

# **Android Studio Giraffe Essentials**

---

Kotlin Edition

Android Studio Giraffe Essentials – Kotlin Edition

ISBN-13: 978-1-951442-76-7

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

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

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

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

Rev: 1.0

## Table of Contents

<b>1. Introduction .....</b>	<b>1</b>
1.1 Downloading the Code Samples .....	1
1.2 Feedback .....	1
1.3 Errata .....	2
<b>2. Setting up an Android Studio Development Environment .....</b>	<b>3</b>
2.1 System requirements .....	3
2.2 Downloading the Android Studio package .....	3
2.3 Installing Android Studio .....	4
2.3.1 Installation on Windows .....	4
2.3.2 Installation on macOS .....	4
2.3.3 Installation on Linux .....	5
2.4 The Android Studio setup wizard .....	5
2.5 Installing additional Android SDK packages .....	6
2.6 Installing the Android SDK Command-line Tools .....	9
2.6.1 Windows 8.1 .....	10
2.6.2 Windows 10 .....	11
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 .....	13
<b>3. Creating an Example Android App in Android Studio .....</b>	<b>15</b>
3.1 About the Project .....	15
3.2 Creating a New Android Project .....	15
3.3 Creating an Activity .....	16
3.4 Defining the Project and SDK Settings .....	16
3.5 Enabling the New Android Studio UI .....	17
3.6 Modifying the Example Application .....	18
3.7 Modifying the User Interface .....	19
3.8 Reviewing the Layout and Resource Files .....	25
3.9 Adding Interaction .....	28
3.10 Summary .....	29
<b>4. Creating an Android Virtual Device (AVD) in Android Studio .....</b>	<b>31</b>
4.1 About Android Virtual Devices .....	31
4.2 Starting the Emulator .....	33
4.3 Running the Application in the AVD .....	34
4.4 Running on Multiple Devices .....	35
4.5 Stopping a Running Application .....	36
4.6 Supporting Dark Theme .....	36
4.7 Running the Emulator in a Separate Window .....	37

## Table of Contents

4.8 Enabling the Device Frame.....	40
4.9 Summary .....	41
<b>5. Using and Configuring the Android Studio AVD Emulator .....</b>	<b>43</b>
5.1 The Emulator Environment .....	43
5.2 Emulator Toolbar Options .....	43
5.3 Working in Zoom Mode .....	45
5.4 Resizing the Emulator Window.....	45
5.5 Extended Control Options.....	45
5.5.1 Location.....	46
5.5.2 Displays.....	46
5.5.3 Cellular .....	46
5.5.4 Battery.....	46
5.5.5 Camera.....	46
5.5.6 Phone .....	46
5.5.7 Directional Pad.....	46
5.5.8 Microphone.....	46
5.5.9 Fingerprint .....	46
5.5.10 Virtual Sensors .....	47
5.5.11 Snapshots.....	47
5.5.12 Record and Playback .....	47
5.5.13 Google Play .....	47
5.5.14 Settings .....	47
5.5.15 Help.....	47
5.6 Working with Snapshots.....	47
5.7 Configuring Fingerprint Emulation .....	48
5.8 The Emulator in Tool Window Mode.....	50
5.9 Creating a Resizable Emulator.....	50
5.10 Summary .....	51
<b>6. A Tour of the Android Studio User Interface .....</b>	<b>53</b>
6.1 The Welcome Screen .....	53
6.2 The Menu Bar .....	54
6.3 The Main Window .....	54
6.4 The Tool Windows .....	56
6.5 The Tool Window Menus.....	59
6.6 Android Studio Keyboard Shortcuts .....	59
6.7 Switcher and Recent Files Navigation .....	60
6.8 Changing the Android Studio Theme .....	61
6.9 Summary .....	62
<b>7. Testing Android Studio Apps on a Physical Android Device.....</b>	<b>63</b>
7.1 An Overview of the Android Debug Bridge (ADB).....	63
7.2 Enabling USB Debugging ADB on Android Devices.....	63
7.2.1 macOS ADB Configuration .....	64
7.2.2 Windows ADB Configuration.....	65
7.2.3 Linux adb Configuration.....	66
7.3 Resolving USB Connection Issues .....	66
7.4 Enabling Wireless Debugging on Android Devices .....	67
7.5 Testing the adb Connection.....	69
7.6 Device Mirroring.....	69

7.7 Summary .....	69
<b>8. The Basics of the Android Studio Code Editor.....</b>	<b>71</b>
8.1 The Android Studio Editor.....	71
8.2 Splitting the Editor Window.....	74
8.3 Code Completion.....	74
8.4 Statement Completion.....	76
8.5 Parameter Information.....	76
8.6 Parameter Name Hints.....	76
8.7 Code Generation.....	76
8.8 Code Folding.....	78
8.9 Quick Documentation Lookup.....	79
8.10 Code Reformatting.....	79
8.11 Finding Sample Code.....	80
8.12 Live Templates.....	80
8.13 Summary.....	81
<b>9. An Overview of the Android Architecture.....</b>	<b>83</b>
9.1 The Android Software Stack.....	83
9.2 The Linux Kernel.....	84
9.3 Android Runtime – ART.....	84
9.4 Android Libraries.....	84
9.4.1 C/C++ Libraries.....	85
9.5 Application Framework.....	85
9.6 Applications.....	86
9.7 Summary.....	86
<b>10. The Anatomy of an Android App.....</b>	<b>87</b>
10.1 Android Activities.....	87
10.2 Android Fragments.....	87
10.3 Android Intents.....	88
10.4 Broadcast Intents.....	88
10.5 Broadcast Receivers.....	88
10.6 Android Services.....	88
10.7 Content Providers.....	89
10.8 The Application Manifest.....	89
10.9 Application Resources.....	89
10.10 Application Context.....	89
10.11 Summary.....	89
<b>11. An Introduction to Kotlin.....</b>	<b>91</b>
11.1 What is Kotlin?.....	91
11.2 Kotlin and Java.....	91
11.3 Converting from Java to Kotlin.....	91
11.4 Kotlin and Android Studio.....	92
11.5 Experimenting with Kotlin.....	92
11.6 Semi-colons in Kotlin.....	93
11.7 Summary.....	93
<b>12. Kotlin Data Types, Variables, and Nullability.....</b>	<b>95</b>
12.1 Kotlin Data Types.....	95

## Table of Contents

12.1.1 Integer Data Types .....	96
12.1.2 Floating-Point Data Types .....	96
12.1.3 Boolean Data Type.....	96
12.1.4 Character Data Type.....	96
12.1.5 String Data Type.....	96
12.1.6 Escape Sequences .....	97
12.2 Mutable Variables.....	98
12.3 Immutable Variables .....	98
12.4 Declaring Mutable and Immutable Variables.....	98
12.5 Data Types are Objects.....	98
12.6 Type Annotations and Type Inference .....	99
12.7 Nullable Type.....	100
12.8 The Safe Call Operator .....	100
12.9 Not-Null Assertion.....	101
12.10 Nullable Types and the let Function.....	101
12.11 Late Initialization (lateinit) .....	102
12.12 The Elvis Operator .....	103
12.13 Type Casting and Type Checking .....	103
12.14 Summary.....	104
<b>13. Kotlin Operators and Expressions .....</b>	<b>105</b>
13.1 Expression Syntax in Kotlin.....	105
13.2 The Basic Assignment Operator.....	105
13.3 Kotlin Arithmetic Operators .....	105
13.4 Augmented Assignment Operators .....	106
13.5 Increment and Decrement Operators .....	106
13.6 Equality Operators.....	107
13.7 Boolean Logical Operators .....	107
13.8 Range Operator .....	108
13.9 Bitwise Operators.....	108
13.9.1 Bitwise Inversion .....	108
13.9.2 Bitwise AND .....	109
13.9.3 Bitwise OR.....	109
13.9.4 Bitwise XOR.....	109
13.9.5 Bitwise Left Shift.....	110
13.9.6 Bitwise Right Shift.....	110
13.10 Summary.....	111
<b>14. Kotlin Control Flow .....</b>	<b>113</b>
14.1 Looping Control flow .....	113
14.1.1 The Kotlin <i>for-in</i> Statement.....	113
14.1.2 The <i>while</i> Loop .....	114
14.1.3 The <i>do ... while</i> loop .....	115
14.1.4 Breaking from Loops.....	115
14.1.5 The <i>continue</i> Statement .....	116
14.1.6 Break and Continue Labels.....	116
14.2 Conditional Control Flow.....	117
14.2.1 Using the <i>if</i> Expressions .....	117
14.2.2 Using <i>if ... else ...</i> Expressions .....	118
14.2.3 Using <i>if ... else if ...</i> Expressions .....	118

14.2.4 Using the <i>when</i> Statement.....	118
14.3 Summary .....	119
<b>15. An Overview of Kotlin Functions and Lambdas .....</b>	<b>121</b>
15.1 What is a Function? .....	121
15.2 How to Declare a Kotlin Function .....	121
15.3 Calling a Kotlin Function.....	122
15.4 Single Expression Functions.....	122
15.5 Local Functions .....	122
15.6 Handling Return Values .....	123
15.7 Declaring Default Function Parameters.....	123
15.8 Variable Number of Function Parameters .....	123
15.9 Lambda Expressions .....	124
15.10 Higher-order Functions .....	125
15.11 Summary .....	126
<b>16. The Basics of Object Oriented Programming in Kotlin .....</b>	<b>127</b>
16.1 What is an Object? .....	127
16.2 What is a Class? .....	127
16.3 Declaring a Kotlin Class.....	127
16.4 Adding Properties to a Class.....	128
16.5 Defining Methods .....	128
16.6 Declaring and Initializing a Class Instance.....	128
16.7 Primary and Secondary Constructors.....	128
16.8 Initializer Blocks.....	131
16.9 Calling Methods and Accessing Properties .....	131
16.10 Custom Accessors .....	131
16.11 Nested and Inner Classes .....	132
16.12 Companion Objects.....	133
16.13 Summary .....	135
<b>17. An Introduction to Kotlin Inheritance and Subclassing .....</b>	<b>137</b>
17.1 Inheritance, Classes and Subclasses.....	137
17.2 Subclassing Syntax .....	137
17.3 A Kotlin Inheritance Example.....	138
17.4 Extending the Functionality of a Subclass .....	139
17.5 Overriding Inherited Methods.....	140
17.6 Adding a Custom Secondary Constructor.....	141
17.7 Using the SavingsAccount Class .....	141
17.8 Summary .....	141
<b>18. An Overview of Android View Binding.....</b>	<b>143</b>
18.1 Find View by Id .....	143
18.2 View Binding .....	143
18.3 Converting the AndroidSample project.....	144
18.4 Enabling View Binding.....	144
18.5 Using View Binding .....	144
18.6 Choosing an Option .....	145
18.7 View Binding in the Book Examples .....	145
18.8 Migrating a Project to View Binding.....	146
18.9 Summary .....	146

<b>19. Understanding Android Application and Activity Lifecycles .....</b>	<b>147</b>
19.1 Android Applications and Resource Management.....	147
19.2 Android Process States .....	147
19.2.1 Foreground Process .....	148
19.2.2 Visible Process .....	148
19.2.3 Service Process .....	148
19.2.4 Background Process.....	148
19.2.5 Empty Process .....	149
19.3 Inter-Process Dependencies .....	149
19.4 The Activity Lifecycle.....	149
19.5 The Activity Stack.....	149
19.6 Activity States .....	150
19.7 Configuration Changes .....	150
19.8 Handling State Change.....	151
19.9 Summary .....	151
<b>20. Handling Android Activity State Changes.....</b>	<b>153</b>
20.1 New vs. Old Lifecycle Techniques.....	153
20.2 The Activity and Fragment Classes.....	153
20.3 Dynamic State vs. Persistent State.....	155
20.4 The Android Lifecycle Methods.....	155
20.5 Lifetimes .....	157
20.6 Foldable Devices and Multi-Resume .....	158
20.7 Disabling Configuration Change Restarts .....	158
20.8 Lifecycle Method Limitations.....	158
20.9 Summary .....	159
<b>21. Android Activity State Changes by Example .....</b>	<b>161</b>
21.1 Creating the State Change Example Project .....	161
21.2 Designing the User Interface .....	162
21.3 Overriding the Activity Lifecycle Methods .....	163
21.4 Filtering the Logcat Panel.....	165
21.5 Running the Application.....	166
21.6 Experimenting with the Activity.....	167
21.7 Summary .....	168
<b>22. Saving and Restoring the State of an Android Activity .....</b>	<b>169</b>
22.1 Saving Dynamic State .....	169
22.2 Default Saving of User Interface State .....	169
22.3 The Bundle Class .....	170
22.4 Saving the State.....	171
22.5 Restoring the State .....	172
22.6 Testing the Application.....	172
22.7 Summary .....	172
<b>23. Understanding Android Views, View Groups and Layouts .....</b>	<b>173</b>
23.1 Designing for Different Android Devices.....	173
23.2 Views and View Groups .....	173
23.3 Android Layout Managers .....	173
23.4 The View Hierarchy .....	175
23.5 Creating User Interfaces.....	176



23.6 Summary .....	176
<b>24. A Guide to the Android Studio Layout Editor Tool .....</b>	<b>177</b>
24.1 Basic vs. Empty Views Activity Templates.....	177
24.2 The Android Studio Layout Editor .....	181
24.3 Design Mode.....	181
24.4 The Palette .....	182
24.5 Design Mode and Layout Views.....	183
24.6 Night Mode .....	184
24.7 Code Mode.....	184
24.8 Split Mode .....	185
24.9 Setting Attributes.....	186
24.10 Transforms .....	187
24.11 Tools Visibility Toggles.....	188
24.12 Converting Views.....	189
24.13 Displaying Sample Data .....	190
24.14 Creating a Custom Device Definition .....	191
24.15 Changing the Current Device.....	191
24.16 Layout Validation .....	192
24.17 Summary .....	193
<b>25. A Guide to the Android ConstraintLayout.....</b>	<b>195</b>
25.1 How ConstraintLayout Works.....	195
25.1.1 Constraints.....	195
25.1.2 Margins.....	196
25.1.3 Opposing Constraints.....	196
25.1.4 Constraint Bias .....	197
25.1.5 Chains.....	198
25.1.6 Chain Styles.....	198
25.2 Baseline Alignment.....	199
25.3 Configuring Widget Dimensions.....	199
25.4 Guideline Helper .....	200
25.5 Group Helper .....	200
25.6 Barrier Helper.....	200
25.7 Flow Helper.....	202
25.8 Ratios .....	203
25.9 ConstraintLayout Advantages .....	203
25.10 ConstraintLayout Availability.....	204
25.11 Summary .....	204
<b>26. A Guide to Using ConstraintLayout in Android Studio .....</b>	<b>205</b>
26.1 Design and Layout Views.....	205
26.2 Autoconnect Mode .....	207
26.3 Inference Mode.....	207
26.4 Manipulating Constraints Manually.....	207
26.5 Adding Constraints in the Inspector .....	209
26.6 Viewing Constraints in the Attributes Window.....	209
26.7 Deleting Constraints.....	210
26.8 Adjusting Constraint Bias .....	211
26.9 Understanding ConstraintLayout Margins.....	211
26.10 The Importance of Opposing Constraints and Bias .....	213

## Table of Contents

26.11 Configuring Widget Dimensions.....	215
26.12 Design Time Tools Positioning.....	216
26.13 Adding Guidelines.....	217
26.14 Adding Barriers.....	219
26.15 Adding a Group.....	220
26.16 Working with the Flow Helper.....	221
26.17 Widget Group Alignment and Distribution.....	221
26.18 Converting other Layouts to ConstraintLayout.....	223
26.19 Summary.....	223
<b>27. Working with ConstraintLayout Chains and Ratios in Android Studio.....</b>	<b>225</b>
27.1 Creating a Chain.....	225
27.2 Changing the Chain Style.....	227
27.3 Spread Inside Chain Style.....	228
27.4 Packed Chain Style.....	228
27.5 Packed Chain Style with Bias.....	228
27.6 Weighted Chain.....	228
27.7 Working with Ratios.....	229
27.8 Summary.....	231
<b>28. An Android Studio Layout Editor ConstraintLayout Tutorial.....</b>	<b>233</b>
28.1 An Android Studio Layout Editor Tool Example.....	233
28.2 Preparing the Layout Editor Environment.....	233
28.3 Adding the Widgets to the User Interface.....	234
28.4 Adding the Constraints.....	237
28.5 Testing the Layout.....	239
28.6 Using the Layout Inspector.....	239
28.7 Summary.....	240
<b>29. Manual XML Layout Design in Android Studio.....</b>	<b>241</b>
29.1 Manually Creating an XML Layout.....	241
29.2 Manual XML vs. Visual Layout Design.....	244
29.3 Summary.....	244
<b>30. Managing Constraints using Constraint Sets.....</b>	<b>245</b>
30.1 Kotlin Code vs. XML Layout Files.....	245
30.2 Creating Views.....	245
30.3 View Attributes.....	246
30.4 Constraint Sets.....	246
30.4.1 Establishing Connections.....	246
30.4.2 Applying Constraints to a Layout.....	246
30.4.3 Parent Constraint Connections.....	246
30.4.4 Sizing Constraints.....	247
30.4.5 Constraint Bias.....	247
30.4.6 Alignment Constraints.....	247
30.4.7 Copying and Applying Constraint Sets.....	247
30.4.8 ConstraintLayout Chains.....	247
30.4.9 Guidelines.....	248
30.4.10 Removing Constraints.....	248
30.4.11 Scaling.....	248
30.4.12 Rotation.....	249

30.5 Summary .....	249
<b>31. An Android ConstraintSet Tutorial.....</b>	<b>251</b>
31.1 Creating the Example Project in Android Studio .....	251
31.2 Adding Views to an Activity .....	251
31.3 Setting View Attributes.....	252
31.4 Creating View IDs.....	253
31.5 Configuring the Constraint Set .....	254
31.6 Adding the EditText View.....	255
31.7 Converting Density Independent Pixels (dp) to Pixels (px).....	256
31.8 Summary .....	257
<b>32. A Guide to Using Apply Changes in Android Studio .....</b>	<b>259</b>
32.1 Introducing Apply Changes.....	259
32.2 Understanding Apply Changes Options .....	259
32.3 Using Apply Changes.....	260
32.4 Configuring Apply Changes Fallback Settings.....	261
32.5 An Apply Changes Tutorial.....	261
32.6 Using Apply Code Changes .....	261
32.7 Using Apply Changes and Restart Activity.....	262
32.8 Using Run App .....	262
32.9 Summary .....	262
<b>33. An Overview and Example of Android Event Handling .....</b>	<b>263</b>
33.1 Understanding Android Events.....	263
33.2 Using the android:onClick Resource .....	263
33.3 Event Listeners and Callback Methods .....	264
33.4 An Event Handling Example .....	264
33.5 Designing the User Interface .....	265
33.6 The Event Listener and Callback Method.....	265
33.7 Consuming Events .....	267
33.8 Summary .....	268
<b>34. Android Touch and Multi-touch Event Handling .....</b>	<b>269</b>
34.1 Intercepting Touch Events .....	269
34.2 The MotionEvent Object .....	270
34.3 Understanding Touch Actions.....	270
34.4 Handling Multiple Touches .....	270
34.5 An Example Multi-Touch Application .....	271
34.6 Designing the Activity User Interface .....	271
34.7 Implementing the Touch Event Listener.....	271
34.8 Running the Example Application.....	274
34.9 Summary .....	274
<b>35. Detecting Common Gestures Using the Android Gesture Detector Class .....</b>	<b>275</b>
35.1 Implementing Common Gesture Detection.....	275
35.2 Creating an Example Gesture Detection Project .....	276
35.3 Implementing the Listener Class.....	276
35.4 Creating the GestureDetectorCompat Instance.....	278
35.5 Implementing the onTouchEvent() Method.....	278
35.6 Testing the Application.....	279

35.7 Summary .....	279
<b>36. Implementing Custom Gesture and Pinch Recognition on Android .....</b>	<b>281</b>
36.1 The Android Gesture Builder Application.....	281
36.2 The GestureOverlayView Class .....	281
36.3 Detecting Gestures.....	281
36.4 Identifying Specific Gestures .....	281
36.5 Installing and Running the Gesture Builder Application .....	281
36.6 Creating a Gestures File .....	282
36.7 Creating the Example Project.....	282
36.8 Extracting the Gestures File from the SD Card .....	282
36.9 Adding the Gestures File to the Project .....	283
36.10 Designing the User Interface .....	283
36.11 Loading the Gestures File .....	284
36.12 Registering the Event Listener.....	285
36.13 Implementing the onGesturePerformed Method.....	285
36.14 Testing the Application.....	286
36.15 Configuring the GestureOverlayView.....	286
36.16 Intercepting Gestures.....	287
36.17 Detecting Pinch Gestures.....	287
36.18 A Pinch Gesture Example Project.....	287
36.19 Summary.....	289
<b>37. An Introduction to Android Fragments.....</b>	<b>291</b>
37.1 What is a Fragment? .....	291
37.2 Creating a Fragment .....	291
37.3 Adding a Fragment to an Activity using the Layout XML File.....	292
37.4 Adding and Managing Fragments in Code .....	294
37.5 Handling Fragment Events .....	295
37.6 Implementing Fragment Communication.....	295
37.7 Summary .....	297
<b>38. Using Fragments in Android Studio - An Example.....</b>	<b>299</b>
38.1 About the Example Fragment Application .....	299
38.2 Creating the Example Project.....	299
38.3 Creating the First Fragment Layout.....	299
38.4 Migrating a Fragment to View Binding .....	301
38.5 Adding the Second Fragment.....	302
38.6 Adding the Fragments to the Activity .....	303
38.7 Making the Toolbar Fragment Talk to the Activity .....	304
38.8 Making the Activity Talk to the Text Fragment .....	307
38.9 Testing the Application.....	308
38.10 Summary.....	308
<b>39. Modern Android App Architecture with Jetpack.....</b>	<b>309</b>
39.1 What is Android Jetpack? .....	309
39.2 The “Old” Architecture.....	309
39.3 Modern Android Architecture .....	309
39.4 The ViewModel Component .....	310
39.5 The LiveData Component.....	310
39.6 ViewModel Saved State.....	311

39.7 LiveData and Data Binding.....	311
39.8 Android Lifecycles .....	312
39.9 Repository Modules.....	312
39.10 Summary.....	313
<b>40. An Android ViewModel Tutorial.....</b>	<b>315</b>
40.1 About the Project .....	315
40.2 Creating the ViewModel Example Project.....	315
40.3 Removing Unwanted Project Elements.....	315
40.4 Designing the Fragment Layout.....	316
40.5 Implementing the View Model.....	317
40.6 Associating the Fragment with the View Model.....	318
40.7 Modifying the Fragment .....	319
40.8 Accessing the ViewModel Data.....	319
40.9 Testing the Project.....	320
40.10 Summary.....	320
<b>41. An Android Jetpack LiveData Tutorial.....</b>	<b>321</b>
41.1 LiveData - A Recap .....	321
41.2 Adding LiveData to the ViewModel.....	321
41.3 Implementing the Observer.....	323
41.4 Summary .....	324
<b>42. An Overview of Android Jetpack Data Binding.....</b>	<b>325</b>
42.1 An Overview of Data Binding.....	325
42.2 The Key Components of Data Binding .....	325
42.2.1 The Project Build Configuration.....	325
42.2.2 The Data Binding Layout File.....	326
42.2.3 The Layout File Data Element .....	327
42.2.4 The Binding Classes.....	328
42.2.5 Data Binding Variable Configuration.....	328
42.2.6 Binding Expressions (One-Way).....	329
42.2.7 Binding Expressions (Two-Way).....	330
42.2.8 Event and Listener Bindings.....	330
42.3 Summary .....	331
<b>43. An Android Jetpack Data Binding Tutorial.....</b>	<b>333</b>
43.1 Removing the Redundant Code.....	333
43.2 Enabling Data Binding .....	334
43.3 Adding the Layout Element.....	335
43.4 Adding the Data Element to Layout File.....	336
43.5 Working with the Binding Class .....	336
43.6 Assigning the ViewModel Instance to the Data Binding Variable .....	337
43.7 Adding Binding Expressions .....	338
43.8 Adding the Conversion Method .....	339
43.9 Adding a Listener Binding.....	339
43.10 Testing the App.....	340
43.11 Summary.....	340
<b>44. An Android ViewModel Saved State Tutorial.....</b>	<b>341</b>
44.1 Understanding ViewModel State Saving.....	341

## Table of Contents

44.2 Implementing ViewModel State Saving .....	341
44.3 Saving and Restoring State .....	342
44.4 Adding Saved State Support to the ViewModelDemo Project .....	343
44.5 Summary .....	344
<b>45. Working with Android Lifecycle-Aware Components .....</b>	<b>345</b>
45.1 Lifecycle Awareness .....	345
45.2 Lifecycle Owners .....	345
45.3 Lifecycle Observers .....	346
45.4 Lifecycle States and Events .....	346
45.5 Summary .....	347
<b>46. An Android Jetpack Lifecycle Awareness Tutorial .....</b>	<b>349</b>
46.1 Creating the Example Lifecycle Project .....	349
46.2 Creating a Lifecycle Observer .....	349
46.3 Adding the Observer .....	350
46.4 Testing the Observer .....	351
46.5 Creating a Lifecycle Owner .....	351
46.6 Testing the Custom Lifecycle Owner .....	353
46.7 Summary .....	353
<b>47. An Overview of the Navigation Architecture Component .....</b>	<b>355</b>
47.1 Understanding Navigation .....	355
47.2 Declaring a Navigation Host .....	356
47.3 The Navigation Graph .....	358
47.4 Accessing the Navigation Controller .....	359
47.5 Triggering a Navigation Action .....	359
47.6 Passing Arguments .....	360
47.7 Summary .....	360
<b>48. An Android Jetpack Navigation Component Tutorial .....</b>	<b>361</b>
48.1 Creating the NavigationDemo Project .....	361
48.2 Adding Navigation to the Build Configuration .....	361
48.3 Creating the Navigation Graph Resource File .....	362
48.4 Declaring a Navigation Host .....	363
48.5 Adding Navigation Destinations .....	364
48.6 Designing the Destination Fragment Layouts .....	366
48.7 Adding an Action to the Navigation Graph .....	367
48.8 Implement the OnFragmentInteractionListener .....	369
48.9 Adding View Binding Support to the Destination Fragments .....	370
48.10 Triggering the Action .....	370
48.11 Passing Data Using Safeargs .....	371
48.12 Summary .....	374
<b>49. An Introduction to MotionLayout .....</b>	<b>375</b>
49.1 An Overview of MotionLayout .....	375
49.2 MotionLayout .....	375
49.3 MotionScene .....	375
49.4 Configuring ConstraintSets .....	376
49.5 Custom Attributes .....	377
49.6 Triggering an Animation .....	379

49.7 Arc Motion.....	380
49.8 Keyframes.....	380
49.8.1 Attribute Keyframes.....	380
49.8.2 Position Keyframes.....	381
49.9 Time Linearity.....	384
49.10 KeyTrigger.....	384
49.11 Cycle and Time Cycle Keyframes.....	385
49.12 Starting an Animation from Code.....	385
49.13 Summary.....	386
<b>50. An Android MotionLayout Editor.....</b>	<b>387</b>
50.1 Creating the MotionLayoutDemo Project.....	387
50.2 ConstraintLayout to MotionLayout Conversion.....	387
50.3 Configuring Start and End Constraints.....	389
50.4 Previewing the MotionLayout Animation.....	392
50.5 Adding an OnClick Gesture.....	392
50.6 Adding an Attribute Keyframe to the Transition.....	394
50.7 Adding a CustomAttribute to a Transition.....	396
50.8 Adding Position Keyframes.....	398
50.9 Summary.....	400
<b>51. A MotionLayout KeyCycle Tutorial.....</b>	<b>401</b>
51.1 An Overview of Cycle Keyframes.....	401
51.2 Using the Cycle Editor.....	405
51.3 Creating the KeyCycleDemo Project.....	406
51.4 Configuring the Start and End Constraints.....	406
51.5 Creating the Cycles.....	408
51.6 Previewing the Animation.....	410
51.7 Adding the KeyFrameSet to the MotionScene.....	410
51.8 Summary.....	412
<b>52. Working with the Floating Action Button and Snackbar.....</b>	<b>413</b>
52.1 The Material Design.....	413
52.2 The Design Library.....	413
52.3 The Floating Action Button (FAB).....	413
52.4 The Snackbar.....	414
52.5 Creating the Example Project.....	415
52.6 Reviewing the Project.....	415
52.7 Removing Navigation Features.....	416
52.8 Changing the Floating Action Button.....	416
52.9 Adding an Action to the Snackbar.....	418
52.10 Summary.....	418
<b>53. Creating a Tabbed Interface using the TabLayout Component.....</b>	<b>419</b>
53.1 An Introduction to the ViewPager2.....	419
53.2 An Overview of the TabLayout Component.....	419
53.3 Creating the TabLayoutDemo Project.....	420
53.4 Creating the First Fragment.....	420
53.5 Duplicating the Fragments.....	422
53.6 Adding the TabLayout and ViewPager2.....	423
53.7 Performing the Initialization Tasks.....	425

## Table of Contents

53.8 Testing the Application.....	427
53.9 Customizing the TabLayout.....	427
53.10 Summary.....	429
<b>54. Working with the RecyclerView and CardView Widgets.....</b>	<b>431</b>
54.1 An Overview of the RecyclerView.....	431
54.2 An Overview of the CardView.....	433
54.3 Summary.....	434
<b>55. An Android RecyclerView and CardView Tutorial.....</b>	<b>435</b>
55.1 Creating the CardDemo Project.....	435
55.2 Modifying the Basic Views Activity Project.....	435
55.3 Designing the CardView Layout.....	436
55.4 Adding the RecyclerView.....	437
55.5 Adding the Image Files.....	437
55.6 Creating the RecyclerView Adapter.....	437
55.7 Initializing the RecyclerView Component.....	439
55.8 Testing the Application.....	440
55.9 Responding to Card Selections.....	441
55.10 Summary.....	443
<b>56. Working with the AppBar and Collapsing Toolbar Layouts.....</b>	<b>445</b>
56.1 The Anatomy of an AppBar.....	445
56.2 The Example Project.....	446
56.3 Coordinating the RecyclerView and Toolbar.....	446
56.4 Introducing the Collapsing Toolbar Layout.....	448
56.5 Changing the Title and Scrim Color.....	451
56.6 Summary.....	452
<b>57. An Overview of Android Intents.....</b>	<b>453</b>
57.1 An Overview of Intents.....	453
57.2 Explicit Intents.....	453
57.3 Returning Data from an Activity.....	454
57.4 Implicit Intents.....	455
57.5 Using Intent Filters.....	456
57.6 Automatic Link Verification.....	456
57.7 Manually Enabling Links.....	459
57.8 Checking Intent Availability.....	460
57.9 Summary.....	461
<b>58. Android Explicit Intents – A Worked Example.....</b>	<b>463</b>
58.1 Creating the Explicit Intent Example Application.....	463
58.2 Designing the User Interface Layout for MainActivity.....	463
58.3 Creating the Second Activity Class.....	464
58.4 Designing the User Interface Layout for SecondActivity.....	465
58.5 Reviewing the Application Manifest File.....	465
58.6 Creating the Intent.....	466
58.7 Extracting Intent Data.....	467
58.8 Launching SecondActivity as a Sub-Activity.....	468
58.9 Returning Data from a Sub-Activity.....	469
58.10 Testing the Application.....	469



58.11 Summary .....	469
<b>59. Android Implicit Intents – A Worked Example .....</b>	<b>471</b>
59.1 Creating the Android Studio Implicit Intent Example Project .....	471
59.2 Designing the User Interface .....	471
59.3 Creating the Implicit Intent .....	472
59.4 Adding a Second Matching Activity .....	473
59.5 Adding the Web View to the UI .....	473
59.6 Obtaining the Intent URL .....	474
59.7 Modifying the MyWebView Project Manifest File .....	475
59.8 Installing the MyWebView Package on a Device .....	476
59.9 Testing the Application .....	477
59.10 Manually Enabling the Link .....	477
59.11 Automatic Link Verification .....	479
59.12 Summary .....	481
<b>60. Android Broadcast Intents and Broadcast Receivers .....</b>	<b>483</b>
60.1 An Overview of Broadcast Intents .....	483
60.2 An Overview of Broadcast Receivers .....	484
60.3 Obtaining Results from a Broadcast .....	485
60.4 Sticky Broadcast Intents .....	485
60.5 The Broadcast Intent Example .....	485
60.6 Creating the Example Application .....	486
60.7 Creating and Sending the Broadcast Intent .....	486
60.8 Creating the Broadcast Receiver .....	487
60.9 Registering the Broadcast Receiver .....	488
60.10 Testing the Broadcast Example .....	489
60.11 Listening for System Broadcasts .....	489
60.12 Summary .....	489
<b>61. An Introduction to Kotlin Coroutines .....</b>	<b>491</b>
61.1 What are Coroutines? .....	491
61.2 Threads vs. Coroutines .....	491
61.3 Coroutine Scope .....	492
61.4 Suspend Functions .....	492
61.5 Coroutine Dispatchers .....	492
61.6 Coroutine Builders .....	493
61.7 Jobs .....	493
61.8 Coroutines – Suspending and Resuming .....	494
61.9 Returning Results from a Coroutine .....	495
61.10 Using withContext .....	495
61.11 Coroutine Channel Communication .....	497
61.12 Summary .....	498
<b>62. An Android Kotlin Coroutines Tutorial .....</b>	<b>499</b>
62.1 Creating the Coroutine Example Application .....	499
62.2 Adding Coroutine Support to the Project .....	499
62.3 Designing the User Interface .....	499
62.4 Implementing the SeekBar .....	501
62.5 Adding the Suspend Function .....	501
62.6 Implementing the launchCoroutines Method .....	502

## Table of Contents

62.7 Testing the App.....	503
62.8 Summary .....	503
<b>63. An Overview of Android Services.....</b>	<b>505</b>
63.1 Intent Service .....	505
63.2 Bound Service.....	505
63.3 The Anatomy of a Service .....	506
63.4 Controlling Destroyed Service Restart Options.....	506
63.5 Declaring a Service in the Manifest File.....	506
63.6 Starting a Service Running on System Startup.....	507
63.7 Summary .....	508
<b>64. Android Local Bound Services – A Worked Example.....</b>	<b>509</b>
64.1 Understanding Bound Services .....	509
64.2 Bound Service Interaction Options .....	509
64.3 A Local Bound Service Example.....	509
64.4 Adding a Bound Service to the Project .....	510
64.5 Implementing the Binder .....	510
64.6 Binding the Client to the Service .....	512
64.7 Completing the Example.....	513
64.8 Testing the Application.....	514
64.9 Summary .....	515
<b>65. Android Remote Bound Services – A Worked Example .....</b>	<b>517</b>
65.1 Client to Remote Service Communication.....	517
65.2 Creating the Example Application.....	517
65.3 Designing the User Interface .....	517
65.4 Implementing the Remote Bound Service.....	517
65.5 Configuring a Remote Service in the Manifest File.....	519
65.6 Launching and Binding to the Remote Service.....	519
65.7 Sending a Message to the Remote Service .....	521
65.8 Summary .....	521
<b>66. An Introduction to Kotlin Flow .....</b>	<b>523</b>
66.1 Understanding Flows.....	523
66.2 Creating the Sample Project .....	523
66.3 Adding the Kotlin Lifecycle Library .....	524
66.4 Declaring a Flow.....	524
66.5 Emitting Flow Data .....	525
66.6 Collecting Flow Data .....	525
66.7 Adding a Flow Buffer .....	526
66.8 Transforming Data with Intermediaries .....	528
66.9 Terminal Flow Operators .....	529
66.10 Flow Flattening.....	530
66.11 Combining Multiple Flows .....	531
66.12 Hot and Cold Flows .....	532
66.13 StateFlow .....	533
66.14 SharedFlow.....	534
66.15 Summary .....	535
<b>67. An Android SharedFlow Tutorial .....</b>	<b>537</b>

67.1 About the Project .....	537
67.2 Creating the SharedFlowDemo Project.....	537
67.3 Designing the User Interface Layout .....	537
67.4 Adding the List Row Layout .....	537
67.5 Adding the RecyclerView Adapter.....	538
67.6 Adding the ViewModel .....	539
67.7 Configuring the ViewModelProvider.....	540
67.8 Collecting the Flow Values.....	541
67.9 Testing the SharedFlowDemo App .....	542
67.10 Handling Flows in the Background.....	542
67.11 Summary.....	545
<b>68. An Overview of Android SQLite Databases .....</b>	<b>547</b>
68.1 Understanding Database Tables.....	547
68.2 Introducing Database Schema .....	547
68.3 Columns and Data Types .....	547
68.4 Database Rows .....	548
68.5 Introducing Primary Keys .....	548
68.6 What is SQLite? .....	548
68.7 Structured Query Language (SQL).....	548
68.8 Trying SQLite on an Android Virtual Device (AVD) .....	549
68.9 The Android Room Persistence Library.....	550
68.10 Summary.....	551
<b>69. The Android Room Persistence Library .....</b>	<b>553</b>
69.1 Revisiting Modern App Architecture .....	553
69.2 Key Elements of Room Database Persistence.....	553
69.2.1 Repository .....	554
69.2.2 Room Database .....	554
69.2.3 Data Access Object (DAO) .....	554
69.2.4 Entities.....	554
69.2.5 SQLite Database .....	554
69.3 Understanding Entities.....	555
69.4 Data Access Objects.....	557
69.5 The Room Database.....	558
69.6 The Repository.....	559
69.7 In-Memory Databases.....	560
69.8 Database Inspector.....	560
69.9 Summary.....	561
<b>70. An Android TableLayout and TableRow Tutorial .....</b>	<b>563</b>
70.1 The TableLayout and TableRow Layout Views.....	563
70.2 Creating the Room Database Project .....	564
70.3 Converting to a LinearLayout.....	564
70.4 Adding the TableLayout to the User Interface.....	565
70.5 Configuring the TableRows .....	566
70.6 Adding the Button Bar to the Layout .....	567
70.7 Adding the RecyclerView.....	568
70.8 Adjusting the Layout Margins.....	569
70.9 Summary.....	569

<b>71. An Android Room Database and Repository Tutorial.....</b>	<b>571</b>
71.1 About the RoomDemo Project.....	571
71.2 Modifying the Build Configuration.....	571
71.3 Building the Entity.....	572
71.4 Creating the Data Access Object.....	573
71.5 Adding the Room Database.....	575
71.6 Adding the Repository.....	575
71.7 Adding the ViewModel.....	578
71.8 Creating the Product Item Layout.....	579
71.9 Adding the RecyclerView Adapter.....	580
71.10 Preparing the Main Activity.....	581
71.11 Adding the Button Listeners.....	582
71.12 Adding LiveData Observers.....	583
71.13 Initializing the RecyclerView.....	583
71.14 Testing the RoomDemo App.....	584
71.15 Using the Database Inspector.....	584
71.16 Summary.....	585
<b>72. Video Playback on Android using the VideoView and MediaController Classes.....</b>	<b>587</b>
72.1 Introducing the Android VideoView Class.....	587
72.2 Introducing the Android MediaController Class.....	588
72.3 Creating the Video Playback Example.....	588
72.4 Designing the VideoPlayer Layout.....	588
72.5 Downloading the Video File.....	589
72.6 Configuring the VideoView.....	589
72.7 Adding the MediaController to the Video View.....	591
72.8 Setting up the onPreparedListener.....	591
72.9 Summary.....	592
<b>73. Android Picture-in-Picture Mode.....</b>	<b>593</b>
73.1 Picture-in-Picture Features.....	593
73.2 Enabling Picture-in-Picture Mode.....	594
73.3 Configuring Picture-in-Picture Parameters.....	594
73.4 Entering Picture-in-Picture Mode.....	595
73.5 Detecting Picture-in-Picture Mode Changes.....	595
73.6 Adding Picture-in-Picture Actions.....	595
73.7 Summary.....	596
<b>74. An Android Picture-in-Picture Tutorial.....</b>	<b>597</b>
74.1 Adding Picture-in-Picture Support to the Manifest.....	597
74.2 Adding a Picture-in-Picture Button.....	597
74.3 Entering Picture-in-Picture Mode.....	598
74.4 Detecting Picture-in-Picture Mode Changes.....	599
74.5 Adding a Broadcast Receiver.....	599
74.6 Adding the PiP Action.....	600
74.7 Testing the Picture-in-Picture Action.....	603
74.8 Summary.....	603
<b>75. Making Runtime Permission Requests in Android.....</b>	<b>605</b>
75.1 Understanding Normal and Dangerous Permissions.....	605
75.2 Creating the Permissions Example Project.....	607

75.3 Checking for a Permission .....	607
75.4 Requesting Permission at Runtime.....	609
75.5 Providing a Rationale for the Permission Request .....	610
75.6 Testing the Permissions App.....	611
75.7 Summary .....	612
<b>76. Android Audio Recording and Playback using MediaPlayer and MediaRecorder .....</b>	<b>613</b>
76.1 Playing Audio .....	613
76.2 Recording Audio and Video using the MediaRecorder Class.....	614
76.3 About the Example Project .....	615
76.4 Creating the AudioApp Project.....	615
76.5 Designing the User Interface .....	615
76.6 Checking for Microphone Availability.....	616
76.7 Initializing the Activity.....	617
76.8 Implementing the recordAudio() Method.....	618
76.9 Implementing the stopAudio() Method.....	618
76.10 Implementing the playAudio() method.....	619
76.11 Configuring and Requesting Permissions .....	619
76.12 Testing the Application.....	621
76.13 Summary .....	621
<b>77. An Android Notifications Tutorial .....</b>	<b>623</b>
77.1 An Overview of Notifications.....	623
77.2 Creating the NotifyDemo Project.....	625
77.3 Designing the User Interface .....	625
77.4 Creating the Second Activity .....	625
77.5 Creating a Notification Channel .....	626
77.6 Requesting Notification Permission .....	627
77.7 Creating and Issuing a Notification .....	630
77.8 Launching an Activity from a Notification.....	632
77.9 Adding Actions to a Notification .....	634
77.10 Bundled Notifications.....	634
77.11 Summary .....	636
<b>78. An Android Direct Reply Notification Tutorial .....</b>	<b>637</b>
78.1 Creating the DirectReply Project .....	637
78.2 Designing the User Interface .....	637
78.3 Requesting Notification Permission .....	638
78.4 Creating the Notification Channel.....	639
78.5 Building the RemoteInput Object.....	640
78.6 Creating the PendingIntent.....	641
78.7 Creating the Reply Action.....	642
78.8 Receiving Direct Reply Input.....	643
78.9 Updating the Notification .....	644
78.10 Summary .....	645
<b>79. Working with the Google Maps Android API in Android Studio .....</b>	<b>647</b>
79.1 The Elements of the Google Maps Android API .....	647
79.2 Creating the Google Maps Project.....	648
79.3 Creating a Google Cloud Billing Account .....	648
79.4 Creating a New Google Cloud Project.....	649

## Table of Contents

79.5 Enabling the Google Maps SDK.....	650
79.6 Generating a Google Maps API Key.....	651
79.7 Adding the API Key to the Android Studio Project.....	652
79.8 Testing the Application.....	652
79.9 Understanding Geocoding and Reverse Geocoding.....	652
79.10 Adding a Map to an Application.....	654
79.11 Requesting Current Location Permission.....	654
79.12 Displaying the User's Current Location.....	656
79.13 Changing the Map Type.....	657
79.14 Displaying Map Controls to the User.....	658
79.15 Handling Map Gesture Interaction.....	658
79.15.1 Map Zooming Gestures.....	658
79.15.2 Map Scrolling/Panning Gestures.....	659
79.15.3 Map Tilt Gestures.....	659
79.15.4 Map Rotation Gestures.....	659
79.16 Creating Map Markers.....	659
79.17 Controlling the Map Camera.....	660
79.18 Summary.....	661
<b>80. Printing with the Android Printing Framework.....</b>	<b>663</b>
80.1 The Android Printing Architecture.....	663
80.2 The Print Service Plugins.....	663
80.3 Google Cloud Print.....	664
80.4 Printing to Google Drive.....	664
80.5 Save as PDF.....	665
80.6 Printing from Android Devices.....	665
80.7 Options for Building Print Support into Android Apps.....	666
80.7.1 Image Printing.....	666
80.7.2 Creating and Printing HTML Content.....	667
80.7.3 Printing a Web Page.....	668
80.7.4 Printing a Custom Document.....	669
80.8 Summary.....	669
<b>81. An Android HTML and Web Content Printing Example.....</b>	<b>671</b>
81.1 Creating the HTML Printing Example Application.....	671
81.2 Printing Dynamic HTML Content.....	671
81.3 Creating the Web Page Printing Example.....	674
81.4 Removing the Floating Action Button.....	674
81.5 Removing Navigation Features.....	674
81.6 Designing the User Interface Layout.....	675
81.7 Accessing the WebView from the Main Activity.....	676
81.8 Loading the Web Page into the WebView.....	676
81.9 Adding the Print Menu Option.....	677
81.10 Summary.....	679
<b>82. A Guide to Android Custom Document Printing.....</b>	<b>681</b>
82.1 An Overview of Android Custom Document Printing.....	681
82.1.1 Custom Print Adapters.....	681
82.2 Preparing the Custom Document Printing Project.....	682
82.3 Creating the Custom Print Adapter.....	683
82.4 Implementing the onLayout() Callback Method.....	684

82.5 Implementing the onWrite() Callback Method .....	687
82.6 Checking a Page is in Range .....	689
82.7 Drawing the Content on the Page Canvas .....	690
82.8 Starting the Print Job .....	692
82.9 Testing the Application.....	693
82.10 Summary .....	693
<b>83. An Introduction to Android App Links.....</b>	<b>695</b>
83.1 An Overview of Android App Links .....	695
83.2 App Link Intent Filters .....	695
83.3 Handling App Link Intents .....	696
83.4 Associating the App with a Website.....	696
83.5 Summary .....	697
<b>84. An Android Studio App Links Tutorial .....</b>	<b>699</b>
84.1 About the Example App .....	699
84.2 The Database Schema .....	699
84.3 Loading and Running the Project.....	699
84.4 Adding the URL Mapping.....	701
84.5 Adding the Intent Filter.....	704
84.6 Adding Intent Handling Code.....	704
84.7 Testing the App.....	707
84.8 Creating the Digital Asset Links File .....	707
84.9 Testing the App Link.....	708
84.10 Summary .....	708
<b>85. An Android Biometric Authentication Tutorial.....</b>	<b>709</b>
85.1 An Overview of Biometric Authentication.....	709
85.2 Creating the Biometric Authentication Project .....	709
85.3 Configuring Device Fingerprint Authentication .....	710
85.4 Adding the Biometric Permission to the Manifest File.....	710
85.5 Designing the User Interface .....	711
85.6 Adding a Toast Convenience Method .....	711
85.7 Checking the Security Settings.....	712
85.8 Configuring the Authentication Callbacks.....	713
85.9 Adding the CancellationSignal.....	714
85.10 Starting the Biometric Prompt .....	714
85.11 Testing the Project.....	715
85.12 Summary .....	716
<b>86. Creating, Testing, and Uploading an Android App Bundle.....</b>	<b>717</b>
86.1 The Release Preparation Process .....	717
86.2 Android App Bundles.....	717
86.3 Register for a Google Play Developer Console Account.....	718
86.4 Configuring the App in the Console .....	719
86.5 Enabling Google Play App Signing.....	720
86.6 Creating a Keystore File .....	720
86.7 Creating the Android App Bundle.....	721
86.8 Generating Test APK Files .....	723
86.9 Uploading the App Bundle to the Google Play Developer Console.....	724
86.10 Exploring the App Bundle .....	725

## Table of Contents

86.11 Managing Testers .....	726
86.12 Rolling the App Out for Testing.....	726
86.13 Uploading New App Bundle Revisions.....	727
86.14 Analyzing the App Bundle File .....	728
86.15 Summary .....	729
<b>87. An Overview of Android In-App Billing .....</b>	<b>731</b>
87.1 Preparing a Project for In-App Purchasing.....	731
87.2 Creating In-App Products and Subscriptions .....	731
87.3 Billing Client Initialization.....	732
87.4 Connecting to the Google Play Billing Library.....	733
87.5 Querying Available Products.....	733
87.6 Starting the Purchase Process.....	734
87.7 Completing the Purchase .....	734
87.8 Querying Previous Purchases.....	735
87.9 Summary .....	736
<b>88. An Android In-App Purchasing Tutorial .....</b>	<b>737</b>
88.1 About the In-App Purchasing Example Project.....	737
88.2 Creating the InAppPurchase Project .....	737
88.3 Adding Libraries to the Project .....	737
88.4 Designing the User Interface .....	738
88.5 Adding the App to the Google Play Store .....	738
88.6 Creating an In-App Product.....	739
88.7 Enabling License Testers .....	739
88.8 Initializing the Billing Client .....	740
88.9 Querying the Product.....	742
88.10 Launching the Purchase Flow .....	743
88.11 Handling Purchase Updates .....	743
88.12 Consuming the Product.....	744
88.13 Restoring a Previous Purchase .....	745
88.14 Testing the App.....	746
88.15 Troubleshooting .....	747
88.16 Summary .....	748
<b>89. Working with Material Design 3 Theming .....</b>	<b>749</b>
89.1 Material Design 2 vs. Material Design 3 .....	749
89.2 Understanding Material Design Theming .....	749
89.3 Material Design 3 Theming .....	749
89.4 Building a Custom Theme.....	751
89.5 Summary .....	752
<b>90. A Material Design 3 Theming and Dynamic Color Tutorial.....</b>	<b>753</b>
90.1 Creating the ThemeDemo Project .....	753
90.2 Designing the User Interface .....	753
90.3 Building a New Theme .....	755
90.4 Adding the Theme to the Project.....	756
90.5 Enabling Dynamic Color Support .....	757
90.6 Previewing Dynamic Colors.....	758
90.7 Summary .....	759
<b>91. An Overview of Gradle in Android Studio.....</b>	<b>761</b>



91.1 An Overview of Gradle .....	761
91.2 Gradle and Android Studio .....	761
91.2.1 Sensible Defaults .....	761
91.2.2 Dependencies.....	761
91.2.3 Build Variants .....	762
91.2.4 Manifest Entries .....	762
91.2.5 APK Signing.....	762
91.2.6 ProGuard Support.....	762
91.3 The Property and Settings Gradle Build File .....	762
91.4 The Top-level Gradle Build File.....	763
91.5 Module Level Gradle Build Files.....	764
91.6 Configuring Signing Settings in the Build File.....	766
91.7 Running Gradle Tasks from the Command Line .....	767
91.8 Summary .....	768
<b>Index.....</b>	<b>769</b>



## 1. Introduction

Fully updated for Android Studio Giraffe and the new UI, this book teaches 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.

Chapters also cover 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, 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/giraffekotlin/index.php>

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

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

### 1.2 Feedback

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

## 1.3 Errata

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

*<https://www.ebookfrenzy.com/errata/giraffekotlin.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 developing 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), including the Android Software Development Kit (SDK), the Kotlin plug-in and the 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 Giraffe 2022.3.1 using the Android API 33 SDK (Tiramisu), which, at the time of writing, are the latest stable releases.

