38.7 Making the Toolbar Fragment Talk to the Activity	292
38.8 Making the Activity Talk to the Text Fragment	295
38.9 Testing the Application.....	296
38.10 Summary.....	296
39. Modern Android App Architecture with Jetpack.....	297
39.1 What is Android Jetpack?	297
39.2 The “Old” Architecture.....	297
39.3 Modern Android Architecture.....	297
39.4 The ViewModel Component	298
39.5 The LiveData Component.....	298
39.6 ViewModel Saved State.....	299
39.7 LiveData and Data Binding.....	300
39.8 Android Lifecycles	300
39.9 Repository Modules.....	300

39.10 Summary	301
40. An Android ViewModel Tutorial	303
40.1 About the Project	303
40.2 Creating the ViewModel Example Project.....	303
40.3 Removing Unwanted Project Elements.....	303
40.4 Designing the Fragment Layout.....	304
40.5 Implementing the View Model.....	305
40.6 Associating the Fragment with the View Model.....	306
40.7 Modifying the Fragment	306
40.8 Accessing the ViewModel Data	307
40.9 Testing the Project.....	307
40.10 Summary	308
41. An Android Jetpack LiveData Tutorial	309
41.1 LiveData - A Recap	309
41.2 Adding LiveData to the ViewModel.....	309
41.3 Implementing the Observer.....	311
41.4 Summary	312
42. An Overview of Android Jetpack Data Binding	313
42.1 An Overview of Data Binding.....	313
42.2 The Key Components of Data Binding	313
42.2.1 The Project Build Configuration	313
42.2.2 The Data Binding Layout File.....	314
42.2.3 The Layout File Data Element	315
42.2.4 The Binding Classes	316
42.2.5 Data Binding Variable Configuration.....	316
42.2.6 Binding Expressions (One-Way).....	317
42.2.7 Binding Expressions (Two-Way).....	318
42.2.8 Event and Listener Bindings.....	318
42.3 Summary	319
43. An Android Jetpack Data Binding Tutorial.....	321
43.1 Removing the Redundant Code.....	321
43.2 Enabling Data Binding	322
43.3 Adding the Layout Element.....	323
43.4 Adding the Data Element to Layout File.....	324
43.5 Working with the Binding Class	324
43.6 Assigning the ViewModel Instance to the Data Binding Variable	326
43.7 Adding Binding Expressions	326
43.8 Adding the Conversion Method	327
43.9 Adding a Listener Binding.....	327
43.10 Testing the App.....	328
43.11 Summary	328
44. An Android ViewModel Saved State Tutorial.....	329
44.1 Understanding ViewModel State Saving.....	329
44.2 Implementing ViewModel State Saving.....	329
44.3 Saving and Restoring State.....	330
44.4 Adding Saved State Support to the ViewModelDemo Project.....	331

44.5 Summary	333
45. Working with Android Lifecycle-Aware Components	335
45.1 Lifecycle Awareness	335
45.2 Lifecycle Owners	335
45.3 Lifecycle Observers	336
45.4 Lifecycle States and Events	336
45.5 Summary	337
46. An Android Jetpack Lifecycle Awareness Tutorial	339
46.1 Creating the Example Lifecycle Project	339
46.2 Creating a Lifecycle Observer	339
46.3 Adding the Observer	340
46.4 Testing the Observer	341
46.5 Creating a Lifecycle Owner	341
46.6 Testing the Custom Lifecycle Owner	343
46.7 Summary	343
47. An Overview of the Navigation Architecture Component	345
47.1 Understanding Navigation	345
47.2 Declaring a Navigation Host	346
47.3 The Navigation Graph	348
47.4 Accessing the Navigation Controller	349
47.5 Triggering a Navigation Action	349
47.6 Passing Arguments	350
47.7 Summary	350
48. An Android Jetpack Navigation Component Tutorial	351
48.1 Creating the NavigationDemo Project	351
48.2 Adding Navigation to the Build Configuration	351
48.3 Creating the Navigation Graph Resource File	352
48.4 Declaring a Navigation Host	353
48.5 Adding Navigation Destinations	354
48.6 Designing the Destination Fragment Layouts	356
48.7 Adding an Action to the Navigation Graph	357
48.8 Implement the OnFragmentInteractionListener	359
48.9 Adding View Binding Support to the Destination Fragments	360
48.10 Triggering the Action	360
48.11 Passing Data Using Safeargs	361
48.12 Summary	364
49. An Introduction to MotionLayout	365
49.1 An Overview of MotionLayout	365
49.2 MotionLayout	365
49.3 MotionScene	365
49.4 Configuring ConstraintSets	366
49.5 Custom Attributes	367
49.6 Triggering an Animation	368
49.7 Arc Motion	370
49.8 Keyframes	370
49.8.1 Attribute Keyframes	370

49.8.2 Position Keyframes	371
49.9 Time Linearity	374
49.10 KeyTrigger	374
49.11 Cycle and Time Cycle Keyframes	375
49.12 Starting an Animation from Code	375
49.13 Summary	376
50. An Android MotionLayout Editor	377
50.1 Creating the MotionLayoutDemo Project	377
50.2 ConstraintLayout to MotionLayout Conversion	377
50.3 Configuring Start and End Constraints	379
50.4 Previewing the MotionLayout Animation	381
50.5 Adding an OnClick Gesture	382
50.6 Adding an Attribute Keyframe to the Transition	383
50.7 Adding a CustomAttribute to a Transition	386
50.8 Adding Position Keyframes	387
50.9 Summary	390
51. A MotionLayout KeyCycle Tutorial	391
51.1 An Overview of Cycle Keyframes	391
51.2 Using the Cycle Editor	395
51.3 Creating the KeyCycleDemo Project	396
51.4 Configuring the Start and End Constraints	396
51.5 Creating the Cycles	398
51.6 Previewing the Animation	400
51.7 Adding the KeyFrameSet to the MotionScene	400
51.8 Summary	402
52. Working with the Floating Action Button and Snackbar	403
52.1 The Material Design	403
52.2 The Design Library	403
52.3 The Floating Action Button (FAB)	403
52.4 The Snackbar	404
52.5 Creating the Example Project	405
52.6 Reviewing the Project	405
52.7 Removing Navigation Features	406
52.8 Changing the Floating Action Button	406
52.9 Adding an Action to the Snackbar	408
52.10 Summary	408
53. Creating a Tabbed Interface using the TabLayout Component	409
53.1 An Introduction to the ViewPager2	409
53.2 An Overview of the TabLayout Component	409
53.3 Creating the TabLayoutDemo Project	410
53.4 Creating the First Fragment	410
53.5 Duplicating the Fragments	412
53.6 Adding the TabLayout and ViewPager2	413
53.7 Creating the Pager Adapter	414
53.8 Performing the Initialization Tasks	415
53.9 Testing the Application	417
53.10 Customizing the TabLayout	417

53.11 Summary	418
54. Working with the RecyclerView and CardView Widgets	419
54.1 An Overview of the RecyclerView	419
54.2 An Overview of the CardView	421
54.3 Summary	422
55. An Android RecyclerView and CardView Tutorial	423
55.1 Creating the CardDemo Project	423
55.2 Modifying the Basic Views Activity Project	423
55.3 Designing the CardView Layout	424
55.4 Adding the RecyclerView	425
55.5 Adding the Image Files	425
55.6 Creating the RecyclerView Adapter	425
55.7 Initializing the RecyclerView Component	427
55.8 Testing the Application	428
55.9 Responding to Card Selections	429
55.10 Summary	430
56. Working with the AppBar and Collapsing Toolbar Layouts	431
56.1 The Anatomy of an AppBar	431
56.2 The Example Project	432
56.3 Coordinating the RecyclerView and Toolbar	432
56.4 Introducing the Collapsing Toolbar Layout	434
56.5 Changing the Title and Scrim Color	437
56.6 Summary	438
57. An Overview of Android Intents	439
57.1 An Overview of Intents	439
57.2 Explicit Intents	439
57.3 Returning Data from an Activity	440
57.4 Implicit Intents	441
57.5 Using Intent Filters	442
57.6 Automatic Link Verification	442
57.7 Manually Enabling Links	445
57.8 Checking Intent Availability	446
57.9 Summary	447
58. Android Explicit Intents – A Worked Example	449
58.1 Creating the Explicit Intent Example Application	449
58.2 Designing the User Interface Layout for MainActivity	449
58.3 Creating the Second Activity Class	450
58.4 Designing the User Interface Layout for SecondActivity	451
58.5 Reviewing the Application Manifest File	451
58.6 Creating the Intent	452
58.7 Extracting Intent Data	453
58.8 Launching SecondActivity as a Sub-Activity	454
58.9 Returning Data from a Sub-Activity	455
58.10 Testing the Application	455
58.11 Summary	455
59. Android Implicit Intents – A Worked Example	457

59.1 Creating the Android Studio Implicit Intent Example Project	457
59.2 Designing the User Interface	457
59.3 Creating the Implicit Intent	458
59.4 Adding a Second Matching Activity.....	459
59.5 Adding the Web View to the UI.....	459
59.6 Obtaining the Intent URL	460
59.7 Modifying the MyWebView Project Manifest File	461
59.8 Installing the MyWebView Package on a Device.....	462
59.9 Testing the Application.....	463
59.10 Manually Enabling the Link	463
59.11 Automatic Link Verification	465
59.12 Summary	467
60. Android Broadcast Intents and Broadcast Receivers	469
60.1 An Overview of Broadcast Intents.....	469
60.2 An Overview of Broadcast Receivers	470
60.3 Obtaining Results from a Broadcast.....	471
60.4 Sticky Broadcast Intents	471
60.5 The Broadcast Intent Example.....	472
60.6 Creating the Example Application	472
60.7 Creating and Sending the Broadcast Intent.....	472
60.8 Creating the Broadcast Receiver	473
60.9 Registering the Broadcast Receiver.....	474
60.10 Testing the Broadcast Example	475
60.11 Listening for System Broadcasts.....	475
60.12 Summary	476
61. An Introduction to Kotlin Coroutines.....	477
61.1 What are Coroutines?	477
61.2 Threads vs Coroutines.....	477
61.3 Coroutine Scope	478
61.4 Suspend Functions.....	478
61.5 Coroutine Dispatchers.....	478
61.6 Coroutine Builders.....	479
61.7 Jobs.....	479
61.8 Coroutines – Suspending and Resuming.....	480
61.9 Returning Results from a Coroutine	481
61.10 Using withContext	481
61.11 Coroutine Channel Communication	483
61.12 Summary	484
62. An Android Kotlin Coroutines Tutorial.....	485
62.1 Creating the Coroutine Example Application.....	485
62.2 Adding Coroutine Support to the Project.....	485
62.3 Designing the User Interface	485
62.4 Implementing the SeekBar.....	487
62.5 Adding the Suspend Function.....	487
62.6 Implementing the launchCoroutines Method.....	488
62.7 Testing the App.....	489
62.8 Summary	489

63. An Overview of Android Services.....	491
63.1 Intent Service.....	491
63.2 Bound Service.....	491
63.3 The Anatomy of a Service	492
63.4 Controlling Destroyed Service Restart Options.....	492
63.5 Declaring a Service in the Manifest File.....	492
63.6 Starting a Service Running on System Startup.....	494
63.7 Summary	494
64. Android Local Bound Services – A Worked Example.....	495
64.1 Understanding Bound Services.....	495
64.2 Bound Service Interaction Options.....	495
64.3 A Local Bound Service Example.....	495
64.4 Adding a Bound Service to the Project	496
64.5 Implementing the Binder	496
64.6 Binding the Client to the Service	498
64.7 Completing the Example.....	499
64.8 Testing the Application.....	500
64.9 Summary	501
65. Android Remote Bound Services – A Worked Example	503
65.1 Client to Remote Service Communication.....	503
65.2 Creating the Example Application.....	503
65.3 Designing the User Interface	503
65.4 Implementing the Remote Bound Service.....	504
65.5 Configuring a Remote Service in the Manifest File.....	505
65.6 Launching and Binding to the Remote Service.....	505
65.7 Sending a Message to the Remote Service	507
65.8 Summary	507
66. An Introduction to Kotlin Flow	509
66.1 Understanding Flows.....	509
66.2 Creating the Sample Project	509
66.3 Adding the Kotlin Lifecycle Library	510
66.4 Declaring a Flow.....	510
66.5 Emitting Flow Data	511
66.6 Collecting Flow Data	511
66.7 Adding a Flow Buffer.....	512
66.8 Transforming Data with Intermediaries	514
66.9 Terminal Flow Operators	515
66.10 Flow Flattening.....	516
66.11 Combining Multiple Flows	517
66.12 Hot and Cold Flows	518
66.13 StateFlow	519
66.14 SharedFlow.....	520
66.15 Summary	521
67. An Android SharedFlow Tutorial	523
67.1 About the Project	523
67.2 Creating the SharedFlowDemo Project.....	523
67.3 Designing the User Interface Layout	523

67.4 Adding the List Row Layout	523
67.5 Adding the RecyclerView Adapter.....	524
67.6 Adding the ViewModel	525
67.7 Configuring the ViewModelProvider.....	526
67.8 Collecting the Flow Values.....	527
67.9 Testing the SharedFlowDemo App	528
67.10 Handling Flows in the Background.....	528
67.11 Summary	531
68. An Overview of Android SQLite Databases	533
68.1 Understanding Database Tables	533
68.2 Introducing Database Schema	533
68.3 Columns and Data Types	533
68.4 Database Rows	534
68.5 Introducing Primary Keys	534
68.6 What is SQLite?	534
68.7 Structured Query Language (SQL).....	534
68.8 Trying SQLite on an Android Virtual Device (AVD)	535
68.9 The Android Room Persistence Library.....	537
68.10 Summary	537
69. The Android Room Persistence Library	539
69.1 Revisiting Modern App Architecture	539
69.2 Key Elements of Room Database Persistence.....	539
69.2.1 Repository	540
69.2.2 Room Database	540
69.2.3 Data Access Object (DAO)	540
69.2.4 Entities.....	540
69.2.5 SQLite Database	540
69.3 Understanding Entities.....	541
69.4 Data Access Objects.....	543
69.5 The Room Database	544
69.6 The Repository.....	545
69.7 In-Memory Databases	546
69.8 Database Inspector.....	546
69.9 Summary	547
70. An Android TableLayout and TableRow Tutorial	549
70.1 The TableLayout and TableRow Layout Views.....	549
70.2 Creating the Room Database Project	550
70.3 Converting to a LinearLayout.....	550
70.4 Adding the TableLayout to the User Interface.....	551
70.5 Configuring the TableRows	552
70.6 Adding the Button Bar to the Layout	553
70.7 Adding the RecyclerView.....	554
70.8 Adjusting the Layout Margins.....	555
70.9 Summary	555
71. An Android Room Database and Repository Tutorial.....	557
71.1 About the RoomDemo Project.....	557
71.2 Modifying the Build Configuration.....	557

Table of Contents

71.3 Building the Entity	557
71.4 Creating the Data Access Object.....	559
71.5 Adding the Room Database.....	560
71.6 Adding the Repository	561
71.7 Adding the ViewModel	564
71.8 Creating the Product Item Layout	565
71.9 Adding the RecyclerView Adapter.....	565
71.10 Preparing the Main Activity	566
71.11 Adding the Button Listeners.....	567
71.12 Adding LiveData Observers	568
71.13 Initializing the RecyclerView.....	569
71.14 Testing the RoomDemo App.....	569
71.15 Using the Database Inspector.....	570
71.16 Summary	570
72. Video Playback on Android using the VideoView and MediaController Classes.....	571
72.1 Introducing the Android VideoView Class	571
72.2 Introducing the Android MediaController Class	572
72.3 Creating the Video Playback Example	572
72.4 Designing the VideoPlayer Layout	572
72.5 Downloading the Video File.....	573
72.6 Configuring the VideoView.....	573
72.7 Adding the MediaController to the Video View.....	575
72.8 Setting up the onPreparedListener	575
72.9 Summary	576
73. Android Picture-in-Picture Mode.....	577
73.1 Picture-in-Picture Features.....	577
73.2 Enabling Picture-in-Picture Mode.....	578
73.3 Configuring Picture-in-Picture Parameters	578
73.4 Entering Picture-in-Picture Mode	579
73.5 Detecting Picture-in-Picture Mode Changes.....	579
73.6 Adding Picture-in-Picture Actions.....	579
73.7 Summary	580
74. An Android Picture-in-Picture Tutorial.....	581
74.1 Adding Picture-in-Picture Support to the Manifest.....	581
74.2 Adding a Picture-in-Picture Button	581
74.3 Entering Picture-in-Picture Mode	581
74.4 Detecting Picture-in-Picture Mode Changes.....	583
74.5 Adding a Broadcast Receiver.....	584
74.6 Adding the PiP Action.....	585
74.7 Testing the Picture-in-Picture Action	587
74.8 Summary	588
75. Making Runtime Permission Requests in Android.....	589
75.1 Understanding Normal and Dangerous Permissions.....	589
75.2 Creating the Permissions Example Project.....	591
75.3 Checking for a Permission	591
75.4 Requesting Permission at Runtime.....	593
75.5 Providing a Rationale for the Permission Request	594

75.6 Testing the Permissions App.....	595
75.7 Summary	596
76. Android Audio Recording and Playback using MediaPlayer and MediaRecorder	597
76.1 Playing Audio	597
76.2 Recording Audio and Video using the MediaRecorder Class.....	598
76.3 About the Example Project	599
76.4 Creating the AudioApp Project.....	599
76.5 Designing the User Interface	599
76.6 Checking for Microphone Availability	600
76.7 Initializing the Activity.....	601
76.8 Implementing the recordAudio() Method.....	602
76.9 Implementing the stopAudio() Method.....	602
76.10 Implementing the playAudio() method.....	603
76.11 Configuring and Requesting Permissions	603
76.12 Testing the Application.....	605
76.13 Summary	605
77. An Android Notifications Tutorial	607
77.1 An Overview of Notifications.....	607
77.2 Creating the NotifyDemo Project.....	609
77.3 Designing the User Interface	609
77.4 Creating the Second Activity	609
77.5 Creating a Notification Channel	610
77.6 Requesting Notification Permission	611
77.7 Creating and Issuing a Notification	614
77.8 Launching an Activity from a Notification.....	616
77.9 Adding Actions to a Notification	618
77.10 Bundled Notifications.....	618
77.11 Summary	620
78. An Android Direct Reply Notification Tutorial	621
78.1 Creating the DirectReply Project.....	621
78.2 Designing the User Interface	621
78.3 Requesting Notification Permission	622
78.4 Creating the Notification Channel.....	623
78.5 Building the RemoteInput Object.....	624
78.6 Creating the PendingIntent.....	625
78.7 Creating the Reply Action.....	626
78.8 Receiving Direct Reply Input.....	627
78.9 Updating the Notification	628
78.10 Summary	629
79. Working with the Google Maps Android API in Android Studio	631
79.1 The Elements of the Google Maps Android API	631
79.2 Creating the Google Maps Project.....	632
79.3 Creating a Google Cloud Billing Account	632
79.4 Creating a New Google Cloud Project	633
79.5 Enabling the Google Maps SDK.....	634
79.6 Generating a Google Maps API Key.....	635
79.7 Adding the API Key to the Android Studio Project.....	636

Table of Contents

79.8 Testing the Application.....	637
79.9 Understanding Geocoding and Reverse Geocoding	637
79.10 Adding a Map to an Application.....	638
79.11 Requesting Current Location Permission.....	639
79.12 Displaying the User's Current Location	640
79.13 Changing the Map Type.....	641
79.14 Displaying Map Controls to the User.....	642
79.15 Handling Map Gesture Interaction.....	643
79.15.1 Map Zooming Gestures.....	643
79.15.2 Map Scrolling/Panning Gestures	643
79.15.3 Map Tilt Gestures.....	643
79.15.4 Map Rotation Gestures.....	644
79.16 Creating Map Markers.....	644
79.17 Controlling the Map Camera	645
79.18 Summary.....	646
80. Printing with the Android Printing Framework	647
80.1 The Android Printing Architecture	647
80.2 The Print Service Plugins	647
80.3 Google Cloud Print.....	648
80.4 Printing to Google Drive.....	648
80.5 Save as PDF	649
80.6 Printing from Android Devices	649
80.7 Options for Building Print Support into Android Apps.....	650
80.7.1 Image Printing.....	650
80.7.2 Creating and Printing HTML Content	651
80.7.3 Printing a Web Page.....	652
80.7.4 Printing a Custom Document	653
80.8 Summary	653
81. An Android HTML and Web Content Printing Example	655
81.1 Creating the HTML Printing Example Application	655
81.2 Printing Dynamic HTML Content	655
81.3 Creating the Web Page Printing Example.....	658
81.4 Removing the Floating Action Button	658
81.5 Removing Navigation Features.....	658
81.6 Designing the User Interface Layout	659
81.7 Accessing the WebView from the Main Activity	660
81.8 Loading the Web Page into the WebView.....	660
81.9 Adding the Print Menu Option.....	661
81.10 Summary.....	663
82. A Guide to Android Custom Document Printing.....	665
82.1 An Overview of Android Custom Document Printing	665
82.1.1 Custom Print Adapters.....	665
82.2 Preparing the Custom Document Printing Project.....	666
82.3 Creating the Custom Print Adapter.....	667
82.4 Implementing the onLayout() Callback Method	668
82.5 Implementing the onWrite() Callback Method	671
82.6 Checking a Page is in Range	673
82.7 Drawing the Content on the Page Canvas	674

82.8 Starting the Print Job	676
82.9 Testing the Application.....	677
82.10 Summary	677
83. An Introduction to Android App Links.....	679
83.1 An Overview of Android App Links	679
83.2 App Link Intent Filters	679
83.3 Handling App Link Intents	680
83.4 Associating the App with a Website.....	680
83.5 Summary	681
84. An Android Studio App Links Tutorial	683
84.1 About the Example App	683
84.2 The Database Schema	683
84.3 Loading and Running the Project.....	684
84.4 Adding the URL Mapping.....	685
84.5 Adding the Intent Filter.....	688
84.6 Adding Intent Handling Code.....	688
84.7 Testing the App.....	691
84.8 Creating the Digital Asset Links File	691
84.9 Testing the App Link.....	692
84.10 Summary	692
85. An Android Biometric Authentication Tutorial.....	693
85.1 An Overview of Biometric Authentication.....	693
85.2 Creating the Biometric Authentication Project	693
85.3 Configuring Device Fingerprint Authentication	694
85.4 Adding the Biometric Permission to the Manifest File.....	694
85.5 Designing the User Interface	695
85.6 Adding a Toast Convenience Method.....	695
85.7 Checking the Security Settings.....	696
85.8 Configuring the Authentication Callbacks.....	697
85.9 Adding the CancellationSignal.....	698
85.10 Starting the Biometric Prompt	698
85.11 Testing the Project.....	699
85.12 Summary	700
86. Creating, Testing and Uploading an Android App Bundle.....	701
86.1 The Release Preparation Process	701
86.2 Android App Bundles.....	701
86.3 Register for a Google Play Developer Console Account.....	702
86.4 Configuring the App in the Console	703
86.5 Enabling Google Play App Signing.....	704
86.6 Creating a Keystore File	704
86.7 Creating the Android App Bundle.....	706
86.8 Generating Test APK Files	707
86.9 Uploading the App Bundle to the Google Play Developer Console.....	708
86.10 Exploring the App Bundle	709
86.11 Managing Testers	710
86.12 Rolling the App Out for Testing.....	710
86.13 Uploading New App Bundle Revisions.....	711

86.14 Analyzing the App Bundle File	712
86.15 Summary	713
87. An Overview of Android In-App Billing	715
87.1 Preparing a Project for In-App Purchasing	715
87.2 Creating In-App Products and Subscriptions	715
87.3 Billing Client Initialization	716
87.4 Connecting to the Google Play Billing Library	717
87.5 Querying Available Products	717
87.6 Starting the Purchase Process	718
87.7 Completing the Purchase	718
87.8 Querying Previous Purchases	719
87.9 Summary	720
88. An Android In-App Purchasing Tutorial	721
88.1 About the In-App Purchasing Example Project	721
88.2 Creating the InAppPurchase Project	721
88.3 Adding Libraries to the Project	721
88.4 Designing the User Interface	722
88.5 Adding the App to the Google Play Store	722
88.6 Creating an In-App Product	723
88.7 Enabling License Testers	723
88.8 Initializing the Billing Client	724
88.9 Querying the Product	726
88.10 Launching the Purchase Flow	727
88.11 Handling Purchase Updates	727
88.12 Consuming the Product	728
88.13 Restoring a Previous Purchase	729
88.14 Testing the App	730
88.15 Troubleshooting	731
88.16 Summary	732
89. An Overview of Android Dynamic Feature Modules	733
89.1 An Overview of Dynamic Feature Modules	733
89.2 Dynamic Feature Module Architecture	733
89.3 Creating a Dynamic Feature Module	734
89.4 Converting an Existing Module for Dynamic Delivery	736
89.5 Working with Dynamic Feature Modules	739
89.6 Handling Large Dynamic Feature Modules	740
89.7 Summary	741
90. An Android Studio Dynamic Feature Tutorial	743
90.1 Creating the DynamicFeature Project	743
90.2 Adding Dynamic Feature Support to the Project	743
90.3 Designing the Base Activity User Interface	744
90.4 Adding the Dynamic Feature Module	745
90.5 Reviewing the Dynamic Feature Module	746
90.6 Adding the Dynamic Feature Activity	747
90.7 Implementing the launchIntent() Method	750
90.8 Uploading the App Bundle for Testing	751
90.9 Implementing the installFeature() Method	752

90.10 Adding the Update Listener.....	753
90.11 Using Deferred Installation	756
90.12 Removing a Dynamic Module	757
90.13 Summary	757
91. Working with Material Design 3 Theming	759
91.1 Material Design 2 vs Material Design 3	759
91.2 Understanding Material Design Theming	759
91.3 Material Design 3 Theming	759
91.4 Building a Custom Theme.....	761
91.5 Summary	762
92. A Material Design 3 Theming and Dynamic Color Tutorial.....	763
92.1 Creating the ThemeDemo Project	763
92.2 Designing the User Interface	763
92.3 Building a New Theme	765
92.4 Adding the Theme to the Project.....	766
92.5 Enabling Dynamic Color Support	767
92.6 Summary	768
93. An Overview of Gradle in Android Studio.....	769
93.1 An Overview of Gradle	769
93.2 Gradle and Android Studio	769
93.2.1 Sensible Defaults	769
93.2.2 Dependencies.....	769
93.2.3 Build Variants	770
93.2.4 Manifest Entries	770
93.2.5 APK Signing.....	770
93.2.6 ProGuard Support.....	770
93.3 The Property and Settings Gradle Build File.....	770
93.4 The Top-level Gradle Build File.....	771
93.5 Module Level Gradle Build Files.....	772
93.6 Configuring Signing Settings in the Build File.....	774
93.7 Running Gradle Tasks from the Command-line	775
93.8 Summary	776
Index.....	777

1. Introduction

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

This book begins with the basics and outlines how to set up an Android development and testing environment, followed by an introduction to programming in Kotlin, including data types, control flow, functions, lambdas, and object-oriented programming. Asynchronous programming using Kotlin coroutines and flow is also covered in detail.

An overview of Android Studio covers areas such as tool windows, the code editor, and the Layout Editor tool. An introduction to the architecture of Android is followed by an in-depth look at the design of Android applications and user interfaces using the Android Studio environment.

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

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

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

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

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

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

1.1 Downloading the Code Samples

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

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

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

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

1.2 Feedback

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

1.3 Errata

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

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

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

2. Setting up an Android Studio Development Environment

Before any work can begin on the development of an Android application, the first step is to configure a computer system to act as the development platform. This involves several steps consisting of installing the Android Studio Integrated Development Environment (IDE) which also includes the Android Software Development Kit (SDK), 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 8/10/11 64-bit
- macOS 10.14 or later running on Intel or Apple silicon
- Chrome OS device with Intel i5 or higher
- Linux systems with version 2.31 or later of the GNU C Library (glibc)
- Minimum of 8GB of RAM
- Approximately 8GB of available disk space
- 1280 x 800 minimum screen resolution

2.2 Downloading the Android Studio package

Most of the work involved in developing applications for Android will be performed using the Android Studio environment. The content and examples in this book were created based on Android Studio Flamingo 2022.2.1 using the Android API 33 SDK (Tiramisu) which, at the time of writing, are the latest versions.

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

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

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

If this page provides instructions for downloading a newer version of Android Studio it is important to note that there may be some minor differences between this book and the software. A web search for “Android Studio Flamingo” should provide the option to download the older version if these differences become a problem. Alternatively, visit the following web page to find Android Studio Flamingo 2022.2.1 in the archives:

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

2.3 Installing Android Studio

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

2.3.1 Installation on Windows

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

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

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

On versions of Windows with a Start menu, the newly installed Android Studio can be launched from the entry added to that menu during the installation. The executable may be pinned to the taskbar for easy access by navigating to the *Android Studio\bin* directory, right-clicking on the *studio64* executable, and selecting the *Pin to Taskbar* menu option (on Windows 11 this option can be found by selecting *Show more options* from the menu).

2.3.2 Installation on macOS

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

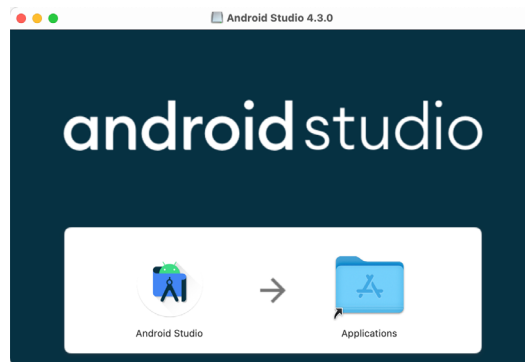


Figure 2-1

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

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

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

2.3.3 Installation on Linux

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

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

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

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

```
./studio.sh
```

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

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

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

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

2.4 The Android Studio setup wizard

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

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

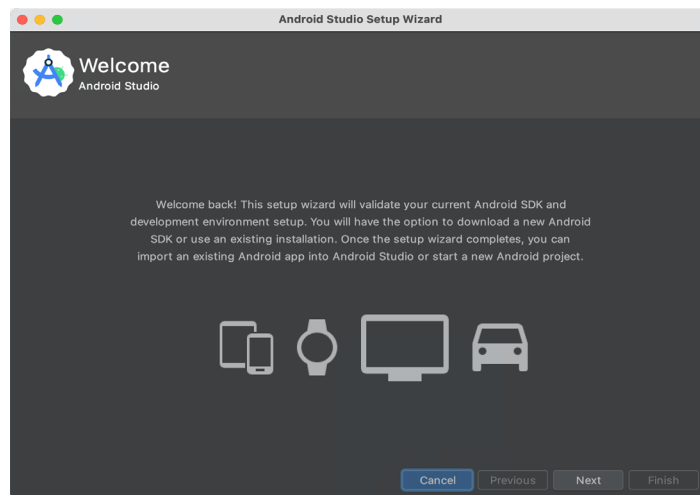


Figure 2-2

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

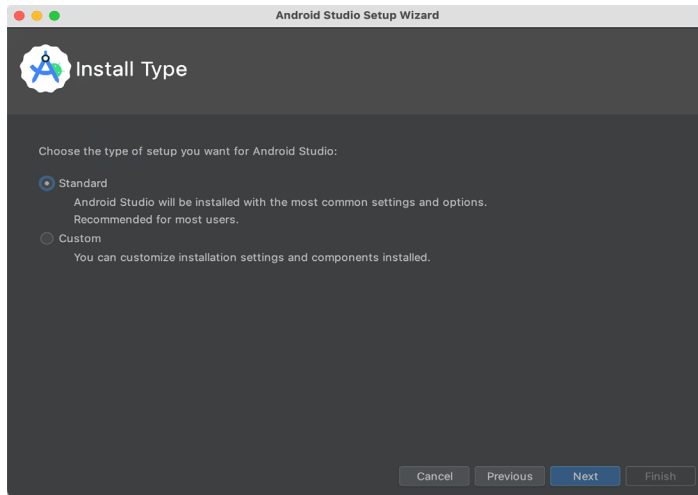


Figure 2-3

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

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

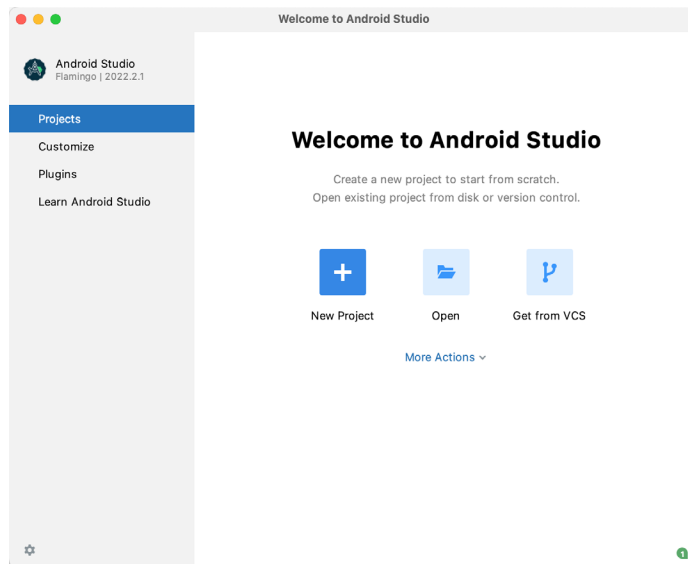


Figure 2-4

2.5 Installing additional Android SDK packages

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

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

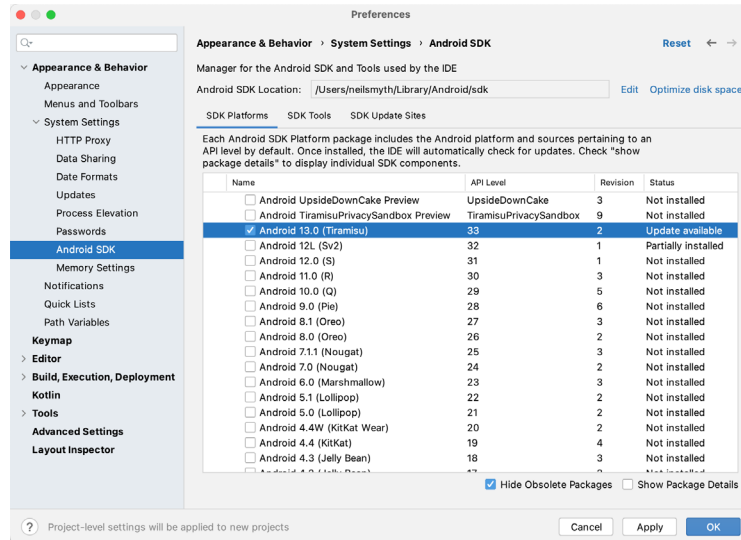


Figure 2-5

Immediately after installing Android Studio for the first time it is likely that only the latest released version of the Android SDK has been installed. To install older versions of the Android SDK simply select the checkboxes corresponding to the versions and click on the *Apply* button. The rest of this book assumes that the Android Tiramisu (API Level 33) SDK is installed.

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

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

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

Figure 2-6

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

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

Setting up an Android Studio Development Environment

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

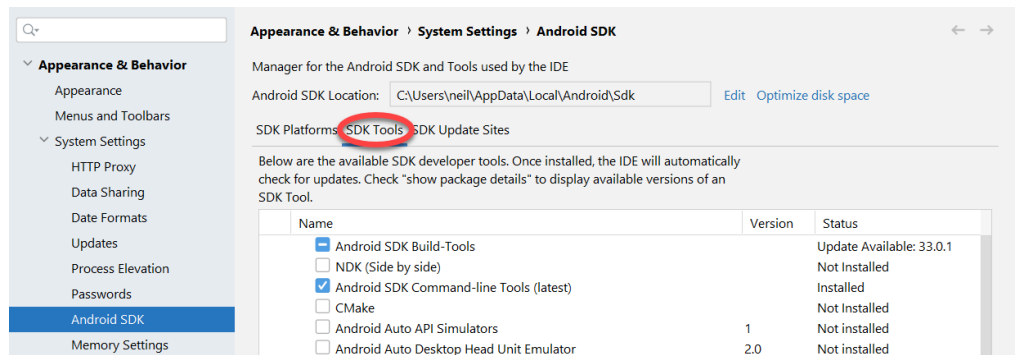


Figure 2-7

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

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

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

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

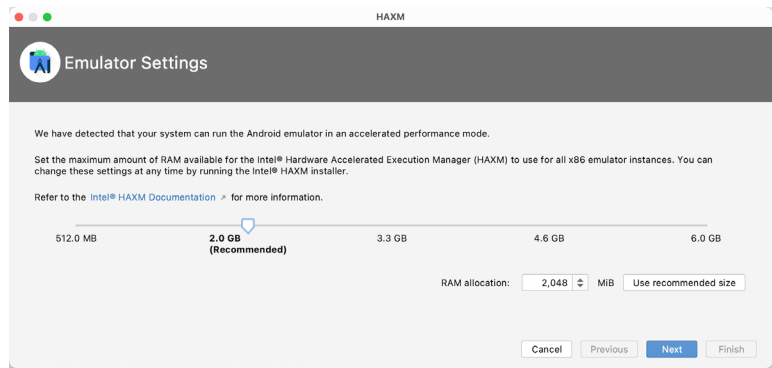


Figure 2-8

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

Apply button again.

2.6 Installing the Android SDK Command-line Tools

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

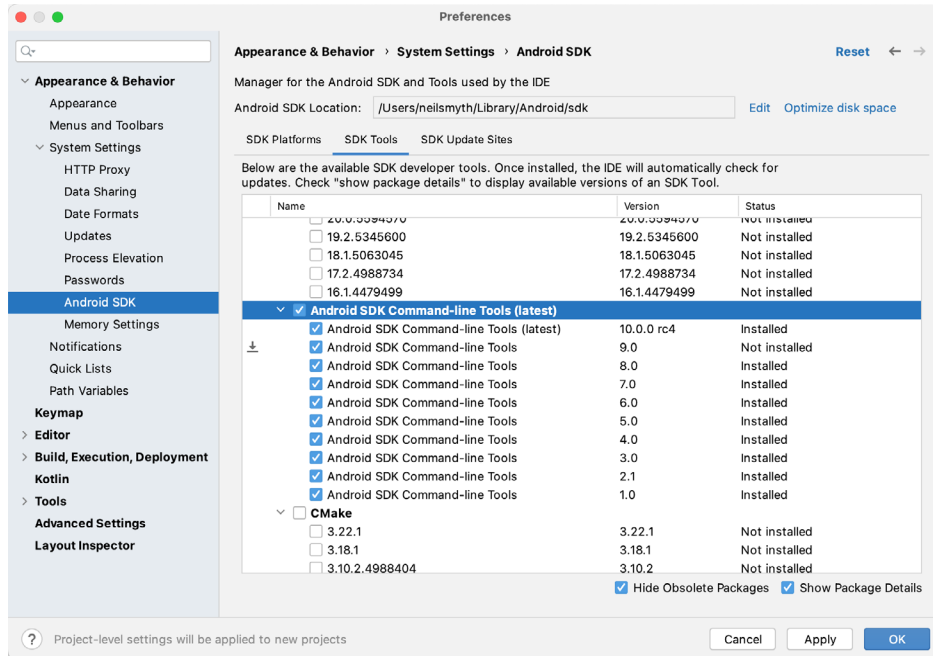


Figure 2-9

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

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

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

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

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

Setting up an Android Studio Development Environment

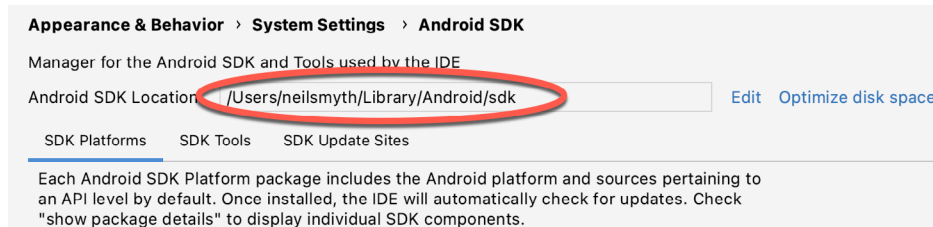


Figure 2-10

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

2.6.1 Windows 8.1

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

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

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

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

```
echo %Path%
```

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

```
adb
```

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

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

```
avdmanager
```

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

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

2.6.2 Windows 10

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

2.6.3 Windows 11

Right-click on the Start icon located in the taskbar and select Settings from the resulting menu. When the Settings dialog appears, scroll down the list of categories and select the “About” option. In the About screen, select *Advanced system settings* from the Related links section. When the System Properties window appears, click on the *Environment Variables...* button. Follow the steps outlined for Windows 8.1 starting from step 3.

2.6.4 Linux

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

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

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

2.6.5 macOS

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

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

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

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

2.7 Android Studio memory management

Android Studio is a large and complex software application that consists of many background processes. Although Android Studio has been criticized in the past for providing less than optimal performance, Google has made significant performance improvements in recent releases and continues to do so with each new version. These improvements include allowing the user to configure the amount of memory used by both the Android Studio IDE and the background processes used to build and run apps. This allows the software to take advantage of systems with larger amounts of RAM.

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

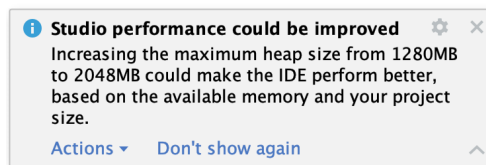


Figure 2-11

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

Setting up an Android Studio Development Environment

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

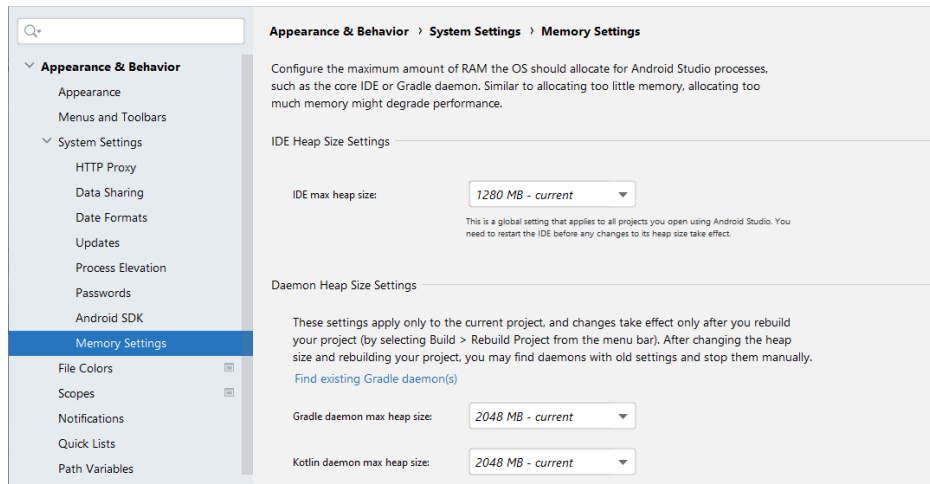


Figure 2-12

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

The IDE heap size setting adjusts the memory allocated to Android Studio and applies regardless of the currently loaded project. On the other hand, when a project is built and run from within Android Studio, a number of background processes (referred to as daemons) perform the task of compiling and running the app. When compiling and running large and complex projects, build time could be improved by adjusting the daemon heap settings. Unlike the IDE heap settings, these daemon settings apply only to the current project and can only be accessed when a project is open in Android Studio. To display the SDK Manager from within an open project, select the *Tools -> SDK Manager...* menu option.

2.8 Updating Android Studio and the SDK

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

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

2.9 Summary

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

3. Creating an Example Android App in Android Studio

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

3.1 About the Project

The project created in this chapter takes the form of a rudimentary currency conversion calculator (so simple, in fact, that it only converts from dollars to euros and does so using an estimated conversion rate). The project will also make use of one of the most basic of Android Studio project templates. This simplicity allows us to introduce some of the key aspects of Android app development without overwhelming the beginner by trying to introduce too many concepts, such as the recommended app architecture and Android architecture components, at once. When following the tutorial in this chapter, rest assured that all of the techniques and code used in this initial example project will be covered in much greater detail in later chapters.

3.2 Creating a New Android Project

The first step in the application development process is to create a new project within the Android Studio environment. Begin, therefore, by launching Android Studio so that the “Welcome to Android Studio” screen appears as illustrated in Figure 3-1:

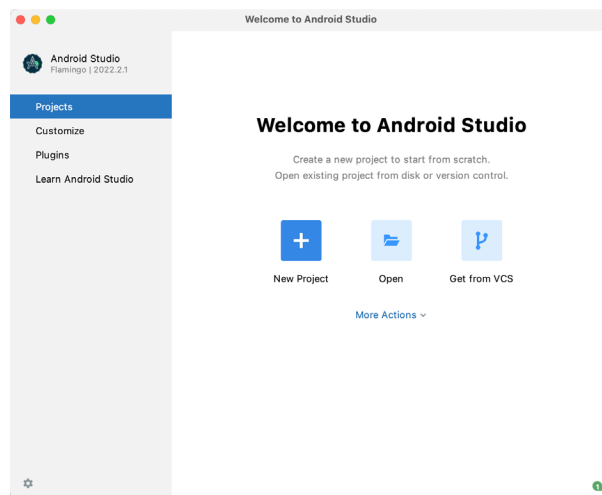


Figure 3-1

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

Creating an Example Android App in Android Studio

simply click on the *New Project* option to display the first screen of the *New Project* wizard.

3.3 Creating an Activity

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

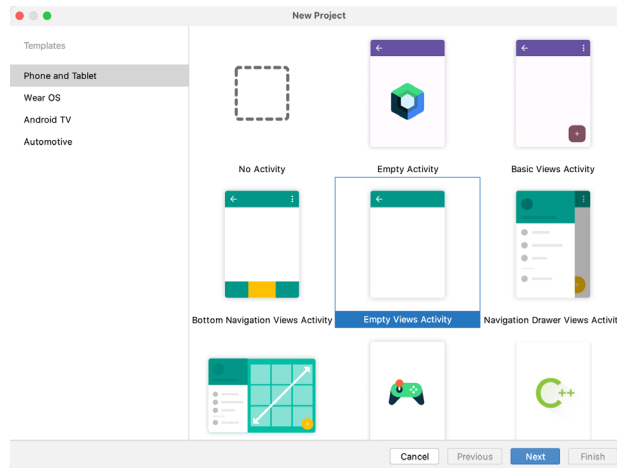


Figure 3-2

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

3.4 Defining the Project and SDK Settings

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

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

```
com.mycompany.androidsample
```

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

```
com.example.androidsample
```

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

Set the minimum SDK setting to API 26: Android 8.0 (Oreo). This is the minimum SDK that will be used in

most of the projects created in this book unless a necessary feature is only available in a more recent version. The objective here is to build an app using the latest Android SDK, while also retaining compatibility with devices running older versions of Android (in this case as far back as Android 8.0). The text beneath the Minimum SDK setting will outline the percentage of Android devices currently in use on which the app will run. Click on the *Help me choose* button (highlighted in Figure 3-3) to see a full breakdown of the various Android versions still in use:

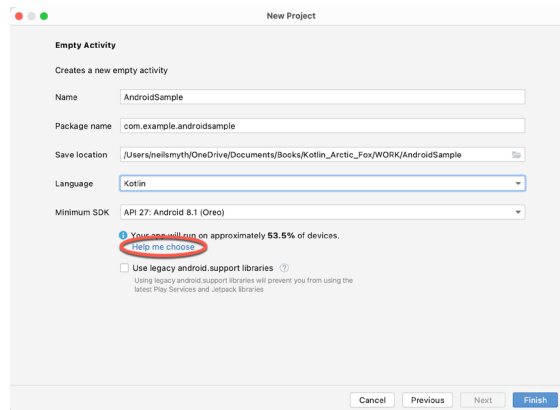


Figure 3-3

Finally, change the *Language* menu to *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.

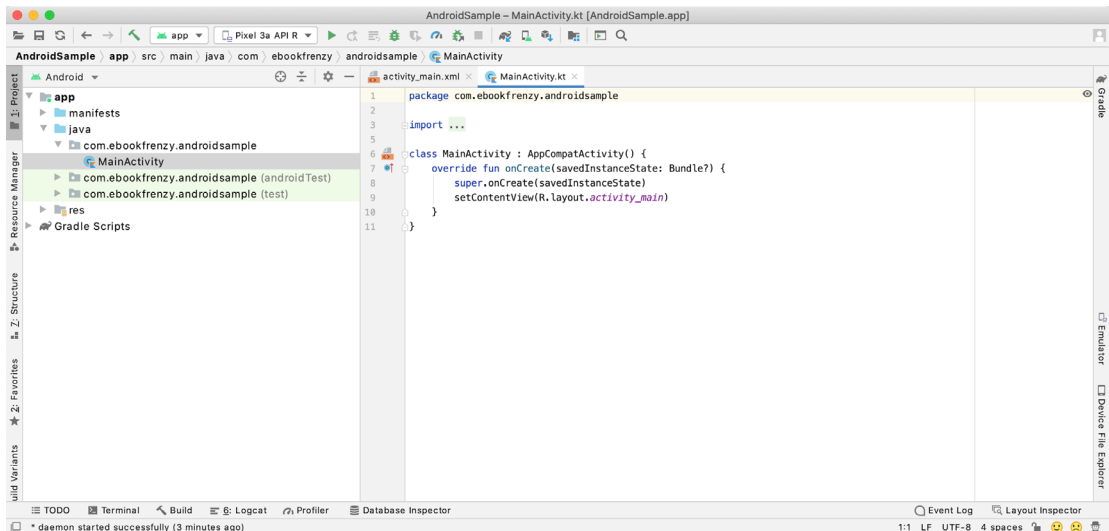


Figure 3-4

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

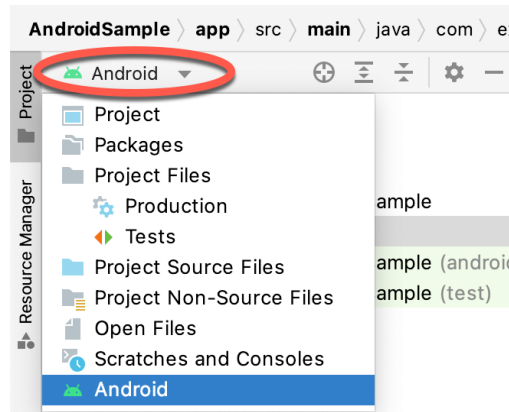


Figure 3-5

3.6 Modifying the User Interface

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

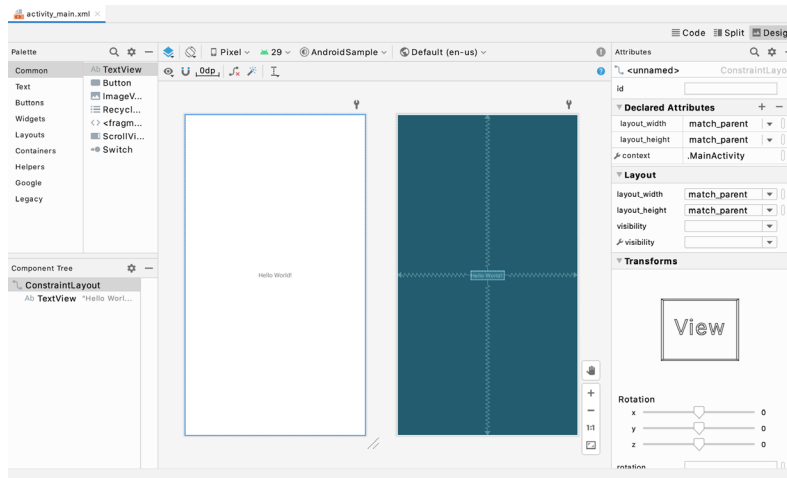




Figure 3-6

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

To change the orientation of the device representation between landscape and portrait simply use the drop down menu immediately to the left of the device selection menu showing the  icon. Use the night mode button () to turn Night mode on and off.

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

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

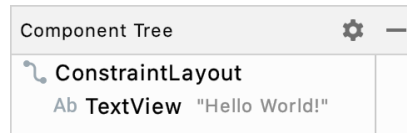


Figure 3-7

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

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



Figure 3-8

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

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

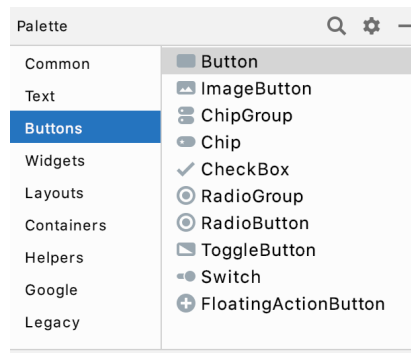


Figure 3-9

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

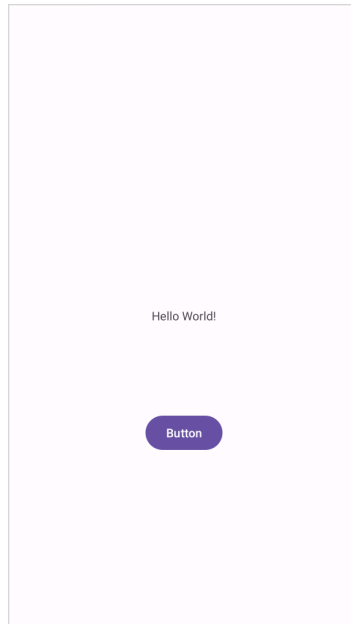


Figure 3-10

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

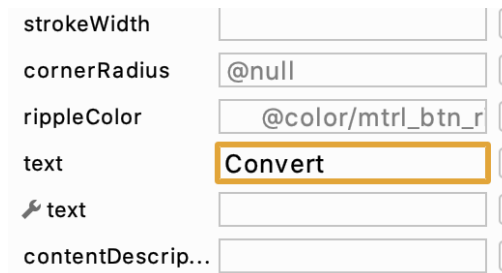


Figure 3-11

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

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

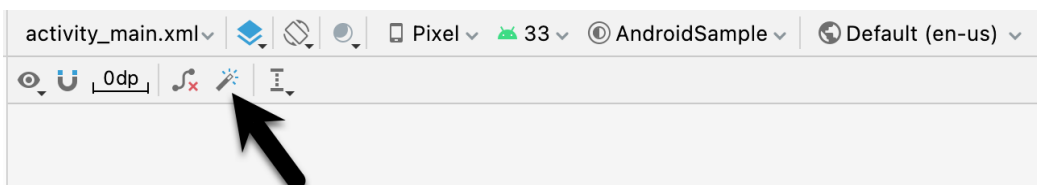


Figure 3-12

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

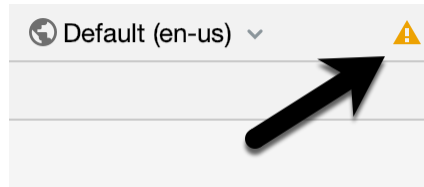


Figure 3-13

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

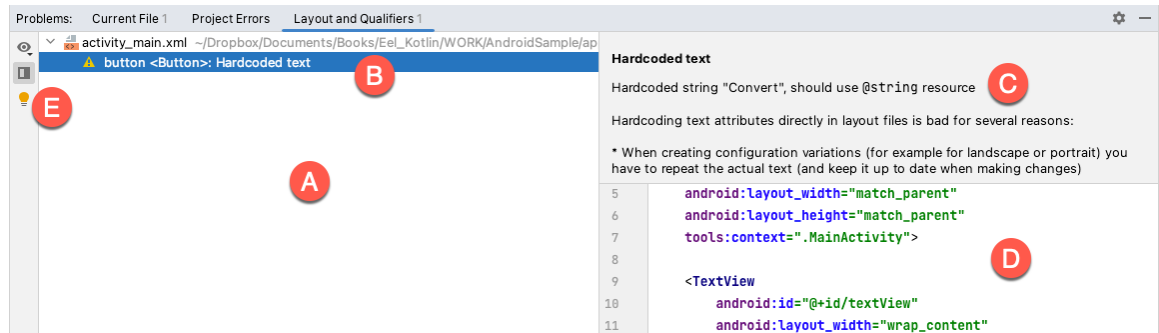


Figure 3-14

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

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

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

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

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

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

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

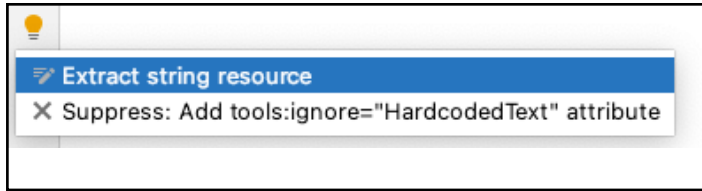


Figure 3-15

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

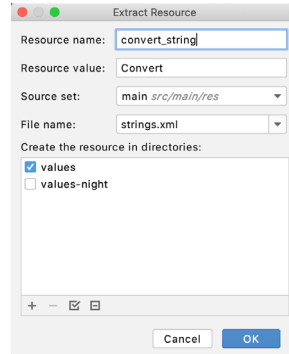


Figure 3-16

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

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

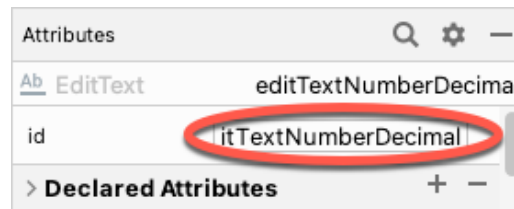


Figure 3-17

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

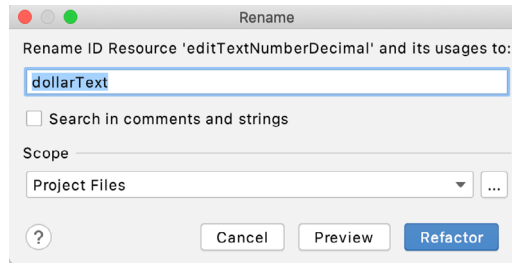


Figure 3-18

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

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

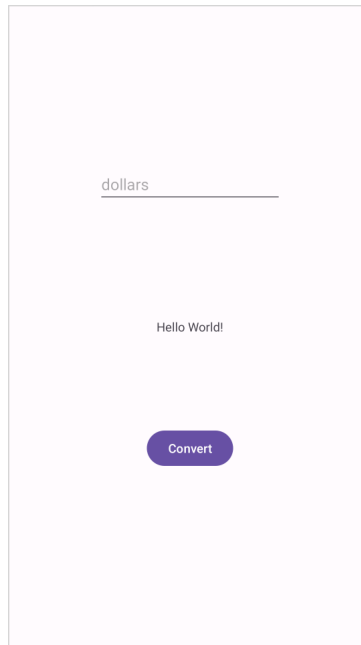


Figure 3-19

3.7 Reviewing the Layout and Resource Files

Before moving on to the next step, we are going to look at some of the internal aspects of user interface design and resource handling. In the previous section, we made some changes to the user interface by modifying the *activity_main.xml* file using the Layout Editor tool. In fact, all that the Layout Editor was doing was providing a user-friendly way to edit the underlying XML content of the file. In practice, there is no reason why you cannot modify the XML directly to make user interface changes and, in some instances, this may actually be quicker than using the Layout Editor tool. In the top right-hand corner of the Layout Editor panel are three buttons as highlighted in Figure 3-20 below:

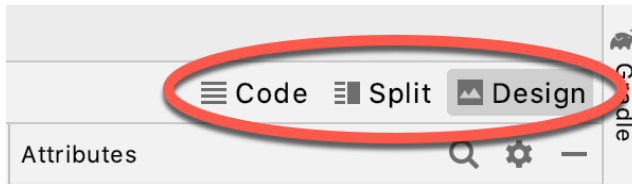


Figure 3-20

By default, the editor will be in *Design* mode whereby just the visual representation of the layout is displayed. The left-most button will switch to *Code* mode to display the XML for the layout, while the middle button enters *Split* mode where both the layout and XML are displayed, as shown in Figure 3-21:

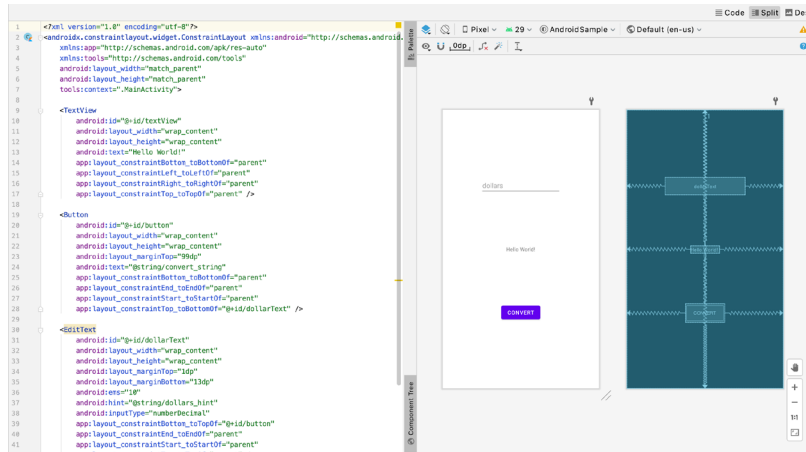


Figure 3-21

As can be seen from the structure of the XML file, the user interface consists of the `ConstraintLayout` component, which in turn, is the parent of the `TextView`, `Button` and `EditText` objects. We can also see, for example, that the `text` property of the `Button` is set to our `convert_string` resource. Although varying in complexity and content, all user interface layouts are structured in this hierarchical, XML based way.