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, there may be differences between this book and the software. A web search for “Android Studio Giraffe” 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 Giraffe 2022.3.1 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 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 system users. When prompted to select the components to install, ensure 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 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 as 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:

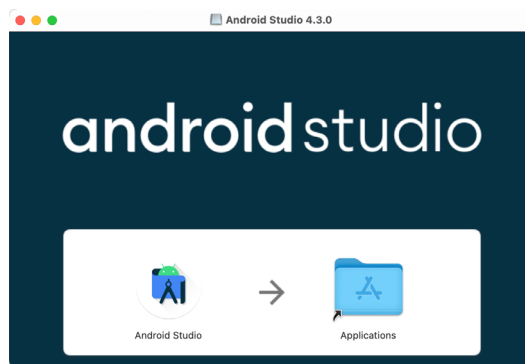


Figure 2-1

To install the package, 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:

```
tar xvfz /<path to package>/android-studio-<version>-linux.tar.gz
```

Note that the Android Studio bundle will be installed into a subdirectory named *android-studio*. Therefore, assuming 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
```

## 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 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:

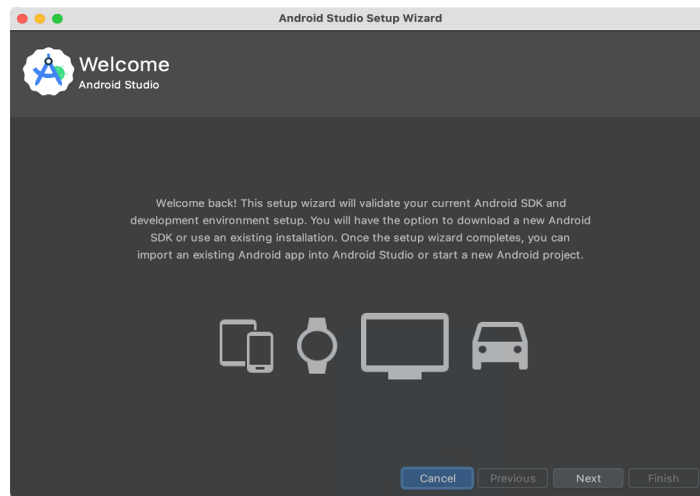


Figure 2-2

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.

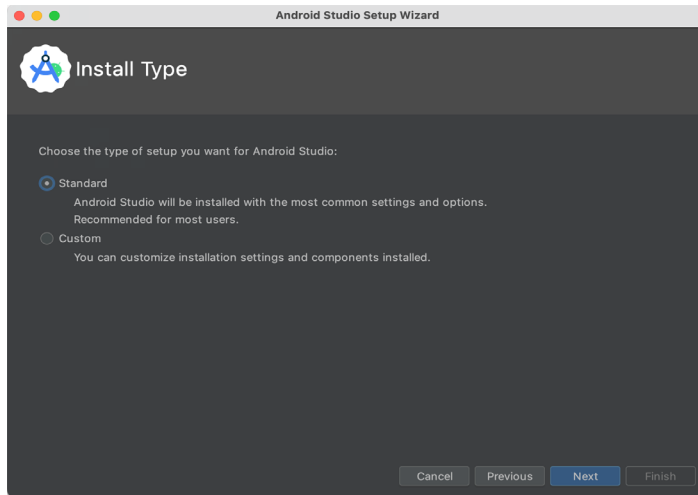


Figure 2-3

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 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:

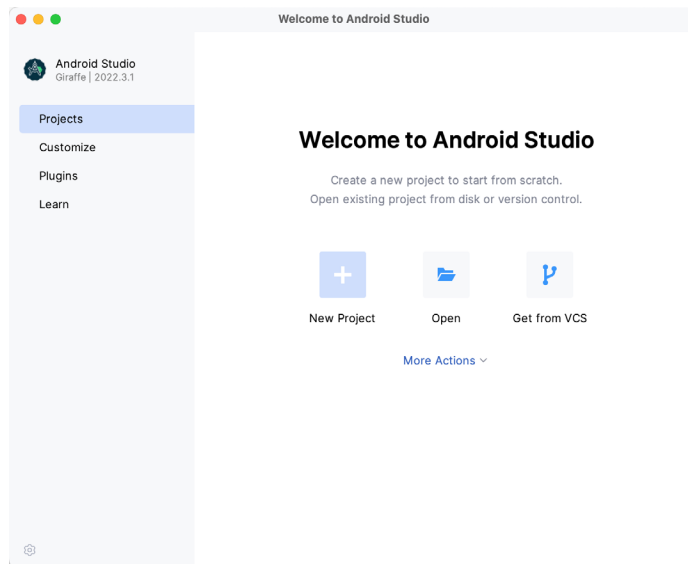


Figure 2-4

## 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 Settings dialog will appear as shown in Figure 2-5:

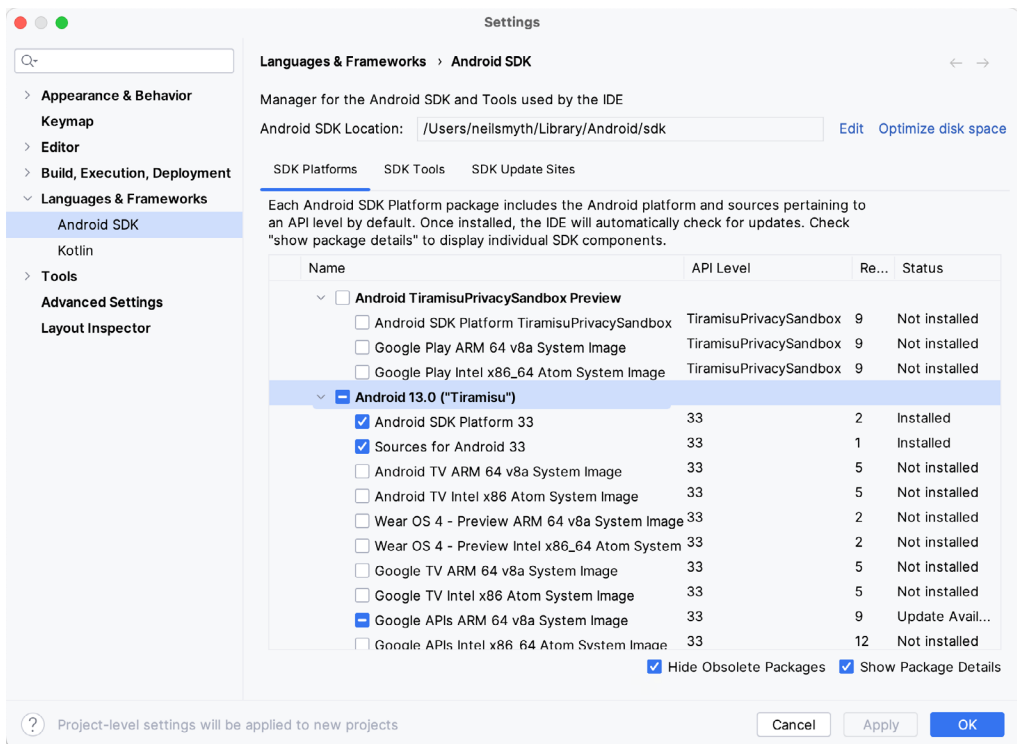


Figure 2-5

Google pairs each release of Android Studio with a maximum supported Application Programming Interface (API) level of the Android SDK. In the case of Android Studio Giraffe, this is Android Tiramisu (API Level 33). This information can be confirmed using the following link:

<https://developer.android.com/studio/releases#api-level-support>

Immediately after installing Android Studio for the first time, it is likely that only the latest supported version of the Android SDK has been installed. To install older versions of the Android SDK, select the checkboxes corresponding to the versions and click 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 ensures 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 the *Apply* button. Click the *OK* button to install the SDK in the resulting confirmation dialog. 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:

## Setting up an Android Studio Development Environment

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 the *Apply* button.

In addition to the Android SDK packages, several tools are also installed for building Android applications. To view the currently installed packages and check for updates, remain within the SDK settings screen and select the SDK Tools tab as shown in Figure 2-7:

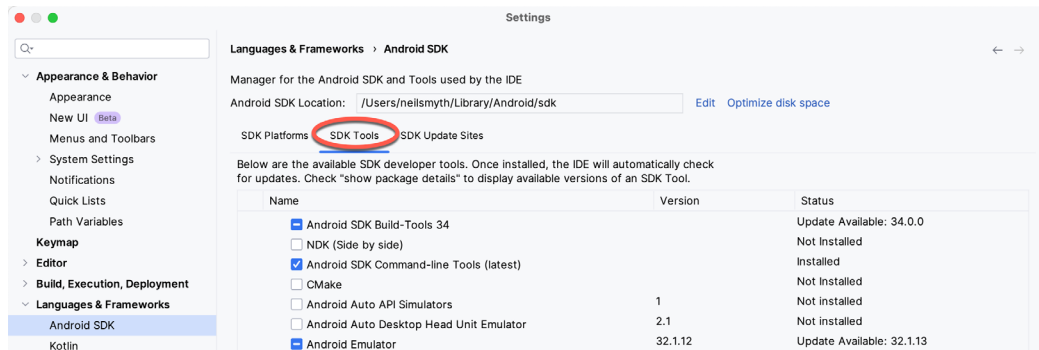


Figure 2-7

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 34

\*Note that 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, select the checkboxes next to those packages and click the *Apply* button to initiate the installation process. If the HAXM emulator settings dialog appears, select the recommended memory allocation:

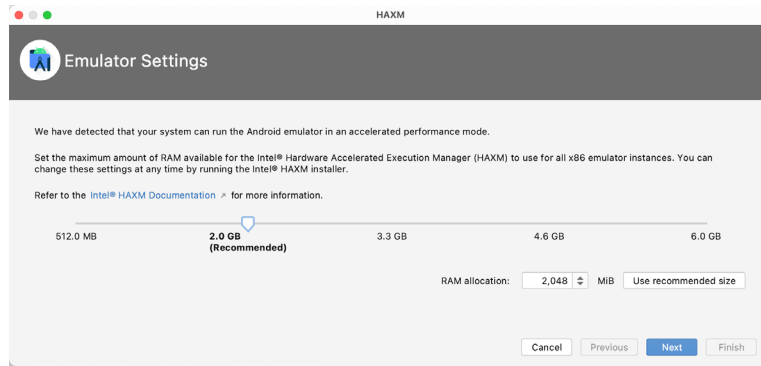


Figure 2-8

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

## 2.6 Installing the Android SDK Command-line Tools

Android Studio includes 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 locate the *Android SDK Command-line Tools (latest)* package as shown in Figure 2-9:

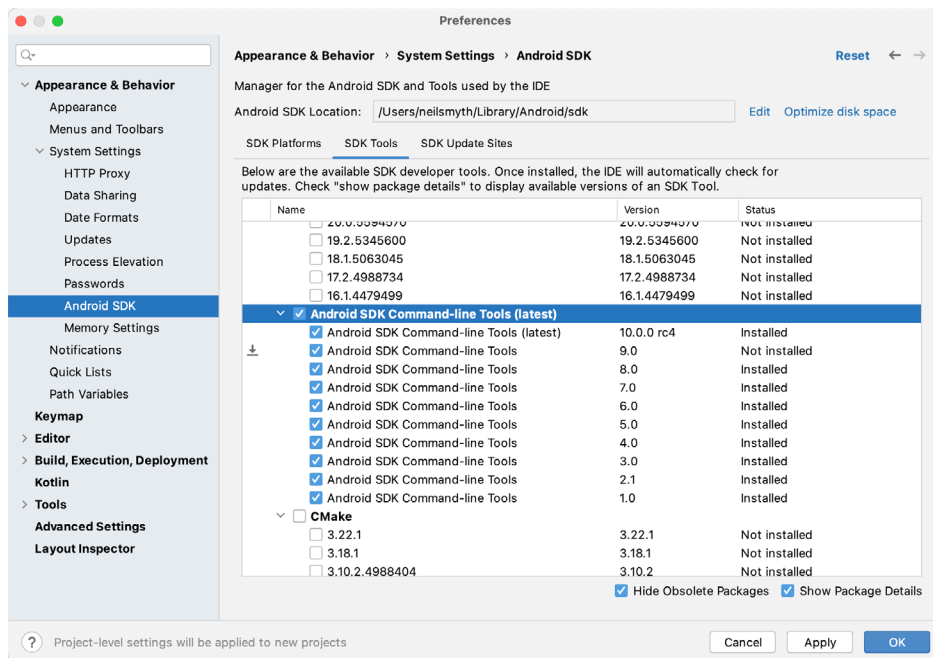


Figure 2-9

If the command-line tools package is not already installed, enable it and click *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.

## Setting up an Android Studio Development Environment

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:

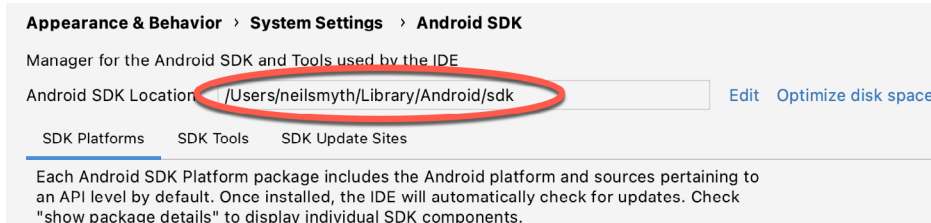


Figure 2-10

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 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 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 the *Environment Variables...* button. Follow the steps outlined for Windows 8.1 starting from step 3.

### 2.6.4 Linux

This configuration can be achieved on Linux 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 with 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:

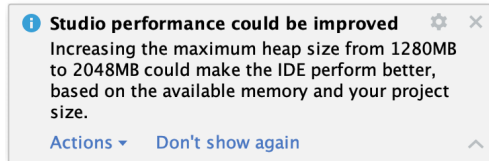


Figure 2-11

To view and modify the current memory configuration, select the *File -> Settings...* main menu option (*Android Studio -> Settings...* on macOS) 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:

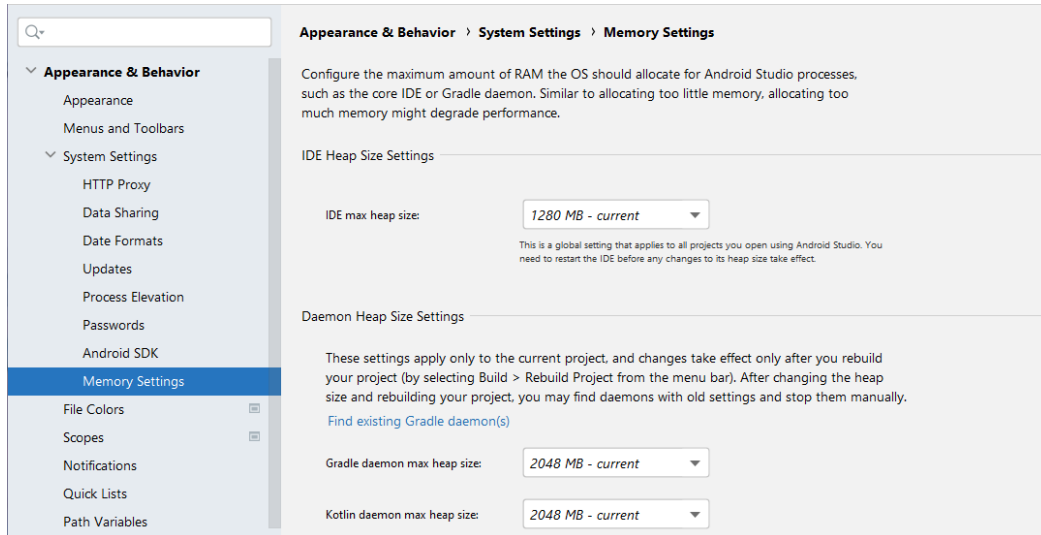


Figure 2-12

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, several 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 from the main menu.

## 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). This chapter covers 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 developing Android applications using the Android Studio IDE. Before moving on to slightly more advanced topics, now is a good time to validate that all 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 creating an Android application project using Android Studio. Once the project has been created, a later chapter will explore using 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 use one of the most basic Android Studio project templates. This simplicity allows us to introduce some key aspects of Android app development without overwhelming the beginner by introducing 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 the techniques and code used in this initial example project will be covered in much greater detail later.

### 3.2 Creating a New Android Project

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

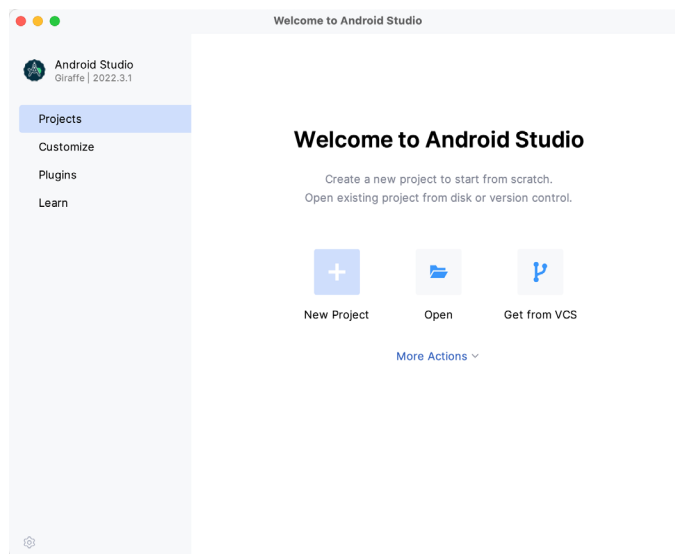


Figure 3-1

## Creating an Example Android App in Android Studio

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

### 3.3 Creating an Activity

The next step is to define the type of initial activity to be created for the application. Options are available to create projects for Phone and Tablet, Wear OS, Television, or Automotive. A range of different activity types is available when developing Android applications, many of which will be covered extensively in later chapters. For this example, however, 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.

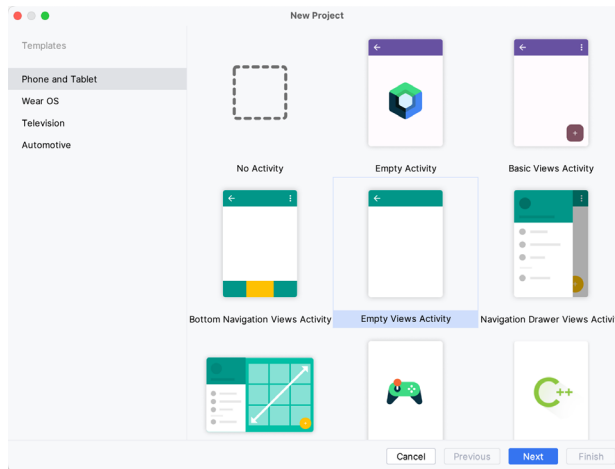


Figure 3-2

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

The *Package name* uniquely identifies 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 application's name. 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 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 (Oreo; Android 8.0). This minimum SDK will be used in most 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 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:

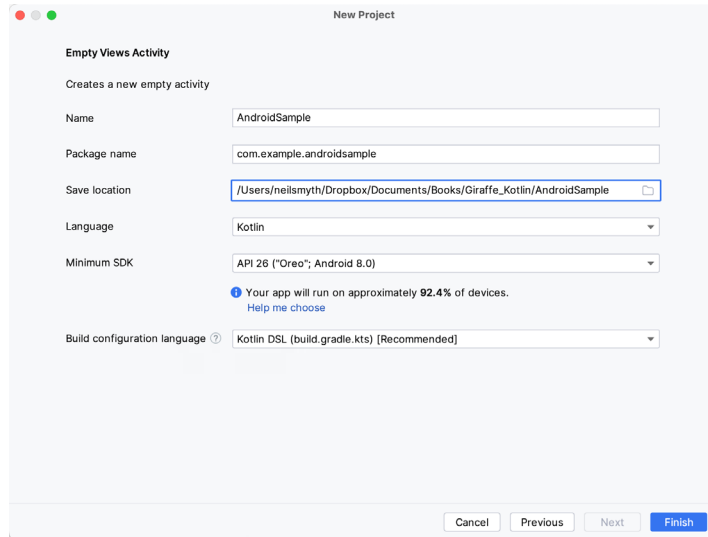


Figure 3-3

Finally, change the *Language* menu to *Kotlin* and select *Kotlin DSL (build.gradle.kts)* as the build configuration language before clicking *Finish* to create the project.

### 3.5 Enabling the New Android Studio UI

Android Studio is transitioning to a new, modern user interface that is not enabled by default in the Giraffe version. If your installation of Android Studio resembles Figure 3-4 below, then you will need to enable the new UI before proceeding:

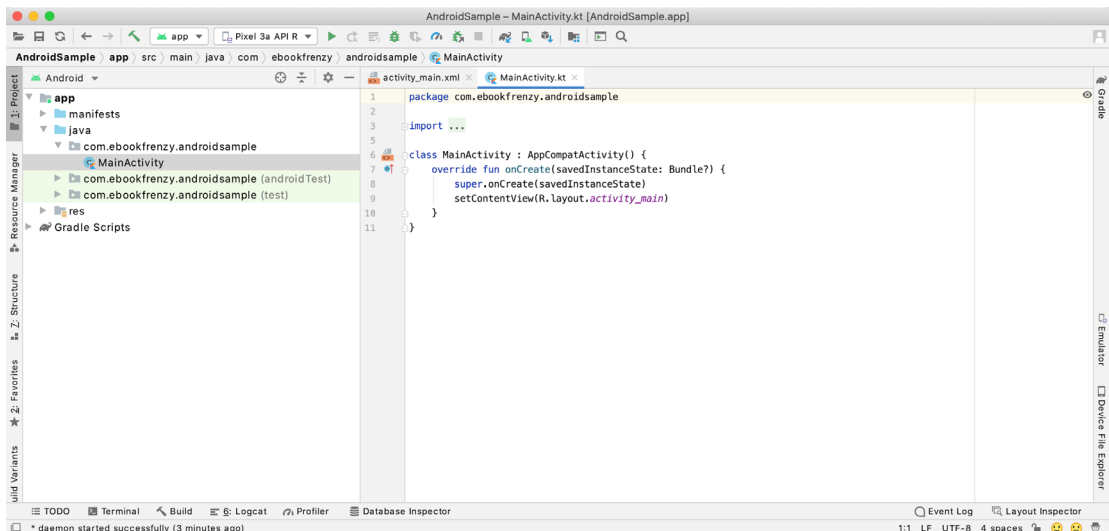


Figure 3-4

## Creating an Example Android App in Android Studio

Enable the new UI by selecting the *File -> Settings...* menu option (*Android Studio -> Settings...* on macOS) and selecting the New UI option under Appearance and Behavior in the left-hand panel. From the main panel, turn on the *Enable new UI* checkbox before clicking Apply, followed by OK to commit the change:

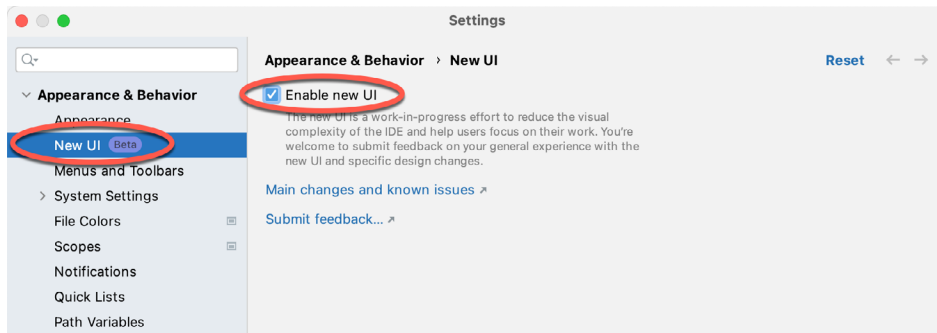


Figure 3-5

When prompted, restart Android Studio to activate the new user interface.

## 3.6 Modifying the Example Application

Once Android Studio has restarted, the main window will reappear using the new UI and containing our AndroidSample project as illustrated in Figure 3-6 below:

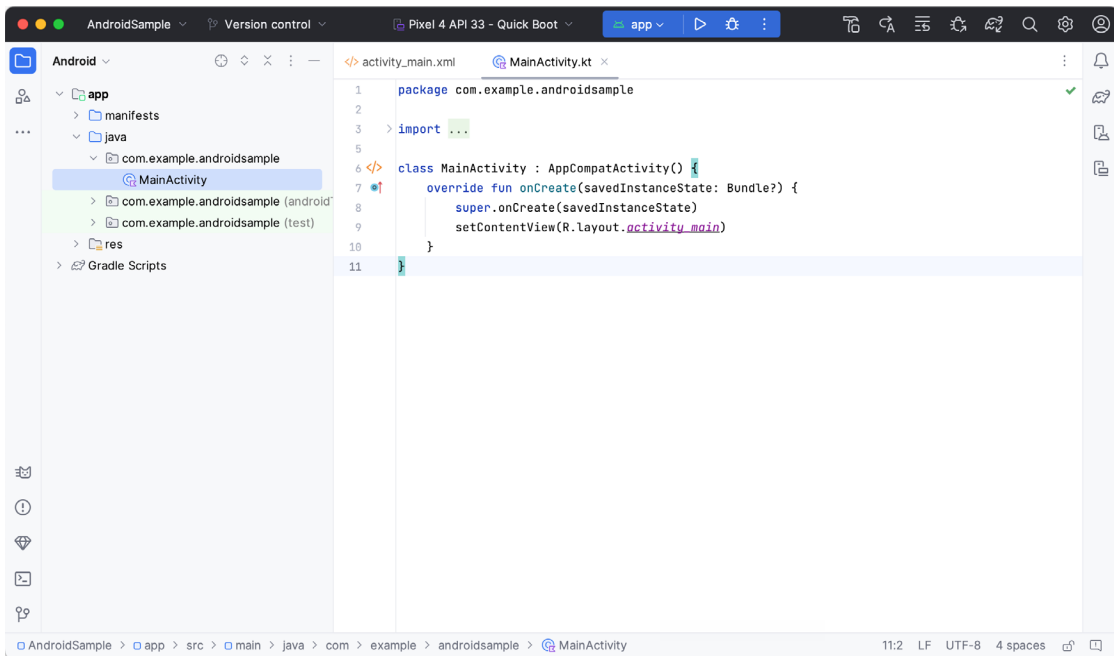


Figure 3-6

The newly created project and references to associated files are listed in the *Project* tool window on the left side of the main project window. The Project tool window has several 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 3-7. If the panel is not currently in Android mode, use the menu to switch mode:

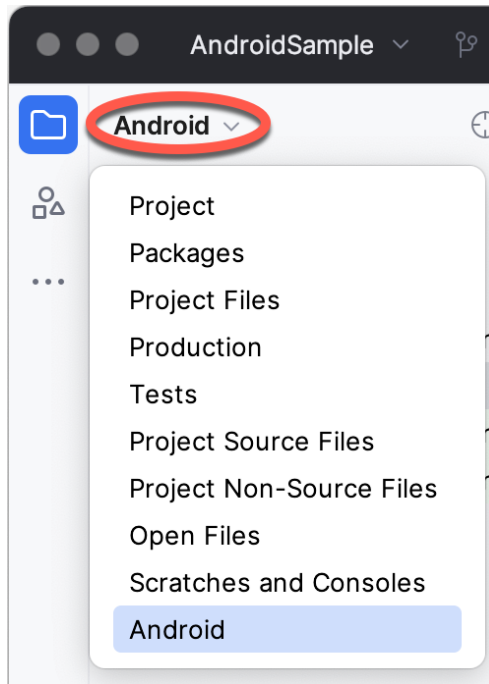


Figure 3-7

### 3.7 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 tool window 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:

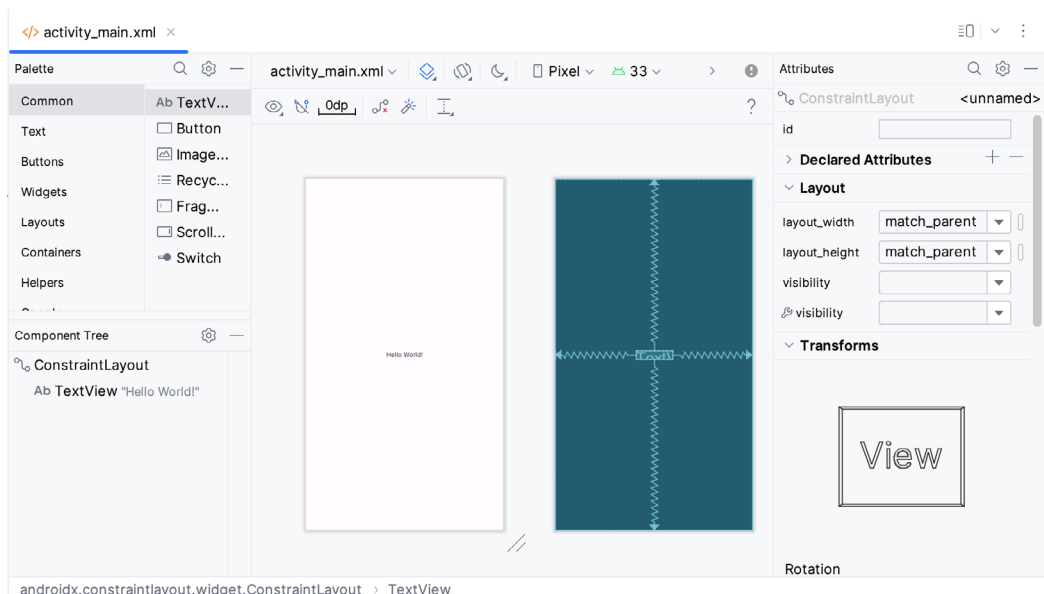




Figure 3-8

## Creating an Example Android App in Android Studio

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 range of other device options are available by clicking on this menu.

Use the System UI Mode button (  ) to turn Night mode on and off for the device screen layout. To change the orientation of the device representation between landscape and portrait, use the drop-down menu showing the  icon.

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. However, it should be noted that not all user interface components are 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 3-9:

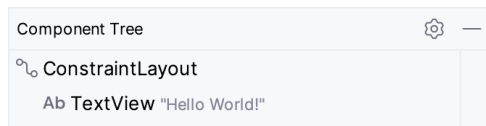


Figure 3-9

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, 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 ensure 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 U-shaped icon. When disabled, the icon appears with a diagonal line through it (Figure 3-10). If necessary, re-enable Autoconnect mode by clicking on this button.



Figure 3-10

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

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

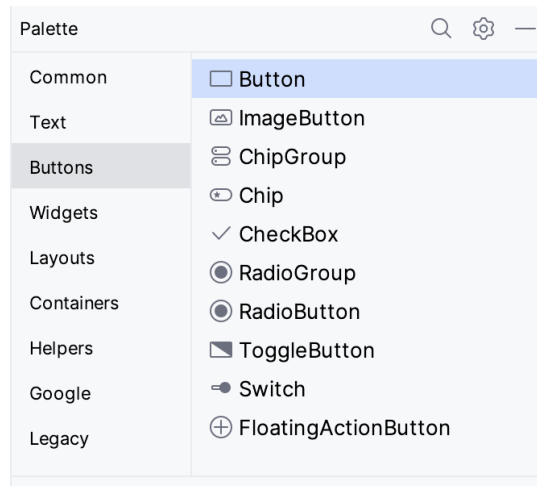


Figure 3-11

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

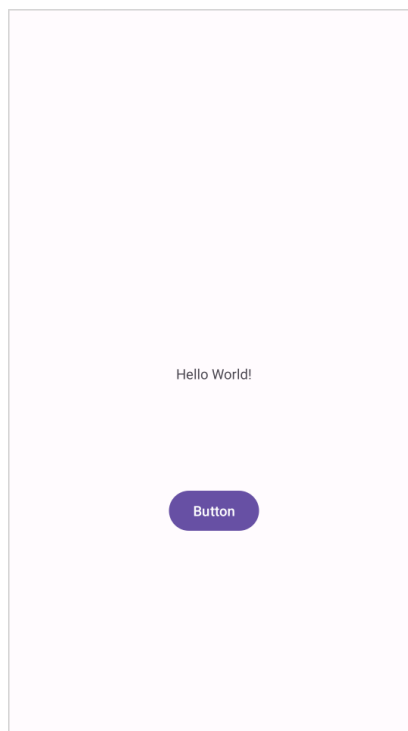


Figure 3-12

The next step is to change the text currently displayed by the *Button* component. The panel located to the right of the design area is the Attributes panel. This panel displays the attributes assigned to the currently selected component in the layout. Within this panel, locate the *text* property in the Common Attributes section and change the current value from “Button” to “Convert”, as shown in Figure 3-13:

## Creating an Example Android App in Android Studio

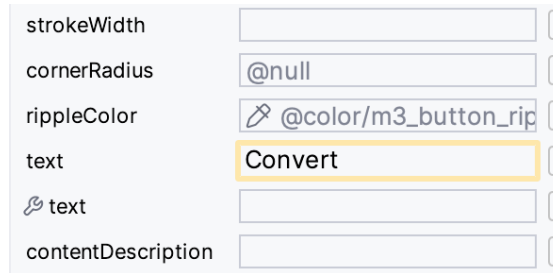


Figure 3-13

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

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

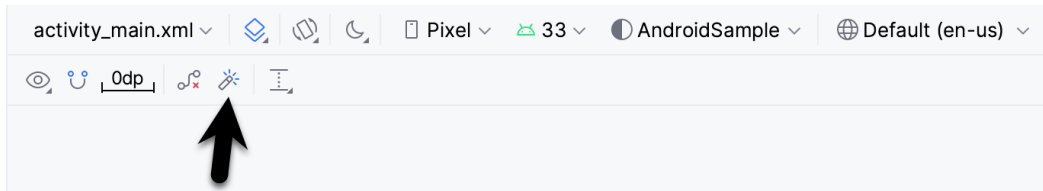


Figure 3-14

It is important to explain the warning button in the top right-hand corner of the Layout Editor tool, as indicated in Figure 3-15. This warning indicates potential problems with the layout. For details on any problems, click on the button:



Figure 3-15

When clicked, the Problems tool window (Figure 3-16) will appear, describing the nature of the problems:

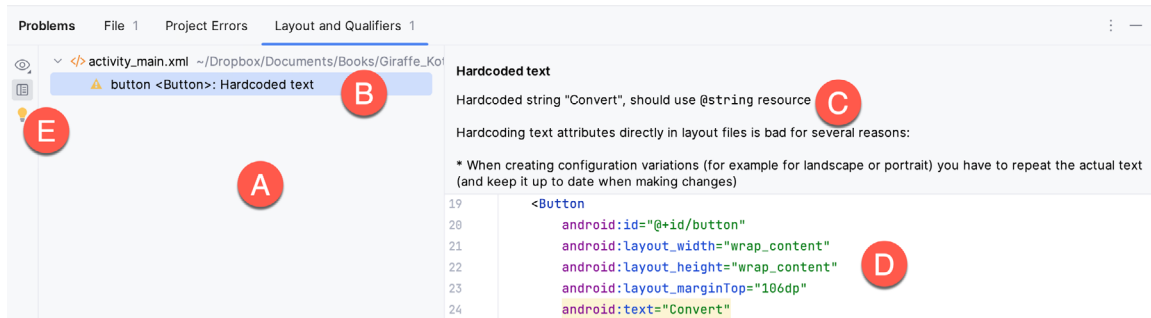


Figure 3-16

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 informs us that a potential issue exists concerning 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 reminds us that attributes and values such as text strings should be stored as *resources* wherever possible when developing Android applications. 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 3-17:

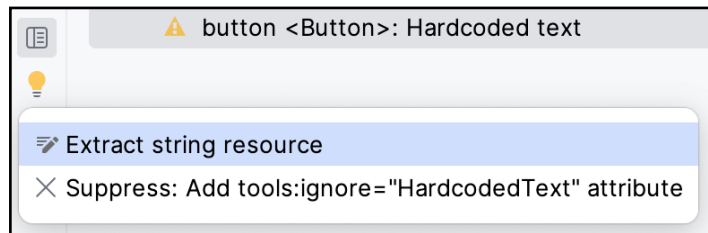


Figure 3-17

After selecting this option, the *Extract Resource* panel (Figure 3-18) 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:

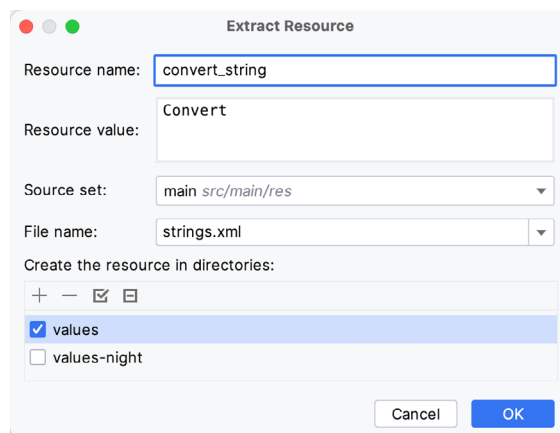


Figure 3-18

## Creating an Example Android App in Android Studio

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 3-19:

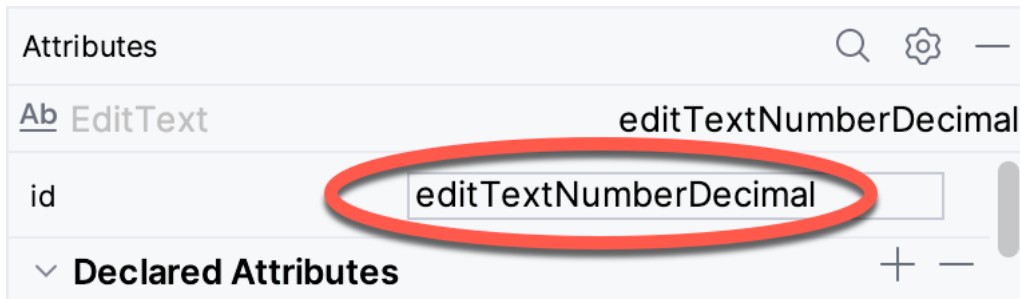


Figure 3-19

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:

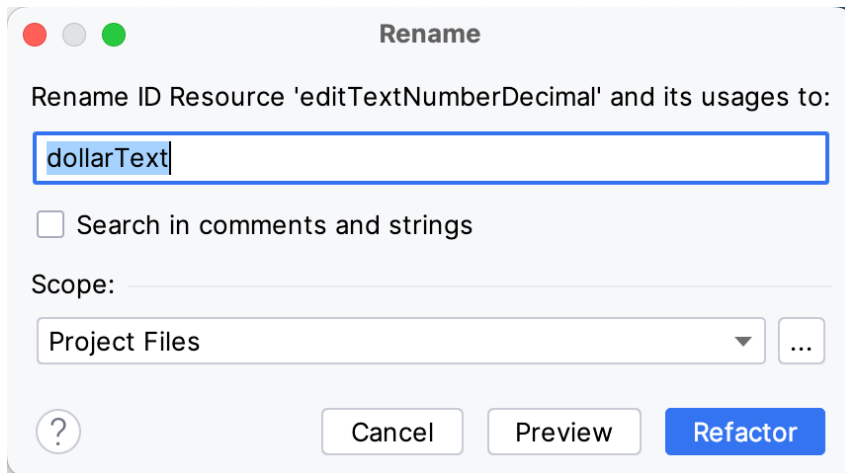


Figure 3-20

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

Add any missing layout constraints by clicking on the *Infer Constraints* button. At this point, the layout should resemble that shown in Figure 3-21:

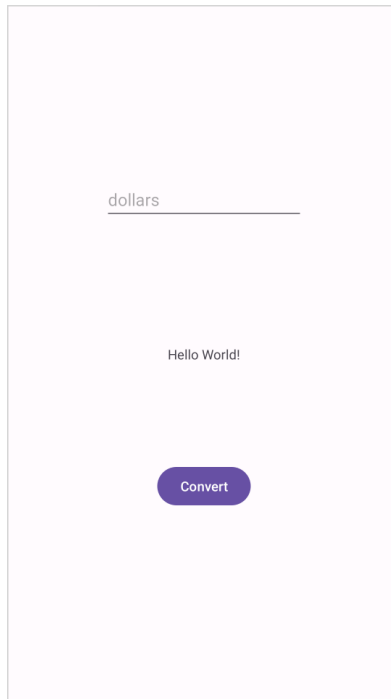


Figure 3-21

### 3.8 Reviewing the Layout and Resource Files

Before moving on to the next step, we will look at some internal aspects of user interface design and resource handling. In the previous section, we changed 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 is the View Modes menu button marked A in Figure 3-22 below:

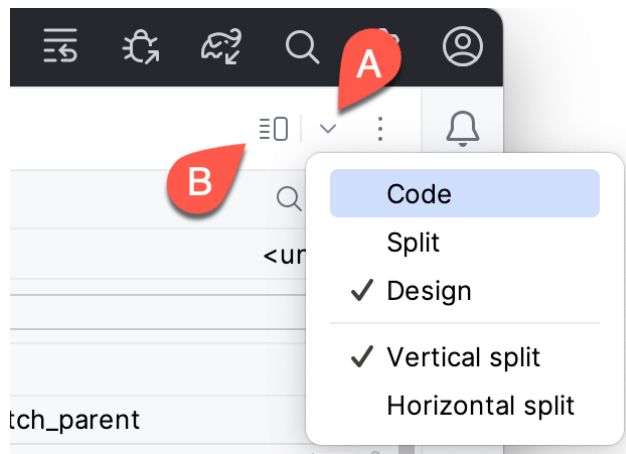


Figure 3-22

By default, the editor will be in *Design* mode, whereby just the visual representation of the layout is displayed.

## Creating an Example Android App in Android Studio

In *Code* mode, the editor will display the XML for the layout, while in *Split* mode, both the layout and XML are displayed, as shown in Figure 3-23:

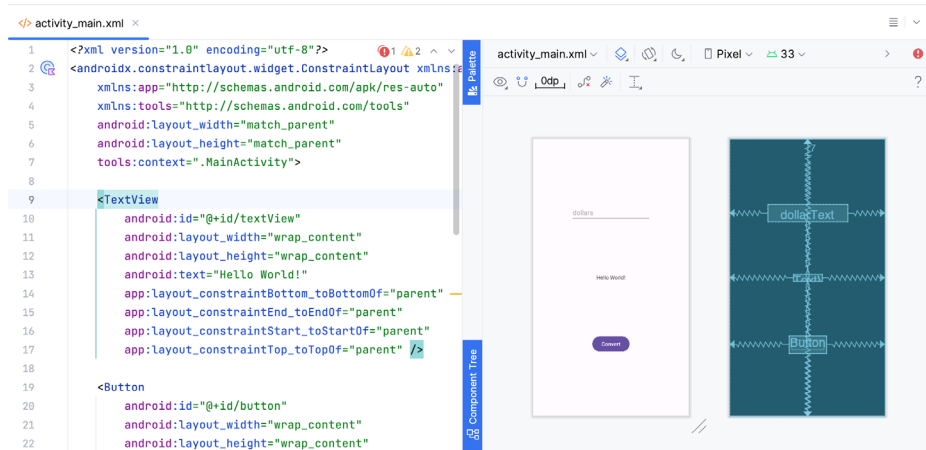


Figure 3-23

The button to the left of the View Modes button (marked B in Figure 3-22 above) is used to toggle between Code and Split modes quickly.

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 complexity and content vary, 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 layout color changes in real-time to match the new setting in the XML file. Note also that a small red square appears in the XML editor's left margin (also called the *gutter*) 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:

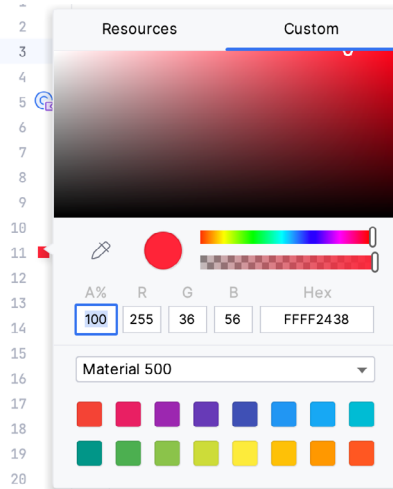


Figure 3-24

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>
```

To demonstrate 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 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:

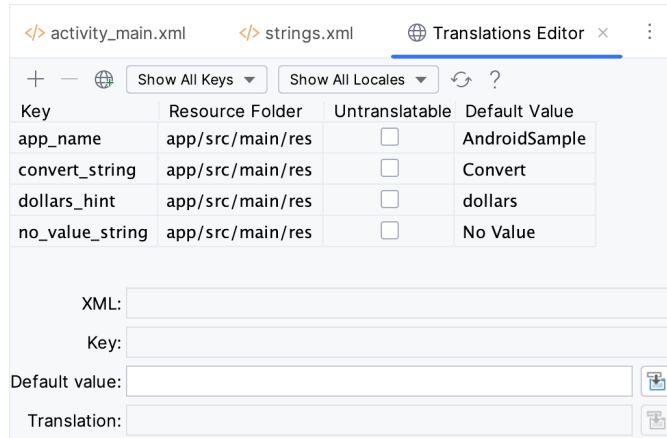


Figure 3-25

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

### 3.9 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 several ways and is covered 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:

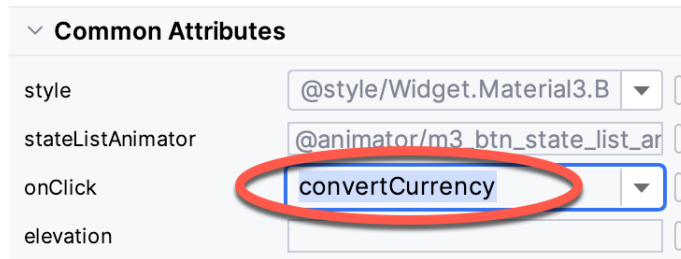


Figure 3-26

Next, 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*”.

### 3.10 Summary

While not excessively complex, several steps are involved in setting up an Android development environment. Having performed those steps, it is worth working through an example to ensure 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 string values, and briefly touched on layouts. Next, we looked at the underlying XML used to store Android application user interface designs.

Finally, an onClick event was added to a Button connected to a method implemented to extract the user input from the EditText component, convert it 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, compiling and running an entire app will be necessary to thoroughly test that it works. An Android application may be tested by installing and running it 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 creating such a virtual device using the Pixel 4 phone as a reference example.

### 4.1 About Android Virtual Devices

AVDs are emulators that allow Android applications to be tested without needing to install the application on a physical Android-based device. An AVD may be configured to emulate various hardware features, including screen size, memory capacity, and the presence or otherwise of features such as a camera, GPS navigation support, or an accelerometer. Several emulator templates are installed as part of the standard Android Studio installation, allowing AVDs to be configured for various 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 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 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 clicking the *Device Manager* button in the right-hand tool window bar, as indicated in Figure 4-1:

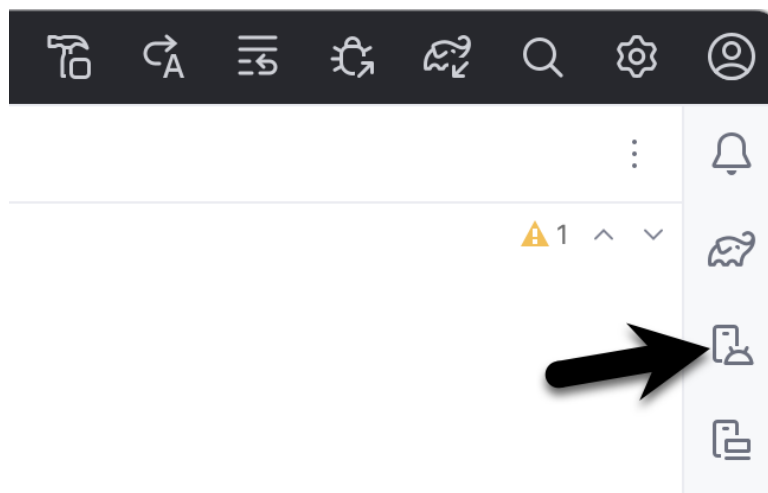


Figure 4-1

Once opened, the manager will appear as a tool window, as shown in Figure 4-2:

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

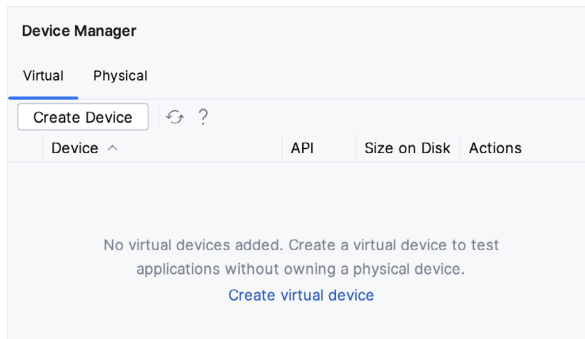


Figure 4-2

If you installed Android Studio for the first time on a computer (as opposed to upgrading an existing Android Studio installation), the installer might have created an initial AVD instance ready for use, as shown in Figure 4-3:

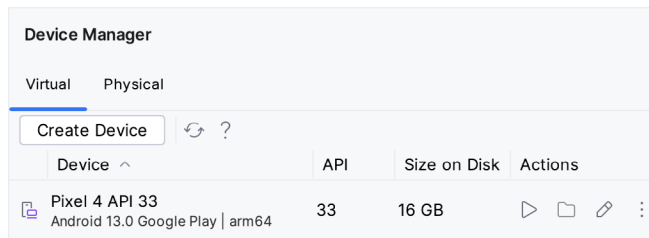


Figure 4-3

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:

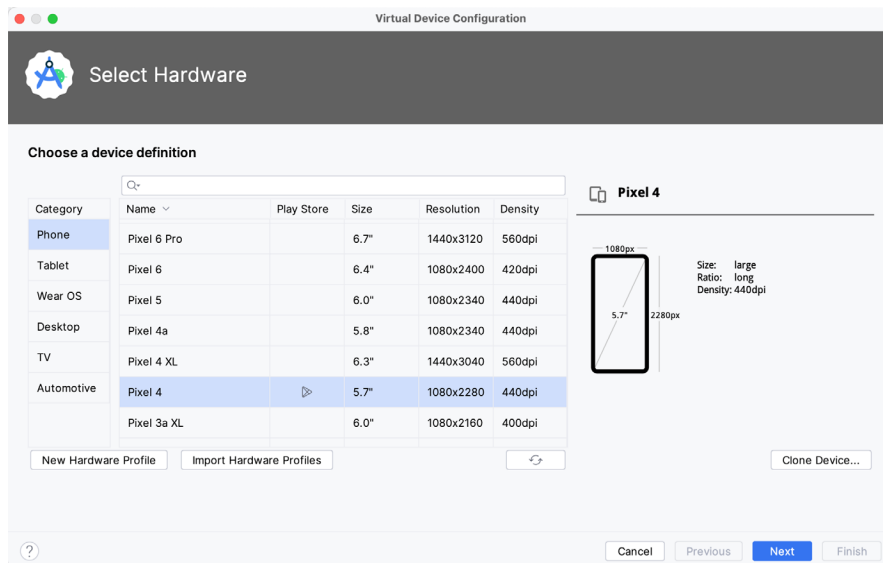


Figure 4-4

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

1. Select the Phone option From the Category panel to display the 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. 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 accept the default name.
5. Click *Finish* to create the AVD.
6. If future modifications to the AVD are necessary, 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 test the newly created AVD emulator, select the emulator from the Device Manager and click 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:

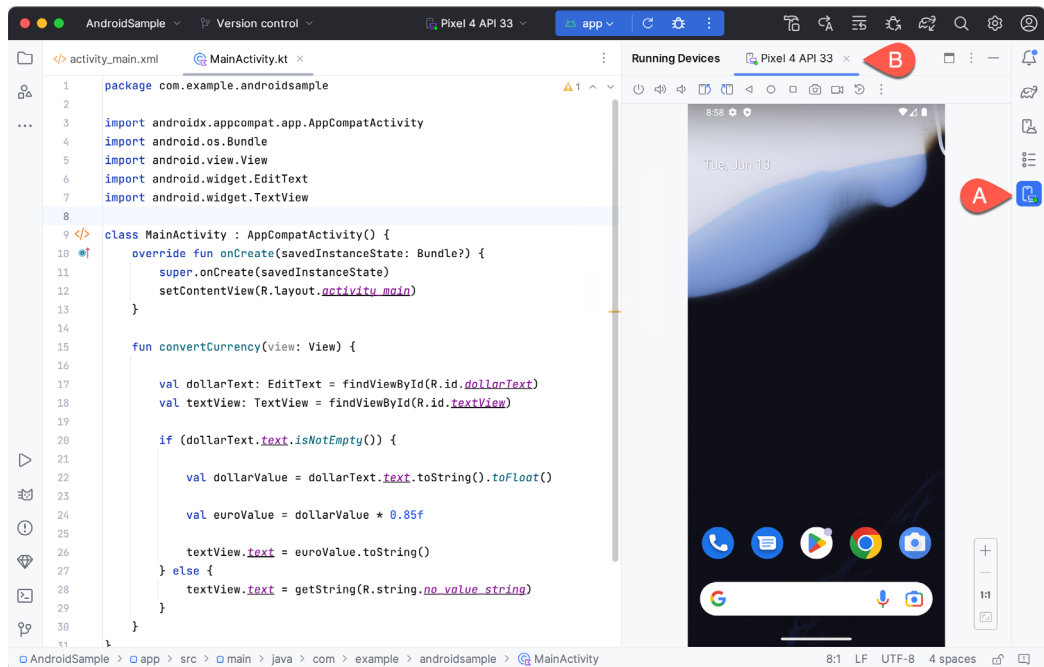


Figure 4-5

To hide and show the emulator tool window, click the Running Devices tool window button (marked A above). Click 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 4-6, for example, shows a tool window with two emulator sessions:

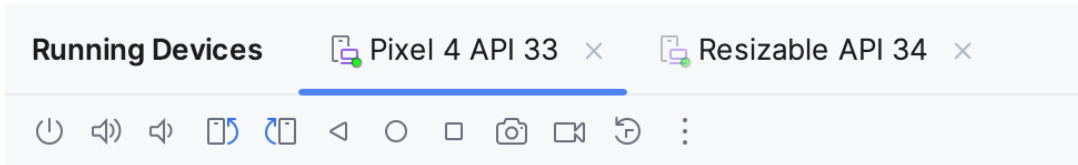


Figure 4-6

To switch between sessions, click on the corresponding tab.

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

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

### 4.3 Running the Application in the AVD

With an AVD emulator configured, the example *AndroidSample* application created in the earlier chapter can now 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 4-7 below), then either click the run button represented by a triangle (B), select the *Run -> Run 'app'* menu option or use the *Ctrl-R* keyboard shortcut:

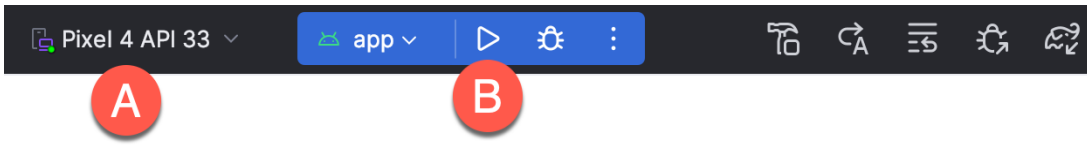


Figure 4-7

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:

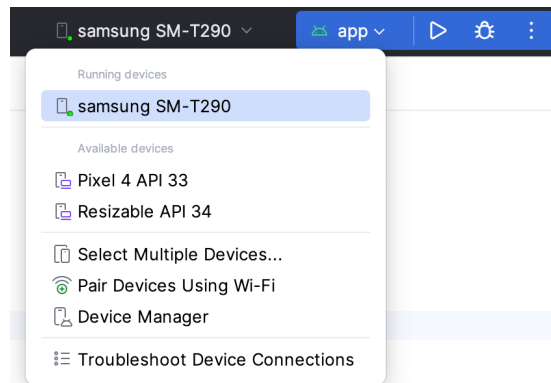


Figure 4-8

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*”):

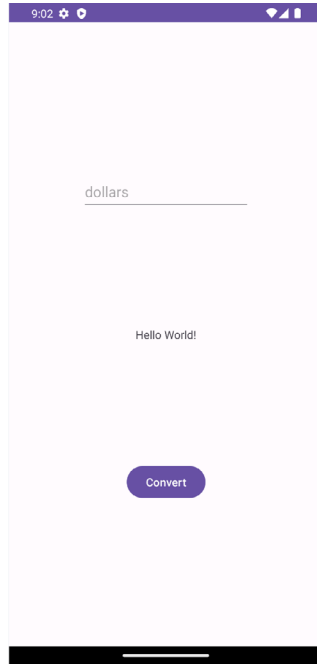


Figure 4-9

Once the run process begins, the Run tool window will appear. The Run tool window will display diagnostic information as the application package is installed and launched. Figure 4-10 shows the Run tool window output from a typical successful application launch:

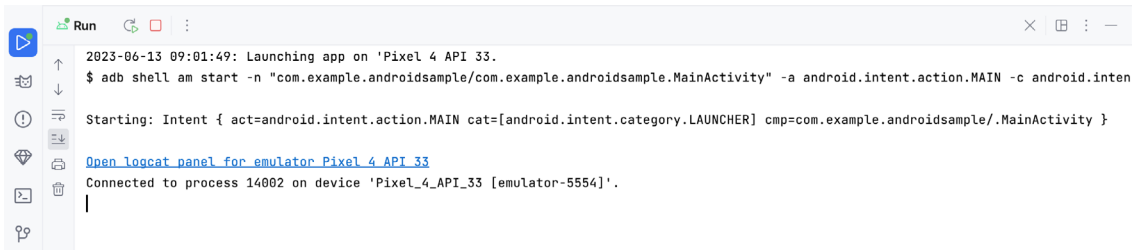


Figure 4-10

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

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

## 4.4 Running on Multiple Devices

The run target menu shown in Figure 4-8 above includes an option to run the app on multiple emulators and devices in parallel. When selected, this option displays the dialog in Figure 4-11, providing a list of 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:

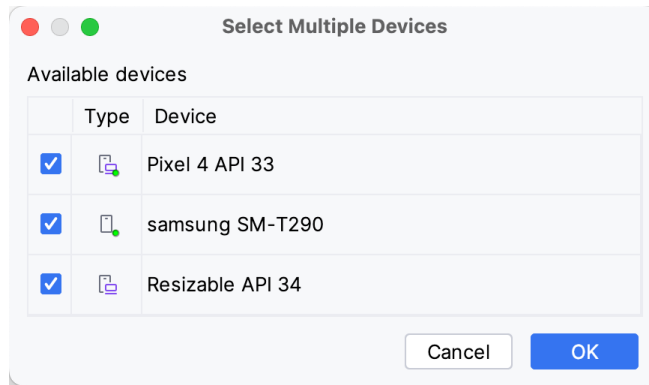


Figure 4-11

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

## 4.5 Stopping a Running Application

To stop a running application, click the stop button located in the main toolbar, as shown in Figure 4-12:

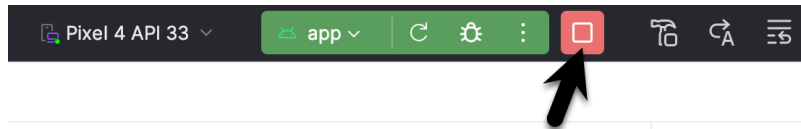


Figure 4-12

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 4-13 below:



Figure 4-13

## 4.6 Supporting Dark Theme

To test how an app behaves when dark theme is enabled, open the Settings app within the running Android instance in the emulator, choose the *Display* category, and enable the *Dark theme* option as shown in Figure 4-14:

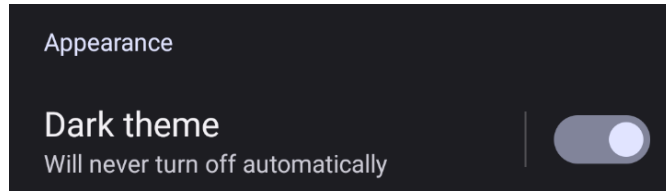


Figure 4-14

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 4-15:

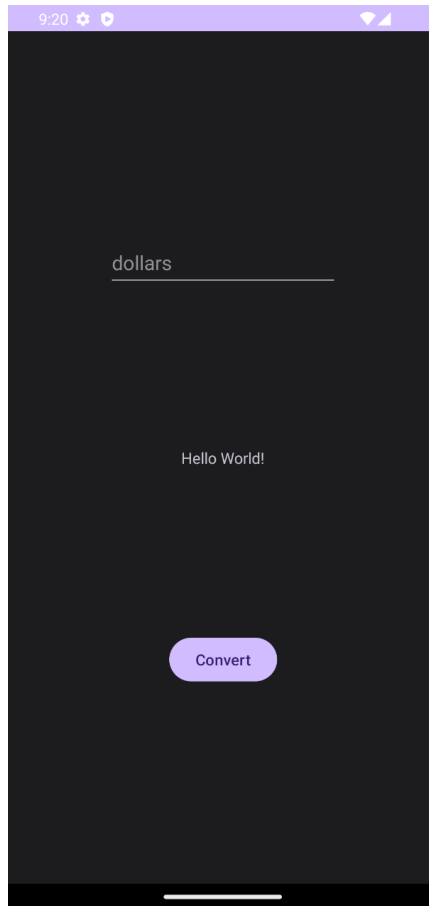


Figure 4-15

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. The emulator can be configured to appear in a separate window within the Settings dialog, which can be displayed by clicking on the IDE and Project Settings button located in the Android Studio toolbar, as highlighted in Figure 4-16:

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

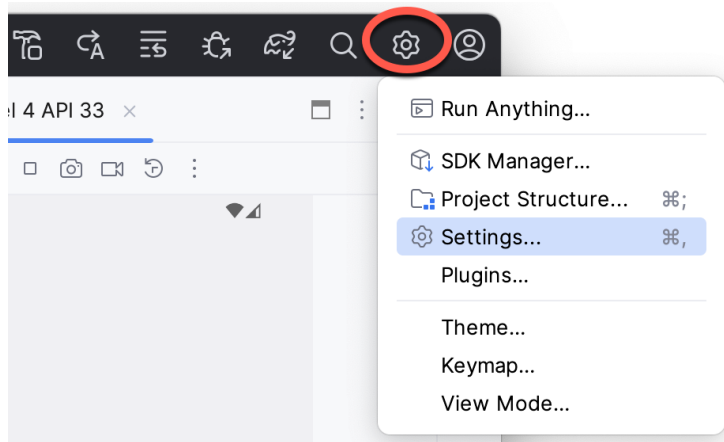


Figure 4-16

Within the Settings dialog, navigate to *Tools -> Emulator* in the side panel, and disable the *Launch in a tool window* option:

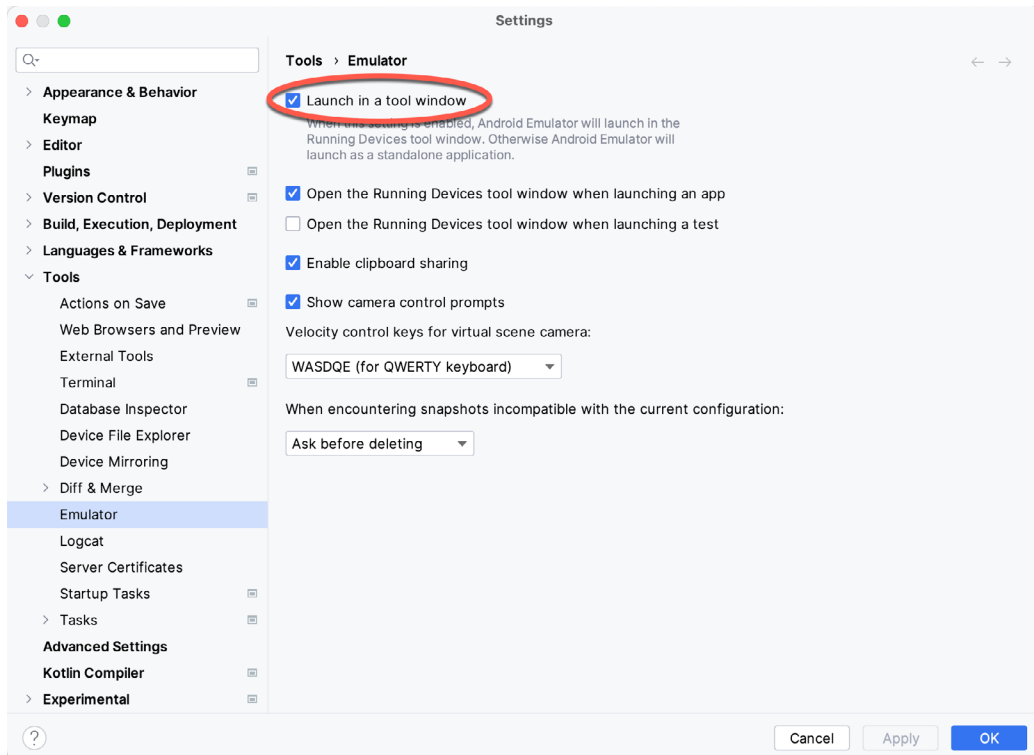


Figure 4-17

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 4-5 above.

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



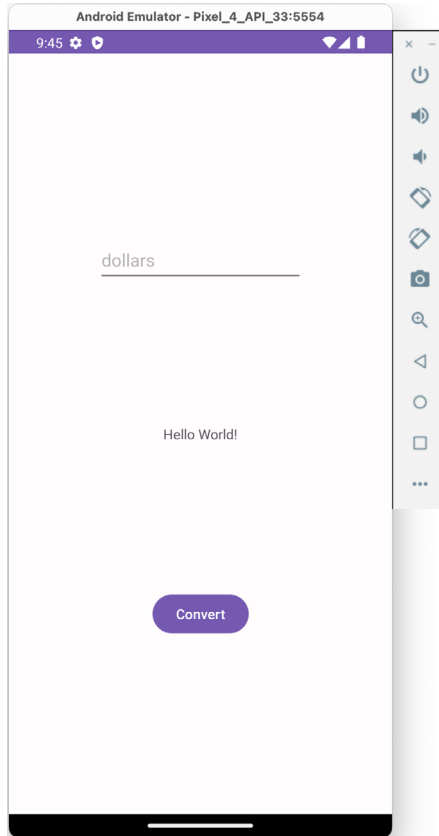


Figure 4-18

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 Running Devices tool window may also be detached from the main Android Studio window from within the tool window Options menu, which is accessed by clicking the button indicated in Figure 4-19:

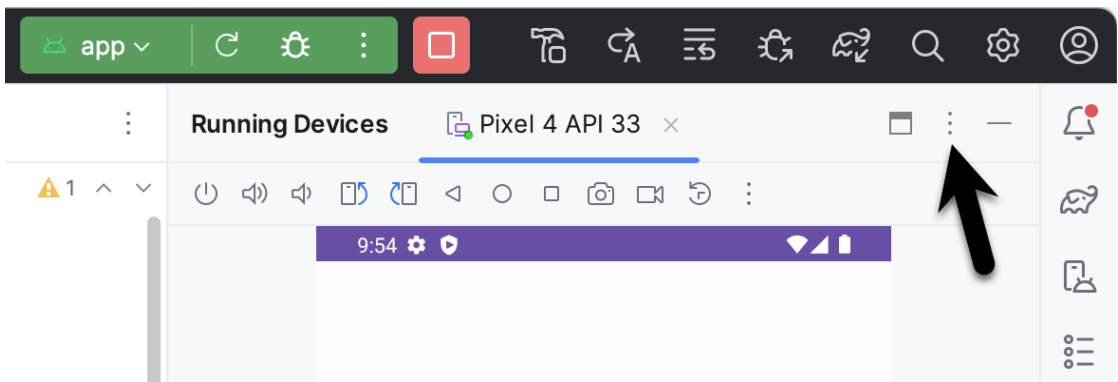


Figure 4-19

From within the Options menu, select *View Mode -> Float* to detach the tool window from the Android Studio main window:

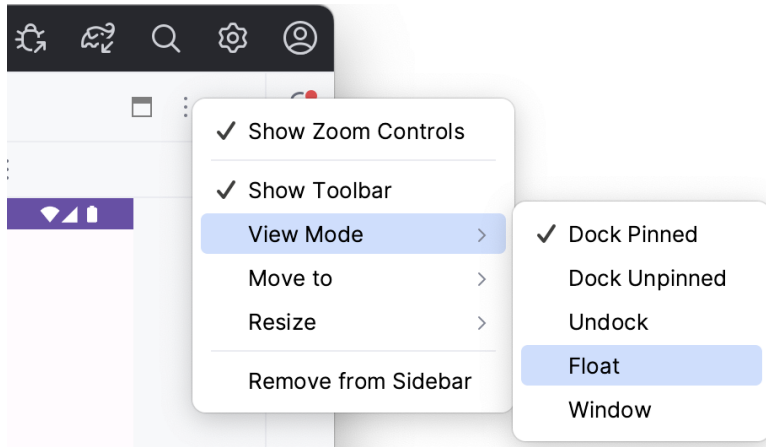


Figure 4-20

To re-dock the Running Devices tool window, click on the Dock button shown in Figure 4-21:

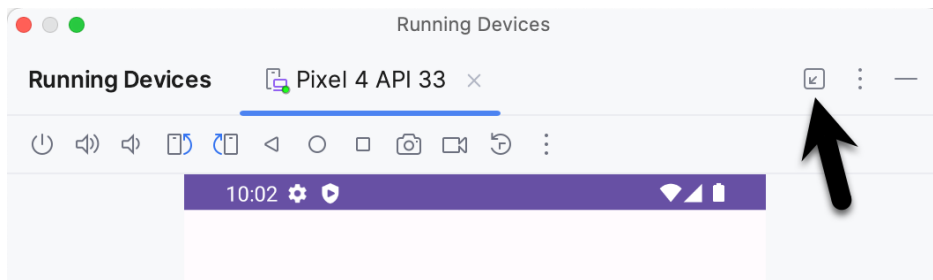


Figure 4-21

## 4.8 Enabling the Device Frame

The emulator can be configured to appear with or without the device frame. To change the setting, exit the emulator, open the Device Manager, select the AVD from the list, and click on the pencil icon in the Actions column. In the settings screen, locate and change the Enable Device Frame option:

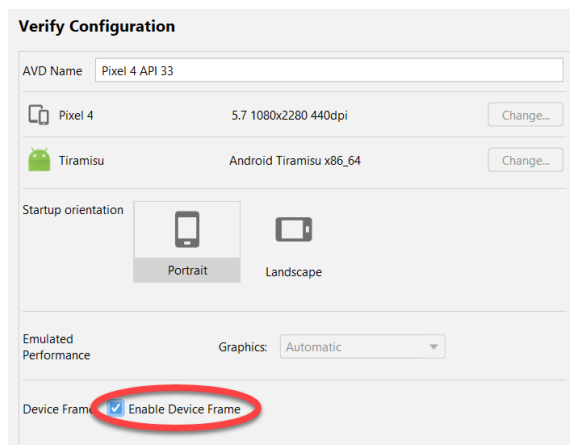


Figure 4-22

Once the device frame has been enabled, the emulator will appear as shown in Figure 4-23 the next time it is launched:

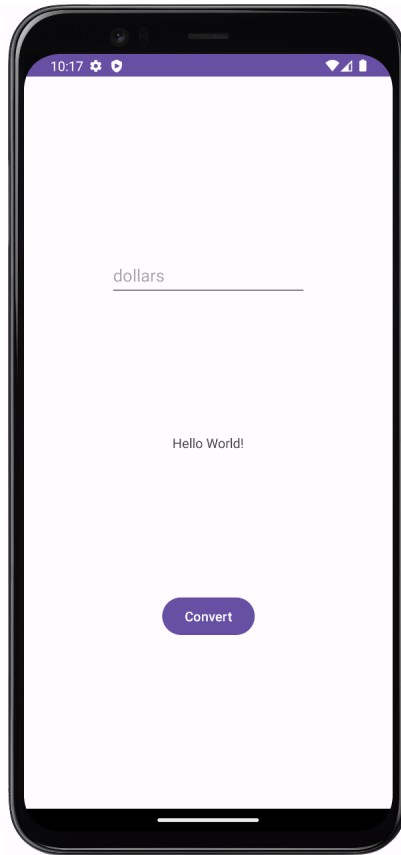


Figure 4-23

### 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 a physical Android device or an Android Virtual Device (AVD) emulator. AVDs are created and managed using the Android Studio Device Manager tool, which may be used 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, it 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 shorten the learning curve in later chapters of the book. With this in mind, this chapter will provide an overview of the various areas and components of 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 the next time it is launched, automatically opening the previously active project.

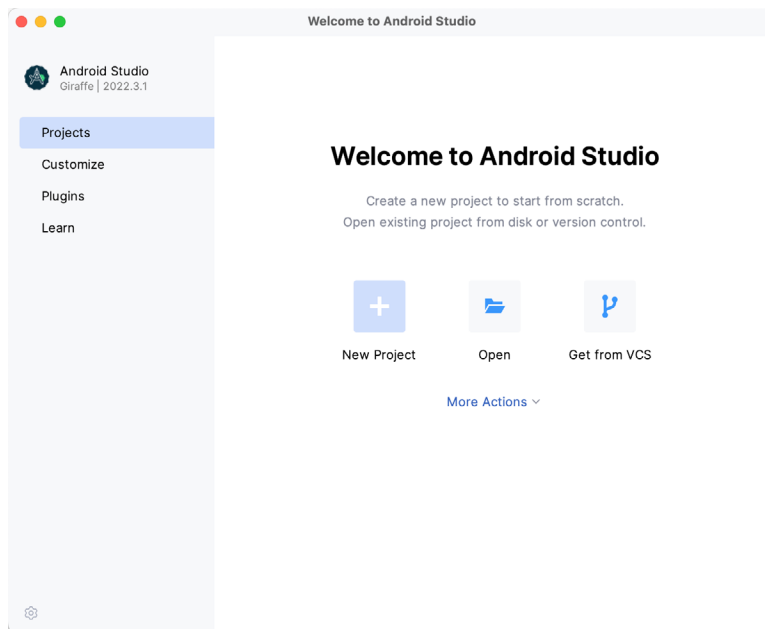


Figure 6-1

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 selecting the More Actions link or using the menu shown in Figure 6-2 when

## A Tour of the Android Studio User Interface

the list of recent projects replaces the More Actions link:

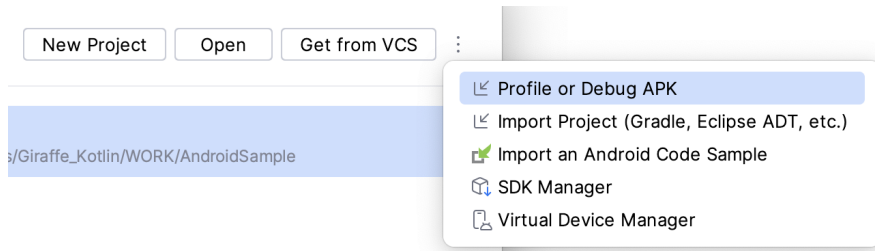


Figure 6-2

## 6.2 The Menu Bar

The Android Studio main window will appear when a new project is created, or an existing one is opened. When multiple projects are open simultaneously, each will be assigned its own main window. The precise configuration of the window will vary depending on the operating system Android Studio is running on and which tools and panels were displayed the last time the project was open. The appearance, for example, of the main menu bar will differ depending on the host operating system. On macOS, Android Studio follows the standard convention of placing the menu bar along the top edge of the desktop, as illustrated in Figure 6-3:

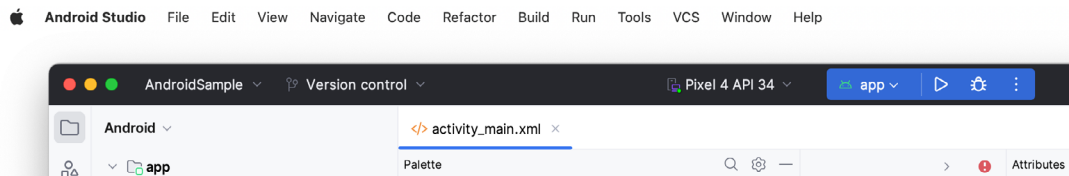


Figure 6-3

When Android Studio is running on Windows or Linux, however, the main menu is accessed via the button highlighted in Figure 6-4:

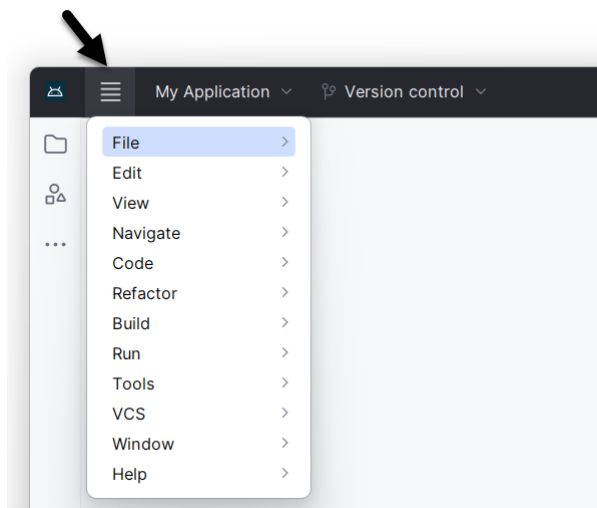


Figure 6-4

## 6.3 The Main Window

Once a project is open, the Android Studio main window will typically resemble that of Figure 6-5:

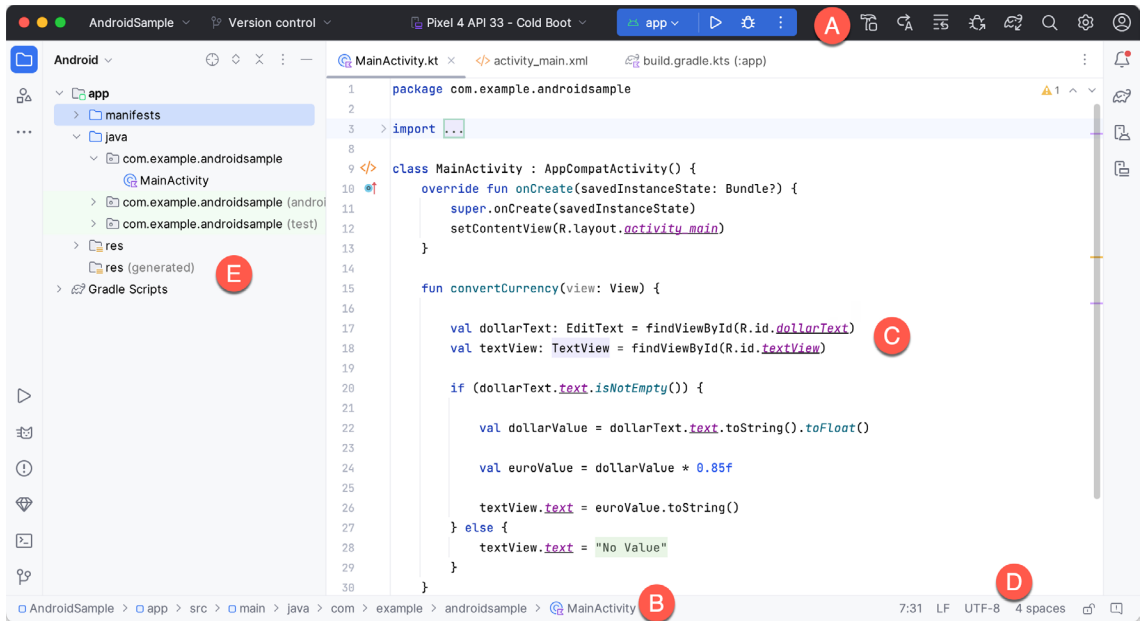


Figure 6-5

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

**A – 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 Toolbar...* menu option. The toolbar menu shown in Figure 6-6 provides a convenient way to perform tasks such as creating and opening projects and switching between windows when multiple projects are open:

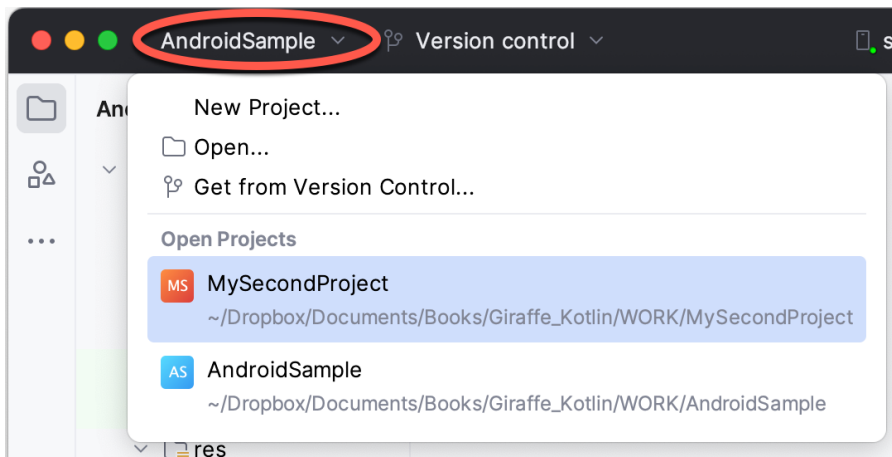


Figure 6-6

**B – 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:

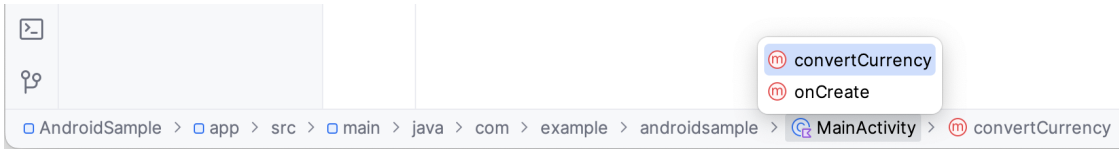


Figure 6-7

Select a method from the list to be taken to the corresponding location within the code editor. You can hide, display, and change the position of this bar using the *View -> Appearance -> Navigation Bar* menu option.

**C – 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 6-8:

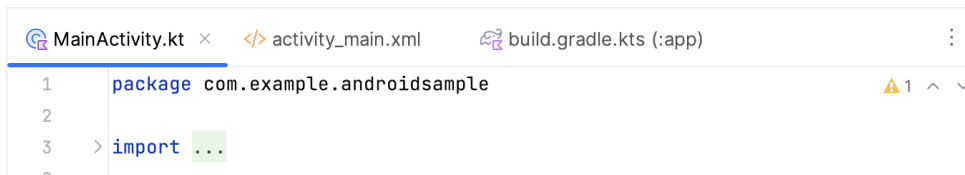


Figure 6-8

**D – Status Bar** – The status bar displays informational messages about the project and the activities of Android Studio. Hovering over items in the status bar will display a description of that field. Many fields are interactive, allowing users to click to perform tasks or obtain more detailed status information.

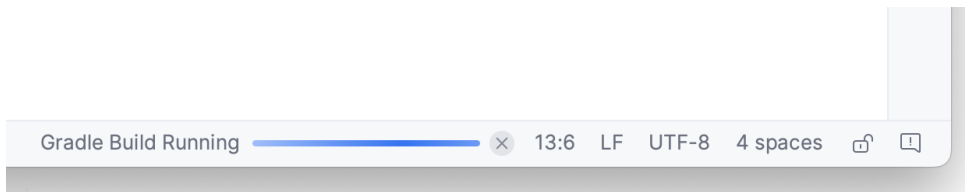


Figure 6-9

The widgets displayed in the status bar can be changed using the *View -> Appearance -> Status Bar Widgets* menu.

**E – 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 available tools within the Android Studio environment.

## 6.4 The Tool Windows

In addition to the project view tool window, Android Studio also includes many other windows, which, when enabled, are displayed *tool window bars* that appear along the left and right edges of the main window and contain buttons for showing and hiding each of the tool windows. Figure 6-10 shows typical tool window bar configurations, though the buttons and their positioning may differ for your Android Studio installation.



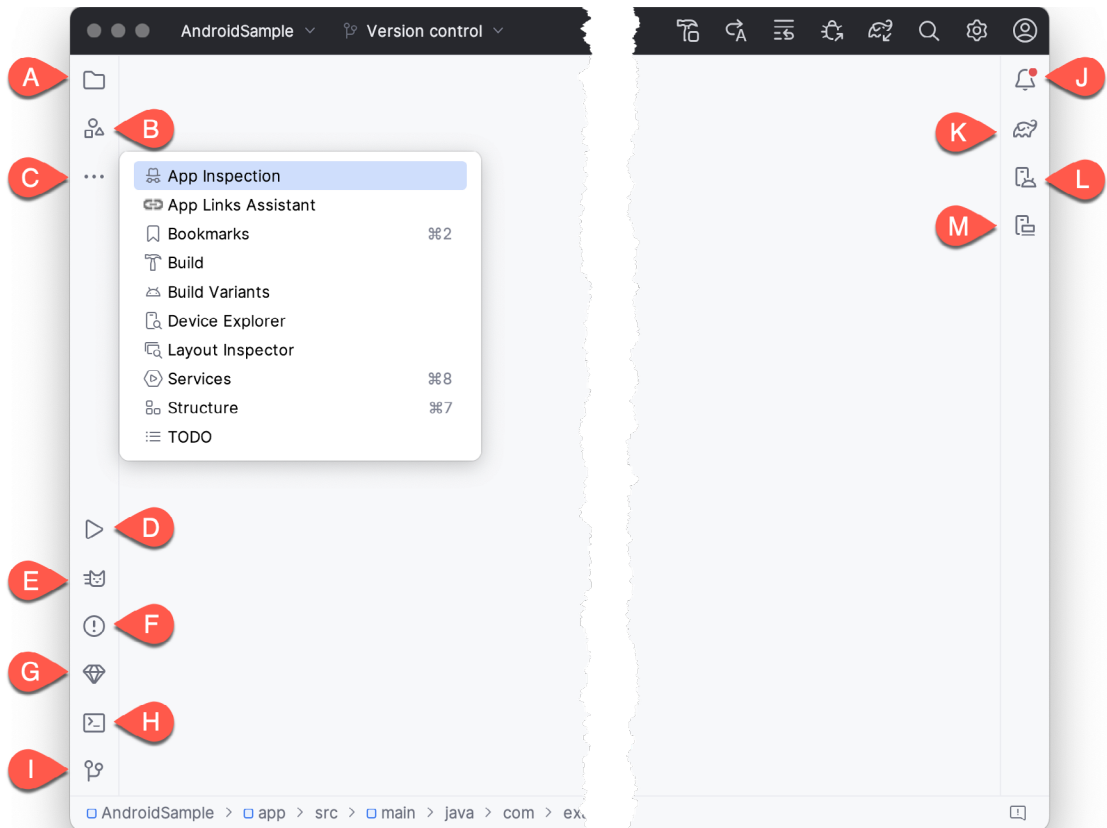


Figure 6-10

Clicking on a button will display the corresponding tool window, while a second click will hide the window. 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.

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

- **Project (A)** – 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 (B)** - A tool for adding and managing resources and assets within the project, such as images, colors, and layout files.
- **More Tool Windows (C)** - Displays a menu containing additional tool windows not currently displayed in a tool window bar. When a tool window is selected from this menu, it will appear as a button in a tool window bar.
- **Run (D)** – 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 fails to install and run on a device or emulator, this window typically provides diagnostic information about the problem.
- **Logcat (E)** – The Logcat tool window provides access to the monitoring log output from a running application

and options for taking screenshots and videos of the application and stopping and restarting a process.

- **Problems (F)** - A central location 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.
- **App Quality Insights (G)** - Provides access to the cloud-based Firebase app quality and crash analytics platform.
- **Terminal (H)** - 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.
- **Version Control (I)** - This tool window is used when the project files are under source code version control, allowing access to Git repositories and code change history.
- **Notifications (J)** - This tool window is used when the project files are under source code version control, allowing access to Git repositories and code change history.
- **Gradle (K)** - The Gradle tool window provides a view of the Gradle tasks that make up the project build configuration. The window lists the tasks 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.
- **Device Manager (L)** - Provides access to the Device Manager tool window where physical Android device connections and emulators may be added, removed, and managed.
- **Running Devices (M)** - 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*”.
- **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 running. The Background Task Inspector allows background worker tasks created using WorkManager to be monitored and managed.
- **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.
- **Build** - The build tool window displays information about the build process while a project is being compiled and packaged and 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.
- **Layout Inspector** - Provides a visual 3D rendering of the hierarchy of components that make up a user interface layout.
- **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.

- **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 opening the Settings dialog and navigating to the *TODO* entry listed under *Editor*.

## 6.5 The Tool Window Menus

Each tool window has its own toolbar along the top edge. The menu buttons within these toolbars vary from one tool to the next, though all tool windows contain an Options menu (marked A in Figure 6-11):

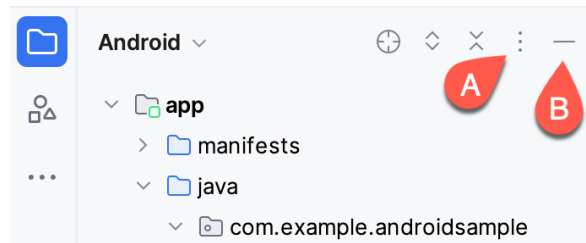


Figure 6-11

The Options menu allows various aspects of the window to be changed. Figure 6-12, for example, shows the Options menu for the Project tool window. Settings 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:

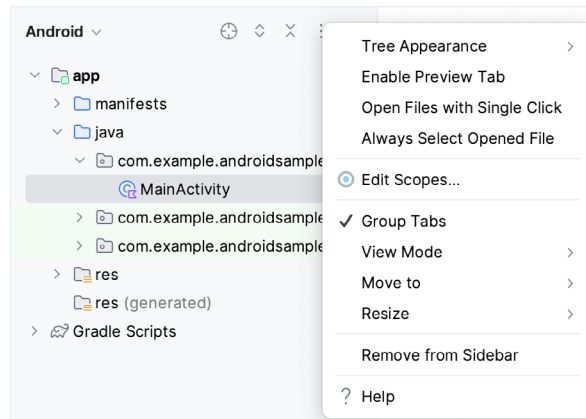


Figure 6-12

All tool windows also include a far-right button on the toolbar (marked B in Figure 6-11 above), providing an additional way to hide the tool window from view. A search of the items within a tool window can be performed by giving that window focus by clicking on 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.

## 6.6 Android Studio Keyboard Shortcuts

Android Studio includes many keyboard shortcuts to save time when performing common tasks. A complete keyboard shortcut keymap listing can be viewed and printed from within the Android Studio project window by selecting the *Help -> Keyboard Shortcuts PDF* menu option. You may also list and modify the keyboard shortcuts by opening the Settings dialog and clicking on the Keymap entry, as shown in Figure 6-13 below:

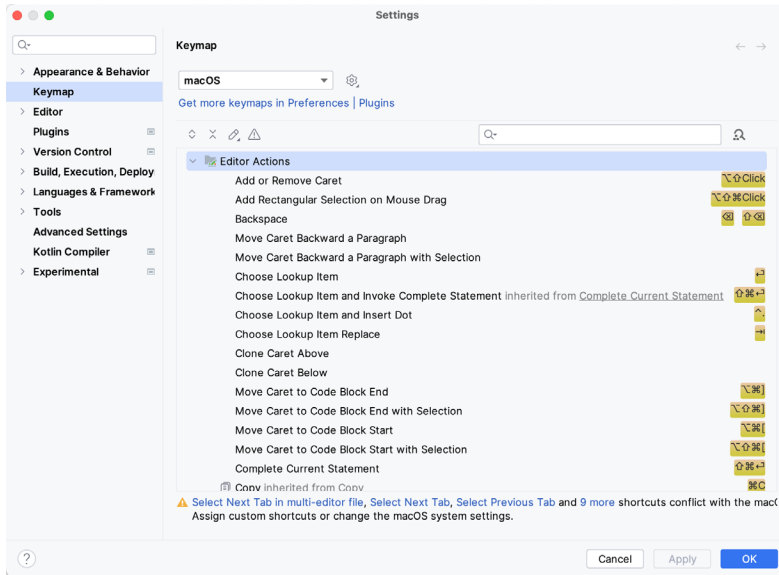


Figure 6-13

## 6.7 Switcher and Recent Files Navigation

Another useful mechanism for navigating within the Android Studio main window involves using 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 6-14).

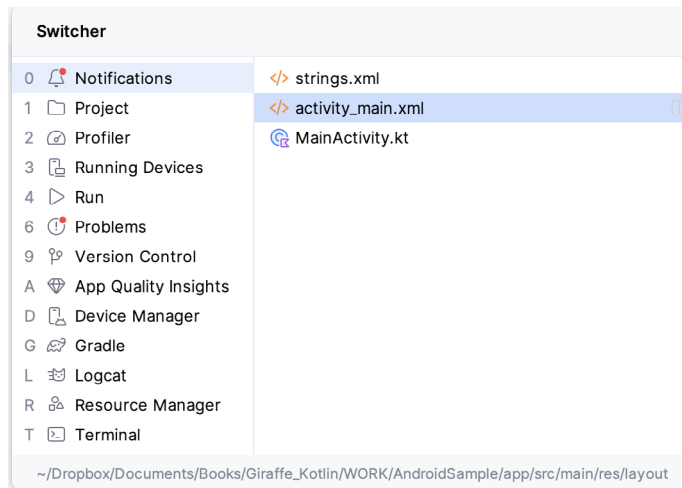


Figure 6-14

Once displayed, the switcher will remain visible 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, the Recent Files panel provides navigation to recently opened files (Figure 6-15). 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 can be used to scroll through the file name and tool window options. Pressing the Enter key will select the currently highlighted item:

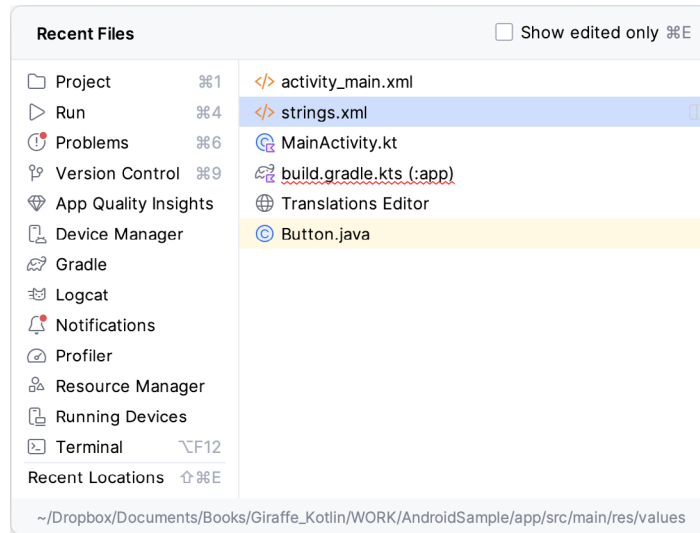


Figure 6-15

## 6.8 Changing the Android Studio Theme

The overall theme of the Android Studio environment may be changed using the Settings dialog. Once the settings dialog is displayed, select the *Appearance & Behavior* option in the left-hand panel, followed by *Appearance*. Then, change the setting of the *Theme* menu before clicking on the OK button. The themes available will depend on the platform but usually include options such as Light, IntelliJ, Windows, High Contrast, and Darcula. Figure 6-16 shows an example of the main window with the Dark theme selected:

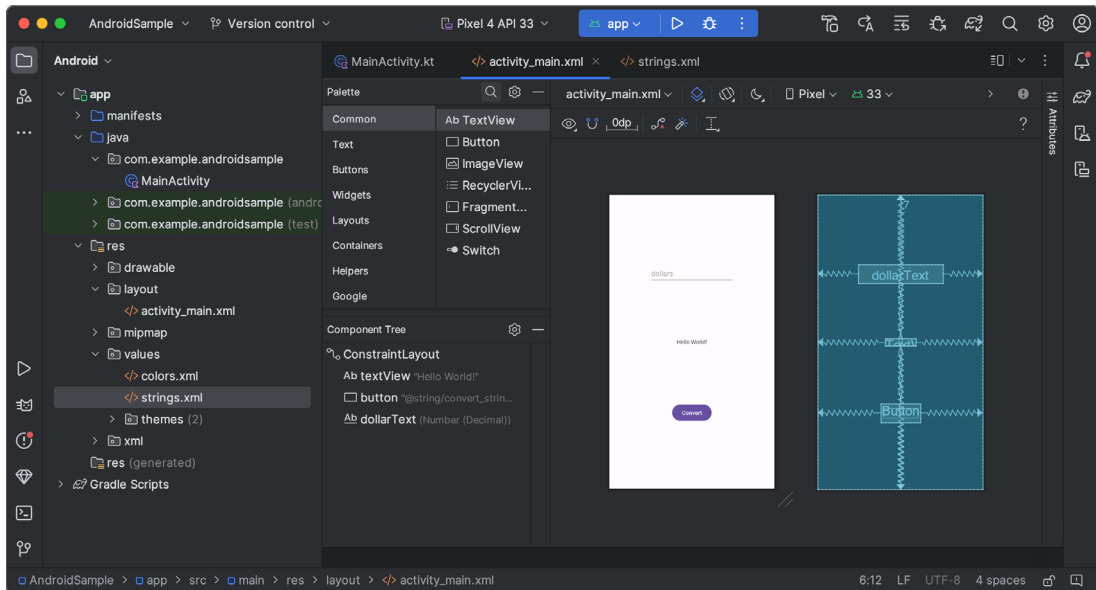


Figure 6-16

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:

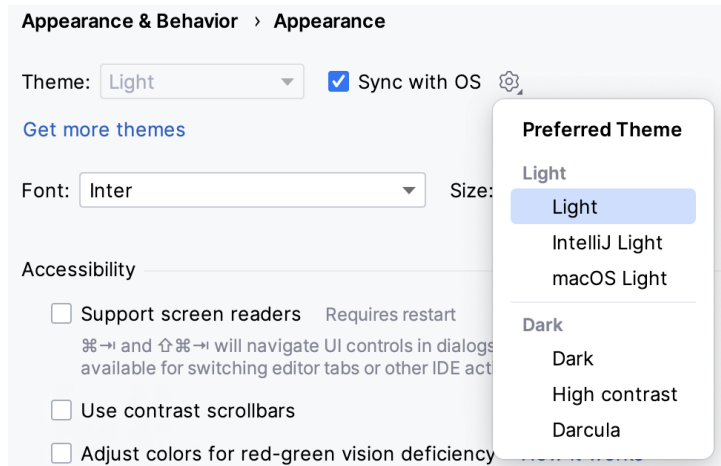


Figure 6-17

## 6.9 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 of the main window.

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.

## 9. An Overview of the Android Architecture

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

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

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

### 9.1 The Android Software Stack

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

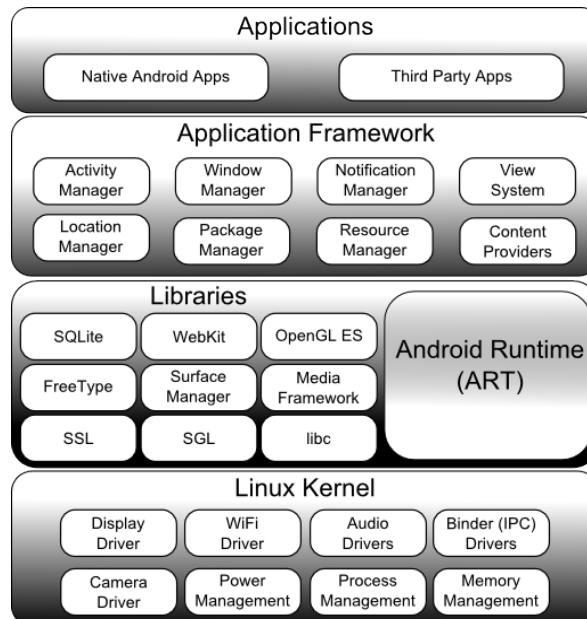


Figure 9-1

## 9.2 The Linux Kernel

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

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

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

## 9.3 Android Runtime – ART

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

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

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

## 9.4 Android Libraries

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

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

- **android.app** – Provides access to the application model and is the cornerstone of all Android applications.
- **android.content** – Facilitates content access, publishing, and messaging between applications and application components.
- **android.database** – Used to access data published by content providers and includes SQLite database management classes.
- **android.graphics** – A low-level 2D graphics drawing API including colors, points, filters, rectangles, and canvases.
- **android.hardware** – Presents an API providing access to hardware such as the accelerometer and light sensor.
- **android.opengl** – A Java interface to the OpenGL ES 3D graphics rendering API.



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

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

### 9.4.1 C/C++ Libraries

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

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

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

## 9.5 Application Framework

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

The Android framework includes the following key services:

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

## 9.6 Applications

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

## 9.7 Summary

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

## 10. The Anatomy of an Android App

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

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

### 10.1 Android Activities

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

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

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

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

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

### 10.2 Android Fragments

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

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

screen.

## 10.3 Android Intents

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

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

## 10.4 Broadcast Intents

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

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

## 10.5 Broadcast Receivers

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

## 10.6 Android Services

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

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

Example situations where a Service might be a practical solution include, as previously mentioned, the streaming of audio that should continue when the application is no longer active or a stock market tracking application that needs to notify the user when a share hits a specified price.

## 10.7 Content Providers

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

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

## 10.8 The Application Manifest

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

## 10.9 Application Resources

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

## 10.10 Application Context

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

## 10.11 Summary

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

Maximum reuse and interoperability are promoted by creating individual, standalone functionality modules in the form of activities and intents while implementing content providers to achieve data sharing between applications.

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

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

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



## 11. An Introduction to Kotlin

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

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

### 11.1 What is Kotlin?

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

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

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

### 11.2 Kotlin and Java

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

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

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

### 11.3 Converting from Java to Kotlin

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

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

### 11.4 Kotlin and Android Studio

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

### 11.5 Experimenting with Kotlin

When learning a new programming language, it is often useful to be able to enter and execute snippets of code. One of the best ways to do this with Kotlin is to use the Kotlin Playground (Figure 11-1) located at <https://play.kotlinlang.org>:

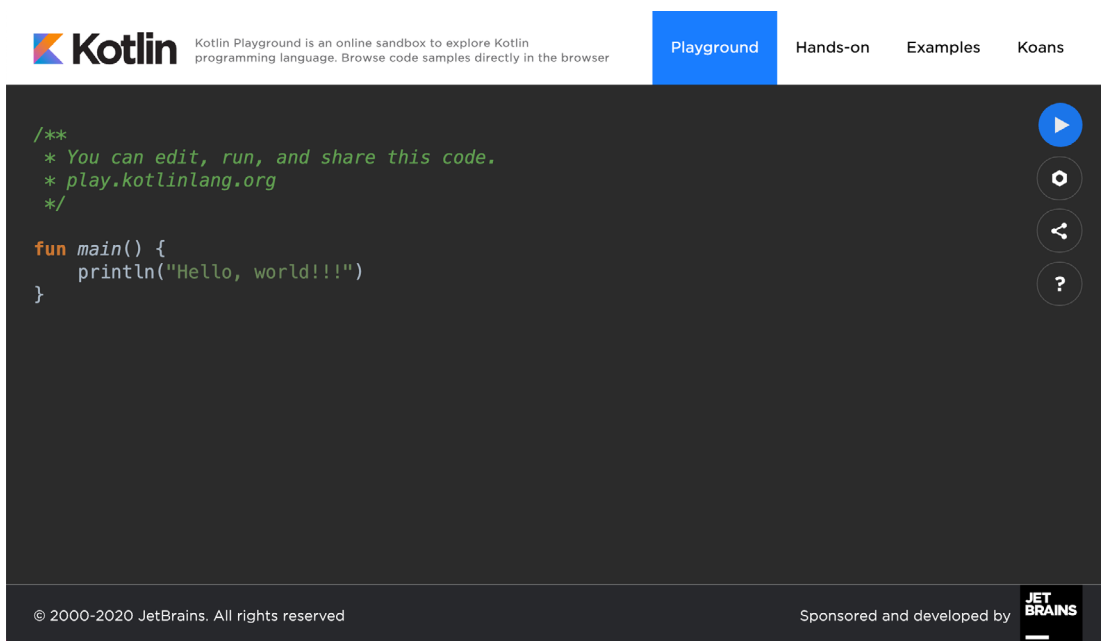


Figure 11-1

In addition to providing an environment in which Kotlin code may be quickly entered and executed, the playground also includes a set of examples and tutorials demonstrating key Kotlin features in action.

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

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

    println("Welcome to Kotlin")

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



```
}
```

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

```
Welcome to Kotlin
i = 1
i = 2
i = 3
i = 4
i = 5
i = 6
i = 7
i = 8
```

Figure 11-2

## 11.6 Semi-colons in Kotlin

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

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

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

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

## 11.7 Summary

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

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

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



## 12. Kotlin Data Types, Variables, and Nullability

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

## 22. Saving and Restoring the State of an Android Activity

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

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

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

### 22.1 Saving Dynamic State

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

### 22.2 Default Saving of User Interface State

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

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

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

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

## Saving and Restoring the State of an Android Activity

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

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

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

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

To configure the `EditText` view such that state will not be saved and restored if the activity is restarted, edit the `activity_main.xml` file so that the entry for the view reads as follows (note that the XML can be edited by switching the Layout Editor to Code view mode as outlined in “*Creating an Example Android App in Android Studio*”):

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

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

## 22.3 The Bundle Class

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

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

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

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

## 22.4 Saving the State

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

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

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

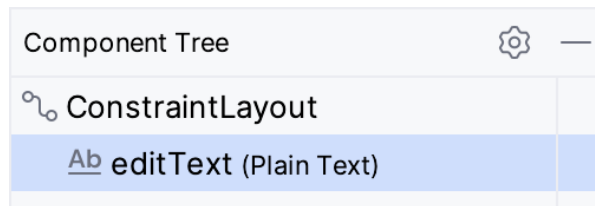


Figure 22-1

We can now obtain the text that the `editText` view contains via the object's `text` property, which, in turn, returns the current text:

```
val userText = binding.editText.text
```

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

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

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

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

    val userText = binding.editText.text
    outState.putCharSequence("savedText", userText)
```

```
}
```

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

## 22.5 Restoring the State

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

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

    val userText = savedInstanceState.getCharSequence("savedText")
    binding.editText.setText(userText)
}
```

## 22.6 Testing the Application

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

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

## 22.7 Summary

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

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

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



## 23. Understanding Android Views, View Groups and Layouts

With the possible exception of listening to streaming audio, a user's interaction with an Android device is primarily visual and tactile. All of this interaction occurs through the user interfaces of the applications installed on the device, including both the built-in applications and any third-party applications installed by the user. Therefore, it should come as no surprise that a critical element of developing Android applications involves designing and creating user interfaces.