As changes are made to the XML layout, these will be reflected in the layout canvas. The layout may also be modified visually from within the layout canvas panel with the changes appearing in the XML listing. To see this in action, switch to Split mode and modify the XML layout to change the background color of the `ConstraintLayout` to a shade of red as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.
android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity"
    android:background="#ff2438" >
    .
    .
</androidx.constraintlayout.widget.ConstraintLayout>
```

Note that the color of the layout changes in real-time to match the new setting in the XML file. Note also that a small red square appears in the left-hand margin (also referred to as the *gutter*) of the XML editor next to the line containing the color setting. This is a visual cue to the fact that the color red has been set on a property. Clicking on the red square will display a color chooser allowing a different color to be selected:

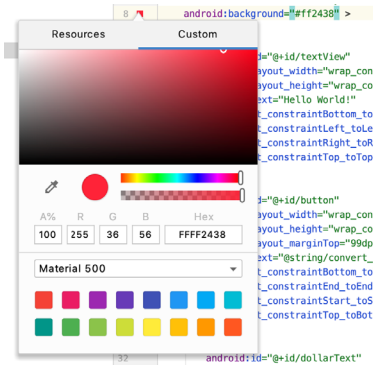


Figure 3-22

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

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

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

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

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

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

Resource strings may also be edited using the Android Studio Translations Editor by clicking on the *Open editor* link in the top right-hand corner of the editor window. This will display the Translation Editor in the main panel of the Android Studio window:

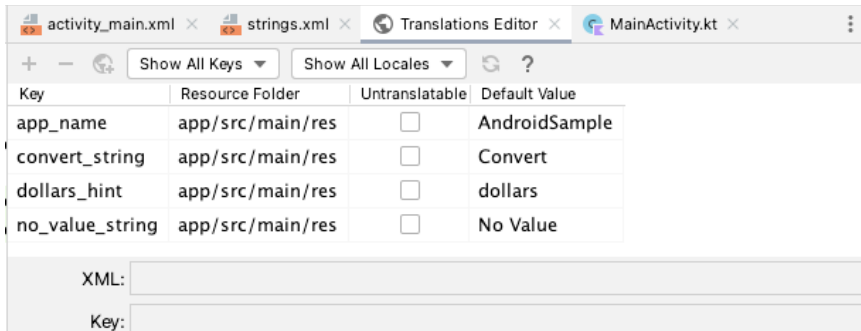


Figure 3-23

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

3.8 Adding Interaction

The final step in this example project is to make the app interactive so that when the user enters a dollar value into the EditText field and clicks the convert button the converted euro value appears on the TextView. This involves the implementation of some event handling on the Button widget. Specifically, the Button needs to be configured so that a method in the app code is called when an *onClick* event is triggered. Event handling can be implemented in a number of different ways and is covered in detail in a later chapter entitled “An Overview and Example of Android Event Handling”. Return the layout editor to Design mode, select the Button widget in the layout editor, refer to the Attributes tool window and specify a method named *convertCurrency* as shown below:

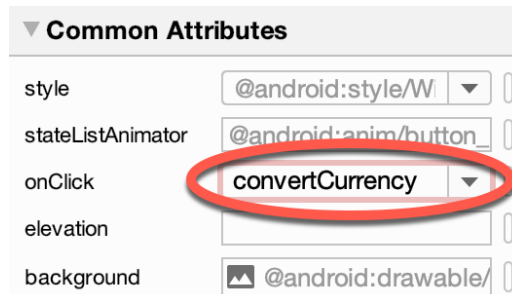


Figure 3-24

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

```
package com.example.androidsample

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.view.View
import android.widget.EditText
import android.widget.TextView
```

```

class MainActivity : AppCompatActivity() {

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

    fun convertCurrency(view: View) {

        val dollarText: EditText = findViewById(R.id.dollarText)
        val textView: TextView = findViewById(R.id.textView)

        if (dollarText.text.isNotEmpty()) {

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

            val euroValue = dollarValue * 0.85f

            textView.text = euroValue.toString()
        } else {
            textView.text = getString(R.string.no_value_string)
        }
    }
}

```

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

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

3.9 Summary

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

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

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

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

Although the Android Studio Preview panel allows us to see the layout we are designing and test basic functionality using interactive mode, it be will necessary to compile and run an entire app to fully test that it works. An Android application may be tested by installing and running it either on a physical device or in an *Android Virtual Device (AVD)* emulator environment. Before an AVD can be used, it must first be created and configured to match the specifications of a particular device model. In this chapter, we will work through the creation of such a virtual device using the Pixel 4 phone as a reference example.

4.1 About Android Virtual Devices

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

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

New AVDs are created and managed using the Android Virtual Device Manager, which may be used either in command-line mode or with a more user-friendly graphical user interface. To create a new AVD, the first step is to launch the AVD Manager. This can be achieved from within the Android Studio environment by selecting the *Tools -> Device Manager* menu option from within the main window.

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

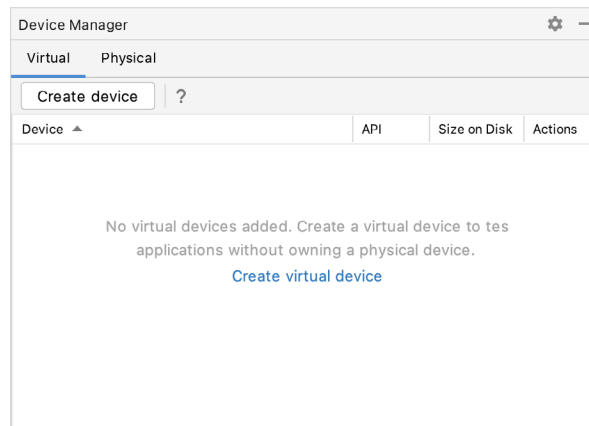


Figure 4-1

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

Creating an Android Virtual Device (AVD) in Android Studio

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

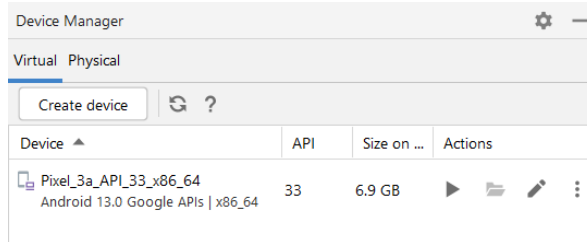


Figure 4-2

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

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

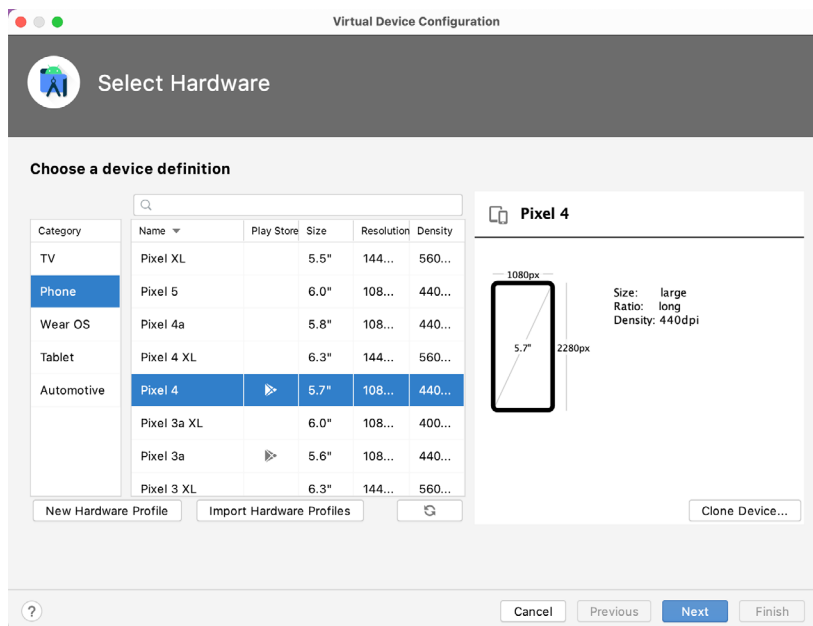


Figure 4-3

Within the dialog, perform the following steps to create a Pixel 4 compatible emulator:

1. From the *Category* panel, select the *Phone* option to display the list of available Android phone AVD templates.
2. Select the *Pixel 4* device option and click *Next*.
3. On the System Image screen, select the latest version of Android. Note that if the system image has not yet been installed a *Download* link will be provided next to the Release Name. Click this link to download and install the system image before selecting it. If the image you need is not listed, click on the *x86 Images* (or *ARM images* if you are running a Mac with Apple Silicon) and *Other images* tabs to view alternative lists.

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

4.2 Starting the Emulator

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

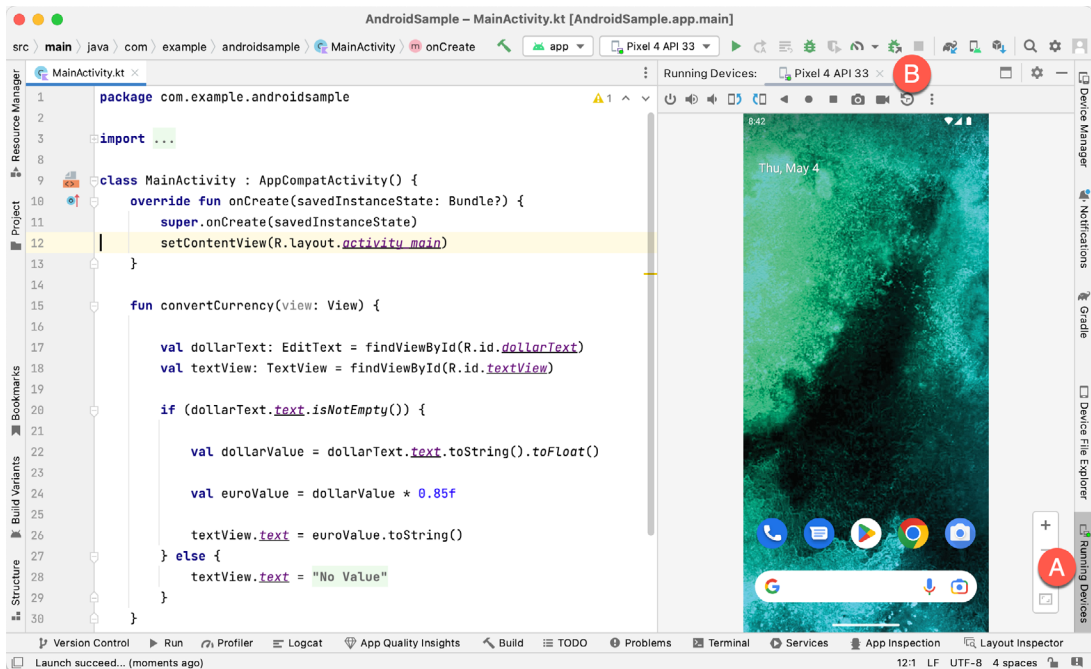


Figure 4-4

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

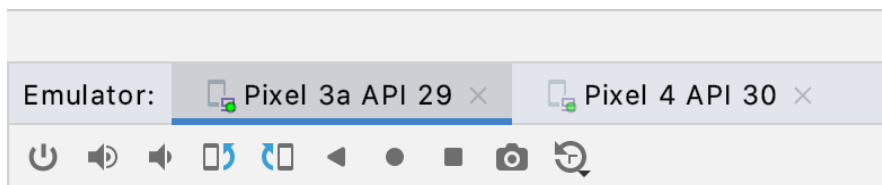


Figure 4-5

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

Creating an Android Virtual Device (AVD) in Android Studio

Although the emulator probably defaulted to appearing in portrait orientation, this and other default options can be changed. Within the Device Manager, select the new Pixel 4 entry and click on the pencil icon in the *Actions* column of the device row. In the configuration screen, locate the *Startup orientation* section and change the orientation setting. Exit and restart the emulator session to see this change take effect. More details on the emulator are covered in the next chapter “*Using and Configuring the Android Studio AVD Emulator*”).

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

4.3 Running the Application in the AVD

With an AVD emulator configured, the example AndroidSample application created in the earlier chapter now can be compiled and run. With the AndroidSample project loaded into Android Studio, make sure that the newly created Pixel 4 AVD is displayed in the device menu (marked A in Figure 4-6 below), then either click on the run button represented by a green triangle (B), select the *Run* -> *Run ‘app’* menu option or use the Ctrl-R keyboard shortcut:

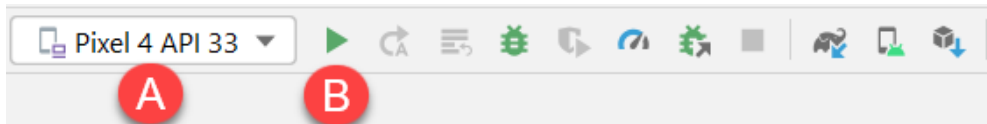


Figure 4-6

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

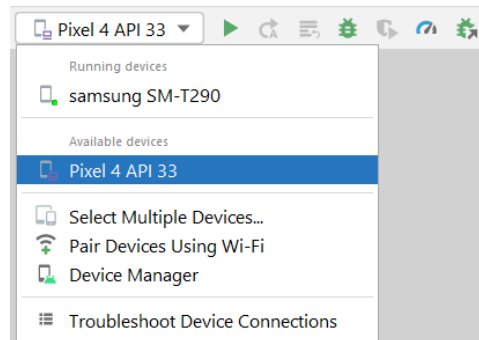


Figure 4-7

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

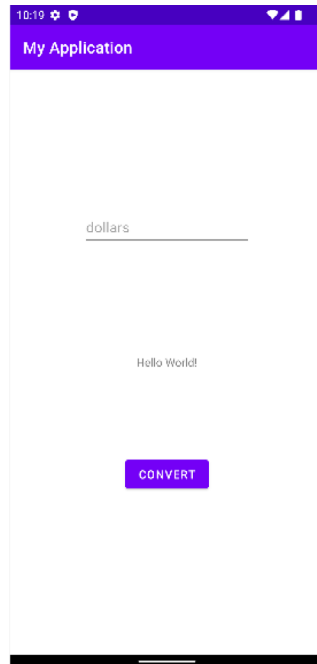


Figure 4-8

If the activity does not automatically launch, check to see if the launch icon has appeared among the apps on the emulator. If it has, simply click on it to launch the application. Once the run process begins, the Run tool window will become available. The Run tool window will display diagnostic information as the application package is installed and launched. Figure 4-9 shows the Run tool window output from a typical successful application launch:

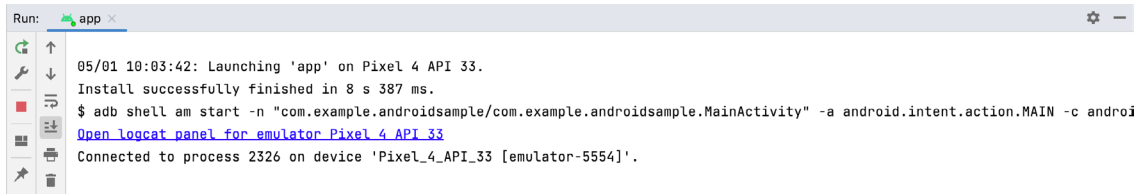


Figure 4-9

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

Assuming that the application loads into the emulator and runs as expected, we have safely verified that the Android development environment is correctly installed and configured. With the app now running, try performing a currency conversion to verify that the app works as intended.

4.4 Running on Multiple Devices

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

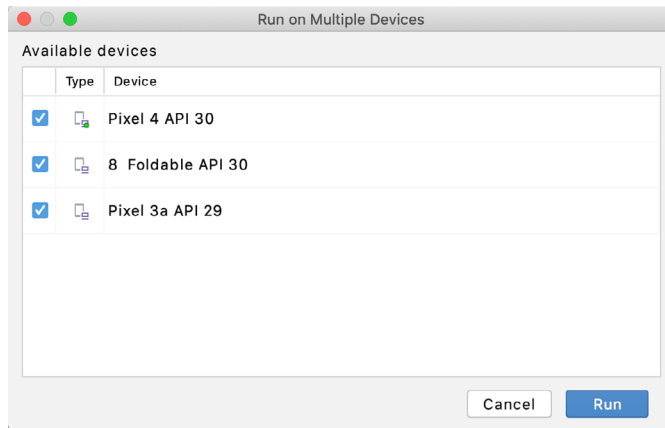


Figure 4-10

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

4.5 Stopping a Running Application

To stop a running application, simply click on the stop button located in the main toolbar as shown in Figure 4-11:

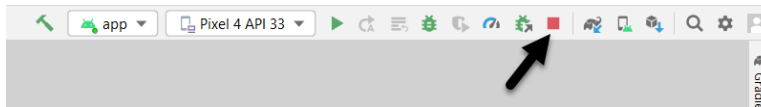


Figure 4-11

An app may also be terminated using the Run tool window. Begin by displaying the *Run* tool window using the window bar button that becomes available when the app is running. Once the Run tool window appears, click the stop button highlighted in Figure 4-12 below:

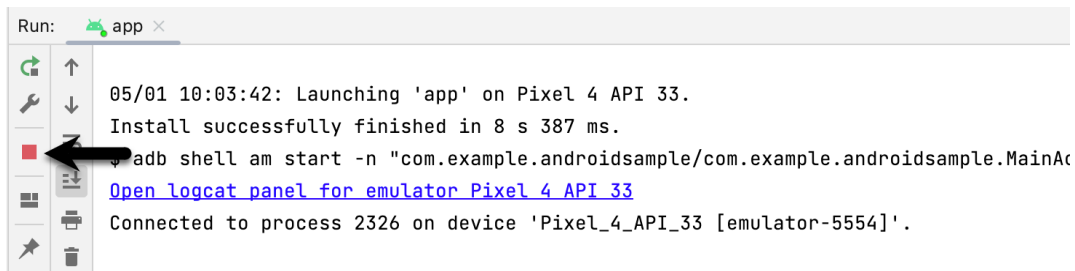


Figure 4-12

4.6 Supporting Dark Theme

Android 10 introduced the much-awaited dark theme, support for which is enabled by default in Android Studio app projects. To test dark theme in the AVD emulator, open the Settings app within the running Android instance in the emulator. Within the Settings app, choose the *Display* category and enable the *Dark theme* option as shown in Figure 4-13 so that the screen background turns black:

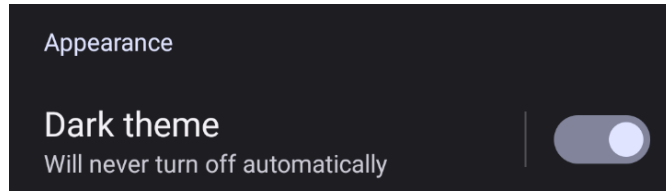


Figure 4-13

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

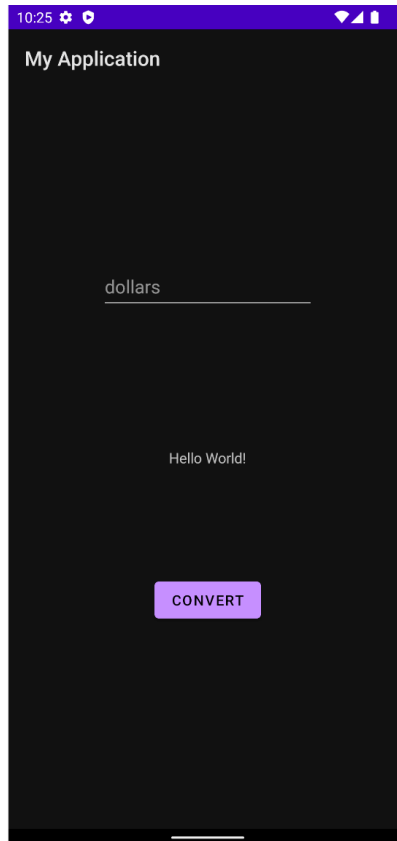


Figure 4-14

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

4.7 Running the Emulator in a Separate Window

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

Creating an Android Virtual Device (AVD) in Android Studio

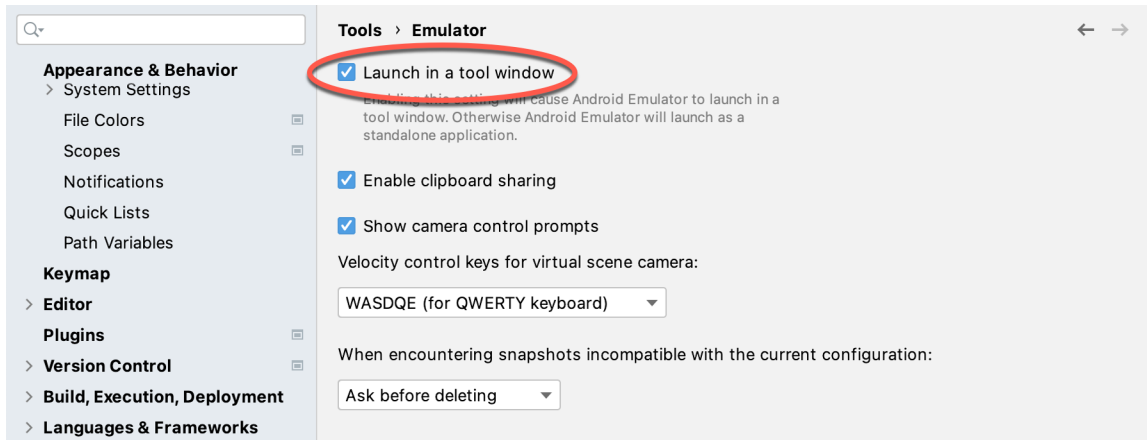


Figure 4-15

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

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

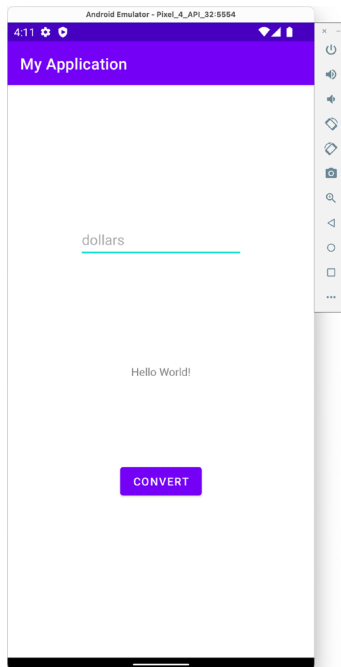


Figure 4-16

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

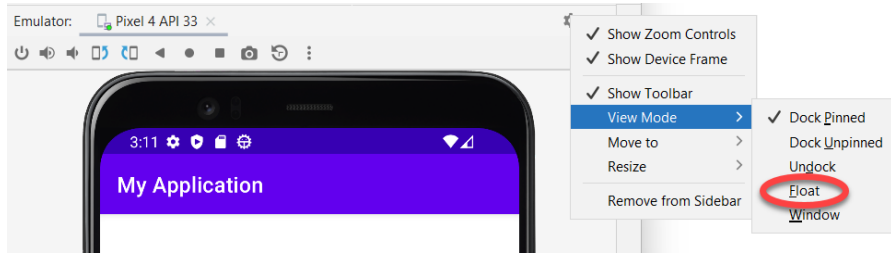


Figure 4-17

4.8 Enabling the Device Frame

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

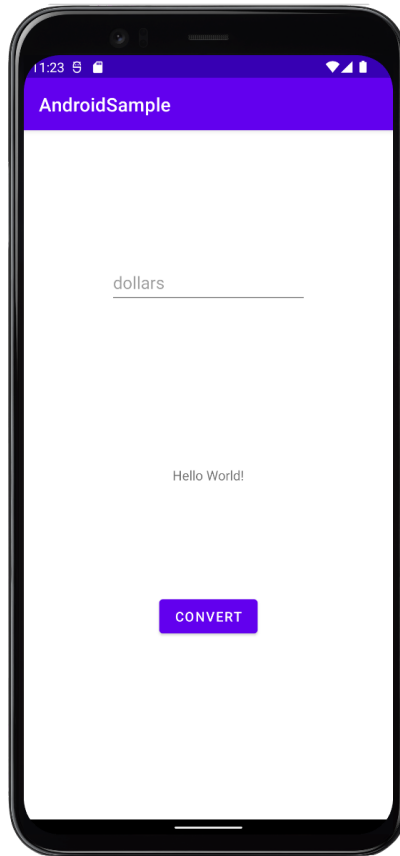


Figure 4-18

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

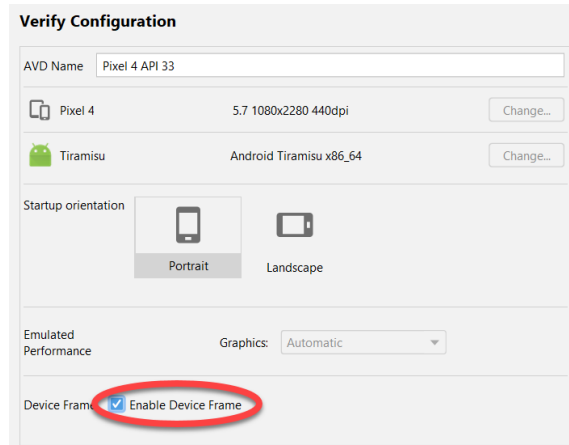


Figure 4-19

4.9 Summary

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

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

6. A Tour of the Android Studio User Interface

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

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

6.1 The Welcome Screen

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

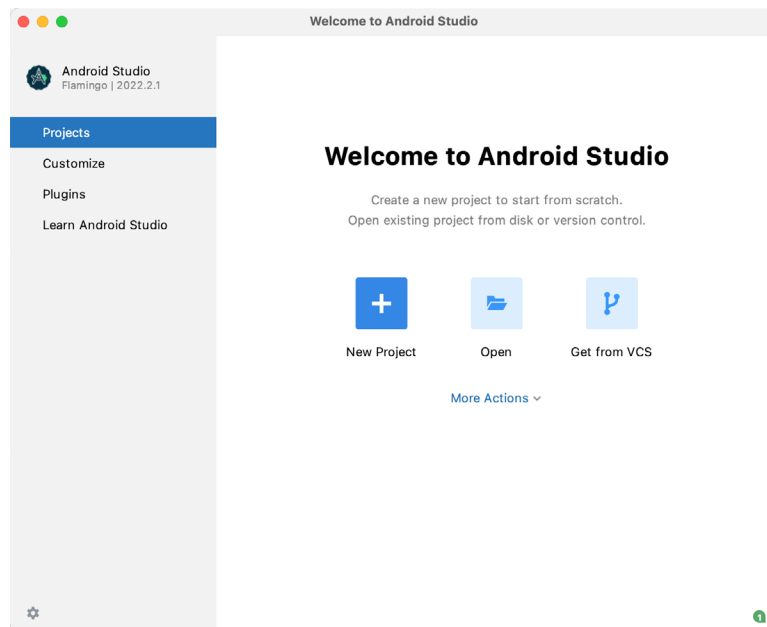


Figure 6-1

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

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

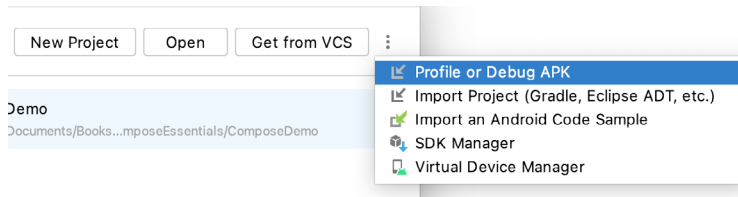


Figure 6-2

6.2 The Main Window

When a new project is created, or an existing one opened, the Android Studio *main window* will appear. When multiple projects are open simultaneously, each will be assigned its own main window. The precise configuration of the window will vary depending on which tools and panels were displayed the last time the project was open, but will typically resemble that of Figure 6-3.

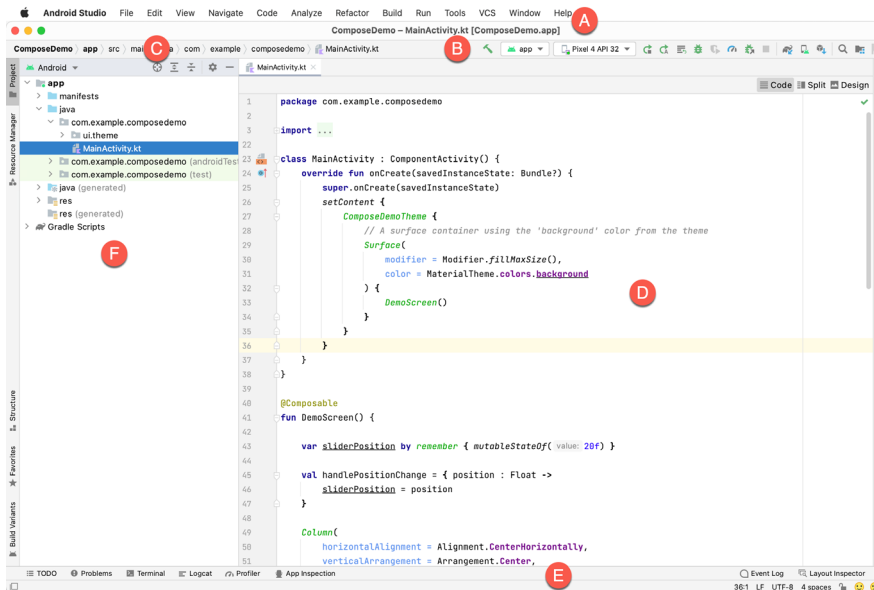


Figure 6-3

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

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

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

C – Navigation Bar – The navigation bar provides a convenient way to move around the files and folders that make up the project. Clicking on an element in the navigation bar will drop down a menu listing the sub-folders and files at that location ready for selection. Similarly, clicking on a class name displays a menu listing methods contained within that class. Select a method from the list to be taken to the corresponding location within the code editor. Hide and display this bar using the *View -> Appearance -> Navigation Bar* menu option.

D – Editor Window – The editor window displays the content of the file on which the developer is currently working. When multiple files are open, each file is represented by a tab located along the top edge of the editor as shown in Figure 6-4:



Figure 6-4

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

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

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

6.3 The Tool Windows

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

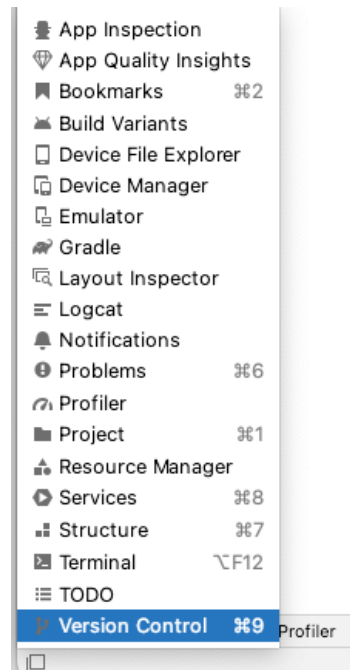


Figure 6-5

A Tour of the Android Studio User Interface

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

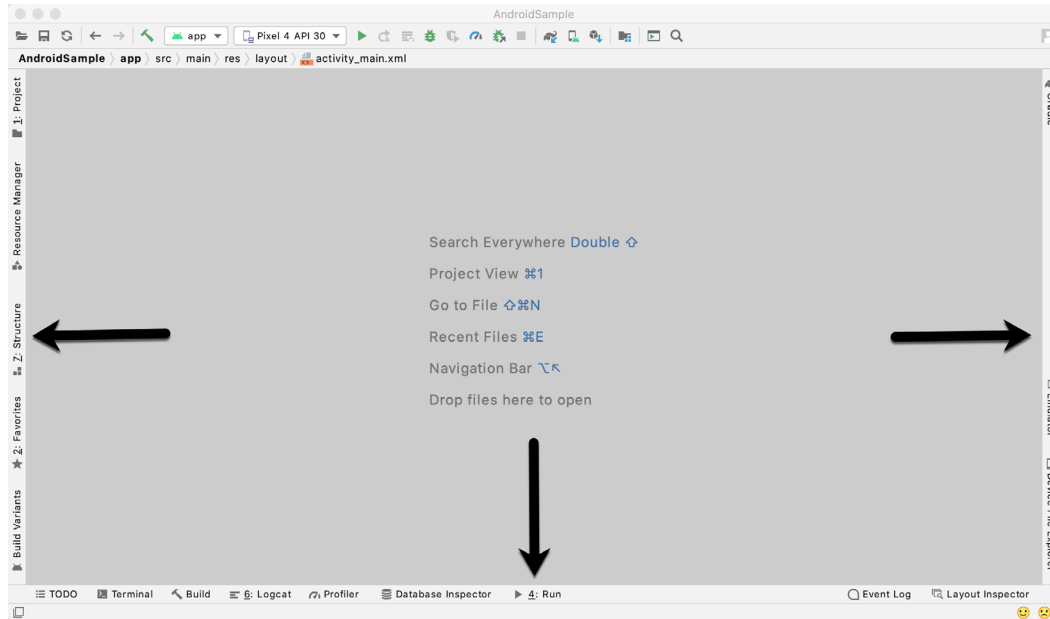


Figure 6-6

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

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

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

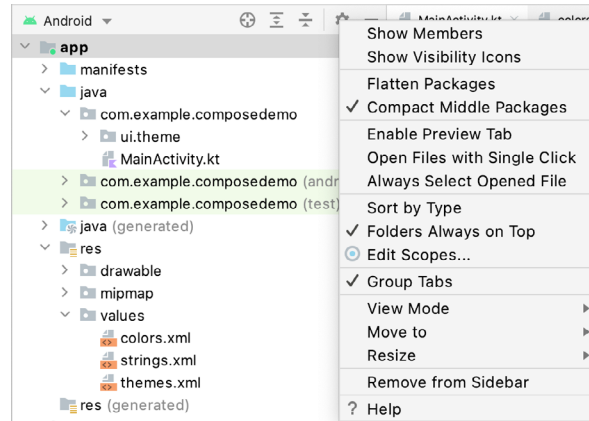


Figure 6-7

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

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

- **App Inspection** - Provides access to the Database and Background Task inspectors. The Database Inspector allows you to inspect, query, and modify your app's databases while the app is running. The Background Task Inspector allows background worker tasks created using WorkManager to be monitored and managed.
- **Build** - The build tool window displays information about the build process while a project is being compiled and packaged and displays details of any errors encountered.
- **Build Variants** - The build variants tool window provides a quick way to configure different build targets for the current application project (for example different builds for debugging and release versions of the application, or multiple builds to target different device categories).
- **Device File Explorer** - Available via the *View -> Tool Windows -> Device File Explorer* menu, this tool window provides direct access to the filesystem of the currently connected Android device or emulator allowing the filesystem to be browsed and files copied to the local filesystem.
- **Device Manager** - Provides access to the Device Manager tool window where physical Android device connections and emulators may be added, removed, and managed.
- **Emulator** - Contains the AVD emulator if the option has been enabled to run the emulator in a tool window as outlined in the chapter entitled *“Creating an Android Virtual Device (AVD) in Android Studio”*.
- **Bookmarks** - The Bookmarks tool window provides quick access to bookmarked files and code lines. For example, right-clicking on a file in the project view allows access to an Add to Bookmarks menu option. Similarly, you can bookmark a line of code in a source file by moving the cursor to that line and pressing the F11 key (F3 on macOS). All bookmarked items can be accessed through this tool window.
- **Gradle** - The Gradle tool window provides a view of the Gradle tasks that make up the project build configuration. The window lists the tasks that are involved in compiling the various elements of the project into an executable application. Right-click on a top-level Gradle task and select the *Open Gradle Config* menu option to load the Gradle build file for the current project into the editor. Gradle will be covered in greater detail later in this book.

- **Layout Inspector** - Provides a visual 3D rendering of the hierarchy of components that make up a user interface layout.
- **Logcat** – The Logcat tool window provides access to the monitoring log output from a running application in addition to options for taking screenshots and videos of the application and stopping and restarting a process.
- **Problems** - A central location in which to view all of the current errors or warnings within the project. Double-clicking on an item in the problem list will take you to the problem file and location.
- **Profiler** – The Android Profiler tool window provides real-time monitoring and analysis tools for identifying performance issues within running apps, including CPU, memory and network usage. This option becomes available when an app is currently running.
- **Project** – The project view provides an overview of the file structure that makes up the project allowing for quick navigation between files. Generally, double-clicking on a file in the project view will cause that file to be loaded into the appropriate editing tool.
- **Resource Manager** - A tool for adding and managing resources and assets such as images, colors, and layout files contained with the project.
- **Run** – The run tool window becomes available when an application is currently running and provides a view of the results of the run together with options to stop or restart a running process. If an application is failing to install and run on a device or emulator, this window will typically provide diagnostic information relating to the problem.
- **Structure** – The structure tool provides a high-level view of the structure of the source file currently displayed in the editor. This information includes a list of items such as classes, methods, and variables in the file. Selecting an item from the structure list will take you to that location in the source file in the editor window.
- **Terminal** – Provides access to a terminal window on the system on which Android Studio is running. On Windows systems, this is the Command Prompt interface, while on Linux and macOS systems this takes the form of a Terminal prompt.
- **TODO** – As the name suggests, this tool provides a place to review items that have yet to be completed on the project. Android Studio compiles this list by scanning the source files that make up the project to look for comments that match specified TODO patterns. These patterns can be reviewed and changed by selecting the *File -> Settings...* menu option (*Android Studio -> Preferences...* on macOS) and navigating to the *TODO* page listed under *Editor*.
- **Version Control** - This tool window is used when the project files are under source code version control, allowing access to Git repositories and code change history.

6.4 Android Studio Keyboard Shortcuts

Android Studio includes an abundance of keyboard shortcuts designed to save time when performing common tasks. A full keyboard shortcut keymap listing can be viewed and printed from within the Android Studio project window by selecting the *Help -> Keymap Reference* menu option. You may also list and modify the keyboard shortcuts by selecting the *File -> Settings...* menu option (*Android Studio -> Preferences...* on macOS) and clicking on the Keymap entry as shown in Figure 6-8 below:

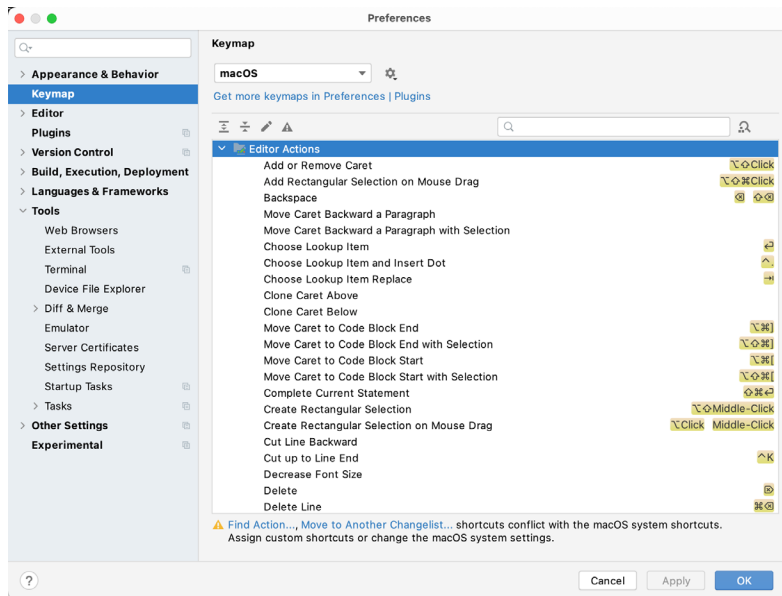


Figure 6-8

6.5 Switcher and Recent Files Navigation

Another useful mechanism for navigating within the Android Studio main window involves the use of the *Switcher*. Accessed via the Ctrl-Tab keyboard shortcut, the switcher appears as a panel listing both the tool windows and currently open files (Figure 6-9).

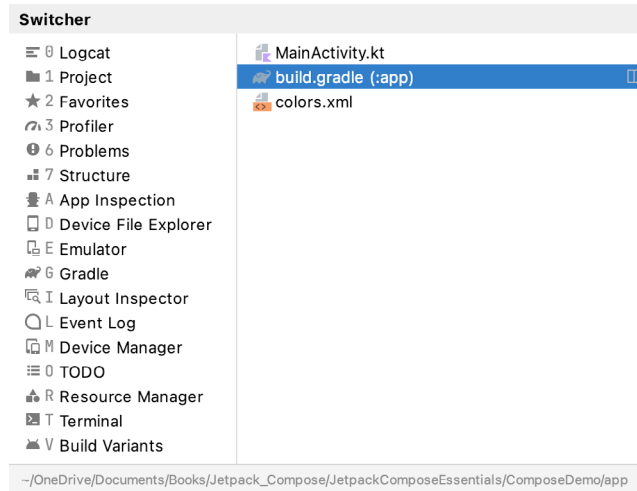


Figure 6-9

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

In addition to the switcher, navigation to recently opened files is provided by the Recent Files panel (Figure 6-10). This can be accessed using the Ctrl-E keyboard shortcut (Cmd-E on macOS). Once displayed, either the mouse pointer can be used to select an option or the keyboard arrow keys used to scroll through the file name

and tool window options. Pressing the Enter key will select the currently highlighted item.

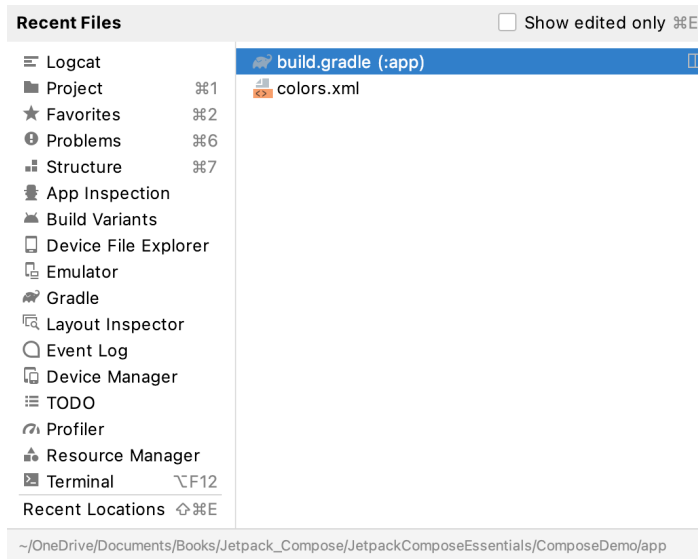


Figure 6-10

6.6 Changing the Android Studio Theme

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

Once the settings dialog is displayed, select the *Appearance & Behavior* option followed by *Appearance* in the left-hand panel and then change the setting of the *Theme* menu before clicking on the *Apply* button. The themes available will depend on the platform but usually include options such as Light, IntelliJ, Windows, High Contrast, and Darcula. Figure 6-11 shows an example of the main window with the Darcula theme selected:

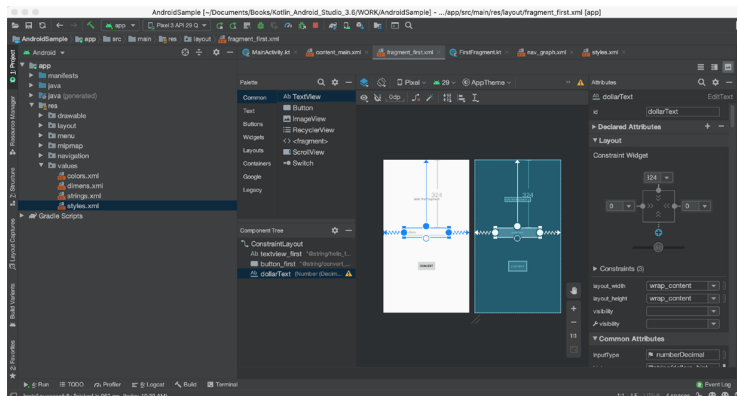


Figure 6-11

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

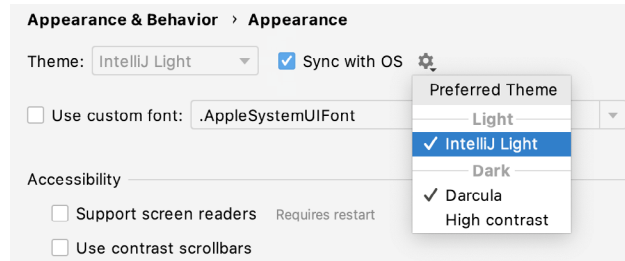


Figure 6-12

6.7 Summary

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

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

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, typecasting, 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://play.kotlinlang.org> 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, can 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. 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 can 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 to handle 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 an 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 messages. Message capacity remaining is
               ${maxcount - inboxCount} messages"

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` - Newline
- `\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 that 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 that 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 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 (except for 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 is then imposed on that variable by the compiler to prevent it from 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 before 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 nonexistent 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 that 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:

Kotlin Data Types, Variables, and Nullability