This chapter covers the Android user interface structure, including an overview of the elements that can be combined to make up a user interface: Views, View Groups, and Layouts.

### 23.1 Designing for Different Android Devices

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

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

### 23.2 Views and View Groups

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

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

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

### 23.3 Android Layout Managers

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

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

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

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

## 23.4 The View Hierarchy

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

A user interface screen is comprised of a view hierarchy with a *root view* positioned at the top of the tree and child views positioned on branches below. The child of a container view appears on top of its parent view and is constrained to appear within the bounds of the parent view's display area. Consider, for example, the user interface illustrated in Figure 23-1:

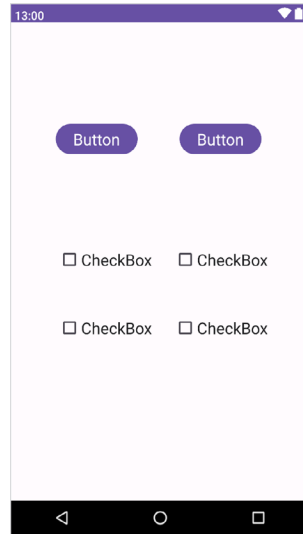


Figure 23-1

In addition to the visible button and checkbox views, the user interface actually includes a number of layout views that control how the visible views are positioned. Figure 23-2 shows an alternative view of the user interface, this time highlighting the presence of the layout views in relation to the child views:

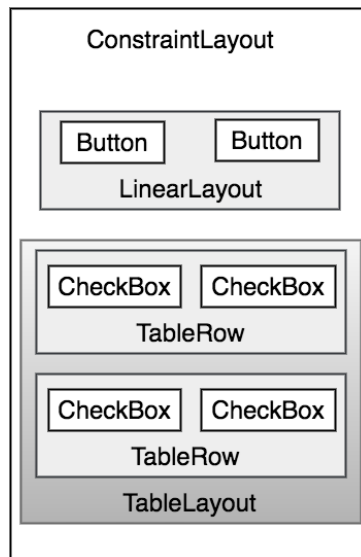


Figure 23-2

As was previously discussed, user interfaces are constructed in the form of a view hierarchy with a root view at the top. This being the case, we can also visualize the above user interface example in the form of the view tree illustrated in Figure 23-3:

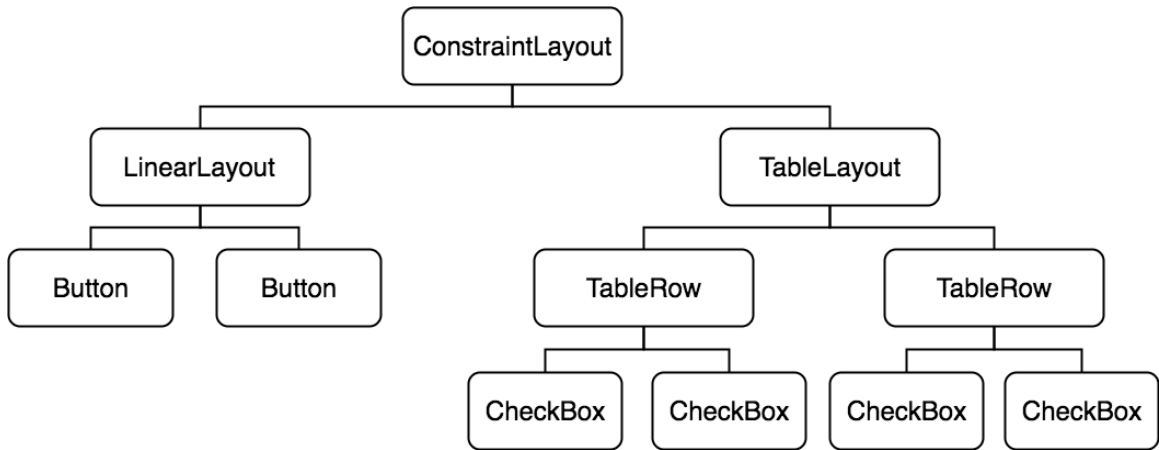


Figure 23-3

The view hierarchy diagram gives probably the clearest overview of the relationship between the various views that make up the user interface shown in Figure 23-1. When a user interface is displayed to the user, the Android runtime walks the view hierarchy, starting at the root view and working down the tree as it renders each view.

## 23.5 Creating User Interfaces

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

## 23.6 Summary

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

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

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

## 25. A Guide to the Android ConstraintLayout

As discussed in the chapter entitled “*Understanding Android Views, View Groups and Layouts*”, Android provides several layout managers to design user interfaces. With Android 7, Google introduced a layout that addressed many of the shortcomings of the older layout managers. This layout, called ConstraintLayout, combines a simple, expressive, and flexible layout system with powerful features built into the Android Studio Layout Editor tool to ease the creation of responsive user interface layouts that adapt automatically to different screen sizes and changes in device orientation.

This chapter will outline the basic concepts of ConstraintLayout, while the next chapter will provide a detailed overview of how constraint-based layouts can be created using ConstraintLayout within the Android Studio Layout Editor tool.

### 25.1 How ConstraintLayout Works

In common with all other layouts, ConstraintLayout manages the positioning and sizing behavior of the visual components (also referred to as widgets) it contains. It does this based on the constraint connections set on each child widget.

To fully understand and use ConstraintLayout, it is essential to gain an appreciation of the following key concepts:

- Constraints
- Margins
- Opposing Constraints
- Constraint Bias
- Chains
- Chain Styles
- Guidelines
- Groups
- Barriers
- Flow

#### 25.1.1 Constraints

Constraints are sets of rules that dictate how a widget is aligned and distanced relative to other widgets, the sides of the containing ConstraintLayout, and special elements called *guidelines*. Constraints also dictate how the user interface layout of an activity will respond to changes in device orientation or when displayed on devices of differing screen sizes. To be adequately configured, a widget must have sufficient constraint connections such that its position can be resolved by the ConstraintLayout layout engine in both the horizontal and vertical

planes.

### 25.1.2 Margins

A margin is a form of constraint that specifies a fixed distance. Consider a Button object that needs to be positioned near the top right-hand corner of the device screen. This might be achieved by implementing margin constraints from the top and right-hand edges of the Button connected to the corresponding sides of the parent ConstraintLayout, as illustrated in Figure 25-1:

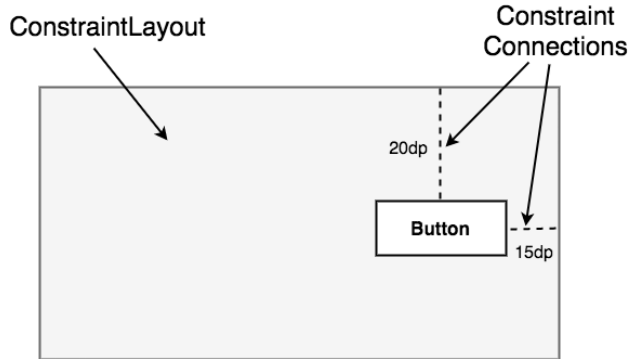


Figure 25-1

As indicated in the above diagram, each of these constraint connections has associated with it a margin value dictating the fixed distances of the widget from two sides of the parent layout. Under this configuration, regardless of screen size or the device orientation, the Button object will always be positioned 20 and 15 device-independent pixels (dp) from the top and right-hand edges of the parent ConstraintLayout, respectively, as specified by the two constraint connections.

While the above configuration will be acceptable for some situations, it does not provide any flexibility in terms of allowing the ConstraintLayout layout engine to adapt the position of the widget to respond to device rotation and to support screens of different sizes. To add this responsiveness to the layout, it is necessary to implement opposing constraints.

### 25.1.3 Opposing Constraints

Two constraints operating along the same axis on a single widget are considered *opposing constraints*. In other words, a widget with constraints on both its left and right-hand sides is considered to have horizontally opposing constraints. Figure 25-2, for example, illustrates the addition of both horizontally and vertically opposing constraints to the previous layout:

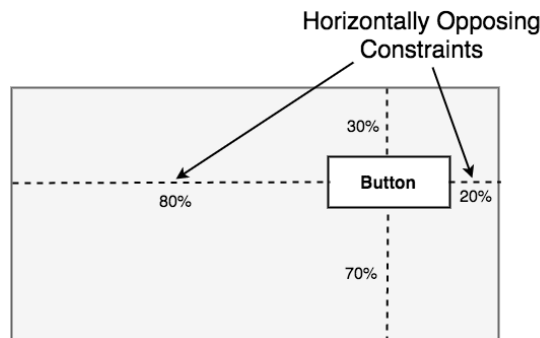


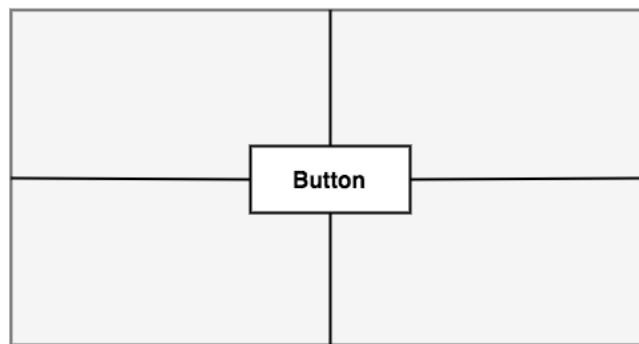
Figure 25-2

The key point to understand here is that once opposing constraints are implemented on a particular axis, the positioning of the widget becomes percentage rather than coordinate-based. Instead of being fixed at 20dp from the top of the layout, for example, the widget is now positioned at 30% from the top. In different orientations and when running on larger or smaller screens, the Button will always be in the same location relative to the dimensions of the parent layout.

It is now important to understand that the layout outlined in Figure 25-2 has been implemented using not only opposing constraints, but also by applying *constraint bias*.

#### 25.1.4 Constraint Bias

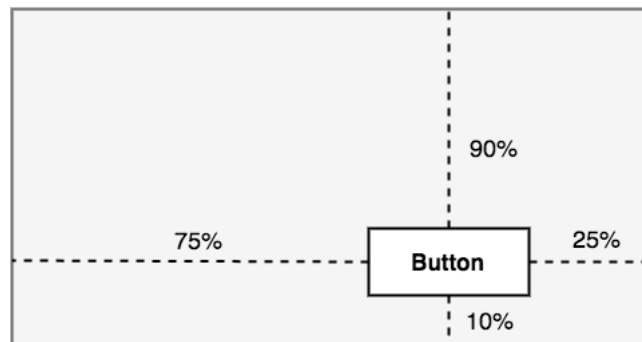
It has now been established that a widget in a ConstraintLayout can potentially be subject to opposing constraint connections. By default, opposing constraints are equal, resulting in the corresponding widget being centered along the axis of opposition. Figure 25-3, for example, shows a widget centered within the containing ConstraintLayout using opposing horizontal and vertical constraints:



**Widget Centered by Opposing Constraints**

Figure 25-3

To allow for the adjustment of widget position in the case of opposing constraints, the ConstraintLayout implements a feature known as *constraint bias*. Constraint bias allows the positioning of a widget along the axis of opposition to be biased by a specified percentage in favor of one constraint. Figure 25-4, for example, shows the previous constraint layout with a 75% horizontal bias and 10% vertical bias:



**Widget Offset using Constraint Bias**

Figure 25-4

The next chapter, entitled “A Guide to Using ConstraintLayout in Android Studio”, will cover these concepts in greater detail and explain how these features have been integrated into the Android Studio Layout Editor tool.

In the meantime, however, a few more areas of the ConstraintLayout class need to be covered.

### 25.1.5 Chains

ConstraintLayout chains provide a way for the layout behavior of two or more widgets to be defined as a group. Chains can be declared in either the vertical or horizontal axis and configured to define how the widgets in the chain are spaced and sized.

Widgets are chained when connected by bi-directional constraints. Figure 25-5, for example, illustrates three widgets chained in this way:

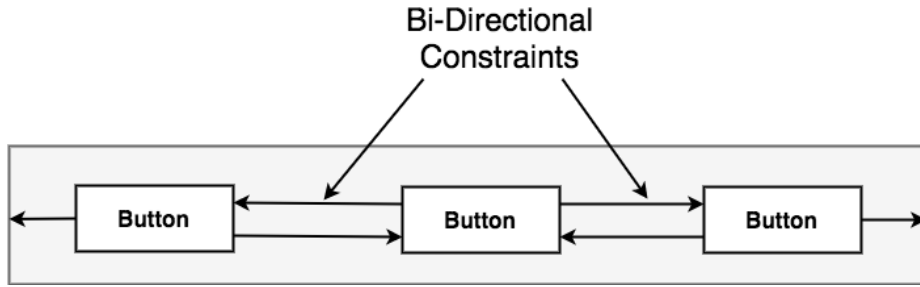


Figure 25-5

The first element in the chain is the *chain head* which translates to the top widget in a vertical chain or, in the case of a horizontal chain, the left-most widget. The layout behavior of the entire chain is primarily configured by setting attributes on the chain head widget.

### 25.1.6 Chain Styles

The layout behavior of a ConstraintLayout chain is dictated by the *chain style* setting applied to the chain head widget. The ConstraintLayout class currently supports the following chain layout styles:

- **Spread Chain** – The widgets within the chain are distributed evenly across the available space. This is the default behavior for chains.



Figure 25-6

- **Spread Inside Chain** – The widgets within the chain are spread evenly between the chain head and the last widget. The head and last widgets are not included in the distribution of spacing.



Figure 25-7

- **Weighted Chain** – Allows the space taken up by each widget in the chain to be defined via weighting properties.



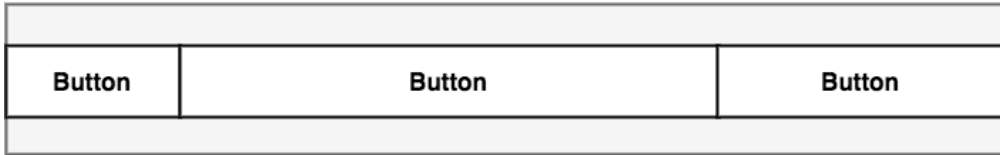


Figure 25-8

- **Packed Chain** – The widgets that make up the chain are packed together without spacing. A bias may be applied to control the horizontal or vertical positioning of the chain relative to the parent container.

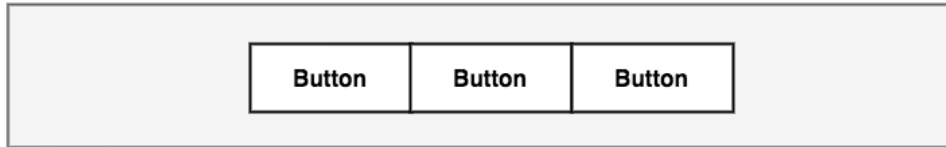


Figure 25-9

## 25.2 Baseline Alignment

So far, this chapter has only referred to constraints that dictate alignment relative to the sides of a widget (typically referred to as side constraints). A common requirement, however, is for a widget to be aligned relative to the content that it displays rather than the boundaries of the widget itself. To address this need, ConstraintLayout provides *baseline alignment* support.

For example, assume that the previous theoretical layout from Figure 25-1 requires a TextView widget to be positioned 40dp to the left of the Button. In this case, the TextView needs to be *baseline aligned* with the Button view. This means that the text within the Button needs to be vertically aligned with the text within the TextView. The additional constraints for this layout would need to be connected as illustrated in Figure 25-10:

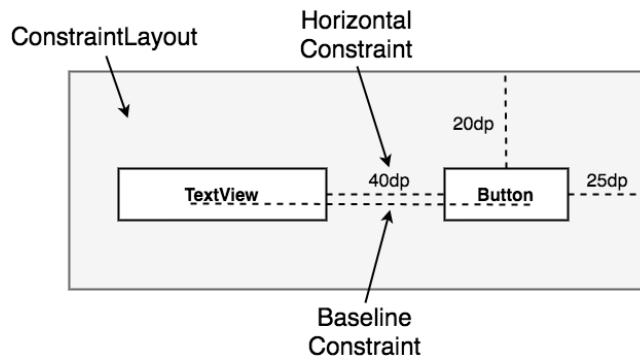


Figure 25-10

The TextView is now aligned vertically along the baseline of the Button and positioned 40dp horizontally from the Button object's left-hand edge.

## 25.3 Configuring Widget Dimensions

Controlling the dimensions of a widget is a key element of the user interface design process. The ConstraintLayout provides three options that can be set on individual widgets to manage sizing behavior. These settings are configured individually for height and width dimensions:

- **Fixed** – The widget is fixed to specified dimensions.
- **Match Constraint** – Allows the widget to be resized by the layout engine to satisfy the prevailing constraints.

Also referred to as the *AnySize* or `MATCH_CONSTRAINT` option.

- **Wrap Content** – The widget’s size is dictated by its content (i.e., text or graphics).

## 25.4 Guideline Helper

Guidelines are special elements available within the `ConstraintLayout` that provide an additional target to which constraints may be connected. Multiple guidelines may be added to a `ConstraintLayout` instance which may, in turn, be configured in horizontal or vertical orientations. Once added, constraint connections may be established from widgets in the layout to the guidelines. This is particularly useful when multiple widgets must be aligned along an axis. In Figure 25-11, for example, three `Button` objects contained within a `ConstraintLayout` are constrained along a vertical guideline:

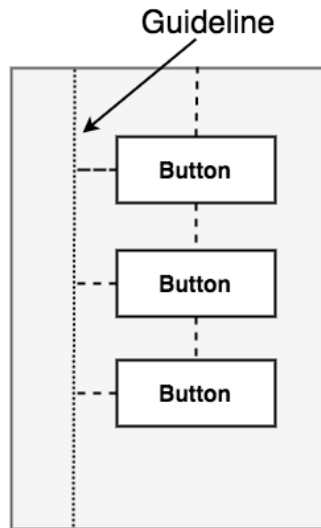


Figure 25-11

## 25.5 Group Helper

This feature of `ConstraintLayout` allows widgets to be placed into logical groups, and the visibility of those widgets controlled as a single entity. A `Group` is a list of references to other widgets in a layout. Once defined, changing the visibility attribute (`visible`, `invisible`, or `gone`) of the group instance will apply the change to all group members. This makes hiding and showing multiple widgets with a single attribute change easy. A single layout may contain multiple groups, and a widget can belong to more than one group. If a conflict occurs between groups, the last group to be declared in the XML file takes priority.

## 25.6 Barrier Helper

Rather like guidelines, barriers are virtual views that can be used to constrain views within a layout. As with guidelines, a barrier can be vertical or horizontal, and one or more views may be constrained to it (to avoid confusion, these will be referred to as *constrained views*). Unlike guidelines, where the guideline remains at a fixed position within the layout, however, the position of a barrier is defined by a set of so-called *reference views*. Barriers were introduced to address an issue that occurs with some frequency involving overlapping views. Consider, for example, the layout illustrated in Figure 25-12 below:

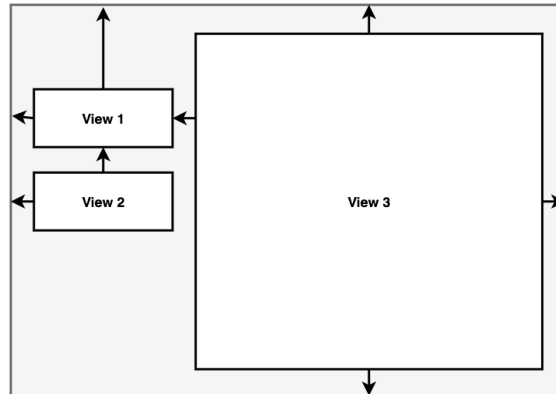


Figure 25-12

The key points to note about the above layout are that the width of View 3 is set to match constraint mode, and the left-hand edge of the view is connected to the right-hand edge of View 1. As currently implemented, an increase in width of View 1 will have the desired effect of reducing the width of View 3:

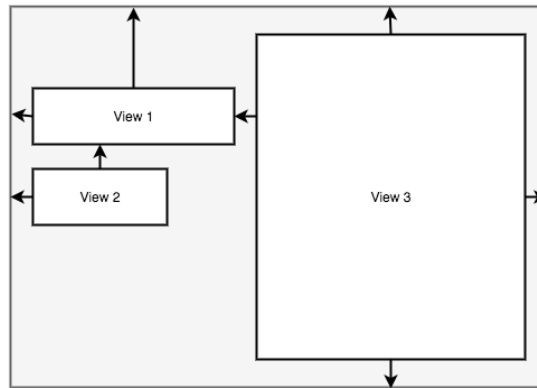


Figure 25-13

A problem arises, however, if View 2 increases in width instead of View 1:

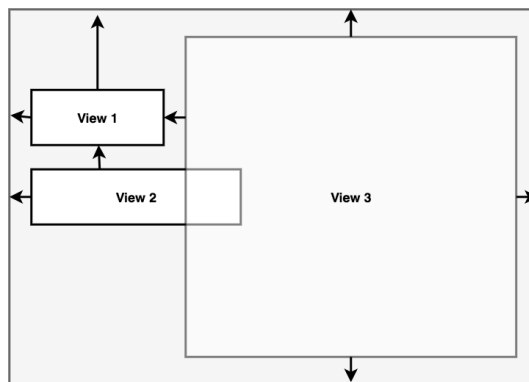


Figure 25-14

Because View 3 is only constrained by View 1, it does not resize to accommodate the increase in width of View

2, causing the views to overlap.

A solution to this problem is to add a vertical barrier and assign Views 1 and 2 as the barrier's *reference views* so that they control the barrier position. The left-hand edge of View 3 will then be constrained relative to the barrier, making it a *constrained view*.

Now when either View 1 or View 2 increases in width, the barrier will move to accommodate the widest of the two views, causing the width of View 3 to change relative to the new barrier position:

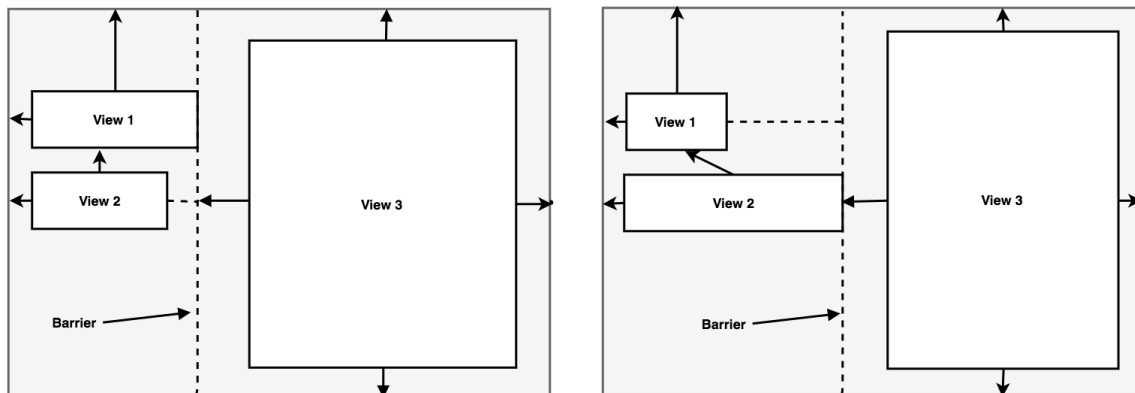


Figure 25-15

When working with barriers, there is no limit to the number of reference and constrained views that can be associated with a single barrier.

## 25.7 Flow Helper

The ConstraintLayout Flow helper allows groups of views to be displayed in a flowing grid-style layout. As with the Group helper, Flow contains references to the views it is responsible for positioning and provides various configuration options, including vertical and horizontal orientations, wrapping behavior (including the maximum number of widgets before wrapping), spacing, and alignment properties. Chain behavior may also be applied to a Flow layout, including spread, spread inside, and packed options.

Figure 25-16 represents the layout of five uniformly sized buttons positioned using a Flow helper instance in horizontal mode with no wrap settings:



Figure 25-16

Figure 25-17 shows the same buttons in a horizontal flow configuration with wrapping set to occur after every third widget:

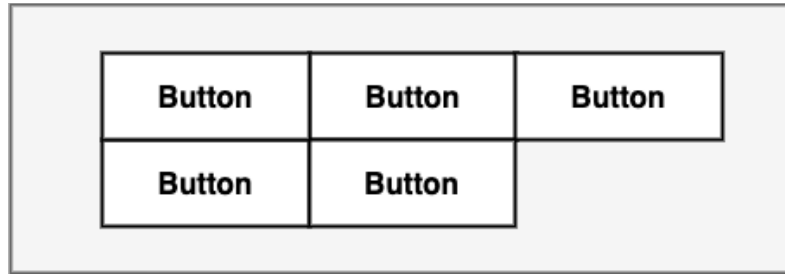


Figure 25-17

Figure 25-18, on the other hand, shows the buttons with wrapping set to chain mode using spread inside (the effects of which are only visible on the second row since the first row is full). The configuration also has the gap attribute set to add spacing between buttons:

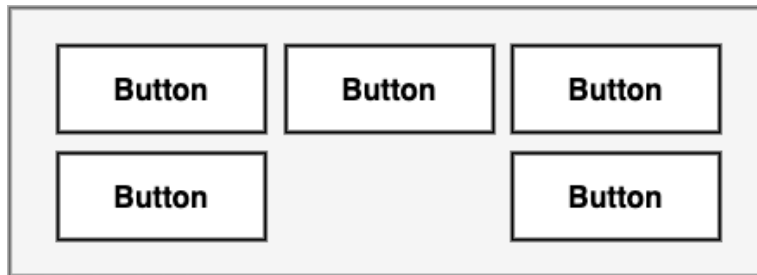


Figure 25-18

As a final demonstration of the flexibility of the Flow helper, Figure 25-19 shows five buttons of varying sizes configured in horizontal, packed chain mode with wrapping after each third widget. In addition, the grid content has been right-aligned by setting a horizontal-bias value of 1.0 (a value of 0.0 would cause left-alignment while 0.5 would center-align the grid content):

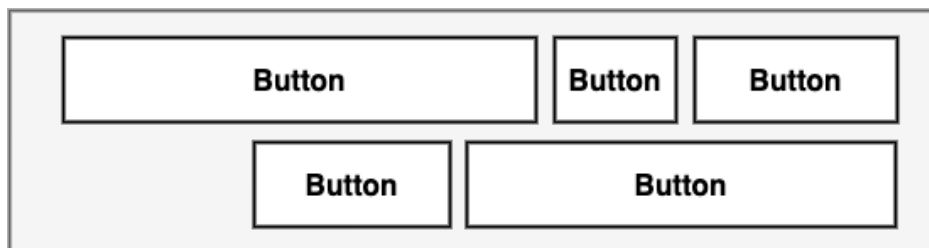


Figure 25-19

## 25.8 Ratios

The dimensions of a widget may be defined using ratio settings. A widget could, for example, be constrained using a ratio setting such that, regardless of any resizing behavior, the width is always twice the height dimension.

## 25.9 ConstraintLayout Advantages

ConstraintLayout provides a level of flexibility that allows many of the features of older layouts to be achieved with a single layout instance where it would previously have been necessary to nest multiple layouts. This can avoid the problems inherent in layout nesting by allowing so-called “flat” or “shallow” layout hierarchies to be designed, leading both to less complex layouts and improved user interface rendering performance at runtime.

ConstraintLayout was also implemented to address the wide range of Android device screen sizes available

today. The flexibility of ConstraintLayout makes it easier for user interfaces to be designed that respond and adapt to the device on which the app is running.

Finally, as will be demonstrated in the chapter entitled “*A Guide to Using ConstraintLayout in Android Studio*”, the Android Studio Layout Editor tool has been enhanced specifically for ConstraintLayout-based user interface design.

### 25.10 ConstraintLayout Availability

Although introduced with Android 7, ConstraintLayout is provided as a separate support library from the main Android SDK and is compatible with older Android versions as far back as API Level 9 (Gingerbread). This allows apps that use this layout to run on devices running much older versions of Android.

### 25.11 Summary

ConstraintLayout is a layout manager introduced with Android 7. It is designed to ease the creation of flexible layouts that adapt to the size and orientation of the many Android devices on the market. ConstraintLayout uses constraints to control the alignment and positioning of widgets relative to the parent ConstraintLayout instance, guidelines, barriers, and the other widgets in the layout. ConstraintLayout is the default layout for newly created Android Studio projects and is recommended when designing user interface layouts. This simple yet flexible approach to layout management allows complex and responsive user interfaces to be easily implemented.

# 26. A Guide to Using ConstraintLayout in Android Studio

As mentioned more than once in previous chapters, Google has made significant changes to the Android Studio Layout Editor tool, many of which were made solely to support user interface layout design using ConstraintLayout. Now that the basic concepts of ConstraintLayout have been outlined in the previous chapter, this chapter will explore these concepts in more detail while also outlining how the Layout Editor tool allows ConstraintLayout-based user interfaces to be designed and implemented.

## 26.1 Design and Layout Views

The chapter entitled “*A Guide to the Android Studio Layout Editor Tool*” explained that the Android Studio Layout Editor tool provides two ways to view the user interface layout of an activity in the form of Design and Layout (also known as blueprint) views. These views of the layout may be displayed individually or, as in Figure 26-1, side-by-side:

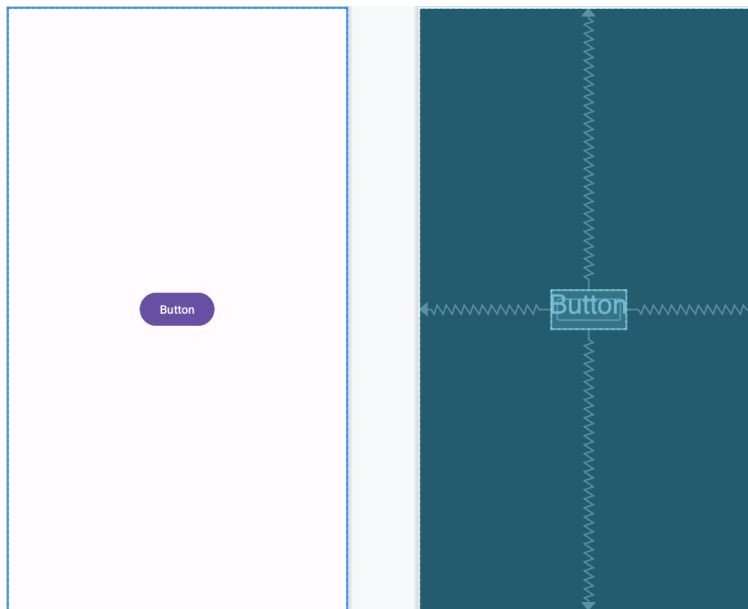


Figure 26-1

The Design view (positioned on the left in the above figure) presents a “what you see is what you get” representation of the layout, wherein the layout appears as it will within the running app. On the other hand, the Layout view displays a blueprint style of view where shaded outlines represent the widgets. As shown in Figure 26-1 above, the Layout view also displays the constraint connections (in this case, opposing constraints used to center a button within the layout). These constraints are also overlaid onto the Design view when a specific widget in the layout is selected or when the mouse pointer hovers over the design area, as illustrated in Figure 26-2:

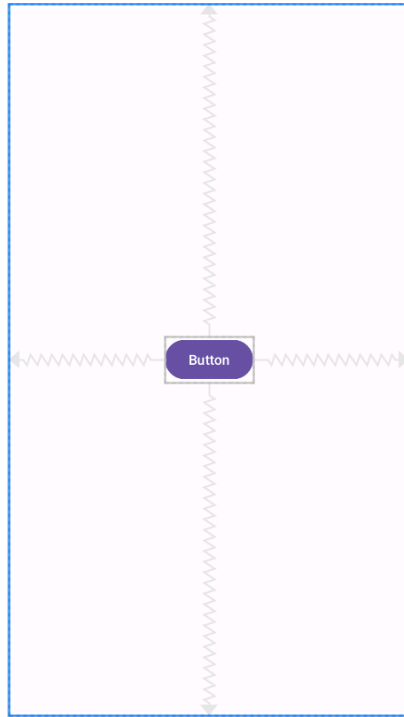


Figure 26-2

The appearance of constraint connections in both views can be changed using the View Options menu shown in Figure 26-3:

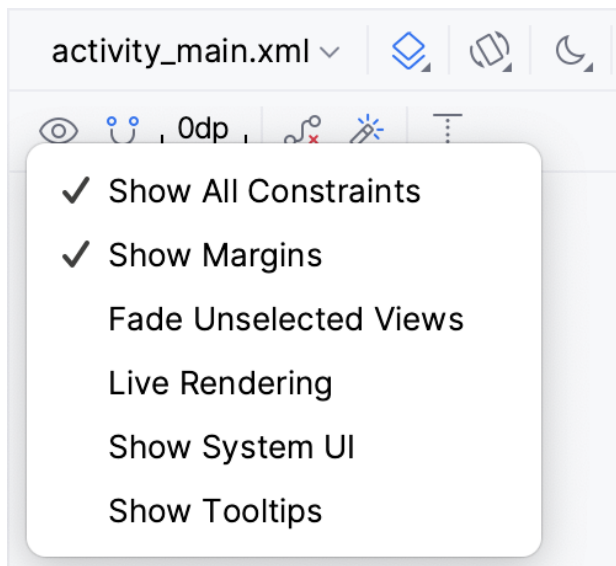


Figure 26-3

In addition to the two modes of displaying the user interface layout, the Layout Editor tool provides three ways of establishing the constraints required for a specific layout design.



## 26.2 Autoconnect Mode

Autoconnect, as the name suggests, automatically establishes constraint connections as items are added to the layout. Autoconnect mode may be turned on and off using the toolbar button indicated in Figure 26-4:

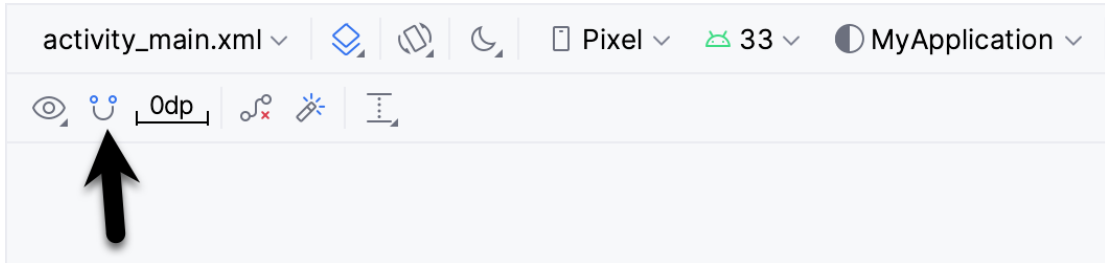


Figure 26-4

Autoconnect mode uses algorithms to decide the best constraints to establish based on the widget’s position and the widget’s proximity to both the sides of the parent layout and other elements. If any of the automatic constraint connections fail to provide the desired behavior, these may be changed manually, as outlined later in this chapter.

## 26.3 Inference Mode

Inference mode uses a heuristic approach involving algorithms and probabilities to automatically implement constraint connections after widgets have already been added to the layout. This mode is usually used when the Autoconnect feature has been turned off, and objects have been added to the layout without any constraint connections. This allows the layout to be designed by dragging and dropping objects from the palette onto the layout canvas and making size and positioning changes until the layout appears as required. Essentially, this involves “painting” the layout without worrying about constraints. Inference mode may also be used during the design process to fill in missing constraints within a layout.

Constraints are automatically added to a layout when the *Infer constraints* button (Figure 26-5) is clicked:

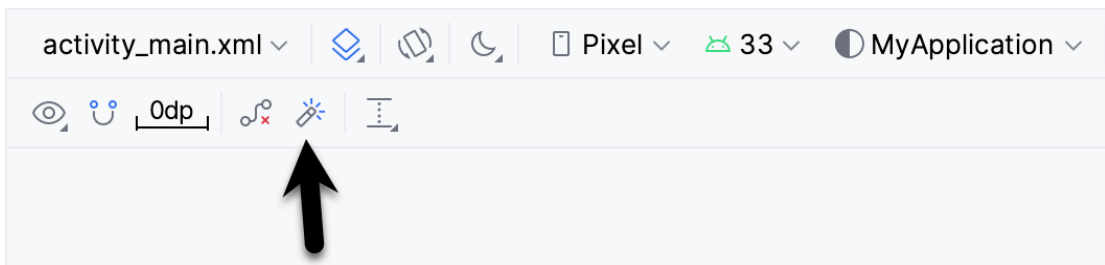


Figure 26-5

As with Autoconnect mode, there is always the possibility that the Layout Editor tool will infer incorrect constraints, though these may be modified and corrected manually.

## 26.4 Manipulating Constraints Manually

The third option for implementing constraint connections is to do so manually. When doing so, it will be helpful to understand the various handles that appear around a widget within the Layout Editor tool. Consider, for example, the widget shown in Figure 26-6:

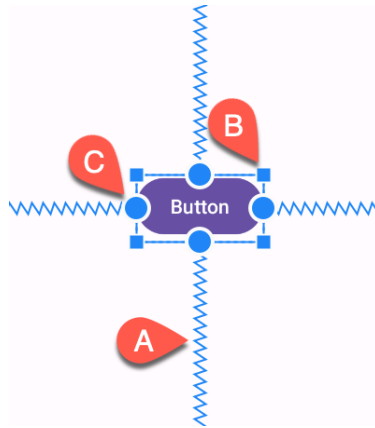


Figure 26-6

The spring-like lines (A) represent established constraint connections leading from the sides of the widget to the targets. The small square markers (B) in each corner of the object are resizing handles which, when clicked and dragged, serve to resize the widget. The small circle handles (C) located on each side of the widget are the side constraint anchors. To create a constraint connection, click on the handle and drag the resulting line to the element to which the constraint is to be connected (such as a guideline or the side of either the parent layout or another widget), as outlined in Figure 26-7. When connecting to the side of another widget, drag the line to the side constraint handle of that widget and release the line when the widget and handle are highlighted:

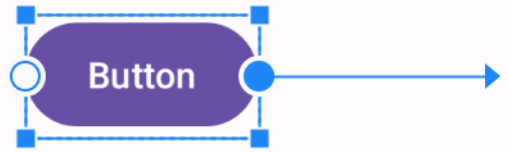


Figure 26-7

If the constraint line is dragged to a widget and released but not attached to a constraint handle, the layout editor will display a menu containing a list of the sides to which the constraint may be attached. In Figure 26-8, for example, the constraint can be attached to the top or bottom edge of the destination button widget:

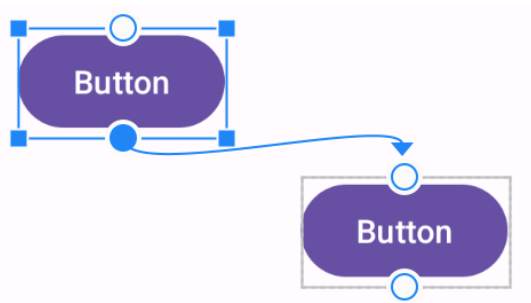


Figure 26-8

An additional marker indicates the anchor point for baseline constraints whereby the content within the widget (as opposed to outside edges) is used as the alignment point. To display this marker, right-click on the widget and select the *Show Baseline* menu option. To establish a constraint connection from a baseline constraint handle, hover the mouse pointer over the handle until it highlights before clicking and dragging to the target (such as the baseline anchor of another widget, as shown in Figure 26-9).

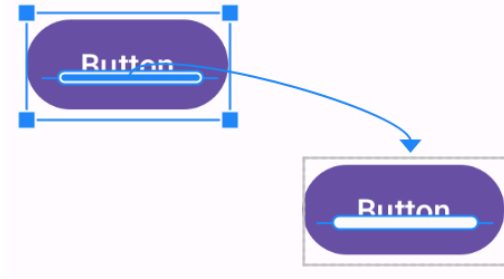


Figure 26-9

To hide the baseline anchors, right-click on the widget again and select the *Hide Baseline* menu option.

## 26.5 Adding Constraints in the Inspector

Constraints may also be added to a view within the Android Studio Layout Editor tool using the *Inspector* panel in the Attributes tool window, as shown in Figure 26-10. The square in the center represents the currently selected view, and the areas around the square the constraints, if any, applied to the corresponding sides of the view:

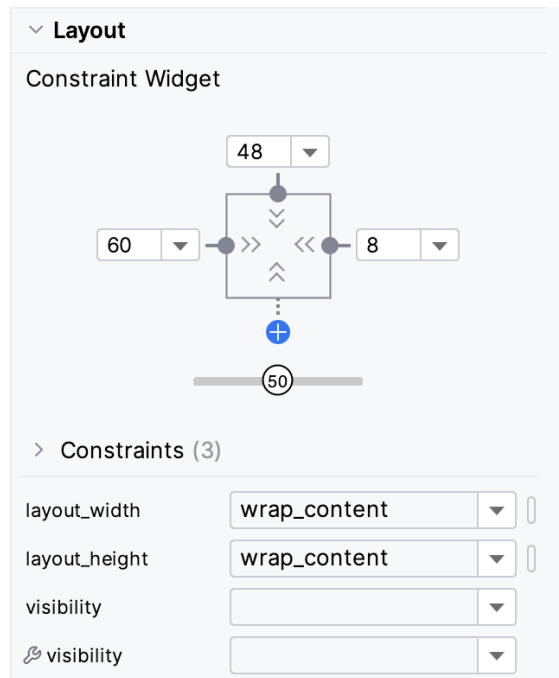


Figure 26-10

The absence of a constraint on the side of the view is represented by a dotted line leading to a blue circle containing a plus sign (as is the case with the view's bottom edge in the above figure). To add a constraint, click on this blue circle, and the layout editor will add a constraint connected to what it considers the most appropriate target within the layout.

## 26.6 Viewing Constraints in the Attributes Window

A list of constraints configured on the currently selected widget can be viewed by displaying the Constraints section of the Attributes tool window, as shown in Figure 26-11 below:

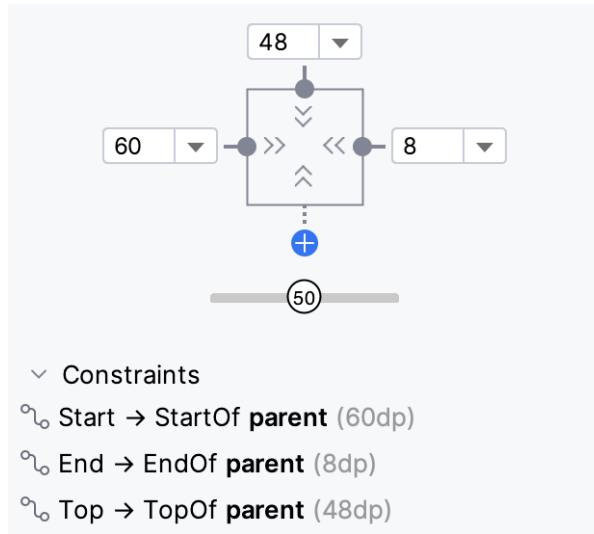


Figure 26-11

Clicking on a constraint in the list will select that constraint within the design layout.

## 26.7 Deleting Constraints

To delete an individual constraint, select the constraint either within the design layout or the Attributes tool window so that it highlights (in Figure 26-12, for example, the right-most constraint has been selected) and tap the keyboard delete key. The constraint will then be removed from the layout.

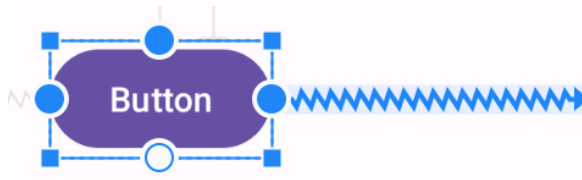


Figure 26-12

Another option is to hover the mouse pointer over the constraint anchor while holding down the Ctrl (Cmd on macOS) key and clicking on the anchor after it turns red:

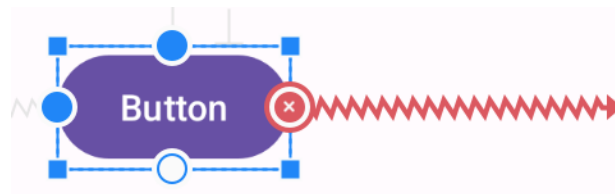


Figure 26-13

Alternatively, remove all of the constraints on a widget by right-clicking on it and selecting the *Clear Constraints of Selection* menu option.

To remove all of the constraints from every widget in a layout, use the toolbar button highlighted in Figure 26-14:

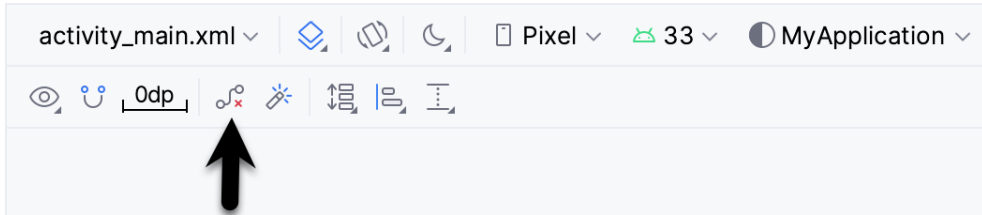


Figure 26-14

## 26.8 Adjusting Constraint Bias

The previous chapter outlined the concept of using bias settings to favor one opposing constraint over another. Bias within the Android Studio Layout Editor tool is adjusted using the *Inspector* located in the Attributes tool window and shown in Figure 26-15. The two sliders indicated by the arrows in the figure are used to control the bias of the currently selected widget's vertical and horizontal opposing constraints.

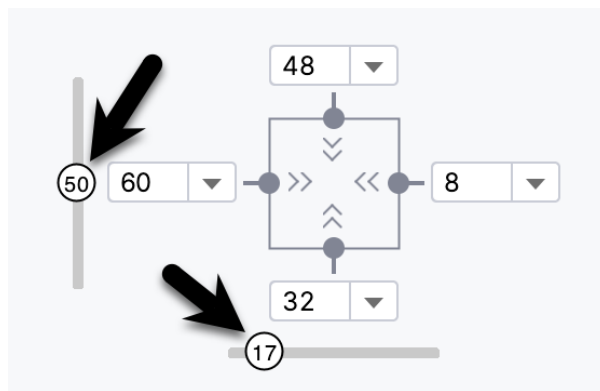


Figure 26-15

## 26.9 Understanding ConstraintLayout Margins

Constraints can be used with margins to implement fixed gaps between a widget and another element (such as another widget, a guideline, or the side of the parent layout). Consider, for example, the horizontal constraints applied to the Button object in Figure 26-16:

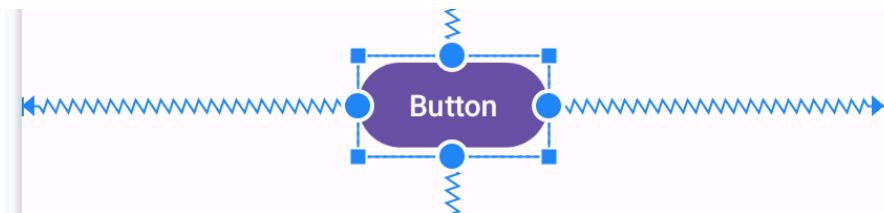


Figure 26-16

As currently configured, horizontal constraints run to the left and right edges of the parent ConstraintLayout. As such, the widget has opposing horizontal constraints indicating that the ConstraintLayout layout engine has some discretion in terms of the actual positioning of the widget at runtime. This allows the layout some flexibility to accommodate different screen sizes and device orientations. The horizontal bias setting can also control the widget's position right up to the right-hand side of the layout. Figure 26-17, for example, shows the same button with 100% horizontal bias applied:



Figure 26-17

ConstraintLayout margins can appear at the end of constraint connections and represent a fixed gap into which the widget cannot be moved, even when adjusting bias or responding to layout changes elsewhere in the activity. In Figure 26-18, the right-hand constraint now includes a 50dp margin into which the widget cannot be moved even though the bias is still set at 100%.



Figure 26-18

Existing margin values on a widget can be modified from within the Inspector. As shown in Figure 26-19, a drop-down menu is being used to change the right-hand margin on the currently selected widget to 16dp. Alternatively, clicking on the current value also allows a number to be typed into the field.

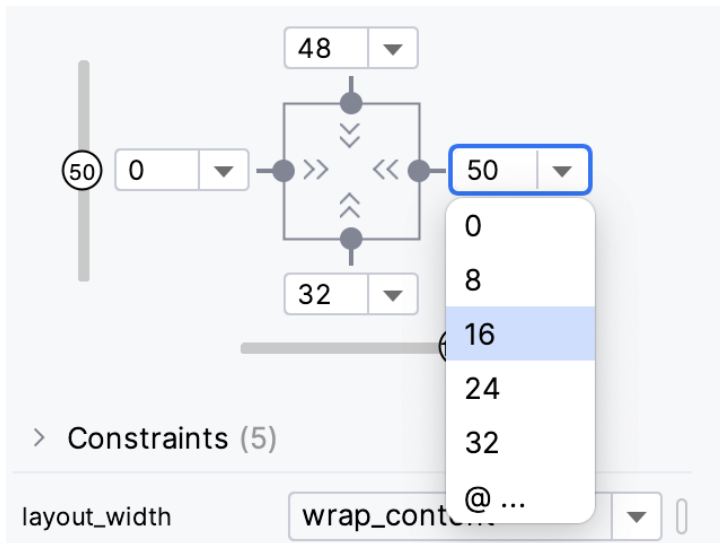


Figure 26-19

The default margin for new constraints can be changed at any time using the option in the toolbar highlighted in Figure 26-20:

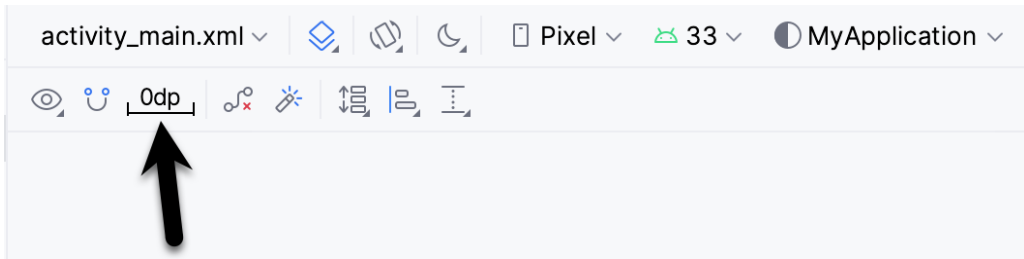


Figure 26-20

## 26.10 The Importance of Opposing Constraints and Bias

As discussed in the previous chapter, opposing constraints, margins, and bias form the cornerstone of responsive layout design in Android when using the ConstraintLayout. When a widget is constrained without opposing constraint connections, those constraints are essentially margin constraints. This is indicated visually within the Layout Editor tool by solid straight lines accompanied by margin measurements, as shown in Figure 26-21.

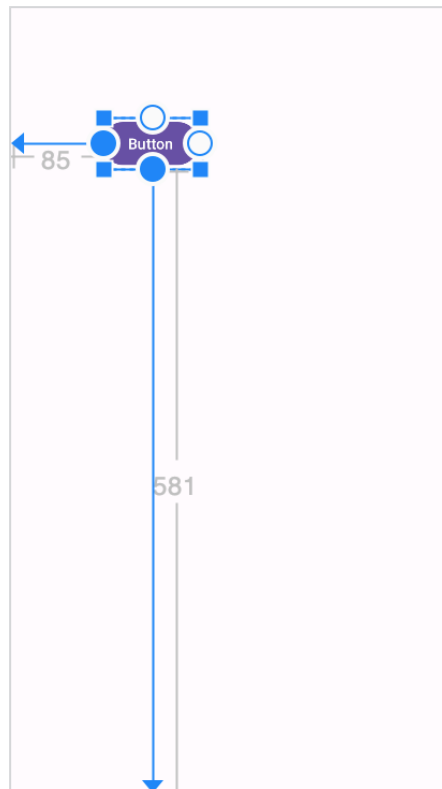


Figure 26-21

The above constraints fix the widget at that position. The result is that if the device is rotated to landscape orientation, the widget will no longer be visible since the vertical constraint pushes it beyond the top edge of the device screen (as is the case in Figure 26-22). A similar problem will arise if the app is run on a device with a smaller screen than that used during the design process.



Figure 26-22

When opposing constraints are implemented, the constraint connection is represented by the jagged spring-like line (the spring metaphor is intended to indicate that the position of the widget is not fixed to absolute X and Y coordinates):

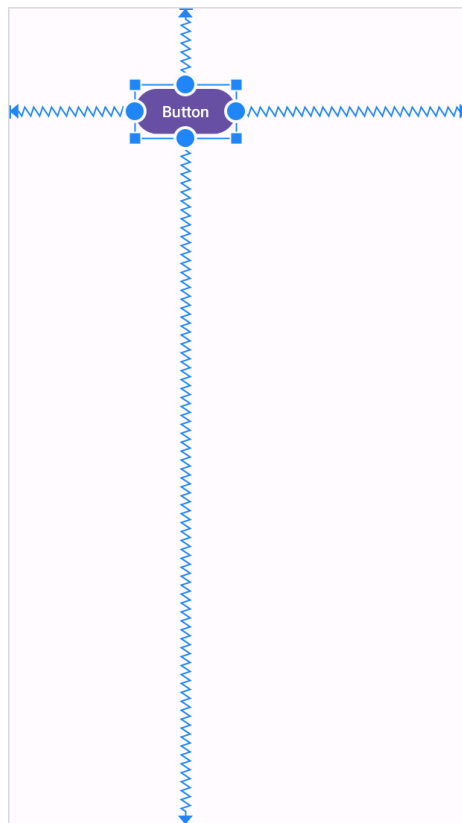


Figure 26-23

In the above layout, vertical and horizontal bias settings have been configured such that the widget will always be positioned 90% of the distance from the bottom and 35% from the left-hand edge of the parent layout. When rotated, therefore, the widget is still visible and positioned in the same location relative to the dimensions of the screen:



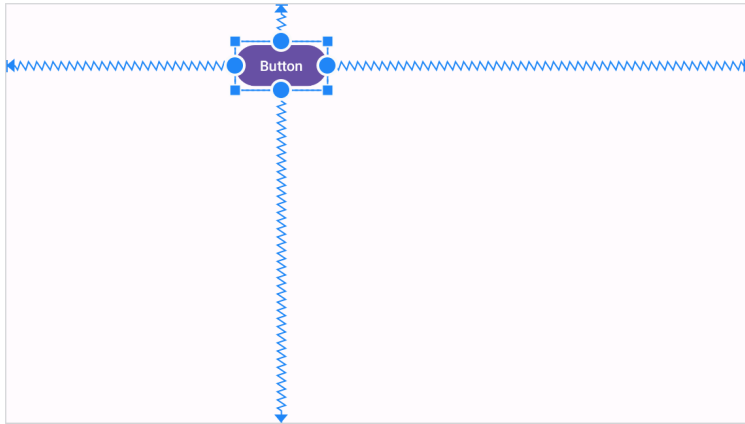


Figure 26-24

When designing a responsive and adaptable user interface layout, it is important to consider bias and opposing constraints when manually designing a user interface layout and correcting automatically created constraints.

## 26.11 Configuring Widget Dimensions

The inner dimensions of a widget within a ConstraintLayout can also be configured using the Inspector. As outlined in the previous chapter, widget dimensions can be set to wrap content, fixed, or match constraint modes. The prevailing settings for each dimension on the currently selected widget are shown within the square representing the widget in the Inspector, as illustrated in Figure 26-25:

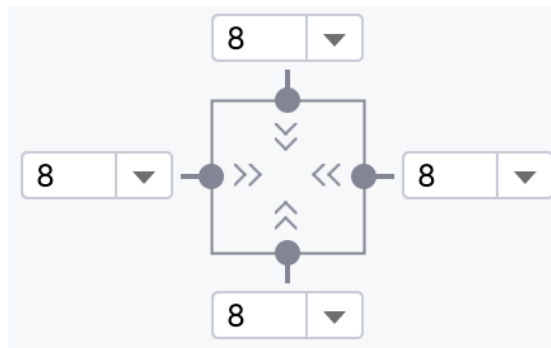


Figure 26-25

The above figure sets the horizontal and vertical dimensions to wrap content mode (indicated by the inward-pointing chevrons). The inspector uses the following visual indicators to represent the three dimension modes:

Fixed Size	
Match Constraint	
Wrap Content	

Table 26-1

To change the current setting, click on the indicator to cycle through the three settings.

In addition, a widget's size can be expanded horizontally or vertically to the maximum amount allowed by the constraints and other widgets in the layout using the Expand Horizontally and Expand Vertically options. These are accessible by right-clicking on a widget within the layout and selecting the Organize option from the resulting menu (Figure 26-26). When used, the currently selected widget will increase in size horizontally or vertically to fill the available space around it.

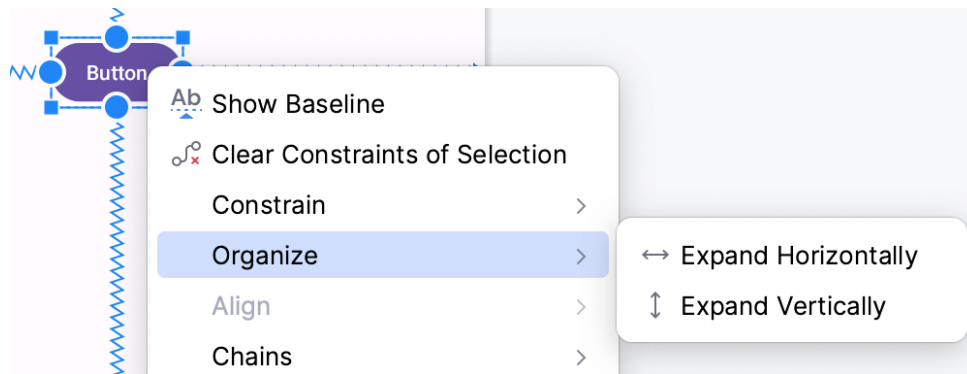


Figure 26-26

### 26.12 Design Time Tools Positioning

The chapter entitled “*A Guide to the Android Studio Layout Editor Tool*” introduced the concept of the *tools* namespace and explained how it can be used to set visibility attributes that only take effect within the layout editor. Behind the scenes, Android Studio also uses tools attributes to hold widgets in position when placed on the layout without constraints. Imagine, for example, a Button placed onto the layout while autoconnect mode is disabled. While the widget will appear to be in the correct position within the preview canvas, when the app is run, it will appear in the top left-hand corner of the screen. This is because the widget has no constraints to tell the ConstraintLayout parent where to position it.

The widget appears to be in the correct location in the layout editor because Android Studio has set absolute X and Y positioning tools attributes to keep it in the correct location until constraints can be added. Within the XML layout file, this might read as follows:

```
<Button
    android:id="@+id/button4"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Button"
    tools:layout_editor_absoluteX="111dp"
    tools:layout_editor_absoluteY="88dp" />
```

Once adequate constraints have been added to the widget, the layout editor will remove these tools attributes. A useful technique to quickly identify which widgets lack constraints without waiting until the app runs is to click on the button highlighted in Figure 26-27 to toggle tools position visibility. Any widgets that jump to the top left-hand corner are not fully constrained and are being held in place by temporary tools absolute X and Y positioning attributes.

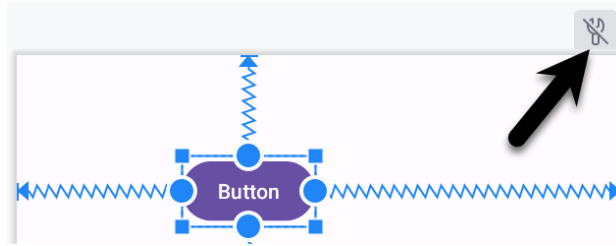


Figure 26-27

### 26.13 Adding Guidelines

Guidelines provide additional elements to which constraints may be anchored. Guidelines are added by right-clicking on the layout and selecting either the *Vertical Guideline* or *Horizontal Guideline* menu option or using the toolbar menu options as shown in Figure 26-28:

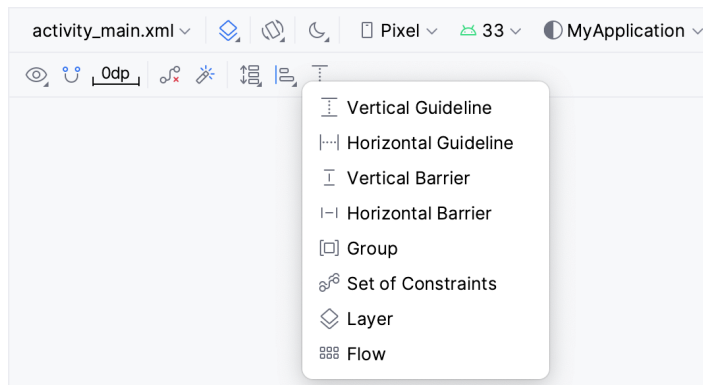


Figure 26-28

Alternatively, horizontal and vertical Guidelines may be dragged from the Helpers section of the Palette and dropped either onto the layout canvas or Component Tree panel as indicated by the arrows in Figure 26-29:

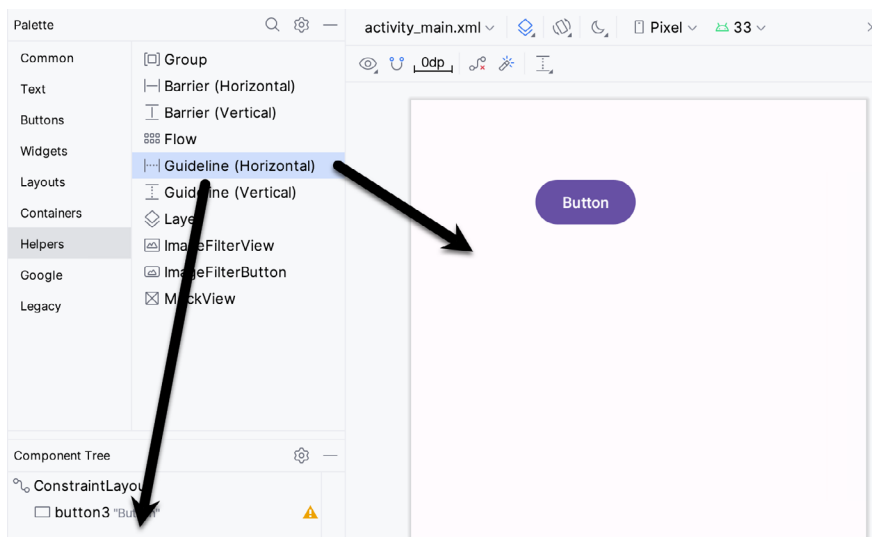


Figure 26-29

## A Guide to Using ConstraintLayout in Android Studio

Once added, a guideline will appear as a dashed line in the layout and may be moved by clicking and dragging the line. To establish a constraint connection to a guideline, click on the constraint handler of a widget and drag it to the guideline before releasing. In Figure 26-30, the left sides of two Buttons are connected by constraints to a vertical guideline.

The position of a vertical guideline can be specified as an absolute distance from either the left or the right of the parent layout (or the top or bottom for a horizontal guideline). For example, the vertical guideline in the figure below is positioned at 97dp from the left-hand edge of the parent:

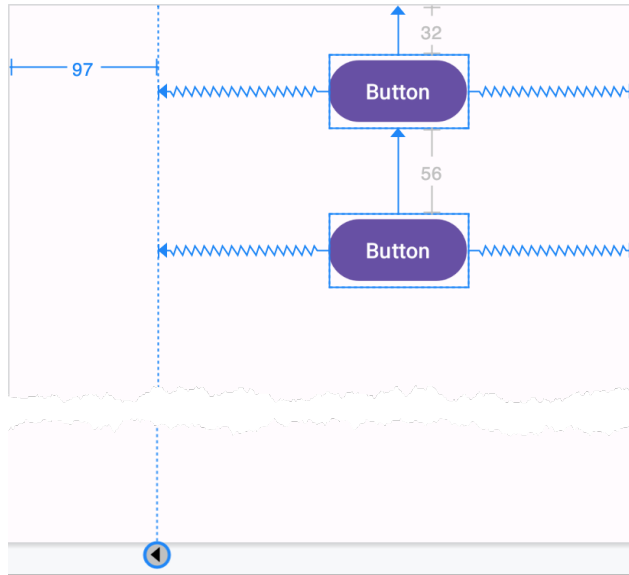


Figure 26-30

Alternatively, the guideline may be positioned as a percentage of the overall width or height of the parent layout. To switch between these three modes, select the guideline and click on the circle at the bottom or end of the guideline (depending on whether the guideline is vertical or horizontal). Figure 26-31, for example, shows a guideline positioned based on percentage:

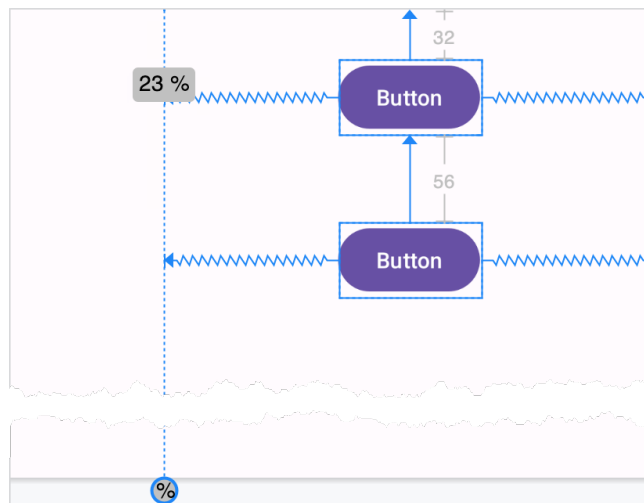


Figure 26-31

## 26.14 Adding Barriers

Barriers are added by right-clicking on the layout and selecting either the *Vertical* or *Horizontal Barrier* option from the *Add helpers* menu or using the toolbar menu options, as shown in Figure 26-28. Alternatively, locate the Barrier types in the Helpers section of the Palette and drag and drop them either onto the layout canvas or the Component Tree panel.

Once a barrier has been added to the layout, it will appear as an entry in the Component Tree panel:

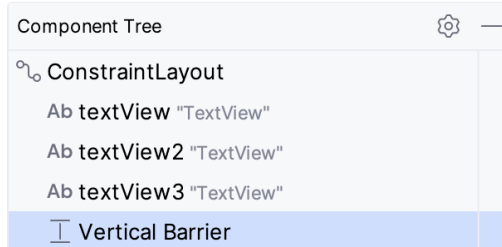


Figure 26-32

To add views as reference views (in other words, the views that control the position of the barrier), drag the widgets from within the Component Tree onto the barrier entry. In Figure 26-33, for example, widgets named textView2 and textView3 have been assigned as the reference widgets for the barrier:

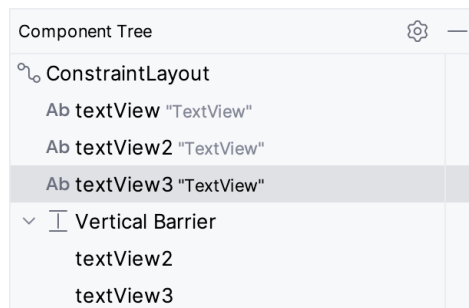


Figure 26-33

After the reference views have been added, the barrier needs to be configured to specify the direction of the barrier relative to those views. This is the *barrier direction* setting and is defined within the Attributes tool window when the barrier is selected in the Component Tree panel:

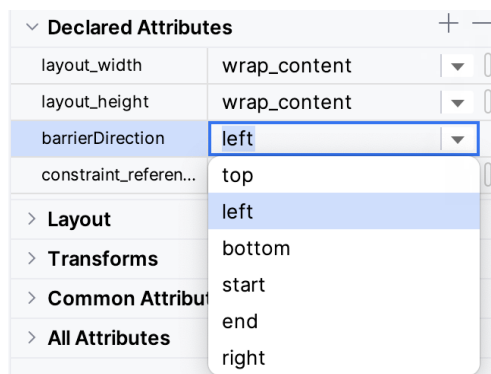


Figure 26-34

The following figure shows a layout containing a barrier declared with `textView1` and `textView2` acting as the reference views and `textView3` as the constrained view. Since the barrier is pushing from the end of the reference views towards the constrained view, the barrier direction has been set to *end*:

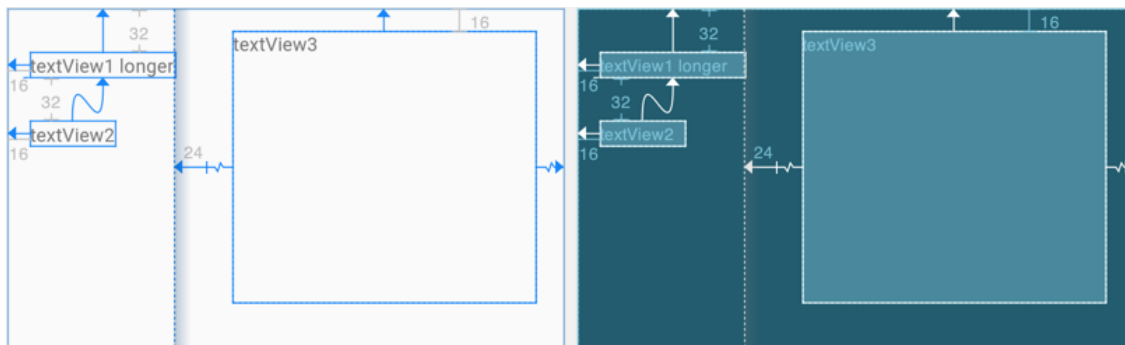


Figure 26-35

## 26.15 Adding a Group

To add a Group to a layout, right-click on the layout and select the *Group* option from the Add *helpers* menu or use the toolbar menu options shown in Figure 26-28. Alternatively, locate the Group item in the Helpers section of the Palette and drag and drop it either onto the layout canvas or Component Tree panel.

To add widgets to the group, select them in the Component Tree and drag and drop them onto the Group entry. Figure 26-36, for example, shows three selected widgets being added to a group:

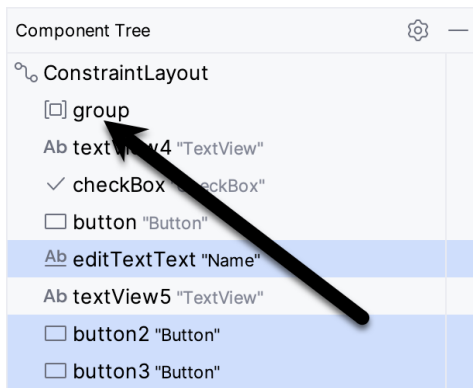


Figure 26-36

Any widgets referenced by the group will appear italicized beneath the group entry in the Component Tree, as shown in Figure 26-37. To remove a widget from the group, select it and tap the keyboard delete key:

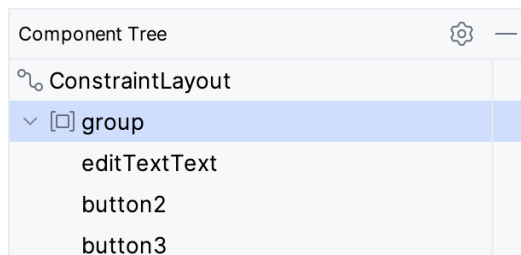


Figure 26-37

Once widgets have been assigned to the group, use the Constraints section of the Attributes tool window to modify the visibility setting:

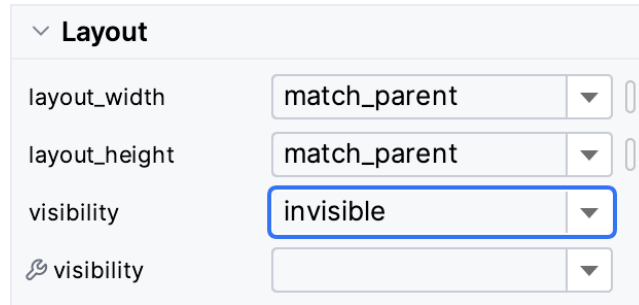


Figure 26-38

## 26.16 Working with the Flow Helper

Flow helpers may be added using either the menu or Palette, as outlined previously for the other helpers. As with the Group helper (Figure 26-36), widgets are added to a Flow instance by dragging them within the Component Tree onto the Flow entry. Having added a Flow helper and assigned widgets to it, select it in the Component Tree and use the Common Attributes section of the Attribute tool window to configure the flow layout behavior:

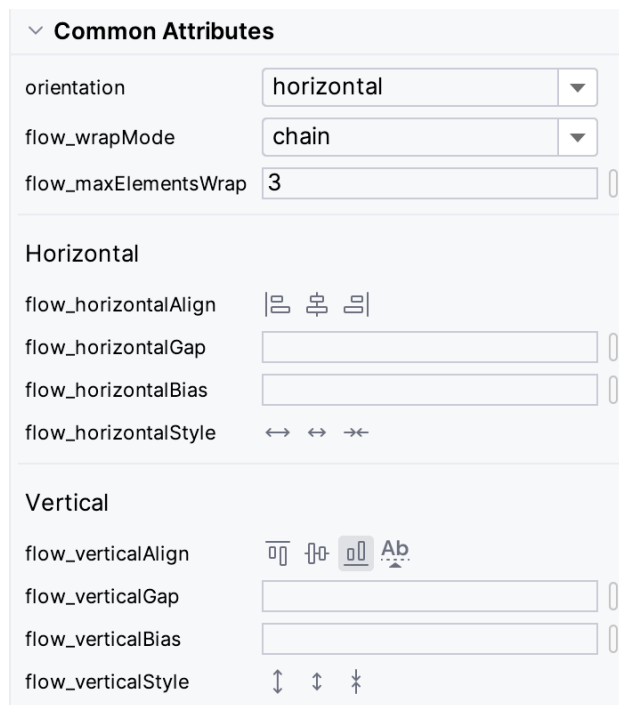


Figure 26-39

## 26.17 Widget Group Alignment and Distribution

The Android Studio Layout Editor tool provides a range of alignment and distribution actions that can be performed when two or more widgets are selected in the layout. Shift-click on each of the widgets to be included in the action, right-click on the layout and make a selection from the many options displayed in the Align menu:

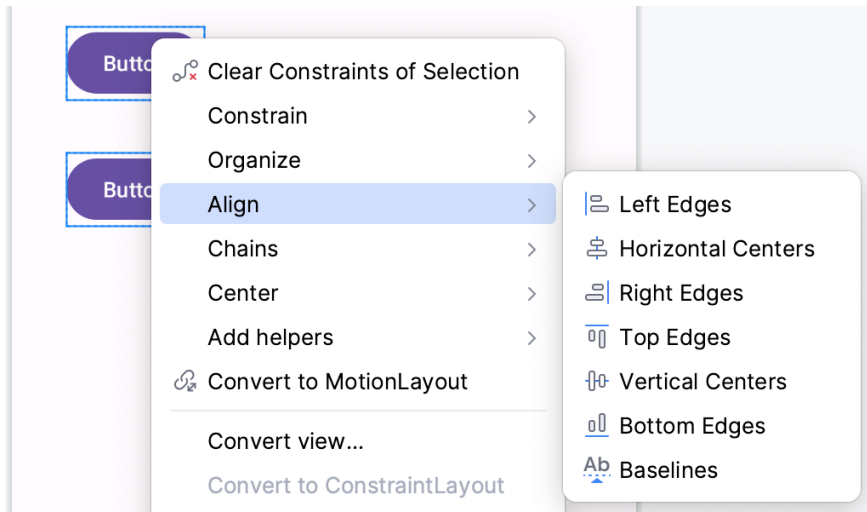


Figure 26-40

As shown in Figure 26-41 below, these options are also accessible via the Align button located in the Layout Editor toolbar:

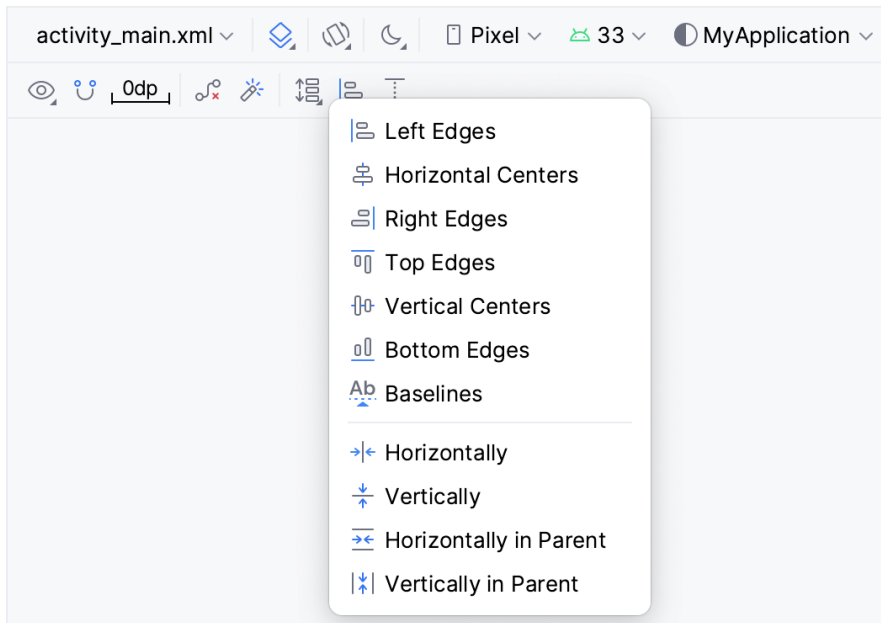


Figure 26-41

Similarly, the Pack menu (Figure 26-42) can be used to collectively reposition the selected widgets so that they are packed tightly together, either vertically or horizontally. It achieves this by changing the widgets' absolute x and y coordinates but does not apply any constraints. The two distribution options in the Pack menu, on the other hand, move the selected widgets so that they are spaced evenly apart in either vertical or horizontal axis and apply constraints between the views to maintain this spacing:



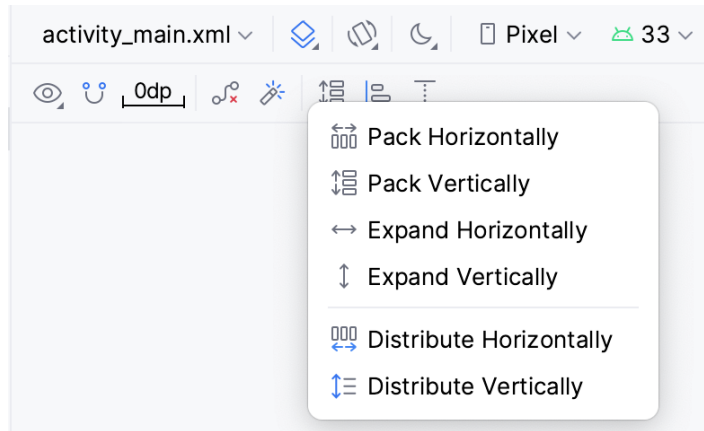


Figure 26-42

## 26.18 Converting other Layouts to ConstraintLayout

For existing user interface layouts that use one or more of the other Android layout classes (such as `RelativeLayout` or `LinearLayout`), the Layout Editor tool provides an option to convert the user interface to use the `ConstraintLayout`.

The Component Tree panel is displayed beneath the Palette when the Layout Editor tool is open and in Design mode. To convert a layout to `ConstraintLayout`, locate it within the Component Tree, right-click on it, and select the *Convert <current layout> to Constraint Layout* menu option:

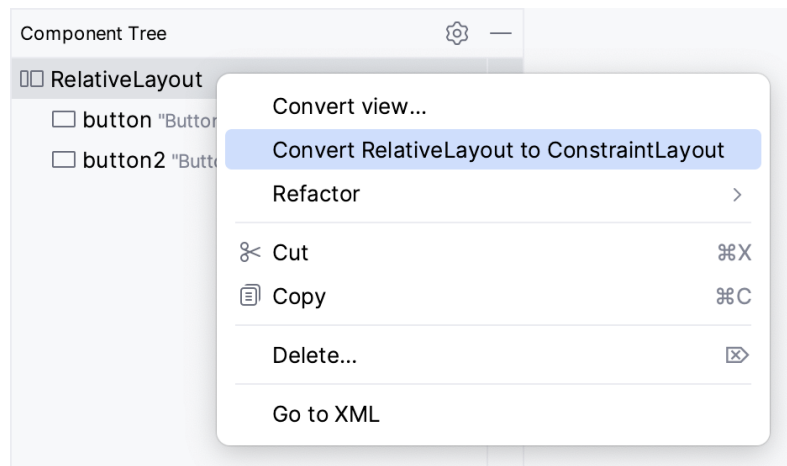


Figure 26-43

When this menu option is selected, Android Studio will convert the selected layout to a `ConstraintLayout` and use inference to establish constraints designed to match the layout behavior of the original layout type.

## 26.19 Summary

A redesigned Layout Editor tool combined with `ConstraintLayout` makes designing complex user interface layouts with Android Studio a relatively fast and intuitive process. This chapter has covered the concepts of constraints, margins, and bias in more detail while also exploring how `ConstraintLayout`-based design has been integrated into the Layout Editor tool.



## 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 use these features within the Layout Editor. Therefore, this chapter’s focus 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 by adding a few lines to an activity’s XML layout resource file or by using some chain-specific features of the Layout Editor.

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

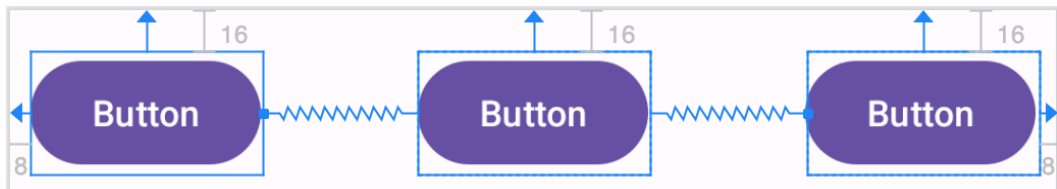


Figure 27-1

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"
```

## Working with ConstraintLayout Chains and Ratios in Android Studio

```
    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 constitutes a ConstraintLayout chain represented visually within the Layout Editor by chain connections, as shown in Figure 27-2 below. Note that the chain has defaulted to the *spread* chain style in this configuration.

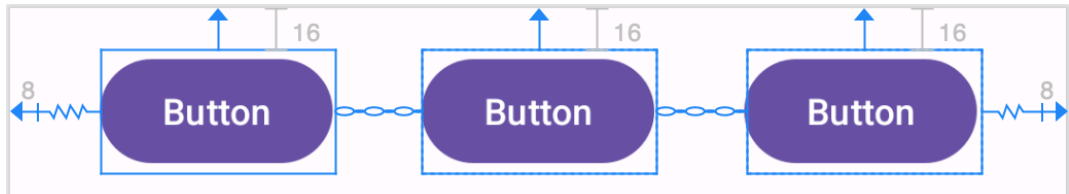


Figure 27-2

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:

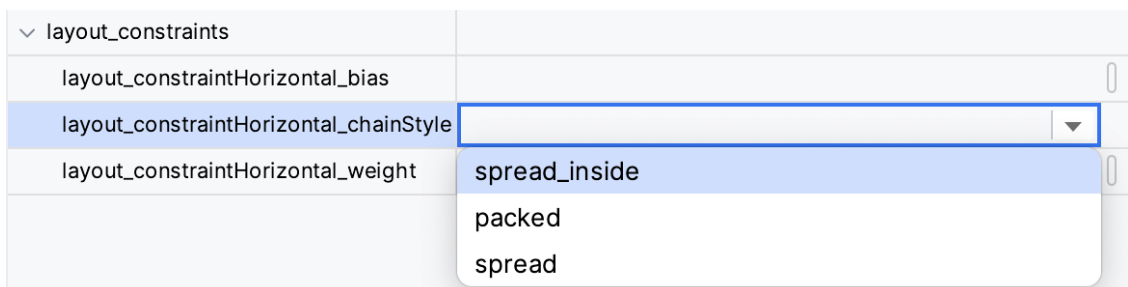


Figure 27-3

## 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:

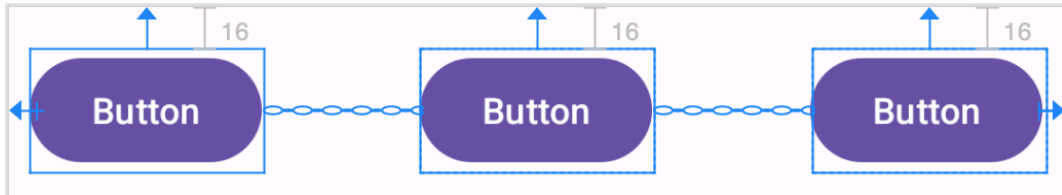


Figure 27-4

## 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:

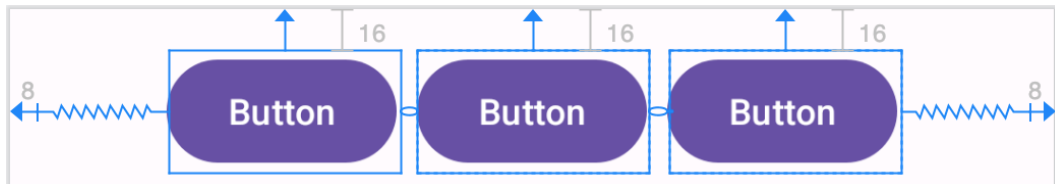


Figure 27-5

## 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 between 0.0 and 1.0, with 0.5 representing the parent's center. 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:

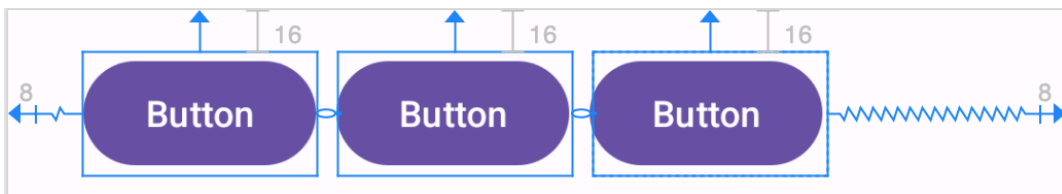


Figure 27-6

## 27.6 Weighted Chain

The final area of chains to explore involves weighting 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 responds 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:

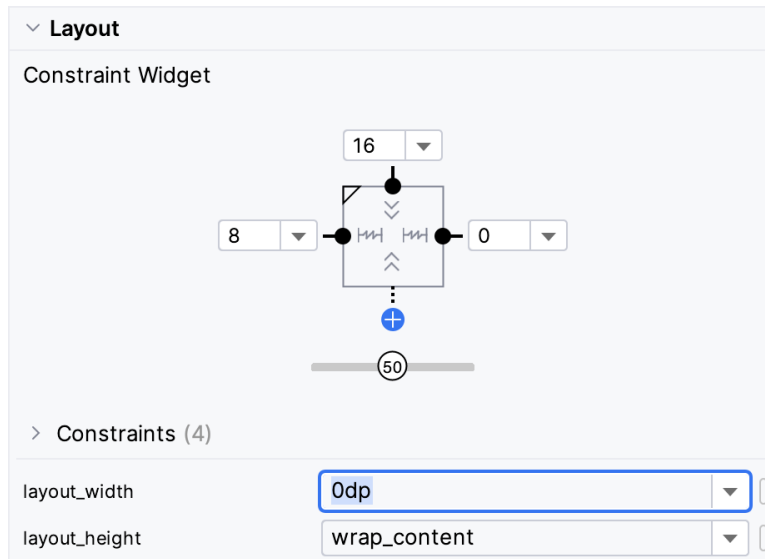


Figure 27-7

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:

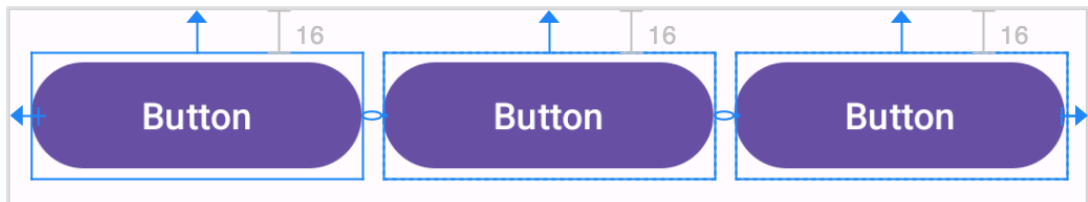


Figure 27-8

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:

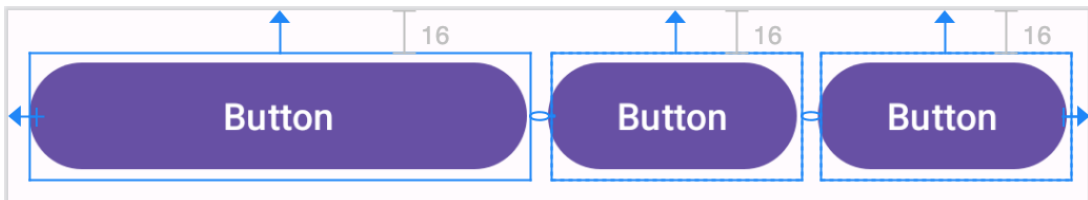


Figure 27-9

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 widget dimension to be sized relative to the widget's other dimension (also referred to as aspect ratio). For example, an aspect ratio setting could 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 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. On the other hand, the width dimension 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 manually editing 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:

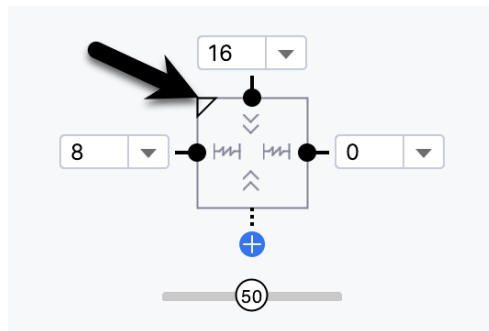


Figure 27-10

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:



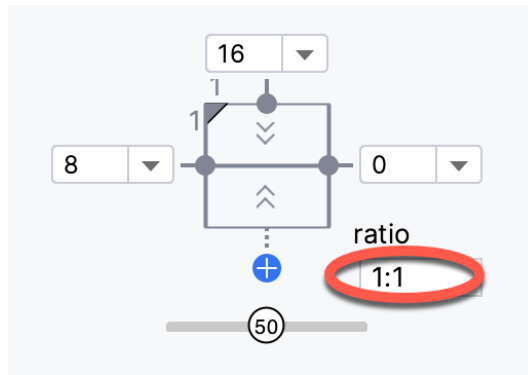


Figure 27-11

## 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.



## 38. Using Fragments in Android Studio - An Example

As outlined in the previous chapter, fragments provide a convenient mechanism for creating reusable modules of application functionality consisting of both sections of a user interface and the corresponding behavior. Once created, fragments can be embedded within activities.

Having explored the general theory of fragments in the previous chapter, this chapter aims to create an example Android application using Android Studio designed to demonstrate the actual steps involved in creating and using fragments and implementing communication between one fragment and another within an activity.

### 38.1 About the Example Fragment Application

The application created in this chapter will consist of a single activity and two fragments. The user interface for the first fragment will contain a toolbar consisting of an EditText view, a SeekBar, and a Button, all contained within a ConstraintLayout view. The second fragment will consist solely of a TextView object within a ConstraintLayout view.

The two fragments will be embedded within the main activity of the application and communication implemented such that when the button in the first fragment is pressed, the text entered into the EditText view will appear on the TextView of the second fragment using a font size dictated by the position of the SeekBar in the first fragment.

Since this application is intended to work on earlier versions of Android, we will need to use the appropriate Android support library.

### 38.2 Creating the Example Project

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

Enter *FragmentExample* into the Name field and specify *com.ebookfrenzy.fragmentexample* as the package name. Before clicking the Finish button, change the Minimum API level setting to API 26: Android 8.0 (Oreo) and the Language menu to Kotlin. Modify the project to use view binding using the steps outlined in *18.8 Migrating a Project to View Binding*.

Return to the *Gradle Scripts* -> *build.gradle.kts* (Module :app) file and add the following directive to the *dependencies* section (keeping in mind that a more recent version of the library may now be available):

```
implementation ("androidx.navigation:navigation-fragment-ktx:2.6.0")
```

### 38.3 Creating the First Fragment Layout

The next step is to create the user interface for the first fragment used within our activity.

This user interface will consist of an XML layout file and a fragment class. While these could be added manually, it is quicker to ask Android Studio to create them for us. Within the project tool window, locate the *app* -> *java* -> *com.ebookfrenzy.fragmentexample* entry and right-click on it. From the resulting menu, select the *New* ->

Fragment -> Gallery... option to display the dialog shown in Figure 38-1 below:

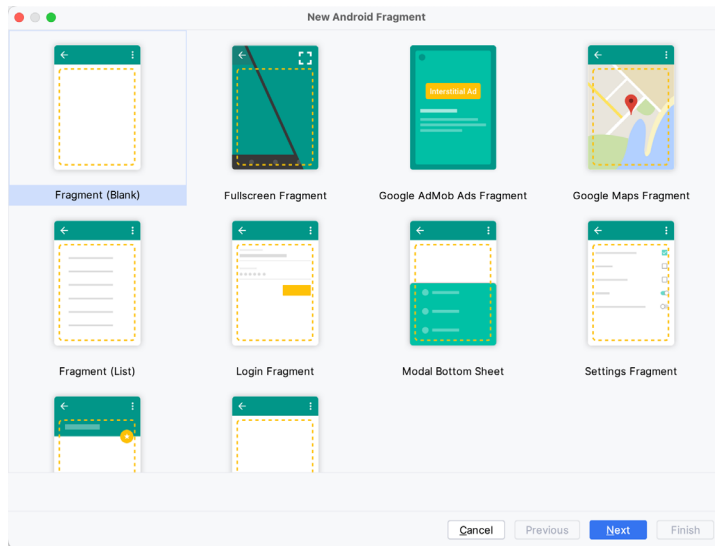


Figure 38-1

Select the *Fragment (Blank)* template before clicking the Next button. On the subsequent screen, name the fragment *ToolbarFragment* with a layout file named *fragment\_toolbar*:

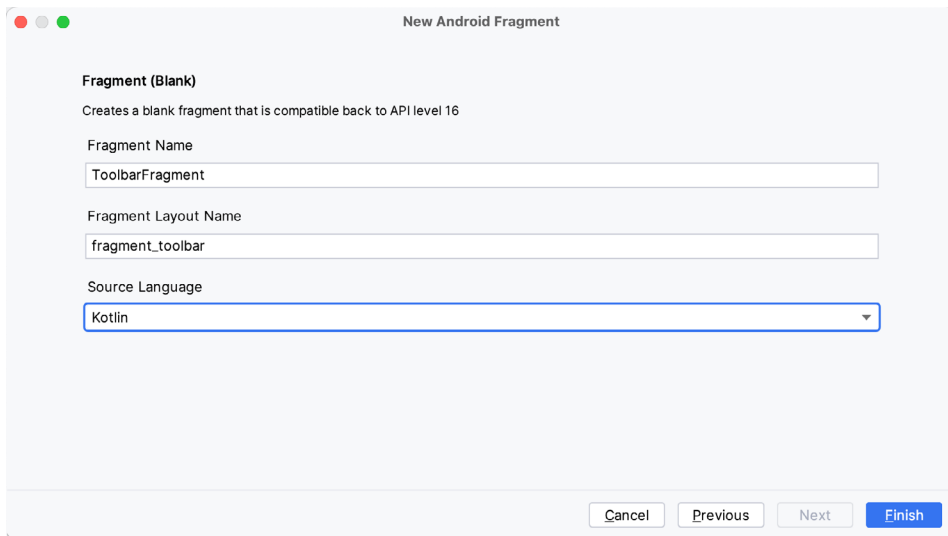


Figure 38-2

Load the *fragment\_toolbar.xml* file into the layout editor using Design mode. Next, right-click on the *FrameLayout* entry in the Component Tree panel and select the *Convert FrameLayout to ConstraintLayout* menu option, accepting the default settings in the confirmation dialog. Change the id from to *constraintLayout*. Ensure that Autoconnect mode is enabled, then select and delete the default *TextView* and add Plain Text, Seekbar, and Button widgets to the layout so that their positions match those shown in Figure 38-3. Finally, change the view ids to *editText1*, *seekBar1*, and *button1*, respectively.

Change the text on the button to read “Change Text”, extract the text to a string resource named *change\_text*,

and remove the Name text from the EditText view. Finally, set the `layout_width` property of the Seekbar to `match_constraint` with margins set to 16dp on the left and right edges.

Use the *Infer constraints* toolbar button to add any missing constraints, at which point the layout should match that shown in Figure 38-3 below:

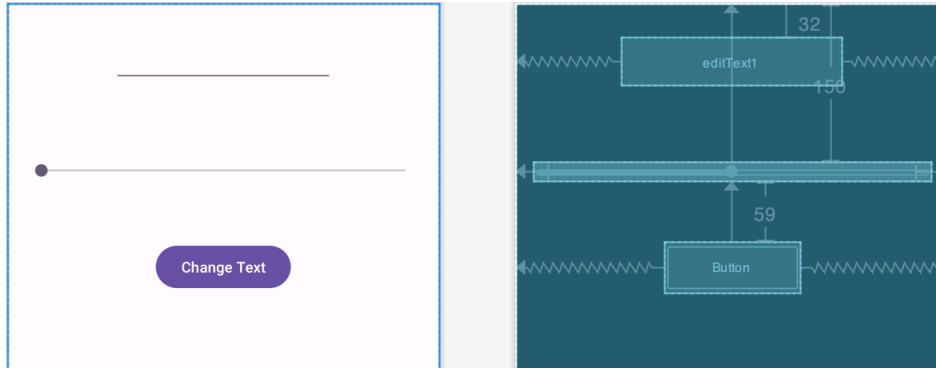


Figure 38-3

## 38.4 Migrating a Fragment to View Binding

As with the Empty Views Activity template, Android Studio does not enable view binding support when new fragments are added to a project. Therefore, we will need to perform this migration before moving to the next step of this tutorial. Begin by editing the `ToolbarFragment.kt` file and importing the binding for the fragment as follows:

```
import com.ebookfrenzy.fragmentexample.databinding.FragmentToolbarBinding
```

Next, locate the `onCreateView()` method and make the following declarations and changes (which also include adding the `onDestroyView()` method to ensure that the binding reference is removed when the fragment is destroyed):

```
.
.
private var _binding: FragmentToolbarBinding? = null
private val binding get() = _binding!!
.
.
override fun onCreateView(
    inflater: LayoutInflater, container: ViewGroup?,
    savedInstanceState: Bundle?
): View? {
return inflater.inflate(R.layout.fragment_toolbar, container, false)
    _binding = FragmentToolbarBinding.inflate(inflater, container, false)
    return binding.root
}

override fun onDestroyView() {
    super.onDestroyView()
    _binding = null
}
```

Once these changes are complete, the fragment is ready to use view binding.

## 38.5 Adding the Second Fragment

Repeating the steps to create the toolbar fragment, add another empty fragment named `TextFragment` with a layout file named `fragment_text`. Once again, convert the `FrameLayout` container to a `ConstraintLayout` (changing the id to `constraintLayout2`) and remove the default `TextView`.

Drag a drop a `TextView` widget from the palette and position it in the center of the layout, using the *Infer constraints* button to add any missing constraints. Change the id of the `TextView` to `textView2`, the text to read “Fragment Two” and modify the `textSize` attribute to 24sp.

On completion, the layout should match that shown in Figure 38-4:

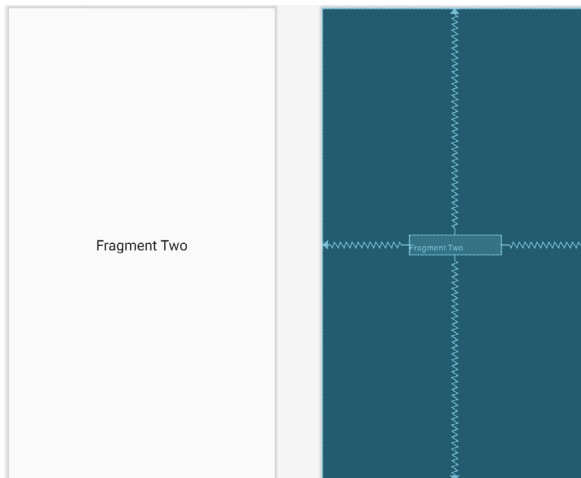


Figure 38-4

Repeat the steps performed in the previous section to migrate the `TextFragment` class to use view binding as follows:

```
.
.
import com.ebookfrenzy.fragmentexample.databinding.FragmentTextBinding
.
.
private var _binding: FragmentTextBinding? = null
private val binding get() = _binding!!
.
.
override fun onCreateView(
    inflater: LayoutInflater, container: ViewGroup?,
    savedInstanceState: Bundle?
): View? {
    return inflater.inflate(R.layout.fragment_text, container, false)
    _binding = FragmentTextBinding.inflate(inflater, container, false)
    return binding.root
}
```

## 38.6 Adding the Fragments to the Activity

The main activity for the application has associated with it an XML layout file named *activity\_main.xml*. For this example, the fragments will be added to the activity using the `<fragment>` element within this file. Using the Project tool window, navigate to the *app -> res -> layout* section of the *FragmentExample* project and double-click on the *activity\_main.xml* file to load it into the Android Studio Layout Editor tool.

With the Layout Editor tool in Design mode, select and delete the default `TextView` object from the layout and select the *Common* category in the palette. Drag the *FragmentContainerView* component from the list of views and drop it onto the layout so that it is centered horizontally and positioned such that the dashed line appears, indicating the top layout margin:

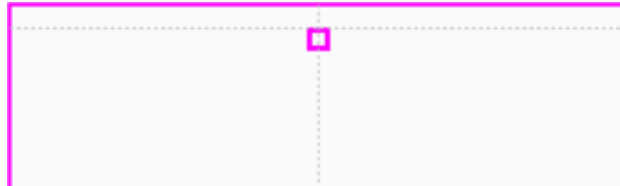


Figure 38-5

On dropping the fragment onto the layout, a dialog will appear displaying a list of Fragments available within the current project, as illustrated in Figure 38-6:



Figure 38-6

Select the `ToolbarFragment` entry from the list and click `OK` to dismiss the `Fragments` dialog. Once added, click the red warning button in the top right-hand corner of the layout editor to display the `Problems` tool window. An *unknown fragments* message will indicate that the Layout Editor tool needs to know which fragment to display during the preview session. Select the `Unknown fragment` item, then click on the *Pick Layout...* link in the right-hand panel as shown in Figure 38-7:

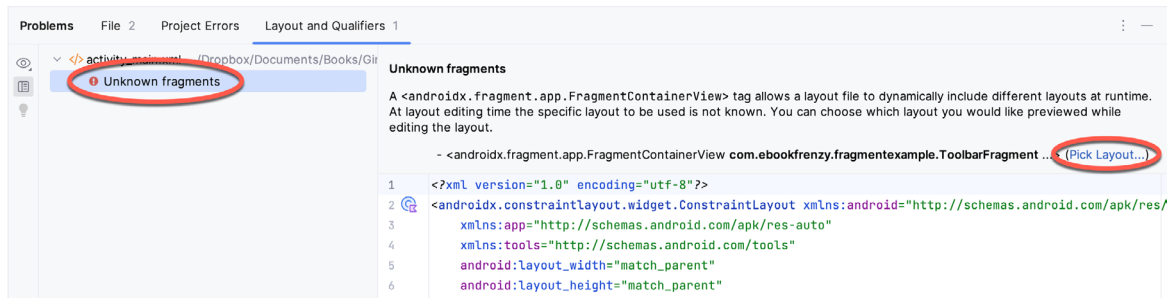


Figure 38-7

In the resulting dialog (Figure 38-8), select the *fragment\_toolbar* entry and then click `OK`:

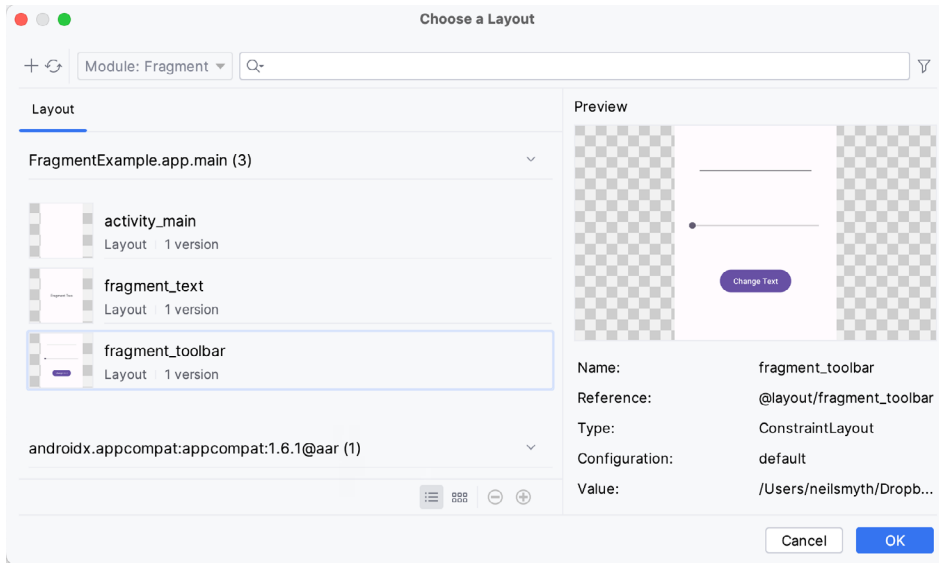


Figure 38-8

With the fragment selected, change the *layout\_width* property to *match\_constraint* so that it occupies the full width of the screen. Click and drag another *FragmentContainerView* entry from the palette and position it so that it is centered horizontally and located beneath the bottom edge of the first fragment. When prompted, select the *TextFragment* entry from the fragment dialog before clicking OK. Display the Problems tool window and repeat the previous steps, this time selecting the *fragment\_text* layout. Use the *Infer constraints* button to establish any missing layout constraints.