```
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 the 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 Late Initialization (lateinit)

As previously outlined, non-null types need to be initialized when they are declared. This can be inconvenient if the value to be assigned to the non-null variable will not be known until later in the code execution. One way around this is to declare the variable using the *lateinit* modifier. This modifier designates that a value will be initialized with a value later. This has the advantage that a non-null type can be declared before it is initialized, with the disadvantage that the programmer is responsible for ensuring that the initialization has been performed before attempting to access the variable. Consider the following variable declaration:

```
var myName: String
```

Clearly, this is invalid since the variable is a non-null type but has not been assigned a value. Suppose, however, that the value to be assigned to the variable will not be known until later in the program execution. In this case, the *lateinit* modifier can be used as follows:

```
lateinit var myName: String
```

With the variable declared in this way, the value can be assigned later, for example:

```
myName = "John Smith"
print("My Name is " + myName)
```

Of course, if the variable is accessed before it is initialized, the code will fail with an exception:

```
lateinit var myName: String
```

```
print("My Name is " + myName)
```

```
Exception in thread "main" kotlin.UninitializedPropertyAccessException: lateinit
property myName has not been initialized
```

To verify whether a `lateinit` variable has been initialized, check the *isInitialized* property on the variable. To do this, we need to access the properties of the variable by prefixing the name with the `::` operator:

```
if (::myName.isInitialized) {
    print("My Name is " + myName)
}
```

12.12 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 if 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"
```

12.13 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 unsafe 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
}
```

12.14 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, typecasting 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.

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")
```

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.

21. 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.

21.1 Creating the State Change Example Project

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

Select the *New Project* option from the welcome screen and, within the resulting new project dialog, choose the Empty Views 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. Use the steps outlined in section 18.8 *Migrating a Project to View Binding* to convert the project to use view binding.

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

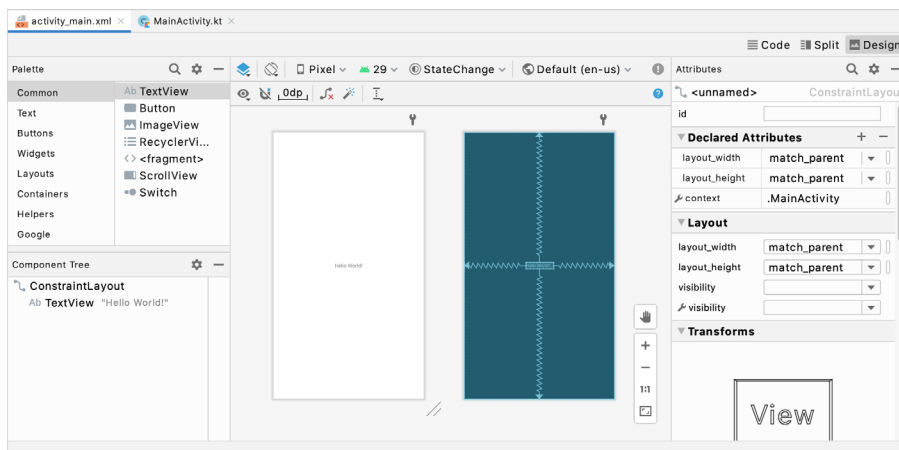


Figure 21-1

21.2 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 21-2.

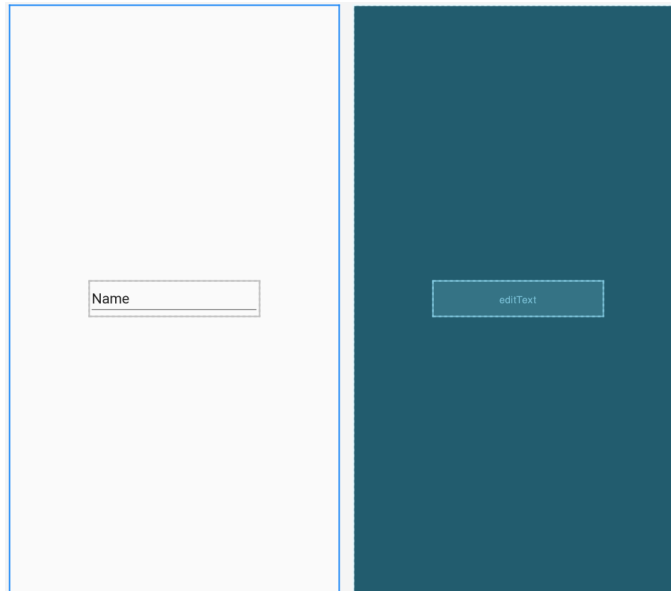


Figure 21-2

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 this example, we will use the default input type to support general text input. To choose a different setting in the future, select the EditText widget in the layout and locate the *inputType* entry within the Attributes tool window. Next, click the flag icon to the left of the current setting to open the list of options, as shown in Figure 21-3 below. The Type menu provides options to restrict the input to text, numbers, dates and times, and phone numbers. The Variations menu provides additional options for the currently selected input type. For example, a variation is available for the text input type for email addresses as input.

Once a type and variation have been chosen, the input type may be customized further using the list of flag checkboxes:

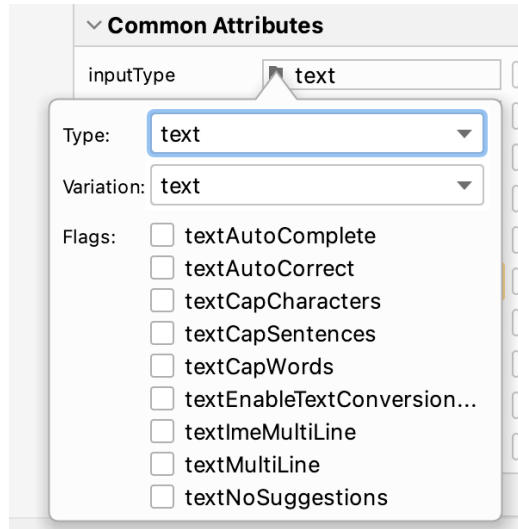


Figure 21-3

Remaining in the Attributes tool window, change the id of the view to *editText* and click on the Refactor button in the resulting dialog.

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

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

21.3 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. If the file is no longer open, navigate to it in the Project tool window panel (*app* -> *java* -> *com.ebookfrenzy.statechange* -> *MainActivity*) and double-click on it to load the file into the editor.

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

```
package com.ebookfrenzy.statechange

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.util.Log

import com.ebookfrenzy.statechange.databinding.ActivityMainBinding

class MainActivity : AppCompatActivity() {

    private lateinit var binding: ActivityMainBinding
```

Android Activity State Changes by Example

```
private val TAG = "StateChange"

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    binding = ActivityMainBinding.inflate(layoutInflater)
    setContentView(binding.root)
    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")
}

```

21.4 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 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.

When displayed while the current app is running, the Logcat tool window will appear as shown in Figure 21-4 below:

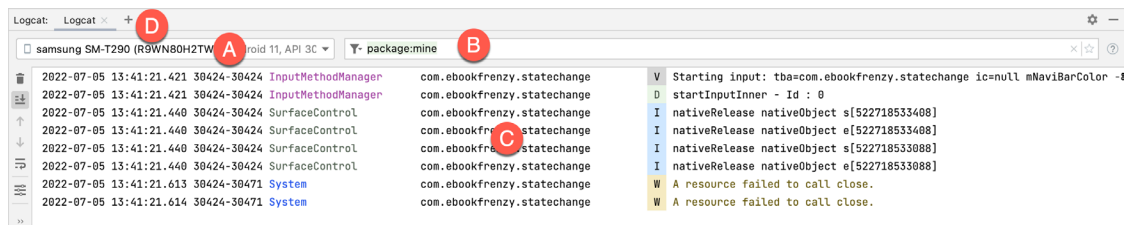


Figure 21-4

The menu marked A in the above figure allows you to select the device or emulator for which log output is to be displayed. This output appears in the output panel marked C. The log output can be filtered by entering options into the field marked B. By default key setting, *package:mine*, restricts the output to log messages generated by the current app package (in this case *com.ebookfrenzy.statechange*). Leaving this field blank will allow log output from the selected device or emulator to be displayed, including diagnostic messages generated by the operating system. Keys may also be combined to further filter the output. For example, we can configure the Logcat panel to display only messages associated with our StateChange tag as follows:

```
package:mine tag:StateChange
```

We can also exclude output by prefixing the key with a minus (-) sign. In addition to the StateChange tag, we might also have diagnostic messages that use a different tag named "OtherTag". To filter the log so that output from this second tag is excluded we could enter the following key options:

```
package:mine tag:StateChange -tag:OtherTag
```

In addition to your own tag values, it is also possible to select from a range of predefined diagnostic tags built into Android. Logcat will display a list of matching tags as you type into the filter field as shown in Figure 21-5:

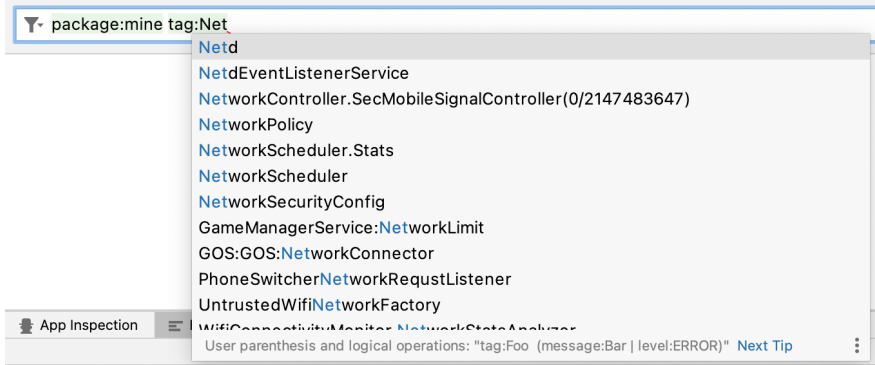


Figure 21-5

The *level* key may be used to control which messages are displayed based on severity. To filter out all messages except error messages, the following key would be used:

```
level:error
```

In addition to *error*, the Logcat panel also supports *verbose*, *info*, *warn* and *assert* level settings.

Logcat also supports multiple log panels, each with its own filter settings. To add another panel, click on the + button marked D in Figure 21-4 above. Switch between different panels using the corresponding tabs, or display them side-by-side by right-clicking on the currently displayed panel and selecting either the *Split-Right* or *Split-Down* menu option to arrange the panels horizontally or vertically. To rename a panel, right-click on the tab and select the *Rename Tab* option. Before proceeding close all but one Logcat panel and configure the filter as follows:

```
package:mime tag:StateChange
```

21.5 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 21-6 below, select the *Run -> Run...* menu option or use the Shift+F10 keyboard shortcut:

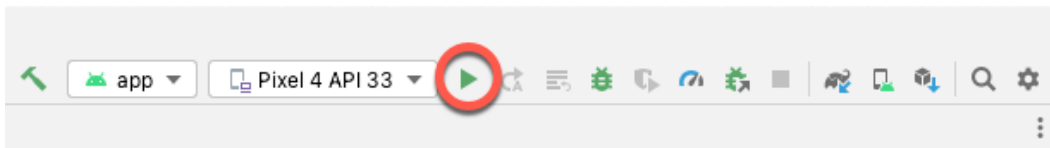


Figure 21-6

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

A review of the Logcat panel should indicate which methods have so far been triggered:

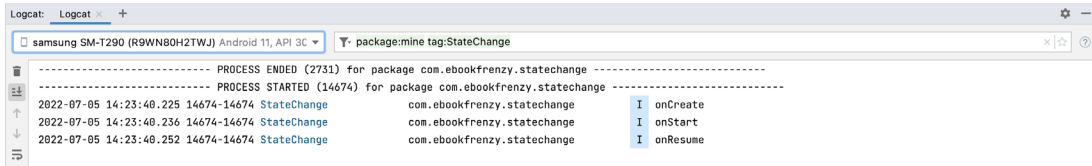


Figure 21-7

21.6 Experimenting with the Activity

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

```
onCreate
onStart
onResume
```

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

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

```
onPause
onStop
onSaveInstanceState
```

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

As outlined in “*Understanding Android Application and Activity Lifecycles*”, the destruction and recreation of an activity can be triggered by making a configuration change to the device, such as rotating from portrait to landscape. To see this in action, simply rotate the device while the *StateChange* application is in the foreground. When using the emulator, device rotation may be simulated using the rotation button located in the emulator toolbar. To complete the rotation, it may also be necessary to tap on the rotation button. This appears at the bottom of the device or emulator screen as shown in Figure 21-8:

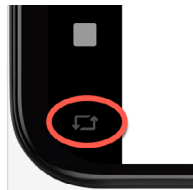


Figure 21-8

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

```
onPause
onStop
onSaveInstanceState
```

Android Activity State Changes by Example

```
onDestroy  
onCreate  
onStart  
onRestoreInstanceState  
onResume
```

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

21.7 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.

27. Working with ConstraintLayout Chains and Ratios in Android Studio

The previous chapters have introduced the key features of the `ConstraintLayout` class and outlined the best practices for `ConstraintLayout`-based user interface design within the Android Studio Layout Editor. Although the concepts of `ConstraintLayout` chains and ratios were outlined in the chapter entitled “*A Guide to the Android ConstraintLayout*”, we have not yet addressed how to make use of these features within the Layout Editor. The focus of this chapter, therefore, is to provide practical steps on how to create and manage chains and ratios when using the `ConstraintLayout` class.

27.1 Creating a Chain

Chains may be implemented either by adding a few lines to the XML layout resource file of an activity or by using some chain specific features of the Layout Editor.

Consider a layout consisting of three `Button` widgets constrained so as to be positioned in the top-left, top-center and top-right of the `ConstraintLayout` parent as illustrated in Figure 27-1:

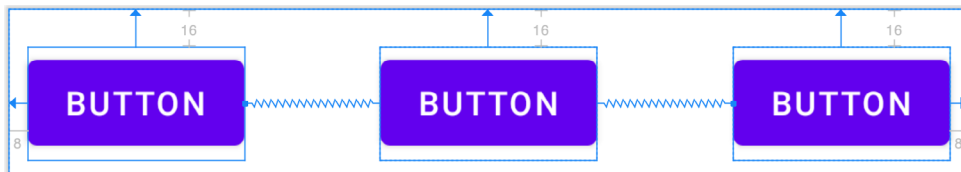


Figure 27-1

To represent such a layout, the XML resource layout file might contain the following entries for the button widgets:

```
<Button
    android:id="@+id/button1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginStart="8dp"
    android:layout_marginTop="16dp"
    android:text="Button"
    app:layout_constraintHorizontal_bias="0.5"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />

<Button
    android:id="@+id/button2"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginEnd="8dp"
```

```
android:layout_marginStart="8dp"
android:layout_marginTop="16dp"
android:text="Button"
app:layout_constraintHorizontal_bias="0.5"
app:layout_constraintEnd_toStartOf="@+id/button3"
app:layout_constraintStart_toEndOf="@+id/button1"
app:layout_constraintTop_toTopOf="parent" />
```

```
<Button
    android:id="@+id/button3"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginEnd="8dp"
    android:layout_marginTop="16dp"
    android:text="Button"
    app:layout_constraintHorizontal_bias="0.5"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
```

As currently configured, there are no bi-directional constraints to group these widgets into a chain. To address this, additional constraints need to be added from the right-hand side of button1 to the left side of button2, and from the left side of button3 to the right side of button2 as follows:

```
<Button
    android:id="@+id/button1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginStart="8dp"
    android:layout_marginTop="16dp"
    android:text="Button"
    app:layout_constraintHorizontal_bias="0.5"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintEnd_toStartOf="@+id/button2" />
```

```
<Button
    android:id="@+id/button2"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginEnd="8dp"
    android:layout_marginStart="8dp"
    android:layout_marginTop="16dp"
    android:text="Button"
    app:layout_constraintHorizontal_bias="0.5"
    app:layout_constraintEnd_toStartOf="@+id/button3"
    app:layout_constraintStart_toEndOf="@+id/button1"
    app:layout_constraintTop_toTopOf="parent" />
```

```

<Button
    android:id="@+id/button3"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginEnd="8dp"
    android:layout_marginTop="16dp"
    android:text="Button"
    app:layout_constraintHorizontal_bias="0.5"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintStart_toEndOf="@+id/button2" />

```

With these changes, the widgets now have bi-directional horizontal constraints configured. This essentially constitutes a ConstraintLayout chain which is represented visually within the Layout Editor by chain connections as shown in Figure 27-2 below. Note that in this configuration the chain has defaulted to the *spread* chain style.

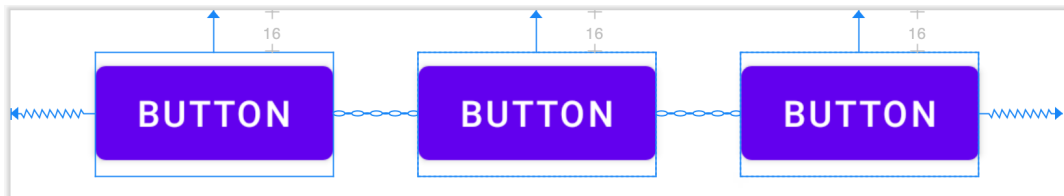


Figure 27-2

A chain may also be created by right-clicking on one of the views and selecting the *Chains -> Create Horizontal Chain* or *Chains -> Create Vertical Chain* menu options.

27.2 Changing the Chain Style

If no chain style is configured, the ConstraintLayout will default to the spread chain style. The chain style can be altered by right-clicking any of the widgets in the chain and selecting the *Cycle Chain Mode* menu option. Each time the menu option is clicked the style will switch to another setting in the order of spread, spread inside and packed.

Alternatively, the style may be specified in the Attributes tool window unfolding the *layout_constraints* property and changing either the *horizontal_chainStyle* or *vertical_chainStyle* property depending on the orientation of the chain:

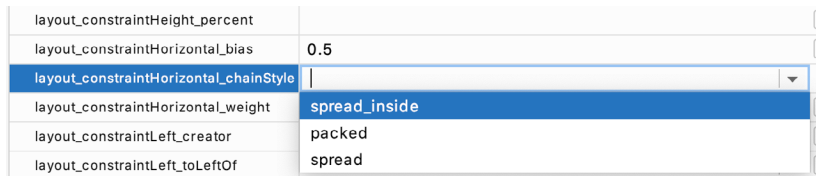


Figure 27-3

27.3 Spread Inside Chain Style

Figure 27-4 illustrates the effect of changing the chain style to the *spread inside* chain style using the above techniques:

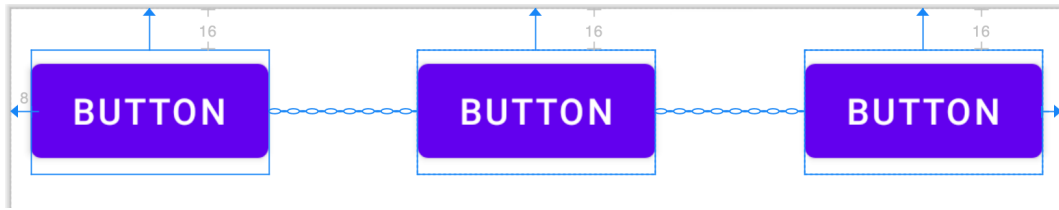


Figure 27-4

27.4 Packed Chain Style

Using the same technique, changing the chain style property to *packed* causes the layout to change as shown in Figure 27-5:

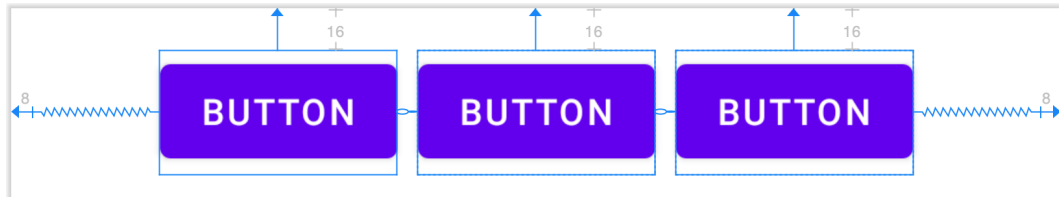


Figure 27-5

27.5 Packed Chain Style with Bias

The positioning of the packed chain may be influenced by applying a bias value. The bias can be any value between 0.0 and 1.0, with 0.5 representing the center of the parent. Bias is controlled by selecting the chain head widget and assigning a value to the `layout_constraintHorizontal_bias` or `layout_constraintVertical_bias` attribute in the Attributes panel. Figure 27-6 shows a packed chain with a horizontal bias setting of 0.2:

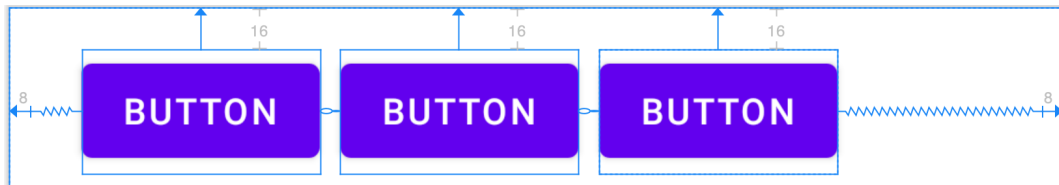


Figure 27-6

27.6 Weighted Chain

The final area of chains to explore involves weighting of the individual widgets to control how much space each widget in the chain occupies within the available space. A weighted chain may only be implemented using the *spread* chain style and any widget within the chain that is to respond to the weight property must have the corresponding dimension property (height for a vertical chain and width for a horizontal chain) configured for *match_constraint* mode. Match constraint mode for a widget dimension may be configured by selecting the widget, displaying the Attributes panel and changing the dimension to *match_constraint* (equivalent to 0dp). In Figure 27-7, for example, the `layout_width` constraint for a button has been set to *match_constraint* (0dp) to indicate that the width of the widget is to be determined based on the prevailing constraint settings:

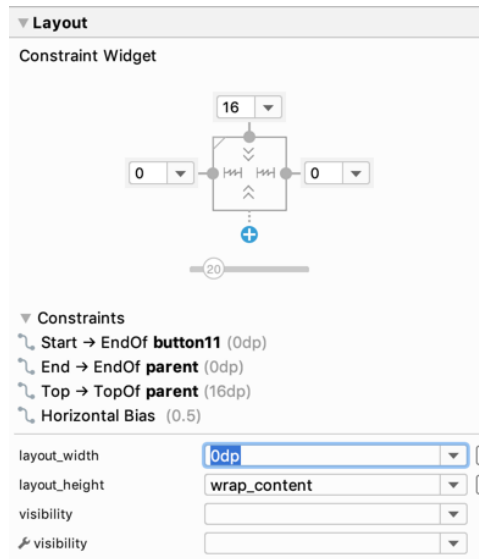


Figure 27-7

Assuming that the spread chain style has been selected, and all three buttons have been configured such that the width dimension is set to match the constraints, the widgets in the chain will expand equally to fill the available space:

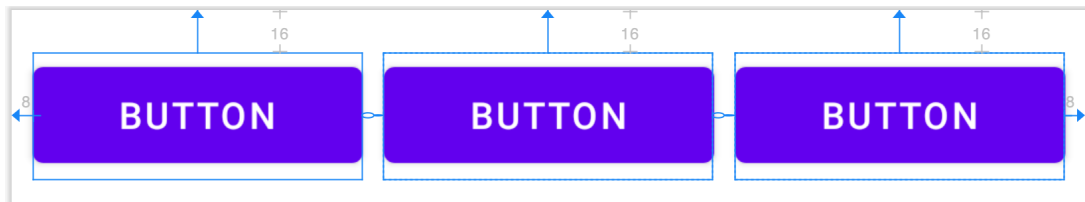


Figure 27-8

The amount of space occupied by each widget relative to the other widgets in the chain can be controlled by adding weight properties to the widgets. Figure 27-9 shows the effect of setting the `layout_constraintHorizontal_weight` property to 4 on button1, and to 2 on both button2 and button3:

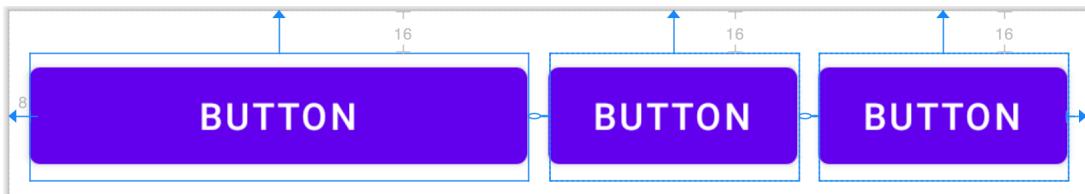


Figure 27-9

As a result of these weighting values, button1 occupies half of the space ($4/8$), while button2 and button3 each occupy one quarter ($2/8$) of the space.

27.7 Working with Ratios

ConstraintLayout ratios allow one dimension of a widget to be sized relative to the widget's other dimension (otherwise known as aspect ratio). An aspect ratio setting could, for example, be applied to an `ImageView` to ensure that its width is always twice its height.

A dimension ratio constraint is configured by setting the constrained dimension to match constraint mode and configuring the `layout_constraintDimensionRatio` attribute on that widget to the required ratio. This ratio value may be specified either as a float value or a `width:height` ratio setting. The following XML excerpt, for example, configures a ratio of 2:1 on an `ImageView` widget:

```
<ImageView
    android:layout_width="0dp"
    android:layout_height="100dp"
    android:id="@+id/imageView"
    app:layout_constraintDimensionRatio="2:1" />
```

The above example demonstrates how to configure a ratio when only one dimension is set to match constraint. A ratio may also be applied when both dimensions are set to match constraint mode. This involves specifying the ratio preceded with either an H or a W to indicate which of the dimensions is constrained relative to the other.

Consider, for example, the following XML excerpt for an `ImageView` object:

```
<ImageView
    android:layout_width="0dp"
    android:layout_height="0dp"
    android:id="@+id/imageView"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintDimensionRatio="W,1:3" />
```

In the above example the height will be defined subject to the constraints applied to it. In this case constraints have been configured such that it is attached to the top and bottom of the parent view, essentially stretching the widget to fill the entire height of the parent. The width dimension, on the other hand, has been constrained to be one third of the `ImageView`'s height dimension. Consequently, whatever size screen or orientation the layout appears on, the `ImageView` will always be the same height as the parent and the width one third of that height.

The same results may also be achieved without the need to manually edit the XML resource file. Whenever a widget dimension is set to match constraint mode, a ratio control toggle appears in the Inspector area of the property panel. Figure 27-10, for example, shows the layout width and height attributes of a button widget set to match constraint mode and 100dp respectively, and highlights the ratio control toggle in the widget sizing preview:

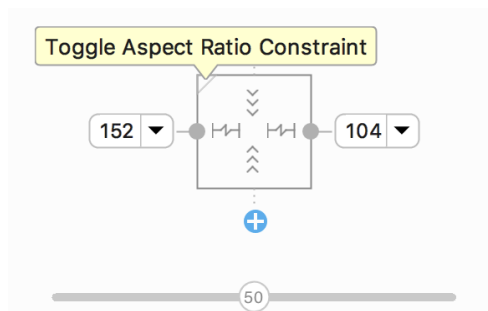


Figure 27-10

By default the ratio sizing control is toggled off. Clicking on the control enables the ratio constraint and displays

an additional field where the ratio may be changed:

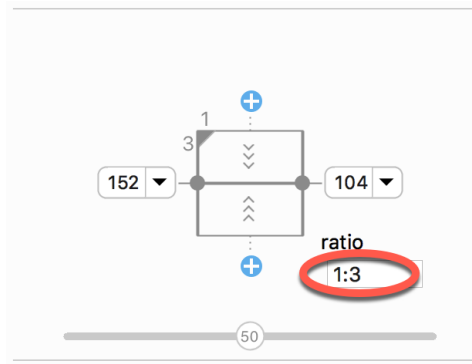


Figure 27-11

27.8 Summary

Both chains and ratios are powerful features of the `ConstraintLayout` class intended to provide additional options for designing flexible and responsive user interface layouts within Android applications. As outlined in this chapter, the Android Studio Layout Editor has been enhanced to make it easier to use these features during the user interface design process.

34. Android Touch and Multi-touch Event Handling

Most Android based devices use a touch screen as the primary interface between user and device. The previous chapter introduced the mechanism by which a touch on the screen translates into an action within a running Android application. There is, however, much more to touch event handling than responding to a single finger tap on a view object. Most Android devices can, for example, detect more than one touch at a time. Nor are touches limited to a single point on the device display. Touches can, of course, be dynamic as the user slides one or more points of contact across the surface of the screen.

Touches can also be interpreted by an application as a *gesture*. Consider, for example, that a horizontal swipe is typically used to turn the page of an eBook, or how a pinching motion can be used to zoom in and out of an image displayed on the screen.

This chapter will explain the handling of touches that involve motion and explore the concept of intercepting multiple concurrent touches. The topic of identifying distinct gestures will be covered in the next chapter.

34.1 Intercepting Touch Events

Touch events can be intercepted by a view object through the registration of an *onTouchListener* event listener and the implementation of the corresponding *onTouch()* callback method or lambda. The following code, for example, ensures that any touches on a *ConstraintLayout* view instance named *myLayout* result in a call to a lambda expression:

```
binding.myLayout.setOnTouchListener {v: View, m: MotionEvent ->
    // Perform tasks here
    true
}
```

Of course, the above code could also be implemented by using a function instead of a lambda as follows, though the lambda approach results in more compact and readable code:

```
binding.myLayout.setOnTouchListener(object : View.OnTouchListener {
    override fun onTouch(v: View, m: MotionEvent): Boolean {
        // Perform tasks here
        return true
    }
})
```

As indicated in the code example, the lambda expression is required to return a Boolean value indicating to the Android runtime system whether or not the event should be passed on to other event listeners registered on the same view or discarded. The method is passed both a reference to the view on which the event was triggered and an object of type *MotionEvent*.

34.2 The MotionEvent Object

The MotionEvent object passed through to the *onTouch()* callback method is the key to obtaining information about the event. Information contained within the object includes the location of the touch within the view and the type of action performed. The MotionEvent object is also the key to handling multiple touches.

34.3 Understanding Touch Actions

An important aspect of touch event handling involves being able to identify the type of action performed by the user. The type of action associated with an event can be obtained by making a call to the *getActionMasked()* method of the MotionEvent object which was passed through to the *onTouch()* callback method. When the first touch on a view occurs, the MotionEvent object will contain an action type of ACTION_DOWN together with the coordinates of the touch. When that touch is lifted from the screen, an ACTION_UP event is generated. Any motion of the touch between the ACTION_DOWN and ACTION_UP events will be represented by ACTION_MOVE events.

When more than one touch is performed simultaneously on a view, the touches are referred to as *pointers*. In a multi-touch scenario, pointers begin and end with event actions of type ACTION_POINTER_DOWN and ACTION_POINTER_UP respectively. To identify the index of the pointer that triggered the event, the *getActionIndex()* callback method of the MotionEvent object must be called.

34.4 Handling Multiple Touches

The chapter entitled “*An Overview and Example of Android Event Handling*” began exploring event handling within the narrow context of a single touch event. In practice, most Android devices possess the ability to respond to multiple consecutive touches (though it is important to note that the number of simultaneous touches that can be detected varies depending on the device).

As previously discussed, each touch in a multi-touch situation is considered by the Android framework to be a *pointer*. Each pointer, in turn, is referenced by an *index* value and assigned an *ID*. The current number of pointers can be obtained via a call to the *getPointerCount()* method of the current MotionEvent object. The ID for a pointer at a particular index in the list of current pointers may be obtained via a call to the MotionEvent *getPointerId()* method. For example, the following code excerpt obtains a count of pointers and the ID of the pointer at index 0:

```
binding.myLayout.setOnTouchListener {v: View, m: MotionEvent ->
    val pointerCount = m.pointerCount
    val pointerId = m.getPointerId(0)
    true
}
```

Note that the pointer count will always be greater than or equal to 1 when the *onTouch* listener is triggered (since at least one touch must have occurred for the callback to be triggered).

A touch on a view, particularly one involving motion across the screen, will generate a stream of events before the point of contact with the screen is lifted. As such, it is likely that an application will need to track individual touches over multiple touch events. While the ID of a specific touch gesture will not change from one event to the next, it is important to keep in mind that the index value will change as other touch events come and go. When working with a touch gesture over multiple events, therefore, it is essential that the ID value be used as the touch reference to make sure the same touch is being tracked. When calling methods that require an index value, this should be obtained by converting the ID for a touch to the corresponding index value via a call to the *findPointerIndex()* method of the *MotionEvent* object.

34.5 An Example Multi-Touch Application

The example application created in the remainder of this chapter will track up to two touch gestures as they move across a layout view. As the events for each touch are triggered, the coordinates, index and ID for each touch will be displayed on the screen.

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

Enter *MotionEvent* into the Name field and specify *com.ebookfrenzy.motionevent* 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.

Adapt the project to use view binding as outlined in section 18.8 *Migrating a Project to View Binding*.

34.6 Designing the Activity User Interface

The user interface for the application's sole activity is to consist of a ConstraintLayout view containing two TextView objects. Within the Project tool window, navigate to *app -> res -> layout* and double-click on the *activity_main.xml* layout resource file to load it into the Android Studio Layout Editor tool.

Select and delete the default “Hello World!” TextView widget and then, with autoconnect enabled, drag and drop a new TextView widget so that it is centered horizontally and positioned at the 16dp margin line on the top edge of the layout:



Figure 34-1

Drag a second TextView widget and position and constrain it so that it is distanced by a 32dp margin from the bottom of the first widget:



Figure 34-2

Using the Attributes tool window, change the IDs for the TextView widgets to *textView1* and *textView2* respectively. Change the text displayed on the widgets to read “Touch One Status” and “Touch Two Status” and extract the strings to resources using the warning button in the top right-hand corner of the Layout Editor.

34.7 Implementing the Touch Event Listener

To receive touch event notifications it will be necessary to register a touch listener on the layout view within the *onCreate()* method of the *MainActivity* activity class. Select the *MainActivity.kt* tab from the Android Studio editor panel to display the source code. Within the *onCreate()* method, add code to register the touch listener and implement code which, in this case, is going to call a second method named *handleTouch()* to which is passed the MotionEvent object:

```
package com.ebookfrenzy.motionevent

import androidx.appcompat.app.AppCompatActivity
```

Android Touch and Multi-touch Event Handling

```
import android.os.Bundle
import android.view.MotionEvent

import com.ebookfrenzy.motionevent.databinding.ActivityMainBinding

class MainActivity : AppCompatActivity() {

    private lateinit var binding: ActivityMainBinding

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)

        binding.root.setOnTouchListener {_, m: MotionEvent ->
            handleTouch(m)
            true
        }
    }
}
```

When we designed the user interface, the parent `ConstraintLayout` was not assigned an ID that would allow us to access it via the view binding mechanism. Since this layout component is the top-most component in the UI layout hierarchy, we have been able to reference it using the *root* binding property in the code above.

The final task before testing the application is to implement the *handleTouch()* method called by the listener. The code for this method reads as follows:

```
private fun handleTouch(m: MotionEvent)
{
    val pointerCount = m.pointerCount

    for (i in 0 until pointerCount)
    {
        val x = m.getX(i)
        val y = m.getY(i)
        val id = m.getPointerId(i)
        val action = m.actionMasked
        val actionIndex = m.actionIndex
        var actionString: String

        when (action)
        {
            MotionEvent.ACTION_DOWN -> actionString = "DOWN"
            MotionEvent.ACTION_UP -> actionString = "UP"
            MotionEvent.ACTION_POINTER_DOWN -> actionString = "PNTR DOWN"
            MotionEvent.ACTION_POINTER_UP -> actionString = "PNTR UP"
            MotionEvent.ACTION_MOVE -> actionString = "MOVE"
```



```

        else -> actionString = ""
    }

    val touchStatus =
        "Action: $actionString Index: $actionIndex ID: $id X: $x Y: $y"

    if (id == 0)
        binding.textView1.text = touchStatus
    else
        binding.textView2.text = touchStatus
    }
}

```

Before compiling and running the application, it is worth taking the time to walk through this code systematically to highlight the tasks that are being performed.

The code begins by identifying how many pointers are currently active on the view:

```
val pointerCount = m.pointerCount
```

Next, the *pointerCount* variable is used to initiate a *for* loop which performs a set of tasks for each active pointer. The first few lines of the loop obtain the X and Y coordinates of the touch together with the corresponding event ID, action type and action index. Lastly, a string variable is declared:

```

for (i in 0 until pointerCount)
{
    val x = m.getX(i)
    val y = m.getY(i)
    val id = m.getPointerId(i)
    val action = m.actionMasked
    val actionIndex = m.actionIndex
    var actionString: String

```

Since action types equate to integer values, a *when* statement is used to convert the action type to a more meaningful string value, which is stored in the previously declared *actionString* variable:

```

when (action)
{
    MotionEvent.ACTION_DOWN -> actionString = "DOWN"
    MotionEvent.ACTION_UP -> actionString = "UP"
    MotionEvent.ACTION_POINTER_DOWN -> actionString = "PNTR DOWN"
    MotionEvent.ACTION_POINTER_UP -> actionString = "PNTR UP"
    MotionEvent.ACTION_MOVE -> actionString = "MOVE"
    else -> actionString = ""
}

```

Finally, the string message is constructed using the *actionString* value, the action index, touch ID and X and Y coordinates. The ID value is then used to decide whether the string should be displayed on the first or second *TextView* object:

```

val touchStatus =
    "Action: $actionString Index: $actionIndex ID: $id X: $x Y: $y"

```