Note that the fragments are now visible in the layout, as demonstrated in Figure 38-9:

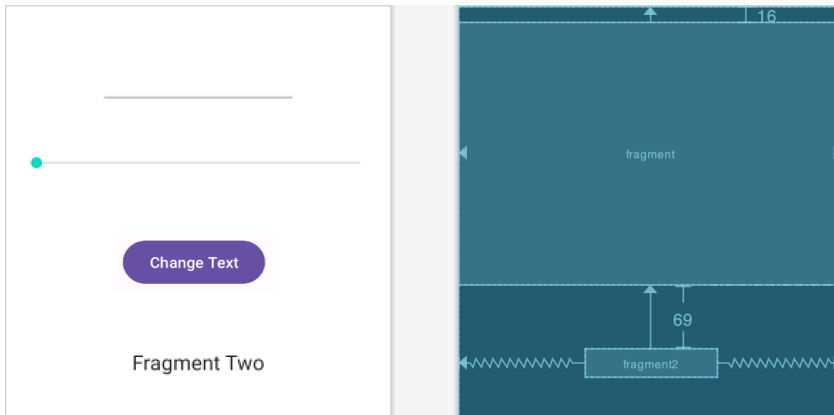


Figure 38-9

Before proceeding to the next step, select the *TextFragment* instance in the layout and, within the Attributes tool window, change the ID of the fragment to *text\_fragment*.

### 38.7 Making the Toolbar Fragment Talk to the Activity

When the user touches the button in the toolbar fragment, the fragment class will need to extract the text from the *EditText* view and the current value of the *SeekBar* and send them to the text fragment. As outlined in “*An Introduction to Android Fragments*”, fragments should not communicate with each other directly, instead using



the activity in which they are embedded as an intermediary.

The first step in this process is ensuring that the toolbar fragment responds to the clicked button. We also need to implement some code to keep track of the value of the SeekBar view. For this example, we will implement these listeners within the `ToolbarFragment` class. Select the `ToolbarFragment.kt` file and modify it so that it reads as shown in the following listing:

```
package com.ebookfrenzy.fragmentexample

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import android.widget.SeekBar
import android.content.Context
.
.
class ToolbarFragment : Fragment(), SeekBar.OnSeekBarChangeListener {
.
.
    var seekvalue = 10
.
.
    override fun onCreateView(view: View, savedInstanceState: Bundle?) {
        super.onCreateView(view, savedInstanceState)

        binding.seekBar1.setOnSeekBarChangeListener(this)
        binding.button1.setOnClickListener { v: View -> buttonClicked(v) }
    }

    private fun buttonClicked(view: View) {

    }

    override fun onProgressChanged(seekBar: SeekBar, progress: Int,
                                   fromUser: Boolean) {
        seekvalue = progress
    }

    override fun onStartTrackingTouch(arg0: SeekBar) {
    }

    override fun onStopTrackingTouch(arg0: SeekBar) {
    }
.
.
```

}

Before moving on, we need to take some time to explain the above code changes. First, the class is declared as implementing the `OnSeekBarChangeListener` interface. This is because the user interface contains a `SeekBar` instance, and the fragment needs to receive notifications when the user slides the bar to change the font size. Implementation of the `OnSeekBarChangeListener` interface requires that the `onProgressChanged()`, `onStartTrackingTouch()`, and `onStopTrackingTouch()` methods be implemented. These methods have been implemented, but only the `onProgressChanged()` method is required to perform a task, in this case, storing the new value in a variable named `seekvalue`, which was declared at the start of the class. Also declared is a variable to store a reference to the `EditText` object.

The `onViewCreated()` method has been added to set up an `OnClickListener` on the button, which is configured to call a method named `buttonClicked()` when a click event is detected. This method is also then implemented, though it does not do anything at this point.

The next phase of this process is to set up the listener that will allow the fragment to call the activity when the button is clicked. This follows the mechanism outlined in the previous chapter:

```
class ToolbarFragment : Fragment(), SeekBar.OnSeekBarChangeListener {
    .
    .
    var seekvalue = 10

    var activityCallback: ToolbarFragment.ToolbarListener? = null

    interface ToolbarListener {
        fun onClick(fontSize: Int, text: String)
    }

    override fun onAttach(context: Context) {
        super.onAttach(context)
        try {
            activityCallback = context as ToolbarListener
        } catch (e: ClassCastException) {
            throw ClassCastException(context.toString()
                + " must implement ToolbarListener")
        }
    }
}

private fun buttonClicked(view: View) {
    activityCallback?.onClick(seekvalue,
        binding.editText1.text.toString())
}
}
```

The above implementation will result in a method named `onClick()` belonging to the activity class being

called when the user clicks the button. All that remains, therefore, is to declare that the activity class implements the newly created `ToolbarListener` interface and to implement the `onButtonClick()` method.

Since the Android Support Library is being used for fragment support in earlier Android versions, the activity also needs to be changed to subclass from `FragmentActivity` instead of `AppCompatActivity`. Bringing these requirements together results in the following modified `MainActivity.kt` file:

```
package com.ebookfrenzy.fragmentexample

import androidx.appcompat.app.AppCompatActivity
import androidx.fragment.app.FragmentActivity
import android.os.Bundle

class MainActivity : FragmentActivity(),
                    ToolbarFragment.ToolbarListener {
    .
    .
    override fun onButtonClick(fontSize: Int, text: String) {
    }
}
```

With the code changes as they currently stand, the toolbar fragment will detect when the user clicks the button and call a method on the activity passing through the content of the `EditText` field and the current setting of the `SeekBar` view. It is now the job of the activity to communicate with the `Text Fragment` and to pass along these values so that the fragment can update the `TextView` object accordingly.

## 38.8 Making the Activity Talk to the Text Fragment

As “*An Introduction to Android Fragments*” outlined, an activity can communicate with a fragment by obtaining a reference to the fragment class instance and then calling public methods on the object. As such, within the `TextFragment` class, we will now implement a public method named `changeTextProperties()` which takes as arguments an integer for the font size and a string for the new text to be displayed. The method will then use these values to modify the `TextView` object. Within the Android Studio editing panel, locate and modify the `TextFragment.kt` file to add this new method:

```
package com.ebookfrenzy.fragmentexample
.
.
class TextFragment : Fragment() {
    .
    .
    fun changeTextProperties(fontSize: Int, text: String)
    {
        binding.textView2.textSize = fontSize.toFloat()
        binding.textView2.text = text
    }
    .
    .
}
```

## Using Fragments in Android Studio - An Example

When the `TextFragment` fragment was placed in the activity's layout, it was given an ID of `text_fragment`. Using this ID, it is now possible for the activity to obtain a reference to the fragment instance and call the `changeTextProperties()` method on the object. Edit the `MainActivity.kt` file and modify the `onButtonClick()` method as follows:

```
override fun onButtonClick(fontSize: Int, text: String) {  
  
    val textFragment = supportFragmentManager.findFragmentById(  
        R.id.text_fragment) as TextFragment  
  
    textFragment.changeTextProperties(fontSize, text)  
}
```

## 38.9 Testing the Application

With the coding for this project now complete, the last remaining task is to run the application. When the application is launched, the main activity will start and will, in turn, create and display the two fragments. When the user touches the button in the toolbar fragment, the `onButtonClick()` method of the activity will be called by the toolbar fragment and passed the text from the `EditText` view and the current value of the `SeekBar`. The activity will then call the `changeTextProperties()` method of the second fragment, which will modify the `TextView` to reflect the new text and font size:

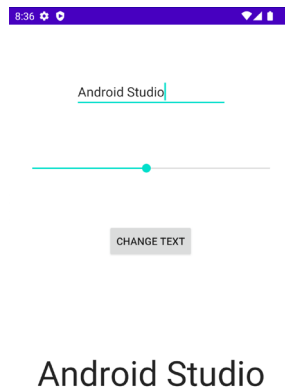


Figure 38-10

## 38.10 Summary

The goal of this chapter was to work through creating an example project to demonstrate the steps involved in using fragments within an Android application. Topics covered included using the Android Support Library for compatibility with Android versions predating the introduction of fragments, including fragments within an activity layout, and implementing inter-fragment communication.

## 39. Modern Android App Architecture with Jetpack

For many years, Google did not recommend a specific approach to building Android apps other than to provide tools and development kits while letting developers decide what worked best for a particular project or individual programming style. That changed in 2017 with the introduction of the Android Architecture Components, which, in turn, became part of Android Jetpack when it was released in 2018.

This chapter provides an overview of the concepts of Jetpack, Android app architecture recommendations, and some key architecture components. Once the basics have been covered, these topics will be covered in more detail and demonstrated through practical examples in later chapters.

### 39.1 What is Android Jetpack?

Android Jetpack consists of Android Studio, the Android Architecture Components, the Android Support Library, and a set of guidelines recommending how an Android App should be structured. The Android Architecture Components are designed to make it quicker and easier to perform common tasks when developing Android apps while also conforming to the key principle of the architectural guidelines.

While all Android Architecture Components will be covered in this book, this chapter will focus on the key architectural guidelines and the ViewModel, LiveData, and Lifecycle components while introducing Data Binding and Repositories.

Before moving on, it is important to understand that the Jetpack approach to app development is optional. While highlighting some of the shortcomings of other techniques that have gained popularity over the years, Google stopped short of completely condemning those approaches to app development. Google is taking the position that while there is no right or wrong way to develop an app, there is a recommended way.

### 39.2 The “Old” Architecture

In the chapter entitled “*Creating an Example Android App in Android Studio*”, an Android project was created consisting of a single activity that contained all of the code for presenting and managing the user interface together with the back-end logic of the app. Until the introduction of Jetpack, the most common architecture followed this paradigm with apps consisting of multiple activities (one for each screen within the app), with each activity class to some degree mixing user interface and back-end code.

This approach led to a range of problems related to the lifecycle of an app (for example, an activity is destroyed and recreated each time the user rotates the device leading to the loss of any app data that had not been saved to some form of persistent storage) as well as issues such as inefficient navigation involving launching a new activity for each app screen accessed by the user.

### 39.3 Modern Android Architecture

At the most basic level, Google now advocates single-activity apps where different screens are loaded as content within the same activity.

Modern architecture guidelines also recommend separating different areas of responsibility within an app into entirely separate modules (a concept referred to as “separation of concerns”). One of the keys to this approach

is the ViewModel component.

## 39.4 The ViewModel Component

The purpose of ViewModel is to separate the user interface-related data model and logic of an app from the code responsible for displaying and managing the user interface and interacting with the operating system. When designed this way, an app will consist of one or more UI Controllers, such as an activity, together with ViewModel instances responsible for handling the data those controllers need.

The ViewModel only knows about the data model and corresponding logic. It knows nothing about the user interface and does not attempt to directly access or respond to events relating to views within the user interface. When a UI controller needs data to display, it asks the ViewModel to provide it. Similarly, when the user enters data into a view within the user interface, the UI controller passes it to the ViewModel for handling.

This separation of responsibility addresses the issues relating to the lifecycle of UI controllers. Regardless of how often the UI controller is recreated during the lifecycle of an app, the ViewModel instances remain in memory, thereby maintaining data consistency. For example, a ViewModel used by an activity will remain in memory until the activity finishes, which, in the single activity app, is not until the app exits.

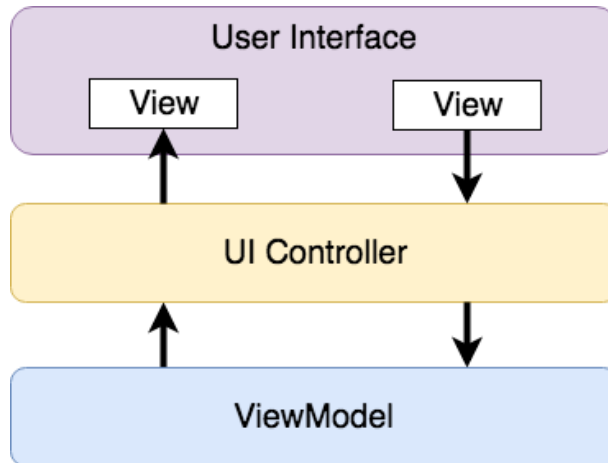


Figure 39-1

## 39.5 The LiveData Component

Consider an app that displays real-time data, such as the current price of a financial stock. The app could use a stock price web service to continuously update the data model within the ViewModel with the latest information. This real-time data is of use only if it is displayed to the user promptly. There are only two ways that the UI controller can ensure that the latest data is displayed in the user interface. One option is for the controller to continuously check with the ViewModel to determine if the data has changed since it was last displayed. However, the problem with this approach is that it could be more efficient. To maintain the real-time nature of the data feed, the UI controller would have to run on a loop, continuously checking for the data to change.

A better solution would be for the UI controller to receive a notification when a specific data item within a ViewModel changes. This is made possible by using the LiveData component. LiveData is a data holder that allows a value to become *observable*. In basic terms, an observable object can notify other objects when changes to its data occur, thereby solving the problem of ensuring that the user interface always matches the data within the ViewModel.

This means, for example, that a UI controller interested in a ViewModel value can set up an observer, which will, in turn, be notified when that value changes. In our hypothetical application, for example, the stock price would

be wrapped in a LiveData object within the ViewModel, and the UI controller would assign an observer to the value, declaring a method to be called when the value changes. When triggered by data change, this method will read the updated value from the ViewModel and use it to update the user interface.

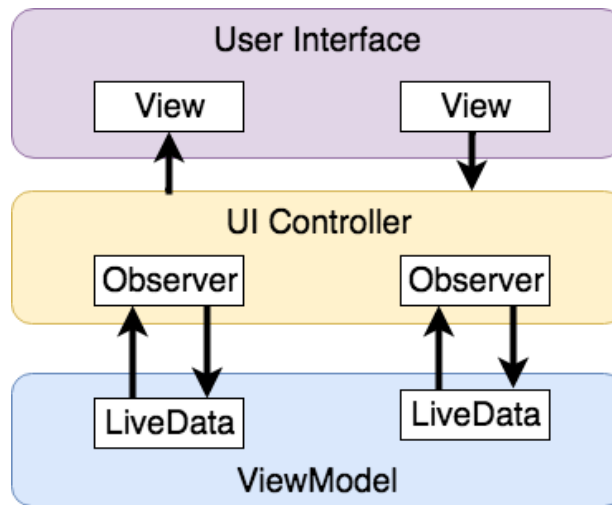


Figure 39-2

A LiveData instance may also be declared as mutable, allowing the observing entity to update the underlying value held within the LiveData object. The user might, for example, enter a value in the user interface that needs to overwrite the value stored in the ViewModel.

Another of the key advantages of using LiveData is that it is aware of the *lifecycle state* of its observers. If, for example, an activity contains a LiveData observer, the corresponding LiveData object will know when the activity's lifecycle state changes and respond accordingly. If the activity is paused (perhaps the app is put into the background), the LiveData object will stop sending events to the observer. Suppose the activity has just started or resumes after being paused. In that case, the LiveData object will send a LiveData event to the observer so that the activity has the most up-to-date value. Similarly, the LiveData instance will know when the activity is destroyed and remove the observer to free up resources.

So far, we've only talked about UI controllers using observers. In practice, however, an observer can be used within any object that conforms to the Jetpack approach to lifecycle management.

## 39.6 ViewModel Saved State

Android allows the user to place an active app in the background and return to it after performing other tasks on the device (including running other apps). When a device runs low on resources, the operating system will rectify this by terminating background app processes, starting with the least recently used app. However, when the user returns to the terminated background app, it should appear in the same state as when it was placed in the background, regardless of whether it was terminated. In terms of the data associated with a ViewModel, this can be implemented using the ViewModel Saved State module. This module allows values to be stored in the app's *saved state* and restored in case of system-initiated process termination. This topic will be covered later in the "*An Android ViewModel Saved State Tutorial*" chapter.

## 39.7 LiveData and Data Binding

Android Jetpack includes the Data Binding Library, which allows data in a ViewModel to be mapped directly to specific views within the XML user interface layout file. In the AndroidSample project created earlier, code had to be written to obtain references to the EditText and TextView views and to set and get the text properties to

reflect data changes. Data binding allows the LiveData value stored in the ViewModel to be referenced directly within the XML layout file avoiding the need to write code to keep the layout views updated.

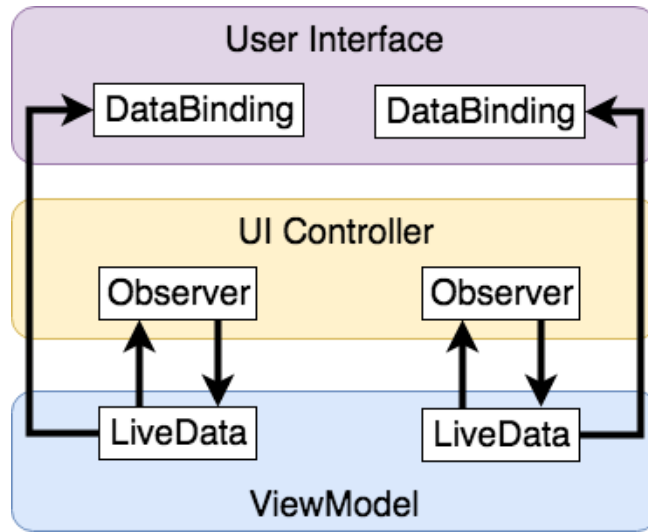


Figure 39-3

Data binding will be covered in greater detail, starting with the chapter “An Overview of Android Jetpack Data Binding”.

## 39.8 Android Lifecycles

The duration from when an Android component is created to the point that it is destroyed is called the *lifecycle*. During this lifecycle, the component will change between different lifecycle states, usually under the operating system’s control and in response to user actions. An activity, for example, will begin in the *initialized* state before transitioning to the *created* state. Once the activity runs, it will switch to the *started* state, from which it will cycle through various states, including *created*, *started*, *resumed*, and *destroyed*.

Many Android Framework classes and components allow other objects to access their current state. *Lifecycle observers* may also be used so that an object receives a notification when the lifecycle state of another object changes. The ViewModel component uses this technique behind the scenes to identify when an observer has restarted or been destroyed. This functionality is not limited to Android framework and architecture components. It may also be built into any other classes using a set of lifecycle components included with the architecture components.

Objects that can detect and react to lifecycle state changes in other objects are said to be *lifecycle-aware*. In contrast, objects that provide access to their lifecycle state are called *lifecycle owners*. The chapter entitled “Working with Android Lifecycle-Aware Components” will cover Lifecycles in greater detail.

## 39.9 Repository Modules

If a ViewModel obtains data from one or more external sources (such as databases or web services, it is important to separate the code involved in handling those data sources from the ViewModel class. Failure to do this would, after all, violate the separation of concerns guidelines. To avoid mixing this functionality with the ViewModel, Google’s architecture guidelines recommend placing this code in a separate *Repository* module.

A repository is not an Android architecture component but a Kotlin class created by the app developer that is responsible for interfacing with the various data sources. The class then provides an interface to the ViewModel, allowing that data to be stored in the model.



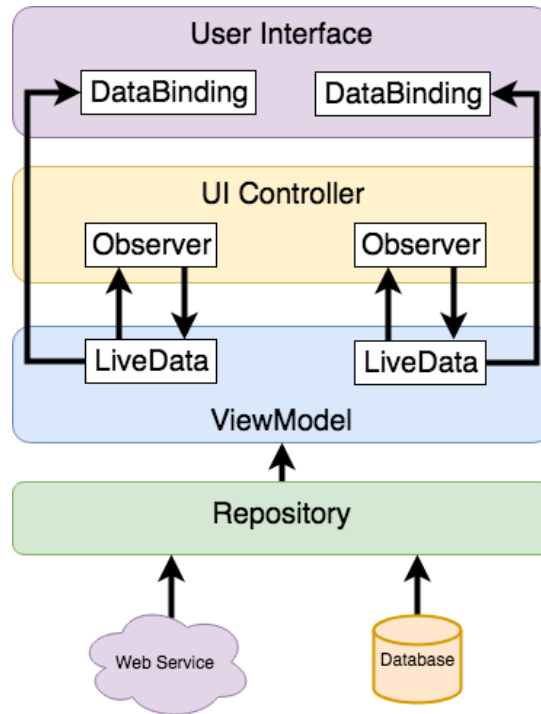


Figure 39-4

### 39.10 Summary

Until recently, Google has tended not to recommend any particular approach to structuring an Android app. That has now changed with the introduction of Android Jetpack, consisting of tools, components, libraries, and architecture guidelines. Google now recommends that an app project be divided into separate modules, each responsible for a particular area of functionality, otherwise known as “separation of concerns”.

In particular, the guidelines recommend separating the view data model of an app from the code responsible for handling the user interface. In addition, the code responsible for gathering data from data sources such as web services or databases should be built into a separate repository module instead of being bundled with the view model.

Android Jetpack includes the Android Architecture Components, designed to make developing apps that conform to the recommended guidelines easier. This chapter has introduced the ViewModel, LiveData, and Lifecycle components. These will be covered in more detail, starting with the next chapter. Other architecture components not mentioned in this chapter will be covered later in the book.



## 41. An Android Jetpack LiveData Tutorial

The previous chapter began building an app to conform to the recommended Jetpack architecture guidelines. These initial steps involved implementing the data model for the app user interface within a `ViewModel` instance.

This chapter will further enhance the app design using the LiveData architecture component. Once LiveData support has been added to the project in this chapter, the next chapters (starting with “*An Overview of Android Jetpack Data Binding*”) will use the Jetpack Data Binding library to eliminate even more code from the project.

### 41.1 LiveData - A Recap

LiveData was previously introduced in the “*Modern Android App Architecture with Jetpack*” chapter. As described earlier, the LiveData component can be used as a wrapper around data values within a view model. Once contained in a LiveData instance, those variables become observable to other objects within the app, typically UI controllers such as Activities and Fragments. This allows the UI controller to receive a notification whenever the underlying LiveData value changes. An observer is set up by creating an instance of the Observer class and defining an `onChange()` method to be called when the LiveData value changes. Once the Observer instance has been created, it is attached to the LiveData object via a call to the LiveData object’s `observe()` method.

LiveData instances can be declared mutable using the `MutableLiveData` class, allowing both the `ViewModel` and UI controller to change the underlying data value.

### 41.2 Adding LiveData to the ViewModel

Launch Android Studio, open the `ViewModelDemo` project created in the previous chapter, and open the `MainViewModel.kt` file, which should currently read 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
    }
}
```

## An Android Jetpack LiveData Tutorial

```
}  
}
```

This stage in the chapter aims to wrap the *result* variable in a `MutableLiveData` instance (the object will need to be mutable so that the value can be changed each time the user requests a currency conversion). Begin by modifying the class so that it now reads as follows, noting that an additional package needs to be imported when making use of `LiveData`:

```
package com.ebookfrenzy.viewmodeldemo  
  
import androidx.lifecycle.ViewModel  
import androidx.lifecycle.MutableLiveData  
  
class MainViewModel : ViewModel() {  
  
    private val rate = 0.74f  
    private var dollarText = ""  
private var result: Float = 0f  
private var result: MutableLiveData<Float> = MutableLiveData()  
  
    fun setAmount(value: String) {  
        this.dollarText = value  
        result = value.toFloat() * rate  
    }  
  
    fun getResult(): Float {  
        return result  
    }  
}
```

Now that the *result* variable is contained in a mutable `LiveData` instance, both the *setAmount()* and *getResult()* methods must be modified. In the case of the *setAmount()* method, a value can no longer be assigned to the *result* variable using the assignment (=) operator. Instead, the `LiveData setValue()` method must be called, passing through the new value as an argument. As currently implemented, the *getResult()* method is declared to return a `Float` value and must be changed to return a `MutableLiveData` object. Making these remaining changes results in the following class file:

```
package com.ebookfrenzy.viewmodeldemo  
  
import androidx.lifecycle.ViewModel  
import androidx.lifecycle.MutableLiveData  
  
class MainViewModel : ViewModel() {  
  
    private val rate = 0.74f  
    private var dollarText = ""  
    private var result: MutableLiveData<Float> = MutableLiveData()  
  
    fun setAmount(value: String) {
```

```

        this.dollarText = value
        result = value.toFloat() * rate
        result.value = value.toFloat() * rate
    }
fun getResult(): Float {
fun getResult(): MutableLiveData<Float> {
    return result
}
}

```

### 41.3 Implementing the Observer

Now that the conversion result is contained within a LiveData instance, the next step is configuring an observer within the UI controller, which, in this example, is the FirstFragment class. Locate the *FirstFragment.kt* class (`app -> java -> <package name> -> FirstFragment`), double-click on it to load it into the editor, and modify the `onViewCreated()` method to create a new Observer instance named *resultObserver*:

```

.
.
import androidx.lifecycle.Observer
.
.
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)

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

    val resultObserver = Observer<Float> {
        result -> binding.resultText.text = result.toString()
    }
.
.
}

```

The *resultObserver* instance declares lambda code which, when called, is passed the current result value, which it then converts to a string and displays on the resultText TextView object. The next step is to add the observer to the result LiveData object, a reference that can be obtained via a call to the `getResult()` method of the ViewModel object. Since updating the result TextView is now the responsibility of the `onChanged()` callback method, the existing lines of code to perform this task can now be deleted:

```

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

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

    val resultObserver = Observer<Float> {
        result -> binding.resultText.text = result.toString()
    }

    viewModel.getResult().observe(viewLifecycleOwner, resultObserver)
}

```

```
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"
    }
}
```

Compile and run the app, enter a value into the dollar field, click on the Convert button, and verify that the converted euro amount appears on the TextView. This confirms that the observer received notification that the result value had changed and called the *onChanged()* method to display the latest data.

Note in the above implementation of the *onViewCreated()* method that the line of code responsible for displaying the current result value each time the method was called was removed. This was originally put in place to ensure that the displayed value was recovered if the Fragment was recreated for any reason. Because LiveData monitors the lifecycle status of its observers, this step is no longer necessary. When LiveData detects that the UI controller was recreated, it automatically triggers any associated observers and provides the latest data. Verify this by rotating the device while a euro value is displayed on the TextView object and confirming that the value is not lost.

Before moving on to the next chapter, close the project, copy the ViewModelDemo project folder, and save it as ViewModelDemo\_LiveData to be used later when saving the ViewModel state.

### 41.4 Summary

This chapter demonstrated the use of the Android LiveData component to ensure that the data displayed to the user always matches that stored in the ViewModel. This relatively simple process consisted of wrapping a ViewModel data value within a LiveData object and setting up an observer within the UI controller subscribed to the LiveData value. Each time the LiveData value changes, the observer is notified, and the *onChanged()* method is called and passed the updated value.

Adding LiveData support to the project has gone some way towards simplifying the design of the project. Additional and significant improvements are also possible using the Data Binding Library, details of which will be covered in the next chapter.

## 52. Working with the Floating Action Button and Snackbar

One of the objectives of this chapter is to provide an overview of the concepts of material design. Originally introduced as part of Android 5.0, material design is a set of design guidelines that dictate how the Android user interface, and that of the apps running on Android, appear and behave.

As part of implementing the material design concepts, Google also introduced the Android Design Support Library. This library contains several components that allow many of the key features of material design to be built into Android applications. Two of these components, the floating action button and the Snackbar, will also be covered in this chapter before introducing many of the other components in subsequent chapters.

### 52.1 The Material Design

The principles of material design define the overall appearance of the Android environment. Material design was created by the Android team at Google and dictates that the elements that make up the user interface of Android and the apps that run on it appear and behave in a certain way in terms of behavior, shadowing, animation, and style. One of the tenets of the material design is that the elements of a user interface appear to have physical depth and a sense that items are constructed in layers of physical material. A button, for example, appears to be raised above the surface of the layout where it resides through shadowing effects. Pressing the button causes the button to flex and lift as though made of a thin material that ripples when released.

Material design also dictates the layout and behavior of many standard user interface elements. A key example is how the app bar located at the top of the screen should appear and how it should behave in relation to scrolling activities taking place within the main content of the activity.

Material design covers a wide range of areas, from recommended color styles to how objects are animated. A full description of the material design concepts and guidelines can be found online at the following link and is recommended reading for all Android developers:

*<https://material.io/design/introduction>*

### 52.2 The Design Library

Many of the building blocks needed to implement Android applications that adopt material design principles are contained within the Android Design Support Library. This library contains a collection of user interface components that can be included in Android applications to implement much of the look, feel, and behavior of material design. Two of the components from this library, the floating action button and Snackbar, will be covered in this chapter, while others will be introduced in later chapters.

### 52.3 The Floating Action Button (FAB)

The floating action button appears to float above the surface of the user interface of an app. It generally promotes the most common action within a user interface screen. A floating action button could be placed on a screen to allow the user to add an entry to a list of contacts or to send an email from within the app. Figure 52-1, for example, highlights the floating action button that allows the user to add a new contact within the standard Android Contacts app:

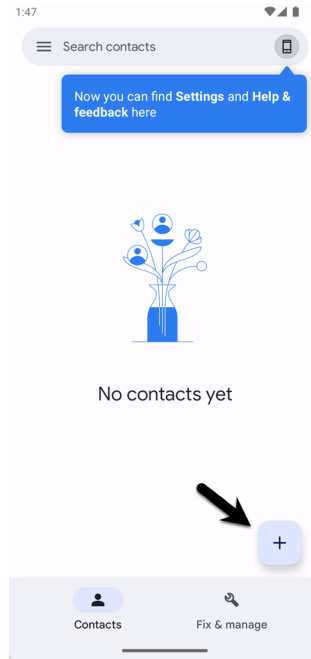


Figure 52-1

Several rules should be followed when using floating action buttons to conform with the material design guidelines. Floating action buttons must be circular and can be either 56 x 56dp (Default) or 40 x 40dp (Mini) in size. The button should be positioned a minimum of 16dp from the edge of the screen on phones and 24dp on desktops and tablet devices. Regardless of the size, the button must contain an interior icon that is 24x24dp in size, and it is recommended that each user interface screen have only one floating action button.

Floating action buttons can be animated or designed to morph into other items when touched. For example, a floating action button could rotate when tapped or morph into another element, such as a toolbar or panel listing related actions.

### 52.4 The Snackbar

The Snackbar component provides a way to present the user with information as a panel at the bottom of the screen, as shown in Figure 52-2. Snackbar instances contain a brief text message and an optional action button that will perform a task when tapped by the user. Once displayed, a Snackbar will either timeout automatically or can be removed manually by the user via a swiping action. During the appearance of the Snackbar, the app will continue to function and respond to user interactions normally.

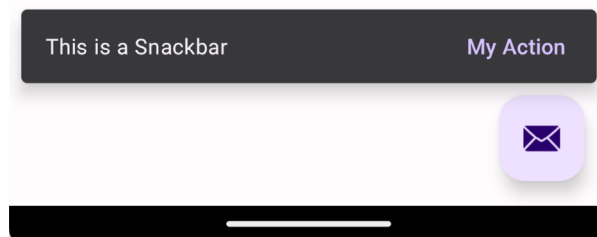


Figure 52-2

In the remainder of this chapter, an example application will be created that uses the basic features of the floating



action button and Snackbar to add entries to a list of items.

## 52.5 Creating the Example Project

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 *FabExample* into the Name field and specify *com.ebookfrenzy.fabexample* 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.

## 52.6 Reviewing the Project

Since the Basic Views Activity template was selected, the activity contains four layout files. The *activity\_main.xml* file consists of a CoordinatorLayout manager containing entries for an app bar, a Material toolbar, and a floating action button.

The *content\_main.xml* file represents the layout of the content area of the activity and contains a NavHostFragment instance. This file is embedded into the *activity\_main.xml* file via the following include directive:

```
<include layout="@layout/content_main" />
```

The floating action button element within the *activity\_main.xml* file reads as follows:

```
<com.google.android.material.floatingactionbutton.FloatingActionButton
    android:id="@+id/fab"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="bottom|end"
    android:layout_marginEnd="@dimen/fab_margin"
    android:layout_marginBottom="16dp"
    app:srcCompat="@android:drawable/ic_dialog_email" />
```

This declares that the button is to appear in the bottom right-hand corner of the screen with margins represented by the *fab\_margin* identifier in the *values/dimens.xml* file (which, in this case, is set to 16dp). The XML further declares that the interior icon for the button is to take the form of the standard drawable built-in email icon.

The blank template has also configured the floating action button to display a Snackbar instance when tapped by the user. The code to implement this can be found in the *onCreate()* method of the *MainActivity.kt* file and reads as follows:

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

The code accesses the floating action button via the view binding and adds an *onClick* listener handler to be called when the button is tapped. This method displays a Snackbar instance configured with a message but no actions.

When the project is compiled and run, the floating action button will appear at the bottom of the screen, as shown in Figure 52-3:

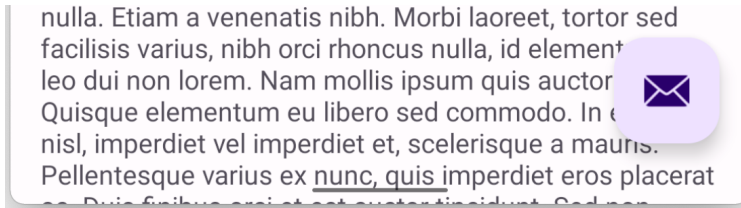


Figure 52-3

Tapping the floating action button will trigger the `onClickListener` handler method causing the Snackbar to appear at the bottom of the screen:

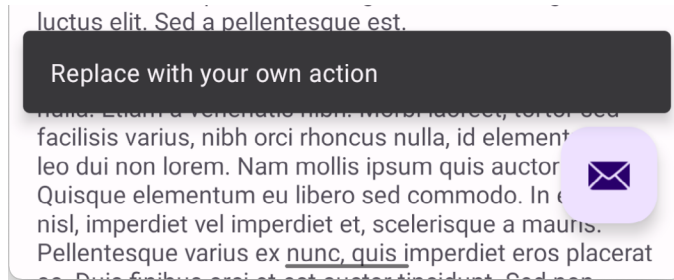


Figure 52-4

## 52.7 Removing Navigation Features

As “A Guide to the Android Studio Layout Editor Tool” outlines, the Basic Views Activity template contains multiple fragments and buttons to navigate from one fragment to the other. These features are unnecessary for this tutorial and will cause problems later if not removed. Before moving ahead with the tutorial, modify the project as follows:

1. 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.
2. Select the `SecondFragment` entry in the Component Tree panel within the editor and tap the keyboard delete key to remove it from the graph.
3. Locate and delete the `SecondFragment.kt` (`app -> java -> <package name> -> SecondFragment`) and `fragment_second.xml` (`app -> res -> layout -> fragment_second.xml`) files.
4. Locate the `FirstFragment.kt` file, double-click on it to load it into the editor, 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)
    }
}
```

## 52.8 Changing the Floating Action Button

Since the objective of this example is to configure the floating action button to add entries to a list, the email icon currently displayed on the button needs to be changed to something more indicative of the action being

performed. The icon that will be used for the button is named *ic\_add\_entry.png* and can be found in the *project\_icons* folder of the sample code download available from the following URL:

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

Locate this image in the file system navigator for your operating system and copy the image file. Right-click on the *app* -> *res* -> *drawable* entry in the Project tool window and select Paste from the menu to add the file to the folder:

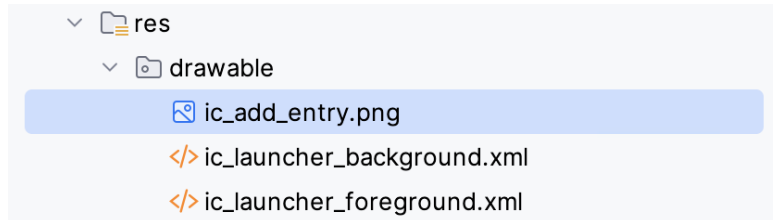


Figure 52-5

Next, edit the *activity\_main.xml* file and change the image source for the icon from *@android:drawable/ic\_dialog\_email* to *@drawable/ic\_add\_entry* as follows:

```
<com.google.android.material.floatingactionbutton.FloatingActionButton
    android:id="@+id/fab"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="bottom|end"
    android:layout_margin="@dimen/fab_margin"
    android:layout_marginBottom="16dp"
    app:srcCompat="@drawable/ic_add_entry" />
```

Within the layout preview, the interior icon for the button will have changed to a plus sign.

We can also make the floating action button do just about anything when clicked by adding code to the *OnClickListener*. The following changes to the *MainActivity.kt* file, for example, calls a method named *displayMessage()* to display a toast message each time the button is clicked:

```
.
.
import android.widget.Toast
.
.
binding.fab.setOnClickListener { view ->
    displayMessage("Fab clicked")
    Snackbar.make(view, "Replace with your own action", Snackbar.LENGTH_LONG)
        .setAction("Action", null).show()
}
.
.
fun displayMessage(message: String) {
    Toast.makeText(this@MainActivity, message, Toast.LENGTH_SHORT).show()
}
```

## 52.9 Adding an Action to the Snackbar

An action may also be added to the Snackbar, which performs a task when tapped by the user. Edit the *MainActivity.kt* file and modify the Snackbar creation code to add an action titled “My Action” configured with an `onClickListener` named *actionOnClickListener* which, in turn, displays a toast message:

```
binding.fab.setOnClickListener { view ->
    showMessage("FAB clicked")
    Snackbar.make(view, "Action complete", Snackbar.LENGTH_LONG)
        .setAction("My Action", actionOnClickListener).show()
}
```

Within the *MainActivity.kt* file, add the listener handler:

```
.
.
import android.view.View
.
.
var actionOnClickListener: View.OnClickListener = View.OnClickListener { view ->
    showMessage("Action clicked")
    Snackbar.make(view, "Action complete", Snackbar.LENGTH_LONG)
        .setAction("My Action", null).show()
}
```

Run the app and tap the floating action button, at which point both the toast message and Snackbar should appear. While the Snackbar is visible, tap the My Action button in the Snackbar and verify that the text on the Snackbar changes to “Action Complete”:

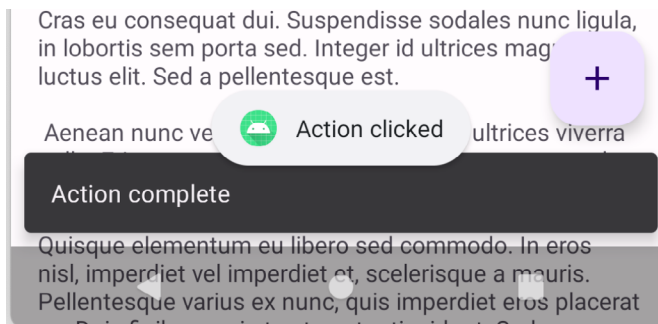


Figure 52-6

## 52.10 Summary

Before working through an example project that uses these features, this chapter has provided a general overview of material design, the floating action button, and the Snackbar.

The floating action button and the Snackbar are part of Android’s material design approach to user interface implementation. The floating action button provides a way to promote the most common action within a particular screen of an Android application. The Snackbar provides a way for an application to present information to the user and allow the user to act upon it.

# 60. Android Broadcast Intents and Broadcast Receivers

In addition to providing a mechanism for launching application activities, intents are also used to broadcast system-wide messages to other components on the system. This involves the implementation of Broadcast Intents and Broadcast Receivers, both of which are the topic of this chapter.

## 60.1 An Overview of Broadcast Intents

Broadcast intents are Intent objects that are broadcast via a call to the `sendBroadcast()`, `sendStickyBroadcast()`, or `sendOrderedBroadcast()` methods of the Activity class (the latter being used when results are required from the broadcast). In addition to providing a messaging and event system between application components, broadcast intents are also used by the Android system to notify interested applications about key system events (such as the external power supply or headphones being connected or disconnected).

When a broadcast intent is created, it must include an action string, optional data, and a category string. As with standard intents, data is added to a broadcast intent using key-value pairs in conjunction with the `putExtra()` method of the intent object. The optional category string may be assigned to a broadcast intent via a call to the `addCategory()` method.

The action string, which identifies the broadcast event, must be unique and typically uses the application's package name syntax. For example, the following code fragment creates and sends a broadcast intent, including a unique action string and data:

```
val intent = Intent()
intent.action = "com.example.Broadcast"
intent.putExtra("MyData", 1000)
sendBroadcast(intent)
```

The above code would successfully launch the corresponding broadcast receiver on an Android device earlier than 3.0. On more recent versions of Android, however, the broadcast receiver would not receive the intent. This is because Android 3.0 introduced a launch control security measure that prevents components of *stopped* applications from being launched via an intent. An application is considered to be in a stopped state if the application has either just been installed and not previously launched or been manually stopped by the user using the application manager on the device. To get around this, however, a flag can be added to the intent before it is sent to indicate that the intent is to be allowed to start a component of a stopped application. This flag is `FLAG_INCLUDE_STOPPED_PACKAGES` and would be used as outlined in the following adaptation of the previous code fragment:

```
val intent = Intent()
intent.action = "com.example.Broadcast"
intent.putExtra("MyData", 1000)
intent.flags = Intent.FLAG_INCLUDE_STOPPED_PACKAGES
sendBroadcast(intent)
```

## 60.2 An Overview of Broadcast Receivers

An application listens for specific broadcast intents by registering a *broadcast receiver*. Broadcast receivers are implemented by extending the Android `BroadcastReceiver` class and overriding the `onReceive()` method. The broadcast receiver may then be registered within code (for example, within an activity) or a manifest file. Part of the registration implementation involves the creation of intent filters to indicate the specific broadcast intents the receiver is required to listen for. This is achieved by referencing the *action string* of the broadcast intent. When a matching broadcast is detected, the `onReceive()` method of the broadcast receiver is called, at which point the method has 5 seconds to perform any necessary tasks before returning. It is important to note that a broadcast receiver does not need to run continuously. If a matching intent is detected, the Android runtime system automatically starts the broadcast receiver before calling the `onReceive()` method.

The following code outlines a template Broadcast Receiver subclass:

```
package com.ebookfrenzy.sendbroadcast

import android.content.BroadcastReceiver
import android.content.Context
import android.content.Intent

class MyReceiver : BroadcastReceiver() {

    override fun onReceive(context: Context, intent: Intent) {
        // TODO: This method is called when the BroadcastReceiver is receiving
        // an Intent broadcast.
        throw UnsupportedOperationException("Not yet implemented")
    }
}
```

When registering a broadcast receiver within a manifest file, a `<receiver>` entry must be added for the receiver.

The following example manifest file registers the above example broadcast receiver:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.broadcastdetector.broadcastdetector"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk android:minSdkVersion="17" />