```
if (id == 0)
    binding.textView1.text = touchStatus
else
    binding.textView2.text = touchStatus
```

34.8 Running the Example Application

Compile and run the application and, once launched, experiment with single and multiple touches on the screen and note that the text views update to reflect the events as illustrated in Figure 34-3. When running on an emulator, multiple touches may be simulated by holding down the Ctrl (Cmd on macOS) key while clicking the mouse button (note that simulating multiple touches may not work if the emulator is running in a tool window):

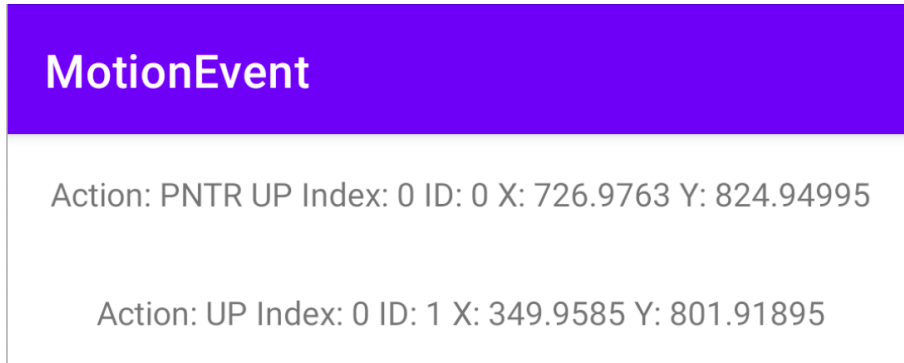


Figure 34-3

34.9 Summary

Activities receive notifications of touch events by registering an `onTouchListener` event listener and implementing the `onTouch()` callback method which, in turn, is passed a `MotionEvent` object when called by the Android runtime. This object contains information about the touch such as the type of touch event, the coordinates of the touch and a count of the number of touches currently in contact with the view.

When multiple touches are involved, each point of contact is referred to as a pointer with each assigned an index and an ID. While the index of a touch can change from one event to another, the ID will remain unchanged until the touch ends.

This chapter has worked through the creation of an example Android application designed to display the coordinates and action type of up to two simultaneous touches on a device display.

Having covered touches in general, the next chapter (entitled “*Detecting Common Gestures Using the Android Gesture Detector Class*”) will look further at touch screen event handling through the implementation of gesture recognition.

40. An Android ViewModel Tutorial

The previous chapter introduced the fundamental concepts of Android Jetpack and outlined the basics of modern Android app architecture. Jetpack defines a set of recommendations describing how an Android app project should be structured while providing a set of libraries and components that make it easier to conform to these guidelines with the goal of developing reliable apps with less coding and fewer errors.

To help reinforce and clarify the information provided in the previous chapter, this chapter will step through the creation of an example app project that uses the ViewModel component. This example will be further enhanced in the next chapter by including LiveData and data binding support.

40.1 About the Project

In the chapter entitled “*Creating an Example Android App in Android Studio*”, a project named `AndroidSample` was created in which all of the code for the app was bundled into the main Activity class file. In the chapter that followed, an AVD emulator was created and used to run the app. While the app was running, we experienced first-hand the kind of problems that occur when developing apps in this way when the data displayed on a `TextView` widget was lost during a device rotation.

This chapter will implement the same currency converter app, this time using the ViewModel component and following the Google app architecture guidelines to avoid Activity lifecycle complications.

40.2 Creating the ViewModel Example Project

When the `AndroidSample` project was created, the *Empty Views Activity* template was chosen as the basis for the project. For this project, however, the *Basic Views Template* template will be used.

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

Enter `ViewModelDemo` into the Name field and specify `com.ebookfrenzy.viewmodeldemo` 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.

40.3 Removing Unwanted Project Elements

As outlined in the “*A Guide to the Android Studio Layout Editor Tool*”, the Basic Views Activity template includes features that will not be needed by all projects. Before adding the ViewModel to the project, we first need to remove the navigation features, the second content fragment, and floating action button as follows:

1. Double-click on the `activity_main.xml` layout file in the Project tool window, select the floating action button and tap the keyboard delete key to remove the object from the layout.
2. Edit the `MainActivity.kt` file and remove the floating action button code from the `onCreate` method as follows:

```
override fun onCreate(savedInstanceState: Bundle?) {  
    .  
    .  
    binding.fab.setOnClickListener { view ->
```

```

Snackbar.make(view, "Replace with your own action", Snackbar.LENGTH_LONG)
.setAnchorView(R.id.fab)
.setAction("Action", null).show()
}
}

```

3. Within the Project tool window, navigate to and double-click on the *app -> res -> navigation -> nav_graph.xml* file to load it into the navigation editor.
4. Within the editor, select the *SecondFragment* entry in the graph panel and tap the keyboard delete key to remove it from the graph.
5. Locate and delete the *SecondFragment.kt* and *fragment_second.xml* files.
6. The final task is to remove some code from the *FirstFragment* class so that the Button view no longer navigates to the now non-existent second fragment when clicked. Edit the *FirstFragment.kt* file and remove the code from the *onViewCreated()* method so that it reads as follows:

```

override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)

    binding.buttonFirst.setOnClickListener {
    findNavController().navigate(R.id.action_FirstFragment_to_SecondFragment)
}
}

```

40.4 Designing the Fragment Layout

The next step is to design the layout of the fragment. First, locate the *fragment_first.xml* file in the Project tool window and double click on it to load it into the layout editor. Once the layout has loaded, select and delete the existing Button, TextView, and ConstraintLayout components. Next, right-click on the *NestedScrollView* instance in the Component Tree panel and select the *Convert NestedScrollView to ConstraintLayout* menu option as shown in Figure 40-1, and accept the default settings in the resulting dialog:

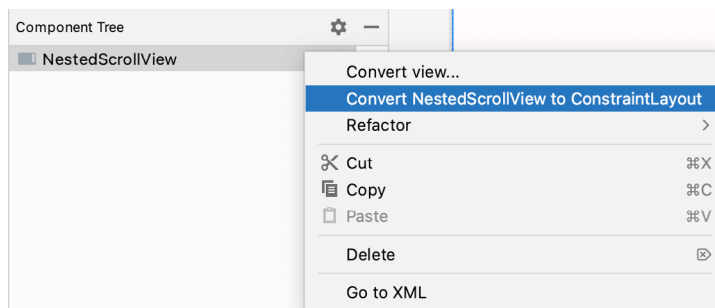


Figure 40-1

Select the converted *ConstraintLayout* component and use the Attributes tool window to change the id to *constraintLayout*.

Add a new *TextView*, position it in the center of the layout and change the id to *resultText*. Next, drag a *Number (Decimal)* view from the palette and position it above the existing *TextView*. With the view selected in the layout refer to the Attributes tool window and change the id to *dollarText*.

Drag a *Button* widget onto the layout so that it is positioned below the *TextView*, and change the text attribute to

read “Convert”. With the button still selected, change the id property to *convertButton*. At this point, the layout should resemble that illustrated in Figure 40-2 (note that the three views have been constrained using a vertical chain):

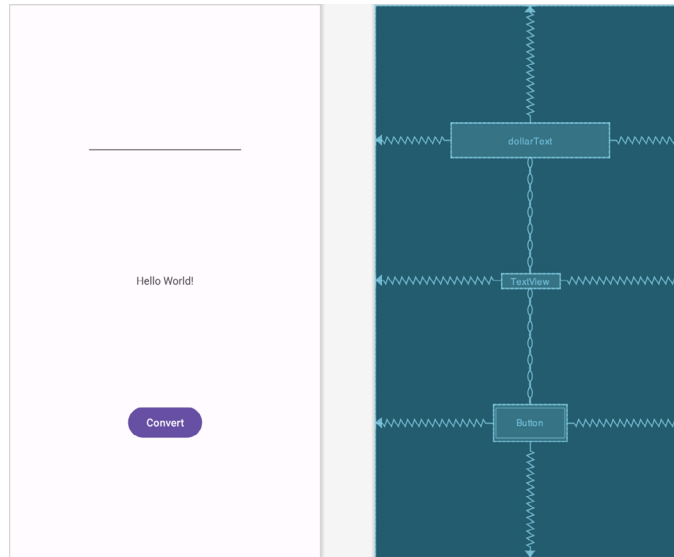


Figure 40-2

Finally, click on the warning icon in the top right-hand corner of the layout editor and convert the hard-coded strings to resources.

40.5 Implementing the View Model

With the user interface layout completed, the data model for the app needs to be created within the view model. Begin by locating the *com.ebookfrenzy.viewmodeldemo* entry in the Project tool window, right-clicking on it and selecting the *New -> Kotlin Class/File* menu option. Name the new class *MainViewModel* and press the keyboard enter key. Edit the new class file so that it reads as follows:

```
package com.ebookfrenzy.viewmodeldemo

import androidx.lifecycle.ViewModel

class MainViewModel : ViewModel() {

    private val rate = 0.74f
    private var dollarText = ""
    private var result: Float = 0f

    fun setAmount(value: String) {
        this.dollarText = value
        result = value.toFloat() * rate
    }

    fun getResult(): Float {
        return result
    }
}
```

```

    }
}

```

The class declares variables to store the current dollar string value and the converted amount together with getter and setter methods to provide access to those data values. When called, the *setAmount()* method takes as an argument the current dollar amount and stores it in the local *dollarText* variable. The dollar string value is converted to a floating point number, multiplied by a fictitious exchange rate and the resulting euro value stored in the *result* variable. The *getResult()* method, on the other hand, simply returns the current value assigned to the *result* variable.

40.6 Associating the Fragment with the View Model

Clearly, there needs to be some way for the fragment to obtain a reference to the ViewModel to be able to access the model and observe data changes. A Fragment or Activity maintains references to the ViewModels on which it relies for data using an instance of the ViewModelProvider class.

A ViewModelProvider instance is created using the ViewModelProvider class from within the Fragment. When called, the class initializer is passed a reference to the current Fragment or Activity and returns a ViewModelProvider instance as follows:

```
val viewModelProvider = ViewModelProvider(this)
```

Once the ViewModelProvider instance has been created, an index value can be used to request a specific ViewModel class. The provider will then either create a new instance of that ViewModel class, or return an existing instance, for example:

```
val viewModel = ViewModelProvider(this) [MyViewModel::class.java]
```

Edit the *FirstFragment.kt* file and override the *onCreate()* method to set up the ViewModelProvider:

```

.
.
import androidx.lifecycle.ViewModelProvider
.
.
class FirstFragment : Fragment() {
.
.
    private lateinit var viewModel: MainViewModel

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        viewModel = ViewModelProvider(this) [MainViewModel::class.java]
    }
.
.

```

With access to the model view, code can now be added to the Fragment to begin working with the data model.

40.7 Modifying the Fragment

The fragment class now needs to be updated to react to button clicks and to interact with the data values stored in the ViewModel. The class will also need references to the three views in the user interface layout to react to button clicks, extract the current dollar value and to display the converted currency amount.

In the chapter entitled “*Creating an Example Android App in Android Studio*”, the `onClick` property of the `Button` widget was used to designate the method to be called when the button is clicked by the user. Unfortunately, this property is only able to call methods on an `Activity` and cannot be used to call a method in a `Fragment`. To get around this limitation, we will need to add some code to the `Fragment` class to set up an `onClick` listener on the button. This can be achieved in the `onViewCreated()` lifecycle method in the `FirstFragment.kt` file as outlined below:

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)

    binding.convertButton.setOnClickListener {

    }
}
```

With the listener added, any code placed within the `onClick()` method will be called whenever the button is clicked by the user.

40.8 Accessing the ViewModel Data

When the button is clicked, the `onClick()` method needs to read the current value from the `EditText` view, confirm that the field is not empty and then call the `setAmount()` method of the `ViewModel` instance. The method will then need to call the `ViewModel`'s `getResult()` method and display the converted value on the `TextView` widget.

Since `LiveData` is not yet being used in the project, it will also be necessary to get the latest result value from the `ViewModel` each time the `Fragment` is created.

Remaining in the `FirstFragment.kt` file, implement these requirements as follows in the `onViewCreated()` method:

```
.
.
.
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)

    binding.resultText.text = viewModel.getResult().toString()

    binding.convertButton.setOnClickListener {
        if (binding.dollarText.text.isNotEmpty()) {
            viewModel.setAmount(binding.dollarText.text.toString())
            binding.resultText.text = viewModel.getResult().toString()
        } else {
            binding.resultText.text = "No Value"
        }
    }
}
```

40.9 Testing the Project

With this phase of the project development completed, build and run the app on the simulator or a physical device, enter a dollar value and click on the `Convert` button. The converted amount should appear on the `TextView` indicating that the UI controller and `ViewModel` re-structuring appears to be working as expected.

When the original `AndroidSample` app was run, rotating the device caused the value displayed on the `resultText`

TextView widget to be lost. Repeat this test now with the ViewModelDemo app and note that the current euro value is retained after the rotation. This is because the ViewModel remained in memory as the Fragment was destroyed and recreated and code was added to the *onViewCreated()* method to update the TextView with the result data value from the ViewModel each time the Fragment re-started.

While this is an improvement on the original AndroidSample app, there is much more that can be achieved to simplify the project by making use of LiveData and data binding, both of which are the topics of the next chapters.

40.10 Summary

In this chapter we revisited the AndroidSample project created earlier in the book and created a new version of the project structured to comply with the Android Jetpack architectural guidelines. The example project also demonstrated the use of ViewModels to separate data handling from user interface related code. Finally, the chapter showed how the ViewModel approach avoids some of the problems of handling Fragment and Activity lifecycles.

61. An Introduction to Kotlin Coroutines

When an Android application is first started, the runtime system creates a single thread in which all application components will run by default. This thread is generally referred to as the *main thread*. The primary role of the main thread is to handle the user interface in terms of event handling and interaction with views in the user interface. Any additional components that are started within the application will, by default, also run on the main thread.

Any code within an application that performs a time consuming task using the main thread will cause the entire application to appear to lock up until the task is completed. This will typically result in the operating system displaying an “Application is not responding” warning to the user. This is far from the desired behavior for any application. Fortunately, Kotlin provides a lightweight alternative in the form of Coroutines. In this chapter, we will introduce Coroutines, including terminology such as dispatchers, coroutine scope, suspend functions, coroutine builders, and structured concurrency. The chapter will also explore channel-based communication between coroutines.

61.1 What are Coroutines?

Coroutines are blocks of code that execute asynchronously without blocking the thread from which they are launched. Coroutines can be implemented without having to worry about building complex `AsyncTask` implementations or directly managing multiple threads. Because of the way they are implemented, coroutines are much more efficient and less resource intensive than using traditional multi-threading options. Coroutines also make for code that is much easier to write, understand and maintain since it allows code to be written sequentially without having to write callbacks to handle thread related events and results.

Although a relatively recent addition to Kotlin, there is nothing new or innovative about coroutines. Coroutines in one form or another have existed in programming languages since the 1960s and are based on a model known as Communicating Sequential Processes (CSP). Kotlin still uses multi-threading behind the scenes, though it does so highly efficiently.

61.2 Threads vs Coroutines

A problem with threads is that they are a finite resource and expensive in terms of CPU capabilities and system overhead. In the background, a lot of work is involved in creating, scheduling and destroying a thread. Although modern CPUs can run large numbers of threads, the actual number of threads that can be run in parallel at any one time is limited by the number of CPU cores (though newer CPUs have 8 cores, most Android devices contain CPUs with 4 cores). When more threads are required than there are CPU cores, the system has to perform thread scheduling to decide how the execution of these threads is to be shared between the available cores.

To avoid these overheads, instead of starting a new thread for each coroutine and then destroying it when the coroutine exits, Kotlin maintains a pool of active threads and manages how coroutines are assigned to those threads. When an active coroutine is suspended it is saved by the Kotlin runtime and another coroutine resumed to take its place. When the coroutine is resumed, it is simply restored to an existing unoccupied thread within the pool to continue executing until it either completes or is suspended. Using this approach, a limited number

of threads are used efficiently to execute asynchronous tasks with the potential to perform large numbers of concurrent tasks without the inherent performance degeneration that would occur using standard multi-threading.

61.3 Coroutine Scope

All coroutines must run within a specific scope which allows them to be managed as groups instead of as individual coroutines. This is particularly important when canceling and cleaning up coroutines, for example when a Fragment or Activity is destroyed, and ensuring that coroutines do not “leak” (in other words continue running in the background when they are no longer needed by the app). By assigning coroutines to a scope they can, for example, all be canceled in bulk when they are no longer needed.

Kotlin and Android provide some built-in scopes as well as the option to create custom scopes using the `CoroutineScope` class. The built-in scopes can be summarized as follows:

- **GlobalScope** – `GlobalScope` is used to launch top-level coroutines which are tied to the entire lifecycle of the application. Since this has the potential for coroutines in this scope to continue running when not needed (for example when an Activity exits) use of this scope is not recommended for use in Android applications. Coroutines running in `GlobalScope` are considered to be using *unstructured concurrency*.
- **ViewModelScope** – Provided specifically for use in `ViewModel` instances when using the Jetpack architecture `ViewModel` component. Coroutines launched in this scope from within a `ViewModel` instance are automatically canceled by the Kotlin runtime system when the corresponding `ViewModel` instance is destroyed.
- **LifecycleScope** – Every lifecycle owner has associated with it a `LifecycleScope`. This scope is canceled when the corresponding lifecycle owner is destroyed making it particularly useful for launching coroutines from within activities and fragments.

For all other requirements, a custom scope will most likely be used. The following code, for example, creates a custom scope named `myCoroutineScope`:

```
private val myCoroutineScope = CoroutineScope(Dispatchers.Main)
```

The `myCoroutineScope` declares the dispatcher that will be used to run coroutines (though this can be overridden) and must be referenced each time a coroutine is started if it is to be included within the scope. All of the running coroutines in a scope can be canceled via a call to the `cancel()` method of the scope instance:

```
myCoroutineScope.cancel()
```

61.4 Suspend Functions

A suspend function is a special type of Kotlin function that contains the code of a coroutine. It is declared using the Kotlin `suspend` keyword which indicates to Kotlin that the function can be paused and resumed later, allowing long-running computations to execute without blocking the main thread.

The following is an example suspend function:

```
suspend fun mySlowTask() {  
    // Perform long-running task here  
}
```

61.5 Coroutine Dispatchers

Kotlin maintains threads for different types of asynchronous activity and, when launching a coroutine, it will be necessary to select the appropriate dispatcher from the following options:

- **Dispatchers.Main** – Runs the coroutine on the main thread and is suitable for coroutines that need to make changes to the UI and as a general purpose option for performing lightweight tasks.

- **Dispatchers.IO** – Recommended for coroutines that perform network, disk, or database operations.
- **Dispatchers.Default** – Intended for CPU intensive tasks such as sorting data or performing complex calculations.

The dispatcher is responsible for assigning coroutines to appropriate threads and suspending and resuming the coroutine during its lifecycle. In addition to the predefined dispatchers, it is also possible to create dispatchers for your own custom thread pools.

61.6 Coroutine Builders

The coroutine builders bring together all of the components covered so far and launch the coroutines so that they start executing. For this purpose, Kotlin provides the following six builders:

- **launch** – Starts a coroutine without blocking the current thread and does not return a result to the caller. Use this builder when calling a suspend function from within a traditional function, and when the results of the coroutine do not need to be handled (sometimes referred to as “fire and forget” coroutines).
- **async** – Starts a coroutine and allows the caller to wait for a result using the `await()` function without blocking the current thread. Use `async` when you have multiple coroutines that need to run in parallel. The `async` builder can only be used from within another suspend function.
- **withContext** – Allows a coroutine to be launched in a different context from that used by the parent coroutine. A coroutine running using the `Main` context could, for example, launch a child coroutine in the `Default` context using this builder. The `withContext` builder also provides a useful alternative to `async` when returning results from a coroutine.
- **coroutineScope** – The `coroutineScope` builder is ideal for situations where a suspend function launches multiple coroutines that will run in parallel and where some action needs to take place only when all the coroutines reach completion. If those coroutines are launched using the `coroutineScope` builder, the calling function will not return until all child coroutines have completed. When using `coroutineScope`, a failure in any of the coroutines will result in the cancellation of all other coroutines.
- **supervisorScope** – Similar to the `coroutineScope` outlined above, with the exception that a failure in one child does not result in cancellation of the other coroutines.
- **runBlocking** – Starts a coroutine and blocks the current thread until the coroutine reaches completion. This is typically the exact opposite of what is wanted from coroutines but is useful for testing code and when integrating legacy code and libraries. Otherwise to be avoided.

61.7 Jobs

Each call to a coroutine builder such as `launch` or `async` returns a `Job` instance which can, in turn, be used to track and manage the lifecycle of the corresponding coroutine. Subsequent builder calls from within the coroutine create new `Job` instances which will become children of the immediate parent `Job` forming a parent-child relationship tree where canceling a parent `Job` will recursively cancel all its children. Canceling a child does not, however, cancel the parent, though an uncaught exception within a child created using the `launch` builder may result in the cancellation of the parent (this is not the case for children created using the `async` builder which encapsulates the exception in the result returned to the parent).

The status of a coroutine can be identified by accessing the `isActive`, `isCompleted` and `isCancelled` properties of the associated `Job` object. In addition to these properties, several methods are also available on a `Job` instance. A `Job` and all of its children may, for example, be canceled by calling the `cancel()` method of the `Job` object, while a call to the `cancelChildren()` method will cancel all child coroutines.

The *join()* method can be called to suspend the coroutine associated with the job until all of its child jobs have completed. To perform this task and cancel the Job once all child jobs have completed, simply call the *cancelAndJoin()* method.

This hierarchical Job structure together with coroutine scopes form the foundation of structured concurrency, the goal of which is to ensure that coroutines do not run for longer than they are required without the need to manually keep references to each coroutine.

61.8 Coroutines – Suspending and Resuming

To gain a better understanding of coroutine suspension, it helps to see some examples of coroutines in action. To start with, let's assume a simple Android app containing a button that, when clicked, calls a function named *startTask()*. It is the responsibility of this function to call a suspend function named *performSlowTask()* using the Main coroutine dispatcher. The code for this might read as follows:

```
private val myCoroutineScope = CoroutineScope(Dispatchers.Main)

fun startTask(view: View) {
    myCoroutineScope.launch(Dispatchers.Main) {
        performSlowTask()
    }
}
```

In the above code, a custom scope is declared and referenced in the call to the launch builder which, in turn, calls the *performSlowTask()* suspend function. Since *startTask()* is not a suspend function, the coroutine must be started using the launch builder instead of the async builder.

Next, we can declare the *performSlowTask()* suspend function as follows:

```
suspend fun performSlowTask() {
    Log.i(TAG, "performSlowTask before")
    delay(5_000) // simulates long-running task
    Log.i(TAG, "performSlowTask after")
}
```

As implemented, all the function does is output diagnostic messages before and after performing a 5-second delay, simulating a long-running task. While the 5-second delay is in effect, the user interface will continue to be responsive because the main thread is not being blocked. To understand why it helps to explore what is happening behind the scenes.

First, the *startTask()* function is executed and launches the *performSlowTask()* suspend function as a coroutine. This function then calls the Kotlin *delay()* function passing through a time value. In fact, the built-in Kotlin *delay()* function is itself implemented as a suspend function so is also launched as a coroutine by the Kotlin runtime environment. The code execution has now reached what is referred to as a suspend point which will cause the *performSlowTask()* coroutine to be suspended while the delay coroutine is running. This frees up the thread on which *performSlowTask()* was running and returns control to the main thread so that the UI is unaffected.

Once the *delay()* function reaches completion, the suspended coroutine will be resumed and restored to a thread from the pool where it can display the Log message and return to the *startTask()* function.

When working with coroutines in Android Studio suspend points within the code editor are marked as shown in the figure below:

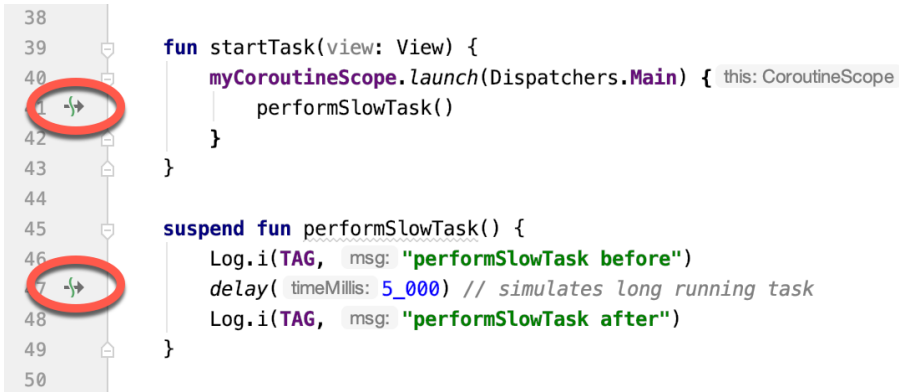


Figure 61-1

61.9 Returning Results from a Coroutine

The above example ran a suspend function as a coroutine but did not demonstrate how to return results. Suppose, however, that the *performSlowTask()* function is required to return a string value which is to be displayed to the user via a *TextView* object.

To do this, we need to rewrite the suspend function to return a *Deferred* object. A *Deferred* object is essentially a commitment to provide a value at some point in the future. By calling the *await()* function on the *Deferred* object, the Kotlin runtime will deliver the value when it is returned by the coroutine. The code in our *startTask()* function might, therefore, be rewritten as follows:

```

fun startTask(view: View) {

    coroutineScope.launch(Dispatchers.Main) {
        statusText.text = performSlowTask().await()
    }
}

```

The problem now is that we are having to use the launch builder to start the coroutine since *startTask()* is not a suspend function. As outlined earlier in this chapter, it is only possible to return results when using the *async* builder. To get around this, we have to adapt the suspend function to use the *async* builder to start another coroutine that returns a *Deferred* result:

```

suspend fun performSlowTask(): Deferred<String> =
    coroutineScope.async(Dispatchers.Default) {
        Log.i(TAG, "performSlowTask before")
        delay(5_000)
        Log.i(TAG, "performSlowTask after")
        return@async "Finished"
    }

```

Now when the app runs, the “Finished” result string will be displayed on the *TextView* object when the *performSlowTask()* coroutine completes. Once again, the wait for the result will take place in the background without blocking the main thread.

61.10 Using withContext

As we have seen, coroutines are launched within a specified scope and using a specific dispatcher. By default, any child coroutines will inherit the same dispatcher as that used by the parent. Consider the following code

designed to call multiple functions from within a suspend function:

```
fun startTask(view: View) {

    coroutineScope.launch(Dispatchers.Main) {
        performTasks()
    }
}

suspend fun performTasks() {
    performTask1()
    performTask2()
    performTask3()
}

suspend fun performTask1() {
    Log.i(TAG, "Task 1 ${Thread.currentThread().name}")
}

suspend fun performTask2() {
    Log.i(TAG, "Task 2 ${Thread.currentThread().name}")
}

suspend fun performTask3 () {
    Log.i(TAG, "Task 3 ${Thread.currentThread().name}")
}
```

Since the *performTasks()* function was launched using the Main dispatcher, all three of the functions will default to the main thread. To prove this, the functions have been written to output the name of the thread in which they are running. On execution, the Logcat panel will contain the following output:

```
Task 1 main
Task 2 main
Task 3 main
```

Imagine, however, that the *performTask2()* function performs some network intensive operations more suited to the IO dispatcher. This can easily be achieved using the *withContext* launcher which allows the context of a coroutine to be changed while still staying in the same coroutine scope. The following change switches the *performTask2()* coroutine to an IO thread:

```
suspend fun performTasks() {
    performTask1()
    withContext(Dispatchers.IO) { performTask2() }
    performTask3()
}
```

When executed, the output will read as follows indicating that the Task 2 coroutine is no longer on the main thread:

```
Task 1 main
Task 2 DefaultDispatcher-worker-1
```

Task 3 main

The `withContext` builder also provides an interesting alternative to using the `async` builder and the `Deferred` object `await()` call when returning a result. Using `withContext`, the code from the previous section can be rewritten as follows:

```
fun startTask(view: View) {

    coroutineScope.launch(Dispatchers.Main) {
        statusText.text = performSlowTask()
    }
}

suspend fun performSlowTask(): String =
    withContext(Dispatchers.Main) {
        Log.i(TAG, "performSlowTask before")
        delay(5_000)
        Log.i(TAG, "performSlowTask after")

        return@withContext "Finished"
    }
}
```

61.11 Coroutine Channel Communication

Channels provide a simple way to implement communication between coroutines including streams of data. In the simplest form this involves the creation of a `Channel` instance and calling the `send()` method to send the data. Once sent, transmitted data can be received in another coroutine via a call to the `receive()` method of the same `Channel` instance.

The following code, for example, passes six integers from one coroutine to another:

```
.
.
import kotlinx.coroutines.channels.*
.
.
val channel = Channel<Int>()

suspend fun channelDemo() {
    coroutineScope.launch(Dispatchers.Main) { performTask1() }
    coroutineScope.launch(Dispatchers.Main) { performTask2() }
}

suspend fun performTask1() {
    (1..6).forEach {
        channel.send(it)
    }
}
```

An Introduction to Kotlin Coroutines

```
suspend fun performTask2() {  
    repeat(6) {  
        Log.d(TAG, "Received: ${channel.receive()}")  
    }  
}
```

When executed, the following logcat output will be generated:

```
Received: 1  
Received: 2  
Received: 3  
Received: 4  
Received: 5  
Received: 6
```

61.12 Summary

Kotlin coroutines provide a simpler and more efficient approach to performing asynchronous tasks than that offered by traditional multi-threading. Coroutines allow asynchronous tasks to be implemented in a structured way without the need to implement the callbacks associated with typical thread-based tasks. This chapter has introduced the basic concepts of coroutines including jobs, scope, builders, suspend functions, structured concurrency and channel-based communication.