    <application
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name" >
        <receiver android:name="MyReceiver" >
            </receiver>
        </application>
</manifest>
```

When running on versions of Android older than Android 8.0, the intent filters associated with a receiver can be placed within the receiver element of the manifest file as follows:

## 61. An Introduction to Kotlin Coroutines

When an Android application is first started, the runtime system creates a single thread in which all 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 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 typically results 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. This chapter 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 worrying 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). Though it does so efficiently, Kotlin still uses multi-threading behind the scenes.

### 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, much 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 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, the Kotlin runtime saves it, and another coroutine resumes to take its place. When the coroutine is resumed, it is 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, allowing them to be managed as groups instead of as individual ones. 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 the app no longer needs them). 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 built-in scopes and 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 tied to the entire application lifecycle. 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 Android applications. Coroutines running in `GlobalScope` are considered to be using *unstructured concurrency*.
- **ViewModelScope** – Provided specifically for `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 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 tasks 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. Using this builder, a coroutine running using the `Main` context could launch a child coroutine in the `Default` context. 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 must occur 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 coroutine will cancel all other coroutines.
- **supervisorScope** – Similar to the `coroutineScope` outlined above, except that a failure in one child does not result in the 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. For example, a `Job` and all of its children may 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, call the `cancelAndJoin()`

method.

This hierarchical Job structure, together with coroutine scopes, form the foundation of structured concurrency, which aims to ensure that coroutines do not run longer than required without manually keeping references to each coroutine.

## 61.8 Coroutines – Suspending and Resuming

It helps to see some coroutine examples in action to understand coroutine suspension better. To start with, let's assume a simple Android app containing a button that, when clicked, calls a function named *startTask()*. This function calls 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. The built-in Kotlin *delay()* function is implemented as a suspend function, so it 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:

## 70. An Android TableLayout and TableRow Tutorial

When the work began on the next chapter of this book (*“An Android Room Database and Repository Tutorial”*), it was originally intended to include the steps to design the user interface layout for the Room database example application. It quickly became evident, however, that the best way to implement the user interface was to use the Android TableLayout and TableRow views and that this topic area deserved a self-contained chapter. As a result, this chapter will focus solely on the user interface design of the database application to be completed in the next chapter, and in doing so, take some time to introduce the basic concepts of table layouts in Android Studio.

### 70.1 The TableLayout and TableRow Layout Views

The TableLayout container view allows user interface elements to be organized on the screen in a table format consisting of rows and columns. Each row within a TableLayout is occupied by a TableRow instance which, in turn, is divided into cells, with each cell containing a single child view (which may be a container with multiple view children).

The number of columns in a table is dictated by the row with the most columns, and, by default, the width of each column is defined by the widest cell in that column. Columns may be configured to be shrinkable or stretchable (or both) such that they change in size relative to the parent TableLayout. In addition, a single cell may be configured to span multiple columns.

Consider the user interface layout shown in Figure 70-1:

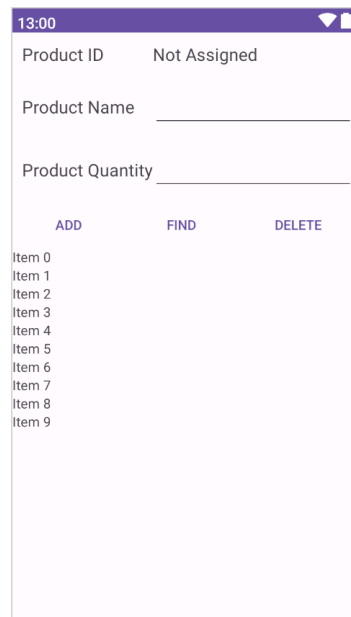


Figure 70-1

From the visual appearance of the layout, it is difficult to identify the TableLayout structure used to design the interface. The hierarchical tree illustrated in Figure 70-2, however, makes the structure a little easier to understand:

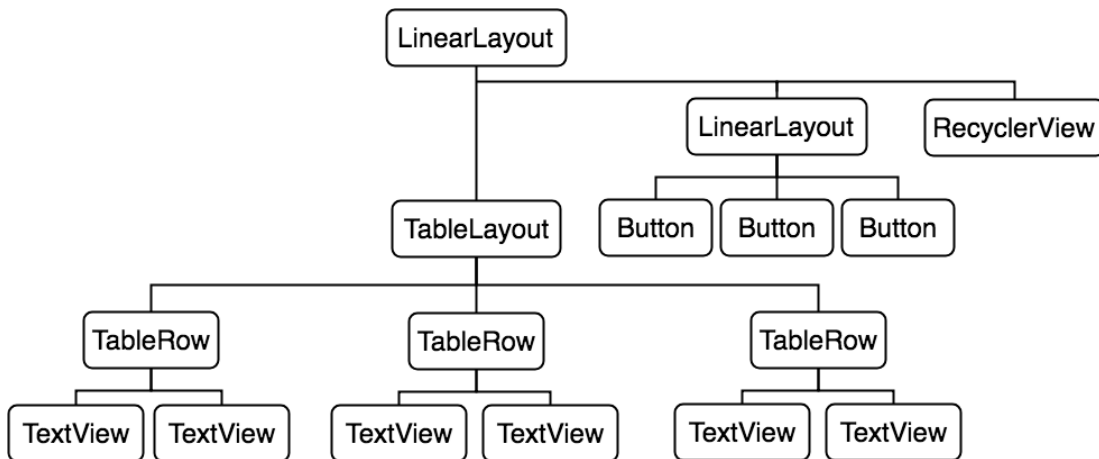


Figure 70-2

The layout comprises a parent LinearLayout view with TableLayout, LinearLayout, and RecyclerView children. The TableLayout contains three TableRow children representing three rows in the table. The TableRows contain two child views, each representing the contents of a table column cell. The LinearLayout child view contains three Button children.

The layout shown in Figure 70-2 is the exact layout required for the database example that will be completed in the next chapter. Therefore, the remainder of this chapter will be used to work step by step through the design of this user interface using the Android Studio Layout Editor tool.

## 70.2 Creating the Room Database Project

Select the *New Project* menu 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 *RoomDemo* into the Name field and specify *com.ebookfrenzy.roomdemo* 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.

Migrate the project to view binding using the steps outlined in section 18.8 *Migrating a Project to View Binding*.

## 70.3 Converting to a LinearLayout

Locate the *activity\_main.xml* file in the Project tool window (*app -> res -> layout*) and double-click on it to load it into the Layout Editor tool. By default, Android Studio has used a ConstraintLayout as the root layout element in the user interface. This needs to be converted to a vertically oriented LinearLayout.

With the Layout Editor tool in Design mode, locate the ConstraintLayout component in the Component Tree panel, right-click on it to display the menu shown in Figure 70-3, and select the *Convert view...* option:

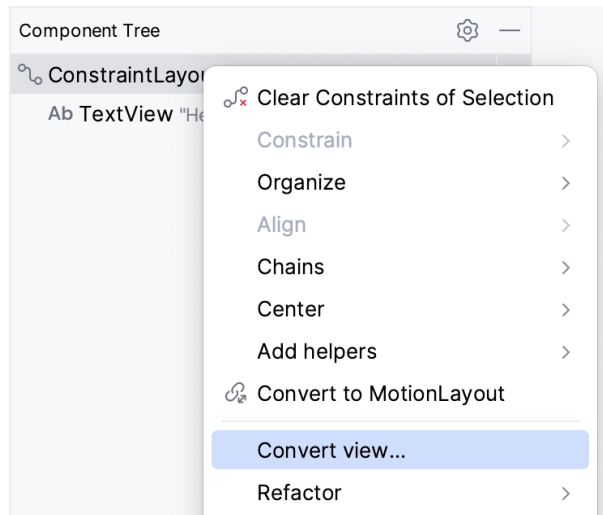


Figure 70-3

In the resulting dialog (Figure 70-4), select the option to convert to a LinearLayout before clicking on the Apply button:

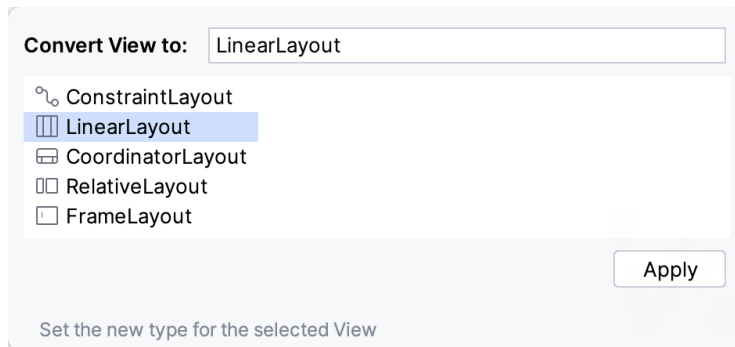


Figure 70-4

By default, the layout editor will have converted the ConstraintLayout to a horizontal LinearLayout, so select the layout component in the Component Tree window, refer to the Attributes tool window, and change the orientation property to *vertical*:

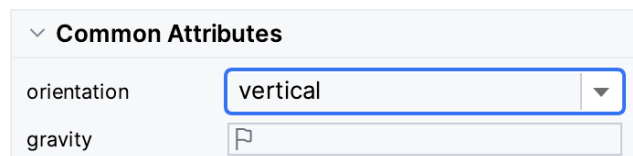


Figure 70-5

With the conversion complete, select and delete the default TextView widget from the layout.

## 70.4 Adding the TableLayout to the User Interface

Remaining in the `activity_main.xml` file and referring to the Layouts category of the Palette, drag a TableLayout view to position it at the top of the LinearLayout canvas area.

Once these initial steps are complete, the Component Tree for the layout should resemble that shown in Figure 70-6.

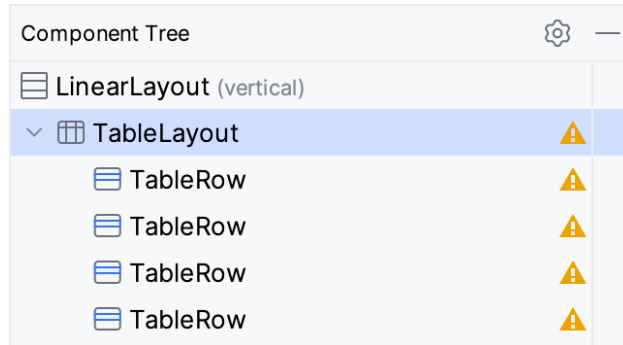


Figure 70-6

Android Studio has automatically added four TableRow instances to the TableLayout. Since only three rows are required for this example, select and delete the fourth TableRow instance. Additional rows may be added to the TableLayout at any time by dragging the TableRow object from the palette and dropping it onto the TableLayout entry in the Component Tree tool window.

With the TableLayout selected, use the Attributes tool window to change the layout\_height property to wrap\_content and layout\_width to match\_parent.

## 70.5 Configuring the TableRows

From within the Text section of the palette, drag two TextView objects onto the uppermost TableRow entry in the Component Tree (Figure 70-7):

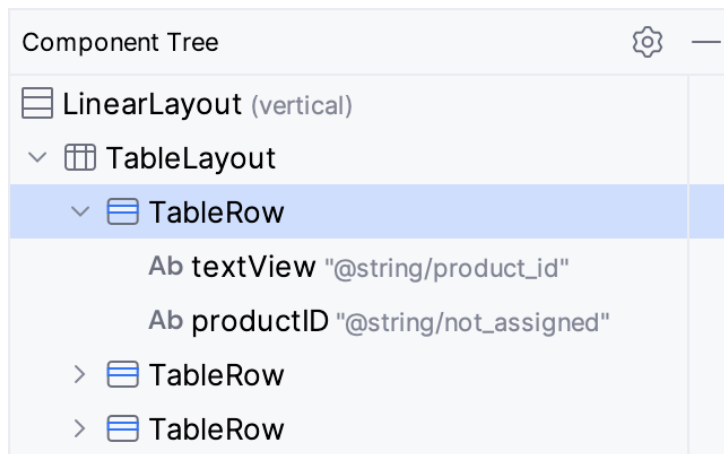


Figure 70-7

Select the left-most TextView within the screen layout and change the text property to “Product ID” in the Attributes tool window. Repeat this step for the rightmost TextView, changing the text to “Not assigned” and specifying an ID value of *productID*.

Drag and drop another TextView widget onto the second TableRow entry in the Component Tree and change the text on the view to read “Product Name”. Locate the Plain Text object in the palette and drag and drop it to position it beneath the Product Name TextView within the Component Tree as outlined in Figure 70-8. Next,

delete the “Name” string from the text property and set the ID to *productName*.

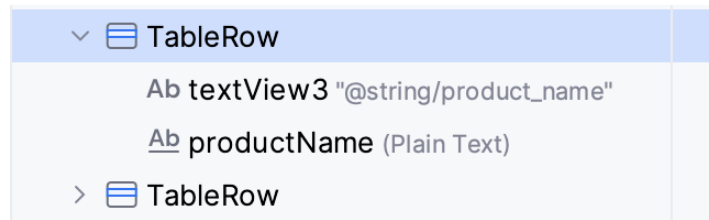


Figure 70-8

Drag and drop another TextView and a Number (Decimal) Text Field onto the third TableRow to position the TextView above the EditText in the hierarchy. Change the text on the TextView to “Product Quantity” and the ID of the EditText object to *productQuantity*.

Shift-click to select all of the widgets in the layout as shown in Figure 70-9 below, and use the Attributes tool window to set the *textSize* property on all of the objects to 18sp:



Figure 70-9

## 70.6 Adding the Button Bar to the Layout

The next step is to add a *LinearLayout (Horizontal)* view to the parent *LinearLayout* view, positioned immediately below the *TableLayout* view. Begin by clicking on the small disclosure arrow to the left of the *TableLayout* entry in the Component Tree so that the *TableRows* are folded away from view. Drag a *LinearLayout (horizontal)* instance from the *Layouts* section of the Layout Editor palette, drop it immediately beneath the *TableLayout* entry in the Component Tree panel, and change the *layout\_height* property to *wrap\_content*:

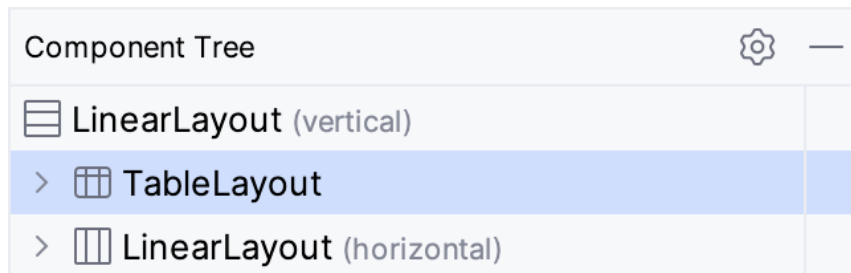


Figure 70-10

Drag three Button objects onto the new *LinearLayout* and assign string resources for each button that read “Add”, “Find” and “Delete” respectively. Buttons in this type of button bar arrangement should generally be displayed with a borderless style. Use the Attributes tool window for each button to change the *style* setting to *Widget.AppCompat.Button.Borderless* and the *textColor* attribute to *?attr/colorPrimary*. Change the IDs for the buttons to *addButton*, *findButton*, and *deleteButton*, respectively.

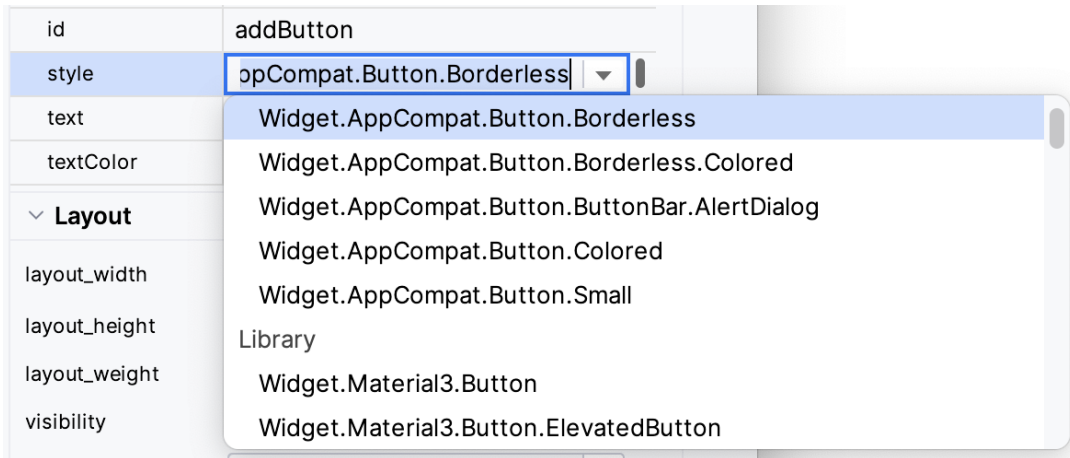


Figure 70-11

With the new horizontal `LinearLayout` view selected in the Component Tree, change the gravity property to `center_horizontal` so that the buttons are centered horizontally within the display. Before proceeding, extract all of the text properties added in the above steps to string resources.

## 70.7 Adding the RecyclerView

In the Component Tree, click on the disclosure arrow to the left of the newly added horizontal `LinearLayout` entry to fold all the children from view.

From the Containers section of the Palette, drag a `RecyclerView` instance onto the Component Tree to position it beneath the button bar `LinearLayout` as shown in Figure 70-12. Ensure the `RecyclerView` is added as a direct child of the parent vertical `LinearLayout` view and not as a child of the horizontal button bar `LinearLayout`.

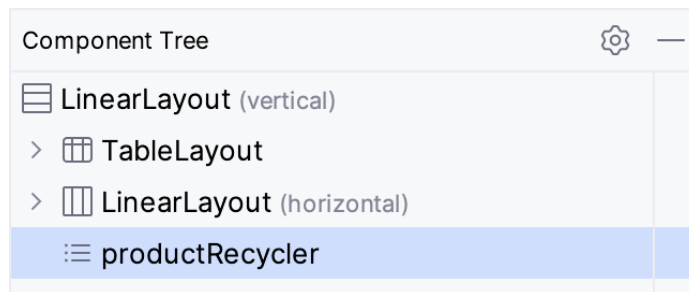


Figure 70-12

With the `RecyclerView` selected in the layout, change the ID of the view to `product_recycler` and set the `layout_height` property to `match_parent`. Before proceeding, check that the hierarchy of the layout in the Component Tree panel matches that shown in the following figure:



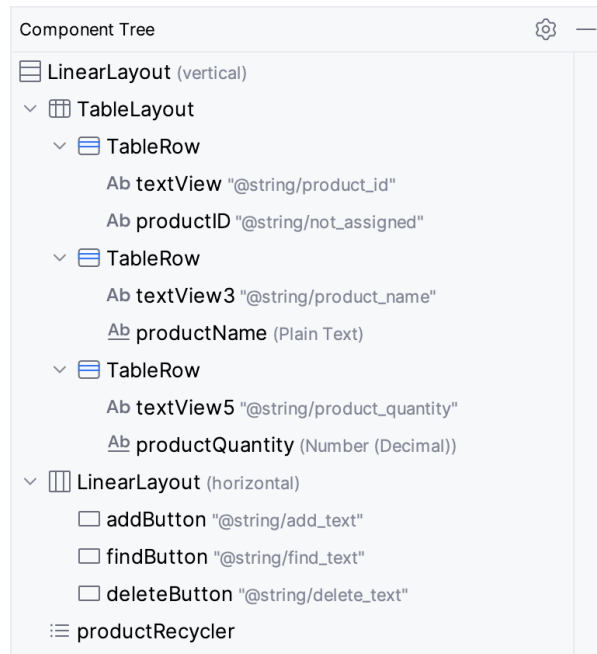


Figure 70-13

## 70.8 Adjusting the Layout Margins

All that remains is to adjust some of the layout settings. Begin by clicking on the first TableRow entry in the Component Tree panel so that it is selected. Hold down the Cmd/Ctrl-key on the keyboard and click on the second and third TableRows, the horizontal LinearLayout, and the RecyclerView so that all five items are selected. In the Attributes panel, locate the *layout\_margin* attributes category and, once located, change the value to 10dp as shown in Figure 70-14:

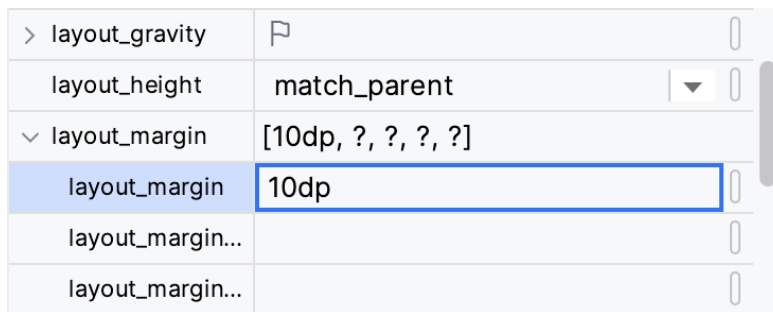


Figure 70-14

With margins set, the user interface should appear as illustrated in Figure 70-1.

## 70.9 Summary

The Android TableLayout container view provides a way to arrange view components in a row and column configuration. While the TableLayout view provides the overall container, each row and the cells contained therein are implemented via instances of the TableRow view. In this chapter, a user interface has been designed in Android Studio using the TableLayout and TableRow containers. The next chapter will add the functionality behind this user interface to implement the SQLite database capabilities using a repository and the Room persistence library.



## 72. Video Playback on Android using the VideoView and MediaController Classes

One of the primary uses for smartphones and tablets is to provide access to online content. Video is a key form of content widely used, especially on tablet devices.

The Android SDK includes two classes that make implementing video playback on Android devices extremely easy to implement when developing applications. This chapter will provide an overview of these two classes, VideoView and MediaController, creating a video playback application.

### 72.1 Introducing the Android VideoView Class

The simplest way to display video within an Android application is to use the VideoView class. This visual component provides a surface on which a video may be played when added to the layout of an activity. Android currently supports the following video formats:

- H.263
- H.264 AVC
- H.265 HEVC
- MPEG-4 SP
- VP8
- VP9

The VideoView class has a wide range of methods that may be called to manage video playback. Some of the more commonly used methods are as follows:

- **setVideoPath(String path)** – Specifies the video media path (as a string) to be played. This can be either a remote video file URL or a video file local to the device.
- **setVideoUri(Uri uri)** – Performs the same task as the *setVideoPath()* method but takes a Uri object as an argument instead of a string.
- **start()** – Starts video playback.
- **stopPlayback()** – Stops the video playback.
- **pause()** – Pauses video playback.
- **isPlaying()** – Returns a Boolean value indicating whether a video is playing.
- **setOnPreparedListener(MediaPlayer.OnPreparedListener)** – Allows a callback method to be called when the video is ready to play.

- **setOnErrorListener(MediaPlayer.OnErrorListener)** - Allows a callback method to be called when an error occurs during the video playback.
- **setOnCompletionListener(MediaPlayer.OnCompletionListener)** - Allows a callback method to be called when the end of the video is reached.
- **getDuration()** - Returns the duration of the video. Will typically return -1 unless called from within the *OnPreparedListener()* callback method.
- **getCurrentPosition()** - Returns an integer value indicating the current position of playback.
- **setMediaController(MediaController)** - Designates a MediaController instance allowing playback controls to be displayed to the user.

### 72.2 Introducing the Android MediaController Class

If a video is played using the VideoView class, the user will not be given any control over the playback, which will run until the end of the video is reached. This issue can be addressed by attaching an instance of the MediaController class to the VideoView instance. The MediaController will then provide a set of controls allowing the user to manage the playback (such as pausing and seeking backward/forwards in the video timeline).

The position of the controls is designated by anchoring the controller instance to a specific view in the user interface layout. Once attached and anchored, the controls will appear briefly when playback starts and may subsequently be restored at any point by the user tapping on the view to which the instance is anchored.

Some of the key methods of this class are as follows:

- **setAnchorView(View view)** - Designates the view to which the controller will be anchored. This designates the location of the controls on the screen.
- **show()** - Displays the controls.
- **show(int timeout)** - Controls are displayed for the designated duration (in milliseconds).
- **hide()** - Hides the controller from the user.
- **isShowing()** - Returns a Boolean value indicating whether the controls are currently visible to the user.

### 72.3 Creating the Video Playback Example

The remainder of this chapter will create an example application that uses the VideoView and MediaController classes to play an MPEG-4 video file.

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 *VideoPlayer* into the Name field and specify *com.ebookfrenzy.videoplayer* as the package name. Before clicking on the Finish button, change the Minimum API level setting to API 33: Android 13 (Tiramisu) and the Language menu to Kotlin. Use the steps in section 18.8 *Migrating a Project to View Binding* to enable view binding for the project.

### 72.4 Designing the VideoPlayer Layout

The user interface for the main activity will consist solely of an instance of the VideoView class. Use the Project tool window to locate the *app -> res -> layout -> activity\_main.xml* file, double-click on it, switch the Layout Editor tool to Design mode, and delete the default TextView widget.

From the Widgets category of the Palette panel, drag and drop a VideoView instance onto the layout to fill the available canvas area, as shown in Figure 72-1. Using the Attributes panel, change the `layout_width` and `layout_height` attributes to `match_constraint` and `wrap_content`, respectively. Also, remove the constraint connecting the bottom of the VideoView to the bottom of the parent ConstraintLayout. Finally, change the ID of the component to `videoView1`.

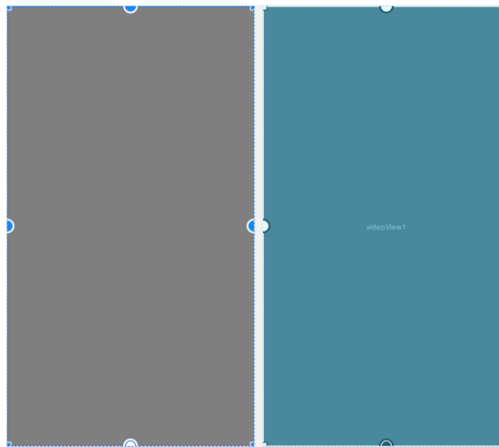


Figure 72-1

## 72.5 Downloading the Video File

The video that will be played by the VideoPlayer app is a short animated movie clip encoded in MPEG-4 format. Using a web browser, navigate to the following URL to play the video:

[https://www.ebookfrenzy.com/android\\_book/demo.mp4](https://www.ebookfrenzy.com/android_book/demo.mp4)

Staying within the browser window, right-click on the video playback, select the option to save or download the video to a local file, and choose a suitable temporary filesystem location, naming the file `demo.mp4`.

Within Android Studio, locate the `res` folder in the Project tool window, right-click on it, select the `New -> Directory` menu option, and enter `raw` into the name field before pressing the Return key. Using the filesystem navigator for your operating system, locate the `demo.mp4` file downloaded above and copy it. Returning to Android Studio, right-click on the newly created `raw` directory and select the `Paste` option to copy the video file into the project. Once added, the `raw` folder should match Figure 72-2 within the Project tool window:

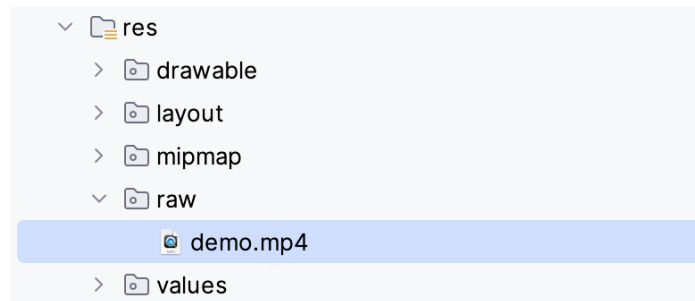


Figure 72-2

## 72.6 Configuring the VideoView

The next step is configuring the VideoView with the video path to be played and then starting the playback. This will be performed when the main activity has initialized, so load the `MainActivity.kt` file into the editor and

## Video Playback on Android using the VideoView and MediaController Classes

modify it as outlined in the following listing:

```
package com.ebookfrenzy.videoplayer
.
.
import android.net.Uri

class MainActivity : AppCompatActivity() {

    private lateinit var binding: ActivityMainBinding

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

        configureVideoView()
    }

    private fun configureVideoView() {

        binding.videoView1.setVideoURI(Uri.parse("android.resource://"
            + packageName + "/" + R.raw.demo))

        binding.videoView1.start()
    }
}
```

This code obtains a reference to the VideoView instance in the layout, assigns to it a URI object referencing the movie file located in the raw resource directory, and then starts the video playing.

Test the application by running it on an emulator or physical Android device. After the application launches, there may be a short delay while video content is buffered before the playback begins (Figure 72-3).

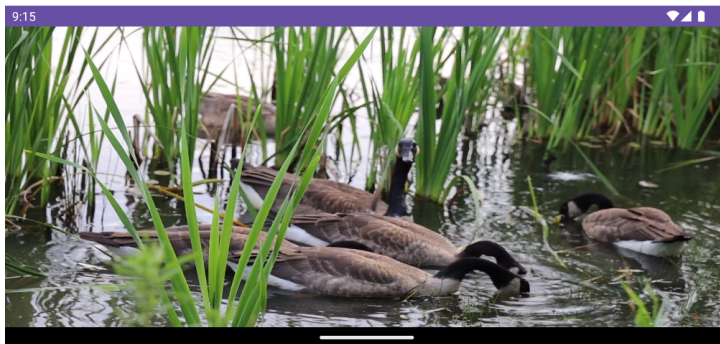


Figure 72-3

This shows how easy it can be to integrate video playback into an Android application. Everything in this example has been achieved using a VideoView instance and three lines of code.

## 72.7 Adding the MediaController to the Video View

As the VideoPlayer application currently stands, there is no way for the user to control playback. As previously outlined, this can be achieved using the MediaController class. To add a controller to the VideoView, modify the *configureVideoView()* method once again:

```
package com.ebookfrenzy.videoplayer
.
.
import android.widget.MediaController
.
.
class MainActivity : AppCompatActivity() {

    private var mediaController: MediaController? = null
    .
    .
    private fun configureVideoView() {

        binding.videoView1.setVideoURI(Uri.parse("android.resource://"
            + packageName + "/" + R.raw.demo))

        mediaController = MediaController(this)
        mediaController?.setAnchorView(binding.videoView1)
        binding.videoView1.setMediaController(mediaController)
        binding.videoView1.start()
    }
}
```

When the application is launched with these changes implemented, tapping the VideoView canvas will cause the media controls to appear over the video playback. These controls should include a Seekbar and fast forward, rewind, and play/pause buttons. After the controls recede from view, they can be restored anytime by tapping on the VideoView canvas again. With just three more lines of code, our video player application now has media controls, as shown in Figure 72-4:

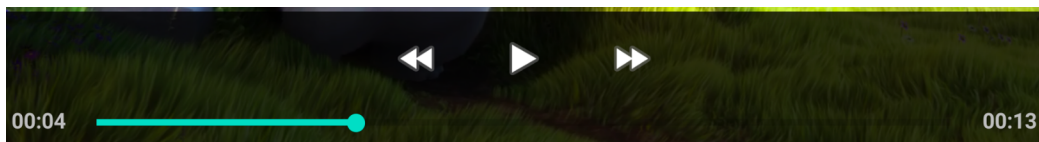


Figure 72-4

## 72.8 Setting up the onPreparedListener

As a final example of working with video-based media, the activity will be extended further to demonstrate the mechanism for configuring a listener. In this case, a listener will be implemented that is intended to output the duration of the video as a message in the Android Studio Logcat panel. The listener will also configure video playback to loop continuously:

```
package com.ebookfrenzy.videoplayer
.
.
.
```

```
import android.util.Log
.
.
class MainActivity : AppCompatActivity() {

    private var TAG = "VideoPlayer"
.
.
    private fun configureVideoView() {

        binding.videoView1.setVideoURI(Uri.parse("android.resource://"
            + packageName + "/" + R.raw.demo))

        mediaController = MediaController(this)
        mediaController?.setAnchorView(binding.videoView1)
        binding.videoView1.setMediaController(mediaController)

        binding.videoView1.setOnPreparedListener { mp ->
            mp.isLooping = true
            Log.i(TAG, "Duration = " + binding.videoView1.duration)
        }
        binding.videoView1.start()
    }
}
```

Now just before the video playback begins, a message will appear in the Android Studio Logcat panel that reads along the lines of the following, and the video will restart after playback ends:

```
2023-06-27 09:25:41.313 3050-3050 VideoPlayer com.ebookfrenzy.videoplayer
I Duration = 25365
```

## 72.9 Summary

Android devices make excellent platforms for the delivery of content to users, particularly in the form of video media. As outlined in this chapter, the Android SDK provides two classes, namely VideoView and MediaController, which combine to make video playback integration into Android applications quick and easy, often involving just a few lines of Kotlin code.



## 73. Android Picture-in-Picture Mode

When multitasking in Android was covered in earlier chapters, Picture-in-picture (PiP) mode was mentioned briefly but not covered in any detail. Intended primarily for video playback, PiP mode allows an activity screen to be reduced in size and positioned at any location on the screen. While in this state, the activity continues to run, and the window remains visible regardless of any other activities running on the device. This allows the user to, for example, continue watching video playback while performing tasks such as checking email or working on a spreadsheet.

This chapter will provide an overview of Picture-in-Picture mode before Picture-in-Picture support is added to the VideoPlayer project in the next chapter.

### 73.1 Picture-in-Picture Features

As explained later in the chapter and demonstrated in the next chapter, an activity is placed into PiP mode via an API call from within the running app. When placed into PiP mode, configuration options may be specified that control the aspect ratio of the PiP window and also define the area of the activity screen to be included. Figure 73-1, for example, shows a video playback activity in PiP mode:

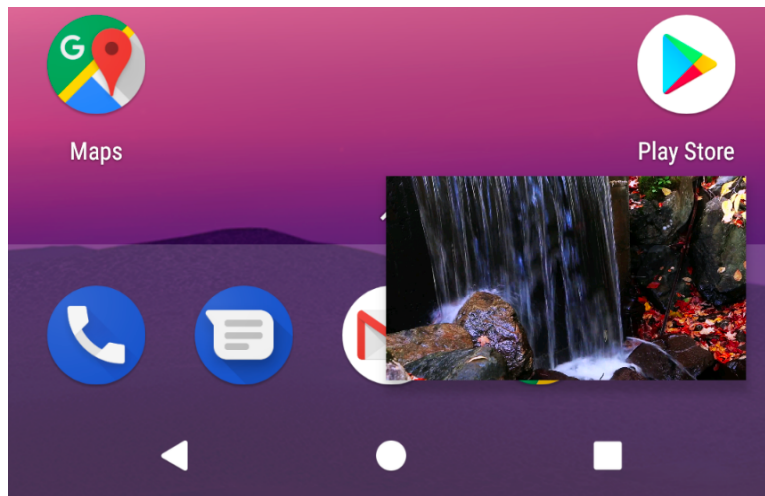


Figure 73-1

Figure 73-2 shows a PiP mode window after the user has tapped it. When in this mode, the window appears larger and includes a full-screen action in the center which, when tapped, restores the window to full-screen mode and an exit button in the top right-hand corner to close the window and place the app in the background. When displayed in this mode, any custom actions added to the PiP window will appear on the screen. In the case of Figure 73-2, the PiP window includes custom play and pause action buttons:

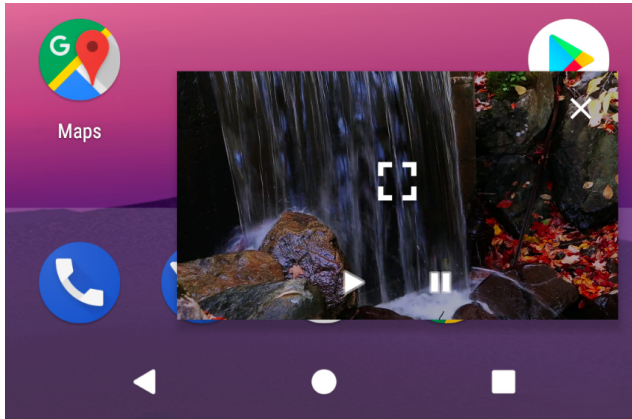


Figure 73-2

The remainder of this chapter will outline how PiP mode is enabled and managed from within an Android app.

## 73.2 Enabling Picture-in-Picture Mode

PiP mode is currently only supported on devices running API 26: Android 8.0 (Oreo) or newer. The first step in implementing PiP mode is to enable it within the project's manifest file. PiP mode is configured on a per-activity basis by adding the following lines to each activity element for which PiP support is required:

```
<activity android:name=".MyActivity"
    android:supportsPictureInPicture="true"
    android:configChanges=
        "screenSize|smallestScreenSize|screenLayout|orientation"
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

The *android:supportsPictureInPicture* entry enables PiP for the activity, while the *android:configChanges* property notifies Android that the activity can handle layout configuration changes. Without this setting, each time the activity moves in and out of PiP mode, the activity will be restarted, resulting in playback restarting from the beginning of the video during the transition.

## 73.3 Configuring Picture-in-Picture Parameters

PiP behavior is defined through the use of the `PictureInPictureParams` class, instances of which can be created using the `Builder` class as follows:

```
val params = PictureInPictureParams.Builder().build()
```

The above code creates a default `PictureInPictureParams` instance with special parameters defined. The following optional method calls may also be used to customize the parameters:

- **setActions()** – Used to define actions that can be performed within the PiP window while the activity is in PiP mode. Actions will be covered in more detail later in this chapter.
- **setAspectRatio()** – Declares the preferred aspect ratio for the appearance of the PiP window. This method takes as an argument a `Rational` object containing the height width/height ratio.

- **setSourceRectHint()** – Takes as an argument a Rect object defining the area of the activity screen to be displayed within the PiP window.

The following code, for example, configures aspect ratio and action parameters within a `PictureInPictureParams` object. In the case of the aspect ratio, this is defined using the width and height dimensions of a `VideoView` instance:

```
val rational = Rational(videoView.width,
    videoView.height)

val params = PictureInPictureParams.Builder()
    .setAspectRatio(rational)
    .setActions(actions)
    .build()
```

Once defined, PiP parameters may be set at any time using the `setPictureInPictureParams()` method as follows:

```
setPictureInPictureParams(params)
```

Parameters may also be specified when entering PiP mode.

### 73.4 Entering Picture-in-Picture Mode

An activity is placed into Picture-in-Picture mode via a call to the `enterPictureInPictureMode()` method, passing through a `PictureInPictureParams` object:

```
enterPictureInPictureMode(params)
```

If no parameters are required, create a default `PictureInPictureParams` object as outlined in the previous section. If parameters have previously been set using the `setPictureInPictureParams()` method, these parameters are combined with those specified during the `enterPictureInPictureMode()` method call.

### 73.5 Detecting Picture-in-Picture Mode Changes

When an activity enters PiP mode, it is important to hide unnecessary views so that only the video playback is visible within the PiP window. When an activity enters PiP mode, it is important to hide unnecessary views so that only the video playback is visible within the PiP window. When the activity re-enters full-screen mode, hidden user interface components must be reinstated. These and other app-specific tasks can be performed by overriding the `onPictureInPictureModeChanged()` method. When added to the activity, this method is called each time the activity transitions between PiP and full-screen modes and is passed a Boolean value indicating whether the activity is currently in PiP mode:

```
override fun onPictureInPictureModeChanged(
    isInPictureInPictureMode: Boolean, newConfig: Configuration) {
    super.onPictureInPictureModeChanged(isInPictureInPictureMode, newConfig)
    if (isInPictureInPictureMode) {
        // Activity entered Picture-in-Picture mode
    } else {
        // Activity entered full-screen mode
    }
}
```

### 73.6 Adding Picture-in-Picture Actions

Picture-in-Picture actions appear as icons within the PiP window when the user taps it. Implementing PiP actions is a multi-step process that begins with implementing a way for the PiP window to notify the activity

## Android Picture-in-Picture Mode

that an action has been selected. This is achieved by setting up a broadcast receiver within the activity and then creating a pending intent within the PiP action, which, in turn, is configured to broadcast an intent for which the broadcast receiver is listening. When the intent triggers the broadcast receiver, the data stored in the intent can be used to identify the action performed and to take the necessary action within the activity.

PiP actions are declared using the `RemoteAction` instances, initialized with an icon, a title, a description, and the `PendingIntent` object. Once one or more actions have been created, they are added to an `ArrayList` and passed through to the `setActions()` method while building a `PictureInPictureParams` object.

The following code fragment demonstrates the creation of the `Intent`, `PendingIntent`, and `RemoteAction` objects together with a `PictureInPictureParams` instance which is then applied to the activity's PiP settings:

```
val actions = ArrayList<RemoteAction>()

val actionIntent = Intent("MY_PIP_ACTION")

val pendingIntent = PendingIntent.getBroadcast(this@MyActivity,
                                             REQUEST_CODE, actionIntent,
                                             FLAG_IMMUTABLE)

val icon = Icon.createWithResource(this, R.drawable.action_icon)

val remoteAction = RemoteAction(icon,
                                "My Action Title",
                                "My Action Description",
                                pendingIntent)

actions.add(remoteAction)

val params = PictureInPictureParams.Builder()
    .setActions(actions)
    .build()

setPictureInPictureParams(params)
```

## 73.7 Summary

Picture-in-Picture mode is a multitasking feature introduced with Android 8.0 designed specifically to allow video playback to continue in a small window while the user performs tasks in other apps and activities. Before PiP mode can be used, it must first be enabled within the manifest file for those activities that require PiP support.

PiP mode behavior is configured using instances of the `PictureInPictureParams` class and initiated via a call to the `enterPictureInPictureMode()` method from within the activity. When in PiP mode, only the video playback should be visible, requiring that any other user interface elements be hidden until full-screen mode is selected. These and other mode transition-related tasks can be performed by overriding the `onPictureInPictureModeChanged()` method.

PiP actions appear as icons overlaid onto the PiP window when the user taps it. When selected, these actions trigger behavior within the activity. The PiP window uses broadcast receivers and pending intents to notify the activity of an action.

# 74. An Android Picture-in-Picture Tutorial

Following the previous chapters, this chapter will take the existing VideoPlayer project and enhance it to add Picture-in-Picture support, including detecting PiP mode changes and adding a PiP action designed to display information about the currently running video.

## 74.1 Adding Picture-in-Picture Support to the Manifest

The first step in adding PiP support to an Android app project is to enable it within the project Manifest file. Open the *manifests -> AndroidManifest.xml* file and modify the activity element to enable PiP support:

```

.
.
<activity
    android:name=".MainActivity"
    android:supportsPictureInPicture="true"
    android:configChanges="screenSize|smallestScreenSize|screenLayout|orientation"
    android:exported="true">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
.
.

```

## 74.2 Adding a Picture-in-Picture Button

As currently designed, the layout for the VideoPlayer activity consists solely of a VideoView instance. As currently designed, the layout for the VideoPlayer activity consists solely of a VideoView instance. A button will now be added to the layout to switch to PiP mode. Load the *activity\_main.xml* file into the layout editor and drag a Button object from the palette onto the layout so that it is positioned as shown in Figure 74-1:



Figure 74-1

Change the text on the button to read “Enter PiP Mode” and extract the string to a resource named *enter\_pip\_mode*. Before moving on to the next step, change the ID of the button to *pipButton* and configure the *onClick* attribute to call a method named *enterPipMode*.

## 74.3 Entering Picture-in-Picture Mode

The `enterPipMode` onClick callback method must now be added to the `MainActivity.kt` class file. Locate this file, open it in the code editor, and add this method as follows:

```

.
.
import android.app.PictureInPictureParams
import android.util.Rational
import android.view.View
import android.content.res.Configuration
.
.
fun enterPipMode(view: View) {

    val rational = Rational(binding.videoView1.width,
        binding.videoView1.height)

    val params = PictureInPictureParams.Builder()
        .setAspectRatio(rational)
        .build()

    binding.pipButton.visibility = View.INVISIBLE
    binding.videoView1.setMediaController(null)
    enterPictureInPictureMode(params)
}

```

The method begins by obtaining a reference to the Button view, then creates a Rational object containing the width and height of the VideoView. A set of Picture-in-Picture parameters is then created using the PictureInPictureParams Builder, passing through the Rational object as the aspect ratio for the video playback. Since the button does not need to be visible while the video is in PiP mode, it is invisible. The video playback controls are also hidden, so the video view will be unobstructed while in PiP mode.

Compile and run the app on a device or emulator running Android version 8 or newer and wait for video playback to begin before clicking on the PiP mode button. The video playback should minimize and appear in the PiP window as shown in :

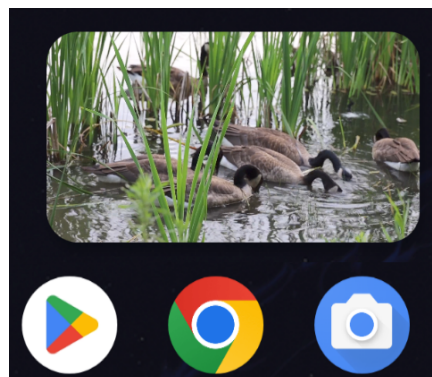


Figure 74-2

Click in the PiP window, then click within the full-screen mode markers that appear in the center of the window. Although the activity returns to full-screen mode, the button and media playback controls remain hidden.

Clearly, some code must be added to the project to detect when PiP mode changes occur within the activity.

## 74.4 Detecting Picture-in-Picture Mode Changes

As discussed in the previous chapter, PiP mode changes are detected by overriding the `onPictureInPictureModeChanged()` method within the affected activity. In this case, the method must be written to detect whether the activity is entering or exiting PiP mode and to take appropriate action to re-activate the PiP button and the playback controls. Remaining within the `MainActivity.kt` file, add this method now:

```
override fun onPictureInPictureModeChanged(
    isInPictureInPictureMode: Boolean, newConfig: Configuration) {
    super.onPictureInPictureModeChanged(isInPictureInPictureMode, newConfig)
    if (isInPictureInPictureMode) {

    } else {
        binding.pipButton.visibility = View.VISIBLE
        binding.videoView1.setMediaController(mediaController)
    }
}
```

When the method is called, it is passed a Boolean value indicating whether the activity is now in PiP mode. The code in the above method checks this value to decide whether to show the PiP button and to re-activate the playback controls.

## 74.5 Adding a Broadcast Receiver

The final step in the project is to add an action to the PiP window. The purpose of this action is to display a Toast message containing the name of the currently playing video. This will require some communication between the PiP window and the activity. One of the simplest ways to achieve this is to implement a broadcast receiver within the activity and use a pending intent to broadcast a message from the PiP window to the activity. Each time the activity enters PiP mode, these steps must be performed, so code must be added to the `onPictureInPictureModeChanged()` method. Locate this method now and begin by adding some code to create an intent filter and initialize the broadcast receiver:

```
.
.
import android.content.BroadcastReceiver
import android.content.Context
import android.content.Intent
import android.content.IntentFilter
import android.widget.Toast

class MainActivity : AppCompatActivity() {
.
.
    private val receiver: BroadcastReceiver? = null
.
.
override fun onPictureInPictureModeChanged(
```

```

    isInPictureInPictureMode: Boolean, newConfig: Configuration) {
    super.onPictureInPictureModeChanged(isInPictureInPictureMode, newConfig)
    if (isInPictureInPictureMode) {
        val filter = IntentFilter()
        filter.addAction(
            "com.ebookfrenzy.videoplayer.VIDEO_INFO")

        val receiver = object : BroadcastReceiver() {
            override fun onReceive(context: Context,
                intent: Intent) {
                Toast.makeText(context,
                    "Favorite Home Movie Clips",
                    Toast.LENGTH_LONG).show()
            }
        }

        registerReceiver(receiver, filter, Context.RECEIVER_EXPORTED)
    } else {
        binding.pipButton.visibility = View.VISIBLE
        binding.videoView1.setMediaController(mediaController)

        receiver?.let {
            unregisterReceiver(it)
        }
    }
}

```

## 74.6 Adding the PiP Action

With the broadcast receiver implemented, the next step is to create a `RemoteAction` object configured with an image to represent the action within the PiP window.

For this example, an image icon file named `ic_info_24dp.xml` will be used. This file can be found in the `project_icons` folder of the source code download archive available from the following URL:

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

Locate this icon file and copy and paste it into the `app -> res -> drawables` folder within the Project tool window:

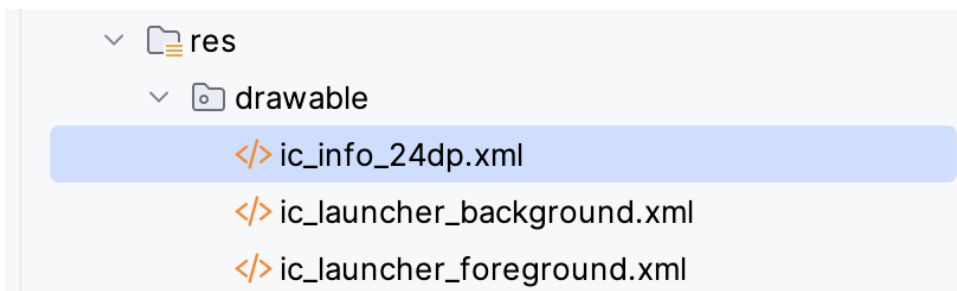


Figure 74-3



The next step is to create an Intent that will be sent to the broadcast receiver. This intent then needs to be wrapped up within a PendingIntent object, allowing the intent to be triggered later when the user taps the action button in the PiP window.

Edit the *MainActivity.kt* file to add a method to create the Intent and PendingIntent objects as follows:

```

.
.
import android.app.PendingIntent
import android.app.PendingIntent.FLAG_IMMUTABLE
.
.
class MainActivity : AppCompatActivity() {

    private val REQUEST_CODE = 101
.
.
    private fun createPipAction() {

        val actionIntent = Intent("com.ebookfrenzy.videoplayer.VIDEO_INFO")

        val pendingIntent = PendingIntent.getBroadcast(this@MainActivity,
            REQUEST_CODE, actionIntent, FLAG_IMMUTABLE)
    }
}

```

Now that both the Intent object and the PendingIntent instance in which it is contained have been created, a RemoteAction object needs to be created containing the icon to appear in the PiP window and the PendingIntent object. Remaining within the *createPipAction()* method, add this code as follows:

```

.
.
import android.app.RemoteAction
import android.graphics.drawable.Icon
.
.
private fun createPipAction() {

    val actions = ArrayList<RemoteAction>()

    val actionIntent = Intent("com.ebookfrenzy.videoplayer.VIDEO_INFO")

    val pendingIntent = PendingIntent.getBroadcast(this@MainActivity,
        REQUEST_CODE, actionIntent, FLAG_IMMUTABLE)

    val icon = Icon.createWithResource(this, R.drawable.ic_info_24dp)

    val remoteAction = RemoteAction(icon, "Info", "Video Info", pendingIntent)
}

```

## An Android Picture-in-Picture Tutorial

```
        actions.add(remoteAction)
    }
```

Now a `PictureInPictureParams` object containing the action needs to be created and the parameters applied so that the action appears within the PiP window:

```
private fun createPipAction() {

    val actions = ArrayList<RemoteAction>()

    val actionIntent = Intent("com.ebookfrenzy.videoplayer.VIDEO_INFO")

    val pendingIntent = PendingIntent.getBroadcast(this@MainActivity,
        REQUEST_CODE, actionIntent, FLAG_IMMUTABLE)

    val icon =
        Icon.createWithResource(this,
            R.drawable.ic_info_24dp)

    val remoteAction = RemoteAction(icon, "Info",
        "Video Info", pendingIntent)

    actions.add(remoteAction)

    val params = PictureInPictureParams.Builder()
        .setActions(actions)
        .build()

    setPictureInPictureParams(params)
}
```

The final task before testing the action is to make a call to the `createPipAction()` method when the activity enters PiP mode:

```
override fun onPictureInPictureModeChanged(
    isInPictureInPictureMode: Boolean, newConfig: Configuration) {
    super.onPictureInPictureModeChanged(isInPictureInPictureMode, newConfig)
    .
    .
    registerReceiver(receiver, filter, Context.RECEIVER_EXPORTED)
    createPipAction()
} else {
    pipButton.visibility = View.VISIBLE
    videoView1.setMediaController(mediaController)
    .
    .
```

## 74.7 Testing the Picture-in-Picture Action

Rerun the app and place the activity into PiP mode. Tap on the PiP window so that the new action button appears, as shown in Figure 74-4:

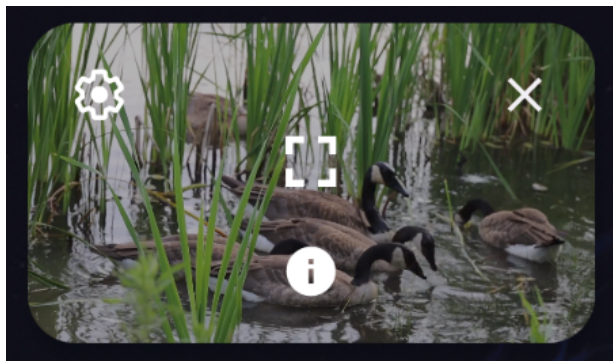


Figure 74-4

Click on the action button and wait for the Toast message to appear, displaying the name of the video:

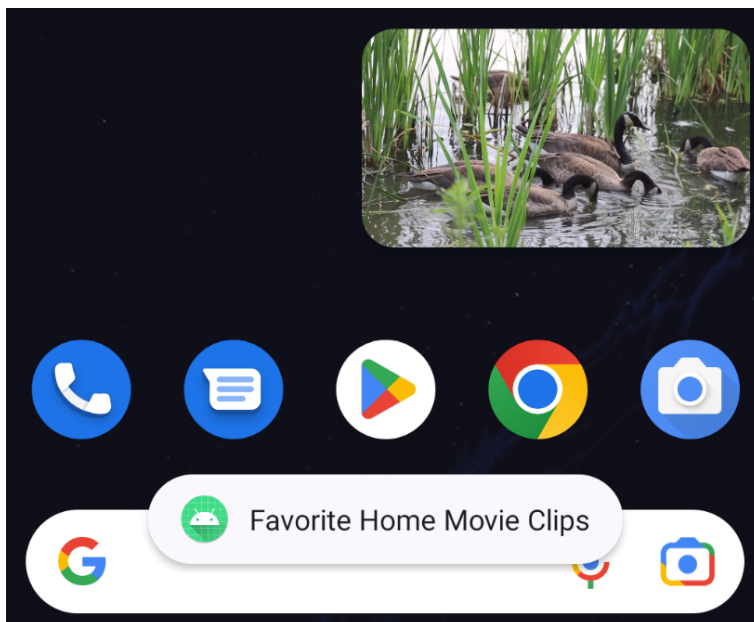


Figure 74-5

## 74.8 Summary

This chapter has demonstrated the addition of Picture-in-Picture support to an Android Studio app project, including enabling and entering PiP mode and implementing a PiP action. This included using a broadcast receiver and pending intents to implement communication between the PiP window and the activity.



# 83. An Introduction to Android App Links

As technology evolves, the traditional distinction between web and mobile content is beginning to blur. One area where this is particularly true is the growing popularity of progressive web apps, where web apps look and behave much like traditional mobile apps.

Another trend involves making the content within mobile apps discoverable through web searches and via URL links. In the context of Android app development, the App Links feature is designed to make it easier for users to discover and access content stored within an Android app, even if the user does not have the app installed.

## 83.1 An Overview of Android App Links

An app link is a standard HTTP URL that is an easy way to link directly to a particular place in your app from an external source such as a website or app. App links (also called *deep links*) are used primarily to encourage users to engage with an app and to allow users to share app content.

App link implementation is a multi-step process that involves the addition of intent filters to the project manifest, implementing link handling code within the associated app activities, and the use of digital asset links files to associate app and web-based content.

These steps can be performed manually by making changes within the project or automatically using the Android Studio App Links Assistant.

These steps can be performed manually by making project changes or automatically using the Android Studio App Links Assistant.

The remainder of this chapter will outline app links implementation in terms of the changes that must be made to a project. The next chapter (*"An Android Studio App Links Tutorial"*) will demonstrate the use of the App Links Assistant to achieve the same results.

## 83.2 App Link Intent Filters

An app link URL needs to be mapped to a specific activity within an app project. This is achieved by adding intent filters to the project's *AndroidManifest.xml* file designed to launch an activity in response to an *android.intent.action.VIEW* action. The intent filters are declared within the element for the activity to be launched and must contain the data outlining the scheme, host, and path of the app link URL. The following manifest fragment, for example, declares an intent filter to launch an activity named `MyActivity` when an app link matching *http://www.example.com/welcome* is detected:

```
<activity android:name="com.ebookfrenzy.myapplication.MyActivity">

    <intent-filter android:autoVerify="true">
        <action android:name="android.intent.action.VIEW" />
        <category android:name="android.intent.category.DEFAULT" />
        <category android:name="android.intent.category.BROWSABLE" />
    </intent-filter>
</activity>
```

```
<data
    android:scheme="http"
    android:host="www.example.com"
    android:pathPrefix="/welcome" />
</intent-filter>
</activity>
```

The order in which ambiguous intent filters are handled can be specified using the *order* property of the intent filter tag as follows:

```
<application>
    <activity android:name=" com.ebookfrenzy.myapplication.MyActivity">
        <intent-filter android:autoVerify="true" android:order="1">
            .
            .
        </intent-filter>
    </activity>
</application>
```

The intent filter will cause the app link to launch the correct activity, but code must still be added to the target activity to handle the intent appropriately.

### 83.3 Handling App Link Intents

In most cases, the launched activity will need to gain access to the app link URL and take specific action based on how the URL is structured. Continuing from the above example, the activity will likely display different content when launched via a URL containing a path of */welcome/newuser* than one with the path set to */welcome/existinguser*.

When the link launches the activity, it is passed an intent object containing data about the action which launched the activity, including a Uri object containing the app link URL. Within the initialization stages of the activity, code can be added to extract this data as follows:

```
val appLinkIntent = intent
val appLinkAction = appLinkIntent.action
val appLinkData = appLinkIntent.data
```

Having obtained the Uri for the app link, the various components that make up the URL path can be used to decide the actions to perform within the activity. In the following code example, the last component of the URL is used to identify whether content should be displayed for a new or existing user:

```
val userType = appLinkData.lastPathSegment

if (userType == "newuser") {
    // display new user content
} else {
    // display existing user content
}
```

### 83.4 Associating the App with a Website

Before an app link will work, an app link URL must be associated with the website on which the app link is based. This is achieved by creating a Digital Asset Links file named *assetlinks.json* and installing it within the website's *.well-known* folder. Note that digital asset linking is only possible for websites that are HTTPS based.

A digital asset links file comprises a *relation* statement granting permission for a target app to be launched using the website's link URLs and a target statement declaring the companion app package name and SHA-256 certificate fingerprint for that project. A typical asset link file might, for example, read as follows:

```
[{  
  "relation": ["delegate_permission/common.handle_all_urls"],  
  "target" : { "namespace": "android_app",  
    "package_name": "<app package name here>",  
    "sha256_cert_fingerprints": ["<app certificate here>"] }  
}]
```

The *assetlinks.json* file can contain multiple digital asset links, allowing a single website to be associated with more than one companion app.

## 83.5 Summary

Android App Links allow app activities to be launched via URL links from external websites and other apps. App links are implemented using intent filters within the project manifest file and intent handling code within the launched activity. Using a Digital Asset Links file, it is also possible to associate the domain name used in an app link with the corresponding website. Once the association has been established, Android no longer needs to ask the user to select the target app when an app link is used.





# 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 by embedding advertising within applications. The most common and lucrative option is to charge the user for purchasing items from within the application after installing it. 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 supports integrating 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 must also register a Google merchant account. 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 must then be uploaded to the console and enabled for in-app purchasing. However, the console will not activate in-app purchasing support for an app unless the Google Play Billing Library has been added to the module-level *build.gradle.kts* 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:

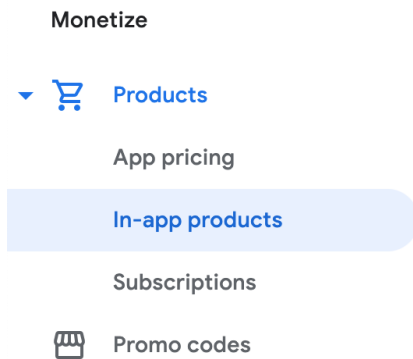


Figure 87-1

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 regularly, such as access to news content or the premium features of an app. When creating a subscription, a *base plan* specifies the price, renewal period (monthly, annually, etc.), and whether the subscription auto-renews. Users can also be given discount offers and 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 canceled by the user
        } else {
```

```

        // Handle errors here
    }
}

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

```

## 87.4 Connecting to the Google Play Billing Library

After successfully creating the Billing Client, the next step is initializing a connection to the Google Play Billing Library. A call must be made to the `startConnection()` method of the billing client instance to establish this connection. Since the connection is performed asynchronously, a `BillingClientStateListener` must 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 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 purchased item.

```
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 must 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 and 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, checking whether a user has already purchased a product or subscription is a common requirement. A list of all the user's previous purchases of a specific type can be generated by calling the

## An Overview of Android In-App Billing

`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 =
    PurchaseResponseListener { 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. This chapter 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. It 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.

## 91. An Overview of Gradle in Android Studio

Up until this point, it has been taken for granted that Android Studio will take the necessary steps to compile and run the application projects that have been created. Android Studio has been achieving this in the background using a system known as *Gradle*.

It is time to look at how Gradle is used to compile and package an application project's various elements and begin exploring how to configure this system when more advanced requirements are needed for building projects in Android Studio.

### 91.1 An Overview of Gradle

Gradle is an automated build toolkit that allows how projects are built to be configured and managed through a set of build configuration files. This includes defining how a project will be built, what dependencies need to be fulfilled to build successfully, and what the build process's end result (or results) should be.

The strength of Gradle lies in the flexibility that it provides to the developer. The Gradle system is a self-contained, command-line-based environment that can be integrated into other environments using plugins. In the case of Android Studio, Gradle integration is provided through the appropriately named Android Studio Plugin.

Although the Android Studio Plug-in allows Gradle tasks to be initiated and managed from within Android Studio, the Gradle command-line wrapper can still be used to build Android Studio-based projects, including on systems on which Android Studio is not installed.

The configuration rules to build a project are declared in Gradle build files and scripts based on the Groovy programming language.

### 91.2 Gradle and Android Studio

Gradle brings many powerful features to building Android application projects. Some of the key features are as follows:

#### 91.2.1 Sensible Defaults

Gradle implements a concept referred to as *convention over configuration*. This means that Gradle has a predefined set of sensible default configuration settings that will be used unless settings in the build files override them. This means that builds can be performed with the minimum configuration required by the developer. Changes to the build files are only needed when the default configuration does not meet your build needs.

#### 91.2.2 Dependencies

Another key area of Gradle functionality is that of dependencies. Consider, for example, a module within an Android Studio project which triggers an intent to load another module in the project. The first module has, in effect, a dependency on the second module since the application will fail to build if the second module cannot be located and launched at runtime. This dependency can be declared in the Gradle build file for the first module so that the second module is included in the application build, or an error flagged if the second module cannot be found or built. Other examples of dependencies are libraries and JAR files on which the project depends to compile and run.

Gradle dependencies can be categorized as *local* or *remote*. A local dependency references an item that is present on the local file system of the computer system on which the build is being performed. A remote dependency refers to an item that is present on a remote server (typically referred to as a *repository*).

Remote dependencies are handled for Android Studio projects using another project management tool named *Maven*. If a remote dependency is declared in a Gradle build file using Maven syntax, then the dependency will be downloaded automatically from the designated repository and included in the build process. The following dependency declaration, for example, causes the AppCompat library to be added to the project from the Google repository:

```
implementation("androidx.appcompat:appcompat:1.6.1")
```

### 91.2.3 Build Variants

In addition to dependencies, Gradle also provides *build variant* support for Android Studio projects. This allows multiple variations of an application to be built from a single project. Android runs on many different devices encompassing a range of processor types and screen sizes. To target as wide a range of device types and sizes as possible, it will often be necessary to build several variants of an application (for example, one with a user interface for phones and another for tablet-sized screens). Through the use of Gradle, this is now possible in Android Studio.

### 91.2.4 Manifest Entries

Each Android Studio project has associated with it an *AndroidManifest.xml* file containing configuration details about the application. Several manifest entries can be specified in Gradle build files which are then auto-generated into the manifest file when the project is built. This capability complements the build variants feature, allowing elements such as the application version number, application ID, and SDK version information to be configured differently for each build variant.

### 91.2.5 APK Signing

The chapter “*Creating, Testing, and Uploading an Android App Bundle*” covered creating a signed release APK file using the Android Studio environment. It is also possible to include the signing information entered through the Android Studio user interface within a Gradle build file to generate signed APK files from the command line.

### 91.2.6 ProGuard Support

ProGuard is a tool included with Android Studio that optimizes, shrinks, and obfuscates Java byte code to make it more efficient and harder to reverse engineer (the method by which others can identify the logic of an application through analysis of the compiled Java byte code). The Gradle build files allow you to control whether or not ProGuard is run on your application when it is built.

## 91.3 The Property and Settings Gradle Build File

The gradle build configuration consists of configuration, property, and settings files. The *gradle.properties* file, for example, contains mostly esoteric settings relating to the command-line flags used by the Java Virtual Machine (JVM), whether or not the project uses the AndroidX libraries and Kotlin coding style support. As a typical user, it is unlikely that you will need to change any of these settings in this file.

The *settings.gradle.kts* file, on the other hand, defines which online repositories are to be searched when the build system needs to download and install any additional libraries and plugins required to build the project and the project name. A typical *settings.gradle.kts* file will read as follows:

```
pluginManagement {
    repositories {
        google()
        mavenCentral()
    }
}
```



```

        gradlePluginPortal()
    }
}
dependencyResolutionManagement {
    repositoriesMode.set(RepositoriesMode.FAIL_ON_PROJECT_REPOS)
    repositories {
        google()
        mavenCentral()
    }
}

rootProject.name = "ThemeDemo"
include(":app")

```

As with the *gradle.properties* file, it is unlikely that changes will need to be made to this file.

## 91.4 The Top-level Gradle Build File

A completed Android Studio project contains everything needed to build an Android application and consists of modules, libraries, manifest files, and Gradle build files.

Each project contains one top-level Gradle build file. This file is listed as *build.gradle.kts* (*Project: <project name>*) and can be found in the project tool window as highlighted in Figure 91-1:

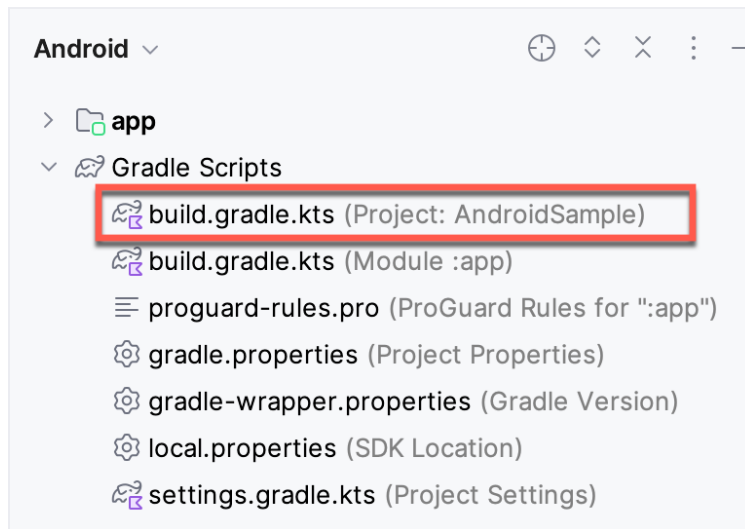


Figure 91-1

By default, the contents of the top-level Gradle build file reads as follows:

```

// Top-level build file where you can add configuration options common to all sub-
// projects/modules.
plugins {
    id("com.android.application") version "8.1.0" apply false
    id("org.jetbrains.kotlin.android") version "1.8.0" apply false
}

```

As it stands, all the file does is declare that remote libraries are to be obtained using the jcenter repository, and that builds depend on the Android plugin for Gradle. In most situations, making any changes to this build file is unnecessary.

### 91.5 Module Level Gradle Build Files

An Android Studio application project is made up of one or more modules. Take, for example, a hypothetical application project named `GradleDemo` which contains modules named `Module1` and `Module2`, respectively. In this scenario, each module will require its own Gradle build file. In terms of the project structure, these would be located as follows:

- `Module1/build.gradle.kts`
- `Module2/build.gradle.kts`

By default, the `Module1` *build.gradle.kts* file would resemble that of the following listing:

```
plugins {
    id("com.android.application")
    id("org.jetbrains.kotlin.android")
}

android {
    namespace = "com.example.gradlesample"
    compileSdk = 33

    defaultConfig {
        applicationId = "com.example.gradlesample"
        minSdk = 26
        targetSdk = 33
        versionCode = 1
        versionName = "1.0"

        testInstrumentationRunner = "androidx.test.runner.AndroidJUnitRunner"
    }

    buildTypes {
        release {
            isMinifyEnabled = false
            proguardFiles(
                getDefaultProguardFile("proguard-android-optimize.txt"),
                "proguard-rules.pro"
            )
        }
    }

    compileOptions {
        sourceCompatibility = JavaVersion.VERSION_1_8
        targetCompatibility = JavaVersion.VERSION_1_8
    }
}
```

```

    kotlinOptions {
        jvmTarget = "1.8"
    }
}

dependencies {

    implementation("androidx.core:core-ktx:1.9.0")
    implementation("androidx.appcompat:appcompat:1.6.1")
    implementation("com.google.android.material:material:1.9.0")
    implementation("androidx.constraintlayout:constraintlayout:2.1.4")
    testImplementation("junit:junit:4.13.2")
    androidTestImplementation("androidx.test.ext:junit:1.1.5")
    androidTestImplementation("androidx.test.espresso:espresso-core:3.5.1")
}

```

As is evident from the file content, the build file begins by declaring the use of the Gradle Android application and Kotlin plug-ins:

```

plugins {
    id("com.android.application")
    id("org.jetbrains.kotlin.android")
}

```

The *android* section of the file declares the project namespace and then states the version of the SDK to be used when building Module1.

```

android {
    namespace = "com.example.gradlesample"
    compileSdk = 33
}

```

The items declared in the defaultConfig section define elements to be generated into the module's *AndroidManifest.xml* file during the build. These settings, which may be modified in the build file, are taken from the settings entered within Android Studio when the module was first created:

```

defaultConfig {
    applicationId = "com.example.gradlesample"
    minSdk = 26
    targetSdk = 33
    versionCode = 1
    versionName = "1.0"

    testInstrumentationRunner = "androidx.test.runner.AndroidJUnitRunner"
}

```

The buildTypes section contains instructions on whether and how to run ProGuard on the APK file when a release version of the application is built:

```

buildTypes {
    release {

```

## An Overview of Gradle in Android Studio

```
        isMinifyEnabled = false
        proguardFiles(
            getDefaultProguardFile("proguard-android-optimize.txt"),
            "proguard-rules.pro"
        )
    }
}
```

As currently configured, ProGuard will not be run when Module1 is built. To enable ProGuard, the *minifyEnabled* entry must be changed from *false* to *true*. The *proguard-rules.pro* file can be found in the module directory of the project. Changes made to this file override the default settings in the *proguard-android.txt* file, which is located in the Android SDK installation directory under *sdk/tools/proguard*.

Since no debug buildType is declared in this file, the defaults will be used (built without ProGuard, signed with a debug key, and debug symbols enabled).

An additional section, entitled *productFlavors*, may also be included in the module build file to enable multiple build variants to be created.

Next, directives are included to specify the version of the Java compiler to be used when building the project:

```
compileOptions {
    sourceCompatibility JavaVersion.VERSION_1_8
    targetCompatibility JavaVersion.VERSION_1_8
}
kotlinOptions {
    jvmTarget = "1.8"
}
```

Finally, the dependencies section lists any local and remote dependencies on which the module depends. The dependency lines in the above example file designate the Android libraries that need to be included from the Android Repository:

```
dependencies {

    implementation("androidx.core:core-ktx:1.9.0")
    implementation("androidx.appcompat:appcompat:1.6.1")
    implementation("com.google.android.material:material:1.9.0")
    .
    .
}
```

Note that the dependency declarations include version numbers to indicate which library version should be included.

## 91.6 Configuring Signing Settings in the Build File

The “*Creating, Testing, and Uploading an Android App Bundle*” chapter of this book covered the steps involved in setting up keys and generating a signed release APK file using the Android Studio user interface. These settings may also be declared within a *signingConfigs* section of the *build.gradle.kts* file. For example:

```
.
.
```

```

defaultConfig {
    .
    .
}
signingConfigs {
    release {
        storeFile file("keystore.release")
        storePassword "your keystore password here"
        keyAlias "your key alias here"
        keyPassword "your key password here"
    }
}
buildTypes {
    .
    .
    .
}

```

The above example embeds the key password information directly into the build file. An alternative to this approach is to extract these values from system environment variables:

```

signingConfigs {
    release {
        storeFile file("keystore.release")
        storePassword System.getenv("KEYSTOREPASSWD")
        keyAlias "your key alias here"
        keyPassword System.getenv("KEYPASSWD")
    }
}

```

Yet another approach is to configure the build file so that Gradle prompts for the passwords to be entered during the build process:

```

signingConfigs {
    release {
        storeFile file("keystore.release")
        storePassword System.console().readLine
            ("\nEnter Keystore password: ")
        keyAlias "your key alias here"
        keyPassword System.console().readLine("\nEnter Key password: ")
    }
}

```

## 91.7 Running Gradle Tasks from the Command Line

Each Android Studio project contains a Gradle wrapper tool to invoke Gradle tasks from the command line. This tool is located in the root directory of each project folder. While this wrapper is executable on Windows systems, it may need to have execute permission enabled on Linux and macOS before it can be used. To enable execute permission, open a terminal window, change directory to the project folder for which the wrapper is needed, and execute the following command:

## An Overview of Gradle in Android Studio

```
chmod +x gradlew
```

Once the file has execute permissions, the location of the file will either need to be added to your \$PATH environment variable or the name prefixed by ./ to run. For example:

```
./gradlew tasks
```

Gradle views project building in terms of several different tasks. A full listing of tasks that are available for the current project can be obtained by running the following command from within the project directory (remembering to prefix the command with a ./ if running on macOS or Linux):

```
gradlew tasks
```

To build a debug release of the project suitable for device or emulator testing, use the assembleDebug option:

```
gradlew assembleDebug
```

Alternatively, to build a release version of the application:

```
gradlew assembleRelease
```

## 91.8 Summary

For the most part, Android Studio performs application builds in the background without any intervention from the developer. This build process is handled using the Gradle system, an automated build toolkit designed to allow how projects are built to be configured and managed through a set of build configuration files. While the default behavior of Gradle is adequate for many basic project build requirements, the need to configure the build process is inevitable with more complex projects. This chapter has provided an overview of the Gradle build system and configuration files within the context of an Android Studio project.

# Index

## Symbols

?. 101  
 <application> 506  
 <fragment> 293  
 <fragment> element 293  
 <receiver> 484  
 <service> 506, 512, 519  
 :: operator 103  
 .well-known folder 457, 480, 696

## A

AbsoluteLayout 174  
 ACCESS\_COARSE\_LOCATION permission 606  
 ACCESS\_FINE\_LOCATION permission 606  
 acknowledgePurchase() method 735  
 ACTION\_DOWN 270  
 ACTION\_MOVE 270  
 ACTION\_POINTER\_DOWN 270  
 ACTION\_POINTER\_UP 270  
 ACTION\_UP 270  
 ACTION\_VIEW 475  
 Active / Running state 150  
 Activity 87, 153  
   adding views in Java code 251  
   class 153  
   creation 16  
   Entire Lifetime 157  
   Foreground Lifetime 157  
   lifecycle methods 155  
   lifecycles 147  
   returning data from 454  
   state change example 161  
   state changes 153  
   states 150  
   Visible Lifetime 157

Activity Lifecycle 149  
 Activity Manager 86  
 ActivityResultLauncher 455  
 Activity Stack 149  
 Actual screen pixels 242  
 adb  
   command-line tool 63  
   connection testing 69  
   device pairing 67  
   enabling on Android devices 63  
   Linux configuration 66  
   list devices 63  
   macOS configuration 64  
   overview 63  
   restart server 64  
   testing connection 69  
   WiFi debugging 67  
   Windows configuration 65  
   Wireless debugging 67  
   Wireless pairing 67  
 addCategory() method 483  
 addMarker() method 659  
 addView() method 245  
 ADD\_VOICEMAIL permission 606  
 android  
   exported 507  
   gestureColor 286  
   layout\_behavior property 447  
   onClick 295  
   process 507, 519  
   uncertainGestureColor 286  
 Android  
   Activity 87  
   architecture 83  
   events 263  
   intents 88  
   onClick Resource 263  
   runtime 84  
   SDK Packages 6

## Index

- android.app 84
- Android Architecture Components 309
- android.content 84
- android.content.Intent 453
- android.database 84
- Android Debug Bridge. *See* ADB
- Android Development
  - System Requirements 3
- Android Devices
  - designing for different 173
- android.graphics 84
- android.hardware 84
- android.intent.action 489
- android.intent.action.BOOT\_COMPLETED 507
- android.intent.action.MAIN 475
- android.intent.category.LAUNCHER 475
- Android Libraries 84
- android.media 85
- Android Monitor tool window 36
- Android Native Development Kit 85
- android.net 85
- android.opengl 84
- android.os 85
- android.permission.RECORD\_AUDIO 615
- android.print 85
- Android Project
  - create new 15
- android.provider 85
- Android SDK Location
  - identifying 10
- 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 83
- Android Studio
  - changing theme 61
  - downloading 3
  - Editor Window 56
  - installation 4
  - Linux installation 5
  - macOS installation 4
  - Navigation Bar 55
  - Project tool window 56
  - setup wizard 5
  - Status Bar 56
  - Toolbar 55
  - Tool window bars 56
  - tool windows 56
  - updating 12
  - Welcome Screen 53
  - Windows installation 4
- android.text 85
- android.util 85
- android.view 85
- android.view.View 176
- android.view.ViewGroup 173, 176
- Android Virtual Device. *See* AVD
  - overview 31
- Android Virtual Device Manager 31
- android.webkit 85
- android.widget 85
- AndroidX libraries 762
- API Key 651
- APK analyzer 728
- APK file 721
- APK File
  - analyzing 728
- APK Signing 762
- APK Wizard dialog 720
- App Architecture
  - modern 309
- AppBar
  - anatomy of 445
- appbar\_scrolling\_view\_behavior 447
- App Bundles 717
  - creating 721
  - overview 717
  - revisions 727



- uploading 724
- AppCompatActivity class 154
- App Inspector 57
- Application
  - stopping 36
- Application Context 89
- Application Framework 85
- Application Manifest 89
- Application Resources 89
- App Link
  - Adding Intent Filter 704
  - Digital Asset Links file 696, 457
  - Intent Filter Handling 704
  - Intent Filters 695
  - Intent Handling 696
  - Testing 708
  - URL Mapping 701
- App Links 695
  - auto verification 456
  - autoVerify 457
  - overview 695
- Apply Changes 259
  - Apply Changes and Restart Activity 259
  - Apply Code Changes 259
  - fallback settings 261
  - options 259
  - Run App 259
  - tutorial 261
- applyToActivitiesIfAvailable() method 757
- Architecture Components 309
- ART 84
- as 103
- as? 103
- asFlow() builder 524
- assetlinks.json , 696, 457
- asSharedFlow() 534
- asStateFlow() 533
- async 493
- Attribute Keyframes 380
- Audio
  - supported formats 613
- Audio Playback 613

- Audio Recording 613
- Autoconnect Mode 207
- Automatic Link Verification 456, 479
- autoVerify 457, 704
- AVD
  - cold boot 48
  - command-line creation 31
  - creation 31
  - device frame 40
  - Display mode 51
  - launch in tool window 40
  - overview 31
  - quickboot 48
  - Resizable 50
  - running an application 34
  - Snapshots 47
  - standalone 37
  - starting 33
  - Startup size and orientation 34

## B

- Background Process 148
- Barriers 200
  - adding 219
  - constrained views 200
- Baseline Alignment 199
- beginTransaction() method 294
- BillingClient 736
  - acknowledgePurchase() method 735
  - consumeAsync() method 735
  - getPurchaseState() method 734
  - initialization 732, 740
  - launchBillingFlow() method 734
  - queryProductDetailsAsync() method 733
  - queryPurchasesAsync() method 736
- BillingResult 747
  - getDebugMessage() 747
- Binding Expressions 329
  - one-way 329
  - two-way 330
- BIND\_JOB\_SERVICE permission 507
- bindService() method 505, 509, 513

## Index

Biometric Authentication 709

- callbacks 713
- overview 709
- tutorial 709

Biometric Prompt 714Bitwise AND 109Bitwise Inversion 108Bitwise Left Shift 110Bitwise OR 109Bitwise Right Shift 110Bitwise XOR 109black activity 16Blank template 177Blueprint view 205BODY\_SENSORS permission 606Boolean 96Bound Service 505, 509

- adding to a project 510
- Implementing the Binder 510
- Interaction options 509

BoundService class 511Broadcast Intent 483

- example 485
- overview 88, 483
- sending 486
- Sticky 485

Broadcast Receiver 483

- adding to manifest file 488
- creation 487
- overview 88, 484

BroadcastReceiver class 484BroadcastReceiver superclass 487buffer() operator 527Build tool window 58Build Variants , 58

- tool window 58

Bundle class 170Bundled Notifications 634

## C

Calendar permissions 606CALL\_PHONE permission 606

CAMERA permission 606Camera permissions 606CameraUpdateFactory class

- methods 660

cancelAndJoin() 493cancelChildren() 493CancellationSignal 714Canvas class 690CardView

- layout file 433
- responding to selection of 441

CardView class 433C/C++ Libraries 85Chain bias 228chain head 198chains 198Chains

- creation of 225

Chain style

- changing 227

chain styles 198Char 96CheckBox 173checkSelfPermission() method 610Circle class 647Code completion 74Code Editor

- basics 71
- Code completion 74
- Code Generation 76
- Code Reformatting 79
- Document Tabs 72
- Editing area 72
- Gutter Area 72
- Live Templates 80
- Splitting 74
- Statement Completion 76
- Status Bar 73

Code Generation 76Code Reformatting 79code samples

- download 1

- cold boot 48
- Cold flows 532
- CollapsingToolBarLayout
  - example 448
  - introduction 448
  - parallax mode 448
  - pin mode 448
  - setting scrim color 451
  - setting title 451
  - with image 448
- collectLatest() operator 526
- Color class 691
- COLOR\_MODE\_COLOR 666, 686
- COLOR\_MODE\_MONOCHROME 666, 686
- combine() operator 531
- Common Gestures 275
  - detection 275
- Communicating Sequential Processes 491
- Companion Objects 133
- Component tree 20
- conflate() operator 526
- Constraint Bias 197
  - adjusting 211
- ConstraintLayout
  - advantages of 203
  - Availability 204
  - Barriers 200
  - Baseline Alignment 199
  - chain bias 228
  - chain head 198
  - chains 198
  - chain styles 198
  - Constraint Bias 197
  - Constraints 195
  - conversion to 223
  - convert to MotionLayout 387
  - deleting constraints 210
  - guidelines 217
  - Guidelines 200
  - manual constraint manipulation 207
  - Margins 196, 211
  - Opposing Constraints 196, 213
  - overview of 195
  - Packed chain 199, 228
  - ratios 203, 229
  - Spread chain 198
  - Spread inside 228
  - Spread inside chain 198
  - tutorial 233
  - using in Android Studio 205
  - Weighted chain 198, 228
  - Widget Dimensions 199, 215
  - Widget Group Alignment 221
- ConstraintLayout chains
  - creation of 225
  - in layout editor 225
- ConstraintLayout Chain style
  - changing 227
- Constraints
  - deleting 210
- ConstraintSet
  - addToHorizontalChain() method 248
  - addToVerticalChain() method 248
  - alignment constraints 247
  - apply to layout 246
  - applyTo() method 246
  - centerHorizontally() method 247
  - centerVertically() method 247
  - chains 247
  - clear() method 248
  - clone() method 247
  - connect() method 246
  - connect to parent 246
  - constraint bias 247
  - copying constraints 247
  - create 246
  - create connection 246
  - createHorizontalChain() method 247
  - createVerticalChain() method 247
  - guidelines 248
  - removeFromHorizontalChain() method 248
  - removeFromVerticalChain() method 248
  - removing constraints 248
  - rotation 249

## Index

- scaling 248
  - setGuidelineBegin() method 248
  - setGuidelineEnd() method 248
  - setGuidelinePercent() method 248
  - setHorizontalBias() method 247
  - setRotationX() method 249
  - setRotationY() method 249
  - setScaleX() method 248
  - setScaleY() method 248
  - setTransformPivot() method 249
  - setTransformPivotX() method 249
  - setTransformPivotY() method 249
  - setVerticalBias() method 247
  - sizing constraints 247
  - tutorial 251
  - view IDs 253
  - ConstraintSet class 245, 246
  - Constraint Sets 246
  - ConstraintSets
    - configuring 376
  - consumeAsync() method 735
  - ConsumeParams 745
  - Contacts permissions 606
  - container view 173
  - Content Provider 86
    - overview 89
  - Context class 89
  - CoordinatorLayout 174, 447
  - Coroutine Builders 493
    - async 493
    - coroutineScope 493
    - launch 493
    - runBlocking 493
    - supervisorScope 493
    - withContext 493
  - Coroutine Dispatchers 492
  - Coroutines 491, 523
    - adding libraries 499
    - channel communication 497
    - GlobalScope 492
    - returning results 495
    - Suspend Functions 492
      - suspending 494
      - tutorial 499
      - ViewModelScope 492
      - vs. Threads 491
  - coroutineScope 493
  - Coroutine Scope 492
  - createPrintDocumentAdapter() method 681
  - Custom Accessors 131
  - Custom Attribute 377
  - Custom Document Printing 669, 681
  - Custom Gesture
    - recognition 281
  - Custom Print Adapter
    - implementation 683
  - Custom Print Adapters 681
  - Custom Theme
    - building 751
  - Cycle Editor 405
  - Cycle Keyframe 385
  - Cycle Keyframes
    - overview 401
- ## D
- dangerous permissions
    - list of 606
  - Dark Theme 36
    - enable on device 36
  - Data Access Object (DAO) 554
  - Database Inspector 560, 584
    - live updates 584
    - SQL query 584
  - Database Rows 548
  - Database Schema 547
  - Database Tables 547
  - Data binding
    - binding expressions 329
  - Data Binding 311
    - binding classes 328
    - enabling 334
    - event and listener binding 330
    - key components 325
    - overview 325

- tutorial 333
- variables 328
- with LiveData 311
- DDMS 36
- Debugging
  - enabling on device 63
- debug.keystore file 457, 479
- Default Function Parameters 123
- DefaultLifecycleObserver 346, 349
- deltaRelative 381
- Density-independent pixels 241
- Density Independent Pixels
  - converting to pixels 256
- Device Definition
  - custom 191
- Device File Explorer 58
- device frame 40
- Device Mirroring 69
  - enabling 69
- device pairing 67
- Digital Asset Links file 696, 457, 457
- Direct Reply Input 643
- Dispatchers.Default 493
- Dispatchers.IO 492
- Dispatchers.Main 492
- dp 241
- DROP\_LATEST 534
- DROP\_OLDEST 534
- Dynamic Colors
  - applyToActivitiesIfAvailable() method 757
  - enabling in Android 757
- Dynamic State 155
  - saving 169

## E

- Elvis Operator 103
- Empty Process 149
- Empty template 177
- Emulator
  - battery 46
  - cellular configuration 46
  - configuring fingerprints 48

- directional pad 46
- extended control options 45
- Extended controls 45
- fingerprint 46
- location configuration 46
- phone settings 46
- Resizable 50
- resize 45
- rotate 44
- Screen Record 47
- Snapshots 47
- starting 33
- take screenshot 44
- toolbar 43
- toolbar options 43
- tool window mode 50
- Virtual Sensors 47
- zoom 44
- enablePendingPurchases() method 735
- enabling ADB support 63
- Escape Sequences 97
- ettings.gradle file 762
- Event Handling 263
  - example 264
- Event Listener 265
- Event Listeners 264
- Events
  - consuming 267
- explicit
  - intent 88
- explicit intent 453
- Explicit Intent 453
- Extended Control
  - options 45

## F

- Files
  - switching between 72
- filter() operator 528
- findPointerIndex() method 270
- findViewById() 143
- Fingerprint

## Index

- emulation 48
- Fingerprint authentication
  - device configuration 710
  - permission 710
  - steps to implement 709
- Fingerprint Authentication
  - overview 709
  - tutorial 709
- FLAG\_INCLUDE\_STOPPED\_PACKAGES 483
- flatMapConcat() operator 531
- flatMapMerge() operator 531
- flexible space area 445
- Float 96
- floating action button 16, 178
  - changing appearance of 416
  - margins 414
  - removing 179
  - sizes 414
- Flow 523
  - asFlow() builder 524
  - asSharedFlow() 534
  - asStateFlow() 533
  - backgroundn handling 542
  - buffering 526
  - buffer() operator 527
  - cold 532
  - collect() 525
  - collecting data 525
  - collectLatest() operator 526
  - combine() operator 531
  - conflate() operator 526
  - declaring 524
  - emit() 525
  - emitting data 525
  - filter() operator 528
  - flatMapConcat() operator 531
  - flatMapMerge() operator 531
  - flattening 530
  - flowOf() builder 524
  - flow of flows 530
  - fold() operator 530
  - hot 532
  - intermediate operators 528
  - library requirements 524
  - map() operator 528
  - MutableSharedFlow 534
  - MutableStateFlow 533
  - onEach() operator 532
  - reduce() operator 529, 530
  - repeatOnLifecycle 544
  - SharedFlow 534
  - single() operator 526
  - StateFlow 533
  - terminal flow operators 529
  - transform() operator 529
  - try/finally 526
  - zip() operator 531
- flowOf() builder 524
- flow of flows 530
- Flow operators 528
- Flows
  - combining 531
  - Introduction to 523
- Foldable Devices 158
  - multi-resume 158
- Foreground Process 148
- Forward-geocoding 653
- Fragment
  - creation 291
  - event handling 295
  - XML file 292
- FragmentActivity class 154
- Fragment Communication 295
- Fragments 291
  - adding in code 294
  - duplicating 422
  - example 299
  - overview 291
- FragmentManager class 425
- FrameLayout 174
- Function Parameters
  - variable number of 123
- Functions 121

**G**

- Geocoder object 654
- Geocoding 652
- Gesture Builder Application 281
  - building and running 281
- Gesture Detector class 275
- GestureDetectorCompat 278
  - instance creation 278
- GestureDetectorCompat class 275
- GestureDetector.OnDoubleTapListener 275, 276
- GestureDetector.OnGestureListener 276
- GestureLibrary 281
- GestureOverlayView 281
  - configuring color 286
  - configuring multiple strokes 286
- GestureOverlayView class 281
- GesturePerformedListener 281
- Gestures
  - interception of 287
- Gestures File
  - creation 282
  - extract from SD card 282
  - loading into application 284
- GET\_ACCOUNTS permission 606
- getAction() method 489
- getDebugMessage() 747
- getFromLocation() method 654
- getId() method 246
- getIntent() method 454
- getPointerCount() method 270
- getPointerId() method 270
- getPurchaseState() method 734
- getService() method 513
- GlobalScope 492
- GNU/Linux 84
- Google Cloud
  - billing account 648
  - new project 649
- Google Cloud Print 664
- Google Drive
  - printing to 664
- GoogleMap 647

- map types 657
- GoogleMap.MAP\_TYPE\_HYBRID 657
- GoogleMap.MAP\_TYPE\_NONE 657
- GoogleMap.MAP\_TYPE\_NORMAL 657
- GoogleMap.MAP\_TYPE\_SATELLITE 657
- GoogleMap.MAP\_TYPE\_TERRAIN 657
- Google Maps Android API 647
  - Controlling the Map Camera 660
  - displaying controls 658
  - Map Markers 659
  - overview 647
- Google Maps SDK 647
  - API Key 651
  - Credentials 651
  - enabling 650
  - Maps SDK for Android 651
- Google Play App Signing 720
- Google Play Console 738
  - Creating an in-app product 738
  - License Testers 739
- Google Play Developer Console 718
- Gradle
  - APK signing settings 766
  - Build Variants 762
  - command line tasks 767
  - dependencies 761
  - Manifest Entries 762
  - overview 761
  - sensible defaults 761
- Gradle Build File
  - top level 763
- Gradle Build Files
  - module level 764
- gradle.properties file 762
- GridLayout 174
- GridLayoutManager 431

**H**

- Handler class 518
- Higher-order Functions 125
- Hot flows 532
- HP Print Services Plugin 663

## Index

HTML printing 667

HTML Printing

example 671

## I

IBinder 505, 511

IBinder object 509, 518

Image Printing 666

Immutable Variables 98

implicit

intent 88

implicit intent 453

Implicit Intent 455

Implicit Intents

example 471

importance hierarchy 147

in 241

INAPP 736

In-App Products 731

In-App Purchasing 737

acknowledgePurchase() method 735

BillingClient 732

BillingResult 747

consumeAsync() method 735

ConsumeParams 745

Consuming purchases 744

enablePendingPurchases() method 735

getPurchaseState() method 734

launchBillingFlow() method 734

Libraries 737

newBuilder() method 732

onBillingServiceDisconnected() callback 741

onBillingServiceDisconnected() method 733

onBillingSetupFinished() listener 741

onProductDetailsResponse() callback 742

Overview 731

ProductDetail 734

ProductDetails 742

products 731

ProductType 736

Purchase Flow 743

PurchaseResponseListener 736

PurchasesUpdatedListener 734

PurchaseUpdatedListener 743

purchase updates 743

queryProductDetailsAsync() 742

queryProductDetailsAsync() method 733

queryPurchasesAsync() 745

queryPurchasesAsync() method 736

runOnUiThread() 743

subscriptions 731

tutorial 737

Initializer Blocks 131

In-Memory Database 560

Inner Classes 132

IntelliJ IDEA 91

Intent 88

explicit 88

implicit 88

Intent Availability

checking for 460

Intent Filters 456

App Link 695

Intents 453

ActivityResultLauncher 455

overview 453

registerForActivityResult() 455, 468

Intent Service 505

Intent URL 474

intermediate flow operators 528

is 103

isInitialized property 103

## J

Java

convert to Kotlin 91

Java Native Interface 85

JetBrains 91

Jetpack 309

overview 309

JobIntentService 505

BIND\_JOB\_SERVICE permission 507

onHandleWork() method 505

join() 493



**K**

- KeyAttribute 380
- Keyboard Shortcuts 59
- KeyCycle 401
  - Cycle Editor 405
  - tutorial 401
- Keyframe 394
- Keyframes 380
- KeyFrameSet 410
- KeyPosition 381
  - deltaRelative 381
  - parentRelative 381
  - pathRelative 382
- Keystore File
  - creation 720
- KeyTimeCycle 401
- keytool 457
- KeyTrigger 384
- Killed state 150
- Kotlin
  - accessing class properties 131
  - and Java 91
  - arithmetic operators 105
  - assignment operator 105
  - augmented assignment operators 106
  - bitwise operators 108
  - Boolean 96
  - break 116
  - breaking from loops 115
  - calling class methods 131
  - Char 96
  - class declaration 127
  - class initialization 128
  - class properties 128
  - Companion Objects 133
  - conditional control flow 117
  - continue labels 116
  - continue statement 116
  - control flow 113
  - convert from Java 91
  - Custom Accessors 131
  - data types 95
  - decrement operator 106
  - Default Function Parameters 123
  - defining class methods 128
  - do ... while loop 115
  - Elvis Operator 103
  - equality operators 107
  - Escape Sequences 97
  - expression syntax 105
  - Float 96
  - Flow 523
  - for-in statement 113
  - function calling 122
  - Functions 121
  - Higher-order Functions 125
  - if ... else ... expressions 118
  - if expressions 117
  - Immutable Variables 98
  - increment operator 106
  - inheritance 137
  - Initializer Blocks 131
  - Inner Classes 132
  - introduction 91
  - Lambda Expressions 124
  - let Function 101
  - Local Functions 122
  - logical operators 107
  - looping 113
  - Mutable Variables 98
  - Not-Null Assertion 101
  - Nullable Type 100
  - Overriding inherited methods 140
  - playground 92
  - Primary Constructor 128
  - properties 131
  - range operator 108
  - Safe Call Operator 100
  - Secondary Constructors 128
  - Single Expression Functions 122
  - String 96
  - subclassing 137
  - Type Annotations 99
  - Type Casting 103

## Index

- Type Checking 103
  - Type Inference 99
  - variable parameters 123
  - when statement 118
  - while loop 114
- ## L
- Lambda Expressions 124
  - lateinit 102
  - Late Initialization 102
  - launch 493
  - launchBillingFlow() method 734
  - layout\_collapseMode
    - parallax 450
    - pin 450
  - layout\_constraintDimensionRatio 230
  - layout\_constraintHorizontal\_bias 228
  - layout\_constraintVertical\_bias 228
  - layout editor
    - ConstraintLayout chains 225
  - Layout Editor 19, 233
    - Autoconnect Mode 207
    - code mode 184
    - Component Tree 181
    - design mode 181
    - device screen 181
    - example project 233
    - Inference Mode 207
    - palette 181
    - properties panel 182
    - Sample Data 190
    - Setting Properties 186
    - toolbar 182
    - user interface design 233
    - view conversion 189
  - Layout Editor Tool
    - changing orientation 20
    - overview 181
  - Layout Inspector 58
  - Layout Managers 173
  - LayoutResultCallback object 687
  - Layouts 173
  - layout\_scrollFlags
    - enterAlwaysCollapsed mode 447
    - enterAlways mode 447
    - exitUntilCollapsed mode 447
    - scroll mode 447
  - Layout Validation 192
  - let Function 101
  - libc 85
  - License Testers 739
  - Lifecycle
    - awareness 345
    - components 312
    - observers 346
    - owners 345
    - states and events 346
    - tutorial 349
  - Lifecycle-Aware Components 345
  - Lifecycle library 524
  - Lifecycle Methods 155
  - Lifecycle Observer 349
    - creating a 349
  - Lifecycle Owner
    - creating a 351
  - Lifecycles
    - modern 312
  - Lifecycle.State.CREATED 544
  - Lifecycle.State.DESTROYED 544
  - Lifecycle.State.INITIALIZED 544
  - Lifecycle.State.RESUMED 544
  - Lifecycle.State.STARTED 544
  - LinearLayout 174
  - LinearLayoutManager 431
  - LinearLayoutManager layout 439
  - Linux Kernel 84
  - list devices 63
  - LiveData 310, 321
    - adding to ViewModel 321
    - observer 323
    - tutorial 321
  - Live Templates 80
  - Local Bound Service 509
    - example 509

- Local Functions 122
- Location Manager 86
- Location permission 606
- Logcat
  - tool window 57
- LogCat
  - enabling 165
- M**
- MANAGE\_EXTERNAL\_STORAGE 607
  - adb enabling 607
  - testing 607
- Manifest File
  - permissions 475
- map() operator 528
- Maps 647
- MapView 647
  - adding to a layout 654
- Marker class 647
- match\_parent properties 241
- Material design 413
- Material Design 2 749
- Material Design 2 Theming 749
- Material Design 3 749
- Material Theme Builder 751
- Material You 749
- measureTimeMillis() function 527
- MediaController
  - adding to VideoView instance 591
- MediaController class 588
  - methods 588
- MediaPlayer class 613
  - methods 613
- MediaRecorder class 613
  - methods 614
  - recording audio 614
- Memory Indicator 73
- Messenger object 518
- Microphone
  - checking for availability 616
- Microphone permissions 606
- mm 241
- MotionEvent 269, 270, 289
  - getActionMasked() 270
- MotionLayout 375
  - arc motion 380
  - Attribute Keyframes 380
  - ConstraintSets 376
  - Custom Attribute 396
  - Custom Attributes 377
  - Cycle Editor 405
  - Editor 387
  - KeyAttribute 380
  - KeyCycle 401
  - Keyframes 380
  - KeyFrameSet 410
  - KeyPosition 381
  - KeyTimeCycle 401
  - KeyTrigger 384
  - OnClick 379, 392
  - OnSwipe 379
  - overview 375
  - Position Keyframes 381
  - previewing animation 392
  - Trigger Keyframe 384
  - Tutorial 387
- MotionScene
  - ConstraintSets 376
  - Custom Attributes 377
  - file 376
  - overview 375
  - transition 376
- moveCamera() method 660
- multiple devices
  - testing app on 35
- Multiple Touches
  - handling 270
- multi-resume 158
- Multi-Touch
  - example 271
- Multi-touch Event Handling 269
- multi-window support 158
- MutableSharedFlow 534
- MutableStateFlow 533

## Index

Mutable Variables 98

My Location Layer 647

## N

Navigation 355

adding destinations 364

overview 355

pass data with safeargs 371

passing arguments 360

stack 355

tutorial 361

Navigation Action

triggering 359

Navigation Architecture Component 355

Navigation Component

tutorial 361

Navigation Controller

accessing 359

Navigation Graph 358, 362

adding actions 367

creating a 362

Navigation Host 356

declaring 363

newBuilder() method 732

normal permissions 605

Notification

adding actions 634

Direct Reply Input 643

issuing a basic 630

launch activity from a 632

PendingIntent 640

Reply Action 642

updating direct reply 644

Notifications

bundled 634

overview 623

Notifications Manager 86

Not-Null Assertion 101

Nullable Type 100

## O

Observer

implementing a LiveData 323

onAttach() method 296

onBillingServiceDisconnected() callback 741

onBillingServiceDisconnected() method 733

onBillingSetupFinished() listener 741

onBind() method 506, 509

onBindViewHolder() method 439

OnClick 379

onClickListener 264, 265, 268

onClick() method 263

onCreateContextMenuListener 264

onCreate() method 148, 155, 506

onCreateView() method 156

onDestroy() method 156, 506

onDoubleTap() method 275

onDown() method 275

onEach() operator 532

onFling() method 275

onFocusChangeListener 264

OnFragmentInteractionListener

implementation 369

onGesturePerformed() method 281

onHandleWork() method 506

onKeyListener 264

onLayoutFailed() method 687

onLayoutFinished() method 687

onLongClickListener 264

onLongPress() method 275

onMapReady() method 656

onPageFinished() callback 672

onPause() method 156

onProductDetailsResponse() callback 742

onReceive() method 148, 484, 485, 487

onRequestPermissionsResult() method 609, 620, 628, 638

onRestart() method 155

onRestoreInstanceState() method 156

onResume() method 148, 156

onSaveInstanceState() method 156

onScaleBegin() method 287

onScaleEnd() method 287

onScale() method 287

onScroll() method 275

OnSeekBarChangeListener 306  
 onServiceConnected() method 509, 512, 519  
 onServiceDisconnected() method 509, 512, 519  
 onShowPress() method 275  
 onSingleTapUp() method 275  
 onStartCommand() method 506  
 onStart() method 155  
 onStop() method 156  
 onTouchEvent() method 275, 287  
 onTouchListener 264  
 onTouch() method 270  
 onCreateView() method 156  
 onViewStatusRestored() method 156  
 OpenJDK 3

## P

Package Explorer 18  
 Package Manager 86  
 PackageManager class 616  
 PackageManager.FEATURE\_MICROPHONE 616  
 PackageManager.PERMISSION\_DENIED 607  
 PackageManager.PERMISSION\_GRANTED 607  
 Package Name 16  
 Packed chain 199, 228  
 PageRange 688, 689  
 Paint class 691  
 parentRelative 381  
 parent view 175  
 pathRelative 382  
 Paused state 150  
 PdfDocument 669  
 PdfDocument.Page 681, 688  
 PendingIntent class 640  
 Permission
 

- checking for 607

 permissions
 

- normal 605

 Persistent State 155  
 Phone permissions 606  
 Pinch Gesture
 

- detection 287
- example 287

Pinch Gesture Recognition 281  
 Position Keyframes 381  
 POST\_NOTIFICATIONS permission 606, 638  
 Primary Constructor 128  
 PrintAttributes 686  
 PrintDocumentAdapter 669, 681  
 Printing
 

- color 666
- monochrome 666

 Printing framework
 

- architecture 663

 Printing Framework 663  
 Print Job
 

- starting 692

 PrintManager service 673  
 Problems
 

- tool window 58

 process
 

- priority 147
- state 147

 PROCESS\_OUTGOING\_CALLS permission 606  
 Process States 147  
 ProductDetail 734  
 ProductDetails 742  
 ProductType 736  
 Profiler
 

- tool window 58

 ProgressBar 173  
 progaurd-rules.pro file 766  
 ProGuard Support 762  
 Project Name 16  
 Project tool window 18, 57  
 pt 241  
 PurchaseResponseListener 736  
 PurchasesUpdatedListener 734  
 PurchaseUpdatedListener 743  
 putExtra() method 453, 483  
 px 242

## Q

queryProductDetailsAsync() 742  
 queryProductDetailsAsync() method 733

## Index

queryPurchaseHistoryAsync() method 736  
queryPurchasesAsync() 745  
queryPurchasesAsync() method 736  
quickboot snapshot 48  
Quick Documentation 79

## R

RadioButton 173  
Range Operator 108  
ratios 229  
READ\_CALENDAR permission 606  
READ\_CALL\_LOG permission 606  
READ\_CONTACTS permission 606  
READ\_EXTERNAL\_STORAGE permission 607  
READ\_PHONE\_STATE permission 606  
READ\_SMS permission 606  
RECEIVE\_MMS permission 606  
RECEIVE\_SMS permission 606  
RECEIVE\_WAP\_PUSH permission 606  
Recent Files Navigation 60  
RECORD\_AUDIO permission 606  
Recording Audio  
    permission 615  
RecyclerView 431  
    adding to layout file 432  
    GridLayoutManager 431  
    initializing 439  
    LinearLayoutManager 431  
    StaggeredGridLayoutManager 431  
RecyclerView Adapter  
    creation of 437  
RecyclerView.Adapter 432, 438  
    getItemCount() method 432  
    onBindViewHolder() method 432  
    onCreateViewHolder() method 432  
RecyclerView.ViewHolder  
    getAdapterPosition() method 442  
reduce() operator 529, 530  
registerForActivityResult() 455  
registerForActivityResult() method 454, 468  
registerReceiver() method 485  
RelativeLayout 174  
Release Preparation 717  
Remote Bound Service 517  
    client communication 517  
    implementation 517  
    manifest file declaration 519  
RemoteInput.Builder() method 640  
RemoteInput Object 640  
Remote Service  
    launching and binding 519  
    sending a message 521  
repeatOnLifecycle 544  
Repository  
    tutorial 571  
Repository Modules 312  
Resizable Emulator 50  
Resource  
    string creation 23  
Resource File 25  
Resource Management 147  
Resource Manager , 57  
result receiver 485  
Reverse-geocoding 653  
Reverse Geocoding 652  
Room  
    Data Access Object (DAO) 554  
    entities 554, 555  
    In-Memory Database 560  
    Repository 554  
Room Database 554  
    tutorial 571  
Room Database Persistence 553  
Room Persistence Library 550, 553  
root element 173  
root view 175  
Run  
    tool window 57  
runBlocking 493  
Running Devices  
    tool window 69  
runOnUiThread() 743

## S

- safeargs
- Safe Call Operator 100
- Sample Data 190
- Saved State 311, 341
  - library dependencies 343
- SavedStateHandle 342
  - contains() method 343
  - keys() method 343
  - remove() method 343
- Saved State module 341
- SavedStateViewModelFactory 342
- ScaleGestureDetector class 287
- Scale-independent 241
- SDK Packages 6
- Secondary Constructors 128
- Secure Sockets Layer (SSL) 85
- SeekBar 299
- sendBroadcast() method 483, 485
- sendOrderedBroadcast() method 483, 485
- SEND\_SMS permission 606
- sendStickyBroadcast() method 483
- Sensor permissions 606
- Service
  - anatomy 506
  - launch at system start 507
  - manifest file entry 506
  - overview 88
  - run in separate process 507
- ServiceConnection class 519
- Service Process 148
- Service Restart Options 506
- setAudioEncoder() method 614
- setAudioSource() method 614
- setBackgroundColor() 246
- setCompassEnabled() method 658
- setContentView() method 245, 251
- setId() method 246
- setMyLocationButtonEnabled() method 658
- setOnClickListener() method 263, 265
- setOnDoubleTapListener() method 275, 278
- setOutputFile() method 614
- setOutputFormat() method 614
- setResult() method 455
- setText() method 172
- settings.gradle.kts file 762
- setTransition() 385
- setVideoSource() method 614
- SHA-256 certificate fingerprint 457
- SharedFlow 534, 537
  - backgroundn handling 542
  - DROP\_LATEST 534
  - DROP\_OLDEST 534
  - in ViewModel 539
  - repeatOnLifecycle 544
  - SUSPEND 535
  - tutorial 537
- shouldOverrideUrlLoading() method 672
- SimpleOnScaleGestureListener 287
- SimpleOnScaleGestureListener class 288
- single() operator 526
- SMS permissions 606
- Snackbar 413, 414, 415
- Snapshots
  - emulator 47
- sp 241
- Spread chain 198
- Spread inside 228
- Spread inside chain 198
- SQL 548
- SQLite 547
  - AVD command-line use 549
  - Columns and Data Types 547
  - overview 548
  - Primary keys 548
- StaggeredGridLayoutManager 431
- startActivity() method 453
- startForeground() method 148
- START\_NOT\_STICKY 506
- START\_REDELIVER\_INTENT 506
- START\_STICKY 506
- State
  - restoring 172
- State Change
  - handling 151

## Index

StateFlow 533  
Statement Completion 76  
Status Bar Widgets 73  
    Memory Indicator 73  
Sticky Broadcast Intents 485  
Stopped state 150  
Storage permissions 607  
String 96  
strings.xml file 27  
Structure  
    tool window 58  
Structured Query Language 548  
Structure tool window 58  
SUBS 736  
subscriptions 731  
supervisorScope 493  
SupportMapFragment class 647  
SUSPEND 535  
Suspend Functions 492  
Switcher 60  
System Broadcasts 489  
system requirements 3

## T

TabLayout  
    adding to layout 423  
    app  
        tabGravity property 428  
        tabMode property 428  
    example 420  
    fixed mode 427  
    getItemCount() method 419  
    overview 419  
TableLayout 174, 563  
TableRow 563  
Telephony Manager 86  
Templates  
    blank vs. empty 177  
Terminal  
    tool window 58  
terminal flow operators 529  
Theme

    building a custom 751  
Theming 749  
    tutorial 753  
Time Cycle Keyframes 385  
TODO  
    tool window 59  
ToolBarListener 296  
tools  
    layout 293  
Tool window bars 56  
Tool windows 56  
Touch Actions 270  
Touch Event Listener  
    implementation 271  
Touch Events  
    intercepting 269  
Touch handling 269  
transform() operator 529  
try/finally 526  
Type Annotations 99  
Type Casting 103  
Type Checking 103  
Type Inference 99

## U

UiSettings class 647  
unbindService() method 505  
unregisterReceiver() method 485  
upload key 720  
URL Mapping 701  
USB connection issues  
    resolving 66  
USE\_BIOMETRIC 710  
user interface state 155  
USE\_SIP permission 606

## V

Video Playback 587  
VideoView class 587  
    methods 587  
    supported formats 587  
view bindings



- enabling 144
- using 144
- View class
  - setting properties 252
- view conversion 189
- ViewGroup 173
- View Groups 173
- View Hierarchy 175
- ViewHolder class 432
  - sample implementation 438
- ViewModel
  - adding LiveData 321
  - data access 319
  - overview 310
  - saved state 341
  - Saved State 311, 341
  - tutorial 315
- ViewModelProvider 318
- ViewModel Saved State 341
- ViewModelScope 492
- ViewPager
  - adding to layout 423
  - example 420
- Views 173
  - Java creation 245
- View System 86
- Virtual Device Configuration dialog 32
- Virtual Sensors 47
- Visible Process 148

## W

- WebViewClient 667, 672
- WebView view 473
- Weighted chain 198, 228
- Welcome screen 53
- while Loop 114
- Widget Dimensions 199
- Widget Group Alignment 221
- Widgets palette 234
- WiFi debugging 67
- Wireless debugging 67
- Wireless pairing 67

- withContext 493, 495
- wrap\_content properties 243
- WRITE\_CALENDAR permission 606
- WRITE\_CALL\_LOG permission 606
- WRITE\_CONTACTS permission 606
- WRITE\_EXTERNAL\_STORAGE permission 607

## X

- XML Layout File
  - manual creation 241
  - vs. Java Code 245

## Z

- zip() operator 531