69. The Android Room Persistence Library

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

69.1 Revisiting Modern App Architecture

The chapter entitled “*Modern Android App Architecture with Jetpack*” introduced the concept of modern app architecture and stressed the importance of separating different areas of responsibility within an app. The diagram illustrated in Figure 69-1 outlines the recommended architecture for a typical Android app:

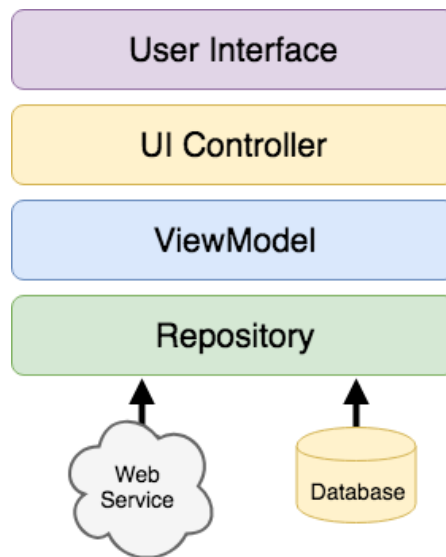


Figure 69-1

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

69.2 Key Elements of Room Database Persistence

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

69.2.1 Repository

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

69.2.2 Room Database

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

69.2.3 Data Access Object (DAO)

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

69.2.4 Entities

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

69.2.5 SQLite Database

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

The architecture diagram in Figure 69-2 illustrates the way in which these different elements interact to provide Room-based database storage within an Android app:

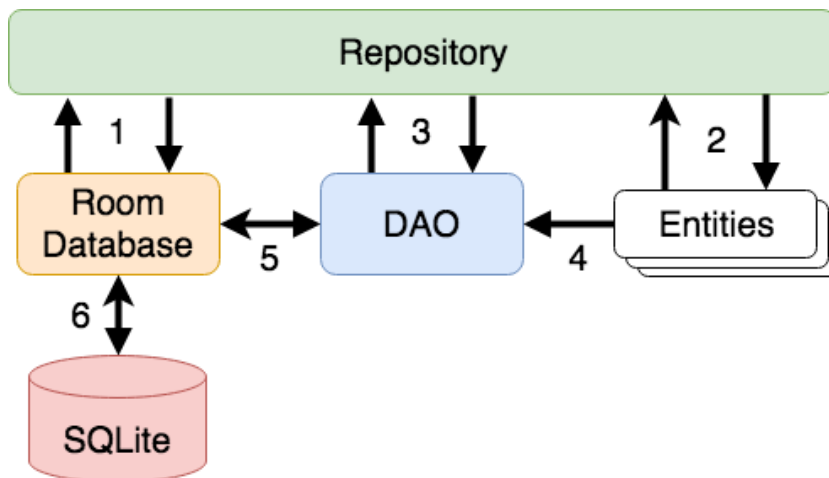


Figure 69-2

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

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

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

69.3 Understanding Entities

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

```
class Customer {

    var id: Int = 0
    var name: String? = null
    var address: String? = null

    constructor() {}

    constructor(id: Int, name: String, address: String) {
        this.id = id
        this.name = name
        this.address = address
    }
    constructor(name: String, address: String) {
        this.name = name
        this.address = address
    }
}
```

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

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

```

@PrimaryKey(autoGenerate = true)
@NonNull
@ColumnInfo(name = "customerId")
var id: Int = 0

@ColumnInfo(name = "customerName")
var name: String? = null
var address: String? = null

constructor() {}

constructor(id: Int, name: String, address: String) {
    this.id = id
    this.name = name
    this.address = address
}

constructor(name: String, address: String) {
    this.name = name
    this.address = address
}
}

```

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

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

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

```

@PrimaryKey(autoGenerate = true)
@NonNull
@ColumnInfo(name = "customerId")
var id: Int = 0

```

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

```

@Ignore
var MyString: String? = null

```

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

existing Customer entity as follows:

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

class Purchase {

    @PrimaryKey(autoGenerate = true)
    @NonNull
    @ColumnInfo(name = "purchaseId")
    var purchaseId: Int = 0

    @ColumnInfo(name = "buyerId")
    var buyerId: Int = 0

    .
    .
}
```

Note that the foreign key declaration also specifies the action to be taken when a parent record is deleted or updated. Available options are CASCADE, NO_ACTION, RESTRICT, SET_DEFAULT and SET_NULL.

69.4 Data Access Objects

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

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

```
@Dao
interface CustomerDao {
}
```

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

```
@Dao
interface CustomerDao {
    @Query("SELECT * FROM customers")
    fun getAllCustomers(): LiveData<List<Customer>>
}
```

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

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

The Android Room Persistence Library

class):

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

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

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

```
@Insert
fun addCustomer(Customer customer)
```

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

```
@Insert
fun insertCustomers(Customer... customers)
```

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

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

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

```
@Delete
fun deleteCustomers(Customer... customers)
```

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

```
@Update
fun updateCustomers(Customer... customers)
```

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

```
@Delete
fun deleteCustomers(Customer... customers): int
```

69.5 The Room Database

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

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

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

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

```

import android.arch.persistence.room.RoomDatabase

@Database(entities = [(Customer::class)], version = 1)
abstract class CustomerRoomDatabase: RoomDatabase() {
    abstract fun customerDao(): CustomerDao

    companion object {

        private var INSTANCE: CustomerRoomDatabase? = null

        internal fun getDatabase(context: Context): CustomerRoomDatabase? {
            if (INSTANCE == null) {
                synchronized(CustomerRoomDatabase::class.java) {
                    if (INSTANCE == null) {
                        INSTANCE =
                            Room.databaseBuilder(
                                context.applicationContext,
                                CustomerRoomDatabase::class.java,
                                "customer_database").build()
                    }
                }
            }
            return INSTANCE
        }
    }
}

```

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

69.6 The Repository

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

```

class CustomerRepository(application: Application) {

    private var customerDao: CustomerDao?

    init {
        val db: CustomerRoomDatabase? =
            CustomerRoomDatabase.getDatabase(application)
        customerDao = db?.customerDao()
    }
}

```

The Android Room Persistence Library

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

```
val allCustomers: LiveData<List<Customer>>?
allCustomers = customerDao.getAllCustomers()
```

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

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

Since some database transactions may take a longer time to complete, running the operations on a separate thread avoids the app appearing to lock up. As will be demonstrated in the chapter entitled “*An Android Room Database and Repository Tutorial*”, this problem can be easily resolved by making use of coroutines (for more information or a reminder of how to use coroutines, refer back to the chapter entitled “*An Introduction to Kotlin Coroutines*”).

69.7 In-Memory Databases

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

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

```
// Create a file storage based database
INSTANCE = Room.databaseBuilder<CustomerRoomDatabase>(context.applicationContext,
    CustomerRoomDatabase::class.java, "customer_database")
    .build()

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

69.8 Database Inspector

Android Studio includes a Database Inspector tool window which allows the Room databases associated with running apps to be viewed, searched and modified as shown in Figure 69-3:

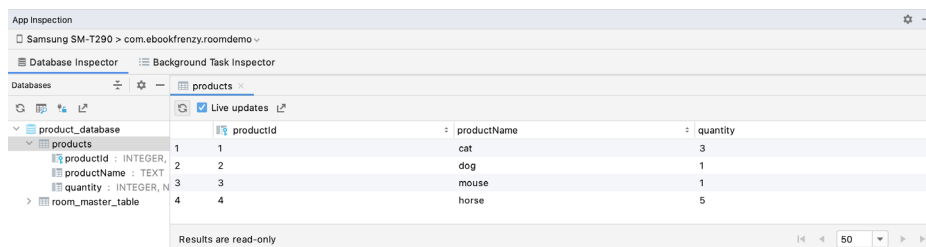


Figure 69-3

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

69.9 Summary

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

With the basics of SQLite and the Room architecture component covered, the next step is to create an example app that puts this theory into practice. Since the user interface for the example application will require a forms based layout, the next chapter, entitled “*An Android TableLayout and TableRow Tutorial*”, will detour slightly from the core topic by introducing the basics of the TableLayout and TableRow views.

87. An Overview of Android In-App Billing

In the early days of mobile applications for operating systems such as Android and iOS, the most common method for earning revenue was to charge an upfront fee to download and install the application. Another revenue opportunity was soon introduced in the form of embedding advertising within applications. Perhaps the most common and lucrative option is now to charge the user for purchasing items from within the application after it has been installed. This typically takes the form of access to a higher level in a game, acquiring virtual goods or currency, or subscribing to premium content in the digital edition of a magazine or newspaper.

Google provides support for the integration of in-app purchasing through the Google Play In-App Billing API and the Play Console. This chapter will provide an overview of in-app billing and outline how to integrate in-app billing into your Android projects. Once these topics have been explored, the next chapter will walk you through creating an example app that includes in-app purchasing features.

87.1 Preparing a Project for In-App Purchasing

Building in-app purchasing into an app will require a Google Play Developer Console account, details of which were covered previously in the “*Creating, Testing and Uploading an Android App Bundle*” chapter. You will also need to register a Google merchant account and configure your payment settings. These settings can be found by navigating to *Setup -> Payments profile* in the Play Console. Note that merchant registration is not available in all countries. For details, refer to the following page:

<https://support.google.com/googleplay/android-developer/answer/9306917>

The app will then need to be uploaded to the console and enabled for in-app purchasing. The console will not activate in-app purchasing support for an app, however, unless the Google Play Billing Library has been added to the module-level *build.gradle* file. When working with Kotlin, the Google Play Kotlin Extensions Library is also recommended:

```
dependencies {  
    .  
    .  
    implementation 'com.android.billingclient:billing:<latest version>'  
    implementation 'com.android.billingclient:billing-ktx:<latest version>'  
    .  
    .  
}
```

Once the build file has been modified and the app bundle uploaded to the console, the next step is to add in-app products or subscriptions for the user to purchase.

87.2 Creating In-App Products and Subscriptions

Products and subscriptions are created and managed using the options listed beneath the Monetize section of the Play Console navigation panel as highlighted in Figure 87-1 below:

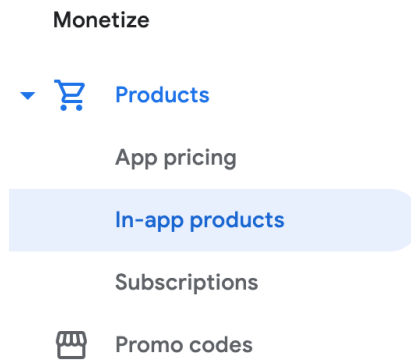


Figure 87-1

Each product or subscription needs an ID, title, description, and pricing information. Purchases fall into the categories of *consumable* (the item must be purchased each time it is required by the user such as virtual currency in a game), *non-consumable* (only needs to be purchased once by the user such as content access), and *subscription*-based. Consumable and non-consumable products are collectively referred to as *managed products*.

Subscriptions are useful for selling an item that needs to be renewed on a regular schedule such as access to news content or the premium features of an app. When creating a subscription, a *base plan* is defined specifying the price, renewal period (monthly, annually, etc.), and whether the subscription auto-renews. Users can also be provided with discount *offers* and given the option of pre-purchasing a subscription.

87.3 Billing Client Initialization

Communication between your app and the Google Play Billing Library is handled by a `BillingClient` instance. In addition, `BillingClient` includes a set of methods that can be called to perform both synchronous and asynchronous billing-related activities. When the billing client is initialized, it will need to be provided with a reference to a `PurchasesUpdatedListener` callback handler. The client will call this handler to notify your app of the results of any purchasing activity. To avoid duplicate notifications, it is recommended to have only one `BillingClient` instance per app.

A `BillingClient` instance can be created using the `newBuilder()` method, passing through the current activity or fragment context. The purchase update handler is then assigned to the client via the `setListener()` method:

```
private val purchasesUpdatedListener =
    PurchasesUpdatedListener { billingResult, purchases ->
        if (billingResult.responseCode ==
            BillingClient.BillingResponseCode.OK
            && purchases != null
        ) {
            for (purchase in purchases) {
                // Process the purchases
            }
        } else if (billingResult.responseCode ==
            BillingClient.BillingResponseCode.USER_CANCELED
        ) {
            // Purchase cancelled by user
        } else {
```

```

        // Handle errors here
    }
}

billingClient = BillingClient.newBuilder(this)
    .setListener(purchasesUpdatedListener)
    .enablePendingPurchases()
    .build()

```

87.4 Connecting to the Google Play Billing Library

After the successful creation of the Billing Client, the next step is to initialize a connection to the Google Play Billing Library. To establish this connection, a call needs to be made to the *startConnection()* method of the billing client instance. Since the connection is performed asynchronously, a *BillingClientStateListener* handler needs to be implemented to receive a callback indicating whether the connection was successful. Code should also be added to override the *onBillingServiceDisconnected()* method. This is called if the connection to the Billing Library is lost and can be used to report the problem to the user and retry the connection.

Once the setup and connection tasks are complete, the *BillingClient* instance will make a call to the *onBillingSetupFinished()* method which can be used to check that the client is ready:

```

billingClient.startConnection(object : BillingClientStateListener {
    override fun onBillingSetupFinished(
        billingResult: BillingResult
    ) {
        if (billingResult.responseCode ==
            BillingClient.BillingResponseCode.OK
        ) {
            // Connection successful
        } else {
            // Connection failed
        }
    }

    override fun onBillingServiceDisconnected() {
        // Connection to billing service lost
    }
})

```

87.5 Querying Available Products

Once the billing environment is initialized and ready to go, the next step is to request the details of the products or subscriptions that are available for purchase. This is achieved by making a call to the *queryProductDetailsAsync()* method of the *BillingClient* and passing through an appropriately configured *QueryProductDetailsParams* instance containing the product ID and type (*ProductType.SUBS* for a subscription or *ProductType.INAPP* for a managed product):

```

val queryProductDetailsParams = QueryProductDetailsParams.newBuilder()
    .setProductList(
        ImmutableList.of(
            QueryProductDetailsParams.Product.newBuilder()

```

An Overview of Android In-App Billing

```
                .setProductId(productId)
                .setProductType(
                    BillingClient.ProductType.INAPP
                )
                .build()
            )
        )
        .build()

billingClient.queryProductDetailsAsync(
    queryProductDetailsParams
) { billingResult, productDetailsList ->
    if (!productDetailsList.isEmpty()) {
        // Process list of matching products
    } else {
        // No product matches found
    }
}
```

The *queryProductDetailsAsync()* method is passed a *ProductDetailsResponseListener* handler (in this case in the form of a lambda code block) which, in turn, is called and passed a list of *ProductDetail* objects containing information about the matching products. For example, we can call methods on these objects to get information such as the product name, title, description, price, and offer details.

87.6 Starting the Purchase Process

Once a product or subscription has been queried and selected for purchase by the user, the purchase process is ready to be launched. We do this by calling the *launchBillingFlow()* method of the *BillingClient*, passing through as arguments the current activity and a *BillingFlowParams* instance configured with the *ProductDetail* object for the item being purchased.

```
val billingFlowParams = BillingFlowParams.newBuilder()
    .setProductDetailsParamsList(
        ImmutableList.of(
            BillingFlowParams.ProductDetailsParams.newBuilder()
                .setProductDetails(productDetails)
                .build()
        )
    )
    .build()

billingClient.launchBillingFlow(this, billingFlowParams)
```

The success or otherwise of the purchase operation will be reported via a call to the *PurchasesUpdatedListener* callback handler outlined earlier in the chapter.

87.7 Completing the Purchase

When purchases are successful, the *PurchasesUpdatedListener* handler will be passed a list containing a *Purchase* object for each item. You can verify that the item has been purchased by calling the *getPurchaseState()* method of the *Purchase* instance as follows:

```

if (purchase.getPurchaseState() == Purchase.PurchaseState.PURCHASED) {
    // Purchase completed.
} else if (purchase.getPurchaseState() == Purchase.PurchaseState.PENDING) {
    // Payment is still pending
}

```

Note that your app will only support pending purchases if a call is made to the *enablePendingPurchases()* method during initialization. A pending purchase will remain so until the user completes the payment process.

When the purchase of a non-consumable item is complete, it will need to be acknowledged to prevent a refund from being issued to the user. This requires the *purchase token* for the item which is obtained via a call to the *getPurchaseToken()* method of the Purchase object. This token is used to create an AcknowledgePurchaseParams instance together with an AcknowledgePurchaseResponseListener handler. Managed product purchases and subscriptions are acknowledged by calling the BillingClient's *acknowledgePurchase()* method as follows:

```

billingClient.acknowledgePurchase(acknowledgePurchaseParams,
                                acknowledgePurchaseResponseListener);
val acknowledgePurchaseParams = AcknowledgePurchaseParams.newBuilder()
    .setPurchaseToken(purchase.purchaseToken)
    .build()

val acknowledgePurchaseResponseListener = AcknowledgePurchaseResponseListener {
    // Check acknowledgement result
}

billingClient.acknowledgePurchase(
    acknowledgePurchaseParams,
    acknowledgePurchaseResponseListener
)

```

For consumable purchases, you will need to notify Google Play when the item has been consumed so that it is available to be repurchased by the user. This requires a configured ConsumeParams instance containing a purchase token and a call to the billing client's *consumePurchase()* method:

```

val consumeParams = ConsumeParams.newBuilder()
    .setPurchaseToken(purchase.purchaseToken)
    .build()

coroutineScope.launch {
    val result = billingClient.consumePurchase(consumeParams)

    if (result.billingResult.responseCode ==
        BillingClient.BillingResponseCode.OK) {
        // Purchase successfully consumed
    }
}

```

87.8 Querying Previous Purchases

When working with in-app billing it is a common requirement to check whether a user has already purchased a product or subscription. A list of all the user's previous purchases of a specific type can be generated by calling

An Overview of Android In-App Billing

the *queryPurchasesAsync()* method of the *BillingClient* instance and implementing a *PurchaseResponseListener*. The following code, for example, obtains a list of all previously purchased items that have not yet been consumed:

```
val queryPurchasesParams = QueryPurchasesParams.newBuilder()
    .setProductType(BillingClient.ProductType.INAPP)
    .build()

billingClient.queryPurchasesAsync(
    queryPurchasesParams,
    purchasesListener
)
.
.
private val purchasesListener =
    PurchasesResponseListener { billingResult, purchases ->

        if (!purchases.isEmpty()) {
            // Access existing active purchases
        } else {
            // No
        }
    }
```

To obtain a list of active subscriptions, change the *ProductType* value from *INAPP* to *SUBS*.

Alternatively, to obtain a list of the most recent purchases for each product, make a call to the *BillingClient* *queryPurchaseHistoryAsync()* method:

```
val queryPurchaseHistoryParams = QueryPurchaseHistoryParams.newBuilder()
    .setProductType(BillingClient.ProductType.INAPP)
    .build()

billingClient.queryPurchaseHistoryAsync(queryPurchaseHistoryParams) {
    billingResult, historyList ->
        // Process purchase history list
}
```

87.9 Summary

In-app purchases provide a way to generate revenue from within Android apps by selling virtual products and subscriptions to users. In this chapter, we have explored managed products and subscriptions and explained the difference between consumable and non-consumable products. In-app purchasing support is added to an app using the Google Play In-app Billing Library and involves creating and initializing a billing client on which methods are called to perform tasks such as making purchases, listing available products, and consuming existing purchases. The next chapter contains a tutorial demonstrating the addition of in-app purchases to an Android Studio project.

Index

Symbols

?. 93
 <application> 492
 <fragment> 281
 <fragment> element 281
 <receiver> 470
 <service> 492, 498, 505
 Code Reformatting 73
 :: operator 95
 .well-known folder 443, 466, 680

A

AbsoluteLayout 166
 ACCESS_COARSE_LOCATION permission 590
 ACCESS_FINE_LOCATION permission 590
 acknowledgePurchase() method 719
 ACTION_DOWN 258
 ACTION_MOVE 258
 ACTION_POINTER_DOWN 258
 ACTION_POINTER_UP 258
 ACTION_UP 258
 ACTION_VIEW 461
 Active / Running state 142
 Activity 79, 145
 adding views in Java code 239
 class 145
 creation 14
 Entire Lifetime 149
 Foreground Lifetime 149
 lifecycle methods 147
 lifecycles 139
 returning data from 440
 state change example 153
 state changes 145
 states 142

Visible Lifetime 149
 ActivityCompat class 595
 Activity Lifecycle 141
 Activity Manager 78
 ActivityResultLauncher 441
 Activity Stack 141
 Actual screen pixels 230
 adb
 command-line tool 57
 connection testing 63
 device pairing 61
 enabling on Android devices 57
 Linux configuration 60
 list devices 57
 macOS configuration 58
 overview 57
 restart server 58
 testing connection 63
 WiFi debugging 61
 Windows configuration 59
 Wireless debugging 61
 Wireless pairing 61
 addCategory() method 469
 addMarker() method 644
 addView() method 234
 ADD_VOICEMAIL permission 590
 android
 exported 493
 gestureColor 274
 layout_behavior property 433
 onClick 283
 process 493, 505
 uncertainGestureColor 274
 Android
 Activity 79
 architecture 75
 events 251
 intents 80
 onClick Resource 251

Index

- runtime 76
- SDK Packages 6
- android.app 76
- Android Architecture Components 297
- android.content 76
- android.content.Intent 439
- android.database 76
- Android Debug Bridge. *See* ADB
- Android Design Support Library 403
- Android Development
 - System Requirements 3
- Android Devices
 - designing for different 165
- android.graphics 76
- android.hardware 76
- android.intent.action 475
- android.intent.action.BOOT_COMPLETED 494
- android.intent.action.MAIN 461
- android.intent.category.LAUNCHER 461
- Android Libraries 76
- android.media 77
- Android Monitor tool window 32
- Android Native Development Kit 77
- android.net 77
- android.opengl 77
- android.os 77
- android.permission.RECORD_AUDIO 599
- android.print 77
- Android Project
 - create new 13
- android.provider 77
- Android SDK Location
 - identifying 9
- Android SDK Manager 8, 10
- Android SDK Packages
 - version requirements 8
- Android SDK Tools
 - command-line access 9
 - Linux 11
 - macOS 11
 - Windows 7 10
 - Windows 8 10
- Android Software Stack 75
- Android Studio
 - changing theme 54
 - downloading 3
 - Editor Window 49
 - installation 4
 - Linux installation 5
 - macOS installation 4
 - Main Window 48
 - Menu Bar 48
 - Navigation Bar 48
 - Project tool window 49
 - setup wizard 5
 - Status Bar 49
 - Toolbar 48
 - Tool window bars 50
 - tool windows 49
 - updating 12
 - Welcome Screen 47
 - Windows installation 4
- android.text 77
- android.util 77
- android.view 77
- android.view.View 168
- android.view.ViewGroup 165, 168
- Android Virtual Device. *See* AVD
 - overview 27
- Android Virtual Device Manager 27
- android.webkit 77
- android.widget 77
- AndroidX libraries 770
- API Key 635
- APK analyzer 712
- APK file 706
 - split 734
- APK File
 - analyzing 712
- APK Signing 770
- APK Wizard dialog 704
- App Architecture
 - modern 297
- AppBar

- anatomy of 431
- appbar_scrolling_view_behavior 433
- App Bundles 701
 - creating 706
 - overview 701
 - revisions 711
 - uploading 708
- AppCompatActivity class 146
- App Inspector 51
- Application
 - stopping 32
- Application Context 81
- Application Framework 77
- Application Manifest 81
- Application Resources 81
- App Link
 - Adding Intent Filter 688
 - Assistant 683
 - Digital Asset Links file 680, 443
 - Intent Filter Handling 688
 - Intent Filters 679
 - Intent Handling 680
 - Testing 692
 - tutorial 683
 - URL Mapping 685
- App Link Assistant 683
- App Links 679
 - auto verification 442
 - autoVerify 443
 - manually enabling 445
 - overview 679
- Apply Changes 247
 - Apply Changes and Restart Activity 247
 - Apply Code Changes 247
 - fallback settings 249
 - options 247
 - Run App 247
 - tutorial 249
- applyToActivitiesIfAvailable() method 767
- Architecture Components 297
- ART 76
- as 95

- as? 95
- asFlow() builder 510
- assetlinks.json , 680, 443
- asSharedFlow() 520
- asStateFlow() 519
- async 479
- Attribute Keyframes 370
- Audio
 - supported formats 597
- Audio Playback 597
- Audio Recording 597
- Autoconnect Mode 196
- Automatic Link Verification 442, 465
- autoVerify 443, 688
- AVD

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

B

- Background Process 140
- Barriers 190
 - adding 208
 - constrained views 190
- Base APK file 734
- Baseline Alignment 189
- beginTransaction() method 282
- BillingClient 720
 - acknowledgePurchase() method 719
 - consumeAsync() method 719
 - getPurchaseState() method 718

Index

- initialization 716, 724
- launchBillingFlow() method 718
- queryProductDetailsAsync() method 717
- queryPurchasesAsync() method 720
- startConnection() method 717
- BillingResult 731
 - getDebugMessage() 731
- Binding Expressions 317
 - one-way 317
 - two-way 318
- BIND_JOB_SERVICE permission 493
- bindService() method 491, 495, 499
- Biometric Authentication 693
 - callbacks 697
 - overview 693
 - tutorial 693
- Biometric Prompt 698
- Bitwise AND 101
- Bitwise Inversion 100
- Bitwise Left Shift 102
- Bitwise OR 101
- Bitwise Right Shift 102
- Bitwise XOR 101
- black activity 14
- Blank template 169
- Blueprint view 195
- BODY_SENSORS permission 590
- Bookmarks 51
- Boolean 88
- Bound Service 491, 495
 - adding to a project 496
 - Implementing the Binder 496
 - Interaction options 495
- BoundService class 497
- Broadcast Intent 469
 - example 472
 - overview 80, 469
 - sending 472
 - Sticky 471
- Broadcast Receiver 469
 - adding to manifest file 474
 - creation 473

- overview 80, 470
- BroadcastReceiver class 470
- BroadcastReceiver superclass 473
- buffer() operator 513
- Build tool window 51
- Build Variants 51, 770
 - tool window 51
- Bundle class 162
- Bundled Notifications 618

C

- Calendar permissions 590
- CALL_PHONE permission 590
- CAMERA permission 590
- Camera permissions 590
- CameraUpdateFactory class
 - methods 645
- cancelAndJoin() 480
- cancelChildren() 479
- CancellationSignal 698
- Canvas class 674
- CardView
 - example 423
 - layout file 421
 - responding to selection of 429
- CardView class 421
- C/C++ Libraries 77
- Chain bias 216
- chain head 188
- chains 188
- Chains
 - creation of 213
- Chain style
 - changing 215
- chain styles 188
- Char 88
- CharSequence 163
- CheckBox 165
- checkSelfPermission() method 594
- Circle class 631
- Code completion 68
- Code Editor

- basics 65
- Code completion 68
- Code Generation 70
- Code Reformatting 73
- Document Tabs 66
- Editing area 66
- Gutter Area 66
- Live Templates 74
- Splitting 67
- Statement Completion 69
- Status Bar 67
- Code Generation 70
- code samples
 - download 1
- cold boot 42
- Cold flows 518
- CollapsingToolbarLayout
 - example 435
 - introduction 434
 - parallax mode 434
 - pin mode 434
 - setting scrim color 437
 - setting title 437
 - with image 434
- collectLatest() operator 512
- collect() operator 511
- Color class 675
- COLOR_MODE_COLOR 650, 670
- COLOR_MODE_MONOCHROME 650, 670
- com.android.application 737
- com.android.dynamic-feature 737
- combine() operator 517
- Common Gestures 263
 - detection 263
- Communicating Sequential Processes 477
- Companion Objects 125
- Component tree 17
- Configuration APK file 734
- conflate() operator 512
- Constraint Bias 187
 - adjusting 200
- ConstraintLayout
 - advantages of 193
 - Availability 194
 - Barriers 190
 - Baseline Alignment 189
 - chain bias 216
 - chain head 188
 - chains 188
 - chain styles 188
 - Constraint Bias 187
 - Constraints 185
 - conversion to 212
 - convert to MotionLayout 377
 - deleting constraints 200
 - guidelines 206
 - Guidelines 190
 - manual constraint manipulation 197
 - Margins 186, 201
 - Opposing Constraints 186, 202
 - overview of 185
 - Packed chain 189, 216
 - ratios 193, 217
 - Spread chain 188
 - Spread inside 215
 - Spread inside chain 188
 - tutorial 221
 - using in Android Studio 195
 - Weighted chain 188, 216
 - Widget Dimensions 189, 204
 - Widget Group Alignment 211
- ConstraintLayout chains
 - creation of 213
 - in layout editor 213
- ConstraintLayout Chain style
 - changing 215
- Constraints
 - deleting 200
- ConstraintSet
 - addToHorizontalChain() method 236
 - addToVerticalChain() method 236
 - alignment constraints 235
 - apply to layout 234
 - applyTo() method 234

Index

- centerHorizontally() method 235
- centerVertically() method 235
- chains 235
- clear() method 236
- clone() method 235
- connect() method 234
- connect to parent 234
- constraint bias 235
- copying constraints 235
- create 234
- create connection 234
- createHorizontalChain() method 235
- createVerticalChain() method 235
- guidelines 236
- removeFromHorizontalChain() method 236
- removeFromVerticalChain() method 236
- removing constraints 236
- rotation 237
- scaling 236
- setGuidelineBegin() method 236
- setGuidelineEnd() method 236
- setGuidelinePercent() method 236
- setHorizontalBias() method 235
- setRotationX() method 237
- setRotationY() method 237
- setScaleX() method 236
- setScaleY() method 236
- setTransformPivot() method 237
- setTransformPivotX() method 237
- setTransformPivotY() method 237
- setVerticalBias() method 235
- sizing constraints 235
- tutorial 239
- view IDs 241
- ConstraintSet class 233, 234
- ConstraintSet.PARENT_ID 234
- Constraint Sets 234
- ConstraintSets
 - configuring 366
- consumeAsync() method 719
- ConsumeParams 729
- Contacts permissions 590
- container view 165
- Content Provider 78
 - overview 81
- Context class 81
- CoordinatorLayout 166, 431, 433
- Coroutine Builders 479
 - async 479
 - coroutineScope 479
 - launch 479
 - runBlocking 479
 - supervisorScope 479
 - withContext 479
- Coroutine Dispatchers 478
- Coroutines 477, 509
 - adding libraries 485
 - channel communication 483
 - GlobalScope 478
 - returning results 481
 - Suspend Functions 478
 - suspending 480
 - tutorial 485
 - ViewModelScope 478
 - vs. Threads 477
- coroutineScope 479
- Coroutine Scope 478
- createPrintDocumentAdapter() method 665
- Custom Accessors 123
- Custom Attribute 367
- Custom Document Printing 653, 665
- Custom Gesture
 - recognition 269
- Custom Print Adapter
 - implementation 667
- Custom Print Adapters 665
- Custom Theme
 - building 761
- Cycle Editor 395
- Cycle Keyframe 375
- Cycle Keyframes
 - overview 391

D

- dangerous permissions 589
 - list of 590
- Dark Theme 32
 - enable on device 32
- Data Access Object (DAO) 540
- Database Inspector 546, 570
 - live updates 570
 - SQL query 570
- Database Rows 534
- Database Schema 533
- Database Tables 533
- Data binding
 - binding expressions 317
- Data Binding 300
 - binding classes 316
 - enabling 322
 - event and listener binding 318
 - key components 313
 - overview 313
 - tutorial 321
 - variables 316
 - with LiveData 300
- DDMS 32
- Debugging
 - enabling on device 57
- debug.keystore file 443, 465
- Default Function Parameters 115
- DefaultLifecycleObserver 336, 339
- deltaRelative 371
- Density-independent pixels 229
- Density Independent Pixels
 - converting to pixels 244
- Device Definition
 - custom 182
- Device File Explorer 51
- device frame 35
- Device Manager 51
- Device Mirroring 63
 - enabling 63
- device pairing 61
- Digital Asset Links file 443, 680, 443
- Direct Reply Input 627

- Direct Reply Notification 621
- Dispatchers.Default 479
- Dispatchers.IO 479
- Dispatchers.Main 478
- dp 229
- DROP_LATEST 520
- DROP_OLDEST 520
- Dynamic Colors
 - applyToActivitiesIfAvailable() method 767
 - enabling 767
 - enabling in Android 767
- Dynamic Delivery 736
- Dynamic Feature APK 734
- Dynamic Feature Module
 - architecture 733
 - overview 733
 - removal 757
 - tutorial 743
- Dynamic Feature Modules
 - deferred installation 739
 - handling of large 740
- Dynamic Feature Support
 - adding to project 743
- Dynamic State 147
 - saving 161

E

- Elvis Operator 95
- Empty Process 141
- Empty template 169
- Emulator 51
 - battery 40
 - cellular configuration 40
 - configuring fingerprints 42
 - directional pad 40
 - extended control options 39
 - Extended controls 39
 - fingerprint 40
 - location configuration 40
 - phone settings 40
 - Resizable 44
 - resize 39

Index

- rotate 38
- Screen Record 41
- Snapshots 41
- starting 29
- take screenshot 38
- toolbar 37
- toolbar options 37
- tool window mode 43
- Virtual Sensors 41
- zoom 38
- enablePendingPurchases() method 719
- enabling ADB support 57
- Escape Sequences 89
- ettings.gradle file 770
- Event Handling 251
 - example 252
- Event Listener 253
- Event Listeners 252
- Event Log 51
- Events
 - consuming 255
- explicit
 - intent 80
- explicit intent 439
- Explicit Intent 439
- Extended Control
 - options 39

F

- Favorites
 - tool window 51
- Files
 - switching between 66
- filter() operator 514
- findPointerIndex() method 258
- findViewById() 135
- Fingerprint
 - emulation 42
- Fingerprint authentication
 - device configuration 694
 - permission 694
 - steps to implement 693

- Fingerprint Authentication
 - overview 693
 - tutorial 693
- FLAG_INCLUDE_STOPPED_PACKAGES 469
- flatMapConcat() operator 517
- flatMapMerge() operator 517
- flexible space area 431
- Float 88
- floating action button 14, 170, 403
 - changing appearance of 406
 - margins 404
 - overview of 403
 - removing 171
 - sizes 404
- Flow 509
 - asFlow() builder 510
 - asSharedFlow() 520
 - asStateFlow() 519
 - backgroundn handling 528
 - buffering 512
 - buffer() operator 513
 - builder 510
 - cold 518
 - collect() 511
 - collecting data 511
 - collectLatest() operator 512
 - combine() operator 517
 - conflate() operator 512
 - declaring 510
 - emit() 511
 - emitting data 511
 - filter() operator 514
 - flatMapConcat() operator 517
 - flatMapMerge() operator 517
 - flattening 516
 - flowOf() builder 510
 - flow of flows 516
 - fold() operator 516
 - hot 518
 - intermediate operators 514
 - library requirements 510
 - map() operator 514

- MutableSharedFlow 520
- MutableStateFlow 519
- onEach() operator 518
- reduce() operator 515, 516
- repeatOnLifecycle 530
- SharedFlow 520
- single() operator 512
- StateFlow 519
- terminal flow operators 515
- transform() operator 515
- try/finally 512
- zip() operator 517
- flow builder 510
- flowOf() builder 510
- flow of flows 516
- Flow operators 514
- Flows
 - combining 517
 - Introduction to 509
- Foldable Devices 150
 - multi-resume 150
- Foreground Process 140
- Forward-geocoding 638
- Fragment
 - creation 279
 - event handling 283
 - XML file 279, 280
- FragmentManager class 146
- Fragment Communication 283
- Fragments 279
 - adding in code 282
 - duplicating 412
 - example 287
 - overview 279
- FragmentManager class 415
- FrameLayout 166
- Function Parameters
 - variable number of 115
- Functions 113
- G**
- Geocoder class 637
- Geocoder object 638
- Geocoding 637
- Gesture Builder Application 269
 - building and running 270
- Gesture Detector class 263
- GestureDetectorCompat 266
 - instance creation 266
- GestureDetectorCompat class 263
- GestureDetector.OnDoubleTapListener 263, 264
- GestureDetector.OnGestureListener 264
- GestureLibrary 269
- GestureLibrary class 269
- GestureOverlayView 269
 - configuring color 274
 - configuring multiple strokes 274
- GestureOverlayView class 269
- GesturePerformedListener 269
- Gestures
 - interception of 275
- Gestures File
 - creation 270
 - extract from SD card 271
 - loading into application 272
- GET_ACCOUNTS permission 590
- getAction() method 475
- getDebugMessage() 731
- getFromLocation() method 638
- getId() method 234
- getIntent() method 440
- getPointerCount() method 258
- getPointerId() method 258
- getPurchaseState() method 718
- getService() method 499
- GlobalScope 478
- GNU/Linux 76
- Google Cloud
 - billing account 632
 - Console 632
 - new project 633
- Google Cloud Print 648
- Google Drive
 - printing to 648

Index

- GoogleMap 631
 - map types 641
- GoogleMap.MAP_TYPE_HYBRID 642
- GoogleMap.MAP_TYPE_NONE 642
- GoogleMap.MAP_TYPE_NORMAL 642
- GoogleMap.MAP_TYPE_SATELLITE 642
- GoogleMap.MAP_TYPE_TERRAIN 642
- Google Maps Android API 631
 - Controlling the Map Camera 645
 - displaying controls 642
 - gesture handling 643
 - Map Markers 644
 - overview 631
- Google Maps SDK 631
 - API Key 635
 - Credentials 635
 - enabling 634
 - Maps SDK for Android 635
- Google Play Billing Library 715
- Google Play Console 722
 - Creating an in-app product 722
 - License Testers 723
- Google Play Developer Console 702
- Go to Line:Column 67
- Gradle
 - APK signing settings 774
 - Build Variants 770
 - command line tasks 775
 - dependencies 769
 - Manifest Entries 770
 - overview 769
 - sensible defaults 769
 - tool window 51
- Gradle Build File
 - top level 771
- Gradle Build Files
 - module level 772
- gradle.properties file 770
- GridLayout 166
- LayoutManager 419

H

- Handler class 504
- Higher-order Functions 117
- Hot flows 518
- HP Print Services Plugin 647
- HTML printing 651
- HTML Printing
 - example 655

I

- IBinder 491, 497
- IBinder object 495, 503, 504
- Image Printing 650
- Immutable Variables 90
- implicit
 - intent 80
- implicit intent 439
- Implicit Intent 441
- Implicit Intents
 - example 457
- in 229
- INAPP 720
- In-App Products 715
- In-App Purchasing 721
 - acknowledgePurchase() method 719
 - BillingClient 716
 - BillingResult 731
 - consumeAsync() method 719
 - ConsumeParams 729
 - Consuming purchases 728
 - enablePendingPurchases() method 719
 - getPurchaseState() method 718
 - Google Play Billing Library 715
 - launchBillingFlow() method 718
 - Libraries 721
 - newBuilder() method 716
 - onBillingServiceDisconnected() callback 725
 - onBillingServiceDisconnected() method 717
 - onBillingSetupFinished() listener 725
 - onProductDetailsResponse() callback 726
 - Overview 715
 - ProductDetail 718
 - ProductDetails 726

- products 715
- ProductType 720
- Purchase Flow 727
- PurchaseResponseListener 720
- PurchasesUpdatedListener 718
- PurchaseUpdatedListener 727
- purchase updates 727
- queryProductDetailsAsync() 726
- queryProductDetailsAsync() method 717
- queryPurchasesAsync() 729
- queryPurchasesAsync() method 720
- runOnUiThread() 727
- startConnection() method 717
- subscriptions 715
- tutorial 721
- Initializer Blocks 123
- In-Memory Database 546
- Inner Classes 124
- Instant Dynamic Feature Module 734
- IntelliJ IDEA 83
- Intent 80
 - explicit 80
 - implicit 80
- Intent Availability
 - checking for 446
- intent filters 439
- Intent Filters 442
 - App Link 679
- intent resolution 442
- Intents 439
 - ActivityResultLauncher 441
 - overview 439
 - registerForActivityResult() 441, 454
- Intent Service 491
- IntentService class 491, 494
- Intent URL 460
- intermediate flow operators 514
- is 95
- isInitialized property 95

J

- Java

- convert to Kotlin 83
- Java Native Interface 77
- JetBrains 83
- Jetpack 297
 - overview 297
- JobIntentService 491
 - BIND_JOB_SERVICE permission 493
 - onHandleWork() method 491
- join() 480

K

- KeyAttribute 370
- Keyboard Shortcuts 52
- KeyCycle 375, 391
 - Cycle Editor 395
 - tutorial 391
- Keyframe 383
- Keyframes 370
- KeyFrameSet 400
- KeyPosition 371
 - deltaRelative 371
 - parentRelative 371
 - pathRelative 372
- Keystore File
 - creation 704
- KeyTimeCycle 375, 391
- keytool 443
- KeyTrigger 374
- Killed state 142
- Kotlin
 - accessing class properties 123
 - and Java 83
 - arithmetic operators 97
 - assignment operator 97
 - augmented assignment operators 98
 - bitwise operators 100
 - Boolean 88
 - break 108
 - breaking from loops 107
 - calling class methods 123
 - Char 88
 - class declaration 119

Index

- class initialization 120
- class properties 120
- Companion Objects 125
- conditional control flow 109
- continue labels 108
- continue statement 108
- control flow 105
- convert from Java 83
- Custom Accessors 123
- data types 87
- decrement operator 98
- Default Function Parameters 115
- defining class methods 120
- do ... while loop 107
- Elvis Operator 95
- equality operators 99
- Escape Sequences 89
- expression syntax 97
- Float 88
- Flow 509
- for-in statement 105
- function calling 114
- Functions 113
- Higher-order Functions 117
- if ... else ... expressions 110
- if expressions 109
- Immutable Variables 90
- increment operator 98
- inheritance 129
- Initializer Blocks 123
- Inner Classes 124
- introduction 83
- Lambda Expressions 116
- let Function 93
- Local Functions 114
- logical operators 99
- looping 105
- Mutable Variables 90
- Not-Null Assertion 93
- Nullable Type 92
- Overriding inherited methods 132
- playground 84

- Primary Constructor 120
- properties 123
- range operator 100
- Safe Call Operator 92
- Secondary Constructors 120
- Single Expression Functions 114
- String 88
- subclassing 129
- Type Annotations 91
- Type Casting 95
- Type Checking 95
- Type Inference 91
- variable parameters 115
- when statement 110
- while loop 106

L

- Lambda Expressions 116
- lateinit 94
- Late Initialization 94
- launch 479
- launchBillingFlow() method 718
- layout_collapseMode
 - parallax 436
 - pin 436
- layout_constraintDimensionRatio 218
- layout_constraintHorizontal_bias 216
- layout_constraintVertical_bias 216
- layout editor
 - ConstraintLayout chains 213
- Layout Editor 16, 221
 - Autoconnect Mode 196
 - code mode 176
 - Component Tree 173
 - design mode 173
 - device screen 173
 - example project 221
 - Inference Mode 197
 - palette 173
 - properties panel 174
 - Sample Data 182
 - Setting Properties 177

- toolbar 174
- user interface design 221
- view conversion 181
- Layout Editor Tool
 - changing orientation 16
 - overview 173
- Layout Inspector 52
- Layout Managers 165
- LayoutResultCallback object 671
- Layouts 165
- layout_scrollFlags
 - enterAlwaysCollapsed mode 433
 - enterAlways mode 433
 - exitUntilCollapsed mode 433
 - scroll mode 433
- Layout Validation 184
- let Function 93
- libc 77
- License Testers 723
- Lifecycle
 - awareness 335
 - components 300
 - observers 336
 - owners 335
 - states and events 336
 - tutorial 339
- Lifecycle-Aware Components 335
- Lifecycle library 510
- Lifecycle Methods 147
- Lifecycle Observer 339
 - creating a 339
- Lifecycle Owner
 - creating a 341
- Lifecycles
 - modern 300
- Lifecycle.State.CREATED 530
- Lifecycle.State.DESTROYED 530
- Lifecycle.State.INITIALIZED 530
- Lifecycle.State.RESUMED 530
- Lifecycle.State.STARTED 530
- LinearLayout 166
- LinearLayoutManager 419

- LinearLayoutManager layout 427
- Linux Kernel 76
- list devices 57
- LiveData 298, 309
 - adding to ViewModel 309
 - observer 311
 - tutorial 309
- Live Templates 74
- Local Bound Service 495
 - example 495
- Local Functions 114
- Location Manager 78
- Location permission 590
- Logcat
 - tool window 52
- LogCat
 - enabling 157

M

- MANAGE_EXTERNAL_STORAGE 591
 - adb enabling 591
 - testing 591
- Manifest File
 - permissions 461
- map() operator 514
- Maps 631
- MapView 631
 - adding to a layout 638
- Marker class 631
- match_parent properties 229
- Material design 403
- Material Design 2 759
- Material Design 2 Theming 759
- Material Design 3 759
- Material Theme Builder 761
- Material You 759
- measureTimeMillis() function 513
- MediaController
 - adding to VideoView instance 575
- MediaController class 572
 - methods 572
- MediaPlayer class 597

Index

- methods 597
 - MediaRecorder class 597
 - methods 598
 - recording audio 598
 - Memory Indicator 67
 - Messenger object 504
 - Microphone
 - checking for availability 600
 - Microphone permissions 590
 - mm 229
 - MotionEvent 257, 258, 277
 - getActionMasked() 258
 - MotionLayout 365
 - arc motion 370
 - Attribute Keyframes 370
 - ConstraintSets 366
 - Custom Attribute 386
 - Custom Attributes 367
 - Cycle Editor 395
 - Cycle Keyframes 375
 - Editor 377
 - KeyAttribute 370
 - KeyCycle 391
 - Keyframes 370
 - KeyFrameSet 400
 - KeyPosition 371
 - KeyTimeCycle 391
 - KeyTrigger 374
 - OnClick 369, 382
 - OnSwipe 369
 - overview 365
 - Position Keyframes 371
 - previewing animation 381
 - starting animation 368
 - Trigger Keyframe 374
 - Tutorial 377
 - MotionScene
 - ConstraintSets 366
 - Custom Attributes 367
 - file 366
 - overview 365
 - transition 366
 - moveCamera() method 645
 - multiple devices
 - testing app on 31
 - Multiple Touches
 - handling 258
 - multi-resume 150
 - Multi-Touch
 - example 259
 - Multi-touch Event Handling 257
 - multi-window support 150
 - MutableSharedFlow 520
 - MutableStateFlow 519
 - Mutable Variables 90
 - My Location Layer 632
- ## N
- Navigation 345
 - adding destinations 354
 - overview 345
 - pass data with safeargs 361
 - passing arguments 350
 - safeargs 350
 - stack 345
 - tutorial 351
 - Navigation Action
 - triggering 349
 - Navigation Architecture Component 345
 - Navigation Component
 - tutorial 351
 - Navigation Controller
 - accessing 349
 - Navigation Graph 348, 352
 - adding actions 357
 - creating a 352
 - Navigation Host 346
 - declaring 353
 - newBuilder() method 716
 - normal permissions 589
 - Notification
 - adding actions 618
 - direct reply 621
 - Direct Reply Input 627

- issuing a basic 614
- launch activity from a 616
- PendingIntent 624
- Reply Action 626
- updating direct reply 628
- Notifications 607
 - bundled 618
 - overview 607
- Notifications Manager 78
- Not-Null Assertion 93
- Nullable Type 92
- O**
- Observer
 - implementing a LiveData 311
- onAttach() method 284
- onBillingServiceDisconnected() callback 725
- onBillingServiceDisconnected() method 717
- onBillingSetupFinished() listener 725
- onBind() method 492, 495, 503
- onBindViewHolder() method 427
- OnClick 369
- onClickListener 252, 253, 256
- onClick() method 251
- onCreateContextMenuListener 252
- onCreate() method 140, 147, 492
- onCreateView() method 148
- on-demand modules 733
- onDestroy() method 148, 492
- onDoubleTap() method 263
- onDown() method 263
- onEach() operator 518
- onFling() method 263
- onFocusChangeListener 252
- OnFragmentInteractionListener
 - implementation 359
- onGesturePerformed() method 269
- onHandleWork() method 491, 492
- onKeyListener 252
- onLayoutFailed() method 671
- onLayoutFinished() method 671
- onLongClickListener 252, 255
- onLongPress() method 263
- onMapReady() method 640
- onPageFinished() callback 656
- onPause() method 148
- onProductDetailsResponse() callback 726
- onReceive() method 140, 470, 471, 473
- onRequestPermissionsResult() method 593, 604, 612, 622
- onRestart() method 148
- onRestoreInstanceState() method 148
- onResume() method 140, 148
- onSaveInstanceState() method 148
- onScaleBegin() method 275
- onScaleEnd() method 275
- onScale() method 275
- onScroll() method 263
- OnSeekBarChangeListener 294
- onServiceConnected() method 495, 498, 505
- onServiceDisconnected() method 495, 498, 505
- onShowPress() method 263
- onSingleTapUp() method 263
- onStartCommand() method 492
- onStart() method 148
- onStop() method 148
- onTouchEvent() method 263, 275
- onTouchListener 252, 257
- onTouch() method 257, 258
- onViewCreated() method 148
- onViewStatusRestored() method 148
- OpenJDK 3
- P**
- Package Explorer 15
- Package Manager 78
- PackageManager class 600
- PackageManager.FEATURE_MICROPHONE 600
- PackageManager.PERMISSION_DENIED 591
- PackageManager.PERMISSION_GRANTED 591
- Package Name 14
- Packed chain 189, 216
- PageRange 672, 673
- Paint class 675
- parentRelative 371

Index

- parent view 167
- pathRelative 372
- Paused state 142
- PdfDocument 653
- PdfDocument.Page 665, 672
- PendingIntent class 624
- Permission
 - checking for 591
- permissions
 - dangerous 589
 - normal 589
- Persistent State 147
- Phone permissions 590
- Pinch Gesture
 - detection 275
 - example 275
- Pinch Gesture Recognition 269
- Play Core Library 739, 743
- Polygon class 631
- Polyline class 631
- Position Keyframes 371
- POST_NOTIFICATIONS permission 590, 622
- Primary Constructor 120
- PrintAttributes 670
- PrintDocumentAdapter 653, 665
- PrintDocumentInfo 670
- Printing
 - color 650
 - monochrome 650
- Printing framework
 - architecture 647
- Printing Framework 647
- Print Job
 - starting 676
- Print Manager 647
- PrintManager service 657
- Problems
 - tool window 52
- PROCESS_OUTGOING_CALLS permission 590
- Process States 139
- ProductDetail 718
- ProductDetails 726

- ProductType 720
- Profiler
 - tool window 52
- ProgressBar 165
- proguard-rules.pro file 774
- ProGuard Support 770
- Project
 - tool window 52
- Project Name 14
- Project tool window 15, 52
- pt 229
- PurchaseResponseListener 720
- PurchasesUpdatedListener 718
- PurchaseUpdatedListener 727
- putExtra() method 439, 469
- px 230

Q

- queryProductDetailsAsync() 726
- queryProductDetailsAsync() method 717
- queryPurchaseHistoryAsync() method 720
- queryPurchasesAsync() 729
- queryPurchasesAsync() method 720
- quickboot snapshot 42
- Quick Documentation 72

R

- RadioButton 165
- Range Operator 100
- ratios 217
- READ_CALENDAR permission 590
- READ_CALL_LOG permission 590
- READ_CONTACTS permission 590
- READ_EXTERNAL_STORAGE permission 591
- READ_PHONE_STATE permission 590
- READ_SMS permission 590
- RECEIVE_MMS permission 590
- RECEIVE_SMS permission 590
- RECEIVE_WAP_PUSH permission 590
- Recent Files Navigation 53
- RECORD_AUDIO permission 590
- Recording Audio

- permission 599
 - RecyclerView 419
 - adding to layout file 420
 - example 423
 - LayoutManager 419
 - initializing 427
 - LinearLayoutManager 419
 - StaggeredLayoutManager 419
 - RecyclerView Adapter
 - creation of 425
 - RecyclerView.Adapter 420, 425
 - getItemCount() method 420
 - onBindViewHolder() method 420
 - onCreateViewHolder() method 420
 - RecyclerView.ViewHolder
 - getAdapterPosition() method 430
 - reduce() operator 515, 516
 - registerForActivityResult() 441
 - registerForActivityResult() method 440, 454
 - registerReceiver() method 471
 - RelativeLayout 166
 - release mode 701
 - Release Preparation 701
 - Remote Bound Service 503
 - client communication 503
 - implementation 504
 - manifest file declaration 505
 - RemoteInput.Builder() method 624
 - RemoteInput Object 624
 - Remote Service
 - launching and binding 505
 - sending a message 507
 - repeatOnLifecycle 530
 - Repository
 - tutorial 557
 - Repository Modules 300
 - requestPermissions() method 593
 - Resizable Emulator 44
 - Resource
 - string creation 19
 - Resource File 21
 - Resource Management 139
 - Resource Manager 52, 78
 - result receiver 471
 - Reverse-geocoding 638
 - Reverse Geocoding 637
 - Room
 - Data Access Object (DAO) 540
 - entities 540, 541
 - In-Memory Database 546
 - Repository 540
 - Room Database 540
 - tutorial 557
 - Room Database Persistence 539
 - Room Persistence Library 537, 539
 - root element 165
 - root view 167
 - Run
 - tool window 52
 - runBlocking 479
 - Running Devices
 - tool window 63
 - runOnUiThread() 727
 - Runtime Permission Requests 589
- ## S
- safeargs 350, 361
 - Safe Call Operator 92
 - Sample Data 182
 - Saved State 299, 329
 - library dependencies 331
 - SavedStateHandle 330
 - contains() method 331
 - keys() method 331
 - remove() method 331
 - Saved State module 329
 - SavedStateViewModelFactory 330
 - ScaleGestureDetector class 275
 - Scale-independent 229
 - SDK Packages 6
 - Secondary Constructors 120
 - Secure Sockets Layer (SSL) 77
 - SeekBar 287
 - sendBroadcast() method 469, 471

Index

- sendOrderedBroadcast() method 469, 471
- SEND_SMS permission 590
- sendStickyBroadcast() method 469
- Sensor permissions 590
- Service
 - anatomy 492
 - launch at system start 494
 - manifest file entry 492
 - overview 80
 - run in separate process 493
- ServiceConnection class 505
- Service Process 140
- Service Restart Options 492
- setAudioEncoder() method 598
- setAudioSource() method 598
- setBackgroundColor() 234
- setCompassEnabled() method 643
- setContentView() method 233, 239
- setId() method 234
- setMyLocationButtonEnabled() method 643
- setOnClickListener() method 251, 253
- setOnDoubleTapListener() method 263, 266
- setOutputFile() method 598
- setOutputFormat() method 598
- setResult() method 441
- setText() method 164
- setTransition() 375
- setVideoSource() method 598
- SHA-256 certificate fingerprint 443
- SharedFlow 520, 523
 - backgroundn handling 528
 - DROP_LATEST 520
 - DROP_OLDEST 520
 - in ViewModel 525
 - repeatOnLifecycle 530
 - SUSPEND 521
 - tutorial 523
- shouldOverrideUrlLoading() method 656
- shouldShowRequestPermissionRationale() method 595
- SimpleOnScaleGestureListener 275
- SimpleOnScaleGestureListener class 277
- single() operator 512
- SMS permissions 590
- Snackbar 403, 404, 405
 - overview of 404
- Snapshots
 - emulator 41
- sp 229
- Space class 166
- split APK files 734
- SplitCompatApplication 738
- SplitInstallManager 739
- Spread chain 188
- Spread inside 215
- Spread inside chain 188
- SQL 534
- SQLite 533
 - AVD command-line use 535
 - Columns and Data Types 533
 - overview 534
 - Primary keys 534
- StaggeredGridLayoutManager 419
- startActivity() method 439
- startConnection() method 717
- startForeground() method 140
- START_NOT_STICKY 492
- START_REDELIVER_INTENT 492
- START_STICKY 492
- State
 - restoring 164
- State Change
 - handling 143
- StateFlow 519
- Statement Completion 69
- status bar 431
- Status Bar Widgets 67
 - Memory Indicator 67
- Sticky Broadcast Intents 471
- Stopped state 142
- Storage permissions 591
- String 88
- strings.xml file 23
- Structure
 - tool window 52

- Structured Query Language 534
- Structure tool window 52
- SUBS 720
- subscriptions 715
- supervisorScope 479
- SupportMapFragment class 631
- SUSPEND 521
- Suspend Functions 478
- Switcher 53
- System Broadcasts 475
- system requirements 3

T

- tab bar 431
- TabLayout 409
 - adding to layout 413
 - app
 - tabGravity property 418
 - tabMode property 418
 - example 410
 - fixed mode 417
 - getItemCount() method 409
 - overview 409
 - scrollable mode 417
- TableLayout 166, 549
- TableRow 549
- Telephony Manager 78
- Templates
 - blank vs. empty 169
- Terminal
 - tool window 52
- terminal flow operators 515
- Theme
 - building a custom 761
- Theming 759
 - Material Theme Builder 761
 - tutorial 763
- Time Cycle Keyframes 375
- TODO
 - tool window 52
- toolbar 431
- ToolBarListener 284

- tools
 - layout 281
- Tool window bars 50
- Tool windows 49
- Touch Actions 258
- Touch Event Listener
 - implementation 259
- Touch Events
 - intercepting 257
- Touch handling 257
- transform() operator 515
- try/finally 512
- Type Annotations 91
- Type Casting 95
- Type Checking 95
- Type Inference 91

U

- UiSettings class 631
- unbindService() method 491
- unregisterReceiver() method 471
- URL Mapping 685
- USB connection issues
 - resolving 60
- USE_BIOMETRIC 694
- user interface state 147
- USE_SIP permission 590

V

- Video Playback 571
- VideoView class 571
 - methods 571
 - supported formats 571
- view bindings 135
 - enabling 136
 - using 136
- View class
 - setting properties 240
- view conversion 181
- ViewGroup 165
- View Groups 165
- View Hierarchy 167

Index

ViewHolder class 420
 sample implementation 426
ViewModel
 adding LiveData 309
 data access 307
 fragment association 306
 overview 298
 saved state 329
 Saved State 299, 329
 tutorial 303
ViewModelProvider 306
ViewModel Saved State 329
ViewModelScope 478
ViewPager 409, 414
 adapter 414
 adding to layout 413
 example 410
Views 165
 Java creation 233
View System 78
Virtual Device Configuration dialog 28
Virtual Sensors 41
Visible Process 140

W

WebViewClient 651, 656
WebView view 459
Weighted chain 188, 216
Welcome screen 47
while Loop 106
Widget Dimensions 189
Widget Group Alignment 211
Widgets palette 222
WiFi debugging 61
Wireless debugging 61
Wireless pairing 61
withContext 479, 481
wrap_content properties 231
WRITE_CALENDAR permission 590
WRITE_CALL_LOG permission 590
WRITE_CONTACTS permission 590
WRITE_EXTERNAL_STORAGE permission 591

X

XML Layout File
 manual creation 229
 vs. Java Code 233

Z

zip() operator 517

